Demystifying Memory Access Patterns of FPGA-Based Graph Processing Accelerators

Jonas Dann, Daniel Ritter  
(firstname.lastname@sap.com)  
SAP SE  
Walldorf, (Germany)

Holger Fröning  
holger.froening@ziti.uni-heidelberg.de  
Heidelberg University  
Heidelberg, (Germany)

ABSTRACT

Recent advances in reprogrammable hardware (e.g., FPGAs) and memory technology (e.g., DDR4, HBM) promise to solve performance problems inherent to graph processing like irregular memory access patterns on traditional hardware (e.g., CPU). While several of these graph accelerators were proposed in recent years, it remains difficult to assess their performance and compare them on common graph workloads and accelerator platforms, due to few open source implementations and excessive implementation effort.

In this work, we build on a simulation environment for graph processing accelerators, to make several existing accelerator approaches comparable. This allows us to study relevant performance dimensions such as partitioning schemes and memory technology, among others. The evaluation yields insights into the strengths and weaknesses of current graph processing accelerators along these dimensions, and features a novel in-depth comparison.

CCS CONCEPTS

• Computing methodologies → Modeling and simulation; • Computer systems organization → Reconfigurable computing; • Hardware → Memory and dense storage.

KEYWORDS

Graph processing, DRAM, HBM, FPGA, Benchmark, Simulation

1 INTRODUCTION

The irregular memory access and little computational intensity inherent to graph processing cause major performance challenges on traditional hardware (e.g., CPUs) due to effects like DRAM row switching and ineffective use of fetched cache lines [1, 4, 11, 17]. Recent advances in reprogrammable hardware like field programmable gate arrays (FPGAs), and memory technology such as DDR4 and high-bandwidth memory (HBM) promise to accelerate common graph problems (e.g., breadth-first search (BFS), PageRank (PR), weakly-connected components (WCC)) [4, 10]. Especially FPGA-based graph processing accelerators (e.g., [25, 28]) show good results for irregular memory access acceleration due to growing sizes of FPGA on-chip memory and custom memory usage. However, there are deficiencies in benchmarking of these accelerators due to a multitude of configurations regarding available FPGAs, memory architectures, workloads, input data, and the lack of accepted benchmark standards [10, 11]. This makes it difficult to assess the implications of different design decisions and optimizations, and — most importantly — to compare the proposed accelerators [1, 11].

Recent work [10] introduced a simulator approach (cf. Fig. 1) that permits to quickly reproduce and compare different graph accelerator solutions in a synthetic, fixed environment. Based on the observation that off-chip memory access dominates the overall graph processing accelerator performance, reproducibility of results is demonstrated with simulating the approximate request flow based on on-chip (in block RAM (BRAM)) and off-chip state and graph data in the off-chip DRAM. The results are summarized in Fig. 2, denoting the simulation percentage error $\varepsilon = \frac{1000|x-y|}{x}$, with simulation performance $s$, on equal memory configurations and graph data sets, compared to the performance numbers $t$ taken from the respective article grouped by accelerators and graph problems.

We get a reasonable mean error of $22.63\%$ with two outliers in BFS on ForeGraph and single-source shortest-paths (SSSP) on HitGraph caused by insufficient specification of root vertices (cf. [10]).

In this work, our objectives are understanding performance and comparability of graph processing accelerators. For assessing these objectives, we leverage the simulation approach from [10] to conceptually analyze and compare existing graph processing accelerator approaches along relevant performance dimensions: (i) accelerator design decisions, (ii) graph problems, (iii) data set characteristics, (iv) memory technology, and (v) memory access optimizations. Based on criteria like reported performance numbers on commodity hardware and sufficient conceptually details, we choose four state-of-the-art systems found in a recent survey [11], namely AccuGraph...
[25], HitGraph [28], ForeGraph [9], and ThunderGP [8] and update Fig. 2 from [10] with simulation error for ForeGraph and ThunderGP. While AccuGraph and HitGraph are orthogonal approaches representing the currently most relevant paradigms, edge- and vertex-centric graph processing (both with horizontal partitioning) [11], ForeGraph is one of the few systems with interval-shard partitioning and compressed edge list, and ThunderGP uses vertical partitioning with a sorted edge list. With that we make the following contributions:

- We provide a classification of existing graph processing accelerators and extract their memory access patterns.
- We conduct a performance comparison and analysis by comprehensively exploring dimensions (i-iii) and (v).
- We analyze graph workloads and accelerator scalability on modern memory like HBM for the first time (dimension (iv)).

Notably, among other insights, we discover a trade-off in the immediate update propagation scheme of AccuGraph and ForeGraph compared to the 2-phase update propagation scheme of HitGraph and ThunderGP. Additionally, we confirm that modern memory like HBM does not necessarily lead to better performance (cf. [19, 20]).

2 BACKGROUND

In this section, we give a brief background on FPGAs and how they interact with current memory technologies, as well as the simulation environment that this work is based on.

2.1 Memory Hierarchy of FPGAs

As a processor architecture platform, FPGA chips allow for mapping custom digital circuit designs (a set of logic gates and their connections) to a grid of resources (i.e., look-up tables, registers) connected with a programmable interconnection network (cf. Fig. 3). For frequently used complex functionality like floating point computation FPGAs contain digital signal processors (DSP). Access to off-chip resources like DRAM and network controllers is possible over I/O pins. The memory hierarchy of FPGAs is split up into on-chip and off-chip memory. On-chip, FPGAs implement distributed memory, that is made up of single registers and is mostly used as storage for working values, and block RAM (BRAM) in the form of SRAM memory components for fast storage of data structures. On modern FPGAs, there is about as much BRAM as there is cache on CPUs (all cache levels combined), but contrary to the fixed cache hierarchies of CPUs, BRAM is finely configurable to the application.

