Graph Processing on FPGAs: Taxonomy, Survey, Challenges

Towards Understanding of Modern Graph Processing, Storage, and Analytics

MACIEJ BESTA*, DIMITRI STANOJEVIC*, Department of Computer Science, ETH Zurich
JOHANNES DE FINE LICHT, TAL BEN-NUN, Department of Computer Science, ETH Zurich
TORSTEN HOEFLER, Department of Computer Science, ETH Zurich

Graph processing has become an important part of various areas, such as machine learning, computational sciences, medical applications, social network analysis, and many others. Various graphs, for example web or social networks, may contain up to trillions of edges. The sheer size of such datasets, combined with the irregular nature of graph processing, poses unique challenges for the runtime and the consumed power. Field Programmable Gate Arrays (FPGAs) can be an energy-efficient solution to deliver specialized hardware for graph processing. This is reflected by the recent interest in developing various graph algorithms and graph processing frameworks on FPGAs. To facilitate understanding of this emerging domain, we present the first survey and taxonomy on graph computations on FPGAs. Our survey describes and categorizes existing schemes and explains key ideas. Finally, we discuss research and engineering challenges to outline the future of graph computations on FPGAs.

CCS Concepts: • **General and reference → Surveys and overviews**; • **Computer systems organization → Reconfigurable computing**; • **Hardware → Reconfigurable logic and FPGAs**; **Hardware accelerators**; **Reconfigurable logic applications**; • **Mathematics of computing** → Graph theory; • **Information systems** → Data layout; • **Theory of computation** → Parallel algorithms; Vector / streaming algorithms;

Additional Key Words and Phrases: FPGAs, Reconfigurable Architectures, Graph Processing, Graph Computations, Graph Analytics, Vertex-Centric, Edge-Centric

ACM Reference format:
Maciej Besta*, Dimitri Stanojevic*, Johannes de Fine Licht, Tal Ben-Nun, and Torsten Hoefler. 2019. Graph Processing on FPGAs: Taxonomy, Survey, Challenges. 31 pages.

1 INTRODUCTION

Graph processing underlies many computational problems in social network analysis, machine learning, computational science, and others [66, 84]. Designing efficient graph algorithms is challenging due to several properties of graph computations such as irregular communication patterns or little locality. These properties, combined with the sheer size of graph datasets (up to trillions of edges [35]), make graph processing and graph algorithms consume large amounts of energy.

Most graph algorithms are communication-heavy rather than compute-heavy: more time is spent on accessing and copying data than on the actual computation. For example, in a Breadth-First

*Both authors contributed equally to the work

Search (BFS) traversal [39], a fundamental graph algorithm, one accesses the neighbors of each vertex. In many graphs, for example various social networks, most of these neighborhoods are small (i.e., contain up to tens of vertices), while some are large (i.e., may even contain more than half of all the vertices in a graph). General purpose CPUs are not ideal for such data accesses: They have fixed memory access granularity based on cache line sizes, do not offer flexible high-degree parallelism, and their caches do not work effectively for irregular graph processing that have little or no temporal and spatial locality. GPUs, on the other hand, offer massive parallelism, but exhibit significantly reduced performance when the internal cores do not execute the same instruction (i.e., warp divergence), which is common in graphs with varying degrees.

Field Programmable Gate Arrays (FPGAs) are integrated circuits that can be reprogrammed using hardware description languages. This allows for rapid prototyping of application-specific hardware. An FPGA consists of an array of logic blocks that can be arbitrarily rewired and configured to perform different logical operations. FPGAs usually use low clock frequencies of ≈100–200MHz, but they enable building custom hardware optimized for a given algorithm. Data can directly be streamed to the FPGA without the need to decode instructions, as done by the CPU. This data can then be processed in pipelines or by a network of processing units that is implemented on the FPGA, expressing parallelism at a massive scale. Another major advantage of FPGAs is the large cumulated bandwidth of their on-chip memory. Memory units on the FPGA, such as block RAM (BRAM), can be used to store reusable data to exploit temporal locality, avoiding expensive interactions with main memory. On a Xilinx Alveo U250 FPGA, 2566 memory blocks with 72 bit ports yield an on-chip bandwidth of 7 TB/s at 300 MHz, compared to 1 TB/s for full 256-bit AVX throughput at maximum turbo clock on a 12-core Intel Xeon Processor E5-4640 v4 CPU. In practice, the advantage of FPGAs can be much higher, as buffering strategies are programmed explicitly, as opposed to the fixed replacement scheme on a CPU.

Developing an application-specific FPGA accelerator usually requires more effort than implementing the same algorithm on the CPU or GPU. There are also many other challenges. For example, modern FPGAs contain in the order of tens of MB of BRAM memory, which is not large enough to hold entire graph data sets used in today's computations. Therefore, BRAM must be used as efficiently as possible, for example by better optimizing memory access patterns. Thus, a significant amount of research has been put into developing both specific graph algorithms on FPGAs and graph processing frameworks that allow to implement various graph algorithms easier, without having to develop everything from scratch [40, 41, 49, 76, 79, 80, 85, 93, 94, 125, 127, 129, 130, 134–136].

This paper provides the first taxonomy and survey that attempts to cover all the associated areas of graph processing on FPGAs. Our goal is to (1) *exhaustively* describe related work, (2) illustrate and explain the *key ideas*, and (3) *systematically* categorize existing algorithms, schemes, techniques, methodologies, and concepts. We focus on all works researching graph computations on FPGAs, both general frameworks as well as implementations of specific graph algorithms.

What Is the Scope of Existing Surveys? To the best of our knowledge, as of yet there is no other survey on FPGAs for graph processing. Only Horawalavithana [128] briefly reviews several hardware accelerators and frameworks for graph computing and discusses problems and design choices. However, the paper only partially focuses on FPGAs and covers only a few selected works.

2 BACKGROUND

We first present concepts used in all the sections and summarize the key symbols in Table 1.

G, $\mathbf{A}$	A graph $G = (V, E)$ and its adjacency matrix; V and E are sets of vertices and edges.
n, m	Numbers of vertices and edges in G; $\|V\| = n$, $\|E\| = m$.
d, $\overline{d}$, D	Average degree, maximum degree, and the diameter of G, respectively.
d_v, N_v	The degree and the sequence of neighbors of a vertex v.
B_{DRAM}	The bandwidth between the FPGA and DRAM.
B_{BRAM}	The bandwidth of a BRAM module.

Table 1. The most important symbols used in the paper.

2.1 Graphs

We model an undirected graph G as a tuple (V, E); V is a set of vertices and $E \subseteq V \times V$ is a set of edges; $|V| = n$ and $|E| = m$. If G is directed, we use the name *arc* to refer to an edge with a specified direction. An edge between vertices v and w is denoted as (v, w). We consider both labeled and unlabeled graphs. If a graph is labeled, $V = \{1, ..., n\}$, unless stated otherwise. We use the name "label" or "ID" interchangeably. N_v and d_v are the neighbors and the degree of a vertex v. G's diameter is D. A subgraph of G is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$. In an *induced* subgraph, E' contains only edges (v, w) such that both v and w are in V'. A path in G is a sequence of edges in G between two vertices v and w: (v, v_1), (v_1, v_2), ..., (v_{n-1}, v_n), (v_n, w).

2.2 Graph Processing Abstractions

Graph algorithms such as BFS can be viewed in either the **traditional combinatorial abstraction** or in the **algebraic abstraction** [18, 73]. In the former, graph algorithms are expressed with data structures such as queues or bags and operations on such structures such as inserting a vertex into a bag [81]. In the latter, graph algorithms are expressed with basic linear algebra structures and operations such as a series of matrix-vector (MV) or matrix-matrix (MM) products over various semirings [74]. Both abstractions have advantages and disadvantages in the context of graph processing. For example, BFS based on MV uses no explicit locking [105] or atomics [106] and has a succinct description. Yet, it may need more work than the traditional BFS [126].

2.3 Graph Representations and Data Structures

We discuss various graph-related structures used in FPGA works.

2.3.1 Adjacency Matrix, Adjacency List, Compressed-Sparse Row. G can be represented as an **adjacency matrix** (AM) or **adjacency lists** (AL). AL uses $O(n \log n)$ bits and AM uses $O\left(n^2\right)$ bits.

When using AL, a graph is stored using a contiguous array with the adjacency data and a structure with offsets to the neighbors of each vertex. When using AM, the graph can be stored using the well-known **Compressed-Sparse Row** (CSR) format [103]. In CSR, the corresponding AM is represented with three arrays: *val*, *col*, and *row*. *val* contains all matrix non-zeros (that correspond to G's edges) in the row major order. *col* contains the column index for each corresponding value in *val*; it has the same size ($O(m)$). Finally, *row* contains starting indices in *val* (and *col*) of the beginning of each row in the AM (size $O(n)$).

2.3.2 Sparse and Dense Data Structures. Data structures used in graph processing, such as bitvectors, can be either **sparse** or **dense**. In the former, one only stores *non-zero elements* from a given structure, together with any indices necessary to provide locations of these elements. In the latter, *zeros* are also stored. In the case of MV, the adjacency matrix is usually sparse (e.g., when using CSR) while the vector can be sparse or dense, respectively resulting in **sparse-sparse (SpMSpV)** and **sparse-dense (SpMV)** MV products. The latter entail more work but offer more potential for vectorization; the former is work-efficient but has more irregular memory accesses.

Graph problem + time complexity of the best (or established) sequential algorithm(s)		**Associated graph/vertex property**	**Associated example algorithm and its time/work complexity (in the PRAM CRCW model [6])**		**Selected application**
Single-Source Shortest Path (SSSP) (unweighted) [39]	$O(m+n)$ [39]	Length of a shortest path	BFS [19]	$O\left(D\overline{d} + D\log m\right)$, $O(m)$	Bipartite testing
SSSP (weighted) [39]	$O(m + n\log n)$ [52], $O(m)$ [115]	Length of a shortest path	Δ–Stepping [88], Bellman-Ford [9]	$O\left(\overline{d} \cdot \overline{W_P} \cdot \log n + \log^2 n\right)^{\dagger}$, $O\left(n + m + \overline{d} \cdot \overline{W_P} \cdot \log n\right)^{\dagger}$	Robotics, VLSI design
All-Pairs Shortest Path (APSP) (unweighted) [39]	$O\left(mn + n^2\right)$ [39], $O(n^3)$ [50]	Length of a shortest path	BFS [19]	$O\left(D\overline{d} + D\log m\right)$, $O(nm)$	Urban planning
All-Pairs Shortest Path (APSP) (weighted) [39]	$O\left(mn + n^2\log n\right)$ [52]	Length of a shortest path	Han et al. [57]	$O\left(n^2\right)$, $O\left(n^3\right)$	Traffic routing
[Weakly, Strongly] Connected Components [39], Reachability	$O(m+n)$ [39]	#Connected components, Reachability	Shiloach-Vishkin [107]	$O(\log n)$, $O(m\log n)$	Verifying connectivity
Triangle Counting (TC) [108]	$O\left(m\overline{d}\right)$, $O\left(m^{3/2}\right)$ [104]	#Triangles	GAPBS kernel [8]	$O\left(\overline{d}^2\right)$, $O\left(m\overline{d}\right)$	Cluster analysis
Minimum Spanning Tree (MST) [39]	$O(m\log n)$ [39], $O(m\alpha(m, n))$ [33]	MST weight	Boruvka [28]	$O(\log n)$, $O(m\log n)$	Design of networks
Maximum Weighted Matching (MWM) [97]	$O\left(mn^2\right)$	MWM weight	Blossom Algorithm [47]	–	Comp. chemistry
Betweenness Centrality (BC) [29] (unweighted)	$O(nm)$ [29]	Betweenness	Parallel Brandes [8, 29]	$O\left(nD\overline{d} + nD\log m\right)$, $O(nm)$	Network analysis
BC [29] (weighted)	$O\left(nm + n^2\log n\right)$ [29]	Betweenness	Parallel Brandes [8, 29]	–	Network analysis
Degree Centrality (DC) [39]	$O(m+n)$ [39]	Degree	Simple listing [39]	$O(1)$, $O(m+n)$	Ranking vertices
PageRank (PR) [96]	$O(Im)$ [8]	Rank	GAPBS kernel [8]	$O\left(I\overline{d}\right)$, $O(Im)$	Ranking websites

Table 2. **Overview of fundamental graph problems and algorithms considered in FPGA works.** †Bounds in expectation or with high probability. $\alpha(n, m)$ is the inverse Ackermann function. $\overline{W_P}$ is the maximum shortest path weight between any two vertices. I is the number of iterations in PageRank.

2.4 Graph Problems and Algorithms

We next present graph algorithms that have been implemented on FPGAs. In the survey, we describe the FPGA designs targeting these algorithms in § 5 (implementations of specific algorithms on the FPGA) and in § 6 (implementations within generic graph processing frameworks on the FPGA). A summary of the fundamental graph problems, algorithms, and properties considered in FPGA-related works can be found in Table 2.

