Towards Flexible Automatic Generation of Graph Processing Gateware

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

ABSTRACT

FPGAs have been demonstrated as promising platforms to accelerate graph processing applications at scale with superior energy-efficiency. However, programming FPGAs is significantly more challenging than similar software solutions. To address this productivity challenge, several graph processing frameworks for FPGA have already been proposed in recent years. These frameworks aim to lower a programmers burden by requiring users to provide only logic specific to the target graph algorithm, while leaving the auto generation of the rest of the hardware design to the framework. In this work, we extend the capability of the GraVF framework and improve the scale of its supported input graphs by a) making the synchronization method independent of the network structure and b) adding support for off-chip memory. The improved system accepts graph sizes an order of magnitude larger than previously reported and provides throughput in the order of 100MTEPS per processing element.

1. INTRODUCTION

Graph processing lies at the heart of many modern data analysis techniques, and with the growth of big data, the size of datasets is ever increasing. Graph structure presents some unique challenges to parallelization, due to its inherent characteristics:

- **Low computational density.** Most graph algorithms perform very little calculation per vertex, making memory access speed the single most important determining factor in graph processing performance.
- **Random, data-driven memory access.** Complicating the above, graph data access patterns are unpredictable and dominated by pointer chasing.
- **Difficult to partition.** Graphs are highly interconnected, meaning that even with the most careful partitioning optimizing for local connections, heavy communication between partitions persists.

This combination of features makes most of the optimizations in modern CPU and GPU platforms moot if not actively counterproductive, as they mainly rely on temporal and spatial locality in data accesses. Caches in particular will often bring in nearby data, which is likely to go unused when processing graphs, thus leading to wasted bandwidth.

FPGA platforms however offer features more suitable to graph algorithms: their internal BRAM memory offers high-throughput, low-latency random access, and their fabric permits implementation of a larger number of low-complexity processing elements capable of shouldering the light computational burden of graph algorithms even at lower clock speeds.

A range of FPGA-based solutions have been proposed, both for individual algorithms [1, 2, 3, 4, 5] and graph analysis frameworks [6, 7, 8, 9, 10]. Our GraVF[9] framework explicitly addresses the programmability challenge, and achieves the highest throughput. However, like most of the proposed frameworks, its graph size is limited by the available BRAM used to store the graph.

In this paper, we present improvements to several aspects of the GraVF framework: We increase capacity by adding support for off-chip memory, we make the network more flexible and scalable by separating the transport mechanism from the synchronization protocol, and we balance computation across PEs using a better partitioning algorithm.

2. BACKGROUND: GRAVF FRAMEWORK

Before detailing our modifications, we present a brief overview of the architecture of the framework. GraVF is a synchronous, vertex-centric graph processing framework. The core algorithm definition in this type of framework is in the form of a *vertex kernel*. Computation is organized in the form of iterative *supersteps*. In each superstep, the vertex kernel is executed concurrently for all active vertices. A superstep is terminated by a global barrier synchronization. Execution completes when no active vertices remain. The following restrictions apply on vertex kernels:

- it may only access its own private vertex data
- it may send messages along its outgoing edges
- messages sent in superstep n will be received in superstep $n + 1$
- a vertex is active in a superstep only if it receives at least one message.

In GraVF, a vertex kernel consists of two functions, *apply* and *scatter*. The apply kernel is executed once for each message received. It can both read and write the vertex data. If neighboring vertices need to be informed of changes, it may send an update to the scatter kernel. When it receives

This work was presented in part at the international symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2017) Bochum, DE, June 7-9, 2017.

an update, the scatter kernel will be iterated over the adjacency list of the vertex, and it may send a message along each outgoing edge.