For storage of larger data, DRAM is attached as off-chip memory (e.g., DDR3, DDR4, or HBM). Exemplary for DRAM organization, we show the internal organization of DDR3 memory in Fig. 3. At the highest level, DRAM is split up into channels each with its own 64-bit bus to the connected chip. For each channel, multiple ranks may operate in parallel but on the same bus to increasing memory capacity. Each rank is then organized as 8 banks with a row buffer and several thousand rows each. At the lowest level, data is stored in DRAM cells (with the smallest number of cells addressable at once being called a column). Read or write requests to data in a bank are served by the row buffer based on three scenarios: (1) When the addressed row is present in the row buffer, the request is served with low latency (e.g., 11ns). (2) If the row buffer is empty, activating the addressed row adds additional latency (e.g., 11ns). (3) However, if the row buffer currently contains a different row, the present row has to be pre-charged (e.g., 11ns), before the addressed row can be activated and the request served. As a reference point, a last level cache miss on current Skylake Intel CPUs takes 17ns at 2.6GHz clock. Additionally, there is a minimum latency between switching rows (e.g., 28ns). Thus, for high performance, row switching should be avoided as much as possible. To achieve good performance, current DRAM additionally employ 8n prefetching (meaning 8 bursts). Thus, 64 bytes are returned for each request which we call a cache line in the following. DDR4 doubles total number of banks at the cost of added latency due to another hierarchy level called bank groups, which group two to four banks. HBM as a new stacked memory technology introduces very high bandwidth in a small package. The single channels have double as many banks (16) as DDR3 with half the prefetch (4n) which however transport double the data per cycle (128-bit). Additionally, HBM has smaller row buffers and can have many more channels in a confined space.

2.2 Memory Access Simulation Environment

Since memory access is the dominating factor in graph processing, the necessity of cycle-accurate simulation of on-chip data flow is relaxed and only the off-chip memory access pattern is modeled in the simulation environment [10]. This means modelling request types, addressing, volume, and ordering. Request type modelling is trivial since it is clear when requests read and write data. For request addressing, we assume that the different data structures lie

---

1We refer to [10] for details on outliers
adjacent in memory as plain arrays. We generate memory addresses according to this memory layout and the width of the array types in bytes. Request volume modelling is mostly based on the size $n$ of the vertex set, the size $m$ of the edge set, average degree $\text{deg}$, and partition number $k$ of a given graph. Lastly, we only simulate request ordering through mandatory control flow caused by data dependencies of requests. We assume that computations and on-chip memory accesses are instantaneous by default. The simulation implementations are further based on memory access abstractions that we introduce in Sect. 3.2 as needed.

A DRAM simulator is an integral part of the simulation environment (cf. Fig. 1). For our purposes we need a DRAM simulator that supports at least DDR3 (for HitGraph [28]) and DDR4 (for AccuGraph [25], ForeGraph [9], and ThunderGP [8]). We chose Ramulator [14] for this work over other alternatives like DRAMSim2 [18] or USIMM [6]. To the best of our knowledge, it is the only DRAM simulator which supports both DDR3 and DDR4 (among many others like LPDDR3/4 and HBM). The relevant Ramulator configuration parameters for this work are: (a) DRAM standard, (b) Number of channels, (c) Number of ranks, (d) DRAM speed specification, and (e) DRAM organization.

3 GRAPH PROCESSING ACCELERATORS

In this section, we introduce basics of graph processing accelerators and thereafter motivate the selection of accelerators for this work and show their implementation in the simulation environment.

3.1 Partition, Iteration, & Update Schemes

Graph processing accelerators utilize two dimensions of graph partitioning: horizontal and vertical. Horizontal partitioning means dividing up the vertex set into equal intervals and letting each partition contain the outgoing edges of one interval. Examples are AccuGraph which uses a horizontally partitioned compressed sparse row (CSR) format of the inverted edges and HitGraph which uses a horizontally partitioned edge list. Vertical partitioning divides the vertex set into intervals like horizontal partitioning but each partition contains the incoming edges of its interval. ThunderGP uses a vertically partitioned edge list. As a third partitioning approach, interval-shard partitioning [29] employs both vertical and horizontal partitioning at once and is used by ForeGraph.

Depending on the underlying graph data structure, graphs are processed based on two fundamentally different approaches: edge- and vertex-centric graph processing. Edge-centric systems (e.g., ForeGraph, HitGraph, and ThunderGP) iterate over the edges as primitives of the graph on an underlying edge list. Vertex-centric systems iterate over the vertices and their neighbors as primitives of the graph on an underlying adjacency list (e.g., CSR). For the vertex-centric approach there further is a distinction into push- and pull-based data flow. A push-based data flow denotes that values are pushed along the forward direction of edges to update neighboring vertices. A pull-based data flow (e.g., AccuGraph) denotes that values are pulled along the inverse direction of edges from neighboring vertices to update the current vertex.

Lastly, there are currently three update propagation schemes. Immediate update propagation directly applies updates to the working value set such that they can be used in the remainder of the current iteration already, 2-phase update propagation collects all updates separately and only applies them in a second phase of each iteration, and level-synchronous update propagation, specifically for BFS, maintains a frontier of vertices with updates in each iteration only on their corresponding values.

3.2 Selected Accelerators

For this work, we selected four graph processing accelerators from the list of accelerators in [11], based on if (i) they run on commodity FPGAs, (ii) the vertex set is not required to fit into on-chip memory, and (iii) the respective article provides enough detail to model the memory access pattern. The accelerators fitting all but one of these criteria can be found in Tab. 1 and the ones we chose to include in this work are highlighted in bold. We excluded the accelerators based on the Convey HC-2 and HMC systems because they are not implementable with commercially available FPGAs anymore and not reproducible. Additionally, only supporting BFS restricts their usefulness. Furthermore, we excluded the systems by Ayupov et al. [3] and Yang et al. [24] because they did not provide sufficient detail to reproduce the results with the simulation environment. The set of accelerators we choose for this work represents the currently highest performing graph processing accelerators and all currently applied partitioning and iteration schemes. In the following, we introduce how they can be implemented in the simulation environment in alphabetical order. Note that developing and verifying a complicated FPGA design usually takes weeks, while the implementation of a new graph accelerator approach in our simulation environment takes days or hours. For more detail than we provide here on the notation and implementation aspects, we refer to [10].

3.2.1 AccuGraph. AccuGraph [25] proposes a flexible accumulator based on a modified prefix-adder able to produce and merge updates to multiple vertices per cycle. Fig. 4 shows the request and control flow modelling of AccuGraph. The controller is triggered and iterates over the graph until there are no more changes in the previous iteration. Each iteration triggers processing of all partitions. Processing of each partition starts with triggering the prefetch request producer which prefetches the partitions $\frac{n}{k}$ vertex values (with $n = |V|$ and the partition count $k$) sequentially which is passed through a cache line memory access abstraction merging adjacent requests to the same cache line into one. Thereafter, values and pointers of all destination vertices are fetched sequentially. Both of those request streams are annotated with callbacks which return control flow for each served request. Those two request streams are merged round-robin, because a value is only useful with the associated pointers. In parallel, neighbors are read from memory sequentially, annotated with their value from the prefetched

---

Footnote: Code is available at: https://github.com/UniHD-CEG/graphulator
This results in purely sequential off-chip requests with all random vertex values, and one edge is materialized for each neighbor with its corresponding source vertex. The aforementioned accumulator thereafter produces updates for each edge. If the source vertex value changes, the result is written back to off-chip memory (unchanged values are filtered by the filter memory access abstraction). All of these request streams are merged by priority with write request taking the highest priority and neighbors the second highest.