2.4.1 Breadth-First Search (BFS). The goal of Breadth-First Search (BFS) [39] is to visit each vertex in G. BFS starts with a specified *root* vertex r and visits all its neighbors N_r. Then, it visits all the unvisited neighbors of r's neighbors, and continues to process each level of neighbors in one iteration. During the execution of BFS, the *frontier of vertices* is a data structure with vertices that have been visited in the previous iteration and might have edges to unvisited vertices. In the very first iteration, the frontier consists only of the root r. In each following i-th iteration, the frontier contains vertices with distance i to the root. The sequential time complexity of BFS is $O(n + m)$.
Traditional BFS In the traditional BFS formulation, a frontier is implemented with a bag. At every iteration, vertices are removed in parallel from the bag and all their unvisited neighbors are inserted into the bag; this process is repeated until the bag is empty.
Algebraic BFS BFS can also be implemented with the MV product over a selected semiring. For example, for the tropical semiring [18], one updates the vector of distances from the root by multiplying it at every iteration by the adjacency matrix.

2.4.2 Connected Components (CC). A connected component is a subgraph of G where any two vertices are connected by some path. A *connected* graph consists of only one connected component. A *disconnected* graph can have several connected components. In the context of directed graphs, a *strongly connected component* (SCC) must contain paths from any vertex to any other vertex. In a *weakly connected component* (WCC), the direction of the edges in a path is not relevant (i.e., computing WCCs is equivalent to finding CCs when treating the input directed graph as undirected). Now, the goal of a CC algorithm is to find all connected components in a given graph G. A simple way to compute CC in linear time is to use BFS and straightforwardly traverse connected components one by one. Another established algorithm for CC has been proposed by Shiloach and Vishkin [107]. It is based on forming trees among the connected vertices and then dynamically shortening them using pointer jumping. In the end, each connected component is represented by one tree consisting of two levels of vertices: a tree root and its children. The parallel (under the CRCW PRAM model [56]) time complexity of this algorithm is $O(\log n)$.

2.4.3 Reachability. Reachability is a problem strongly related to CC. Namely, it answers the question of whether there exists a path between any two vertices. Algorithms for finding connected components can be used to solve this problem.

2.4.4 Single-Source Shortest-Paths (SSSP). A shortest path between two vertices v, w is a path where either the number of edges or, in the case of a weighted graph, the sum of all weights in the path, is the smallest out of all paths between v and w. In the Single-Source Shortest-Paths (SSSP) problem, one finds the shortest path between a given source vertex and all other vertices in the graph. Two well-known solutions are Dijkstra's algorithm [45] and the Bellman-Ford algorithm [9, 51]. The Bellman-Ford algorithm has a sequential time complexity of $O(nm)$ and can be used for graphs with negative edge weights, while Dijkstra's algorithm, if implemented with a Fibonacci heap, has a better sequential time complexity of $O(n \log n + m)$ but cannot handle negative edges.

2.4.5 All-Pairs Shortest-Paths (APSP). The All-Pairs Shortest-Paths (APSP) problem is to find the shortest paths between all pairs of vertices in the graph. One solution, called Johnson's algorithm, is to use the SSSP algorithms such as Dijkstra and Bellman-Ford [67]. In case of unweighted graphs, the algorithm can be further reduced to BFS. The worst-case sequential runtime is $O(n^2 \log n + nm)$. Johnson's algorithm for weighted graphs requires a Fibonacci heap, which may be difficult to implement. Another solution is the Floyd-Warshall algorithm [50], which has the $O(n^3)$ sequential time complexity and is based on dynamic programming.

2.4.6 Minimum Spanning Tree (MST). A spanning tree of a graph is defined as a tree subgraph that includes all the vertices of the graph and a subset of the edges with minimal size. An MST is thus a spanning tree where the edge weight sum is minimal. There exist several algorithms to find the MST of a graph, notably Boruvka's algorithm [28], which runs in parallel at a time complexity of $O(\log n)$; the sequential Prim's Algorithm [101] with $O(m + n \log n)$ complexity (using a Fibonacci heap); and Kruskal's Algorithm [77] with $O(m \log n)$ time complexity.

2.4.7 PageRank (PR). PageRank (PR) [96] is an iterative centrality algorithm that obtains the rank $r(v)$ of each vertex v:

$$r(v) = \frac{1-f}{n} + \sum_{w \in N_v} \frac{f \cdot r(w)}{d_v}$$

where f is a parameter called the damp factor [96]. PR is used to rank websites. Intuitively, a vertex is deemed "important" (has a high rank) if it is being referenced by other high rank vertices.

The papers covered in this survey implement the traditional iterative PR algorithm where, in each iteration, all edges are processed and the PR of every vertex is recomputed according to the above equation. Usually the maximum number of iterations is set, the algorithm halts if the maximum difference between the ranks of a vertex in two iterations converges below a given threshold.

2.4.8 Graphlet Counting (GC), Triangle Counting (TC). Graphlets are small connected induced subgraphs. An example graphlet is a triangle: a cycle with three vertices. The problem of counting graphlets (GC) is to count the number of different graphlets in a graph. There exist many algorithms for counting graphlets of different sizes. For example, TC can be solved in $O(m^{3/2})$ time [104].

2.4.9 Betweenness Centrality (BC). Centrality measures of vertices in a graph determine the "importance" of each vertex . One such measure is Betweenness Centrality [90], which is defined by the ratio of shortest paths in the graph that pass through a given vertex . More formally:

$$BC(v) = \sum_{\substack{u,w \in V \\ u,w \neq v}} \frac{P_v\,(u,w)}{P\,(u,w)},$$

where $P(u,w)$ indicates the number of shortest paths between u and w and P_v is the number of shortest paths that pass through v. To compute BC for every vertex, one can use the Brandes algorithm [29] in parallel [109], which exhibits a total work of $O\,(nm)$.

2.4.10 Maximum Matching (MM). A matching is defined to be a set of edges $E' \subseteq E$, where every vertex in the pairs of E' is unique, i.e., edges do not share vertices. Maximum Cardinality Matching and Maximum Weighted Matching (MWM) are commonly computed types of matchings, where the former is a matching that maximizes $|E'|$ and the latter (only applicable to weighted graphs) maximizes the sum of the edge weights in E'. MWM can be computed exactly using the Blossom Algorithm [47], which is inherently sequential.

2.4.11 Graph-Related Applications. In the same way that some graph algorithms (e.g., BFS) can be implemented with linear algebra operators, many applications outside of graph theory formulate problems as graphs in order to benefit from increased performance. One such example is Deep Neural Networks and the Stochastic Gradient Descent algorithm: in the former, sparsely-connected layers of neurons can be represented as a graph that should be traversed [131]; whereas in the latter, the algorithm itself creates dependencies that can be modeled as a fine-grained graph and scheduled dynamically [69].

Another graph-related application considered in the surveyed works is stereo matching [113], where one accepts a pair of stereo images and outputs the disparity map with depth information of all pixels. This problem is solved with the Tree-Reweighted Message Passing (TRW-S) algorithm [113].

Finally, one work considers spreading activation [82]. Here, a given graph is traversed (starting from a selected subset of vertices) in a manner similar to that of a neural network: certain values called activities are propagated through the graph, updating properties of vertices.

2.4.12 Challenges in Graph Processing for FPGAs. Most challenges for efficient graph processing on FPGAs are similar across different algorithms. BFS, the algorithm considered most often, exhibits irregular memory access patterns because it is hard to predict which vertices will be accessed in the next iteration, before iterating through the neighborhood of vertices currently in the frontier. Vertices with small distances between them in G are not necessarily close to one another in memory. Furthermore, most vertices in graphs used in today's computations have small neighborhoods and thus prefetching subsequent memory blocks does not always improve performance.

2.5 Graph Programming Paradigms, Models, and Techniques

We also present graph programming models used in the surveyed works. A detailed description can be found in work by Kalavri et al. [68].

2.5.1 Vertex-Centric Model. In the vertex-centric model [75, 86], the programmer expresses a graph algorithm from the perspective of vertices. One programs an algorithm by developing a (usually small) routine that is executed *for each vertex in the graph concurrently*. In this routine, one usually has access to the neighbors of a given vertex. Such as approach can lead to many random memory accesses as neighboring vertices may be stored in different regions of the memory. Still, it is often used because many important algorithms such as BFS or PageRank can easily be implemented in this model.

2.5.2 Edge-Centric Streaming Model. In the edge-centric model [102], edges are streamed to the processor in the order in which they are stored in the graph data structure. An edge consists of the labels of the two connected vertices and optionally the edge weight. The processor processes edges one by one and, if necessary, it updates some associated data structures. This way of accessing the graph has major advantages because of its sequential memory access pattern, which improves spatial locality. A disadvantage of this approach is the restriction on the order of loading and thus processing edges. This makes the model less suitable for certain algorithms, for example BFS or SSSP. For example, the edge-centric BFS requires several passes over the edges, with each pass only processing those edges that are currently in the frontier. This takes $O(Dm)$ time, a factor of D more than when using the traditional BFS variant with $O(m)$ time [19].

2.5.3 Gather-Apply-Scatter Model (GAS). Gather-Apply-Scatter (GAS) [78, 83] is similar to the vertex-centric approach. It also offers the vertex-centric view on graph algorithms. However, it additionally splits each iteration into three parts: gather, apply, and scatter. In the **gather** phase, a vertex collects information about its neighboring vertices or edges and optionally reduces them to a single value σ. In the **apply** stage, the state of the vertex is updated based on the previously computed σ, and possibly the properties of the vertex neighbors. Finally, in the **scatter** phase, each vertex propagates its new state to its neighbors. This value will then again be collected in the gather phase of the next iteration. These three phases can be implemented as individual components and for example connected as a pipeline system or in a network of distributed components.

2.5.4 Bulk-Synchronous Parallel (BSP) Model. Bulk-Synchronous Parallel (BSP) [118] is a model for designing and analyzing general (i.e., *not* specifically graph-related) algorithms. Each iteration in the model is called a **superstep**. After each superstep, parallel processes are synchronized using a barrier. Similarly to the GAS model, each iteration is divided into three phases. In first phase, each process conducts any required local computation. In the next phase, the processes send and receive messages. Finally, a barrier synchronization guarantees that the next super-step only begins after all local computations in the current super-step are finished and all messages in this super-step are exchanged. BSP is frequently used to model and analyze distributed graph algorithms [30, 46, 55].

2.5.5 Asynchronous Execution. While BSP imposes a strict global barrier after each iteration, in an asynchronous execution model [83], some processes can advance to the next iteration even before others have finished the previous iteration. In the context of iterative graph algorithms such as PageRank or Shiloach-Vishkin, this enables some vertices to propagate their associated values (e.g., their ranks) to their neighbors more often than others, i.e, not just once per iteration. This can accelerate convergence, but it also requires more complex synchronization mechanisms than in algorithms that use synchronous models.

2.5.6 MapReduce (MR). MapReduce (MR) [44] is a well-known programming model for processing data sets in a parallel, distributed setting. An algorithm based on the MapReduce model is usually composed of several iterations, and each iteration consists of three phases: map, shuffle, reduce. In the **map** phase, a certain map function is applied to every input value and an intermediary result is generated. Next, in the **shuffle** phase, compute nodes can redistribute the outcomes of the map phase. In the final **reduce** phase, a certain reduction function is performed by each node on the data received in the shuffle phase. Such MapReduce iterations can be used to expressed some graph algorithms [37].

2.5.7 Substream-Centric. The substream-centric approach [13] is a paradigm designed to compute semi-streaming graph algorithms efficiently. In substream-centric algorithms, an input stream of data is divided into substreams, processed independently to increase parallelism while lowering communication costs. Specifically, the semi-streaming model assumes that the input is a sequence of edges, which can be accessed only sequentially, as a stream. The main memory (can be randomly accessed) is assumed to be of size $O(n \text{ polylog } n)$. Usually, only one pass over the input stream is allowed, but some algorithms assume a small (usually constant or logarithmic) number of passes. The approach then divides the incoming stream of edges into substreams, processes each substream independently, and merges these results to form the final algorithm outcome.

2.6 FPGA Architecture and Terminology

Field-programmable gate arrays (FPGAs) are reconfigurable computing devices that contain a large number of programmable units that can be used to solve specific computational problems (see Figure 1). These logic units include lookup tables (LUTs) to implement combinatorial logic, flip-flops to implement registers, and a programmable interconnect. FPGAs often also provide more specialized units, such as block RAM (BRAM) for bulk storage, and DSP units for accelerating common arithmetic operations.

In contrast to application-specific integrated circuits (ASICs), which are only “configured” once during the manufacturing process, FPGAs can be reconfigured as often as needed. This allows improving and changing the architecture, applying bug-fixes, or using FPGAs to rapidly prototype hardware designs, which can later be manufactured as ASICs. FPGAs additionally allow reconfiguration on the fly to solve different tasks [58].

