Heterogeneous Memory Subsystem for Natural Graph Analytics

Abraham Addisie, Hiwot Kassa, Opeoluwa Matthews, and Valeria Bertacco
Computer Science and Engineering, University of Michigan
Email: {abrahad, hiwot, luwa, valeria}@umich.edu

***Abstract*— As graph applications become more popular and diverse, it is important to design efficient hardware architectures that maintain the flexibility of high-level graph programming frameworks. Prior works have identified the memory subsystem of traditional chip multiprocessors (CMPs) as a key source of inefficiency; however, these solutions do not exploit locality that exists in the structure of many real-world graphs. In this work, we target graphs that follow a power-law distribution, for which there is a unique opportunity to significantly boost the overall performance of the memory subsystem. We note that many natural graphs, derived from web, social networks, even biological networks, follow the power law, that is, 20% of the vertices are linked to 80% of the edges. Based on this observation, we propose a novel memory subsystem architecture that leverages this structural graph locality. Our architecture is based on a heterogeneous cache/scratchpad memory subsystem and a lightweight compute engine, to which the cores can offload atomic graph operations. Our architecture provides 2x speedup, on average, over a same-sized baseline CMP running the Ligra framework, while requiring no modification to the Ligra programming interface.**

I. Introduction

In recent years, graph-based algorithms have been widely deployed, being the root of computation for a wide range of applications, from web page ranking [34], to distance computation for online maps [17], to protein-to-protein interaction [7], and human brain functional connectivity analyses [15]. Researchers have proposed several software frameworks to address this growing family of computational needs, including distributed computing systems [18], [21], [26] and single-node chip multi-processor (CMP) solutions [25], [36], [38], [40], [44], [47]. Distributed solutions are well suited for tackling large graphs, while single-node ones are optimized for graphs that fit within a single processing node. The root of the performance advantage of these latter solutions lies in the lower cost of communication, since all data is stored within the same CMP node's storage space. Indeed, the increasing demands for high-performance graph processing have inspired several solutions that specialize both the datapath and the memory subsystem of the processing unit for graph-based algorithms and data structures [5], [19], [33], [42]. However, these proposals suffer from two key limitations: 1) they provide application-specific datapaths, which restrict what graph algorithms can be executed on the architecture; 2) they do not exploit the graph connectivity and, in turn, the locality within a graph's structure, thus missing the opportunity to leverage optimizations from prior research on data locality.

In this work, we strive to overcome precisely these limitations by designing an architecture that is sufficiently general to execute any graph algorithm and, at the same time, takes advantage of the structure of natural graphs, one of the most common types of graphs occurring in large data collections. As shown in Figure 1, our architecture, called **O**ptimized **ME**mory Subsystem Architecture for Natural **G**raph **A**nalytics

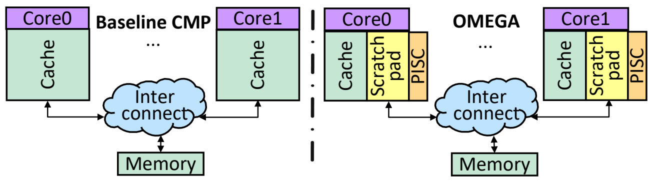


Fig. 1. **OMEGA overview**. OMEGA provides a novel memory architecture to support graph-based computation in CMPs. The architecture is implemented by replacing a portion of each core's cache with a scratchpad memory, organized to store the most highly-connected vertices, and augmented with a simple processing unit (PISC) to carry out simple vertex computations in-situ.

(**OMEGA**), can be deployed seamlessly in one of the CMP-based single-node frameworks currently available, thus providing further performance benefits to the end user, without the need to re-code the application. Our architecture can operate on any type of graph and is designed to best leverage the characteristics of natural graphs.

Our optimizations exploit the power-law properties of many natural graphs. Many real-world data collections are structured as natural graphs, that is, they follow the power-law distribution: 80% of the edges are incident to approximately 20% of the vertices [16], [18]. Examples include web connectivity graphs [34], social networks [13], biological networks such as protein-to-protein interactions [7], and human brain functional networks [15]. One of the key ideas in designing our architecture was to equip it with **on-chip scratchpads** that can accommodate the highly-connected vertices, thus reducing off-chip accesses during an algorithm's execution. Traditional caches cannot effectively harness graph power-law properties; the specially-managed scratchpads overcome these limitations. Note that, generally speaking, accesses to vertex data lack spatial locality, as these frequently accessed vertices are unlikely to be placed adjacent to each other. In contrast, accesses to edge data from caches are more likely to exhibit spatial locality. To support both these needs, we support two granularities for transfers to/from the cores of the CMP: a conventional cache-line-size for accesses to/from caches, and a word-size granularity to/from scratchpads (to avoid inflating transfers with cache-line-size accesses).

In addition, we equipped OMEGA with lightweight compute engines, called the **Processing in Scratchpad (PISC) units, to offload computation and to address key graph-computation bottlenecks**. PISC units are particularly useful as many graph-based applications are bottlenecked by frequent but simple operations (including atomic) on the graph's data structures. Since we have mapped the most frequently accessed data to our on-chip scratchpads, augmenting each scratchpad with a PISC significantly reduces latency by eliminating many scratchpad-core transfers.

Contributions. OMEGA makes the following contributions:

• A distributed scratchpad architecture that can optimize vertex access for natural graphs abiding by a power-law distribution.

```
for src in vertices
    thread X execute PageRankUpdate(src)

PageRankUpdate(src):
    for edge(in,out) in src.outGoingEdge
    out.next_pagerank +=src.curr_pagerank/src.outDegree
```

Fig. 2. **Pseudo-code for a sequential *PageRank*,** computing the *next_pagerank* of a vertex based on the *curr_pagerank* of its neighbors. This access pattern often leads to high cache miss rates and data contention.

