FlexGraph: A Reconfigurable Graph Analytics Accelerator for Heterogeneous CPU-FPGA Architectures

Blaise-Pascal Tine
Georgia Institute of Technology
btine3@gatech.edu

Sudhakar Yalamanchili
Georgia Institute of Technology
sudha@gatech.edu

ABSTRACT

In recent years, The end of Dennard Scaling is pushing the computer architecture community towards designing more specialized and energy efficient systems. This move has led to a new application domain for FPGAs and the emergence of heterogeneous CPU-FPGA computing platforms, enabling the design of new energy efficient FPGA accelerators for domain specific applications. Graph Analytics, one the largest applications in production data centers today, faces scaling challenges with its ever increasing workload size, inherent sparsity and memory-bound characteristic. We present FlexGraph, a flexible and energy-efficient Graph Analytics Accelerator for heterogeneous CPU-FPGA architectures. FlexGraph uses a Doubly Compressed Sparse Matrix format to eliminate unnecessary data transfers and reduce storage requirements. Our architecture allows support for other compressed format by decoupling the matrix data structure traversal from the compute and memory units. We introduce memory access hardware primitives for unstructured fine-grained accesses on cache-coherent shared memory. Flexgraph uses a C++ Hardware Generation Language to extend its vertex programming model with domain centric reconfigurability in an integrated single source development environment, providing a design efficient alternative to High Level Synthesis. Flexgraph achieves X GFlop/s, performing Nx faster than HLS implementation.

1. INTRODUCTION

The end of Dennard Scaling [1] has moved the focus of computer architecture designers towards power and energy efficient architectures. However, energy efficient solutions such as ASIC designs present a serious limitation in flexibility and production cycle. These constraints have pushed production data centers towards using FPGA-based accelerators [2] for their reconfigurability and energy savings when compared to general-purpose graphics processing units (GPUs). This trend has led to emergence of heterogeneous CPU-FPGA computing platforms [3] [4], enabling fast development of new energy efficient accelerators [5] for domain specific applications. Graph Analytics is one the largest applications in production data centers today [6], spanning domains such as bio-informatics, social network, mining, cyber-security, etc. Graph Analytics performance suffers from workload imbalance, frequent updates, limited data locality and low compute communication ratio making it memory-bound and energy inefficient. This problem is further exacerbated with the ever increasing size of its dataset, presenting a scalability challenge for the industry. Several solutions have been proposed to address this problem both at the software level, with better algorithms and programming abstraction [7] [8] [9] [10] [11], and at the hardware level with custom accelerators [12] [13] [14]. Most Graph Analytics accelerators [12] [13] [14] define a proprietary graph data structure to exploit efficient computation on their hardware, limiting flexibility on the host processor for efficient software processing. FlexGraph uses sparse matrices as underlying data structure to enable efficient computation in a shared CPU-FPGA environment where both the host processor and the accelerator are modifying the same graph. However, accessing unstructured fine-grained data on a cache coherent FPGA fabric pauses challenges because of the restricted coarse-grained cache line access granularity and longer miss penalty. Additionally, generalized sparse matrix storage encoding COO, ELL, CSR, CSC, lack the same level of compaction as directly using non-zero edges list [12], pausing a problem for energy efficiency and memory bandwidth for hyper-sparse Graph Analytics application. FlexGraph uses a Doubly Compressed Column Based Sparse matrix encoding similar to GraphMath [9], however introducing additional indirections to the matrix traversal path. We decoupled FlexGraph's matrix traversal unit from the rest of the compute and memory fabric to scaling by increasing memory bandwidth and provide support for other formats without altering the rest of the system.

