GraVF: A Vertex-Centric Distributed Graph Processing Framework on FPGAs

Nina Engelhardt, Hayden Kwok-Hay So
Department of Electrical and Electronic Engineering, The University of Hong Kong
Email: {nengel, hso}@eee.hku.hk

Abstract—**FPGAs are promising platforms to efficiently execute distributed graph algorithms. Unfortunately, they are notoriously hard to program, especially when the problem size and system complexity increases. In this paper, we propose GraVF, a high-level design framework for distributed graph processing on FPGAs. It leverages the vertex-centric paradigm, which is naturally distributed and requires the user to define only very small kernels and their associated message semantics for the target application. The user design may subsequently be elaborated and compiled to the target system automatically by the framework. To demonstrate the flexibility and capabilities of the proposed framework, 4 graph algorithms with distinct requirements have been implemented, namely breadth-first search, PageRank, single source shortest path, and connected component. Results show that the proposed framework is capable of producing FPGA designs with performance comparable to similar custom designs while requiring only minimal input from the user.**

I. Introduction

Graphs are important data structures that are at the heart of many important applications including machine learning, bioinformatics and data mining. Unfortunately, while there are plenty of graph algorithm development frameworks available for software systems of all scales, there is currently a lack of general programming frameworks for distributed graph processing on FPGAs. Existing works, while having successfully demonstrated the performance benefit of FPGA-based graph processing, were highly customized to the particular graph application and must rely on expert hardware designers with intimate understanding of the underlying hardware to achieve the demonstrated performance.

To address these productivity challenges, we propose GraVF (Graph processing with Vertex-centric algorithms on FPGAs), a vertex-centric, distributed framework for graph computation on FPGAs in this paper. We argue that the vertex-centric graph processing model, while being relatively simple, is well suited for FPGA implementations and is powerful enough to encapsulate many important graph algorithms to execute even on distributed FPGA clusters with relative ease. GraVF is implemented in Migen, a Python-based metaprogramming language for generating hardware [1]. With Migen, users provide to the framework small kernels specific to their applications, as well as data type definitions for messages that should be passed between processing elements. Utilizing the Migen cycle-accurate simulator, users are able to simulate their designs before they are implemented on the target FPGA.

To demonstrate the flexibility of GraVF, 4 graph algorithms with distinct requirements were implemented: breadth-first search, PageRank, single-source shortest path, and connected components. In our current implementation, we can achieve edge traversal speeds of 3-3.5 billion edges per second on a single FPGA.

As such, we consider the main contributions of this work are the following:

- We demonstrate the feasibility and flexibility of implementing distributed graph algorithms on FPGAs with a vertex-centric programming model;
- As a productivity enhancement tool, we have implemented a high-level design framework that automatically generates distributed graph processing hardware for FPGAs with minimal user input;
- We introduce a split apply-scatter kernel for each graph processing PE that improves pipeline performance while reducing system-wide minimum deadlock-free message buffer requirement.

The rest of the paper is organized as follows: Section II describes the programming model for the GraVF framework. Section III describes the architecture of our prototype implementation. Section IV presents performance data collected on our prototype to evaluate the feasibility of such a framework. Section V situates our work in the context of previous FPGA-based graph computation efforts. Finally, we conclude in section VI.

II. Programming Model

In this section, we describe the programming model for our GraVF framework. Frameworks such as Pregel [2] or GraphLab [3] have shown that vertex-centric models are especially well suited for distributed systems. We therefore adopt a synchronous vertex-centric model, and introduce some adaptations specific to our FPGA system to improve performance.

A. Vertex-Centric Graph Processing

In a vertex-centric programming model, algorithms are described in the form of a computational kernel, to be run simultaneously and independently by each vertex. Vertices may only access their own data and that of their incoming edges, and may only communicate with vertices along outgoing edges. The small scope and lack of centralized control allow easy partitioning of both data and computation.

