Large-Scale Graph Processing on FPGAs with Caches for Thousands of Simultaneous Misses

Mikhail Asiatici
Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH–1015 Lausanne, Switzerland
mikhail.asiatici@epfl.ch

Paolo Ienne
Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH–1015 Lausanne, Switzerland
paolo.ienne@epfl.ch

***Abstract*—Efficient large-scale graph processing is crucial to many disciplines. Yet, while graph algorithms naturally expose massive parallelism opportunities, their performance is limited by the memory system because of irregular memory accesses. State-of-the-art FPGA graph processors, such as ForeGraph and FabGraph, address the memory issues by using scratchpads and regularly streaming edges from DRAM, but then they end up wasting bandwidth on unneeded data. Yet, where classic caches and scratchpads fail to deliver, FPGAs make powerful unorthodox solutions possible. In this paper, we resort to extreme nonblocking caches that handle tens of thousands of outstanding read misses. They significantly increase the ability of memory systems to coalesce multiple accelerator accesses into fewer DRAM memory requests; essentially, when latency is not the primary concern, they bring the advantages expected from a very large cache at a fraction of the cost. We prove our point with an adaptable graph accelerator running on Amazon AWS f1; our implementation takes into account all practical aspects of such a design, including the challenges involved when working with modern multidie FPGAs. Running classic algorithms (*PageRank*, *SCC*, and *SSSP*) on large graphs, we achieve 3× geometric mean speedup compared to state-of-the-art FPGA accelerators, 1.1–5.8× higher bandwidth efficiency and 3.0–15.3× better power efficiency than multicore CPUs, and we support much larger graphs than the state-of-the-art on GPUs.**

***Index Terms*—graph, MOMS, nonblocking cache, DRAM, FPGA**

I. Introduction

Graphs are the most effective data representation in a wealth of domains, including social networks [31], [42], drug discovery, [48], genomics [8], and robot navigation [7]; this makes the efficient processing of large graphs crucial in many disciplines.

While graph problems are usually embarrassingly parallel, the performance of graph algorithms on traditional platforms is in practice limited by the bandwidth of the memory system as either the edge or the node set are typically accessed irregularly [17], [18]. GPUs are also a poor fit for algorithms with irregular control flow and memory access patterns, and require some heavy preprocessing of the graph to achieve good performance [28], [50]. This is problematic on dynamic graphs or in the common scenario where graphs are generated by another application and are used no more than a few times [37]. ASICs provide excellent performance by customizing processing pipelines and the memory system to the access patterns typical of graph processing [24], [52]. However, their fabrication requires months and involves large NRE costs, especially if they are to be implemented on the advanced technology nodes used to evaluate their performance in simulation. While FPGAs cannot reach the performance of ASICs, they are now available in data centers [2], [41], [51]. Today, anyone can deploy immediately and for less than a dollar per hour an FPGA graph accelerator, even tightly integrated in a more complex pipeline directly in the cloud.

The accessibility and cost of FPGAs is now comparable to that of GPUs, while retaining the hardware flexibility of ASICs. This makes them an attractive platform to accelerate algorithms with divergent control and irregular memory accesses.

A. The Challenge of Irregular Accesses

While both vertex- [36] and edge-centric [43] approaches have been proposed on FPGA, the latter choice appears to be more common among recent solutions targeting large-scale graphs [15], [44]. Considering that the edge set is usually larger than the node set, streaming the edges indeed limits the range of the irregular memory accesses to the smallest one of those sets. And, if the entire node set fits in on-chip memory, random accesses to external memory are entirely eliminated. When this is not the case, sorting the edges by source and destination node turns node accesses to sequential; this, however, makes preprocessing more expensive and super linear in the number of edges. Since accesses are generally too irregular to make traditional caches effective (Fig. 1a), state-of-the-art FPGA accelerators for graph processing [15], [44] mitigate the problem by partitioning the node set in tiles (intervals) and accessing the node set in a tiled fashion (Fig. 1b). This only requires edges to be partitioned by source and destination interval, which has lower complexity than sorting. However, transferring nodes at the granularity of tiles may cause unnecessary data transfers as not all nodes are always accessed in every iteration (as suggested in Fig. 1b). In addition, the number of tile transfers between on- and off-chip memory is quadratic in the number of nodes, leading to even more redundant data transfers. As a result, node transfers dominate the total execution time if the node set is much larger than the amount of on-chip memory.

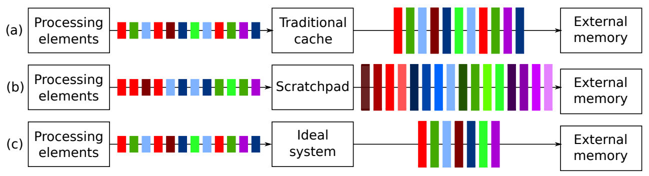

Fig. 1. Performance of memory systems when accesses are irregular. Cache lines belonging to the same tile when using scratchpads are identified by different shades of the same color. Traditional caches (a) are effective only when the reuse distance is short enough, which is rarely the case with large-scale graph workloads. Statically-managed scratchpads (b) need strictly ordered accesses and transfer efficiently tiles of data; thus, they guarantee that all accesses are hits, but usually also transfer data that are never used. An ideal, infinite cache (c) would request only useful cache lines and exactly once. Miss-optimized memory systems push caches in this direction for a reasonable area cost.

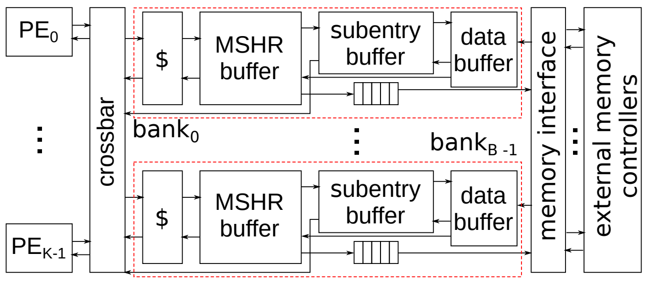

Fig. 2. Top-level architecture of a miss-optimized memory system (MOMS) [6]. Requests from K processing elements (PEs) are handled by B shared banks. Each bank consist of an optional cache and thousands of MSHRs and subentries to handle as many outstanding misses.

B. Optimizing Miss Handling, Not Maximizing Hits

We have shown in our previous work that a *miss-optimized memory system* (MOMS) [6] increases read bandwidth of DRAMs when accesses are irregular and applications are latency insensitive. MOMSes are based on the same insights behind nonblocking caches—that is, minimizing stalls and reusing each memory response to serve as many pending misses as possible. By scaling up the maximum number of outstanding misses from tens to tens of thousands, MOMSes maximize the opportunities for data reuse, bringing them closer to the ideal cache system (Fig. 1c) without needing unrealistically large cache arrays. The key insight is that, from a throughput perspective, secondary misses (or *MSHR hits*) are equivalent to cache hits in that both can be served without stalls nor extra memory requests, while requiring less on-chip memory. This makes MOMSes more area-efficient than traditional caches on latency-insensitive applications with irregular memory access patterns that expose thousands of outstanding reads. Last but not least, we have shown that such highly associative structures can be implemented effectively on FPGAs and our design is available open-source [4].

C. A Multi-Die FPGA Graph Accelerator

The main intuition of this paper is to use a MOMS to support the irregular read accesses typical of graph processing. Our insight is that the common skewed edge distribution results in many opportunities for request merging as some nodes are requested orders of magnitude more often than others; however, low-degree nodes are still common enough for traditional caches to stall too frequently and hurt throughput, whereas MOMSes tolerate a large number of misses much better. And, by taking advantage of the dynamic, fine-grained operation of MOMSes, we avoid the redundant data transfers typical of tiling. Because the MOMS only handles reads, we still buffer destination nodes in on-chip memory; however, the number of statically scheduled node transfers is now linear in the number of intervals rather than quadratic. This reduces the node transfer overhead for large graphs and, at the same time, lowers the amount of on-chip memory required without increasing the complexity of preprocessing beyond the linear cost of edge partitioning.

Our graph accelerator is adaptable to wide classes of graph algorithms. It is based on multiple out-of-order processing elements (PEs) each handling thousands of hardware threads—one per edge—in a simultaneous multithreading fashion to mask the latency of memory and MOMS. The system has been designed with modern multidie FPGAs in mind, as these pose extra design challenges which are essential to address. Such challenges also imposed modifications to the MOMS design, which has also been extended to implement private and two-level architectures that scale better on large FPGAs. Our graph accelerator with improved MOMS has been evaluated on the Xilinx UltraScale+ FPGAs available on the Amazon AWS f1 instances. This sets us apart from prior FPGA solutions which have been tested only in simulation [15], [44] and whose performance is unclear once connected to shells or to DRAM controllers physically constrained to separate dies. We achieved a 3.0× speedup compared to the state-of-the-art solutions on FPGA (FabGraph), 1.1–5.8× higher bandwidth- and 3.0–15.3× power-efficiency than those on CPUs (Ligra and GraphMat), and 4.7× geometric mean speedup on PageRank compared to GPUs (Gunrock).