Our contributions are as follows:

1. A specialized Graph Analytics Accelerator for heterogeneous CPU-FPGA fabric that provide efficient collaborative computation while maximizing energy efficiency.
2. A decoupled data structure traversal and compute architecture for sparse matrices enabling maximum bandwidth utilization and compute scaling.
3. We introduce hardware primitives for unstructured fine-grained accesses on coherent shared memory architectures.
4. A Doubly-Compressed Sparse Matrix-Sparse Vector Multiplication Accelerator implementation on FPGA. 5. A domain specific accelerator with single source programming and reconfiguration environment providing a design-efficient alternative to High-Level Synthesis.

The remainder of this paper is organized as follows: Section 2 provides a background and motivation for the FlexGrah accelerator, section 3 describes FlexGraph's architecture, section 4 describe FlexGraph's software-hardware codesign using Cash [15] framework, section 5 describes the experimental setup, section 6 describes our results analysis, section 7 describes the related work, section 8 summarizes our main contribution and results.

2. BACKGROUND AND MOTIVATION

In this section, we provide the background and motivation for this work.

2.1 Graph Vertex Processing

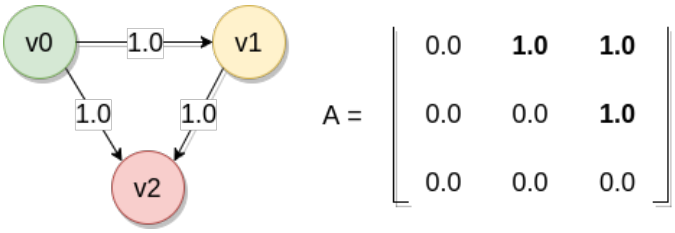

Figure 1: Sample Graph Representation

$$\#Edges = A^T \times Identity \tag{1}$$

$$\#Edges = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 2 \end{pmatrix} \tag{2}$$

Figure 2: Incoming Edges Algorithm

```
GraphProgram(G, P, N) :
  for N iterations :
    x := Assign(P)
    y := Process(G,x)
    P := Apply(P,y)

e.g. FindIncomingEdges(G=A, P=I, N=1) :
       Assign(P)     := P
  with Process(G,x)  := G * x
       Apply(P,y)    := y
```

Listing 1: Graph Vertex Processing Model

A large variety of programming models have been proposed for describing Graph algorithms, expressing computation using vertex operations on matrices [9] [11], [10], [16] [7] task-based models [8] or domain-specific languages [17]. The vertex programming model has show great adoption for its ease of abstraction for describing algorithms using Linear Algebra and its efficient computation on commodity multi-core processors.

Figure 1 shows a simple 3 vertices graph example with its corresponding 3x3 matrix representation. For every edge between a source and destination vertex, the corresponding row and column entry in the matrix is activate. For instance, the matrix entry in row 0 and column 1 represents the edge between source *v0* and destination *v1*. using the matrix representation, we can apply an algorithm on the graph to calculate the total number of incoming edges at each vertex. This algorithm can be expressed using a simple matrix-vector operation as shown in equations (1) and (2) in Figure 2. Several graph algorithms, including Breadth First Search (BFS) [18], PageRank [19], Single Source Shortest-Path (SSSP) [20], can be described similarly using the vertex programming model [9].

Listing 1 shows the three stages of a generalized graph vertex processing model - *Assign*, *Process* and *Apply*, given a graph *G*, some vertex properties *P* and an iteration count *N*. The *Assign* stage generates an input vector *x* using the vertex properties *P*. The *Process* stage applies the input vector *x* to the graph *G* and generates an output vector *y*. The *Apply* gather the resulting output vector to update the vertex properties for the next iteration. The graph program executes the three stages for several iterations until it converges. A direct mapping of our incoming edges algorithm to this processing model is also provided in Listing 1, where the identity vector is used for the vector properties. The *Assign* and *Apply* stages of this program are simple identity operators, while the *Process* stage perform a matrix-vector multiplication. The input graph can be represented as a sparse matrix and the computation done using a Sparse Matrix-Sparse Vector Multiplication (SpMSpV) kernel which can be accelerated on multi-core CPUs [9] or on GPUs [16]. FlexGraph's objective is to provide a more energy efficient graph processing backend using a FPGA for hardware acceleration [2].