While CPUs and GPUs are instruction-driven, FPGA designs are usually data-driven, which means that an FPGA can process data directly without having to first decode instructions, and does not access a centralized register file or any cache hierarchy. This is usually more power efficient, as instruction decoding, register file lookup, and cache lookup account for the majority of power consumed on instruction-based architectures [61].

The reconfigurability comes with the cost of a lowered frequency, usually about 3-10 times lower than that of CPUs, and with less specialized components, e.g., floating point operations are often not native operations, and must be implemented with general purpose logic. Still, carefully engineered FPGA designs can outperform CPU implementations, by exploiting massive parallelism, typically in the form of deep pipelines. As long as there are no feedback data dependencies between iterations of an iterative algorithm, arbitrarily complex computations can be implemented as a pipeline that can produce one result per cycle. Application-specific instructions that are not a part of a CPU instruction set, e.g., a novel hash function, can be implemented on an FPGA to deliver a result every cycle, whereas a CPU implementation would potentially require many CPU instructions.

When FPGA performance models are discussed, we denote the bandwidth between the FPGA and DRAM as B_{DRAM}. The bandwidth of a single BRAM module is denoted as B_{BRAM}.

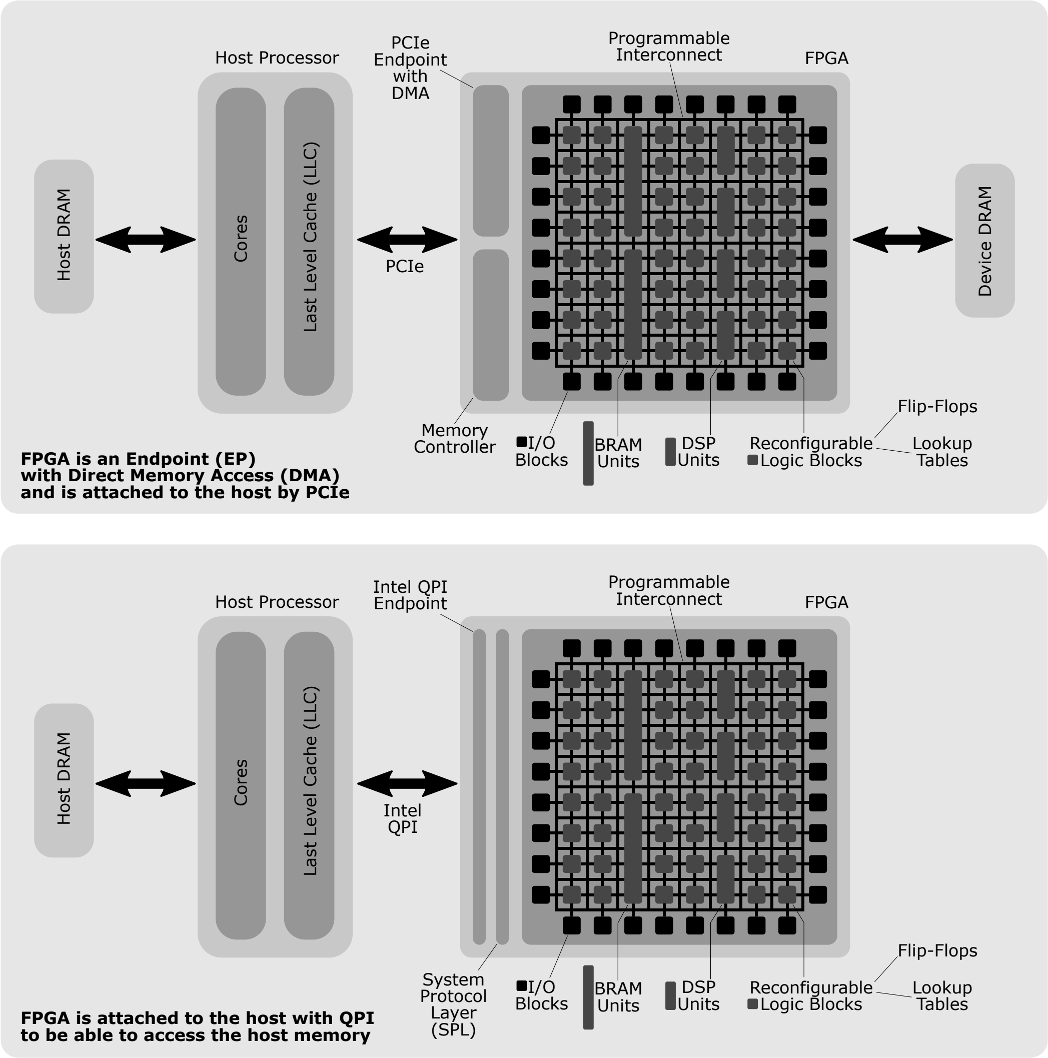
Host Processor
PCIe Endpoint with DMA
Programmable Interconnect
FPGA
Host DRAM
Cores
Last Level Cache (LLC)
PCIe
Device DRAM
Memory Controller
I/O Blocks
BRAM Units
DSP Units
Reconfigurable Logic Blocks
Flip-Flops
Lookup Tables
FPGA is an Endpoint (EP) with Direct Memory Access (DMA) and is attached to the host by PCIe
Host Processor
Intel QPI Endpoint
Programmable Interconnect
FPGA
Host DRAM
Cores
Last Level Cache (LLC)
Intel QPI
System Protocol Layer (SPL)
I/O Blocks
BRAM Units
DSP Units
Reconfigurable Logic Blocks
Flip-Flops
Lookup Tables
FPGA is attached to the host with QPI to be able to access the host memory

Fig. 1. Illustration of an FPGA and of two possible hybrid FPGA–CPU computation systems.

2.6.1 FPGA Programming Languages. The traditional way to program an FPGA is to use a hardware description language (HDL) such as Verilog, VHDL, or SystemC. These languages describe the cycle-by-cycle behavior of hardware at the register transfer level (RTL), which can then be synthesized to the underlying hardware resources and used to configure an FPGA. The low-level nature of these languages means that they lack many of the high-level concepts that are found in software programming languages. An alternative is to generate HDL using a high-level synthesis (HLS) tool, where the hardware description is derived from a more high-level imperative description. Many HLS tools exist [89], and most are based on C, C++ or OpenCL, using directives to aid the developer in expressing architectural features, such as pipelines, hardware replication, and fast memory allocation, in the generated RTL code. Other approaches include the Scala-based Chisel [5] that offers a more productive environment for RTL development, and the commercial MaxCompiler [99], which compiles to hardware from a dataflow-oriented Java-based language.

2.6.2 Coarsening of FPGA Features. Recent development has seen increasing specialization and diversification in FPGA architectures. Intel's Arria 10 and Stratix 10 families of FPGAs offer native 32-bit floating point (FP32) units, which greatly reduces the area usage of these operations (although 64-bit floating point is still costly), and simplifies certain patterns by supporting native accumulation of FP32 data. Stratix 10 FPGAs also expose "HyperFlex" [63] registers, a new family of dedicated routing registers aimed at improving frequency results, in order to narrow the gap to CPU and GPU clock rates. Xilinx UltraScale+ devices add a new class of on-chip RAM called UltraRAM [123], that expose access ports of similar width to traditional block RAM, but have larger capacity, allowing large amounts of memory to be stored on the chip without requiring as many individual RAM blocks to be combined. Finally, the Versal [124] family of Xilinx devices puts the FPGA on the same chip as an array of "AI engines", capable of performing more traditional SIMD-style arithmetic operations, adding to the compute potential in a hybrid ASIC/FPGA fashion. Common for these trends is a *coarsening* of components, sacrificing some flexibility for more raw performance and on-chip memory bandwidth for suitable workloads.

2.6.3 Integration with Hybrid Memory Cubes. A number of surveyed works relies on using the combination of FPGAs and the Hybrid Memory Cube (HMC) technology [98]. HMC dramatically improves the bandwidth of DRAM. An HMC unit consists of multiple DRAM dies that are connected using the *through-silicon-via* (TSV) technology. For example, it offers a 8–10× bandwidth improvement over DDR4 and is optimized for parallel random memory access [98]. Compared to traditional DRAM, HMC has much smaller page sizes (16B), which offers more performance for random memory accesses. Furthermore, HMC implements near memory computing in the form of locking and read-modify-write operations that are computed directly by the HMC unit instead of the CPU. It is even possible to atomically modify individual bits without having to first read the corresponding bytes.

3 TAXONOMY

In this section, we describe how we categorize the surveyed work. We summarize the most relevant papers in Table 3. We group separately generic graph processing frameworks and specific algorithm implementations. Each group is sorted chronologically. Selected columns in this table constitute criteria used to categorize the surveyed FPGA works, see also Figure 2.

The first such criterion is **generality**, i.e., whether a given FPGA scheme is focused on a particular graph problem or whether it constitutes a generic framework that facilitates implementing different graph algorithms. Another criterion is a used **graph programming paradigm, model, or technique**. We describe the general paradigms, models, and techniques in detail in § 2.5. However, certain techniques for graph processing are specific to FPGAs; we cover such techniques separately in § 4. Note that many implementations are not based on any particular paradigm or model and they do not use any particular general technique; we denote such works with "None".

We also distinguish between works that target a single FPGA and ones that scale to **multiple FPGAs**. Finally, we consider the used **programming language** and the **storage location** of the *whole* processed graph datasets. In the latter, "DRAM", "SRAM", or "HMC" indicates that the input dataset is located in DRAM, SRAM, or HMC, and it is streamed in and out of the FPGA during processing (i.e., only a part of the input dataset is stored in BRAM at a time). Contrarily, "BRAM" indicates that the whole dataset is assumed to be located in BRAM. "Hardwired" indicates that the input dataset is encoded in the FPGA reconfigurable logic.

4 FPGA-SPECIFIC GRAPH PROGRAMMING TECHNIQUES

We discuss separately graph programming techniques that are unique to FPGAs.

Reference (scheme name)	Venue	Generic Design[1]	Considered Problems[2] (§ 2.4)	Programming Model or Technique[4] (§ 2.5)	Used Language	Multi FPGAs[4]	Input Location[5]	$n^\dagger$	$m^\dagger$
Kapre [71] **(GraphStep)**	FCCM'06	👍	spreading activation* [82]	BSP	unsp.	👍	BRAM	220k	550k
Weisz [92] **(GraphGen)**	FCCM'14	👍	TRW-S*, CNN* [112]	Vertex-Centric	unsp.	👎	DRAM	110k	221k
Kapre [70] **(GraphSoC)**	ASAP'15	👍	SpMV	Vertex-Centric, BSP	C++ (HLS)	👍	BRAM	17k	126k
Dai [40] **(FPGP)**	FPGA'16	👍	BFS	None	unsp.	👍	DRAM	41.6M	1.4B
Oguntebi [93] **(GraphOps)**	FPGA'16	👍	BFS, SpMV, PR, Vertex Cover	None	MaxJ (HLS)	👎	BRAM	16M	128M
Zhou [134]	FCCM'16	👍	SSSP, WCC, MST	Edge-Centric	unsp.	👎	DRAM	4.7M	65.8M
Engelhardt [49] **(GraVF)**	FPL'16	👍	BFS, PR, SSSP, CC	Vertex-Centric	Migen (HLS)	👎	BRAM	128k	512k
Dai [41] **(ForeGraph)**	FPGA'17	👍	PR, BFS, WCC	None	unsp.	👍	DRAM	41.6M	1.4B
Zhou [136]	SBAC-PAD'17	👍	BFS, SSSP	Hybrid (Vertex- and Edge-Centric)	unsp.	👎	DRAM	10M	160M
Ma [85]	FPGA'17	👍	BFS, SSSP, CC, TC, BC	Transactional Memory [16, 59]	System-Verilog	👍	DRAM	24M	58M
Lee [79] **(ExtraV)**	FPGA'17	👍	BFS, PR, CC, AT* [60]	Graph Virtualization	C++ (HLS)	👎	DRAM	124M	1.8B
Zhou [135]	CF'18	👍	SpMV, PR	Edge-Centric, GAS	unsp.	👎	DRAM	41.6M	1.4B
Yang [125]	report (2018)	👍	BFS, PR, WCC	None	OpenCL	👎		4.85M	69M
Yao [127]	report (2018)	👍	BFS, PR, WCC	None	unsp.	👎	BRAM	4.85M	69M
Babb [4]	report (1996)	👎	SSSP	None	Verilog	👍	Hardwired	512	2051
Dandalis [43]	report (1999)	👎	SSSP	None	unsp.	👍	Hardwired	2048	32k
Tommiska [116]	report (2001)	👎	SSSP	None	VHDL	👎	BRAM	64	4096
Mencer [87]	FPL'02	👎	Reachability, SSSP	None	PAM- -Blox II	👎	Hardwired (3-state buffers)	88	7744
Bondhugula [27]	IPDPS'06	👎	APSP	Dynamic Program.	unsp.	👎	DRAM	unsp.	
Sridharan[110]	TENCON'09	👎	SSSP	None	VHDL	👎	BRAM	64	88
Wang [121]	ICFTP'10	👎	BFS	None	SystemC		DRAM	65.5k	1M
Betkaoui [21]	FTP'11	👎	GC	Vertex-Centric	Verilog	👍	DRAM	300k	3M
Jagadeesh [65]	report (2011)	👎	SSSP	None	VHDL	👎	Hardwired	128	466
Betkaoui [22]	FPL'12	👎	APSP	Vertex-Centric	Verilog	👍	≈ DRAM	38k	72M
Betkaoui[23]	ASAP'12	👎	BFS	Vertex-Centric	Verilog	👍	DRAM	16.8M	1.1B
Attia [2] **(CyGraph)**	IPDPS'14	👎	BFS	Vertex-Centric	VHDL	👍	DRAM	8.4M	536M
Ni [91]	report (2014)	👎	BFS	None	Verilog	👎	DRAM, SRAM	16M	512M
Zhou [132]	IPDPS'15	👎	SSSP	None	unsp.	👎	DRAM	1M	unsp.
Zhou [133]	ReConFig'15	👎	PR	Edge-Centric	unsp.	👎	DRAM	2.4M	5M
Umuroglu [117]	FPL'15	👎	BFS	None	Chisel		≈ DRAM	2.1M	65M
Lei [80]	report (2016)	👎	SSSP	None	unsp.	👎	DRAM	23.9M	58.2M
Zhang [129]	FPGA'17	👎	BFS	MapReduce	unsp.	👎	HMC	33.6M	536.9M
Zhang [130]	FPGA'18	👎	BFS	None	unsp.		HMC		
Kohram [76]	FPGA'18	👎	BFS	None	unsp.	👎	HMC		
Besta [13]	FPGA'19	👎	MM	Substream-Centric	Verilog	👎	DRAM	4.8M	117M