### 3.2.2 ForeGraph
ForeGraph [9] stores the edges of the shards as compressed 32-bit edges with two 16-bit vertex identifiers each. This is possible due to the interval size being limited to 65,536 vertices. In each iteration, ForeGraph prefetches the source intervals one after another and for each source interval processes its corresponding shards by additionally prefetching the destination interval and sequentially reading and processing the edges (Fig. 5). This results in purely sequential off-chip requests with all random vertex value requests during edge processing being served by caches on-chip. After a shard has been processed, the destination interval is sequentially written back to off-chip memory. To achieve competitive performance, ForeGraph instantiates $p$ processing elements (PE). These each work on their own set of source intervals and share the memory access in round-robin fashion.

### 3.2.3 HitGraph
HitGraph [28] execution starts with triggering a controller that itself triggers iterations of edge-centric processing until there were no value changes in the previous iteration (Fig. 6). In each iteration, the controller first schedules all $k$ partitions for the scatter phase (producing updates), before scheduling all partitions to the gather phase (applying updates). Partitions are assigned beforehand to channels of the memory (four channels in [28]) and there are $p$ processing elements (PE), one for each channel.

The scatter phase starts by prefetching the $\frac{n}{kp}$ values of the current partition. After all requests are produced, the prefetch step triggers the edge reading step that reads all $\frac{n}{kp}$ edges of the partition. For each edge request, we attach a callback that triggers producing an update request. The target address depends on its destination vertex that can be part of any of the $k$ partitions. Thus, there is a crossbar (unique to this accelerator) that routes each update request to a cache line access abstraction for each partition to sequentially write into a partition-specific update queue.

Similar to scatter, the gather phase starts with prefetching the $\frac{n}{kp}$ vertex values of the current partition sequentially. After value requests have been produced, the prefetch producer triggers the update producer, which sequentially reads the update queue written by the scatter phase before. For each update we register a callback that triggers the value write. The value writes are not necessarily sequential, but especially for iterations where a lot of values are written, there is a lot of locality.

All request streams in each PE are merged directly into one stream without any specific merging logic, since only one producer is producing requests at a time. Since all PEs are working on independent channels and Ramulator only offers one endpoint for all channels combined, we employ a round-robin merge of the PE requests in order not to starve any channel.

### 3.2.4 ThunderGP
Like HitGraph, ThunderGP [7] is based on a 2-phase update propagation scheme based on edge-centric iteration over the input graph. The graph is vertically partitioned into $k$ partitions and each partition is split up into $p$ chunks where $p$ is equal to the number of memory channels. Each memory channel contains the whole vertex value set of the graph, its corresponding vertex values, and one edge is materialized for each neighbor with its corresponding source vertex. The aforementioned accumulator thereafter produces updates for each edge. If the source vertex value changes, the result is written back to off-chip memory (unchanged values are filtered by the filter memory access abstraction). All of these request streams are merged by priority with write request taking the highest priority and neighbors the second highest.

### Table 1: FPGA-accelerated graph systems in context of most relevant design and implementation decisions

| Identifier          | Supported graph problems | System     | Iter. scheme | Flow       | Partitioning | Binary rep. | Update prop. |
|---------------------|--------------------------|------------|--------------|------------|--------------|-------------|--------------|
| ForeGraph [9]       | BFS, PR, WCC             | Simulation | Edge-centric | n/a        | Interval-shard | Compr. edge list | Immediate    |
| HitGraph [28]       | BFS, PR, WCC, SSSP, SpMV | FPGA       | Edge-centric | n/a        | Horizontal   | Sorted edge list | 2-phase      |
|AccuGraph [25]      | BFS, PR, WCC             | FPGA       | Edge-centric | n/a        | Vertical     | Sorted edge list | 2-phase      |
| Betkaoui et al. [5] | BFS                      | Convey HC-2 | Vertex-centric | Push       | Horizontal   | CSR         | Level-synch. |
| CyGraph [2]         | BFS                      | Convey HC-2 | Vertex-centric | Push       | Horizontal   | CSR         | Level-synch. |
| Ayupov et al. [3]   | BFS, PR, WCC             | Simulation | Vertex-centric | Push       | None         | CSR (in-place) | Level-synch. |
| TorusBFS [15]       | BFS                      | Convey HC-2 | Vertex-centric | Push       | None         | CSR         | Level-synch. |
| Yang et al. [24]    | BFS, PR, WCC             | FPGA       | Hybrid       | n/a        | Vertical     | EBs & CSR   | 2-phase      |
| Zhang et al. [26]   | BFS                      | HMC FPGA   | Vertex-centric | Push       | Hybrid       | None        | CSR         | Level-synch. |

Iter. scheme: Iteration scheme; Binary rep.: Binary representation; Update prop.: Update propagation; n/a: Not applicable; EBs: Edge blocks

![Figure 5: ForeGraph](image1)

![Figure 6: HitGraph (adapted from [10])](image2)