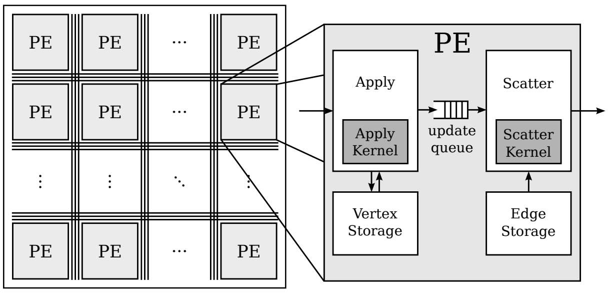

Figure 1: GraVF Architecture overview

The GraVF architecture follows this general model. It consists of several processing elements (PEs) that exchange messages through an on-chip interconnect. The vertices of the graph are partitioned among the processing elements, with each PE responsible for executing the vertex kernel on its subset of vertices. As shown in Fig. 1, each PE has separate apply and scatter modules, with the apply module having exclusive access to a private memory storing the vertex data of its subset of vertices, and the scatter module having read access to a memory containing the adjacency lists of the PE's assigned vertices.

2.1 Apply Module

The vertex kernels are split into two phases, which are executed by separate modules within the PE. The Apply module, shown in Fig. 2, receives incoming messages from other PEs and updates the vertex state based on the received data. It therefore needs read-write access to the vertex data.

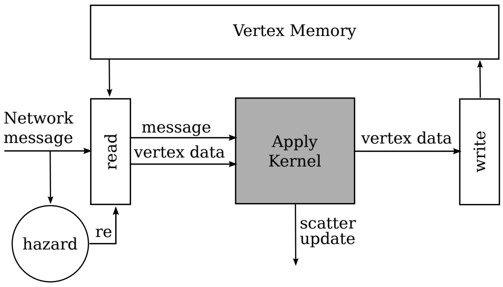

Figure 2: Apply Module

When an incoming message from the previous superstep is received by a PE, the destination vertex ID is used to look up the associated state in the vertex memory. Both message and vertex state are then passed to the Apply kernel, which later produces the updated state data to be written back. If the algorithm requires neighbors to be notified of state updates, the Apply kernel also produces update data. This data is passed on to the scatter module for communication with neighbor vertices. As the module is fully pipelined, a read after write hazard could arise if messages for the same vertex are received in quick succession. A hazard detection module keeps track of which vertices are currently in use by the apply kernel and stalls messages in case of hazard until the modified data is written back.

2.2 Scatter Module

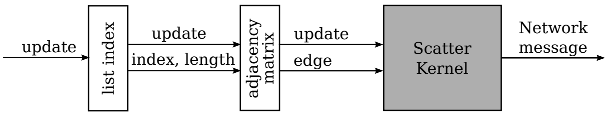

Figure 3: Scatter Module

Updates from the Apply module are passed to the scatter module, which generates the messages for the next superstep. The updates from the Apply module include the sender vertex ID, which is used to look up the vertex's adjacency list. Adjacency lists are stored in compressed sparse row (CSR) format, which consists of two arrays, each implemented as a separate BRAM memory. The first array stores for each vertex the index at which the adjacency list may be found in the second array, as well as the length of the adjacency list. The index and length information are then used to iterate over the edge data from the second array. The edges are passed one by one to the scatter kernel, along with the update data from the apply module. The scatter kernel can then produce one message per edge, to be sent over the interconnect and ultimately received by the apply module hosting the destination vertex when treating the next superstep.

2.3 Synchronization

Barrier synchronization between supersteps is achieved via barrier messages exchanged over the interconnect. Each PE will send a message to all PEs (including itself) when finishing a superstep. Therefore, when a PE has received a barrier message from all PEs, this indicates that there are no more messages to be treated in this superstep and it may proceed to the next. The barrier message separates messages from superstep k and $k+1$. In GraVF, the barrier protocol is highly dependent on the network architecture: n PEs are connected using $n \times n$ dedicated links so that messages sent from a PE A to a PE B will always arrive in-order and with no interference from messages concerning other PEs. The network endpoint of a PE can then accept messages from a given link until it receives a barrier message, at which point it will no longer take messages from this link until it has received a barrier message on all n links.