The vertex kernels are executed iteratively in synchronous processing steps called "supersteps", with barrier synchronization across the whole system between supersteps. In each superstep, messages sent during the previous superstep are received and handled, and messages for the next superstep are generated. If no messages were generated in the previous superstep, computation terminates.

Extending this standard Pregel-like model, we ask the user to split their vertex kernel implementation into two parts, implemented as two separate hardware modules:

Apply takes as input the current state of the vertex-associated data and an incoming message. It should return the (potentially modified) vertex-associated data, and optionally an update to be passed on to the scatter kernel.

Scatter takes an update produced by *Apply*, an outgoing edge, and the total number of neighbors, and produces the final message contents to be sent to each neighbor.

We illustrate the specification of algorithms in GraVF with the Single-Source Shortest Path example. [1] Given a start vertex s, this algorithm computes the path with minimal distance to each vertex in the graph starting from s. Initially, all vertices set their distance to infinity. The source sets its distance to zero, and propagates this to its neighbors. Iteratively, each node that receives a message will compare its current distance to the sender's distance plus the weight of the edge linking them. If this is a shorter path, it updates its current distance and sends a message to its neighbors to notify them of the new value.

```
self.comb += [
    If(self.state_in.dist > self.message_in.dist,
        self.state_out.dist.eq(self.message_in.dist),
        self.update_valid.eq(self.valid_in)
    ).Else(
        self.state_out.dist.eq(self.state_in.dist),
        self.update_valid.eq(0)
    ),
    self.update_out.dist.eq(self.message_in.dist),
    self.state_valid.eq(self.valid_in),
    self.nodeid_out.eq(self.nodeid_in),
    self.update_sender.eq(self.nodeid_in),
    self.barrier_out.eq(self.barrier_in),
    self.ready.eq(self.update_ack)
]
```

Listing 1. SSSP Apply Kernel

Listing 1 shows the complete apply kernel code for SSSP in Migen. The computation is simple enough to be implementable combinatorially. If the received distance `self.message_in.dist` is smaller than the previously stored distance `self.state_in.dist`, the state is updated and a new update is sent (`self.update_valid` is set).

```
self.sync += If(self.message_ack,
    self.message_out.dist.eq(self.update_in.dist
        + self.edgedata_in.dist),
    self.neighbor_out.eq(self.neighbor_in),
    self.sender_out.eq(self.sender_in),
    self.round_out.eq(self.round_in),
    self.valid_out.eq(self.valid_in),
    self.barrier_out.eq(self.barrier_in)
)
self.comb += self.ready.eq(self.message_ack)
```

Listing 2. SSSP Scatter Kernel

[1]The code of our other three sample algorithms (PageRank, BFS, and Connected Components) can be found online at https://github.com/nakengelhardt/fpgagraphlib

The scatter kernel implementation for SSSP is shown in listing 2. For each outgoing edge, it adds the edge's weight `self.edgedata_in.dist` to the new distance broadcast by the vertex `self.update_in.dist`. In this case, for the purpose of illustration, a pipelined implementation with an intermediary register was realized through the use of the Migen `sync` keyword. The guarding `If` implements pipeline enable by the flow control signal `self.message_ack`.

The user also defines four data widths: the vertex data, the edge data, the (intra-PE) update payload, and the (inter-PE) message payload. Fixed by the framework, the update header also includes the sender's vertex ID, and the message header includes both sender and receiver ID. Listing 3 shows how these data types are defined in Migen.

```
message_layout = [("dist", 32, DIR_M_TO_S)]
update_layout = [("dist", 32, DIR_M_TO_S)]
node_storage_layout = [("dist", 32),
                       ("parent", "nodeidsize")]
edge_storage_layout = [("dist", 32)]
```