2.2 Intel Heterogeneous CPU-FPGA Platform

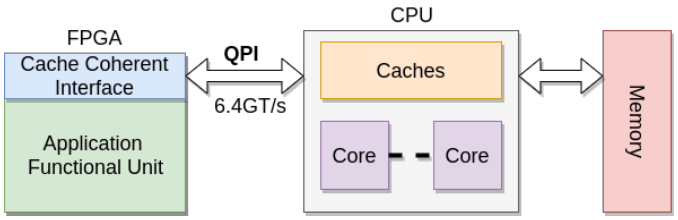

Figure 3: Intel HARP Architecture

2.3 Collaborative CPU-FPGA Computation

3. FLEXGRAPH ARCHITECTURE

In this section, we describes the overall architecture of the FlexGraph Accelerator.

3.1 The DCSC Matrix Format

Most Data Analytics graphs are represented in a sparse matrix format to save memory space due to the large amount of empty edges that exist. FlexGraph's matrices are stored using a Doubly Compressed Sparse Column (DCSC) [21] format. The format allows minimal traversal into the sparse matrix structure, saving necessary memory bandwidth when fetching empty columns. Figure 4 shows a sample matrix A = {(0,5,a), (0,7,b), (6,3,c), (7,1,d)} with four non-zero edges (src, dst, weight) encoded using DCSC versus the conventional CSC [22] format. The traversal over CSC requires ac-

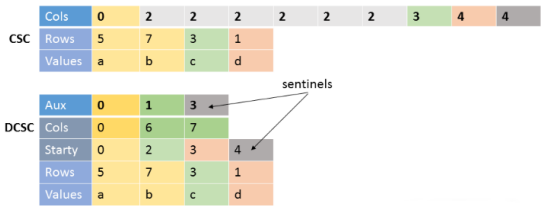

Figure 4: DCSC hyperspace matrix format

cessing all consecutive column ranges *(cols[i+2] - cols[i])* even though most of the distances are empty. DCSC saves on bandwidth by using an additional indirection buffer inside its structure to only encode non-zero columns. It is important to note that the saving this format introduced additional space and compute overhead that only pays off when the matrix is very large and sparse, a prevalent characteristic of Graph Analytics datasets.

3.2 The Sparse Matrix Multiplication Kernel

FlexGraph architecture implements a custom Sparse Matrix Vertex Multiplication (SPMV) kernel. Its architecture diverges from conventional implementations in two ways: First, it uses the dirty masks from the input vertices to select the edges to process and return a new dirty masks capturing the intersection of the active input vertices and the non-zero edges actually visited. Second, it matrix data structure (DCSC) contains an additional indirection buffer for accessing the matrix columns. These two properties present unique performance challenges when designing the accelerator. Listing 1 shows the pseudo-code of the SPMV kernel. The program iterates through each column and fetch the corresponding input vertex active mask to check if the column should be process, then access all the non-zero rows of the matrix for that column to evaluate each edge. In the code's comments on the right are highlighted the different types performance hogs present; semi-random memory accesses (*), fully-random memory accesses (!) and control branches (?). The memory accesses for the columns and rows data (lines 4 and 9) are semi-random because only the access cannot be predicted and after that the address is simply incremented. FlexGraph's architecture attempts to address some of those performance hogs using several optimizations detailed in section 4.

```
1  def SPMV_kernel(x_values, x_activemask) :
2    y_values = {0}, y_activemask = {0}
3      for col in (c_start, c_end) :
4        (a_x, r_start, r_end) = coldata[col]
       # *
5        x_value  = x_values[a_x]
       # !
6        x_active = x_activemask[a_x]
       # !
7        if (x_active):
       # ?
8          for row in (r_start, r_end):
9            (a_y, a_value) = rowdata[row]
       # *
10           y_values[a_y] += a_value * x_value
       # !
11           x_activemask[a_y] = true
       # !
12   return (y_values, y_activemask)
```