This organization is easy to implement without risk of deadlock, but it comes with several drawbacks: The use of $n \times n$ links requires significant resources, especially as the number of PEs increases. It also leads to routing issues in larger FPGAs, decreasing the maximum clock speed of the design.

3. INCREASING THE FLEXIBILITY OF THE NETWORK PROTOCOL

This section presents how we modified the synchronization protocol to enable the use of various network topologies. The previous protocol required guaranteed in-order delivery of messages, with each PE being able to exert backpressure separately on messages originating from different PEs. Our modifications reduce the network requirements to only guaranteed delivery (no lost messages) with separate backpressure per superstep.

3.1 Out-of-order delivery

To support out-of-order delivery of messages, we make the network endpoint count all sent and received messages in a superstep, for each PE separately. When a PE A emits a barrier, the endpoint inserts into the barrier message sent to each PE B the number of messages sent from A to B in this superstep. On the receiving side, the network endpoint at B will compare the number of messages received to the number of messages expected from A, and if any messages are still missing, wait until they arrive before passing the barrier to PE B.

We repurpose the `dest_id` field, which in normal messages holds the destination vertex's ID, but is unused in barrier messages which only have a destination PE, to hold the message count.

3.2 Differentiating between Supersteps

When messages can arrive out-of-order, some mechanism is needed to identify which superstep a message belongs to. As there can be only two supersteps overlapping computation at any point in time, it would be tempting to assume that a parity bit would be sufficient to separate successive supersteps. However, while there can be only two concurrent supersteps inside the PEs, in a context where messages can be reordered, there can be messages from up to three supersteps in existence in the network simultaneously.

The following scenario, involving a system with 2 PEs A and B, illustrates how messages from three separate supersteps can come to coexist in the network:

1. A and B run superstep $n - 1$, which sends messages for superstep n
2. Both A and B finish superstep $n-1$ and send barriers indicating last message for superstep n
3. Messages for A are held up in the network, messages for B get delivered and cause B to run superstep n, which sends messages for $n + 1$
4. B receives barriers indicating last message for n from both A and B, and terminates superstep n
5. B starts processing messages for the superstep $n+1$ as it knows (due to the barrier for the superstep n) that there are no more messages for B in superstep n
6. B, during superstep $n+1$, sends messages for superstep $n + 2$ to A

If the messages for superstep n with destination A sent in step 1 have not arrived yet, and messages can be reordered by the network, a simple parity bit is not enough to distinguish them from the messages for superstep $n + 2$ with destination A sent in step 6.

It is not possible for messages from any earlier or later superstep to coexist. If a message for superstep $n - 1$ were to still exist, no PE could end superstep n (step 4), since at least one PE would still be waiting for the message in question and could not send its barrier yet. Similarly, no PE could start sending messages for superstep $n + 3$, since no PE could finish superstep $n + 1$ before the last messages for superstep n have arrived.

We therefore add a configuration parameter "number of channels", and a (usually 2-bit) field to the message format containing the superstep number modulo the number of channels. The choice of the number of channels is at the network designer's discretion. For a network that aims for high performance by maintaining separate physical channels for each superstep, using the minimum 3 necessary for correctness would reduce the area investment. In contrast, the choice of 4 simplifies the handling of the 2-bit field and would be most suited for a network that aims to reduce area use by minimizing buffers, and that stalls messages of all but the oldest superstep at the sender.

3.3 Detecting Termination

Computation terminates when no message is sent in a superstep. In GraVF, each PE detects its own inactivity if it receives two barriers in a row with no messages inbetween, and raises a signal. If all PEs raise this signal simultaneously, the system is stopped. This solution is contrary to the distributed nature of the system, so we replace it with the following:

The outgoing network endpoint of a PE already keeps track of the number of messages sent after our modifications in part 3.1. We add a 1-bit field to the barrier message format noting inactivity of a PE. When the PE sends a barrier but did not send any messages in this superstep, it sets the bit field `inactive` to 1.