Table 3. Summary of the features of selected works sorted by publication date. [1]**Generic Design**: this criterion indicates whether a given scheme provides a graph processing framework that supports more than one graph algorithm (👍) or whether it focuses on concrete graph algorithm(s) (👎). [2]**Considered Problems**: this column lists graph problems (or algorithms) that are explicitly considered in a given work; they are all explained in § 2.4. [3]**Used Programming Paradigm, Model, or Technique**: this column specifies programming paradigms and models used in each work; they are all discussed in § 2.5 and § 4. "None" indicates that a given scheme does not use any particular general programming model or paradigm or technique. [4]**Multi FPGAs**: this criterion indicates whether a given scheme scales to multiple FPGAs (👍) or not (👎). [5]**Input Location**: this column indicates the location of the *whole* input graph dataset. "DRAM", "SRAM", or "HMC" indicates that it is located in DRAM, SRAM, or HMC, and it is streamed in and out of the FPGA during processing (i.e., only a part of the input dataset is stored in BRAM at a time). Contrarily, "BRAM" indicates that the whole dataset is assumed to be located in BRAM. "Hardwired" indicates that the input dataset is encoded in the FPGA reconfigurable logic. $n^\dagger$, $m^\dagger$: these two columns contain the numbers of vertices and edges used in the largest graphs considered in respective works. In any of the columns, "unsp." indicates that a given value is not specified.

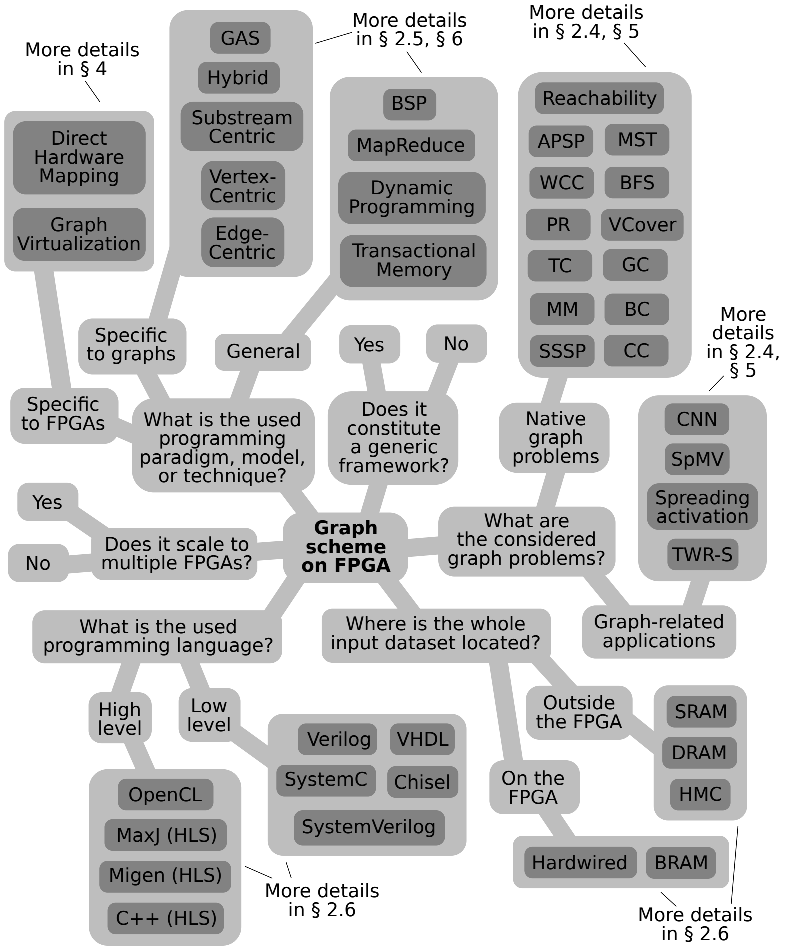

Fig. 2. The categorization of the considered domains of graph processing on FPGAs. All the categories are gathered in a form of a tree. The tree root in the centre represents all graph processing schemes implemented on FPGAs. The children of the tree root correspond to various general criteria that can be used to categorize FPGA graph processing schemes, e.g., whether or not a given FPGA scheme constitutes a generic graph processing framework or whether it is an implementation of a particular algorithm.

4.1 Direct Hardware Mapping

One of the earliest approaches in FPGA graph processing was to to map a graph completely onto the FPGA using logical units and wires to physically represent every single vertex and edge. The obvious limitation of all these approaches is that the size of input graphs is limited by the amount of the FPGA reconfigurable resources.

In 1996, Babb et al. [4] introduced *dynamic computation structures*, a technique that compiles a given graph problem to Verilog code and then maps it to an FPGA. Each vertex of the input graph is physically represented on the FPGA. This system was then able to solve the Bellman-Ford algorithm to find all shortest paths from a specified source vertex. The approach requires to reconfigure the FPGA whenever the input graph changes.

Later, Mencer and Huelsbergen [62, 87] presented a more flexible design where a graph topology can be changed on the FPGA using tri-state buffers to represent entries of the adjacency matrix.

This, however, still requires $O(n)$ space on the chip. Simple graph problems, such as reachability of some vertex b from another vertex a, can be solved by propagating a signal through a and checking whether it reaches b. Other problems, such as SSSP or CC, can be solved in a similar manner with slightly more complex circuits.

Finally, Dandalis et al. [43] represent vertices as processing elements but store edges in the main memory so that they can be loaded dynamically to represent same-size graphs with different edge sets without having to reconfigure the FPGA. Jagadeesh et al. [65] implement a similar design but propose changes to the design of the processing elements to reduce the number of cycles per iteration. Both approaches can compute arbitrary graphs as long as the FPGA has been configured with enough processing elements.

4.2 Graph Virtualization

Graph Virtualization is a technique proposed by Lee et al. [79], where the program running on the host processor is provided with the illusion that the input graph resides in the main memory and is stored using some simple format such as Adjacency Array, while in reality the graph is stored in a more complex, possibly multi-level and compressed form on a storage managed by an accelerator such as FPGA. The motivation is to use the accelerator to offload tasks related with graph decompression and filtering from the host processor. Additionally, graph virtualization enables the accelerator to apply various optimizations and functionalities to the data, for example multi-versioning, without affecting processor functions or programmability. This technique can be used together with any accelerator, not only an FPGA. However, as the proposed design is implemented on an FPGA system, we include it in this survey.

5 SPECIFIC GRAPH ALGORITHMS ON FPGA

We now discuss selected hardware implementations of individual graph algorithms. Such schemes form one significant part of research works dedicated to graph processing on FPGAs.

5.1 BFS

Works on BFS constitute the largest fraction of graph processing schemes on FPGAs. We now describe selected works, focusing on explaining key ideas and summarizing important outcomes.

5.1.1 Using Hybrid CPU-FPGA Processing. **One idea** for efficient BFS traversals is to combine the different compute characteristics of the FPGA and the CPU. Specifically, Umuroglu et al. [117] present an efficient BFS implementation on a CPU-FPGA hybrid system. The paper focuses especially on small-world graphs [122] which have a very low diameter. In such graphs, the size of the frontier exhibits a specific pattern throughout the BFS algorithm. The frontier remains fairly small in the first several iterations, but then grows quickly in the next steps, to become small again at the end of the traversal [7]. As the authors estimate, the frontier contains on average about 75% of all the vertices in the considered graphs during the fourth iteration. The authors use this observation while splitting the work between the CPU and the FPGA: small frontiers do not offer much parallelism but can be efficiently computed on the CPU while large frontiers can be parallelized on the FPGA. The implementation described in the paper thus *computes the first and last steps on the CPU and uses the FPGA only to compute the steps with the large frontiers*. The implemented BFS is expressed using the language of linear algebra (i.e., frontier expansion is implemented as multiplying the adjacency matrix and a vector that stores the current frontier, see § 2.2 for more details).

Another idea in the work by Umuroglu et al. [117] is to *read the whole frontier array sequentially*. In BFS, an important operation is to verify whether a certain vertex is in the frontier. Instead of querying only for those vertices, the authors propose to read the whole frontier array into BRAM and thus remove the need for random memory accesses. Because of the small-world graph assumption, we know that the frontiers will contain a significant amount of vertices.

Various Insights The authors name three important ways for using a large portion of the available DRAM bandwidth: A high rate of requests to make the latency of individual requests less significant, using large bursts of requests, and a sequential access pattern to increase the number of row buffer hits [117]. The authors argue that it is more efficient to treat sparse bit vectors as dense (see § 2.3.2 for an explanation of sparse and dense structures) and read them sequentially instead of accessing (most often randomly) only the parts that are know to contain the required data.

Remarks on Evaluation As the vector that is the result of the MV multiplication (i.e., the result of the frontier expansion) is stored in BRAM, the BRAM poses a hard limit on the size of graphs that can be computed using this approach. In fact, the authors were unable to use graphs with more than 2 million nodes and 65 million edges due to the limited on-chip BRAM size. The size of the result vector is n words; the paper reports that 82% of the BRAM is used for the result vector.

5.1.2 Using Hybrid Memory Cubes (HMC). The **key idea** due to Zhang et al. [129] is to use Hybrid Memory Cubes (HMC) to implement an efficient BFS implementation on FPGAs (we discuss HMC in more detail in § 2.6.3 and § 7.1.1). The authors build an analytical performance model of the HMC memory access latencies that allows them to analyze performance bottlenecks. HMC allows to select the size of the transferred data payload to be anything between 16 bytes and 128 bytes. The analysis shows that, depending on data locality [114], either small or large read granularity is more efficient. For example, reading a list of neighbors in a BFS traversal has better data locality than updating the parent of a single vertex. Thus, **the main insight** is that *different payload sizes for accessing different data structures in BFS can be used to optimize performance.*

In contrast to Umuroglu et al. [117], the authors do not use a hybrid FPGA-CPU setting and run the whole BFS on the FPGA, including iterations with smaller frontiers. In such iterations, the bitmaps are very sparse and entries should be accessed without having to iterate the whole array. The **proposed solution** is to use a level of indirection in a form of a second smaller bitmap where each entry represents an OR reduction of k entries in the larger bitmap. This second bitmap is conveniently small enough such that it can be stored in BRAM (k can be set to be arbitrarily large).

Implementation Details The authors implement BFS in a framework based on the MapReduce paradigm. Mapper modules read the current frontier of the BFS and extract the adjacency lists of those vertices. Reducer modules then update the parents of the newly visited vertices and store them in the new frontier. The frontiers are stored as bitmaps in the main memory.

Remarks on Evaluation The described work enables processing arbitrarily large graphs. However, the used HMC module had a limited capacity of 4GB, limiting the size of used graphs.