II. Extending Miss-Optimized Memory Systems

Caches are known for being ineffective for graph processing. For example, the hit rate of L2 caches in a CPU during graph traversal is only 10% [9]. Nonblocking caches mitigate the impact of misses on throughput by temporarily storing both miss addresses and request IDs into *miss status holding registers* (MSHRs) instead of stalling until the data returns, increasing the number of concurrent memory operations and achieving higher *memory-level parallelism*. Misses are grouped by cache line: this way, each cache line is requested only once and, once its data returns, it will be used to serve all of its pending misses [20]. Therefore, MSHRs are implemented as a content-addressable memory, looked up (i) on every miss to determine whether the respective cache line has been already requested, and (ii) whenever a cache line returns from main memory, to identify its pending misses to serve. However, the number of MSHRs, and thus the maximum number of outstanding misses,

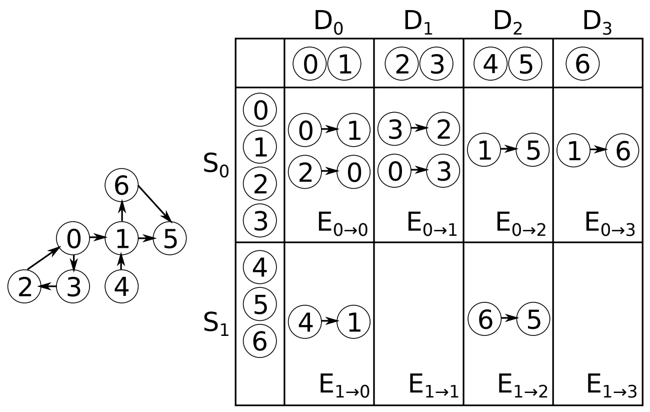

Fig. 3. Example of interval-based graph partitioning for the graph shown on the left, assuming N_s = 4 and N_d = 2. Edges are partitioned in shards $E_{s \to d}$ based on the respective source and destination node intervals.

Template 1 Programming Model

```
continue = true
iter = 0
while iter < max_iter and continue do
    active_srcs_next = {false}
    continue = false
    for $d \in [0, Q_d - 1]$ do                          ▷ In parallel across multiple PEs
        for all $i \in D_d$ do          ▷ Transferring $D_d$ from DRAM to BRAM
            $V_{BRAM}[i] = init(V_{const}[i], V_{DRAM,in}[i], const)$
        for all $s \in [0, Q_s - 1]$ do                          ▷ Streaming edges
            if active_srcs[s] then
                for all $e$ in $E_{s \to d}$ do
                    if $e_{src} \in D_d$ and use_local_src then
                        new = $gather(V_{BRAM}[e_{src}], V_{BRAM}[e_{dst}], e_w)$
                    else
                        new = $gather(V_{DRAM,in}[e_{src}], V_{BRAM}[e_{dst}], e_w)$
                    if new $\neq V_{BRAM}[e_{dst}]$ or always_active then
                        active_srcs_next[src_interval(d)] = true
                        continue = true
                    $V_{BRAM}[e_{dst}]$ = new
        for $i \in D_d$ do          ▷ Transferring $D_d$ from BRAM to DRAM
            $V_{DRAM,out}[i] = apply(V_{BRAM}[i])$
    active_srcs = active_srcs_next
```

is usually very limited: for example, Intel Sandy Bridge and Haswell can only support up to ten outstanding L1 misses [25]. While it has been shown that, on realistic processors, there is little benefit in pushing memory-level parallelism beyond this limit [35], [49], throughput-oriented accelerators that can easily emit thousands of outstanding reads may instead benefit from dramatically increasing the number of MSHRs [6].

The miss-optimized memory system, or MOMS, that we use in this paper is an extreme version of a multibanked nonblocking cache that supports tens of thousands of outstanding misses by storing MSHRs into ordinary RAM (abundant in FPGAs) and by using cuckoo hashing instead of fully associative lookup [6]. This significantly increases memory-level parallelism: the latency and the contention on the memory system is leveraged to maximize the reuse opportunities of in-flight cache lines without necessarily depending on their longer-term storage in the expensive data array of the cache. As a result, MOMSes introduce new points in the design space of nonblocking caches; FPGAs make it possible to explore and implement them easily. Many such points are Pareto-optimal and ideally suited for throughput-oriented accelerators.

Our prior work only considered MOMSes that are shared among multiple accelerators, as shown in Fig. 2. However, we found that MOMS bank conflicts were never considered and, in fact, severely limit throughput, especially on large systems. To tackle this bottleneck, we propose *private* and *two-level* MOMSes and we show in Section V that they generally outperform shared MOMSes. While MOMSes have been originally proposed to reduce the DRAM traffic, in a two-level MOMS we use private MOMSes to reduce traffic to the shared MOMS, which in turn reduces contention and improves throughput. In addition, while we previously designed and evaluated MOMSes only on a mid-range, single-die FPGA connected to a single DDR3 controller, in Section IV we describe the techniques we introduced to efficiently scale it up on a large, multidie FPGA connected to four DDR4 channels.

III. Graph Processing Model

A graph G consists of a node set V and an edge set E of sizes N and M respectively. We consider G to be a directed graph; undirected graphs can be easily handled by duplicating each edge. We consider graph algorithms that associate a value to every node and iteratively update them for a fixed number of iterations or until convergence.

A. Graph Partitioning

As described more in detail in Section III-B, we adopt an edge-centric model which iterates over the entire edge set and, in principle, may access both source and destination nodes in an arbitrary order. Interval-based partitioning is a common lightweight ($O(M)$) preprocessing technique that provides an arbitrary degree of locality to the accesses to the node set [32], [43]. Nodes are partitioned in Q disjoint intervals and edges into Q^2 *shards*, where shard $E_{i \to j}$ contains all the edges that have source and destination node in intervals S_i and D_j respectively. Shards can then be streamed while the respective source and destination intervals are both in on-chip memory, providing high performance irrespective of the access pattern.

Because we use a MOMS to avoid buffering source nodes in on-chip memory, we could, in principle, partition edges in Q shards based on the destination interval alone. However, we keep the source node partitioning for two reasons: (1) to avoid processing edges whose source interval does not contain any node that has been updated in the previous iteration and (2) to apply the edge compression mechanism introduced by ForeGraph [15] and described in Section III-C. Therefore, as shown in Fig. 3, we partition edges into $Q_s \times Q_d$ shards based on Q_s source and Q_d destination node intervals; such intervals can now have different sizes N_s and N_d since they serve different purposes.

TABLE I
EXAMPLES OF ALGORITHM-SPECIFIC PARAMETERS FOR TEMPLATE 1.

	PageRank	SCC	SSSP
$V_{const}[i]$	$OD[i]$	*not used*	*not used*
initial $V_{DRAM,in}[i]$	$\frac{0.15}{N \times OD[i]}$	i	$i = source \ ?\ 0 : \infty$
const	$\frac{0.85}{N}$	*not used*	*not used*
(v_c, v_{DRAM}, c)	(c, v_{const})	v_{DRAM}	
$gather(u, v, w)$	$v[0] + u$	$\min(u, v)$	$\min(u + w, v)$
$apply(v)$	$0.15 \times \frac{v[0]}{v[1]}$	v	v
use_local_src	false	true	true
always_active	true	false	false

B. Programming Model

Template 1 presents the programming model implemented by our accelerator. It represents an execution framework that can be configured to implement a variety of graph algorithms by customizing the functions $init()$, $gather()$, $apply()$, the initial node values $V_{DRAM,in}$, a per-node constant vector V_{const}, a global constant scalar const, and two control flags use_local_src and always_active. The model is based on the edge-centric Gather-Sum-Apply-Scatter (GAS) [23], [26] and generalizes the model used by ForeGraph [15] and FabGraph [44]. Table I shows three examples of graph algorithms implemented using our model. The main purpose of the $init()$ function, not present in the original GAS model, is to enable additional optimizations in some algorithms. For example, we can implement PageRank as in ForeGraph [15]: instead of reading both score (PR) and outdegree (OD) irregularly and recomputing the normalized score $d \times \frac{PR}{OD}$ for each source node, we read the constant OD sequentially once upon BRAM initialization and use it to normalize the score before sending it to DRAM. This reduces the size of each irregular read from 64 to 32 bits and allows to compute the normalized score only once per node; denormalizing the score has negligible overhead as it only requires sequential memory operations and can be done only once after the last iteration.

The model supports both synchronous and asynchronous execution, unlike ForeGraph [15] and FabGraph [44] that only support the latter. For synchronous execution, $V_{DRAM,in}$ and $V_{DRAM,out}$ are swapped after every iteration, meaning that the node values that are read during execution are updated only at the end of each iteration. For asynchronous execution, $V_{DRAM,in}$ and $V_{DRAM,out}$ point to the same array in memory: as a result, the $gather()$ function at line 15 will read updated values as soon as they appear in DRAM. If, in addition, (1) V_{BRAM} and $V_{DRAM,in}$ use the same format and (2) the algorithm remains correct even when $gather()$ uses partial node values, use_local_src can be set to true: whenever the source node is in the current destination group, it will be read from the local BRAM, using the most up-to-date version available in the system and reducing the traffic to DRAM. For the examples in Table I, SCC and SSSP satisfy both requirements while PageRank never satisfies (2) as partial scores in BRAM may underestimate the final score.

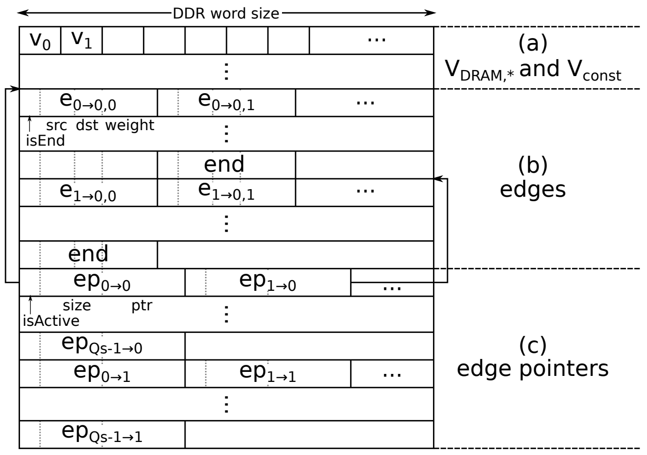

Fig. 4. Graph layout in memory, consisting of (i) node initialization values, (ii) edges in compressed format and organized by shard, and (iii) edge pointers.