The incoming network endpoint receives a barrier message from each PE for each superstep. It detects termination if all PEs have set the `inactive` bit in the same superstep. Any single PE may then be chosen to be responsible for communicating termination to the outside world.

In combination, the modifications in this section allow substitution of any network that can transfer packets of the appropriate size, with the only restriction that packets in different channels may not block each other.

4. USING OFF-CHIP MEMORY TO IMPROVE GRAPH SIZE

In this section, we describe how the GraVF framework was modified to make use of HMC memory. We first describe the memory interface offered by our evaluation platform, then how it was integrated into the GraVF architecture.

4.1 Memory interface

Hybrid Memory Cube technology offers a significant increase in bandwidth compared to traditional DDR memory. In a HMC device, several layers of DRAM banks are stacked in a 3D architecture connected by through-silicon vias to a CMOS base layer implementing multiple memory controllers running in parallel. The memory controllers communicate with the host over full-duplex 30Gb/s serial links via a packetized protocol. The HMC will reorder requests at will to improve performance. To associate requests with their respective responses, each request packet contains a 9-bit tag uniquely identifying the transaction, chosen by the host among any tag not currently in use by an outstanding request. The response will be identified by the same tag.

Generating the amount of requests necessary to utilize this bandwith from the much slower FPGA fabric requires a number of parallel accesses. The Micron HMC controller IP therefore multiplexes each link among 5 memory ports with a data width of 128 bit running at 187.5 MHz. Each memory port presents three independent channels: command channel (Table 1), write data channel (Table 2) and response channel (Table 3).

Table 1: Command Channel Layout

cmd	4 bit	request type (read, write, ...)
addr	33 bit	address
size	4 bit	request size
tag	6 bit	request identifier
cmd_ready	1 bit	flow control
cmd_valid	1 bit	flow control

Table 2: Write Data Channel Layout

wr_data	128 bit	data (write request)
wr_data_ready	1 bit	flow control
wr_data_valid	1 bit	flow control

The salient points are as follows: The smallest adressable unit for regular read and write operations is 16 bytes. All addresses should be 16 byte aligned. Requests and replies are decoupled and can occur in random order. Each port has a subspace of 64 tags for its own use, with the controller reserving 3 bits to associate ports with requests and replies. Flow control occurs for requests (the controller can refuse requests by setting `cmd_ready` to low), but not for replies, so the receiver needs to ensure that it will always have buffers available to store replies for all in-flight requests.

4.2 Implementation

There are two main storage modules in the design: the vertex memory and the edge memory. We elect to move the much larger edge data to HMC memory, leaving the smaller vertex data in BRAM. The vertex data is frequently read and written, and controlling for hazards is already an issue of some significance even with single-cycle memory accesses. The long latency of HMC operations would significantly increase the rate of hazard-induced stalls. The edge data, in contrast, is read-only during the computation and is accessed at most once per superstep.

We relocate the adjacency list storage into off-chip memory and replace the portion of the scatter module that, given an update and the location of the adjacency list, iterates over the edges and presents them one by one to the scatter kernel together with the update data. Both sides of this module have flow control using ready/valid signals: It may tell the upstream to hold off on the next update by deasserting ready, and it must be able to wait in case the downstream scatter kernel is not ready to accept the next edge.

We solve this flow control issue by having three stages in our module running independently. We use the same 6 bit tags used to identify HMC commands to track the progress of updates across the stages. Two buffers serve to save data while it is being processed. They are addressed using the tags, and therefore have 64 entries each.

Fig. 4 shows the flow of data across the module. The first stage accepts the input requests. It retrieves a currently unused tag from the free tag queue, sends a read command of the adjacency list to the HMC port's command channel, and saves the update data to the update buffer for later retrieval. If the adjacency list is longer than the read data width of the HMC port, multiple requests will be sent. For the purposes of the rest of the module these are indistinguishable from requests generated by separate updates, as they use their own tag. If the HMC port is not ready or no free tags are available, the upstream is stalled.