![Figure 7: ThunderGP](image3)
Table 2: Graphs used often by systems in Tab. 1 (real-world graphs from SNAP [16]; Graph500 generator for R-MAT)

| Name          | $|V|$ | $|E|$ | Dir. | Degs. | $D_{\text{avg}}$ | $\phi$ | SCC |
|---------------|-----|-----|------|------|-----------------|-------|-----|
| twitter (tw)  | 41.7M | 1,468.4M | 20 | 20 | 35.25 | 75 | 0.80 |
| live-journal (lj) | 4.8M | 69.0M | 20 | 20 | 14.23 | 20 | 0.79 |
| orkut (or)    | 3.1M | 117.2M | 20 | 20 | 76.28 | 9 | 1.00 |
| wikki-talk (wt) | 2.4M | 5.0M | 20 | 20 | 2.10 | 11 | 0.05 |
| pokec (pk)    | 1.6M | 30.6M | 20 | 20 | 37.51 | 14 | 1.00 |
| youtube (yt)  | 1.2M | 3.0M | 20 | 20 | 5.16 | 20 | 0.98 |
| dblp (db)     | 426.0K | 1.0M | 20 | 20 | 4.93 | 21 | 0.74 |
| slashdot (sd) | 82.2K | 948.4K | 20 | 20 | 11.54 | 13 | 0.87 |
| roadnet-ca (rd) | 2.8M | 2.8M | 20 | 20 | 2.81 | 849 | 0.99 |
| berk-stan (bk) | 685.2K | 7.6M | 20 | 20 | 11.09 | 714 | 0.49 |
| rmat-24-16 (r24) | 16.8M | 268.4M | 20 | 20 | 16.00 | 19 | 0.02 |
| rmat-21-86 (r21) | 2.1M | 180.4M | 20 | 20 | 86.00 | 14 | 0.10 |

Dir.: Directed; Degs.: Degree distribution on log. scale; SCC: Ratio of vertices in the largest strongly-connected component to $n$; $\phi$: yes; $\Theta$: no

Table 3: DRAM configurations

| Identifier          | Type       | Chan. | Ranks | Data rate | BW   | Size | RBS |
|---------------------|------------|-------|-------|-----------|------|------|-----|
| AccuGraph           | DDR4       | 1     | 1     | 2400 MT/s | 19.2Gb/s | 2Gb | 8KB |
| ForeGraph           | DDR4       | 1     | 1     | 2400 MT/s | 19.2Gb/s | 4Gb | 8KB |
| HitGraph            | DDR3       | 4     | 2     | 1600 MT/s | 12.8Gb/s | 8Gb | 8KB |
| ThunderGP           | DDR4       | 4     | 1     | 2400 MT/s | 19.2Gb/s | 16Gb | 8KB |
| Default             | DDR4       | 1−4   | 1     | 2400 MT/s | 19.2Gb/s | 16Gb | 8KB |
| DDR3                | DDR3       | 1−4   | 1     | 2133 MT/s | 17.1Gb/s | 8Gb | 8KB |
| HBM                 | HBM        | 1−8   | n/a   | 1000 MT/s | 16.6Gb/s | 4Gb | 2KB |

Chan.: Channels; BW: Bandwidth / Chan.; RBS: Row buffer size; n/a: not applicable

two vertex identifiers wide and a weighted edge is an additional 32 bits wider due to the attached edge weight. This is sensible for all accelerators we encountered, since there are no excessively large benchmark graphs or requirements on more precision.

Tab. 3 shows the DRAM configurations used in the respective paper of the selected accelerators as well as the DRAM configurations we use in this work. The default is the DDR4 configuration, since this is the most common in the systems the selected accelerators run on. The DDR3 and HBM configurations are used to compare performance on different memory technologies in Sect. 4.4. By default, we use one channel, but also do a scale test for HitGraph and ThunderGP due to their support for multiple channels.

We consider the five graph problems breadth first search (BFS), PageRank (PR), weakly connected components (WCC), sparse matrix-vector multiplication (SpMV), and single-source shortest-paths (SSSP). However, SpMV and SSSP require edge weights which only HitGraph and ThunderGP support.

We use Ramulator\(^\text{2}\) commit dd326 and add a function to flush the stats for multiple consecutive runs. To compile Ramulator and the simulation environment we use clang++ 5.0.1. For Ramulator with C++11 and for the simulation environment with C++17 and -D_FILE_OFFSET_BITS=64 to be able to read files larger than 2GB.

In subsequent performance measurements, we use the Graph500 benchmark’s MTEPS definition, which specifies MTEPS as $|E|/t_{\text{exec}}$, where $t_{\text{exec}}$ denotes the execution time. Notably, this is different to the measure that most graph processing accelerator articles report (i.e., total number of edges read during execution divided by execution time), which we call MREPS. MREPS do not normalize the runtime to graph size but rather denote raw edge processing performance. For both MTEPS and MREPS higher is better. Raw performance numbers can be found in Appendix A.

4.2 Design Decisions & Graph Problems

Fig. 8 shows a comparison of the four graph processing accelerators on all graphs from Tab. 2 for BFS, PR (one iteration), and WCC. While we start with a general performance analysis with respect to accelerator design decisions and graph problems (performance dimensions (i) and (ii)) in this subsection, we refer back to Fig. 8 throughout the whole of Sect. 4 for more particular performance effects. We also introduce four critical performance metrics in Fig. 9 to explain our observations throughout Sect. 4.

Number of iterations has the biggest impact on performance since each iteration entails prefetching, edge, and value reading and writing (Fig. 9(a)). Edge reading is the dominating factor of each iteration. Thus, bytes per edge (Fig. 9(b)) and edges per iteration (Fig. 9(d)) are the two second most important performance metrics. Finally, values read

---

\(^{1}\)Root vertices: tw - 2748769; lj - 772860; or - 1386825; wt - 17540; pk - 315318; yt - 140229; dh - 9799; ad - 30279; r1d - 1166467; bk - 546279; rz4 - 535262; rz5 - 74769

\(^{2}\)Ramulator, visited 5/21: https://github.com/CMU-SAFARI/ramulator
Figure 8: Comparison of graph processing accelerators grouped by graphs for BFS, PR, and WCC (DDR4, single-channel)

Overall, performance of PR is the highest because only one iteration is performed (cf. Fig. 8). BFS and WCC performance is overall similar. For BFS and WCC, performance on the bk and rd graph — which we break out in separate plots with a difference y-axis scale — is significantly lower. This, however, is expected as bk and rd are graphs with a large diameter and thus require many more iterations to complete. We notice that AccuGraph and ForeGraph on average perform better than HitGraph and ThunderGP. To explain this, we additionally notice that AccuGraph and ForeGraph finish in significantly less iterations over the graph for BFS and WCC (not shown) than HitGraph and ThunderGP relative to the graph’s diameter (cf. Fig. 9(a)). This is possible due to the immediate update propagation scheme of AccuGraph and ForeGraph leading to convergence to the result in less iterations (insight 1). The iteration reduction of immediate update propagation is even more pronounced for WCC leading to a more pronounced performance advantage of AccuGraph and ForeGraph over HitGraph and ThunderGP for WCC when compared with BFS (cf. Fig. 8). Additionally, AccuGraph and ForeGraph read significantly less bytes per edge on average (cf. Fig. 9(b)). This is due to the CSR data structure of AccuGraph and the compressed edges for ForeGraph (insight 2). For AccuGraph the exact number of bytes depends on the density of the graph which we discuss in Sect. 4.3. ForeGraph always needs 4 bytes per edge, 2 bytes per vertex identifier, and some additional bytes to prefetch the value intervals. HitGraph and ThunderGP need 8 bytes for each edge plus prefetching values and reading updates.