C. Graph Encoding and Memory Layout

Our accelerator accepts graphs described in coordinate format (COO): a list of tuples (`src`, `dst`, `weight` (optional)), one per edge. Our preprocessing only requires edges to be partitioned according to their shard; this preprocessing has $O(M)$ complexity as opposed to other approaches [24], [52] that require edges to be sorted at least by source node (sometimes implicitly if graphs have to be converted to CSR format), which has $O(MlogM)$ complexity.

The entire memory layout is shown in Fig. 4. The first section contains the vertex arrays $V_{DRAM,in}$, V_{const} (if used by the algorithm) and allocates memory for $V_{DRAM,out}$ if the execution is synchronous. This is followed by all the edges organized by shard. Because the highest bits of source and destination nodes are implicit in the shard, each edge explicitly stores only the offsets (i.e., the lowest bits) within the respective source and destination groups. Since each word retrieved from the DRAM is usually wide enough to contain multiple edges, we append a special terminating edge at the end of each shard to ensure PEs will ignore any following data in the last DRAM word. It is indeed not possible for PEs to use an edge counter for this purpose as edges may return out-of-order from multiple DRAM channels (see Section IV-C). By using 15 bits for the destination node offset, 16 bits for the source node offset, and one bit for the `isTerminatingEdge` flag, we always use 32 bits per unweighted edge even for graphs that have tens of millions of nodes. This is similar to the edge compression technique used in ForeGraph and FabGraph [15], [44] except for the `isTerminatingEdge` flag. For weighted graphs, source and destination are followed by the edge weight. Because each shard may contain an arbitrary number of edges, we use an array of *edge pointers* to identify starting address and size of each shard, as well as the active_srcs flag for that shard (thus whether the shard should be streamed in at all or not as shown on line 10 of Template 1). All this fits into 64 bits.

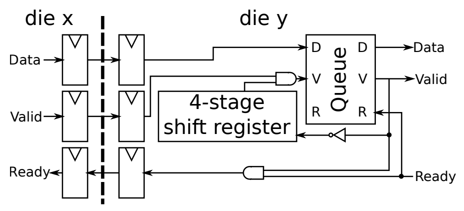

Fig. 5. Inter-die crossing circuit for signals with handshake. All the crossings are buffered on both ends with no combinational logic in between. Since it takes two cycles for the ready signal on die y to propagate to die x and, by that time, there may be up to two tokens in the crossing registers, the queue needs at least four slots to buffer all of them.

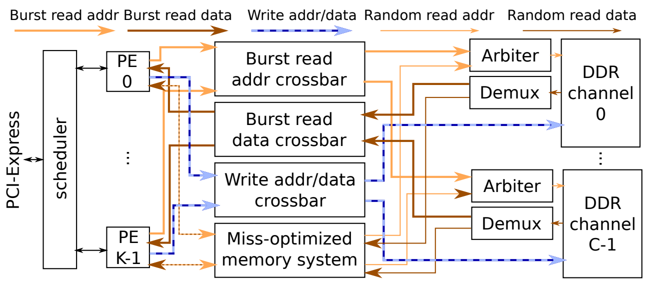

Fig. 6. Top-level system architecture. PEs pull jobs from the scheduler, which exposes a single job per destination interval. Burst reads and writes for node initialization, writeback, and edge streaming are forwarded to the respective memory channel, which are interleaved every 2,048 bytes. Irregular short reads to retrieve the source node values are handled by a MOMS.

IV. System Architecture

We first introduce the die crossing logic (Section IV-A) that we used in our multidie-aware top-level system (Section IV-B). In Section IV-C we then discuss the details of our generic out-of-order multithreaded processing elements. Finally, in Section IV-E we present the node reordering techniques that we considered in order to maximize both workload balancing among PEs and cache line reuse.

A. Die Crossing Logic

As high-performance FPGAs increasingly consist of multiple dies and FPGA boards offer multiple DRAM channels that are physically locked to a specific die, supporting multiple dies becomes crucial for fully exploiting all the available resources. Even though FPGA CAD tools expose multidie FPGAs as single devices, special care is needed to handle die crossings as they are particularly scarce and slow compared to intra-die interconnections. As a result, the presence of multiple dies must be taken into account early in the design process by (1) minimizing the number of inter-die connections and (2) making sure that all inter-die connections are registered on both ends and do not include any combinational components.. For inter-die connections that have handshake signals, we used the crossing logic shown in Fig. 5.

B. Top-Level Architecture

Fig. 6 shows the top-level system architecture. We target systems that comprise an FPGA connected to one or more

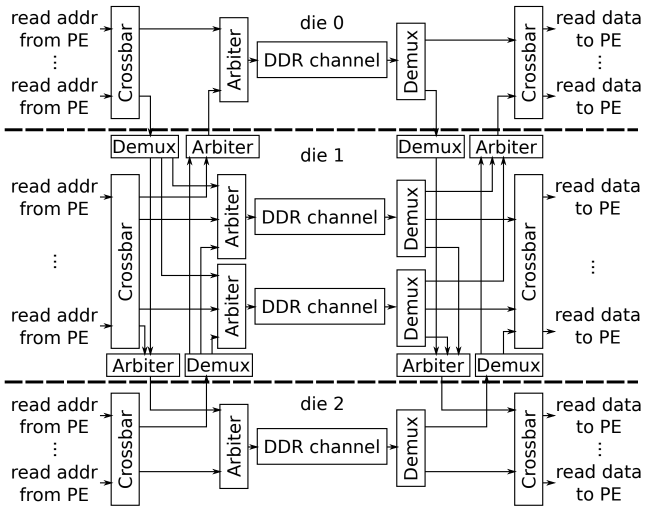

Fig. 7. Multidie-aware interconnect architecture for burst reads (the one for burst writes is analogous). Requests and responses are first routed to the target die and then to the target resource within the die.

external memory channels. In the latter case, we interleave the addresses of each channel every 2,048 bytes of global address space seen by the PEs to maximize aggregate bandwidth.

The scheduler contains memory-mapped registers that are used to transfer configuration parameters such as (1) the number of node groups and (2) the addresses of the node and edge pointers arrays shown in Fig. 4. The same interface is also used to start the accelerator and to notify its completion to the main processor. During execution, PEs pull jobs from the scheduler through an arbiter. Each job is associated to a node destination group and consists of (1) the base addresses in $V_{const}[i]$ and $V_{DRAM,in}$ for its node group (2) the base address in $V_{DRAM,out}$ to which the PE will write the final value of the node group, (3) the base address of the edge pointers for the node group, and (4) the index of the node group, used by the PE to notify the completion of the job.

In order to maximize resource utilization, PEs are scattered across multiple dies. Two distinct paths exist between PEs and DRAM controllers: a multidie-aware MOMS, used for the random short reads that retrieve the value of source nodes by dereferencing the edge source indices, and one for burst reads and writes, used in all the other transfers (initial destination node values, edge pointers, edges and final node values). The latter is, from a high-level perspective, equivalent to the bus interconnect logic commonly provided by IP vendors except for being platform independent and multidie-aware. This interconnect logic is made multidie-aware by (1) splitting each of the three crossbars (for read address, read data, and write address/data) into a first crossbar per die that routes transactions to the appropriate die and (2) a set of arbiters per die that forward transactions to the appropriate resource in the same die (DRAM controller or PEs for requests or responses respectively) as shown in Fig. 7.

Our multidie-aware MOMS has two differences compared

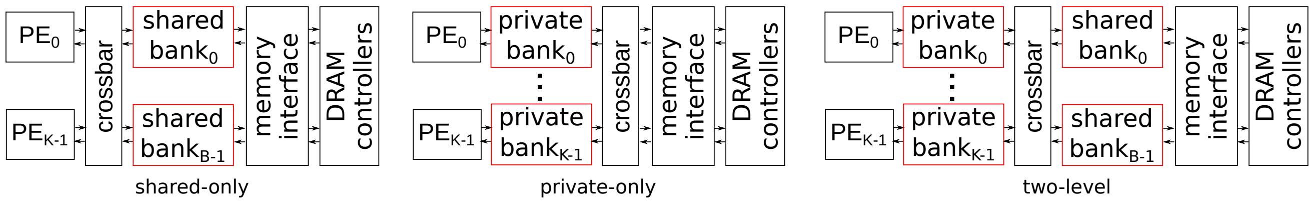

Fig. 8. Overview of the MOMS architectures that we considered. In addition to the shared-only MOMS proposed in our prior work [6], we now evaluate also private-only and two-level MOMSes.

to our original MOMS [6]. Firstly, connections to and from the MOMS crossbars (which forward requests from PEs to multiple parallel banks and responses in the opposite direction) use the crossing logic shown in Fig. 5. Secondly, each bank is statically allocated to a single DRAM channel: if each bank could target any DRAM channel, an extra crossbar would have been required between banks and DRAM controllers. This also allows us to assign banks to the same die as the respective DRAM controller, which further reduces the number of die crossings.

In addition to the shared MOMS architecture proposed previously and discussed above, we also consider architectures that (1) only have MOMSes that are private to each PE and (2) cascade private MOMS banks and a single shared MOMS, like a two-level cache. All architectures are summarized in Fig. 8. Unlike the shared MOMS, private MOMS banks can be accessed by each PE without contention; however, they may increase the overall traffic to DRAM as no inter-PE request coalescing is performed. Two-level architectures combine the best of both approaches and we will show in Section V that, despite the increased circuit complexity which may decrease parallelism and frequency, they provide the highest performance in most scenarios.

C. PE Architecture

Fig. 9 shows the internal structure of a PE. Each PE contains a DMA unit that handles all the sequential data transfers: node initialization, edge pointer retrieval, edge streaming (DRAM to PE), and node writeback (PE to DRAM). Upon acceptance of a job, the PE will first read the initial value of all the nodes in the destination group. To minimize the initialization time and because DRAM controllers often expose ports that are much wider than node values, we write four node values per cycle. Once the node initialization is completed, the PE requests edge pointers and, if the respective source group is active, the PE will start requesting edges from that group.