Table 3: Response Channel Layout

rd_data	128 bit	data read
rd_data_tag	6 bit	request identifier
rd_data_valid	1 bit	data valid
errstat	7 bit	error code
dinv	1 bit	data invalid

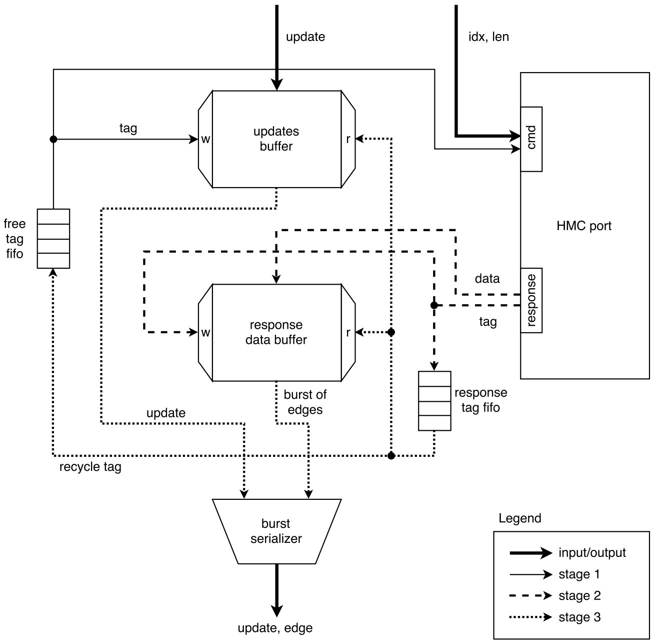

Figure 4: HMC Lookup of Adjacency List

The second stage accepts responses from the HMC port's response channel. Since responses cannot be stalled and must be accepted when they arrive, this stage stores the response in the response data buffer until such time as the third stage can process it. The tag is passed into the response tag queue.

The third stage takes a tag from the response tag queue and uses it to look up both the original update data and the response data. The response data, at 16 bytes wide, is a burst containing several edges. A serializer iterates over the edges and passes them one by one to the scatter kernel along with the update data. Finally, the tag is returned to the free tag queue to be reused.

Because the HMC controller can return read requests in random order, some mechanism is needed to maintain the internal ordering of the updates when a barrier is received. Messages belonging to the same superstep do not need to follow a specific order with respect to each other, so during normal processing edges can simply be passed on to the scatter kernel in any order. In the case of a barrier, however, to ensure that the system will remain deadlock-free, all updates from the superstep before the barrier have to be emitted before the barrier, and all updates from the superstep after the barrier have to stay after the barrier. When a barrier message is received, we therefore flush the module by waiting until all tags are returned to the free tag queue before passing on the barrier directly to the scatter module (bypassing the three stages now guaranteed to be inactive) and accepting the first update of the next superstep.

5. PARTITIONING THE INPUT GRAPH

How the vertices of the input graph are partitioned can have a significant effect on the performance of the system. Traditionally, graph partitioning algorithms have to make a tradeoff between two concerns:

- **Balanced computation** Since the computation is barrier synchronized, the PEs should have equal workloads to avoid wait times.
- **Prioritize local communication** In most architectures, some PEs are "closer" than others, with latency increasing and bandwidth decreasing as this notion of distance increases. It is favorable to cluster highly connected subgraphs on the same PE to minimize communication network load.

The second case is generally of higher importance than the first as soon as off-chip communication occurs. In our single-FPGA system however, all PEs are equally distant, as even messages that a PE sends to itself will cross the network to travel from the outgoing endpoint to the incoming endpoint. Therefore we only need to consider the first point.