5.1.3 Customizing Graph Representations. The **main idea** by Attia et al. [2] is to *use a custom graph representation to reduce the memory bandwidth usage* in their BFS design [2]. The representation is based on the CSR format. **One observation** the authors make is that the information stored in the *row* array is only needed until a vertex has been visited for the first time. Thus, the authors propose to additionally store both the visited flag as well as the distance of a visited vertex in the *row* array. In their design, the least significant bit of an entry $row[i]$ indicates whether the vertex i has been visited. If i has been visited, the rest of the entry stores the distance to the source vertex. Otherwise, $row[i]$ stores the offset of its neighborhood in *col* as well as the number of neighbors. In **another optimization**, instead of pushing vertex to the frontier queue [19], the authors propose to push the value $row[i]$ since only this value is needed in the subsequent iteration. It should be noted that this version of BFS only outputs the distance of each vertex from the source but not their parents. All these optimizations improve spatial locality and reduce the amount of data that has to be read from DRAM.

5.1.4 Others. Other implementations of BFS on FPGAs have been proposed [23, 91, 121]. They come with various optimizations and schemes, for example Wang et al. [121] deliver a message-passing multi-softcore architecture with a variety of optimizations (dual-threaded processing, using bitmaps for output, and pipelining), Ni et al. [91] combine using SRAM chips and DRAM modules with optimized pipelining, and Betkaoui et al. [23] decouple computation and communication while maintaining a large number of memory requests in flight at any given time, to make use of the parallel memory subsystem. Finally, FPGAs are used to accelerate a distributed BFS traversal [12]. Specifically, they implement virtual-to-physical address mapping in a cluster of GPUs based on the Remote Direct Memory Access [54] technology.

5.2 SSSP

Tommiska and Skyttä [116] implement Dijkstra's SSSP algorithm on the FPGA using VHDL. The input graph, represented as an adjacency matrix, and the result data are stored on the internal storage of the FPGA, which drastically limits the size of graphs that can be computed. **An interesting feature** of this design, which is also found in some later works [132], is *using the comparator block which is able to process multiple edges in parallel to find the one with the smallest distance.*

The **key idea** in the work by Sridharan et al. [110] is to use a linear-programming based solution for certain graph problems on the FPGA, *which forms the only work on solving graph problems on FPGAs using linear programming*. However, their design also limits the size of graphs that can be processed; the authors could only test their implementation on a graph with 44 vertices.

Zhou et al. [132] developed an FPGA graph accelerator for solving SSSP using the Bellman-Ford algorithm. The first **key feature** of this architecture is, unlike in previous approaches, *storing the input graph in DRAM*. The **second key feature** is that the architecture is *fully pipelined* with the ability to process multiple edges concurrently; p edges are read in parallel and the corresponding data is driven through multiple architecture blocks, each of which can process p instances at once. The first stage is the sorting block, which makes sure that if more than one of the p edges target the same destination vertex, only the one with the minimal value is used to produce an update. Next, the memory read block fetches the weights of the targeted destination vertices from DRAM. These values are then passed to the computation block, which compares the proposed updated weight (i.e, the new distance) and the current weight of the destination vertices and decides whether to perform the update. Finally, the memory write block stores the updated values to DRAM. It also counts the number of successful updates during one whole iteration and makes sure to terminate the algorithm if no more updates have occurred. Finally, the **third key part** of the architecture is a *data-forwarding technique* to prevent race conditions that can occur when two edges with the same destination are processed in consecutive cycles. The authors propose to sort the edges by their destination vertices since all updates to the same vertex would follow consecutively and the minimal value could be found easily in the sorting block, thus reducing the number of updates that have to be written to DRAM. The performance limitation in the design is the bandwidth and the high number of random memory accesses that have to be made.

A slightly different approach is taken by Lei at al. [80], using a version of the so called "eager" Dijkstra algorithm [48] instead of Bellman-Ford. The algorithm works with a priority queue and can be parallelized by removing multiple vertices from the queue in each iteration. A similar priority queue such as in work by Sun and Srikanthan [111], called systolic array priority queue (SAPQ), is implemented on the FPGA. However, to support the processing of large graphs, elements that overflow the queue are moved to DRAM. The benefit of this queue implementation is that enqueuing an element and dequeuing the minimum element are performed in constant time. In each iteration, λ vertices are picked from the priority queue and processed in one of λ processing elements in parallel. Similarly to Zhou et al's approach [132], this design suffers from the large amount of off-chip memory accesses. The authors derive a performance model based on the total number of memory accesses. Namely, in each iteration, $(512 + 64d)\lambda$ memory accesses are required for reading the graph data and $1024d\lambda$ memory accesses are needed to store the results (d is the average degree in the input graph). It should be noted that one iteration in this case corresponds to reading λ vertices from the priority queue and *not* traversing the whole data structure. Since different DRAM modules are used to store the graph data, the priority queue, and the results, the performance bottleneck depends on the module which is used most. Thus, assuming that the bandwidth is the limiting factor, the total processing time is $1024dL\lambda \backslash B_{DRAM}$ for L iterations.

5.3 APSP

Bondhugula et al. [26, 27] solve the APSP using a *parallel version of the Floyd-Warshall algorithm*. The input graph is stored in off-chip memory and only the required *slices* (i.e., parts of the adjacency matrix) are streamed to the FPGA BRAM modules. The tiling scheme by Venkataraman, et al. [120] is used to partition the graph into multiple tiles of size $B \times B$, where B is a scheme parameter. The architecture contains B pipelined processing elements, with each one being able to process up to l values from a row per cycle. These values are passed from one processing element to another until they are finally stored back to the memory by the last processing element. The tile size is limited by the resources available on the FPGA.

Betkaoui et al. [22] solve APSP for unweighted graphs by *running BFS from each vertex*. The design contains multiple processing units, each of which is able to run one BFS instance in parallel with others. Each processing unit has its own local memory but is also connected to a shared memory system where the graph representation is stored. The **key idea** (for optimizing memory accesses) is to *use the on-chip memory for data requiring irregular patterns* (e.g., the current status of vertices) and *the off-chip memory for data that can be accessed sequentially* (e.g., the edges of the graph). For this, they design a bitmap scheme that can efficiently store the visit status of vertices (i.e., whether each vertex was visited) and emulates queues used in the algorithm. Since each of the three bitmaps contain at most n bits, the authors make the assumption that these bitmaps can fit in the on-chip memory. However, since each of the p processing element stores three bitmaps of size n, the total space requirement is $3pn$. Thus, this assumption does not hold for larger graphs.

5.4 PageRank

The only work which describes an FPGA accelerator specifically tailored for PageRank was done by Zhou et al. [133]. The authors use a two-phase algorithm (based on the GAS model, see § 2.5) similar to their general approaches in FPGA graph processing frameworks [134, 135], which we discuss in more detail in § 6.9. During each iteration, the scatter phase processes all edges, generates updates, and stores the updates in off-chip memory. The gather phase reduces the updates and applies them to the array with ranks. The **main focus** of the approach is to reduce the number of random memory accesses during the algorithm. This is achieved by reading the edges and the vertex properties in the scatter phase sequentially, reading the updates and writing the updated PageRank values in the gather phase sequentially, and reducing the number of random memory writes in the scatter phase by leveraging the order in which the edges are being processed to merge some of the updates before writing them to DRAM.

5.5 Application-Driven Graph Problems

Finally, there have been attempts at implementing various specific graph-related applications on FPGAs. An example is DNA Assembly based on De Bruijn graphs [38] using FPGAs [100, 119].

6 GENERIC GRAPH PROCESSING FRAMEWORKS ON FPGA

A lot of effort was placed in designing generic frameworks for facilitating the implementation of graph algorithms on the FPGA. The goal of such a framework is to (1) allow the user to easily and rapidly develop their graph algorithms, using some proposed graph programming abstraction or programming model, and (2) generate an FPGA implementation from such user code. This yields **two major advantages** to custom FPGA designs: *portability* and *programmability*. First, the user algorithm is not tied to one specific FPGA device but can be compiled to various FPGA devices. Second, the framework is usually built around a specific programming model and any graph algorithm that is compatible with this model can be implemented without having to change the core of the framework. In the following, we present selected FPGA graph frameworks chronologically.

6.1 [2006] GraphStep [71]

GraphStep by Kapre et al. [71] is *one of the first approaches to design a generic FPGA architecture* that is not specific to only one graph algorithm. In GraphStep, every vertex is abstracted to be an *actor* that can send messages and invoke selected pre-programmed methods on its neighboring vertices. The computation associated with each vertex is divided into several steps; these steps are synchronized across all the vertices. First, each vertex receives input messages coming from its neighbors. Second, a vertex awaits the synchronization barrier. Third, a vertex updates its state. Finally, a vertex sends messages to its neighbors.

The used model resembles BSP, with a difference that communicating objects are graph vertices and communication can only occur between adjacent vertices. It also resembles the vertex-centric model introduced later in the Pregel paper [86]. Such a model can be realized in several ways, among others, being hardware mapped like the early works of Bobb and Mencer (see § 4.1) where every vertex is physically represented on the FPGA, or by having multiple processing elements on the FPGA, with each element taking care of a certain range of vertices.

High-degree vertices are decomposed into multiple vertices with smaller degrees to avoid bottlenecks. Additionally, a selected graph partitioning scheme that minimizes sizes of cuts [31] is used to maximize the locality of vertices stored on the same memory block.

Multiple processing engines are distributed on the FPGA, each of them being able to process one edge per cycle using a pipelined architecture. One such processing engine needs 320 Virtex-2 slices, and thus a XC2V6000 FPGA can hold around 32 such processing engines, connected with a Butterfly topology [15, 42]. The input graph data resides in BRAM instead of external memory since the FPGA on-chip bandwidth is one to two orders of magnitude higher than the off-chip bandwidth to and from DRAM. For larger graphs that do not fit the on-chip memory, they propose to distribute the data among multiple FPGAs. This way the bandwidth would scale with the size of the data. 64K edges can be stored on the BRAM of one XC2V6000 FPGA and hence 16 such FPGAs are required to store and process the whole ConceptNet. 48 additional FPGAs are used to route the communication. In total, 64 FPGA are required to solve a graph problem with 550,000 edges; only 25% of the incorporated FPGAs perform the actual computation.

GraphStep implements the spreading activation of the ConceptNet knowledge base [82], as an example application of their framework.

The authors compare their approach with a sequential implementation on a 3.4GHz Pentium-4 Xeon and obtain speedups of 10–20×.

6.2 [2011] Framework by Betkaoui et al. [21]

The next FPGA graph processing framework is built upon a vertex-centric approach. The authors use the graphlet counting problem as a case study. The leading design choice behind their work is to maximize the utilization of the available parallelism by implementing a large number of processing units on the FPGA. All processing units have a direct access to the shared off-chip memory, which consists of multiple memory banks, to increase throughput and allow memory operations to be performed in parallel. This approach can even be scaled to multiple FPGA units.

On the other hand, no form of caching is used and the data is not prefetched to BRAM. This leads to large latencies when accessing data from DRAM. These latencies are alleviated by and their cost is outweighed with the large amount of used parallelism.

The authors implement their design on the Convey HC-1, a hybrid system, featuring an Intel Xeon dual-core host CPU and four programmable Xilinx Virtex 5 FPGAs [3]. The FPGAs are connected to 16 DDR2 memory channels over a crossbar. There are 8 independent memory controllers with connections to each device, each one implemented on a V5LX110 FPGA. The memory itself consists of 16 special Scatter-Gather-DIMMs, totaling 8GB of memory and a peak bandwidth of 80 GB/s. These memory units offer very good random memory access performance.

6.3 [2014] GraphGen [92]

GraphGen is a framework that can compile graph algorithms, which are expressed according to the vertex-centric programming paradigm, onto a target FPGA. The main focus is on enabling developers to program graph applications on hardware accelerators without requiring any knowledge about the specific platform.

In contrast to previous graph frameworks, GraphGen allows to store graph data in DRAM, thus enabling the processing of much larger graphs that would otherwise not fit in BRAM. For this, the graph is partitioned (according to the 1D graph partitioning scheme [25]) into smaller subgraphs, each of which can fit in BRAM. The compiler generates an FPGA program for each subgraph. The compilation procedure is thus dependent on both the graph algorithm and the graph data, which means that the FPGA kernel needs to be recompiled whenever the graph data changes or the user wishes to compute the same algorithm on another data set.

The computation revolves around an update function that is executed on every vertex of the graph. This function is specified as a series of custom graph instructions. The user needs to define these instructions and provide appropriate RTL implementations for them. They are then integrated in the graph processor and used by the compiler to construct the update function.

The architecture is based on the CoRAM abstraction [36], a memory abstraction model for FPGAs. The model specifies buffers which are used for the transfer of data between the FPGA and other devices. The data transport between CoRAM buffers and external DRAM is then specified using a C-like language and managed by so called control threads.