For each received edge, the source node value will be either retrieved from DRAM through the MOMS or from the local BRAM, if use_local_src is active and the source node is in the current destination interval. Once the source node value is provided by either the MOMS or the BRAM, it is sent to the $gather()$ pipeline together with the edge state. Because writing the output of the $gather()$ pipeline is a read-modify-write operation on the destination node, we use forwarding (whenever possible) or stalling logic to ensure that the $gather()$ pipeline always receives the latest version of the destination node value. For algorithms where always_active is false (such as SCC and SSSP), the $gather()$ pipeline also returns an `updated` flag, which is set whenever a destination node has been updated and is used to implement line 16 of Template 1.

Once all the edges have been streamed, the destination node memory is written back to DRAM and the PE notifies the completion of the job to the scheduler together with the destination group's `updated` flag, if it exists.

D. Handling Efficiently Out-of-Order Responses

When the data is interleaved across multiple DRAM channels, responses may return out-of-order whenever they hit multiple channels, even when each individual channel responds in-order. Since nodes must be initialized in a specific order, to prevent out-of-order responses and avoid expensive burst reordering, the PE will never issue more than one outstanding read burst for initial node values. We found this not to be an issue if we use a 64-entry, 512-bit wide DMA queue and issue the next 32-beat burst as soon as the queue has enough space to hold it. Bursts for edges, on the other hand, may be shorter than 32 beats as the number of edges in a shard is not necessarily a multiple of the number of edges in a 32-beat burst and we found that limiting each PE to a single outstanding request for edges results too frequently in an empty read queue. However, unlike node initial values, edges may be streamed out-of-order, provided that each burst is paired to the corresponding source group as, in our compressed edge format, it defines the high bits of the source node. Therefore, we tag each edge burst request with an ID that is unique for each source group and use the ID that returns with the edge data to stream the right source edge prefix to the downstream logic.

To maximize both MLP and the effectiveness of the MOMS [6] when requesting source nodes from DRAM through the MOMS, the PE must send thousands of outstanding reads to the MOMS. Since each edge can be processed independently, we treat them as separate threads: when the source node data request is sent to the MOMS, we store the thread state (destination node offset and edge weight) and suspend it; when a response returns, we retrieve the respective state and resume the thread execution. This mechanism is implemented by the MOMS interface and the MOMS itself as shown in Fig. 10. For weighted graphs, we tag each request with a unique ID of size ID_{size} provided by the free ID queue, which we also use to store and retrieve destination node offset and edge weight in

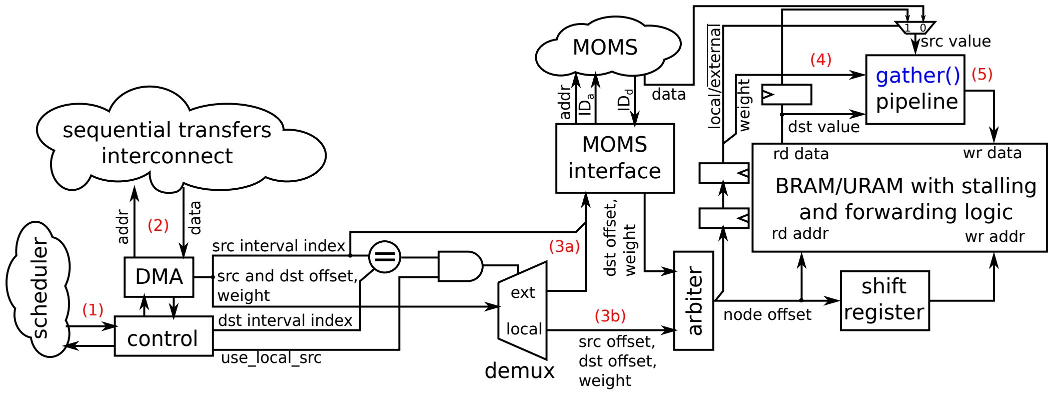

Fig. 9. Architecture of a PE. After obtaining a job (1), edges are fetched by the DMA (2) by dereferencing the active edge pointers. Source node values are fetched through the MOMS (3a) unless use_local_src (see Template 1) is enabled (3b). The MOMS interface, shown in Fig. 10, stores the state associated to each edge while waiting for responses. Once available, node values and edge weight are forwarded to the *gather*() pipeline (4) and the destination node value is updated in BRAM (5). The logic that handles node initialization and writeback is not shown but simply implements direct connections between DMA and BRAM through *init*() and *apply*() respectively.

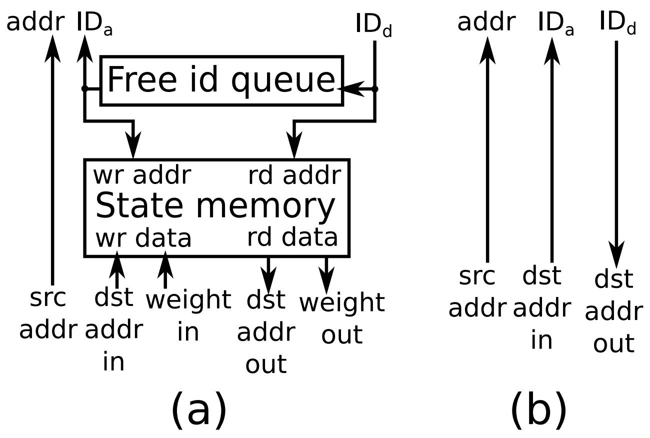

Fig. 10. Available MOMS interfaces, which are responsible for retrieving the state associated to each out-of-order MOMS response. For weighted graphs we use the architecture (a), which uses a queue to keep track of the available ids and which stores, for each id, the destination node offset and edge weight in the state memory BRAM. For unweighted graphs, the state reduces to the destination node offset. Considering that its size is comparable to that of the unique ids produced by the architecture (a), we implement the optimized interface (b) that uses the destination node offset directly as an id without duplicating information that is anyway stored in the MOMS.

the state memory. The BRAM cost per thread of this solution is ID_{size} bits in the free ID queue, ID_{size} bits in the MOMS subentry buffer [6], and size of destination node offset and edge weight in the state memory. For unweighted graphs, the state reduces to the 15-bit destination node offset, whose size is comparable to ID_{size} (as we target thousands of simultaneous threads). Therefore, pairing each destination offset to a unique ID using the circuit in Fig. 10a would require approximately $3 \times ID_{size}$ bits. We lower this size to ID_{size} bits per thread by using directly the destination offset as an ID, as shown in Fig. 10b. In other words, we use the MOMS itself to store the entire edge state. By doing so, in addition, the maximum number of threads is only limited by the MOMS capacity instead of the smallest between the capacities of the MOMS and of the state memory.

E. Node Reordering

Graphs described in coordinate format implicitly assign a unique integer label to each node, which also defines the address of the node value in memory. However, this labeling is, in principle, arbitrary, and while it does not affect algorithm correctness, it has a dramatic impact on performance. Because node values are usually smaller than a cache line, placing nodes that are tightly connected with each other close in the memory space improves the cache hit rate [19] or, in the case of MOMS, the opportunities for memory response reuse. Faldu et al. [19] indeed showed that, in many graph benchmarks, labeling preserves tight clusters.

On the other hand, jobs corresponding to different destination intervals may be processed in parallel; therefore, it is desirable to arrange nodes in destination intervals in such a way that the number of in-edges per interval is as balanced as possible. With respect to Fig. 3, this means distributing edges as uniformly as possible among columns D_i. Note that, to use bursts to transfer destination intervals between on- and off-chip memory, nodes belonging to the same destination interval should be contiguous in memory as in Fig. 3. Both ForeGraph [15] and FabGraph [44] statically schedule intervals to PEs and read source nodes from on-chip scratchpads that are private to each PE. Therefore, the workload balancing among PEs is very critical. Because of cluster preservation, paired with the common power-law degree distribution, they found that placing consecutive nodes in the same interval—i.e., computing the destination interval of node i as $n_{i,d} = \left\lfloor \frac{n_i}{N_d} \right\rfloor$—results in a very skewed workload distribution. Therefore, they both propose to use a *hash-based* relabeling such that node i is placed in interval $n_{i,d} = n_i \bmod Q_d$ instead, which results in a more uniform workload distribution.

In our case, however, the PE-level balancing is less critical as jobs are dynamically scheduled to PEs and 1–2 orders of magnitude more numerous than PEs. By letting each PE pull a new job whenever idle instead of forcing each of them to process the same number of jobs, we found that our N_{PE} PEs generally achieve a good workload balance even without hash-based relabeling as long as the largest jobs are smaller than $\frac{M}{N_{PE}}$. In contrast, maximizing cache line reuse becomes more critical. In particular, hash-based partitioning may destroy any cluster that is preserved in the original labeling [19], hurting cache line reuse. Therefore, we keep cache lines as they are and hash entire cache lines among destination intervals.

Orthogonally to cache line reordering, we also evaluate a technique introduced by Faldu et al. called *DBG reordering* [19] prior to cache line hashing to handle graphs whose initial labeling does not preserve tightly connected communities. DBG coarsely partition nodes in 8 groups according to their out-degree, following the intuition that clustering nodes with high out-degree together will lead to higher cache line reuse. This has $O(N)$ complexity, which, for most graphs, is even lower than the $O(M)$ cost of partitioning and cache line reordering. We evaluate the cost and benefit of these techniques in Section V-C.