The original GraVF partitioning algorithm was designed for uniform graphs and expected users to set PE parameters so that the available vertex storage exactly matched the graph size. It loads vertices consecutively, first using all available space in PE 0 before starting to fill PE 1. If the graph does not fill the system completely, this leads to unbalanced loads, and even PEs with no assigned vertices at all. Exacerbating this, in a randomly ordered edgelist, high-degree vertices have a higher chance of being discovered early, and this assignment method will cluster them all in the early PEs.

We instead distribute vertices using a round-robin assignment strategy. Unless the edgelist is sorted in a very peculiar manner, high-degree vertices have an equal chance to be assigned to any PE. On scale-free graphs, where vertices have a highly variable degree, we obtain outgoing edge distributions (a proxy for workload size) with standard deviations around 5%, compared with over 200% for the original.

6. EVALUATION

We use a Micron AC-510 FPGA board as a testing platform to evaluate our modified system. This board connects a Xilinx Kintex Ultrascale KU060 FPGA to a 4GB Micron HMC 1.1 chip using two half-width full-duplex links. Each link serves five user ports inside the FPGA, as described in section 4.1. One port is reserved for communication with the host PC, leaving nine useable ports for the design. We therefore use designs with 9 PEs for our evaluation.

The kronecker R-MAT generator from the graph500 benchmark[11] is used to generate input graphs. It has two parameters: scale ($\log_2$ of the number of vertices) and edgefactor (ratio between vertices and edges in the graph). We generate graphs of various scales with the default edgefactor of 16 to compare the capacity of the different systems. The graphs produced by this R-MAT generator are scale-free (i.e. the degree distribution follows a power law). They more closely resemble many real-world workloads, but they are significantly more challenging to process than the uniform graphs used in [9].

We evaluate the effect of our modifications by comparing the following three designs:

- **Original** The baseline before our modifications.
- **BRAM** Applying all modifications except the external memory described in section 4.
- **HMC** The improved system with all modifications described in this paper, including external memory.

Two benchmark algorithms, breadth-first search (BFS) and PageRank (PR) are used. For each design and each algorithm, we find the largest scale of graph that can fit into our target platform. Table 4 shows how the systems compare for BFS and PR respectively.

Table 4: System Comparison

	Design	Scale	Runtime	MTEPS
BFS	Original	13	31 ms	229
	BRAM	14	13.5 ms	1105
	HMC	17	131 ms	1001
PR	Original	13	1155 ms	187
	BRAM	14	561 ms	801
	HMC	16	1073 ms	1789

Comparison between the original GraVF and the BRAM-only improved version shows that the original partitioning algorithm of GraVF is highly unfavorable for scale-free input graphs. With the improved partitioning algorithm, performance returns to over 100 MTEPS per PE, the same as obtained on uniform graphs in [9]. This also shows that our modifications to the synchronization protocol did not adversely affect performance.

The addition of off-chip memory frees enough BRAM resources to increase graph size by an extra three scale factors, or roughly one order of magnitude. According to the GUPS benchmark included with the PicoFramework on the AC-510, the practical maximum bandwidth achievable with random read-only 16 byte accesses is 6.4 GB/s. With an edge taking 4 bytes to store, this corresponds to a performance bound of 1.6 GTEPS. Our accesses are not truly random, as with an average degree of 16, reading one adjacency list uses a mean of four consecutive requests. This allows us to reach 112% of the projected performance.

7. RELATED WORK

BFS, due to its ease of implementation and its prominence as the graph500 benchmark, has been used to explore the performance of FPGA-based systems with many different memory technologies. These highly specialized implementations can give an idea of the maximum performance obtainable from various platforms: Umuroglu et al.[12] extract 172 MTEPS from a ZedBoard, Wang et al.[3] obtain 790 MTEPS using DDR3, Zhang et al.[5] achieve 166 MTEPS on the same Micron AC-510 as our work, CyGraph[4] and Betkaoui et al.[2] both reach similar speeds of 2.5-3 GTEPS using Convey's high-bandwidth HC-2 system, and Convey themselves boasted of 14.6 GTEPS from their later MX-100[1]. These results also underline again that graph algorithms are primarily limited by memory bandwidth.