• A novel PISC (processing in scratchpad) engine architecture, which provides lower energy and latency for simple computations over the data stored in its associated scratchpad.
• We evaluated OMEGA by modifying a 16-core CMP gem5 [11] setup so that half of the cache space is repurposed to scratchpad storage, and by augmenting it with PISC engines (whose area overhead is estimated $\ll 1\%$). We found that OMEGA provides, on average, a 2x speedup over Ligra on a conventional CMP.

II. Background

Power-law graphs. A power law is a functional relationship such that the output varies with the power of the input. In the context of graphs, power-law graphs follow the power law in their vertex-to-edge connectivity, that is, a few vertices are connected to the majority of the edges, with a long tail of vertices connected to very few edges. In practical approximations, a graph is said to follow the power law if 20% of its vertices are connected to approximately 80% of the edges [30]. Many practical graphs follow the power law, including web-derived graphs, electrical power grids, citation networks, and collaboration networks of movie actors [8], [9]; in addition to graphs arising from social networks [13], biological networks [7], and human brain functional networks [15]. The authors of [8], [9] argued that the reason for such abundant occurrence of power-law distributions in graphs is a mechanism called "preferential attachment": a new vertex joining a graph would most likely connect to an already popular vertex.
Graph-based algorithmic frameworks. In recent years, it has become more common to carry out graph-based computations using high-level frameworks. Several distributed frameworks, running on networks of computers, have been proposed: PowerGraph [18], GraphChi [21], Pregel [26], *etc*. More recently, solutions relying on single-node chip multiprocessor (CMP) architectures have also gained momentum: Ligra [38], GraphMat [40], Polymer [44], X-Stream [36], GridGraph [47], MOSAIC [25], GraphBIG [29], *etc*. In this work, our focus is on CMP solutions, as powerful, high-end servers with large and low-cost memory have sufficient capability to run many real-world graph algorithms in single-node machines.

The single-node frameworks proposed can be broadly classified as either *vertex-centric* (Ligra, GraphMat, Polymer, GraphBIG) or *edge-centric* (X-Stream, GridGraph). The vertex-centric approach has gained popularity because it provides a simple programming paradigm. Particularly, Ligra and GraphMat combine the simplicity of a vertex-centric approach with high performance [27], [46]. In particular, Ligra's ability to operate on graph vertices using a sparsely- or densely-stored active vertex list and by scattering vertex-data by outgoing or incoming edge order makes it suitable for natural graphs as the connectivity of their vertices is highly variable [10].
Graph-based algorithms. Figure 2 provides the pseudo-code of a basic *PageRank* algorithm for a vertex-centric framework.

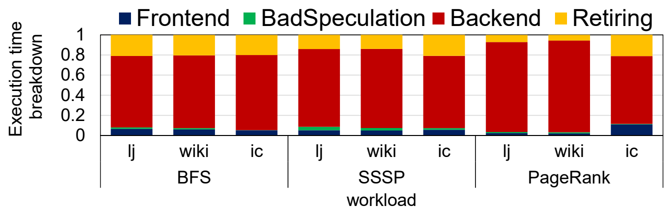


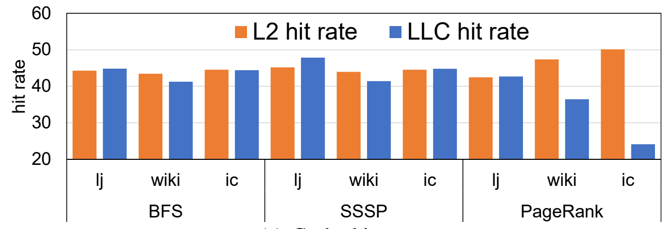
Fig. 3. **Execution breakdown using TMAM [43] metrics**. Profiling several real-world graph workloads using vTune on an Intel Xeon E5-2630 v3 processor with a last-level-cache of 20MB shows that the performance is mostly limited by the execution pipeline's throughput (backend).

Each vertex of the graph represents a webpage, while edges correspond to links between pages. In this PageRank implementation, each thread "X" iterates through all the outgoing edges of its assigned source vertex, to find all the destination vertices and update their page rank values. Note that updating the page rank value may lead to major performance bottlenecks, since i) it leads to several random memory accesses and ii) many distinct threads may attempt to update the same vertex at the same time.
Graph data structures. It is common to provide three data structures to represent a graph: "vertex property" (***vtxProp***), "edge list" (***edgeList***), and non-graph data (***nGraphData***). *vtxProp* stores the values that must be computed for each vertex, *e.g.*, *next_pagerank* from Figure 2; *edgeList* maintains both sets of outgoing and incoming edges for each vertex; and *nGraphData* includes all data structures that are not part of *vtxProp* or *edgeList*. For many graph processing frameworks, most random accesses occur within *vtxProp*, whereas accesses to *edgeList* are frequently sequential, as each vertex has many outgoing and/or incoming edges. Furthermore, *nGraphData* data is generally small in size and entails sequential access-patterns (*e.g.*, loop counters).

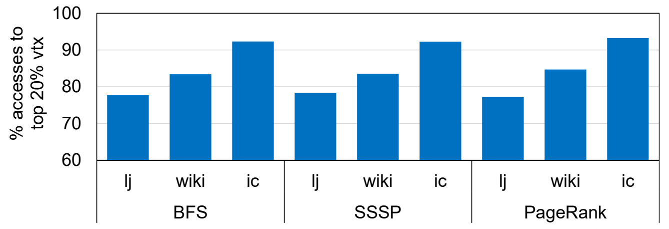
III. Motivation

Bottlenecks in natural graph analytics. To analyze where the bottlenecks lie in graph processing, we profiled a number of graph-based applications over Ligra, a framework optimized for natural graphs [10], on an Intel Xeon E5-2630 processor with a 20MB last-level cache. We used Intel's VTune to gather the "Top-down Microarchitecture Analysis Method" (TMAM) metrics [43]. For this analysis, we used representative graph-based algorithms running on real-world datasets. The results, plotted in Figure 3, show that applications are significantly *backend bounded*, that is, the main cause of performance bottlenecks is the inability of the cores' pipelines to efficiently complete instructions. Furthermore, the execution time break down of the *backend* portion reveals that they are mainly bottlenecked by memory wait time (memory bounded) with an average value of 71%. From this analysis, we can infer that the biggest optimization opportunity lies in improving the memory structure to enable faster access completions. Note that other prior works have also reached similar conclusions [5], [28]. Hence, our primary goal, in this work, is to develop an optimized architecture for the memory system of a CMP to boost the performance of graph processing.