Listing 2: Pseudo-code for SPMV kernel

3.3 FlexGraph Microarchitecture

Externally, FlexGraph input and output signals implement an Accelerator Functional Unit (AFU) interface defined by Intel's Accelerator Abstraction Layer (AAL) [3]. AFUs implementing the interface are able to bind with the FPGA's board support package (BSP) and seamlessly communicate with the AAL software runnign on the host processor. Listing 2 shows AAL device interface implementation using Cocoh's API. Our FlexGraph Accelerator's class in Cocoh simply derives from this interface and override the *initialize()* function to provide its implementation and Cocoh takes care of generating the corresponding verilog module. It implements three input signals *(start, qpi_in, ctx)* and two output signals *(qpi_out, done)*. The AFU starts execution when the *start* signal is asserted and communicate completion by asserting the *done* signal. The *ctx* signal provides application's specific data like constants and buffers address in the case of FlexGraph. The *qpi* in/out signals implement Intel Quick Path Interconnect (QPI) interface [23], providing single channel read/write ports for accessing external shared memory. The FPGA socket hosts a 64-byte cache coherent interface (CCI) [24] that connects to the host processor's last level cache (LLC) via QPI.

```
1  class aal_device {
2  public:
3    virtual out_t initialize()(
4      const ch_logic& start,
5      const qpi::in_t& qpi_in,
6      const afu_ctx_t& ctx,
7      qpi::out_t& qpi_out,
8      ch_logic& done
9    ) const = 0;
10 };
```

Listing 3: AAL Device Interface in Cocoh C++

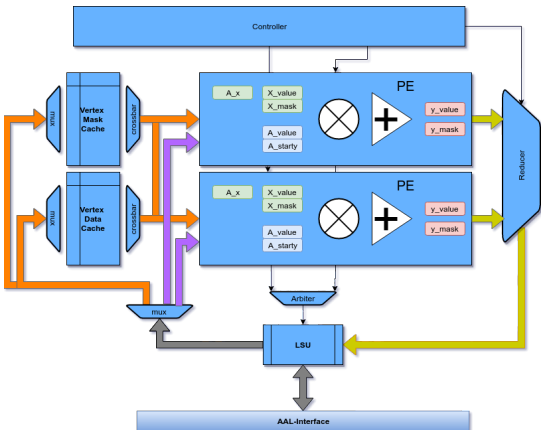

Figure 5: FlexGraph Microarchitecture

Figure 5 illustrates FlexGraph accelerator microarchitecture. It is comprised of four main module types, a controller, processing elements (PEs), a load store unit (LSU) and vertex caches. The main controller is responsible for starting the

accelerator, scheduling tasks for execution, reporting hardware counters and terminating the execution. The processing elements execute partition tasks assigned to them by the controller and communicate with the LSU to access the matrix and vertex data for their partition. They are also responsible for sending their final output result back to memory. The LSU is the module responsible for managing external communication between the accelerator and memory via the QPI interface. It directly binds to the QPI ports defined in listing 2. The vertex caches store intermediate vertex data and active masks for sharing between the processing elements. FlexGraph processing pipeline sightly resembles the SPMV execution steps illustrated in listing 1. We made several important modifications to it to improve performance.

4. FLEXGRAPH PARALLELIZATION

The first optimization we performed in FlexGraph early during the design phase was the support of multiple processing units. The reasoning behind it was to extract the maximum bandwidth out of the LSU such that the QPi is always busy processing a request when some processing element are stalled waiting for their data to return. the other advantage of parallelization for FlexGraph is to increase the overall accelerator throughput by processing multiple a larger chunk of the workload per unit of time. The scheduling of the partitions for execution on the processing elements is controlled by a dispatch unit inside the main controller module. The dispatch unit fetches partitions data from the LSU and pass down each partition in a first come first serve fashion to the processing elements.