Beyond single-algorithm FPGA implementations, several frameworks for graph processing on FPGA have been proposed. Vertex-centric programming models, which have proven well-suited from both a programmability and a performance perspective on CPU systems[13], are an equally popular choice on FPGA.

GraphStep[6] is an early exploration of the potential of FPGAs for graph processing, with a vertex-centric programming model based on BSP, a precursor to Pregel[14]. Its model is therefore quite similar to the Pregel-inspired GraVF. GraphGen[7] is of particular interest since, like GraVF, it provides a fully automatic compilation chain. Unfortunately its single-PE system lacks scalability, with pipelining inside the user-provided kernels the only source of parallelism. GraphSoC[8] uses a custom softcore-based approach where vertex kernels are described as four C++ functions, which are turned into custom instructions using high-level synthesis. It is not mentioned how ForeGraph[10] is programmed, but it adopts the GraphChi[15] data handling methods, so it is likely to also be vertex-centric. All of the above frameworks except GraphGen and ForeGraph use internal BRAM to store the graph, and are consequently limited in graph size. ForeGraph and GraphStep also connect multiple FPGAs together to improve the capacity of their system. Table 5 summarizes the performance of these works.

Table 5: Performance of other works (normalized per FPGA board). (*)Denotes results obtained in simulation only.

Design	Input Size	MTEPS
GraphStep(*)	64K Edges	168-450
GraphGen	341K Edges	459
GraphSoc	126K Edges	approx. 100
ForeGraph(*)	490M Edges	464
This work	3.7M Edges	1001

8. CONCLUSION

In this paper, we propose several improvements to the GraVF framework. We modify the synchronization mechanism to allow substitution of interconnects. A better partitioning algorithm allows efficient handling of scale-free graphs. Addition of off-chip memory increases the maximum graph size the system can handle by one order of magnitude, while preserving system throughput of 100 MTEPS per PE. In the future, we plan to extend this work to multiple FPGAs to further increase the capacity of the system.

9. ACKNOWLEDGEMENTS

We thank Micron for the loan of the AC-510 development platform and tools. This work was supported in part by the Research Grants Council of Hong Kong (Project GRF 17245716), and the Croucher Foundation (Croucher Innovation Award 2013).

10. REFERENCES

[1] Convey Computer. New Convey MX demonstrates leading power/performance on Graph 500 benchmark, November 2012. Press release.

[2] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk. A reconfigurable computing approach for efficient and scalable parallel graph exploration. In *23rd International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, July 2012.

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

[4] O.G. 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*, pages 228–235, May 2014.

[5] J. Zhang, S. Khoram, and J. Li. Boosting the performance of FPGA-based graph processor using Hybrid Memory Cube: A case for breadth first search. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2017.

[6] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T.E. Uribe, T.F. Knight Jr., and A. DeHon. GraphStep: A system architecture for sparse-graph algorithms. In *14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2006.

[7] 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 *22nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 25–28. IEEE, 2014.

[8] N. Kapre. Custom FPGA-based soft-processors for sparse graph acceleration. In *26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2015.

[9] N. Engelhardt and H. K. H. So. GraVF: A vertex-centric distributed graph processing framework on FPGAs. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016.

[10] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang. Foregraph: Exploring large-scale graph processing on multi-FPGA architecture. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2017.

[11] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the Graph500. *Cray Users Group (CUG)*, 2010.

[12] Y. Umuroglu, D. Morrison, and M. Jahre. Hybrid breadth-first search on a single-chip FPGA-CPU heterogeneous platform. In *International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2015.

[13] R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2):25:1–25:39, October 2015.

[14] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *ACM SIGMOD International Conference on Management of Data*. ACM, 2010.

[15] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale Graph Computation on Just a PC. In *10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2012.