V. Experimental Results

After presenting our experimental setup, FPGA-specific die assignments, and benchmarks in Section V-A, we analyze the impact of the number of PEs and of different MOMS architectures on PageRank, SCC, and SSSP in Section V-B. We then analyze the impact and cost of the various preprocessing techiques (Section V-C), the impact of number of memory channels, and thus bandwidth, on performance (Section V-D), and assess the contribution of the cache arrays to the measured throughput (Section V-E). In Section V-F we compare our performance with the state-of-the-art on CPUs, GPUs, and FPGAs and we conclude in Section V-G by presenting the resource utilization and operating frequency of our designs.

A. Experimental Setup

The system has been written in RTL using Chisel 3 and synthesized using Vivado 2019.1. The code is fully parametric in terms of number of PEs and memory channels and their distribution on the different dies, as well as node size and type, $init()$, $gather()$, and $apply()$ functions, MOMS organization, and many other dimensions. Our evaluation has been performed on the Amazon AWS f1 instances, which feature a Virtex UltraScale+ FPGA connected to the host PC via PCI express and to four 16 GB DDR4 channels, all through a placed-and-routed proprietary shell. Each DDR4 channel has a theoretical bandwidth of 16 GB/s; however, the shell seems to be optimized for bursts and the maximum bandwidth that we measured was only about 8 GB/s per channel if only single requests are used. We tried using a DynaBurst MOMS [5] that can send bursts of requests to memory but we found the benefit to be too low to compensate for the corresponding area and delay increase.

Our target FPGA spans over three dies (*SLR*s in Xilinx terminology), with 25–35 % of the resources of the bottom and central SLR reserved for the shell. The central SLR hosts two memory controllers while the other two SLRs have one controller each. We assign the shared MOMS crossbar to the central SLR and each bank to the respective memory channel's SLR. We found that assigning 30%, 15%, and 55% of the PEs to the bottom, central, and top SLR respectively provide a good area balancing. Private MOMS, when existing, are assigned to the same SLR as the respective PE.

Each PE holds 32,768 destination nodes in URAM; each node requires 32 bits in SCC and SSSP and 64 bits in PageRank. For the PageRank PEs, which operate on single-precision floating point, we implemented the $gather()$ and $apply()$ functions in Table I using Vivado HLS. Because its $gather()$ pipeline has a 4-cycle latency, it may have to be stalled to handle RAW hazards. The $gather()$ functions of SCC and SSSP, which operate on 32-bit unsigned integers, are implemented in Chisel and fully combinational, meaning that no stalls are required. The state memory and the free ID queue of the SSSP PEs have 8,192 slots and are implemented in BRAM. We run PageRank for 10 iterations and the other algorithms until convergence.

As benchmarks, we used a set of real world and synthetic large graphs, whose main properties are summarized in Table II. For SSSP, we added random integer weights between 0 and 255 [52]. If not specified, we enable both hashing and DBG.

TABLE II
Benchmarks properties.

	Benchmark	N	M
WT	wiki-Talk [34]	2.39M	5.02M
DB	dbpedia-link [29]	18.3M	172M
UK	uk-2005 [10], [16]	39.5M	936M
IT	it-2004 [10], [16]	41.3M	1.15B
SK	sk-2005 [10], [16]	50.6M	1.95B
MP	twitter_mpi [11], [29]	52.6M	1.96B
RV	twitter_rv [31]	61.6M	1.47B
FR	com-friendster [34], [53]	65.6M	1.81B
WB	webbase-2001 [10], [16]	118M	1.02B
24	RMAT-24 [12], [27]	16.8M	268M
25	RMAT-25 [12], [27]	33.6M	537M
26	RMAT-26 [12], [27]	67.1M	1.07B

B. Architecture Exploration

We performed an extensive design space exploration present the most significant design points in Fig. 11. We set a target frequency of 250 MHz and discard systems that run at less than 185 MHz. Each cache bank contains 256 kB of direct-mapped cache, 4,096 MSHRs, and 32,768 subentries. While MSHRs are implemented in BRAM as in our original MOMS [6], cache arrays and subentry buffers are deep enough to efficiently use the more abundant URAM. Private MOMSes have 4,096 MSHRs and 49,152 subentries and have an output data width of 64 bits when part of a two-level MOMS: higher widths dramatically increase the number of inter-die crossing and thus routing congestion, resulting in longer critical paths or routing failures. Private MOMSes have 256 kB of 4-way set-associative cache when there is no shared MOMS; in two-level architectures, we increased the private cache as much as possible until timing degradation. Still, many two-level

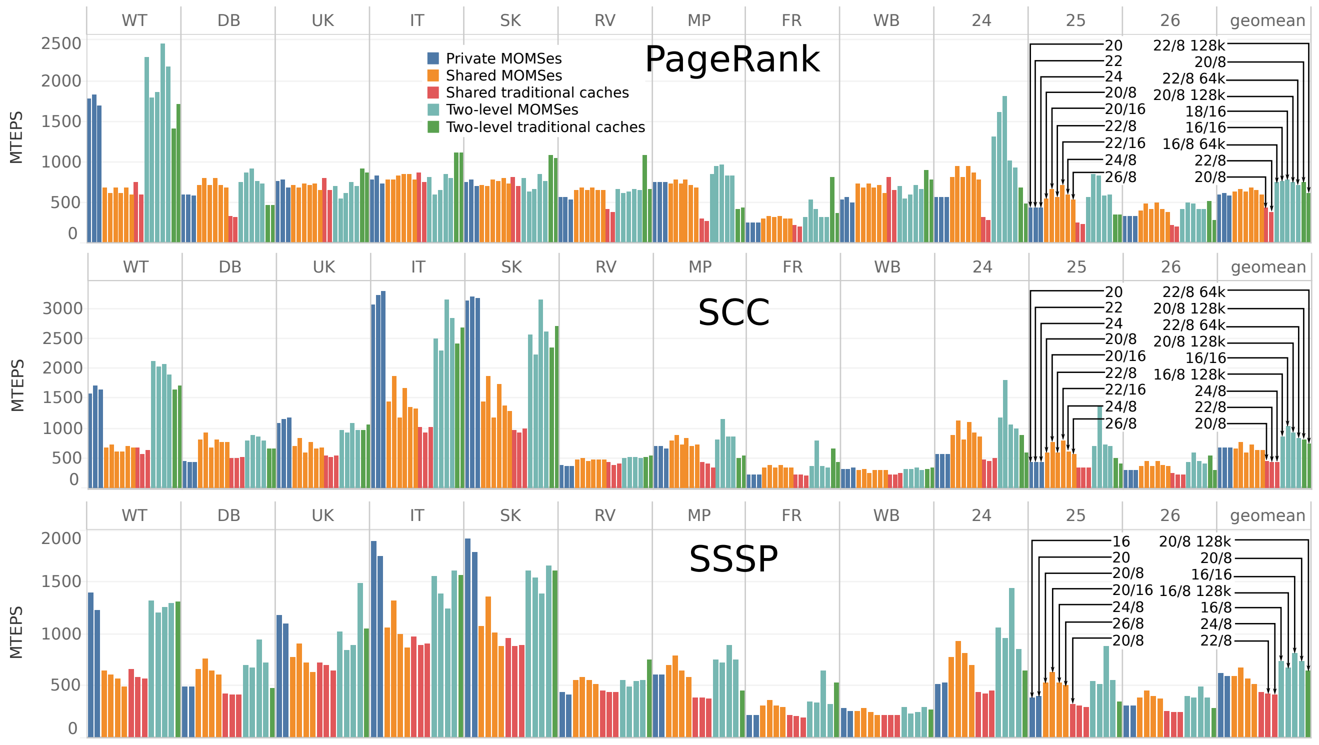

Fig. 11. Throughput on PageRank, SCC, and SSSP for different architectures. For shared and two-level architectures, the label **X/Y Zk** indicates X PEs and Y MOMS banks with Z kB of private cache. The two-level architectures with 16 banks generally provide the highest performance, balancing PE peak throughput, amount of conflicts in the shared MOMS, memory efficiency, and routing congestion.

MOMSes have no private cache at all since they bring little benefit to most benchmarks. Traditional caches have 16 MSHRs and 8 subentries per MSHR per private cache and per shared bank—more associative MSHRs lower the maximum frequency for no performance gain.

As shown in Fig. 11, two-level architectures provide the highest performance in geometric mean as fewer requests reach the shared MOMS, leading to fewer conflicts, while providing reuse among multiple PE without extra memory requests. The architectures with 16 shared banks even without private cache generally outperform those with more PEs and 8 banks, suggesting that inter-PE conflicts remain critical. This is confirmed by the poor performance of shared MOMSes, which cannot benefit from some filtering from the private MOMS. In contrast, by reducing bank conflicts, private and two-level MOMSes provide up to 3× higher throughput than our prior shared MOMSes [6]. Private MOMSes alone tend to be limited by the excessive amount of redundant requests. There are however important exceptions: IT, SK, and, to a lesser extent, UK and WT perform well also without shared MOMSes as they have higher locality and the benefit from the elimination of conflicts is higher than the increase in memory traffic. They also benefit from traditional two-level caches whenever they leave space for more PEs than two-level MOMSes or run at higher frequency.

SCC achieves the highest throughput among all the applications. PageRank is throttled by RAW stalls due to the 4-cycle *gather*() pipeline, especially frequent on IT, SK, UK, and WB. For SSSP, (1) the overhead of state memory and free ID queue reduces parallelism and operating frequency and (2) weighted edges consume twice the bandwidth than unweighted ones.

Fig. 12 shows the throughput on SCC as a function of the cache hit rate (in either cache level) for the architectures shown in Fig. 11. While traditional architectures need high hit rates to reach their peak performance, MOMSes often outperform them despite the lower hit rate, suggesting that cache arrays are less critical in MOMSes. To validate this hypothesis, we also considered the same systems with all the cache arrays deactivated and hence always achieving a 0% hit rate. While traditional caches naturally lose practically all of their performance, MOMSes have little throughput degradation on most benchmarks, meaning that thousands of MSHRs can essentially replace cache arrays at a fraction of the area cost when latency is irrelevant.