4.1 DCSC Matrix partitioning

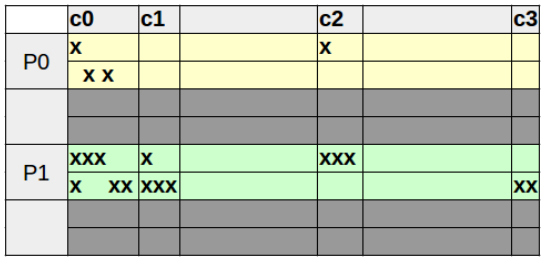

Figure 6: DCSC Matrix Partitioning

To enable the parallelization of FlexGraph tasks, the DCSC sparse matrix is first partitioned into aligned partitions of 32 consecutive rows containing non-zero edges. we choose a partition size of 32 mainly to match the 32-bit size of the bitmasks encoding the vertices that are active. These masks are used by the software on the host processor to determine the regions of the acceleration's output buffer that have been updated. Figure 6 illustrates sample matrix partitioning in which the non-zero horizontal regions have been broken into two partitions P0 and P1. The non-zero column ranges *(c0, c1, c2, c3)* represent the selected chunks covered by each partition. Partition P0 will only contain ranges *c0* and *c2* while P2 has non-zeros in all four ranges. The partitioning scheme is not ideal because of the workload in-balance that might exist when the number of non-zero varies disproportionally between partitions.

4.2 Synchronising Memory Accesses

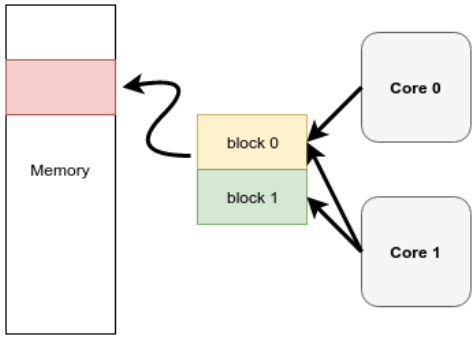

Figure 7: Active Masks Write Synchronization

An implementation challenge we faced when supporting multiple processing elements in FlexGraph was the synchronization of write accesses for the output active masks. Because the LSU data transfer granularity is 64 byte (matching CCI [24]), FlexGraph need to manage shared write accesses to the same block to avoid having a processing element override the content of another one. Luckily this doesn't pose any problem for writing out partition output values because the 32 rows within a partition occupy 32 * 4 bytes = 2 x 64 bytes blocks. Because the partitioning reinforces 32 rows alignment, each processing element can safely write into their assigned blocks. However, writing out the active masks poses a challenge because a single 4-byte mask encodes the active states of all 32 rows in a partition. This causes the processing elements to potentially share a single 64-byte block when writing the masks out to memory. To alleviate the contention, FlexGraph keep N active 64-byte blocks locally assigned to either one of the N processing elements. It assigns an ownership bit mask to each block to track their reference count and the processing element assigned to them. It also tracks the current address value assigned to each block. When a processing element is ready to write its 4-byte active mask, it passes the mask address and value to the LSU. The LSU goes to last active block the processing element previous wrote to and check if the block address matches. If so, it simply adds the new content to the existing block. If there is no match, it clears its ownership bit and flushes the block to memory if the ownership mask goes to zero. Then it looks up the other active block if anyone already has the address to use it, otherwise if it acquires the last unused block. Figure 7 shows a simplified illustration of the scheme with two processing elements. We don't need to keep more than N blocks in local storage for this scheme to work because the block address references are always incremental and never regress, meaning that there is always going to be free block available to use. It is important to also point out that for this scheme to work, the operation has to be atomic. FlexGraph has a single communication channel between all processing elements and the LSU which does some round robin arbitration and blocks the write mask request until it is committed. We also investigated an alternative solution to avoid this synchronization, which is simply increasing the partition size to 16 * 32 rows, allowing the aggregated active masks to occupy a full 64-byte block. They were two major issues with

this approach. Firstly, the total size of all resident partitions will be too large if we support multiple processing elements. Secondly, it will worsen the workload imbalance that is already present with 32 rows.