Listing 3. SSSP Data Type Definitions

The split vertex kernel model allows us to address two commonly cited shortcomings of the Pregel model: The first is unbalanced computation, where one processing element has to deal with many more vertex kernel executions than others, leaving vast swathes of the system idle, waiting at the barrier for this straggler. The second is that all messages sent in one superstep need to be stored until the start of the next superstep. Since there can be as many messages sent in one superstep as there are edges in the graph, and the number of edges is often one or two orders of magnitude greater than the number of vertices, the storage requirements can be prohibitive. The combination of a "floating barrier", which moves through each processing element's pipeline independently, with a relocation of the main storage queue in-between the apply and scatter components in the same pipeline enables us to counteract both issues while leaving the synchronous programming model intact. The details of this approach can be found in the architecture description in section III-2.

III. Architecture

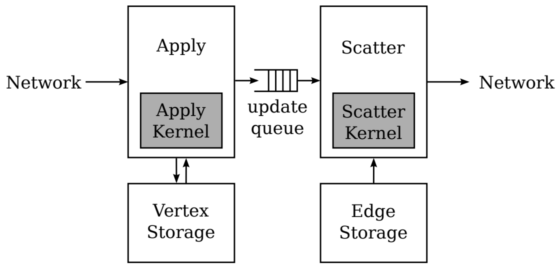


Fig. 1. Processing Element Architecture

To execute the user application in hardware, GraVF incorporates the user-supplied graph processing kernels and message type definitions to form the underlying execution architecture. This architecture consists of a fixed network of identical processing elements (PEs), with only the user-provided apply and scatter kernels being switched out for each algorithm, and storage resized according to the user-defined data types.

Each vertex in the graph is assigned to a processing element, which will store the vertex's data and run the vertex kernel whenever a message for this vertex is received. The network is responsible for transporting messages to the processing element containing the destination vertex and storing it until the next superstep. It also carries out the barrier synchronization between supersteps.

At each endpoint in the network, the PE structure in Fig. 1 is reproduced. The processing element is composed of the *Apply* and *Scatter* components. The user-provided apply and scatter kernels (shown shaded) are connected within these components. All steps are pipelined and impose no restriction on the latency or throughput of the user kernel implementations.

1) Message Passing and Barrier Synchronization: For synchronization of supersteps among the PEs, we implement a distributed barrier algorithm similar to the "floating barrier" described in [4]. This algorithm relies upon the observation that in a fully pipelined system, it is not necessary for correctness that supersteps are actually physically separated in time; it is sufficient that each processing stage only starts the next superstep once it has received and processed all messages sent in the previous superstep from all processing elements. Each processing element therefore sends a barrier marker to all PEs when it finishes a superstep.

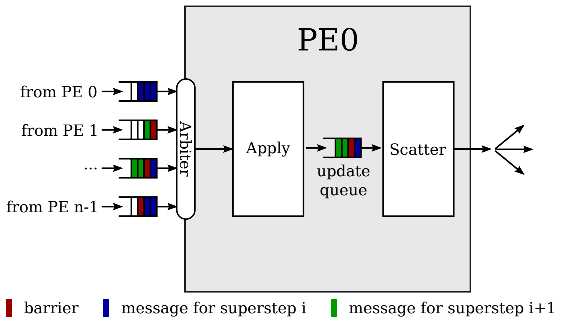


Fig. 2. Floating Barrier

The arbiter at the receiving end needs to let pass only messages destined for the current superstep, and block messages for the next superstep until it receives barrier markers from all PEs. In our current implementation messages are sorted into separate queues for each sending PE (cf. Fig. 2). The arbiter selects messages in a round-robin fashion from the queues with available messages to forward to the apply component. Once a barrier marker reaches the front of the queue (as in the case of the queue for messages from PE 1), no further messages are withdrawn from this queue until all queues show barrier markers. The arbiter will then withdraw all barrier markers simultaneously and signal the barrier to the apply component.