The framework applies several additional optimizations to reduce the bandwidth usage. First, double buffering is used to prevent stalls while waiting for new data. Second, edges are sorted according to the order in which they are accessed. Thus, memory access requests can be coalesced to reduce the number of independent memory accesses. Third, pipelining is used in the processing elements to achieve high clock rates. The GraphGen compiler schedules instruction such that data hazards are avoided. Moreover, the incorporated programming model allows multiple edges to be read the same cycle. Finally, custom SIMD (single-instruction multiple-data) instructions are supported to allow processing data elements together.

The authors provide two case studies for GraphGen, stereo matching using the Tree-Reweighted Message Passing algorithm (TRW-S) [112] and handwriting recognition using a Convolutional

Neural Network (CNN) [10, 11, 95]. When compared to software implementations, they achieve a speedup of 14.6× and 2.9× in FPGA implementations compiled by GraphGen for TRW-S and CNN, respectively. Both applications come from the computer vision domain and work on 2D images that are modeled as 2D grids. The advantage of such graph data sets is their regularity: every pixel is only adjacent to the eight other pixels around it. Thus, it is relatively straightforward to split the graph into subgraphs while still maintaining high locality.

6.4 [2015] GraphSoC [70, 72]

GraphSoc [70, 72] is an FPGA graph processing framework that comes with an instruction set architecture for performing algorithms on sparse graphs. The input graph is stored entirely on the FPGA BRAMs using the CSR format. The FPGA contains multiple processing elements which are responsible for processing subsets of the graph and are connected with a Network-on-Chip (NOC) [14]. Each processing element is implemented as a 3-stage pipeline in which an instruction is fetched, decoded and then executed. Instead of having general-purpose registers, the architecture has special dedicated registers for vertices, edges, and instructions. There are also registers dedicated to counting loops that allow zero-overhead looping.

Graph algorithms are expressed using the BSP model. There are four customizable instructions, *send, receive, accum*, and *update*, that can be specified to implement the needed functionality of a wide range of algorithms.

The authors implement GraphSoc in high-level C++ and use High-Level Synthesis (HLS) to generate RTL. They also use the PaToH partitioner [32] to minimize the number of inter-partition connections when distributing the vertices among the processing elements.

If the data set is too large to fit in the BRAM, multiple FPGA devices are used. For this case, the authors describe the design of a Beowulf cluster of 32 Zync Z7020 boards for the computation of large sparse graphs [72].

6.5 [2016] FPGP [40]

FPGP [40] is a framework that uses *intervals* and *shards* to improve locality of processed graphs and to enable the processing of arbitrarily large data sets by storing only the currently relevant graph partitions in BRAM. Vertices are partitioned into P intervals based on their vertex IDs. Each interval has a corresponding shard (also called a *bucket*) that contains all edges that point towards one of the vertices in this interval. A shard is further divided into sub-shards, based on the source vertex of the edges. Edges inside each sub-shard are then sorted by their destination vertex. This partitioning scheme is chosen so that the order of updating vertices corresponds to the intervals. This makes it possible to have multiple processing units on an FPGA or even multiple FPGAs compute different intervals in parallel. While the edges are streamed from memory shard by shard, the vertices are computed interval by interval. This partitioning scheme is further described in a paper describing the NXgraph processing system, developed by the same authors [34] (NXgraph is a graph processing system designed for efficient graph computation on single servers).

The FPGP framework enables the user to express different graph algorithms by only implementing the *kernel function*, which is executed for each edge by dedicated processing units. The kernel function takes as input the properties of the source and destination vertices of an edge and computes the new property of the destination vertex. During the computation of an interval the updated vertices are stored locally on in BRAM and then later written to the shared memory together with other vertices in the interval.

The general scheme behind the framework works as follows. For every interval I_j, the framework processes all edges in the corresponding consecutively stored sub-shards $S_{i,j}$, where i denotes the interval to which the sources vertices of the edges belong. For every sub-shard $S_{i,j}$, the properties of the vertices in interval i are loaded to the FPGA and the kernel function is performed for every edge in the sub-shard. After the whole shard is processed, the updated interval I_j is stored back to the shared memory and the next interval is loaded to BRAM. Depending on the algorithm, different numbers of iterations are needed. Thus, the entire source property array is being loaded P times. Since P can be expressed as $\frac{n}{M_{BRAM}}$, M_{BRAM} being the available BRAM memory on the chip, the total communication cost for reading vertex properties is in $O(n^2)$.

The authors build a model to describe the performance of the framework. They derive formulas that describe the effect of several factors on the performance, including limited bandwidth. FPGP uses two different memory systems: local DRAM banks that store the edges and a shared off-chip DRAM that stores the vertex properties. The communication for loading the vertex properties used for one iteration of an algorithm is given as $\frac{Pnb_v}{N_{chip}B_{share}}$ where b_v is the number of bits per vertex property, N_{chip} is the number of FPGAs, and B_{share} is the bandwidth to the shared memory.

The authors evaluate the performance of FPGP with a BFS implementation on both the Twitter graph and the Yahoo web graph and compare the results with frameworks built for the CPU to show that their FPGA implementation can outperform CPU implementations.

6.6 [2016] GraphOps [93]

In GraphOps, the authors provide a set of building blocks from which graph algorithms can be composed. These blocks are then placed on the FPGA and connected together. Example blocks include ForAllPropRdr (it fetches the vertex properties of all the neighbors of a vertex from memory to the FPGA), NbrPropRed (it performs a reduction on the vertex properties of a neighborhood. The reduction can be customized depending on the algorithm), and ElemUpdate (it writes the updated value of a vertex back to memory). The authors show how to use these three blocks and some utility and control-flow blocks to implement PageRank and other algorithms.

For example, during one iteration, the PageRank algorithm would execute the three operations for every vertex in the graph. The first block reads the vertex properties from memory, then this data is passed to the second block to perform the reduction. Finally, the last block receives the updated value and the address of the vertex and updates the memory location accordingly.

The authors implement a range of different graph algorithms to illustrate the flexibility of GraphOps. In their evaluation, they conclude that the bottleneck for all implementations is the memory bandwidth between FPGA and DRAM. Parallelism is only available in the pipelined architecture of the blocks and inside individual blocks.

GraphOps does not describe how to implement multiple parallel pipelines. Moreover, data that is fetched from memory is only used to update a single vertex, but never reused among multiple vertices. Such reuse could potentially lower the communication costs and thus improve the performance. Since multiple blocks can issue memory read or write requests simultaneously, the memory channel needs to serve different memory interfaces. This, as well as the fact that the neighborhoods are not accessed in the order in which they are stored in memory, leads to a large number of random memory requests which place an even harder burden on the memory bandwidth. The authors conclude that performance of GraphOps corresponds to only 1/6 of the theoretically available throughput.

To alleviate the memory bandwidth bottleneck between FPGA and DRAM, GraphOps uses a locality-optimized graph representation which trades redundancy for better locality and thus overall less communication (see also § 7.1.3).

Instead of storing the properties of vertices as an array of size n, they propose to replicate the property value for every single incoming edge. The "*locally optimized property array*" is thus an array of size m where every entry i represents the property of the destination vertex of the edge that is stored in entry i of the adjacency array. This design has two major benefits. First, it removes one level of indirection when accessing the property of an adjacent vertex. In a traditional adjacency array, one first needs to access the index of the adjacent neighbor and then use it to access the property array. In GraphOps, one can directly access the property of a neighbor without having to know its ID. Second, the properties of all neighboring vertices are now stored consecutively in memory. This improves spatial locality as there is now only one random memory access needed to access all properties of adjacent vertices. On the other hand, updates have to be propagated to the replicated entries.

6.7 [2016] GraVF [49]

GraVF [49] offers flexibility and ease of programming by only specifying two functions used as building blocks for graph algorithms. GraVF is based on the BSP model. In each superstep, the Apply function defines how a vertex property is updated based on the incoming messages and the Scatter function defines which messages are sent to the vertex neighbors. Vertices are divided among the processing elements on the FPGA. The synchronization step is implemented with a barrier algorithm similar to that of Wang et al. [121]. All vertices and edges are stored on the FPGA, limiting the size of graphs that can be processed by the framework. The largest graph that the authors were able to test on a Xilinx Virtex 7 had 128k vertices and 512k edges.

6.8 [2017] ForeGraph [41]

In their second FPGA graph framework, ForeGraph [41], Dai and others focus on efficient scaling to multiple FPGAs. The driving observation in their work is that multiple FPGAs enable more on-chip storage, more parallelism, and more higher bandwidth. The solution behind ForeGraph is to use a separate off-chip DRAM module for each FPGA. Similarly to the strategy in FPGP, the input graph is partitioned into P intervals and shards. However, now every partition corresponds to one of P FPGA devices. Each FPGA is thus processing the edges that lead to its share of the graph but needs to access the source vertex properties from other FPGA devices. Since the vertex properties of a whole partition may not fit in the BRAM of its FPGA, the partitions are further divided into Q sub-blocks in the same way.

Both FPGP and ForeGraph are based on the assumption that the updated vertex properties can be computed in intervals and that after all edges leading to some interval i have been processed, the vertex properties of the interval would not have to be updated again until the next iteration of the algorithm. However, there exist graph algorithms that make an update of some vertex w when traversing an edge (u, v) where neither $u = w$ nor $v = w$ and also w does not lie in the same interval as v. An example is the Shiloach-Vishkin algorithm where previous components have to be accessed and reassigned [107]. This and similar cases are handled by a graph processing paradigm where all updates are first collected in some temporary memory and then merged together. An example would be edge-centric frameworks from Zhou and Prasanna [135].

6.9 [2016 – 2018] Frameworks by Zhou and Prasanna [134–136]

Throughout multiple papers Zhou and Prasanna follow the edge-centric paradigm where, instead of accessing neighborhoods of vertices, all edges are streamed to the FPGA and processed in the order in which they arrive, similarly to the well-known system X-Stream [102]. This results in sequential reads to the graph data structure and leads to better bandwidth usage. As the edges are sorted by their source vertex, the required vertex properties can also be read sequentially by buffering small intervals of vertex properties in BRAM.

The framework follows the Gather-Scatter programming model. In the first (scatter) phase, all edges are processed and the resulting updates are stored into buckets, sorted by the interval in which the updated vertex is located. Because of the limited size of BRAM modules, these buckets reside in DRAM and thus random memory writes are required. In the second (gather) phase, the updates are again streamed to the FPGA, merged together using an algorithm-specific reduction function and then written to the vertex property array in DRAM.

Random writes to DRAM often lead to row-conflicts, taking place when two consecutive memory accesses target different rows in DRAM [64]. To reduce the number of row conflicts in the scatter phase, the authors propose to sort the edges inside each partition by their destination vertex. Thus, updates to the same row in DRAM happen consecutively. Given that there are P partitions, one can show that there will be at most $O(P^2)$ row-conflicts. Additionally, since now updates that target the same destination vertex are processed consecutively, these updates can be reduced on the FPGA before being stored to the update bucket, reducing communication.

To lower power consumption, the authors temporarily deactivate BRAM modules through the 'enable' port [1] when they are not needed. Parallelism is implemented in the form of concurrent pipelines. Each pipeline can fetch and process a new edge in each cycle. There are three stages in each pipeline which perform slightly different for the scatter and the gather phase. In the scatter phase, the pipeline first reads the property of the source vertex from BRAM, then computes the update and finally writes the update to DRAM. In the gather phase, each pipeline reads an update message, reads the destination vertex and its property, performs the reduction, and stores the property back to BRAM. After all updates of an interval have been processed, the vertex properties are written back to DRAM.

One problem with the approach occurs when multiple pipelines concurrently process updates with the same destination vertex. For this case, the authors implement a combining network which reduces concurrently processed updates in case they have the same destination.

The authors observe that some algorithms, such as BFS or SSSP, are not a perfect fit for the edge-centric model as during each iteration of the algorithm only certain parts of the graph needs to be accessed. Thus, in an edge-centric model, several iterations through the edge list would need to take place (e.g., D iterations in BFS), and in many of these iterations only a few edges would actually be needed. Consequently, similarly to the observation by Umuroglu et al. [117], the authors take into account that the size of the frontier can vary drastically between different iterations. They observe that if the frontier is large enough, the edge-centric iteration is still more performant than randomly accessing only the required neighborhoods of the vertices in the frontier. However, for smaller frontiers, the vertex-centric approach is more efficient. Thus, they implement a hybrid approach that dynamically switches between the two paradigms based on the number of vertices in the frontier [136].

7 KEY PROBLEMS AND SOLUTIONS

In this section, we separately present the key problems that make graph processing on FPGAs challenging and we discuss how they are approached by various works. We focus on two key problems that are addressed by the vast majority of works dedicated to graph processing on FPGAs: insufficient bandwidth between the FPGA and DRAM (§ 7.1) and low internal FPGA storage (§ 7.2). Both of these challenges are strongly related in that most schemes that address one of them are also suitable for the other one. Thus, the structure of this section is not rigid.