5. FLEXGRAPH OPTIMIZATIONS

This section describes some optimizations we added to FlexGraph to improve its performance.

5.1 Optimizing Memory Accesses via Stream Buffers

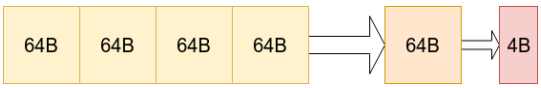

Figure 8: Stream Buffers

As stated before, FlexGraph's LSU transfer granularity is 64 bytes, which is much larger than the 4 bytes of elements accessed by the processing elements. To save on bandwidth, the LSU uses stream buffers that fetch a 64-byte blocks of data but extracts 4-byte elements at the time. It is implemented using a fifo structure in the back-end where 64-byte responses from QPI go into. In the front-end, there is a temporary 64-byte block that is extracted from the fifo on demand to deliver 4-byte element at the time. We employ a shift register to extract the 4-byte element to pass it to the processing element. Figure 8 shows an illustration of the stream buffer concept.

5.2 Caching Vertex Values and Masks

Another optimization we implemented in FlexGraph is the caching of vertex values and masks. Looking at the SPMV speudo-code in listing 1, we can observe in lines 5 and 6 that there is a random memory access to buffers *x_values* and *x_activemask*. FlexGraph employs a stream buffer like concept to consume 64-byte vertex data once they arrive inside the processing element to each iteration loop. However, the same block can be referenced by another processing element and sharing them inside the cache structure can save unnecessary memory traffic. We implemented two small fully associative caches with first-in-First-out replacement to hold active vertex data during execution. The caches are implemented inside the LSU which is responsible for managing them. When a vertex request arrives from a processing element, the LSU first looks up the cache if the block is already present and return it. If the block is not there, it sends the request to QPI. QPI output interface support a 14-bit metadata field that is used to identify the block once it is returned. We use that field to store the index of the block such that we can compute the cache tag when it arrives. Because the cache is small, we implemented a one cycle tag lookup for the LSU to known if the block is present and accessible.

5.3 Executing Non-blocking Memory writes

FlexGraph memory writes are all non-blocking, this applies to both the output values and active masks. This allows the processing elements to push their write request and resume execution while the LSU is processing them. Upon write responses from the QPI channel, the LSU internally keeps track of the count of outstanding requested to ensure that all writes have completed. The QPI interface exposes two write response ports by which the memory replies could be sent. This allows servicing two responses simultaneously. The LSU implements a mechanism to process both channels simultaneously when they are active. At completion time when all processing elements are done executing the current run, the controller waits for all outstanding write requests to complete before asserting the *done* signal.

6. FLEXGRAPH SOFTWARE-HARDWARE CODESIGN

7. RELATED WORK

In this section, we discuss prior works related to this project.

7.1 GraphMat

GraphMat [9] is a high-performance Graph Analytics Framework for multi-core CPU platforms. It defines a C++ template-based vertex programming model for describing various graph algorithms in software and use Sparse Matrix Vector Multiplication (SPMV) as underlying compute kernel for efficient parallel processing. GraphMat also uses a doubly-compressed DCSC [21] matrix format for storage and compute efficiency. FlexGraph's programming model is an adaptation of GraphMat's model for collaborative computation with an FPGA accelerator.

7.2 Graphicionado

Graphicionado [12] is a much recent project that is similar to FlexGraph. In this work, the authors implemented a graph accelerator targeting Intel Xeon platform. Graphicionado also uses a vertex programming model of graphs computation similar to GraphMat [9]. It is a more specialized accelerator compared to FlexGraph in that it implements the entire graph algorithm in hardware, sacrificing flexibility for efficiency. Although the design allows some reconfigurability using FPGA, it still add many inconveniences compared to software. FlexGraph on the other hand only accelerate the graph edges computation in hardware, most of the algorithm is still written expressed in C++ and runs on the host processor like in GraphMat. FlexGraph reconfigurability is minimal and restricted to the SPMV module (changing the matrix datatype or the reduce operation). Graphicionado main architecture strength is their efficient optimizations of the graph access patterns to reduce stalls inside the pipeline.