C. Preprocessing Cost and Impact

Fig. 13 shows the PageRank performance on the 18/16 MOMS depending on the preprocessing technique used (trends are similar on the other applications). Most benchmarks benefit from hashing, especially those with fewer nodes, which results in fewer jobs and a more critical load balancing. The higher speedup without hashing suggests that, in those cases, grouping the most connected nodes in the same destination intervals and

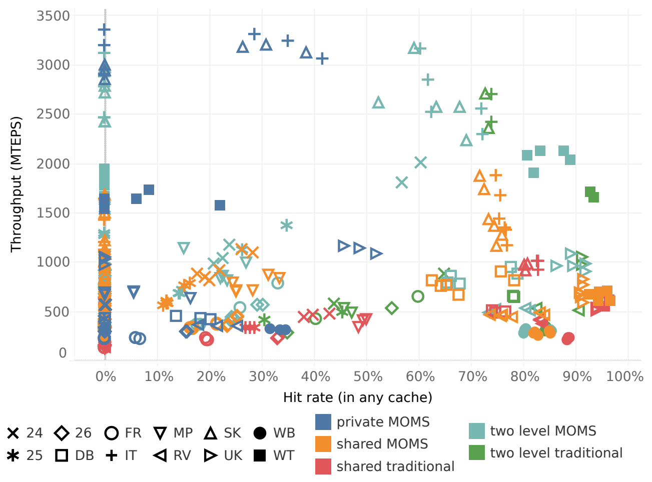

Fig. 12. Throughput on SCC versus cache hit rate for the architectures shown in Fig. 11, including the same architectures completely without cache arrays (with only MSHRs and subentries). Compared to traditional caches, MOMSes achieve higher performance at lower (or even zero) cache hit rate, meaning that cache arrays can be made smaller or even removed altogether with essentially no performance penalty.

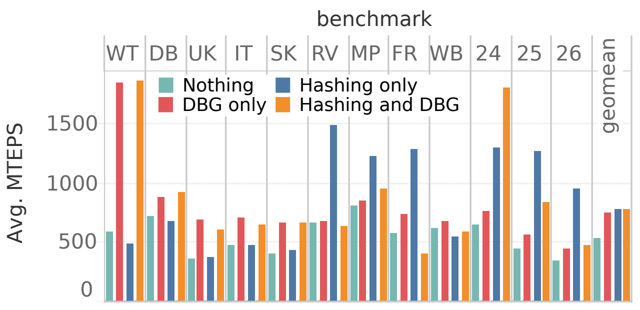

Fig. 13. PageRank throughput on the 18/16 two-level MOMS architecture depending on the preprocessing used.

thus processing their numerous edges one after another is more important than having uniform job size. In addition, when the labeling does not preserve the original graph communities (FR, MP, RV, and the RMATs) using DBG [19]—thus "breaking" the initial cache lines—provides a significant speedup.

Table III shows the preprocessing time of all the benchmarks on a 20-core Intel Xeon E5-2698 excluding disk I/O. We use OpenMP to parallelize most of the operations. Our preprocessing is generally lightweight and all the steps besides partitioning are optional, allowing to trade preprocessing time for runtime efficiency or to quickly explore the preprocessing design space to maximize the performance for a given application.

D. Memory Bandwidth Scalability

Fig. 14 shows the throughput as a function of the number of DDR4 channels, for the two-level 16/16 MOMS and for PageRank on FabGraph. We used the theoretical model described by Equations (2) to (7) in the FabGraph paper [44] to estimate its performance considering that edges are always active; we compute for it a very optimistic estimation that uses

TABLE III
PREPROCESSING TIME IN SECONDS.

	partitioning	hashing	DBG
WT	0.04	0.05	0.03
DB	0.94	1.17	0.31
UK	2.68	6.65	0.99
IT	3.78	6.36	1.32
SK	6.71	12.0	3.40
RV	12.7	10.4	2.76
MP	15.5	14.4	2.55
FR	20.7	18.0	3.06
WB	4.48	4.99	1.43
24	1.58	1.91	0.56
25	3.16	4.66	0.65
26	8.40	7.97	1.58

the ideal DRAM bandwidth of 16 GB/s per channel, ignoring the 50% bandwidth limitation imposed by the AWS shell on single request, any other implementation difficulties related to multidie design, and the handling of RAW conflicts of a floating-point PageRank implementation (FabGraph implements PageRank using integers and thus the initiation interval of its key pipeline is 1 instead of the realistic 4). We can identify two categories of benchmarks: (1) compute-bound (IT, SK, UK, WB, and WT) and (2) memory-bound (all the others). Benchmarks of the first category have good locality and need less than four channels to achieve peak performance, being rather limited by PE parallelism and, in PageRank, RAW conflicts that occur because our *gather*() function has a 4-cycle latency. These are indeed the benchmarks that benefit the most from private MOMSes or traditional caches in Fig. 11. On PageRank and SSSP, some compute-bound benchmarks even decrease their performance on 4-channel systems as they operate at lower frequency due to the higher number of SLR crossings resulting from the use of all SLRs. The performance of the memory-bound benchmarks, instead, scales essentially linearly with the memory bandwidth. In geometric mean, FabGraph performs better than our system on one memory channel but scales less than ideally because the performance becomes more and more limited by the internal bandwidth between their L1 and L2 cache, between which transfers are particularly numerous on large graphs. In addition, being a simulation-only analysis, it does not take into account how SLRs affect routing congestion.

E. Impact of Caches

Fig. 12 already showed that MOMSes do not need high cache hit rates to achieve peak performance and that cache-less MOMSes are often more competitive than traditional caches. Fig. 15 shows more in detail the impact on performance of adding the cache arrays and/or dramatically scaling up the MSHR array by comparing the SCC throughput of a two-level 20/8 MOMS and traditional cache, with and without 2.5 MiB and 2 MiB of private and shared cache respectively. While the traditional cache has a 2.2× throughput decrease without the cache array, the performance drop for the MOMS is only 10% (both in geometric mean), meaning that MSHRs can in practice replace the cache array with little difference in terms

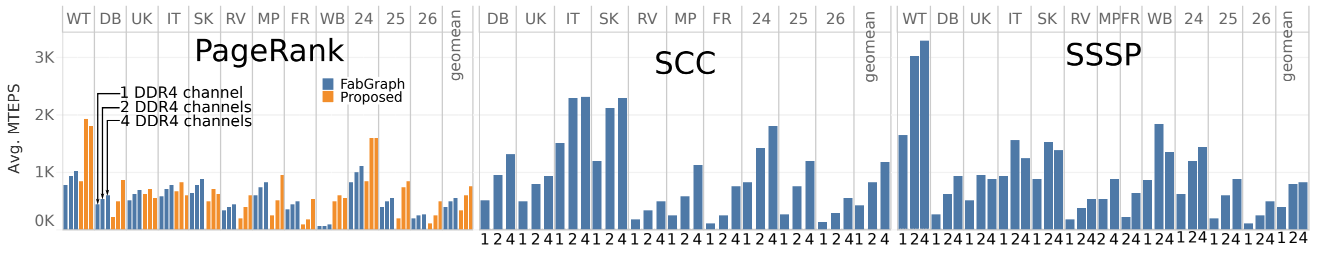

Fig. 14. Scalability of throughput as a function of the number of DDR4 channels for the two-level 16/16 MOMS architecture and for PageRank on FabGraph. The 4-channel PageRank and SSSP system have lower operating frequency than the 2-channel ones due to the higher number of SLR crossings, which increase congestion. For SCC, where the frequency is constant, the throughput of our system generally scales linearly with the available memory bandwidth except for the benchmarks that become compute-bound already with two memory channels (IT, SK, UK, WB, and WT). Note that one 16 GB channel does not have enough memory to run SSSP on FR and MP. Note also that FabGraph numbers are optimistic estimations based on the theoretical bandwidth and disregarding any SLR-related implementation issues and RAW conflicts.

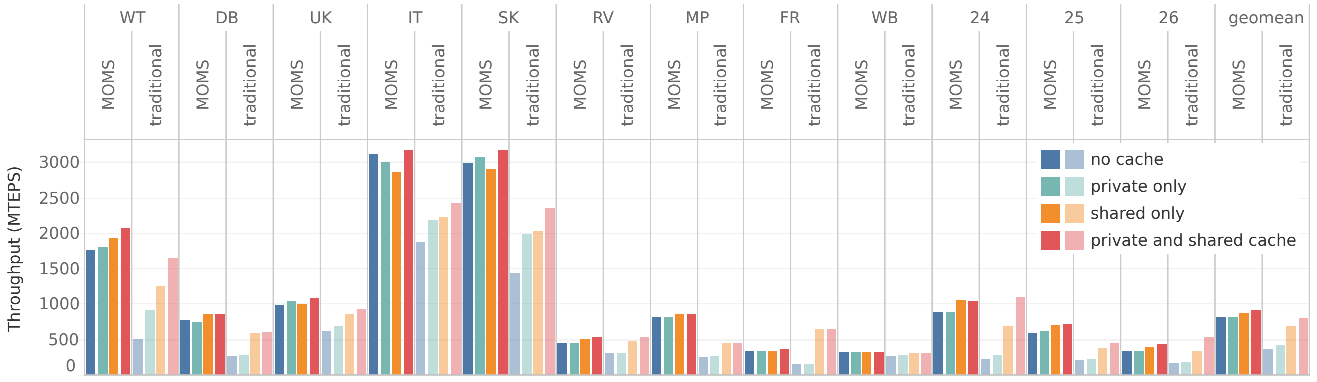

Fig. 15. Throughput on SCC for the 20/8 two-level MOMS and traditional cache, with and without private and/or shared cache. As in the 16/16 two-level MOMS, the contribution of cache arrays to the MOMS throughput is essentially negligible, especially for the private cache.

of throughput. Indeed, the cache-less MOMS has essentially the same performance as the full traditional cache despite using 25% fewer memory bits. When present, shared caches are generally more useful than private ones except for IT and SK which posses significant private reuse opportunities. In fact, on those benchmarks, the cache-less MOMS performs better than the MOMS with only private or shared cache arrays. When there is a shared cache array, shared hits will have much lower latency than misses, which gives less time to the private MOMS to accumulate secondary misses. As a result, the private MOMS needs to handle more responses, which slows down the throughput of requests as both compete for the same pipeline [6]. On the other hand, adding a private cache array introduces a point of contention between hit and miss data from cache and subentry buffer respectively, just before the MOMS response output. This bottleneck does not exist when the cache array is absent and all responses return from the miss path; however, except for IT, it is generally shadowed by other bottlenecks (memory, shared MOMS contention, request/response pipeline sharing) which do benefit from the presence of a private cache array.