Our proposed approach is unique when compared with prior works because i) we strive to fully exploit the locality that exists in the structure of natural graphs, and ii) we want to maintain transparency to the user as much as possible, thus we strive to minimize – or better, avoid – architectural modifications that require modifications to the graph-processing framework or the graph applications running on it.



(a) Cache hit rate



(b) Percentage of accesses to the top 20% most-connected vertices

Fig. 4. **Cache profiling on traditional CMP architectures**. Left-side: profiling several graph workloads on vTune on Intel Xeon reports hit rates below 50% on L2 and LLC. Right-side: Over 75% of the accesses to *vtxProp* target the 20% most connected vertices, a locality pattern not captured by regular caches.

Inefficiencies in locality exploitation in natural graphs. We then performed a more in-depth analysis of the bottlenecks in the memory subsystem by monitoring last-level cache hit rates for the same graph-based applications. Figure 4(a) shows that all workloads experience a relatively low hit rate. We suspect that the main culprit is the lack of locality in the *vtxProp* accesses, as also suggested by [5], [29]. Moreover, in analyzing the distribution of accesses within *vtxProp*, Figure 4(b) reports which fraction of the accesses targeted the 20% of vertices with highest in-degree: this subset of the vertices is consistently responsible for over 75% of the accesses. The study overall suggests that, if we could accelerate the access of this 20% of vertices in *vtxProp*, then we could significantly reduce the memory access bottleneck, and, in turn, the pipeline backend bottleneck pinpointed by the TMAM analysis.

On-chip communication and atomic instruction overheads. We also observed that, when using conventional caches to access the vertex data structure, each access must transfer data at cache-line granularity, potentially leading to up to 8x overhead in on-chip traffic, since the vertex's *vtxProp* entry most often fits in one word and a cache line is 64 bytes. In addition, when a vertex's data is stored on a remote L2 bank, the transfer may incur significant on-chip communication latency overhead (17 cycles with the baseline setup of our evaluation). Finally, atomic accesses may incur the most significant performance overhead of all those discussed: based on the approach in [28], we estimated the overhead entailed by the use of atomic instructions by comparing overall performance against that of an identical PageRank application where we replaced each atomic instruction with a regular read/write. The result reveals an overhead of up to 50%. Note that these on-chip communication and atomic instructions overheads are particularly high, even for small and medium graph datasets, which could comfortably fit in on-chip storage. We found these kinds of datasets to be abundant in practice, *e.g.*, the *vtxProp* of over 50% of the datasets in [22] can fit on an Intel Xeon E5-2630's 20MB of on-chip storage. Hence, we believe that a viable memory subsystem architecture solution must holistically address atomic instructions, on-chip communication, and off-chip communication overheads.

Limitations of graph pre-processing solutions. Prior works have exploited the locality inherent in a graph by relying on offline vertex-reordering algorithms. We also have deployed some of these solutions on natural graphs, including in-degree, out-degree, and SlashBurn-based reordering [24], but have found limited benefits. More specifically, in-degree- and out-degree-based reorderings provide higher last-level cache hit rates, +12% and +2% respectively, as frequently accessed vertices are stored together in a cache block; however, they also create high load imbalance. For all of the reordering algorithms, we perform load balancing by fine-tuning the scheduling of the OpenMP implementation. We found that the best speedup over the original ordering was 8% for in-degree, 6.3% for out-degree, and no improvement with SlashBurn. Other works [45] have reported similar results, including a slowdown from in-degree-based ordering. Another common graph preprocessing technique is graph slicing/partitioning, which provides good performance at the expense of requiring significant changes to the graph framework [45].

IV. Workload Characterization

To guide the design of our proposed architecture, we gathered a diverse pool of real-world datasets from various popular sources [1], [2], [22] and report their key characteristics in Table I. The datasets include: soc-Slashdot0811 (*sd*), ca-AstroPh (*ap*), rMat (*rMat*), orkut-2007 (*orkut*), ljournal-2008 (*lj*), enwiki-2013 (*wiki*), indochina-2004 (*ic*), uk-2002 (*uk*), twitter-2010 (*twitter*), roadNet-CA (*rCA*), roadNet-PA (*rPA*), Western-USA (*USA*). Datasets collected vary in size, type of graph (directed vs. undirected), and whether they abide by the power law or not in their structures. Indeed, we purposely included a few datasets that do not follow the power law to evaluate the difference in performance impact of our solution. The table also reports in-degree and out-degree connectivity of the 20% most-connected vertices. The in-degree/out-degree connectivity measures the fraction of incoming/outgoing edges connected to the 20% most-connected vertices. Note that those datasets following the power law have very high connectivity values.