7.3 GraphOps

GraphOps [14] is another graph accelerator also targeting FPGAs like FlexGraph. In this work, the authors propose an accelerator architecture that efficiently process a graph in memory encoded using a proposed locality-optimized scheme. Their main contribution is the graph storage representation which provides an efficient memory access pattern. A drawback of their proposal is that fact that the host processor has to pre-process the graph before the accelerator can consume it, which can add some considerable latency, considering that Graph Analytics algorithms tend to update the graph frequently.

8. REFERENCES

[1] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pp. 365–376, June 2011.

[2] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," *SIGARCH Comput. Archit. News*, vol. 42, pp. 13–24, June 2014.

[3] P. Gupta, "Intel xeon+fpga platform for the data center." `http://reconfigurablecomputing4themasses.net/files/2.2%20PK.pdf`.

[4] Y. L. Fei Chen, "Fpga acceleration in a power8 cloud." `https://openpowerfoundation.org/blogs/fpga-acceleration-in-a-power8-cloud`.

[5] Y.-k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "A quantitative analysis on microarchitectures of modern cpu-fpga platforms," in *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, (New York, NY, USA), pp. 109:1–109:6, ACM, 2016.

[6] D. Gage, "The new shape of big data." `http://www.wsj.com/articles/SB10001424127887323452204578288264046780392`, 2013.

[7] L. Page, S. Brin, R. Motwani, and T. Winograd, "Apache spark's api for graph and graph-parallel computation." `http://spark.apache.org/graphx/`.

[8] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 456–471, ACM, 2013.

[9] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proc. VLDB Endow.*, vol. 8, pp. 1214–1225, July 2015.

[10] 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*, SIGMOD '10, (New York, NY, USA), pp. 135–146, ACM, 2010.

[11] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, pp. 716–727, Apr. 2012.

[12] T. J. Ham, W. Lisa, S. Narayanan, S. Nadathur, and M. Margaret, "Graphicionado: A high-performance and energy efficient accelerator for graph analytics," in *the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[13] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "Tesseract: A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 105–117, ACM, 2015.

[14] T. Oguntebi and K. Olukotun, "Graphops: A dataflow library for graph analytics acceleration," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, (New York, NY, USA), pp. 111–117, ACM, 2016.

[15] B. P. Tine, "Cash: A c++ api and simulator for hardware," 2017.

[16] Z. Fu, M. Personick, and B. Thompson, "Mapgraph: A high level api for fast development of high performance graph analytics on gpus," in *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, (New York, NY, USA), pp. 2:1–2:6, ACM, 2014.

[17] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-marl: A dsl for easy and efficient graph analysis," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (New York, NY, USA), pp. 349–362, ACM, 2012.

[18] wikipedia, "Breadth-first search." `https://en.wikipedia.org/wiki/Breadth-first_search`.

[19] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web.," Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[20] wikipedia, "Shortest path problem." `https://en.wikipedia.org/wiki/Shortest_path_problem`.

[21] A. Buluc and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–11, April 2008.

[22] wikipedia, "Sparse matrix." `https://en.wikipedia.org/wiki/Sparse_matrix`.

[23] wikipedia, "Intel quickpath interconnect." `https://en.wikipedia.org/wiki/Intel_QuickPath_Interconnect`.

[24] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia, H. Mitchel, S. Subhaschandra, A. Sheiman, T. Whisonant, and P. Gupta, "A reconfigurable computing system based on a cache-coherent fabric," in *2011 International Conference on Reconfigurable Computing and FPGAs*, pp. 80–85, Nov 2011.