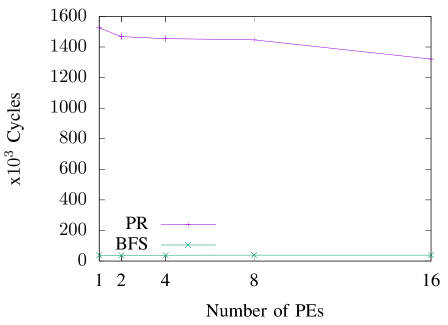
Finally, the floating barrier is also used to detect termination: each PE tracks the last two messages it received. If both are barrier messages, no vertices were active on this PE in the last superstep. While this condition holds, the PE signals inactivity. If all PEs signal inactivity simultaneously, no messages at all were sent in the previous superstep, and the system is halted.

2) Deadlock Free Queue Sizing: The sizing of the queues is important as deadlock may result when queue capacity is reached. Each vertex can send one message per superstep to each of its neighbors, for a total of up to $|V|^2$ messages (in the worst case of a fully connected graph). This represents a prohibitive amount of storage necessary to avoid deadlock in a truly synchronous system where supersteps are physically separated in time.

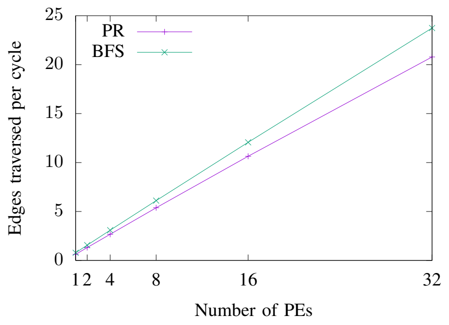
However, in our architecture it is possible to leverage the pipelined nature of the apply and scatter components to reduce the storage requirements. Within a pipeline, storage can be moved up or down at will, so we move the principal storage capacity of the PE to a FIFO queue between the Apply and Scatter components. The apply component can emit no more than one update per vertex, for a system-wide total of $|V|$ updates per superstep. With the floating barrier, there can be overlap of up to two supersteps, so we choose an update queue size of $2\,|V|/n$ per PE to guarantee that no deadlock can occur as the update queue is never full. Since only the scatter component transforms each update into multiple messages for the vertex's neighbors, this novel way to shift the burden of buffering drastically reduces the overall storage requirements from $|V|^2$ to $2\,|V|$.

IV. Evaluation

CC and SSSP are very similar to BFS in their execution characteristics, so for brevity we evaluate the prototype's performance using two example algorithms, PageRank and BFS.



(a) Weak scaling (8192 vertices per PE)



(b) Strong scaling (131071 vertices total)

Fig. 3. Strong and weak scaling results for PageRank and BFS

PageRank and BFS represent two extremes in terms of algorithm characteristics: the PageRank vertex kernel contains several floating-point operations with a combined latency of around a hundred cycles, and sends the maximum possible number of messages, one for each edge in the graph, each superstep. In contrast, the BFS vertex kernel is combinatorial (0 cycles latency), and each vertex will only send a message to its neighbors once over the total run of the algorithm.

To explore the scaling behavior of our prototype, we synthesized and ran systems with 1 to 32 processing elements on the Xilinx Virtex 7 (xc7vx690tffg1157-2) FPGA on the AlphaData ADM-PCIE-7V3 platforms. We examine both strong scaling (dividing a given graph size among a growing number of PEs) and weak scaling (with a fixed number of vertices per PE, graph size grows along with the system). Fig 3 shows that we obtain linear results for strong scaling on both algorithms, and even slightly improving runtime on weak scaling experiments as the increased storage from the n^2 arbiter queues reduces the impact of stragglers. At 32 PEs, our system processes 21-24 edges per cycle, corresponding to edge traversal speeds of 3-3.5 billion edges per second.

We also use the synthesis to report resource usage. For a system size of 16 processing elements, we can handle a graph with 128k vertices and 512k edges and obtain the post-implementation utilization detailed in table I.