F. Comparison with the State of the Art

For each of the three graph applications, we consider (1) the two architecture-preprocessing combinations with the highest geometric mean throughput and (2) the architecture-preprocessing combination with the highest performance on a given benchmark. We consider scenario (1) as representative of a general-purpose, possibly hardened graph processor, while scenario (2) shows the highest possible performance that can be achieved by taking advantage of the reprogrammability of FPGAs and using an architecture that is highly optimized for a specific situation. We compare them to (a) FabGraph [44], (b) Gunrock [50], (c) Ligra [46], and (d) GraphMat [3], [47] whenever the respective graph application is available.

TABLE IV
MEMORY BANDWIDTH AND POWER CONSUMPTION OF THE PLATFORMS CONSIDERED IN FIG. 16. *THE GPU POWER IS OVERESTIMATED AS IT INCLUDES THE POWER CONSUMED BY THE ENTIRE BOARD.

	Platform	Ext. mem. bandwidth	Power
This work, FabGraph	FPGA	64 GB/s	23 W
Gunrock	GPU	900 GB/s	300 W*
Ligra, GraphMat	CPU	233 GB/s	224 W

For FabGraph, we use the theoretical model as in Section V-D, which can only be applied to PageRank and ignores RAW stalls. We ran Gunrock on the NVIDIA Tesla V100 with 16 GB of HBM2 memory available on the AWS p3 instances. We evaluated Ligra and GraphMat on a dual socket 2.5 GHz

Intel Xeon E5-2680 v3 with 12 cores and 24 logical threads each, connected to 16 1,833 MT/s DDR4 channels. Only Ligra can benefit from DBG and, thus, we added it to the graph.

Table IV summarizes bandwidth and power consumption. For the FPGA, we report the maximum power reported by the `fpga-describe-local-image` API [21] while for the CPU we read the Intel RAPL registers [13] using CPU Energy Meter [38]. Both measurements exclude external memory and are therefore comparable. Unfortunately, we could not obtain the same kind of data for the GPU and thus we use the TDP, which corresponds to the absolute maximum consumption of the entire board, including HBM2 memory.

On PageRank, our generic architectures outperform the original Ligra, FabGraph, and Gunrock by 2.1×, 1.4×, and 2.1× (geomean) respectively, while the geomean speedup of the specialized architectures increases to 4.5×, 3.0×, and 4.5× respectively and to 1.3× and 1.9× over GraphMat and Ligra with DBG respectively. On SCC and SSSP, our architectures remain competitive with the CPU baselines in absolute terms and are 1.1×–3.5× (generic) and 2.3–5.8× (specialized) more bandwidth-efficient and 3.0–9.4× (generic) and 6.1–15.3× (specialized) more power efficient. On SSSP, Gunrock achieves excellent performance by keeping track of the frontier at the granularity of single nodes as opposed to larger source intervals; however, with only 16 GB of memory, it can only run the five smallest benchmarks, while with the same amount of memory we could run all benchmarks except FR and MP.

G. Resource Utilization

Fig. 17 shows the resource utilization of the highest performing designs of Section V-F. While LUTs and FFs are mostly used in the interconnect, BRAMs and URAMs are used in both PEs and MOMSes. DSPs are in general underutilized, even in the floating-point PageRank. Note that we report the average utilization across the area not occupied by the shell; the utilization per SLR, which is the main factor that affects routability, is higher and peaks at 90% of LUTs in the central SLR for the two-level 16/16 PageRank system without significantly affecting the operating frequency, which remains between 196 MHz and 227 MHz for all of these designs.

VI. Related Work

On CPUs, GraphChi [32] is an out-of-core graph processing system that first introduced the concept of shards to confine random accesses to a smaller range that can be cached in main memory. X-Stream [43] introduced the edge-centric scatter-gather model, where edges are streamed and do not need to be sorted but only partitioned. Frameworks for in-core processing include GraphMat [47], Galois [39], and Totem [22], which also supports hybrid CPU-GPU systems. On a dual-socket Intel Xeon E5-2695 v3 with 28 cores, 240 W TDP, and 136 GB/s of memory bandwidth combined, Aasawat et al. [1] reported 1.3, 1.8, and 9.0 GTEPS for PageRank on RMAT-24 for Galois, GraphMat, and Totem respectively. We achieve 1.8 GTEPS with half the DRAM bandwidth and a 15× lower power. Both Galois and Totem only support graphs in CSR format, where

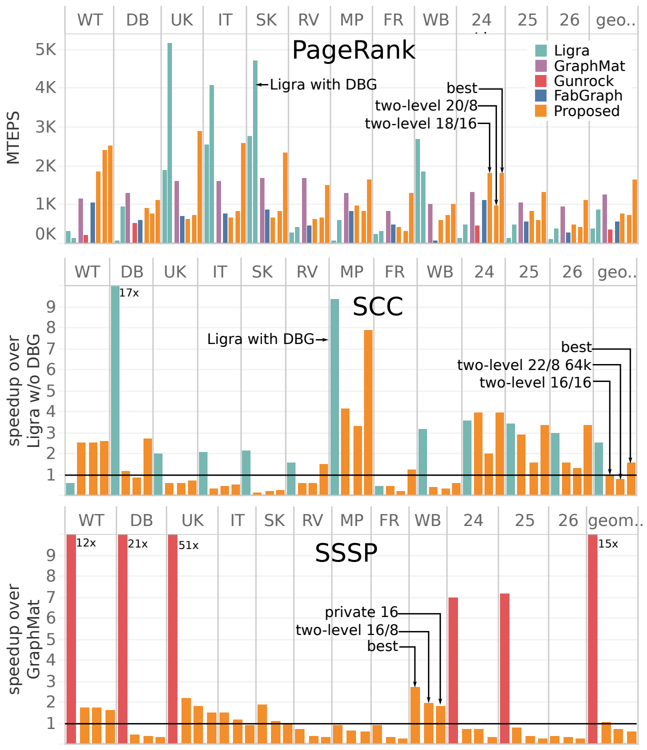

Fig. 16. Comparison with state of the art on CPU, GPU, and FPGA. In many cases, our results are, in absolute terms, competitive to or better than those of the best contender. Considering the bandwidth and power gap among the platforms outlined in Table IV, our solutions are 1.1–5.8× more bandwidth-efficient and 3.0–15.3× more bandwidth-efficient than CPUs and GPUs on PageRank and SCC. Gunrock is more bandwidth- and energy-efficient than our solutions on DB and UK on SSSP but runs only for small benchmarks.

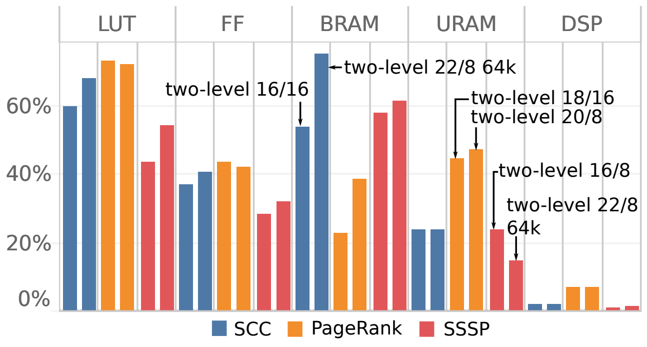

Fig. 17. Relative utilization of resources for the top two architectures of each application. Designs are mostly limited by LUTs, used especially in the interconnect networks, and BRAM.

edges are sorted by source node. During preprocessing, Totem also sorts edges by vertex degree. Our approach does not need any edge sorting but a faster linear-time partitioning. GPOP [33] is a cache-, work-, and memory-efficient framework that also does not require any edge sorting and achieves significant speedup over Ligra and GraphMat. When running PageRank, SSSP, and SCC on the RV and FR benchmarks presented in Table II, our best architecture is 0.22–5.0× faster and 2.2–49×

more energy efficient.

GPUs have a memory bandwidth (and a power budget) that is at least an order of magnitude larger than that of FPGAs; however, it is challenging to fit the irregular workload and memory accesses typical of graphs into the GPU SIMD execution model. One solution, adopted for example by CuSha [28], is to rely on offline preprocessing to balance the workload and group memory accesses, which can constitute a relevant overhead on graphs that are dynamic or used only a few times [37]. While Gunrock [50] shifts this overhead to runtime, Tigr [40] uses a lighter offline preprocessing to convert irregular graphs into equivalent, more regular ones. On a 21M- and a 59M-node graphs, Tigr achieves at most 10% speedup on PageRank compared to Gunrock, which is much lower than the 1.5–12× speedup that our system sports against Gunrock on the same application.