We also measured performance on weighted graphs for HitGraph and ThunderGP on SSSP and SpMV. However, there were no significant differences in performance besides overall longer runtimes due to bigger edge size (because of edge weights) compared to BFS and PR respectively. Thus, we do exclude SSSP and SpMV plots for brevity (but show runtime measurements in the appendix Tab. 5).

4.3 Data Set Characteristics: Graph Properties

In this subsection, we discuss performance effects observable in Fig. 8 due to the graph properties (performance dimension (iii)) size ($|E|$), density ($D_{avg}$), and skewness of the degree distribution (as Pearson’s moment coefficient of skewness $E[(D-\mu)^3]$ with $D$ the degrees of the graph). The first trend we notice is AccuGraph and ForeGraph performance decreasing relative to HitGraph and ThunderGP for large graphs like r24 and tw. For the immediate update propagation scheme, destination vertex values need to be present when processing an edge which leads to loading these values many times instead of just once for update application (cf. Fig. 9(c)). Thus, immediate update propagation leads to disproportionately more value reads for large graphs (insight 3). Particular to AccuGraph, we see that performance is especially good for small graphs with only one partition ($|V| < 1,024,000$ for our configuration) such as sd, db, and bk. AccuGraph saves vertex value reads for these graphs with skipping the prefetch step because the values are already in on-chip memory (cf. Fig. 9(c)). However, for large graphs, AccuGraph still needs $n + 1$ CSR pointers for each partition leading to less savings in bytes per edge with horizontal partitioning (insight 4). HitGraph and ThunderGP performance is very similar in general. We only see a significant difference in performance for the tw graph due to ThunderGP reading many more values because of vertical partitioning scheme. HitGraph counteracts excessive value reads with an optimization described in Sect. 4.5.

All accelerator approaches benefit from dense graphs, with the effect being more pronounced for AccuGraph and ForeGraph (only working a full potential when $D_{avg} > 16$) due to a significant amount of pipeline stalls for sparse graphs. For accelerators with immediate update propagation and sparse graphs like db, yt, and rd, vertex value reads make up significantly more of the runtime (addition to insight 3). Additionally, AccuGraph performance suffers

per iteration (Fig. 9(c)) can especially play a role for sparse and large graphs. We only show BFS plots in Fig. 9 for brevity because the numbers are very similar for PR and WCC.

Overall, performance of PR is the highest because only one iteration is performed (cf. Fig. 8). BFS and WCC performance is overall similar. For BFS and WCC, performance on the bk and rd graph — which we break out in separate plots with a difference y-axis scale — is significantly lower. This, however, is expected as bk and rd are graphs with a large diameter and thus require many more iterations to complete. We notice that AccuGraph and ForeGraph on average perform better than HitGraph and ThunderGP. To explain this, we additionally notice that AccuGraph and ForeGraph finish in significantly less iterations over the graph for BFS and WCC (not shown) than HitGraph and ThunderGP relative to the graph’s diameter (cf. Fig. 9(a)). This is possible due to the immediate update propagation scheme of AccuGraph and ForeGraph leading to convergence to the result in less iterations (insight 1). The iteration reduction of immediate update propagation is even more pronounced for WCC leading to a more pronounced performance advantage of AccuGraph and ForeGraph over HitGraph and ThunderGP for WCC when compared with BFS (cf. Fig. 8). Additionally, AccuGraph and ForeGraph read significantly less bytes per edge on average (cf. Fig. 9(b)). This is due to the CSR data structure of AccuGraph and the compressed edges for ForeGraph (insight 2). For AccuGraph the exact number of bytes depends on the density of the graph which we discuss in Sect. 4.3. ForeGraph always needs 4 bytes per edge, 2 bytes per vertex identifier, and some additional bytes to prefetch the value intervals. HitGraph and ThunderGP need 8 bytes for each edge plus prefetching values and reading updates.

We also measured performance on weighted graphs for HitGraph and ThunderGP on SSSP and SpMV. However, there were no significant differences in performance besides overall longer runtimes due to bigger edge size (because of edge weights) compared to BFS and PR respectively. Thus, we do exclude SSSP and SpMV plots for brevity (but show runtime measurements in the appendix Tab. 5).

4.3 Data Set Characteristics: Graph Properties

In this subsection, we discuss performance effects observable in Fig. 8 due to the graph properties (performance dimension (iii)) size ($|E|$), density ($D_{avg}$), and skewness of the degree distribution (as Pearson’s moment coefficient of skewness $E[(D-\mu)^3]$ with $D$ the degrees of the graph). The first trend we notice is AccuGraph and ForeGraph performance decreasing relative to HitGraph and ThunderGP for large graphs like r24 and tw. For the immediate update propagation scheme, destination vertex values need to be present when processing an edge which leads to loading these values many times instead of just once for update application (cf. Fig. 9(c)). Thus, immediate update propagation leads to disproportionately more value reads for large graphs (insight 3). Particular to AccuGraph, we see that performance is especially good for small graphs with only one partition ($|V| < 1,024,000$ for our configuration) such as sd, db, and bk. AccuGraph saves vertex value reads for these graphs with skipping the prefetch step because the values are already in on-chip memory (cf. Fig. 9(c)). However, for large graphs, AccuGraph still needs $n + 1$ CSR pointers for each partition leading to less savings in bytes per edge with horizontal partitioning (insight 4). HitGraph and ThunderGP performance is very similar in general. We only see a significant difference in performance for the tw graph due to ThunderGP reading many more values because of vertical partitioning scheme. HitGraph counteracts excessive value reads with an optimization described in Sect. 4.5.

All accelerator approaches benefit from dense graphs, with the effect being more pronounced for AccuGraph and ForeGraph (only working a full potential when $D_{avg} > 16$) due to a significant amount of pipeline stalls for sparse graphs. For accelerators with immediate update propagation and sparse graphs like db, yt, and rd, vertex value reads make up significantly more of the runtime (addition to insight 3). Additionally, AccuGraph performance suffers

per iteration (Fig. 9(c)) can especially play a role for sparse and large graphs. We only show BFS plots in Fig. 9 for brevity because the numbers are very similar for PR and WCC.