7.1 Insufficient DRAM–FPGA Bandwidth

The first key problem is the low bandwidth between the FPGA and the main memory. This which makes graph processing memory-bound as most graph algorithms perform relatively simple computations on large amounts of data. Moreover, many graph algorithms require significant numbers of random accesses to DRAM [19]. This incurs further overheads as randomly accessing values from DRAM is substantially slower than loading data that is stored consecutively. We now present several solutions proposed in the FPGA literature for this particular issue.

7.1.1 High-Performance Memory Interfaces. Several works use modern memory interfaces that enable high bandwidth. The Hybrid Memory Cube (HMC) technology dramatically improves the bandwidth of DRAM. Benefits for graph processing on FPGA coming from HMCs are due to (1) the HMC substantial bandwidth improvement over traditional DRAM, (2) optimized parallel random memory access, and (3) in-memory locking and atomic operations. Now, HMCs are used by various FPGA solutions for graph processing, starting from the work by Zhang et al. [129]. Two more recent approaches have been able to further increase the performance of graph algorithms on FPGAs equipped with HMC by using several techniques such as relabeling indices according to their degrees [130] and incorporating degree-aware adjacency list compression [76].

7.1.2 Near-Data Computing on FPGA. Certain efforts propose to implement near-data computing architectures on FPGAs to ensure area and energy efficiency. For example, Heterogeneous Reconfigurable Logic (HRL) [53] is a reconfigurable architecture that implements near-data processing functionalities. HRL combines coarse-grained and fine-grained reconfigurable logic blocks. Among its other features are the separation of routing network into ones related to data and control signals. The former is based on a static bus-based routing fabric while the latter uses fine-grained bit-level design. Moreover, HRL uses specialized units for more efficient handling of branch operations and irregular data layouts. Now, the authors evaluate HRL on a wide spectrum of workloads, among others graph algorithms such as SSSP. HRL improves performance (per Watt) by 2.2× over FPGA and 1.7× over coarse-grained reconfigurable logic (CGRA), and it achieves 92% of the peak performance of a native near-data computing system based on custom accelerators for graph analytics and other workloads.

7.1.3 Data Redundancy for Sequential Accesses. As loading consecutive memory words is significantly more efficient that random memory accesses, some FPGA works focus on developing graph data layouts that exhibit more potential for sequential accesses. One example is the GraphOps [93] graph processing framework, where instead of storing the properties of vertices as an array of size n, the property values are replicated for every single incoming edge. Other examples are graph processing frameworks and implementations of specific algorithms where edges are streamed between the FPGA and the main memory, according to the edge-centric graph programming model [134–136].

7.1.4 Merging Updates on FPGA. A common step in graph algorithms is the reduction (i.e., merging) of updates that target the same vertex. For example, in BFS, multiple vertices in the frontier might be connected with the same vertex, but only one of them should be declared as its parent. Another example is PageRank, where several updates to a vertex are summed. Writing all these updates to DRAM unnecessarily stresses the available bandwidth as the updates could already be reduced on the FPGA before being stored to DRAM.

This solution to the limited bandwidth problem is proposed by Zhou et al. [135] in their optimized data layout that allows to combine some updates directly on the FPGA such that only one value has to be written to DRAM. In their edge-centric framework, the graph is split into partitions according to the source vertices (i.e., vertices with identical source vertices are placed inside the same partition). Then, the edges inside each partition are sorted by their destination vertex. Thus, *edges with the same destination vertex are processed consecutively and the corresponding updates be combined.* More details on this framework is provided in § 6.9.

Dai et al. [40, 41] reduce updates that target the same graph partition on the FPGA BRAM. This is achieved by partitioning the graph and sorting the edges by their destination vertex. Thus, all updates that target the same vertex are processed in the same batch. The size of the partitions is chosen such that the interval of updates fits BRAM.

7.1.5 Graph Compression. Another way to reduce the amount of communication between the FPGA and the main memory is graph compression [17, 20]. For example, Lee et al. [79] use compression for accelerating graph algorithms. The authors use their own compression scheme based on the established Webgraph framework [24]. They achieve compression ratios between 1.9 and 9.36, where the highest compression rates are obtained for web graphs.

7.2 Insufficient FPGA Internal Memory

Modern FPGAs host a set of configurable memory modules, called BRAM, which allow to store and access small amounts of data directly on the FPGA instead of the DRAM on the host machine. This memory can be compared to the cache in modern CPUs, only that the FPGA decides which data is stored in BRAM. Unfortunately, the capacity of BRAM is fairly small when compared to DRAM.

The problem of insufficient amounts of BRAM to store full graph datasets is complementary to the previous problem of insufficient bandwidth between FPGA and DRAM: fast and unlimited access to DRAM would diminish or even invalidate the importance of the ability to store large data structures in BRAM (and vice versa). Thus, some of the approaches described in the following sections could also be used to address the problem of insufficient DRAM–FPGA bandwidth.

Some existing approaches simply assume that the whole graph data set can be loaded into BRAM [4, 27, 49, 65, 71, 87, 92]. However, this approach limits the size of graphs that can be processed, see Table 3 for the details on the largest graphs used in surveyed works.

7.2.1 Partitioning the Graph. A common solution for processing large data structures with insufficient storage space is to split the data structure into smaller partitions which are loaded and executed one after the other. Partitioning a graph is difficult because no partition is fully separated from others: While processing one partition, one might have to update vertices that are stored in other partitions that are still residing in the larger, slower memory.

Zhou and Prasanna use the edge-centric approach of X-Stream but leave out the shuffle phase by directly storing the updates in sets corresponding to the partitions [134]. In another paper, the same authors use the same model with some improvements, such as merging the updates directly on the FPGA if the updates correspond to the same partition [135]. In their approach, the graph is divided into k partitions of approximately similar sizes. Each partition consists of an interval (range) of vertices, a shard containing edges that have one source vertex in the interval, and a set of values representing the updates that come from incoming edges (called **bins**). In the scatter phase, updates are written to the corresponding bins. In the gather phase, these updates are combined to compute new vertex values. Now, because of the data layout when processing one partition, all relevant vertex properties can be loaded into BRAM. Updates, however, can also target vertices from other partitions and thus have to be written to DRAM during the scatter phase.

Some works use more sophisticated partitioning approaches to improve the locality of vertices inside each partition and thus reduce the inter-partition communication. In GraphStep [71], the authors use the UMPack's multi-level partitioner [31]. Another work [70] uses the hypergraph partitioning tool PaToH [32]. Elaborate partitioning can reduce communication but it usually entails some additional pre-processing cost.

7.2.2 Using multiple FPGAs. Many graph FPGA papers argue in favor of using multiple FPGAs and thus scaling computational resources and achieving better performance [2, 4, 21, 23, 40, 41, 70, 71]. Some of these papers require the use of multiple FPGA when processing larger graphs that do not fit the BRAM modules of a single FPGA. Other designs aim to allow the scaling of performance above the performance that can be delivered by a single FPGA. One common problem with using multiple FPGAs is the communication overhead. In FPGP [40], the devices use a shared memory for the vertex properties which poses a bandwidth problem if too many devices use the same memory. ForeGraph [41] solves this problem by using a separate DRAM module for each device and an interconnect between all devices through which updates are transferred.

8 CONCLUSION

Graph processing on FPGAs is an important area of research as it can be used to accelerate numerous graph algorithms by reducing the amount of consumed power. Yet, it contains a diverse set of algorithms and processing frameworks, with a plethora of techniques and approaches. We present the first survey that analyzes the rich world of graph processing on FPGAs. We list and categorize the existing work, discuss key ideas, and present key insights and design choices. Our work can be used by architects and developers willing to select the best FPGA scheme in a given setting.

REFERENCES

[1] "Reducing Power Consumption in Xilinx FPGAs". available at: https://vhdlguru.blogspot.com/2011/07.

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

[3] W. Austin, V. Heuveline, and J.-P. Weiss. *Convey HC-1 hybrid core computer-The potential of FPGAs in numerical simulation*. KIT, 2010.

[4] J. W. Babb, M. Frank, and A. Agarwal. Solving graph problems with dynamic computation structures. In *High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*, volume 2914, pages 225–237. International Society for Optics and Photonics, 1996.

[5] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.

[6] P. Beame and J. Hastad. Optimal bounds for decision problems on the crcw pram. *Journal of the ACM (JACM)*, 36(3):643–670, 1989.

[7] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.

[8] S. Beamer, K. Asanović, and D. Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[9] R. Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

[10] T. Ben-Nun, M. Besta, S. Huber, A. N. Ziogas, D. Peter, and T. Hoefler. A Modular Benchmarking Infrastructure for High-Performance and Reproducible Deep Learning. IEEE, May 2019. Accepted at the 33rd IEEE International Parallel & Distributed Processing Symposium (IPDPS'19).

[11] T. Ben-Nun and T. Hoefler. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. *CoRR*, abs/1802.09941, Feb. 2018.

[12] M. Bernaschi, M. Bisson, E. Mastrostefano, and D. Rossetti. Breadth first search on apenet+. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 248–253. IEEE, 2012.

[13] M. Besta, M. Fischer, T. Ben-Nun, J. D. F. Licht, and T. Hoefler. Substream-Centric Maximum Matchings on FPGA. Feb. 2019. In Proceedings of the 27th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.

[14] M. Besta, S. M. Hassan, S. Yalamanchili, R. Ausavarungnirun, O. Mutlu, and T. Hoefler. Slim noc: A low-diameter on-chip network topology for high energy efficiency and scalability. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 43–55. ACM, 2018.

[15] M. Besta and T. Hoefler. Slim fly: A cost effective low-diameter network topology. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 348–359. IEEE Press, 2014.

[16] M. Besta and T. Hoefler. Accelerating irregular computations with hardware transactional memory and active messages. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 161–172. ACM, 2015.

[17] M. Besta and T. Hoefler. Survey and taxonomy of lossless graph compression and space-efficient graph representations. *arXiv preprint arXiv:1806.01799*, 2018.

[18] M. Besta, F. Marending, E. Solomonik, and T. Hoefler. Slimsell: A vectorizable graph representation for breadth-first search. In *Proc. IEEE IPDPS*, volume 17, 2017.

[19] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 93–104. ACM, 2017.

[20] M. Besta, D. Stanojevic, T. Zivic, J. Singh, M. Hoerold, and T. Hoefler. Log (graph): a near-optimal high-performance graph representation. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, page 7. ACM, 2018.

[21] B. Betkaoui, D. B. Thomas, W. Luk, and N. Przulj. A framework for fpga acceleration of large graph problems: Graphlet counting case study. In *Field-Programmable Technology (FPT), 2011 International Conference on*, pages 1–8. IEEE, 2011.

[22] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk. Parallel fpga-based all pairs shortest paths for sparse networks: A human brain connectome case study. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 99–104, Aug 2012.

[23] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk. A reconfigurable computing approach for efficient and scalable parallel graph exploration. In *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, pages 8–15. IEEE, 2012.

[24] P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602. ACM, 2004.

[25] E. G. Boman, K. D. Devine, and S. Rajamanickam. Scalable matrix computations on large scale-free graphs using 2d graph partitioning. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2013.

[26] U. Bondhugula, A. Devulapalli, J. Dinan, J. Fernando, P. Wyckoff, E. Stahlberg, and P. Sadayappan. Hardware/software integration for fpga-based all-pairs shortest-paths. In *Field-Programmable Custom Computing Machines, 2006. FCCM'06. 14th Annual IEEE Symposium on*, pages 152–164. IEEE, 2006.

[27] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, and P. Sadayappan. Parallel fpga-based all-pairs shortest-paths in a directed graph. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pages 10 pp.–, April 2006.

[28] O. Boruvka. O jistém problému minimálním. 1926.

[29] U. Brandes. A faster algorithm for betweenness centrality*. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[30] E. Cáceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, and S. W. Song. Efficient parallel graph algorithms for coarse grained multicomputers and bsp. In *International Colloquium on Automata, Languages, and Programming*, pages 390–400. Springer, 1997.

[31] A. E. Caldwell, A. B. Kahng, and I. L. Markov. Improved algorithms for hypergraph bipartitioning. In *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*, pages 661–666. ACM, 2000.

[32] U. V. Catalyurek and C. Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on parallel and distributed systems*, 10(7):673–693, 1999.

[33] B. Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *Journal of the ACM (JACM)*, 47(6):1028–1047, 2000.

[34] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, and H. Yang. Nxgraph: An efficient graph processing system on a single machine. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 409–420. IEEE, 2016.

[35] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[36] E. S. Chung, J. C. Hoe, and K. Mai. Coram: an in-fabric memory architecture for fpga-based computing. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 97–106. ACM, 2011.

[37] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science & Engineering*, 11(4):29–41, 2009.

[38] P. E. Compeau, P. A. Pevzner, and G. Tesler. How to apply de bruijn graphs to genome assembly. *Nature biotechnology*, 29(11):987, 2011.