Graph-based algorithms. We considered several popular graph algorithms, outlined below and discussed in more detail in [38], [40]. *PageRank (PageRank)* iteratively calculates an influence value for each vertex based on the popularity of its neighbors, until the value converges. *Breadth-First Search (BFS)* traverses the graph breadth-first, starting from an assigned root node, and assigning a parent to each reachable vertex. *Single-Source Shortest-Path (SSSP)* traverses a graph as *BFS*, while computing the shortest distance from the root vertex to each vertex in the graph. *Betweenness Centrality (BC)* computes, for each vertex, the number of shortest paths that go through that vertex. *Radii (Radii)* estimates the maximum radius of the graph, that is, the shortest distance between the furthest pair of vertices in the graph. *Connected Components (CC)*, in an undirected graph, finds all subgraphs such that within each subgraph all vertices can reach one another, and there are no paths between vertices belonging to different subgraphs. *Triangle Counting (TC)*, in an undirected graph, computes the number of triangles *i.e.*, the number of vertices that have two adjacent vertices that are also adjacent to each other. *k-Core (KC)*, in an undirected graph, identifies a maximal-size connected subgraph comprising only vertices of degree $\geq k$.

As shown in Table II, the size of the *vtxProp* entry varies based on the algorithm, from 4 to 12 bytes per vertex. In ad-

TABLE I

GRAPH DATASET CHARACTERIZATION

Characteristic	*sd*	*ap*	*rMat*	*orkut*	*wiki*	*lj*	*ic*	*uk*	*twitter*	*rPA*	*rCA*	*USA*
#vertices (M)	0.07	0.13	2	3	4.2	5.3	7.4	18.5	41.6	1	1.9	6.2
#edges (M)	0.9	0.39	25	234	101	79	194	298	1468	3	5.5	15
type	dir.	undir.	dir.	dir.	dir.	dir.	dir.	dir.	dir.	undir.	undir.	undir.
in-degree con.	62.8	100	93	58.73	84.69	77.35	93.26	84.45	85.9	28.6	28.8	29.35
out-degree con.	78.05	100	93.8	58.73	60.97	75.56	73.37	44.05	74.9	28.6	28.8	29.35
power law	yes	yes	yes	yes	yes	yes	yes	yes	yes	no	no	no
reference	[22]	[22]	[22]	[2]	[2]	[2]	[2]	[2]	[2]	[22]	[22]	[1]

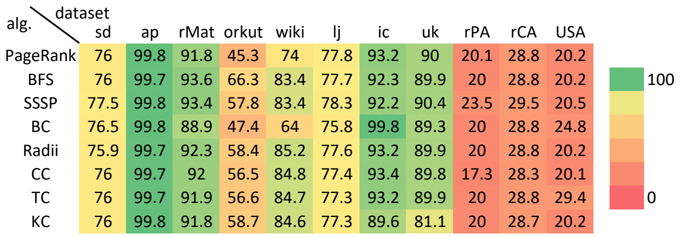
TABLE II

GRAPH-BASED ALGORITHM CHARACTERIZATION

Characteristic	*PageRank*	*BFS*	*SSSP*	*BC*	*Radii*	*CC*	*TC*	*KC*
atomic operation type	fp add	unsigned comp.	signed min & bool comp.	fp add	signed or & signed min	unsigned min	signed add	signed add
%atomic operation	high	low	high	medium	high	high	low	low
%random access	high	high	high	high	high	high	low	low
vtxProp entry size	8	4	8	8	12	8	8	4
#vtxProp	1	1	2	1	3	2	1	1
active-list	no	yes	yes	yes	yes	yes	no	no
read src vtx's *vtxProp*	no	no	yes	yes	yes	yes	no	no

dition, several algorithms, *e.g.*, *SSSP*, require multiple *vtxProp* structures, of variable entry size and stride. Note that the size of *vtxProp* entry determines the amount of on-chip storage required to process a given graph efficiently. Moreover, Table II also highlights that most algorithms perform a high fraction of random accesses and atomic operations, *e.g.*, *PageRank*. However, depending on the algorithm, the framework might employ techniques to reduce the number of atomic operations. For example, in the case of *BFS*, Ligra performs atomic operations only after checking if a parent was not assigned to a vertex. In this case, the algorithm performs many random accesses, often just as expensive. Note that there are graph frameworks that do not rely upon atomic operations, *e.g.*, GraphMat. Such frameworks partition the dataset so that only a single thread modifies *vtxProp* at a time, and thus the optimization targets the specific operations performed on *vtxProp*. Finally, note that many graph algorithms process only a subset of the vertices per iteration, hence they maintain a list of active vertices (*active-list*), which incurs a significant performance penalty. Hence, it is crucial to differentiate algorithms by whether they need to maintain such a list or not.

Many algorithms operating on natural graphs experience <50% hit-rate on the last-level cache (Figure 4(a)), despite a highly-skewed connectivity among the vertices of the graph. Our graph workload characterization, reported in Figure 5, reveals that a hit rate of 50% or greater can be achieved on the randomly-accessed portion of the vertex data (*vtxProp*). As shown in the Figure, for graphs that follow the power law, up to 99% of the *vtxProp* requests can be served by accommodating just the top 20% most-connected vertices in on-chip storage, which is practical for many graphs encountered in real-world applications. The per-vertex storage requirement depends on the algorithm: for example, Ligra uses 4 bytes for *BFS*, 8 bytes for *PageRank*, and 12 bytes for *Radii* (Table II). Our analysis on storage requirements reveals that 20% of the vertices for the graphs in Table I can be mapped to a fraction of today's on-chip storage sizes. Among the graphs considered, only *uk* and *twitter* would require more than 16MB of on-chip storage to attain that goal. *uk* requires up to 42MB, whereas *twitter* requires 64MB. In fact, by re-purposing IBM's total L2 + L3 caches (132MB) to store 20% vertices, up to 164 million vertices (1.64 billion edges, assuming R-MAT's [12] default edge-vertex ratio) could be allocated to on-chip storage.