Overall, performance of PR is the highest because only one iteration is performed (cf. Fig. 8). BFS and WCC performance is overall similar. For BFS and WCC, performance on the bk and rd graph — which we break out in separate plots with a difference y-axis scale — is significantly lower. This, however, is expected as bk and rd are graphs with a large diameter and thus require many more iterations to complete. We notice that AccuGraph and ForeGraph on average perform better than HitGraph and ThunderGP. To explain this, we additionally notice that AccuGraph and ForeGraph finish in significantly less iterations over the graph for BFS and WCC (not shown) than HitGraph and ThunderGP relative to the graph’s diameter (cf. Fig. 9(a)). This is possible due to the immediate update propagation scheme of AccuGraph and ForeGraph leading to convergence to the result in less iterations (insight 1). The iteration reduction of immediate update propagation is even more pronounced for WCC leading to a more pronounced performance advantage of AccuGraph and ForeGraph over HitGraph and ThunderGP for WCC when compared with BFS (cf. Fig. 8). Additionally, AccuGraph and ForeGraph read significantly less bytes per edge on average (cf. Fig. 9(b)). This is due to the CSR data structure of AccuGraph and the compressed edges for ForeGraph (insight 2). For AccuGraph the exact number of bytes depends on the density of the graph which we discuss in Sect. 4.3. ForeGraph always needs 4 bytes per edge, 2 bytes per vertex identifier, and some additional bytes to prefetch the value intervals. HitGraph and ThunderGP need 8 bytes for each edge plus prefetching values and reading updates.

We also measured performance on weighted graphs for HitGraph and ThunderGP on SSSP and SpMV. However, there were no significant differences in performance besides overall longer runtimes due to bigger edge size (because of edge weights) compared to BFS and PR respectively. Thus, we do exclude SSSP and SpMV plots for brevity (but show runtime measurements in the appendix Tab. 5).

4.3 Data Set Characteristics: Graph Properties

In this subsection, we discuss performance effects observable in Fig. 8 due to the graph properties (performance dimension (iii)) size ($|E|$), density ($D_{avg}$), and skewness of the degree distribution (as Pearson’s moment coefficient of skewness $E[(D-\mu)^3]$ with $D$ the degrees of the graph). The first trend we notice is AccuGraph and ForeGraph performance decreasing relative to HitGraph and ThunderGP for large graphs like r24 and tw. For the immediate update propagation scheme, destination vertex values need to be present when processing an edge which leads to loading these values many times instead of just once for update application (cf. Fig. 9(c)). Thus, immediate update propagation leads to disproportionately more value reads for large graphs (insight 3). Particular to AccuGraph, we see that performance is especially good for small graphs with only one partition ($|V| < 1,024,000$ for our configuration) such as sd, db, and bk. AccuGraph saves vertex value reads for these graphs with skipping the prefetch step because the values are already in on-chip memory (cf. Fig. 9(c)). However, for large graphs, AccuGraph still needs $n + 1$ CSR pointers for each partition leading to less savings in bytes per edge with horizontal partitioning (insight 4). HitGraph and ThunderGP performance is very similar in general. We only see a significant difference in performance for the tw graph due to ThunderGP reading many more values because of vertical partitioning scheme. HitGraph counteracts excessive value reads with an optimization described in Sect. 4.5.

All accelerator approaches benefit from dense graphs, with the effect being more pronounced for AccuGraph and ForeGraph (only working a full potential when $D_{avg} > 16$) due to a significant amount of pipeline stalls for sparse graphs.8 For accelerators with immediate update propagation and sparse graphs like db, yt, and rd, vertex value reads make up significantly more of the runtime (addition to insight 3). Additionally, AccuGraph performance suffers

8Detailed performance by average degree can be found in Fig. 13 in Appendix B
for sparse graphs because the ratio between pointers and neighbors in the CSR data structure is higher. We also observe performance differences by skewness of degree distribution.\(^9\) Performance for AccuGraph and ForeGraph drops for graphs with high skewness (e.g., wt and tw). The accelerators are only working at their full potential at low to moderate skewness. We also identify pipeline stalls caused by edge materialization on the CSR data structure as a problem for AccuGraph for high degree distribution skewness, similar to sparse graphs (insight 5). For ForeGraph we identify partition skew as the main cause of reduced performance which we discuss in more detail in Sect. 4.5 but can be observed in Fig. 9(d).

4.4 Memory Technology: DRAM Types

In Fig. 10(a), we show average speedup of DRAM types (DDR3 and HBM) over DDR4 for all four accelerators (performance dimension (iv)). We observe that modern memory (e.g., DDR4 or HBM) does not necessarily perform better than the older DDR3, despite higher theoretical throughput. This stems from lower bandwidth utilization and higher latency of requests (insight 6) which is explained by extremely low utilization of parallelism in the memory due to mostly sequential reading of very few data structures at once. AccuGraph and ForeGraph show more row hits due to write requests reusing rows in the row buffer of read requests (cf. Fig. 10(b)). To achieve very good bandwidth utilization, approaches have to utilize either even more locality (ForeGraph and AccuGraph) or more memory parallelism (AccuGraph with its CSR data structure). Additionally, there is no inherent benefit in using HBM for graph processing accelerators when the accelerator does not scale to multi-channel setups or scales poorly. The bandwidth utilization goes up slightly for HBM (cf. Fig. 10(b)) but at a cost of significantly more latency inducing row misses and conflicts due to HBM’s smaller row buffers. Thus, we conclude that HitGraph and ThunderGP have higher potential adapting to HBM.

Figure 11 shows multi-channel scalability of HitGraph and ThunderGP. AccuGraph and ForeGraph are not enabled for multi-channel operation and thus excluded. For measurements on more than one channel we assume the clock frequency (reported by the respective paper) achieved with the 4-channel designs for both HitGraph and ThunderGP for the simulation environment. However, as a limitation of these measurements this may not be exactly representative of the performance on real hardware because the clock speed could be slightly higher for two channels and lower for eight channels. For HitGraph, we see almost linear performance improvements when increasing the number of channels and see super-linear improvements for the roadnet-ca graph. This is due to improved effect of partition skipping resulting in significantly less requests to memory (insight 7). For ThunderGP, we see mostly sub-linear improvement in performance. We explain the ThunderGP performance with its vertical partitioning scheme which leads to every PE working on values from all vertices such that all updates have to be applied to all channels, limiting performance (insight 8). However, the scaling seems to benefit from dense input graphs like the orkut graph. An effect that we also observe is that DDR4 performance scales very well for two channels due to better bank parallelism utilization compared to DDR3 and HBM leading to better latency.