[39] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[40] G. Dai, Y. Chi, Y. Wang, and H. Yang. Fpgp: Graph processing framework on fpga a case study of breadth-first search. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 105–110, New York, NY, USA, 2016. ACM.

[41] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang. Foregraph: Exploring large-scale graph processing on multi-fpga architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 217–226, New York, NY, USA, 2017. ACM.

[42] W. J. Dally and B. P. Towles. *Principles and practices of interconnection networks*. Elsevier, 2004.

[43] A. Dandalis, A. Mei, and V. K. Prasanna. Domain specific mapping for solving graph problems on reconfigurable devices. In *International Parallel Processing Symposium*, pages 652–660. Springer, 1999.

[44] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[45] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[46] D. Ediger and D. A. Bader. Investigating graph algorithms in the bsp model on the cray xmt. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1638–1645. IEEE, 2013.

[47] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17(3):449–467, 1965.

[48] N. Edmonds, A. Breuer, D. P. Gregor, and A. Lumsdaine. Single-source shortest paths with the parallel boost graph library. In *The Shortest Path Problem*, pages 219–248, 2006.

[49] N. Engelhardt and H. K.-H. So. Gravf: A vertex-centric distributed graph processing framework on fpgas. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pages 1–4. IEEE, 2016.

[50] R. W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[51] L. R. Ford Jr. Network flow theory. Technical report, RAND CORP SANTA MONICA CA, 1956.

[52] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[53] M. Gao and C. Kozyrakis. Hrl: Efficient and flexible reconfigurable logic for near-data processing. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pages 126–137. Ieee, 2016.

[54] R. Gerstenberger, M. Besta, and T. Hoefler. Enabling highly-scalable remote memory access programming with mpi-3 one sided. *Scientific Programming*, 22(2):75–91, 2014.

[55] L. Gianinazzi, P. Kalvoda, A. De Palma, M. Besta, and T. Hoefler. Communication-avoiding parallel minimum cuts and connected components. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 219–232. ACM, 2018.

[56] P. B. Gibbons. A more practical pram model. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 158–168. ACM, 1989.

[57] Y. Han, V. Y. Pan, and J. H. Reif. Efficient parallel algorithms for computing all pair shortest paths in directed graphs. *Algorithmica*, 17(4):399–415, Apr 1997.

[58] S. Hauck and A. DeHon. *Reconfigurable computing: the theory and practice of FPGA-based computation*, volume 1. Elsevier, 2010.

[59] M. Herlihy and J. E. B. Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[60] S. Hong, S. Salihoglu, J. Widom, and K. Olukotun. Simplifying scalable graph processing with a domain-specific language. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 208. ACM, 2014.

[61] M. Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, Feb 2014.

[62] L. Huelsbergen. A representation for dynamic graphs in reconfigurable hardware and its application to fundamental graph algorithms. In *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, pages 105–115. ACM, 2000.

[63] Intel. Understanding How the New Intel HyperFlex FPGA Architecture Enables Next-Generation High-Performance Systems. Technical report.

[64] B. Jacob, S. Ng, and D. Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[65] G. R. Jagadeesh, T. Srikanthan, and C. Lim. Field programmable gate array-based acceleration of shortest-path computation. *IET computers & digital techniques*, 5(4):231–237, 2011.

[66] B. Jiang. A short note on data-intensive geospatial computing. In *Information Fusion and Geographic Information Systems*, pages 13–17. Springer, 2011.

[67] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM)*, 24(1):1–13, 1977.

[68] V. Kalavri, V. Vlassov, and S. Haridi. High-level programming abstractions for distributed graph processing. *IEEE Transactions on Knowledge and Data Engineering*, 30(2):305–324, 2018.

[69] R. Kaleem, S. Pai, and K. Pingali. Stochastic gradient descent on gpus. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, GPGPU-8, pages 81–89, New York, NY, USA, 2015. ACM.

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

[71] N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, F. Thomas Jr, A. DeHon, et al. Graphstep: A system architecture for sparse-graph algorithms. In *Field-Programmable Custom Computing Machines, 2006. FCCM'06. 14th Annual IEEE Symposium on*, pages 143–151. IEEE, 2006.

[72] N. Kapre and P. Moorthy. A case for embedded fpga-based socs in energy-efficient acceleration of graph problems. *Supercomput. Front. Innov.: Int. J.*, 2(3):76–86, July 2015.

[73] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, et al. Mathematical foundations of the graphblas. *arXiv preprint arXiv:1606.05790*, 2016.

[74] J. Kepner and J. Gilbert. *Graph algorithms in the language of linear algebra*, volume 22. SIAM, 2011.

[75] A. Khan and C. Aggarwal. Query-friendly compression of graph streams. In *Advances in Social Networks Analysis and Mining (ASONAM), 2016 IEEE/ACM International Conference on*, pages 130–137. IEEE, 2016.

[76] S. Khoram, J. Zhang, M. Strange, and J. Li. Accelerating graph analytics by co-optimizing storage and access on an fpga-hmc platform. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 239–248. ACM, 2018.

[77] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[78] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: large-scale graph computation on just a PC. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, 2012.

[79] J. Lee, H. Kim, S. Yoo, K. Choi, H. P. Hofstee, G.-J. Nam, M. R. Nutter, and D. Jamsek. Extrav: boosting graph processing near storage with a coherent accelerator. *Proceedings of the VLDB Endowment*, 10(12):1706–1717, 2017.

[80] G. Lei, Y. Dou, R. Li, and F. Xia. An fpga implementation for solving the large single-source-shortest-path problem. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 63(5):473–477, 2016.

[81] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 303–314. ACM, 2010.

[82] H. Liu and P. Singh. Conceptnet—a practical commonsense reasoning tool-kit. *BT technology journal*, 22(4):211–226, 2004.

[83] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. *preprint arXiv:1006.4990*, 2010.

[84] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry. Challenges in Parallel Graph Processing. *Par. Proc. Let.*, 17(1):5–20, 2007.

[85] X. Ma, D. Zhang, and D. Chiou. Fpga-accelerated transactional execution of graph workloads. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 227–236. ACM, 2017.

[86] 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*, pages 135–146. ACM, 2010.

[87] O. Mencer, Z. Huang, and L. Huelsbergen. Hagar: Efficient multi-context graph processors. In *International Conference on Field Programmable Logic and Applications*, pages 915–924. Springer, 2002.

[88] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[89] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, Oct 2016.

[90] M. E. Newman. A measure of betweenness centrality based on random walks. *Social networks*, 27(1):39–54, 2005.

[91] S. Ni, Y. Dou, D. Zou, R. Li, and Q. Wang. Parallel graph traversal for fpga. *IEICE Electronics Express*, 11(7):20130987–20130987, 2014.

[92] 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*, pages 25–28, May 2014.

[93] 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, pages 111–117, New York, NY, USA, 2016. ACM.

[94] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk. Energy efficient architecture for graph analytics accelerators. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 166–177. IEEE, 2016.

[95] M. O'Neill. Neural network for recognition of handwritten digits (2006). *Source: http://www. codeproject. com/KB/library/NeuralNetRecognition. asp x*, 2016.

[96] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[97] C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity.* Courier Corporation, 1998.

[98] J. T. Pawlowski. Hybrid memory cube (hmc). In *2011 IEEE Hot chips 23 symposium (HCS)*, pages 1–24. IEEE, 2011.

[99] O. Pell, O. Mencer, K. H. Tsoi, and W. Luk. Maximum performance computing with dataflow engines. In *High-performance computing using FPGAs*, pages 747–774. Springer, 2013.

[100] C. Poirier, B. Gosselin, and P. Fortier. Dna assembly with de bruijn graphs using an fpga platform. *IEEE/ACM transactions on computational biology and bioinformatics*, 15(3):1003–1009, 2018.

[101] R. C. Prim. Shortest connection networks and some generalizations. *Bell system technical journal*, 36(6):1389–1401, 1957.

[102] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[103] Y. Saad. Sparskit: A basic tool kit for sparse matrix computations. 1990.

[104] T. Schank. Algorithmic aspects of triangle-based network analysis. 2007.

[105] P. Schmid, M. Besta, and T. Hoefler. High-performance distributed rma locks. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 19–30. ACM, 2016.

[106] H. Schweizer, M. Besta, and T. Hoefler. Evaluating the cost of atomic operations on modern architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 445–456. IEEE, 2015.

[107] Y. Shiloach and U. Vishkin. An o (log n) parallel connectivity algorithm. Technical report, Computer Science Department, Technion, 1980.

[108] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 149–160. IEEE, 2015.

[109] E. Solomonik, M. Besta, F. Vella, and T. Hoefler. Scaling betweenness centrality using communication-efficient sparse matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 47. ACM, 2017.

[110] K. Sridharan, T. Priya, and P. R. Kumar. Hardware architecture for finding shortest paths. In *TENCON 2009-2009 IEEE Region 10 Conference*, pages 1–5. IEEE, 2009.

[111] C. Sun, E. W. Chew, N. Shaikh Husin, and M. Khalil-Hani. Accelerating graph algorithms with priority queue processor. In *Regional Postgraduate Conference on Engineering and Science (RPCES 2006)*, pages 257–262, 2006.

[112] J. Sun, N.-N. Zheng, and H.-Y. Shum. Stereo matching using belief propagation. *IEEE Transactions on pattern analysis and machine intelligence*, 25(7):787–800, 2003.

[113] R. Szeliski, R. Zabih, D. Scharstein, O. Veksler, V. Kolmogorov, A. Agarwala, M. Tappen, and C. Rother. A comparative study of energy minimization methods for markov random fields with smoothness-based priors. *IEEE transactions on pattern analysis and machine intelligence*, 30(6):1068–1080, 2008.

[114] A. Tate, A. Kamil, A. Dubey, A. Größlinger, B. Chamberlain, B. Goglin, C. Edwards, C. J. Newburn, D. Padua, D. Unat, et al. Programming abstractions for data locality. PADAL Workshop 2014, April 28–29, Swiss National Supercomputing Center ..., 2014.

[115] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM (JACM)*, 46(3):362–394, 1999.

[116] M. Tommiska and J. Skyttä. Dijkstra's shortest path routing algorithm in reconfigurable hardware. In *International Conference on Field Programmable Logic and Applications*, pages 653–657. Springer, 2001.

[117] Y. Umuroglu, D. Morrison, and M. Jahre. Hybrid breadth-first search on a single-chip fpga-cpu heterogeneous platform. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2015.

[118] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[119] B. S. C. Varma, K. Paul, M. Balakrishnan, and D. Lavenier. Fassem: Fpga based acceleration of de novo genome assembly. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 173–176. IEEE, 2013.

[120] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya. A blocked all-pairs shortest-paths algorithm. *Journal of Experimental Algorithmics (JEA)*, 8:2–2, 2003.

[121] 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*, pages 70–77. IEEE, 2010.

[122] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440, 1998.

[123] Xilinx. UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices. Technical report, 06 2016.

[124] Xilinx. Versal: The First Adaptive Compute Acceleration Platform (ACAP). Technical report, 02 2018.

[125] C. Yang. An efficient dispatcher for large scale graphprocessing on opencl-based fpgas. *arXiv preprint arXiv:1806.11509*, 2018.

[126] C. Yang, Y. Wang, and J. D. Owens. Fast sparse matrix and sparse vector multiplication algorithm on the GPU. In *Par. and Dist. Proc. Symp. Work. (IPDPSW), IEEE Intl.*, pages 841–847. IEEE, 2015.

[127] P. Yao. An efficient graph accelerator with parallel data conflict management. *arXiv preprint arXiv:1806.00751*, 2018.

[128] Y.S.Horawalavithana. On the Design of an Efficient Hardware Accelerator for Large Scale Graph Analytics.

[129] 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 *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 207–216, New York, NY, USA, 2017. ACM.

[130] J. Zhang and J. Li. Degree-aware hybrid graph traversal on fpga-hmc platform. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 229–238. ACM, 2018.

[131] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen. Cambricon-x: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct 2016.

[132] S. Zhou, C. Chelmis, and V. K. Prasanna. Accelerating large-scale single-source shortest path on fpga. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 129–136. IEEE, 2015.

[133] S. Zhou, C. Chelmis, and V. K. Prasanna. Optimizing memory performance for fpga implementation of pagerank. In *ReConFig*, pages 1–6, 2015.

[134] S. Zhou, C. Chelmis, and V. K. Prasanna. High-throughput and energy-efficient graph processing on fpga. In *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*, pages 103–110. IEEE, 2016.

[135] S. Zhou, R. Kannan, H. Zeng, and V. K. Prasanna. An fpga framework for edge-centric graph processing. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pages 69–77. ACM, 2018.

[136] S. Zhou and V. K. Prasanna. Accelerating graph analytics on cpu-fpga heterogeneous platform. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 137–144. IEEE, 2017.