TABLE I
RESOURCE USAGE

PEs	16	
Vertices	131072	
Edges	524288	
Clock	150 MHz	
	PageRank	BFS
BRAM	1096.5/1470 (74.59%)	634.5/1470 (43.16%)
LUT	113776/433200 (26.26%)	39986/433200 (9.23%)
FF	80074/866400 (9.24%)	21502/866400 (2.48%)
DSPs	32/3600 (0.89%)	0

V. Related Work

Graph algorithms are suitable candidates for acceleration on FPGAs, as several case studies implementing individual algorithms have shown [4], [5]. Most prominently, in 2011 Convey Corporation implemented the Graph500 supercomputing benchmark for their hybrid CPU-FPGA systems. They entered the ranking in 25th place, placing them on a competitive level with large clusters despite the fact that their system only comprised a single machine [6]. Nonetheless, research on graph processing frameworks for FPGA is only getting started. Betkaoui et al. [7] suggested an architecture in 2011, but only studied its possible mechanics in a case study, wherein they followed the proposed structure by hand. An actual implementation of the framework seems not to have been pursued. The most consequent work to date in the direction of graph processing frameworks for FPGA is GraphGen for CoRAM [8]. However, GraphGen only implements a single graph processing element on the FPGA, thereby eliminating the advantage conferred by the FPGA's massively parallel fabric. It is also unclear how it would scale to graphs larger than a single machine can handle. More recently, Kapre proposed a softcore-based graph processing architecture for FPGA called GraphSoC [9]. It uses a similar vertex-centric approach, but with a more restrictive model that asks the user to divide their kernels into four separate phases (receive, accumulate, update, send). GraphSoC uses HLS to produce a custom processor datapath based on the user algorithm specification. Our system would equally allow the user to use HLS to produce the apply and scatter kernels, or any other language or method that produces an instanciable netlist.

VI. Conclusion & Future Work

In this paper, we presented GraVF, a framework for graph computation on FPGA. Our synchronous vertex-centric programming model reduces implementation effort, requiring the user to implement only two small kernels in the language of their choice. Our fully pipelined, distributed message-passing architecture leverages the concept of a floating barrier to massively reduce storage requirements, while maintaining the synchronous abstraction. We evaluate a prototype implementation and show linear scaling up to 32 processing elements, and a sustained performance of 3-3.5 billion traversed edges per second. In the future, we plan to grow the scale of graphs that GraVF can handle. This includes adding support for storing the edge data in off-chip memory to the GraVF architecture, as well as an improved network allowing communication between multiple FPGA.

References

[1] M-Labs, "Migen," http://m-labs.hk/gateware.html.

[2] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. ACM, 2010, pp. 135–146.

[3] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "GraphLab: A new framework for parallel machine learning," in *Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence*, ser. UAI '10. AUAI Press, 2010.

[4] Q. Wang, W. Jiang, Y. Xia, and V. Prasanna, "A message-passing multi-softcore architecture on FPGA for breadth-first search," in *Field-Programmable Technology (FPT), 2010 International Conference on*, Dec 2010, pp. 70–77.

[5] O. Attia, T. Johnson, K. Townsend, P. Jones, and J. Zambreno, "Cygraph: A reconfigurable architecture for parallel breadth-first search," in *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, May 2014, pp. 228–235.

[6] C. Computer, "Convey computer doubles Graph500 performance, develops new graph personality," http://www.conveycomputer.com/files/2413/5095/9078/SC11_Graph500_Release.Final.pdf, November 2011, press release.

[7] B. Betkaoui, D. B. Thomas, W. Luk, and N. Przulj, "A framework for FPGA acceleration of large graph problems: graphlet counting case study," in *2011 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2011, pp. 1–8.

[8] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin, "GraphGen: An FPGA framework for vertex-centric graph computation," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2014, pp. 25–28.

[9] N. Kapre, "Custom FPGA-based soft-processors for sparse graph acceleration," in *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*, July 2015, pp. 9–16.