Another point we want to highlight is that HitGraph scales linearly in memory footprint with the number of memory channels and ThunderGP scales sub-linearly. HitGraph needs \( n + m + n \) while ThunderGP requires \( n + c + m + n + c \) space in memory where \( n \) is the size of the vertex value array, \( m \) is the size of the edge array and \( c \) is the number of channels. This not only means higher memory usage but also number of reads and writes not scaling linearly with number of channels for vertical partitioning (insight 9).

4.5 Memory Access Optimizations

Each accelerator proposes a set of optimizations to reduce load on the memory or partition skew (performance dimension (v)). In the following, we describe the different optimization approaches. Previous work [10] introduced the two optimizations prefetch (Pref.) and partition skipping (Skip.) for AccuGraph. Prefetching is skipped when the on-chip partition is equal to the to-be-prefetched partition. Partitions are skipped completely if none of their source values changed. Figure 12 shows a small improvement on the baseline at no significant added complexity. ForeGraph shows bigger performance improvements for its three optimizations. Edge shuffling (Shuf.) is a preprocessing step that repacks the edges such that the edge lists of \( p \) shards are zipped

\(^9\)Detailed performance by skewness can be found in Fig. 14 in Appendix B
into one (where $p$ is the number of PEs). This alone leads to reduced performance due to aggravated load imbalance with partitions (due to padding in the form of null edges) but improves PE utilization when combined with stride mapping (Map.). Shard skipping (Skip.) is employed for shards with unchanged source intervals compared to the previous iteration (equal to partition skipping for AccuGraph). Finally, stride mapping renames vertex intervals such that intervals are sets of vertices with a constant stride instead of consecutive vertices. In total, the optimizations improve performance for all graphs we tested on (cf. Fig. 12). However, we observe lower than average performance for sd, db, and rd. For those graphs, among others, the interval-shard partitioning introduces a lot of partition skew (especially in combination with edge shuffling) leading to many more edges read than necessary to process the graph (cf. Fig. 9(d)). Additionally, ForeGraph performance is higher for WCC on lk and rd when compared to BFS which is explained by less edges read due to more partition skipping (addition to insight 7).

HitGraph, like AccuGraph and ForeGraph, employs partition skipping (Skip.) with similar effectiveness. As a second optimization, HitGraph applies edge sorting by destination vertex (Sort), increasing locality to the gather phases value writing. The edge sorting for HitGraph prepares the data structure for update combining (Cmb.). Updates with the same destination vertex are combined into one in the shuffle phase which reduces the number of updates $a$ from $a = |E|$ to $a < |V| \times p$ with number of PEs $p$. As a second optimization to update generation, a bitmap with cardinality $n$ in BRAM saves for each vertex if its value was changed in the last iteration. This enables update filtering (Filt.) of updates from inactive vertices, saving a significant number of update writes.

ThunderGP proposes an offline scheduling of chunks to memory channels (Schd.) based on a heuristic predicting execution time. Chunks are greedily scheduled such that the overall predicted execution time is as similar as possible. Based on our measurements in Fig. 12 this however does not make a big difference. Additionally, ThunderGP does zero-degree vertex removal which we disabled for all runs because it is a pre-processing step applicable to all graph processing systems but hides performance ramifications of highly skewed degree distributions, e.g., for wiki-talk or rmat-21-86.

4.6 Discussion

In summary, from this comprehensive analysis of the four accelerators we gained nine insights that we categorize and group as trade-offs where possible in this section. Our biggest finding is a trade-off between lower iteration count of immediate update propagation for graph problems like BFS and WCC when compared to 2-phase update propagation (insight 1, similarly on CPU [21]), and reading vertex values many more times for large graphs (insight 3). As an open challenge we propose finding an approach to reduce vertex value reads for immediate update propagation (e.g., similar to update filtering for HitGraph) to lower the impact of graph size on the accelerator performance (open challenge (a)). As a second trade-off we found that CSR significantly reduces bytes per edge and values read for small and dense graphs (insight 2) at a trade-off of reading more bytes per edge and values for large and sparse graphs when using horizontal partitioning (insight 4). As a last trade-off we found that large partitions reduce partition overhead while small partitions can significantly benefit partition skipping leading to super-linear performance increases (insight 7). We noticed that high skewness in degree distribution can lead to performance degradation, e.g., for accelerators using CSR or interval-shard partitioning (insight 5). Additionally, vertical partitioning leads to poor channel scalability (insight 8) and memory footprint for multi-channel setups (insight 9). Regarding modern memory (e.g., HBM), we saw that trading of more latency with higher bank-level parallelism does not necessarily lead to better performance (insight 6, generally for modern memory [12]). Thus, we propose as an open challenge to investigate schemes to improve utilization of bank-level parallelism in modern memories (open challenge (b)). Lastly, we see and open challenge on enabling the immediate update propagation scheme for multi-channel (open challenge (c)).

5 RELATED WORK

In [12], the interactions of workloads and DRAM types is explored with Ramulator in great detail. They study CPU-based workloads but do not cover FPGA accelerators.

Accelerator simulation [27] introduces a simulation for HitGraph which generates the sequence of requests, but instead of simulating DRAM runtime, it assumes that every request results in a row buffer hit and models the performance along the cycles needed for processing the data and approximated pipelines stalls. They, however, do not show performance numbers generated with this simulation. [23] uses Ramulator as the underlying DRAM simulator for a custom cycle-accurate simulation of Graphicionado [13]. However, this incurs very high implementation time.

Modern memory technologies In line with our findings on HBM, Schmidt et al. [19] found that it is not trivial to attain good performance on another relatively new memory technology hybrid memory cube (HMC). [20] also confirms that it is crucial to consider how HBM should be used in FPGA-based accelerator designs.

Trade-offs for cloud graph processing Xu et al. benchmark different CPU-based graph processing systems in the cloud in [22]. For a limited set of graphs they study the performance of three systems. They find that GridGraph [29], which initially introduced the interval-shard partitioning, performs well, which we also observe for ForeGraph on single-channel systems.