ASICs offer an order of magnitude higher clock rate, density, and energy efficiency than FPGAs [30] at the cost of significantly higher NRE costs and fabrication times. For example, Graphicionado [24] achieves 4.5 GTEPS for PageRank and 0.2 GTEPS for SSSP on the RV graph with a similar memory bandwidth as ours (we achieve 1.5 and 0.7 GTEPS respectively), while GraphDynS [52] achieves more than 85 GTEPS on RMAT-26 on an HBM whose bandwidth is only 8× larger than that of our DDR4. Both solutions have a clock that is 4–5× faster than ours and use significantly more on-chip memory than us: 64 MB and 32 MB compared to, at most, 9 MB.

On FPGA, FabGraph [44] represents the state-of-the-art of large-scale graph processing on a single FPGA that only needs a linear-complexity preprocessing. Its extension, FabGraph+ [45], focuses on optimizing the PCIe transfer when the FPGA DRAM cannot store the entire graph, a problem also tackled by FPGP [14] that is orthogonal to efficiently processing the graph once in its dedicated DRAM. ForeGraph [15] uses a very similar model as FabGraph but also supports multi-FPGA processing. HitGraph [54] outperforms our system on RMAT-24 but not on extremely sparse graphs like WT and, in addition, requires edges to be sorted by destination node. Except for FPGP [14], whose BFS performance on TW is 1.7× less bandwidth-efficient than our more complex SSSP on the same graph, all of these works have only been tested in simulation and do not address the challenges related to multidie partitioning that affect modern large FPGAs.

VII. Conclusion

Graph processing is a key building block of applications in very different domains; yet, achieving good performance is challenging due to the irregular workload distribution, control flow, and memory accesses, especially when graphs are large. We show that FPGAs are an attractive and accessible options to tackle the first two problems and that miss-optimized memory systems are helpful to maximize memory bandwidth utilization on graphs with tens of millions of nodes and billions of edges. We demonstrate our approach on PageRank, SCC, and SSSP, achieving 3× geometric mean speedup compared to state-of-the-art on FPGAs, 1.1–5.8× higher bandwidth efficiency and 3.0–15.3× power efficiency than multicore CPUs, and the ability to scale to very large-scale graphs compared to reference GPU implementations. Our system, to the best of our knowledge, is the first that can run graph processing on multidie FPGAs, pushing the boundary of efficient large-scale graph analytics on a single node in the cloud.

References

[1] Tanuj Kr Aasawat, Tahsin Reza, and Matei Ripeanu. How well do CPU, GPU and hybrid graph processing frameworks perform? In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 458–466. IEEE, 2018.

[2] Amazon.com, Inc. *AWS Announces Seven New Compute Services and Capabilities to Support an Even Wider Range of Workloads.*

[3] Michael J Anderson, Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Theodore L Willke, and Pradeep Dubey. Graphpad: Optimized graph primitives for parallel and distributed platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 313–322. IEEE, 2016.

[4] M. Asiatici and P. Ienne. DynaBurst: Dynamically Assemblying DRAM Bursts over a Multitude of Random Accesses - Public Github repository. https://github.com/m-asiatici/dynaburst/.

[5] Mikhail Asiatici and Paolo Ienne. DynaBurst: Dynamically assemblying DRAM bursts over a multitude of random accesses. In *Proceedings of the 29th International Conference on Field-Programmable Logic and Applications*, pages 254–262, 2019.

[6] Mikhail Asiatici and Paolo Ienne. Stop Crying Over Your Cache Miss Rate: Handling Efficiently Thousands of Outstanding Misses in FPGAs. In *Proceedings of the 27th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 310–319, February 2019.

[7] Tim Bailey, Eduardo Mario Nebot, JK Rosenblatt, and Hugh F Durrant-Whyte. Data association for mobile robot navigation: A graph theoretic approach. In *Proceedings of the 2000 International Conference on Robotics and Automation*, volume 3, pages 2512–2517, San Francisco, California, USA, 2000.

[8] Raji Balasubramanian, Thomas LaFramboise, Denise Scholtens, and Robert Gentleman. A graph-theoretic approach to testing associations between disparate sources of functional genomics data. *Bioinformatics*, 20(18):3353–3362, 2004.

[9] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Li Zhao, Xiaowei Jiang, and Yuan Xie. Analysis and optimization of the memory hierarchy for graph processing workloads. In *Proceedings of the 26th International Symposium on High-Performance Computer Architecture*, pages 373–386, 2019.

[10] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[11] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P. Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. In *In Proceedings of the 4th International AAAI Conference on Weblogs and Social Media (ICWSM)*, Washington DC, USA, May 2010.

[12] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 International Conference on Data Mining*, pages 442–446, Brighton, United Kingdom, 2004.

[13] Intel Corp. Intel 64 and IA-32 Architectures Software Developer Manuals. https://software.intel.com/en-us/articles/intel-sdm.

[14] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong 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*, pages 105–110, 2016.

[15] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. Foregraph: Exploring large-scale graph processing on multi-FPGA architecture. In *Proceedings of the 25th International Symposium on Field Programmable Gate Arrays*, pages 217–226, Monterey, California, USA, 2017.

[16] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1), December 2011.

[17] Assaf Eisenman, Lucy Cherkasova, Guilherme Magalhaes, Qiong Cai, and Sachin Katti. Parallel graph processing on modern multi-core servers: New findings and remaining challenges. In *24th IEEE International*

Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), pages 49–58, 2016.

[18] Assaf Eisenman, Ludmila Cherkasova, Guilherme Magalhaes, Qiong Cai, Paolo Faraboschi, and Sachin Katti. Parallel graph processing: Prejudice and state of the art. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, pages 85–90, 2016.

[19] Priyank Faldu, Jeff Diamond, and Boris Grot. A closer look at lightweight graph reordering. In *Proceedings of the 2019 International Symposium on Workload Characterization (IISWC)*, pages 1–13, Orlando, Florida, USA, 2019.

[20] K. I. Farkas and N. P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 211–222, Chicago, Ill., 1994.

[21] Amazon Web Services Discussion Forum. Does fpga-describe-local-image -S 0 -M include DRAM? https://forums.aws.amazon.com/thread.jspa?messageID=950323&#950323, 2020.

[22] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques (PACT)*, pages 345–354, 2012.

[23] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Bellevue, Washington, USA, 2012.

[24] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proceedings of the 49th Annual International Symposium on Microarchitecture*, pages 1–13, 2016.

[25] Intel Corp. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

[26] 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.

[27] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. Scalable SIMD-efficient graph processing on GPUs. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, pages 39–50, San Francisco, California, USA, 2015.

[28] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, pages 239–252, Vancouver, BC, Canada, 2014.

[29] Jérôme Kunegis. KONECT: The Koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, page 1343–1350, New York, NY, USA, 2013. Association for Computing Machinery.

[30] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 26(2):203–215, 2007.

[31] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, pages 591–600, 2010.

[32] Aapo Kyrola, Guy Blelloch, and Carlos 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.

[33] Kartik Lakhotia, Rajgopal Kannan, Sourav Pati, and Viktor Prasanna. GPOP: A scalable cache-and memory-efficient framework for graph processing over parts. *ACM Transactions on Parallel Computing (TOPC)*, 7(1):1–24, 2020.

[34] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[35] Sheng Li, Ke Chen, Jay B. Brockman, and Norman P. Jouppi. Performance impacts of non-blocking caches in out-of-order processors. HPL Tech Report, HP Labs, 2011.

[36] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz 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, 2010.

[37] Andrew McGregor. Graph stream algorithms: a survey. *ACM SIGMOD Record*, 43(1):9–20, 2014.

[38] SoSy Lab LMU Munich. CPU Energy Meter. https://github.com/sosy-lab/cpu-energy-meter.

[39] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 456–471, 2013.

[40] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming irregular graphs for GPU-friendly graph processing. *ACM SIGPLAN Notices*, 53(2):622–636, 2018.

[41] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the 41st Annual International Symposium on Computer Architecture*, pages 13–24, 2014.

[42] Louise Quick, Paul Wilkinson, and David Hardcastle. Using Pregel-like Large Scale Graph Processing Frameworks for Social Network Analysis. In *Proceedings of the 2012 International Conference on Advances in Social Networks Analysis and Mining*, pages 457–463, Istanbul, Turkey, 2012.

[43] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 472–488, 2013.

[44] Zhiyuan Shao, Ruoshi Li, Diqing Hu, Xiaofei Liao, and Hai Jin. Improving performance of graph processing on FPGA-DRAM platform by two-level vertex caching. In *Proceedings of the 27th International Symposium on Field Programmable Gate Arrays*, pages 320–329, Seaside, California, USA, 2019.

[45] Zhiyuan Shao, Chenhao Liu, Ruoshi Li, Xiaofei Liao, and Hai Jin. Processing grid-format real-world graphs on DRAM-based FPGA accelerators with application-specific caching mechanisms. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 13(3):1–33, 2020.

[46] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 135–146, 2013.

[47] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. GraphMat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 8(11), 2015.

[48] Ichigaku Takigawa and Hiroshi Mamitsuka. Graph mining: procedure, application to drug discovery and recent advances. *Drug discovery today*, 18(1-2):50–57, 2013.

[49] James Tuck, Luis Ceze, and Josep Torrellas. Scalable cache miss handling for high memory-level parallelism. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*, pages 409–422, Orlando, Fla., December 2006.

[50] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 2016.

[51] Xilinx, Inc. *Baidu Deploys Xilinx FPGAs in New Public Cloud Acceleration Services*.

[52] Mingyu Yan, Xing Hu, Shuangchen Li, Abanti Basak, Han Li, Xin Ma, Itir Akgun, Yujing Feng, Peng Gu, Lei Deng, et al. Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 615–628, 2019.

[53] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[54] Shijie Zhou, Rajgopal Kannan, Viktor K Prasanna, Guna Seetharaman, and Qing Wu. HitGraph: High-throughput graph processing framework on FPGA. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2249–2264, 2019.