alg. \ dataset	sd	ap	rMat	orkut	wiki	lj	ic	uk	rPA	rCA	USA
PageRank	76	99.8	91.8	45.3	74	77.8	93.2	90	20.1	28.8	20.2
BFS	76	99.7	93.6	66.3	83.4	77.7	92.3	89.9	20	28.8	20.2
SSSP	77.5	99.8	93.4	57.8	83.4	78.3	92.2	90.4	23.5	29.5	20.5
BC	76.5	99.8	88.9	47.4	64	75.8	99.8	89.3	20	28.8	24.8
Radii	75.9	99.7	92.3	58.4	85.2	77.6	93.2	89.9	20	28.8	20.2
CC	76	99.7	92	56.5	84.8	77.4	93.4	89.8	17.3	28.3	20.1
TC	76	99.7	91.9	56.6	84.7	77.3	93.2	89.9	20	28.8	29.4
KC	76	99.8	91.8	58.7	84.6	77.3	89.6	81.1	20	28.7	20.2

Fig. 5. **Accesses to the 20% most connected vertices.** The heat map shows the fraction of accesses to *vtxProp* that refer to the 20% most-connected vertices. 100 indicates that all accesses are to those vertices. Data for *twitter* is omitted because of its extreme profiling runtime.

V. OMEGA ARCHITECTURE

The OMEGA architecture, shown on the right side of Figure 6, employs a heterogeneous storage architecture that comprises both conventional caches and distributed scratchpads. Each scratchpad is augmented with a lightweight compute-engine (PISC), which executes atomic operations offloaded from the core. The scratchpads store the most-accessed portion of the *vtxProp* data structure, which is where most of the random accesses occur. By doing so, most accesses to *vtxProp* are served from on-chip storage, and off-chip memory accesses are minimized. In contrast, the *edgeArray* and *nGraphData* data structures are stored in conventional caches, since they are mostly accessed sequentially, thus benefiting from cache optimization techniques. While the scratchpads help minimize off-chip memory accesses, the PISC reduces overheads of executing atomic operations on general-purpose cores. These overheads are due to on-chip communication latency related to frequent *vtxProp* access from remote scratchpads and atomic operations causing the core's pipeline to be on-hold until their completion [28]. The PISCs, each colocated with a scratchpad, minimize these overheads by providing computational capability near the relevant data location, performing the atomic update by leveraging the two co-located units. In contrast, a number of inefficiencies arise if a core is used, even a local one: first, the significant performance penalty for entering and exiting an interrupt service routine, up to 25ns [5]. Second, since atomic operations are mostly simple ones, a simple PISC can complete them using less energy than a full-blown CPU.

To illustrate the operation of the OMEGA architecture, we use the sample graph on the left of Figure 6. The vertices

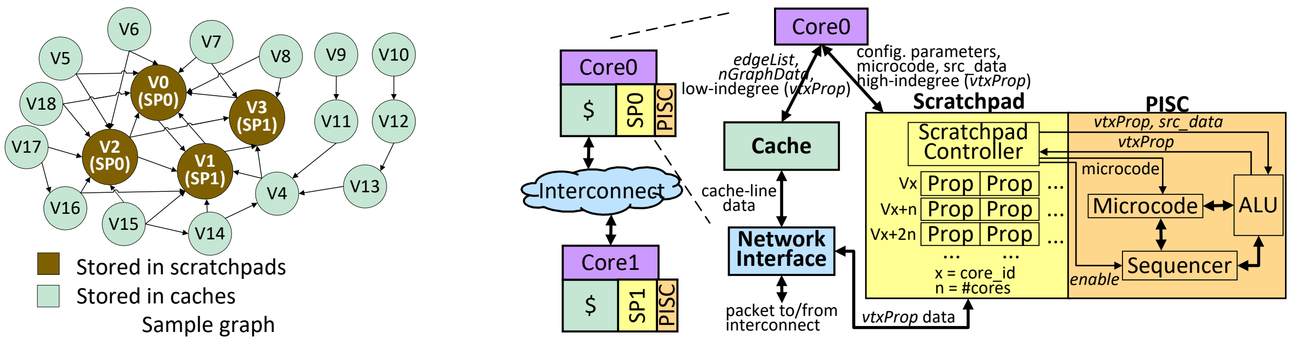


Fig. 6. **OMEGA architecture**. Left-side - Sample graph that follows power law: The vertices are reordered based on their in-degree, with lower ID indicating a higher connectivity. Right-side - OMEGA's heterogeneous cache/SP architecture: for each core, OMEGA adds a scratchpad and a PISC. For the sample graph of the left-side, the *vtxProp* for the most-connected vertices (V0 to V3) is partitioned across all on-chip scratchpads. The rest of the vertices are stored in regular caches. The PISCs execute atomic instructions that are offloaded from the cores.

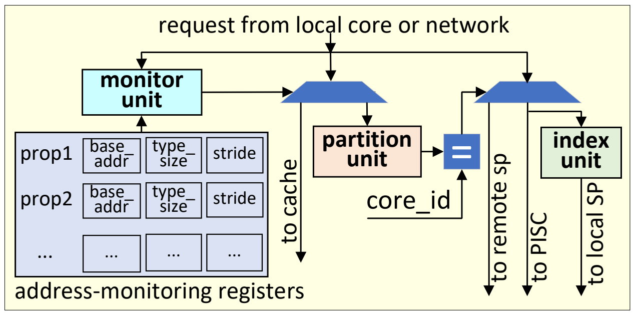


Fig. 7. **Scratchpad controller**. The scratchpad controller orchestrates access to the scratchpad. It uses a pre-configured set of address monitoring registers to filter requests destined to the scratchpads. The partition unit determines if a request is to a local scratchpad or a remote one. To identify the scratchpad line of a request, the *index unit* is employed.

are reordered by decreasing in-degree, with V0 being first. OMEGA partitions the *vtxProp* of the most-connected vertices, in our case V0 to V3, among all on-chip scratchpads. All other data, that is, the *vtxProp* for the remaining vertices (V4 to V18), *edgeArray*, and *nGraphData*, are stored in the regular cache hierarchy. This partitioning scheme enables OMEGA to serve most *vtxProp* accesses from on-chip scratchpads, by providing storage for just a small portion of the structure. For graphs that follow the power law, it should be sufficient to store just 20% of *vtxProp* to scratchpads, for OMEGA to serve 80% of the accesses from there.