6 CONCLUSION

This work addresses an important shortcoming of graph processing accelerators, namely comparability of graph processing performance. We approach this matter by extending the DRAM-based simulation environment proposed in [10] and compare the performance of four well-known graph accelerators (i.e., AccuGraph, ForeGraph, HitGraph, and ThunderGP) along performance dimensions relevant for graph processing (cf. dimensions (i)–(v)). We found performance effects based on accelerator design decisions (insights 5, 8, 9), issues in utilization of modern memory technologies (insight 6), and several interesting trade-offs (insights 1–4, 7).

We propose to conduct future work on the identified open challenges (a)–(c), i.e., further improving the immediate update propagation scheme for large graphs, leveraging the potential of HBM for graph processing, and multi-channel scalability of the immediate update propagation scheme. Additionally, we see a need for standardization of benchmark techniques in the field of graph processing accelerators, as sketched in this work.
A RAW DATA

Table 4: DDR4 (single-channel) runtime measurements (in seconds) with all optimizations enabled

| DDR4 | AccuGraph | ForeGraph | HitGraph | ThunderGP |
|------|-----------|-----------|----------|-----------|
| rd    | 0.0117    | 0.0155    | 0.0059   | 0.0111    |
| db    | 0.0117    | 0.0155    | 0.0059   | 0.0111    |
| yf    | 0.0121    | 0.0155    | 0.0059   | 0.0111    |
| pi    | 0.0121    | 0.0155    | 0.0059   | 0.0111    |
| tw    | 0.0121    | 0.0155    | 0.0059   | 0.0111    |
| lk    | 0.0121    | 0.0155    | 0.0059   | 0.0111    |
| or    | 0.0121    | 0.0155    | 0.0059   | 0.0111    |
| rj    | 0.0121    | 0.0155    | 0.0059   | 0.0111    |

Table 5: Weighted graph runtime measurements (in seconds) with all optimizations enabled on DDR4 (single-channel)

| Graph | SSSP | SpMV | SSSP | SpMV |
|-------|------|------|------|------|
| rd    | 0.0114 | 0.0012 | 0.0222 | 0.0012 |
| db    | 0.0459 | 0.0030 | 0.0469 | 0.0029 |
| yf    | 0.0848 | 0.0096 | 0.1271 | 0.0084 |
| pi    | 0.5014 | 0.0695 | 0.7501 | 0.0747 |
| tw    | 0.0750 | 0.0111 | 0.6868 | 0.0085 |
| or    | 1.8002 | 0.2639 | 2.2647 | 0.2821 |
| lj    | 1.0300 | 0.0964 | 1.3311 | 0.0884 |
| tw    | 18.6132 | 2.0955 | 32.4852 | 2.0255 |
| bk    | 5.2940 | 0.0994 | 5.6896 | 0.0998 |
| rd    | 5.3037 | 0.0105 | 5.1446 | 0.0885 |
| rj    | 1.4582 | 0.1904 | 1.9629 | 0.2173 |
| r24   | 3.2229 | 0.3124 | 5.0438 | 0.3355 |

Table 6: DDR3 and HBM (single-channel) runtime measurements (in seconds) with all optimizations enabled for BFS

| Graph | DDR3 | HBM | DDR3 | HBM | DDR3 | HBM |
|-------|------|-----|------|-----|------|-----|
| rd    | 0.0014 | 0.0017 | 0.0131 | 0.0157 | 0.0064 | 0.0090 |
| db    | 0.0094 | 0.0114 | 0.0221 | 0.0264 | 0.0273 | 0.0382 |
| yf    | 0.0200 | 0.0244 | 0.0274 | 0.0327 | 0.0526 | 0.0736 |
| pi    | 0.0970 | 0.1157 | 0.1101 | 0.1316 | 0.0275 | 0.0389 |
| tw    | 0.0241 | 0.0303 | 0.0269 | 0.0321 | 0.0484 | 0.0671 |
| or    | 0.3955 | 0.4708 | 0.3905 | 0.4668 | 0.9660 | 1.3605 |
| lj    | 0.2335 | 0.2867 | 0.3584 | 0.4282 | 0.6045 | 0.8461 |
| tw    | 9.0370 | 11.2454 | 17.9232 | 21.4115 | 11.4310 | 16.3588 |
| bk    | 1.3712 | 1.6510 | 4.2011 | 5.0245 | 2.9800 | 4.1829 |
| rd    | 1.1917 | 1.4289 | 6.6240 | 7.9176 | 3.1720 | 4.4374 |
| db    | 0.2651 | 0.3168 | 4.0602 | 4.8586 | 0.7626 | 1.0785 |
| rj    | 1.6698 | 2.2924 | 1.0779 | 1.2862 | 1.5798 | 2.4812 |

Table 7: Multi-channel scalability runtime measurements (in seconds) with all optimizations enabled for BFS

| DRAM | #Channels | HiGraph | ThunderGP |
|------|-----------|---------|------------|
| rd    | 0.0214    | 0.3640  | 0.3433 |
| db    | 0.0214    | 0.3640  | 0.3433 |
| yf    | 0.0405    | 0.2221  | 0.3515 |
| pi    | 0.0405    | 0.2221  | 0.3515 |
| tw    | 0.0405    | 0.2221  | 0.3515 |
| or    | 0.0405    | 0.2221  | 0.3515 |
| rj    | 0.0405    | 0.2221  | 0.3515 |

Table 8: Runtime measurements (in seconds) with different optimizations enabled for BFS

| Accelerator Optimization | Graph | db | or | rd |
|--------------------------|-------|----|----|----|
| Prefetch skipping        | AccuGraph | 0.0118 | 0.5071 | 1.3834 |
| Partition skipping       | ForeGraph | 0.0107 | 0.5071 | 1.3834 |
| None                     | ThunderGP | 0.0262 | 2.5996 | 16.6342 |
| Edge shuffling           | ForeGraph | 0.0936 | 5.5188 | 46.3402 |
| Shard skipping           | HitGraph | 0.0191 | 0.5149 | 4.3149 |
| Stride mapping           | HitGraph | 0.0268 | 0.4736 | 8.0324 |
| Update combining         | ThunderGP | 0.0149 | 0.4858 | 1.8349 |

Figure 13: Performance by average degree of the graph
Figure 14: Performance by skewness of degree distribution