A. *Scratchpad architecture*

A high-level scratchpad architecture is shown in Figure 6. The scratchpad is organized as a directly-mapped storage. For each line, we store all *vtxProp* entries (*Props*) of a vertex, thus all *Props* of a vertex can be retrieved with a single access, which is beneficial for the efficient execution of atomic operations. In addition, an extra bit is added for each *vtxProp* entry to track the *active-list* using a dense representation [38]. The access to the scratchpad is managed by the scratchpad controller, as discussed below.

Scratchpad controller. Figure 7 shows the scratchpad controller. It includes a set of *address-monitoring registers*, the *monitor unit*, the *partition unit*, and the *index unit*. The scratchpad controller receives normal read and write requests, as well as atomic-operation requests from both the local core and other scratchpads via the interconnect. Upon receiving them, its *monitor unit* determines if the request should be routed to the regular caches or to the scratchpads. For this purpose, the *monitor unit* relies on a set of *address monitoring registers*. The *address monitoring registers* are shown on the left-side of Figure 7. For each *vtxProp*, we maintain its *start_addr*, *type_size*, and *stride*. The *start_addr* is the same as the base address of the *vtxProp*. The *type_size* is the size of the primitive data type stored by the *vtxProp*. For instance, for *PageRank*, the "next_pagerank" *vtxProp* maintains a primitive data type of "double", hence its *type_size* would be 8 bytes. The *stride* is usually the same as the *type_size* except when the *vtxProp* is part of a "struct" data structure. In that case, it is determined by subtracting the first two consecutive *vtxProp* addresses. All of these registers are configured by the graph framework at the beginning of an application's execution. If the scratchpad controller determines that a request is for the regular cache, it ignores it, as it will be handled by the regular cache controller. If it is for the scratchpads, the *partition unit* is used to determine whether the request is for a local scratchpad or for a remote one. Either way, if the request is not an atomic operation, the scratchpad controller simply forwards the request to the scratchpad. If the request is for a remote scratchpad, the scratchpad controller forwards the request to it via the interconnect using a special packet. The *index unit* is used to identify the line number of the request to perform the actual scratchpad access. For an atomic operation request, the scratchpad controller reads the required *vtxProp* of the request from the scratchpad, initiates its atomic execution on the PISC engine, and then writes the result back to the scratchpad. Note that, while the execution of the atomic operation is in progress, the scratchpad controller blocks all requests that are issued to the same vertex.

B. *PISC (Processing-in-Scratchpad) unit*

When we deployed the on-chip scratchpads to store the most-accessed portion of the *vtxProp* and, thus, bound most of the random-access requests to on-chip storage, we found that these requests were most often to remote scratchpads. That is, the general-purpose cores would issue requests that, for the most part, were served by remote scratchpads. This effect creates bottlenecks due to the interconnect transfer latency. Thus, to address this issue, we augmented our solution with simple PISC engines, to offload such requests from the processor cores. Due to their simplicity, PISCs enable lower energy and latency for data accesses, similarly to the benefits of Processing-in-Memory (PIM) units. Each scratchpad is augmented with a PISC engine that executes the atomic operations of the algorithm in execution, *e.g.*, floating-point addition for *PageRank* algorithm. Figure 8 illustrates how PISCs work. The sample graph on the left is being processed by Core0 running the *PageRank* algorithm: starting with the source vertex V4, Core0 updates the next pagerank for all its outgoing edges (in

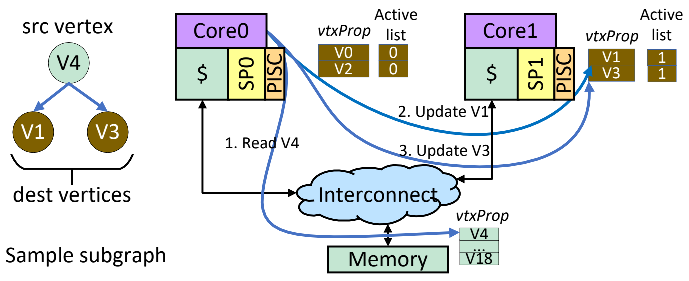


Fig. 8. **Atomic operation offloading**. The PISC executes atomic operations offloaded by the cores. When Core0, executing *PageRank* on the sample graph, process the V4-V1 edge, it transfers the *vtxProp* of V4 to Core1's PISC, which takes care of updating the value of V1's *next_pagerank* atomically.

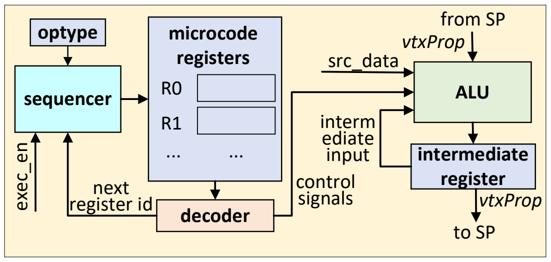


Fig. 9. **PISC architecture**. A PISC includes a simple ALU engine, implementing several atomic operations to support a wide range of graph workloads. The sequencing logic controls the execution of the corresponding microcode.

this case, V1 and V3). Without a PISC, this function entails two remote *vtxProp* read operations from Core1's scratchpad, along with a significant read latency cost. However, with a PISC, Core0 can simply send a message to Core1's PISC requesting the computation of the *next_pagerank* for V4, and then advance to processing other vertices.

PISC architecture. A high-level architecture of a PISC unit is shown in Figure 9. One of the main components of a PISC is an ALU engine, which supports several operations corresponding to the atomic operations of the algorithms discussed in Section X. For instance, *PageRank* requires "floating point addition", *BFS* requires "unsigned integer comparison", and *SSSP* requires "signed integer min" and "Bool comparison". In addition, the *microcode registers* store the sequence of micro-operations implementing each atomic operation. The *sequencer* is responsible for interpreting the incoming atomic operation command, reading/writing from/to the scratchpad, and controlling the execution of the microcode.

Maintaining the *active-list*. As discussed earlier, several graph-based algorithms leverage an *active-list* to keep track of the set of vertices that must be processed in the next iteration of the algorithm. As the algorithm proceeds, the *active-list* is frequently updated: we offloaded this activity from processor cores, too, to avoid a penalty in on-chip latency due to cores waiting on PISC engines' completion. Note that there are two kinds of *active-list* data structures: *dense-active-lists* and *sparse-active-lists* [38]. To update a *dense-active-list* in our OMEGA solution, the local PISC sets a bit in the scratchpad corresponding to the vertex entry on which the atomic operation is operating. To update the *sparse-active-list*, the PISC writes the ID of the active vertex to a list structure, stored in memory via the L1 data cache.

C. Source vertex buffer

Many graph algorithms first read a source vertex's *vtxProp* (see Table II), apply one or more operations to it, and then use the result (*src_data*) to update the *vtxProp* of the source's adjacent vertices. For instance, the *SSSP* algorithm (pseudo-code shown in Figure 10) reads a source vertex's *vtxProp*, in this case *ShortestLen*, adds the *edgeLen* to it, then uses the result to update the *ShortestLen* of all its adjacent vertices. When executing this sequence of operations using the regular cache hierarchy, a single read to the source vertex would bring the data to L1 cache, and all subsequent read requests (up to the number of outgoing edges of the source vertex) will be served from there. However, when using the scratchpads, distributed across all the cores, a read access to a vertex's information by a remote core could incur a significant interconnect-transfer latency (an average of 17 processor cycles in our implementation). To minimize this latency, we introduced a new read-only small storage-structure called the *source vertex buffer*. Every read request to a source vertex's *vtxProp* is first checked against the contents of this buffer. If the request cannot be served from it, the request is forwarded to the remote scratchpad. Upon a successful read from a remote scratchpad, a copy of the vertex data retrieved will be placed in the buffer to serve future requests to the same vertex. Note that, since all buffer's entries are invalidated at the end of each algorithm's iteration, and the source vertex *vtxProp* is not updated until that point, there is no need to maintain coherence between this buffer and the distributed scratchpads. Figure 11 illustrates how the buffer works. The sample graph on the left is being processed by Core0 running the *SSSP* algorithm: starting with the source vertex V3, Core0 updates the next *ShortestLen* for all its outgoing edges (in this case, V0 followed by V1). To perform the update for V0, Core0 reads the *ShortestLen* of V3 from SP1, entailing a remote scratchpad access. Upon a successful read, a copy of V3's *ShortestLen* will be stored in the buffer. Then, when Core0 attempts to read the *ShortestLen* of V3 again, in this case to update V1, the read will be satisfied from the buffer.

D. Reconfigurable scratchpad mapping

As discussed earlier, the performance of most graph algorithms is limited by their random access patterns to the *vtxProp* data structure. However, some algorithms also perform a significant number of sequential accesses to the same data structure. For the portion of the *vtxProp* that is mapped to the scratchpads, OMEGA's mapping scheme affects the efficiency of such sequential accesses. To understand this scenario in more detail, we include a code snippet of *PageRank* at the top of Figure 12, showing a sequential access to *vtxPorp*. The code copies *vtxProp* (*next_pagerank*) to another temporary data structure (*curr_pagerank*). The *vtxProp* is stored on the scratchpad while the temporary structure is stored in cache. As shown in the bottom-side of Figure 12, given an interleaving-based mapping with a chunk size of 1, and an OpenMP scheduling based on assigning an equally-sized chunk for each thread (2 for the code snippet shown in the Figure), the activity of copying *vtxProp* of V1 by Core0 and *vtxProp* of V2 by Core1 would involve remote scratchpad accesses. To avoid them, the chunk size that OMEGA uses to map the *vtxProp* to the scratchpads is pre-configured to match that of the chunk size configured in the framework's OpenMP scheduling scheme. This setting make it so potentially remote scratchpad accesses become local ones when performing sequential accesses.

E. On-chip communication

Communication granularity of scratchpad. Although a large portion of the accesses to *vtxProp* are served from the scratchpads, these accesses still lack spatial locality. Such a

```
for s in Vertices
    for d in s.outGoingEdge
        update(s,d,edgeLen[s][d]);
update (s, d, edgeLen) {
        newDist = ShortestLen[s] + edgeLen;
        [atomic_]ShortestLen[d] = min(ShortestLen[d], newDist);
        [atomic_]Visited[d] = 1;
}
```

Fig. 10. **Pseudo-code for *SSSP***. The algorithm considers each pair of vertices, "s" and "d", adds "s" current shortest length ("ShortestLen") to the distance to "d" and then updates the "ShortestLen" of "d". To keep track of visited vertices, their corresponding *Visited* tag is set to 1.

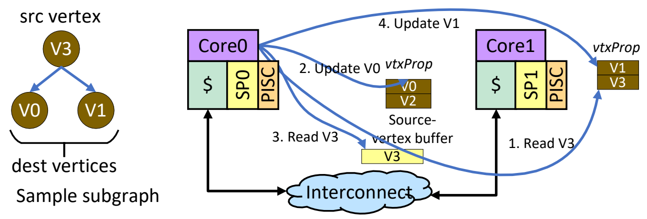


Fig. 11. **Source vertex buffer**. In processing the graph, first V3's *vtxProp* is read from a remote scratchpad, and a copy is created locally in the "source vertex buffer". Subsequent reads to the same entry are then served from it.

deficiency causes most accesses from the cores to be served from remote scratchpads. Consequently, frequent accesses to the remote scratchpads at a cache-line granularity wastes on-chip communication bandwidth. To address this aspect, OMEGA accesses the scratchpads at a word-level granularity. The actual size of the scratchpad access depends on the *vtxProp* entry type, and it ranges from 1 byte (corresponding to a "Bool" *vtxProp* entry) to 8 bytes (corresponding to a "double" *vtxProp* entry) in the workloads that we considered (see Table II). For communication with the remote scratchpads, OMEGA uses custom packets with a size of up to 64-bits, as the maximum *type_size* of a *vtxProp* entry is 8 bytes. Note that this size is smaller than the bus-width of a typical interconnect architecture (128 bits in our evaluation), and its size closely resembles the control messages of conventional coherence protocols (*e.g.*, "ack" messages). Using a word-level granularity instead of a conventional cache-block size enables OMEGA to reduce the on-chip traffic by a factor of up to 2x.

F. Adopting software frameworks

Lightweight source-to-source translation. High-level frameworks such as Ligra must be slightly adapted to benefit from OMEGA. To this end, we developed a lightweight source-to-source translation tool. Note that source-to-source translation implies that the resulting framework will be in the same programming language as the original one, hence no special compilation techniques is required. The tool performs two main kinds of translation. First, it generates code to configure OMEGA's microcode and other registers. To configure the microcode registers, it parses a pre-annotated "update" function by the framework developers (an example for *SSSP* is shown in Figure 10) and generates code comprising a series of store instructions to a set of memory-mapped registers. This code is the microcode to be written to each PISC. Note that the microcode contains a relatively simple set of operations, mostly implementing the algorithm's atomic operations. For example, the microcode for *PageRank* would involve reading the stored page-rank value from the scratchpad, followed by performing floating-point addition and writing the result back to the scratchpad. In addition to the microcode, the tool generates code for configuring OMEGA, including the *optype*

```
#pragma omp parallel for
for(int i = 0; i < 4; i++) {curr_pagerank[i] = next_pagerank[i];}
```

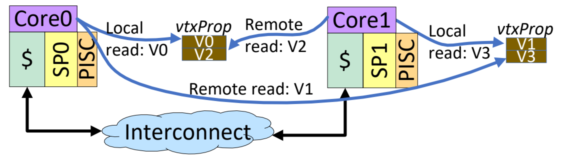


Fig. 12. **Cost of mismatched chunk sizes for scratchpad mapping and OpenMP scheduling.** Given a scratchpad mapping based on interleaving with a chunk size of 1, and different from the OpenMP scheduling (shown with a chunk size of 2) the mismatch in chunk size causes half of the accesses to *vtxProp* to be from remote scratchpads.

```
update (s, d, edgeLen) {
    *mem_mapped_reg1 = ShortestLen[s] + edgeLen;
    *mem_mapped_reg2 = d;
}
```

Fig. 13. **Code generated for *SSSP* with the source-to-source translation tool**. Microcode for the update function of Figure 10. The new computed *ShortestLen* is written in memory-mapped register 1, and the ID of the destination vertex is written to memory-mapped register 2.

(the atomic operation type), the start address of *vtxProp*, the number of vertices, the per-vertex entry size, and its stride. The code for all these configurations is executed at the beginning of the application execution. In addition to configuration code, the tool translates the annotated "update" function into equivalent code that contains a series of store instructions to a set of memory mapped registers. This code is executed whenever the framework executes the "update" function offloaded to the PISC engines. To highlight this transformation, we show the "update" function of *SSSP* algorithm in Figure 10 and its translated code in Figure 13. To verify the functionality of the tool across multiple frameworks, we applied the tool to GraphMat [40] in addition to Ligra [38].

VI. Graph preprocessing

OMEGA's benefits rely on identifying highly-connected vertices of a graph dataset, and mapping their *vtxProp* to the on-chip scratchpads. Broadly speaking, there are two approaches to achieve this purpose: dynamic and static approaches. With a dynamic approach, the highly-connected vertices can be identified by using a hardware cache with a replacement policy based on vertex connectivity and a word granularity cache-block size, as suggested in other works [20], [31], [41]. However, this solution incurs significant area and power overheads as each vertex must store tag information, *e.g.*, 2x overhead for *BFS* assuming 32 bits per tag entry and 32 bits per *vtxProp* entry. To remedy the high overhead of the dynamic approach, a static approach based on reordering the graph using offline algorithms can be utilized. Any kind of reordering algorithm can be used with OMEGA, as long as it produces a monotonically decreasing/increasing ordering of popularity of vertices. Both in-degree- and out-degree-based reordering algorithms provide such ordering. We found in practice that in-degree-based reordering captures a larger portion of the natural graphs' connectivity as shown in Table I. Slashburn-based reordering, however, produces suboptimal results for our solution as it strives to create community structures instead of a monotonically reordered dataset based on vertex connectivity. We considered three variants of in-degree-based reordering algorithms: 1) sorting the complete set of vertices, which has an average-case complexity of $v \log v$, where v is the number of vertices; 2) sorting only the top 20% of the vertices, which has the same average-case time complexity; and 3) using an "n-th element" algorithm that reorders a list of vertices so that all vertices stored before the n-th index in the list have connectivity higher than those after (in our case, *n* would be the 20% index mark). This algorithm has a linear average-case time complexity. We chose the third option in our evaluation since it provides slightly better performance and has a very-low reordering overhead. However, a user might find the first option more beneficial if the storage requirement for 20% of the vertices is significantly larger than the available storage.

VII. Scaling scratchpad usage to large graphs

Our prior discussion about the scratchpad architecture is based on the assumption that OMEGA's storage can accommodate a significant portion (≈20%) of the *vtxProp* of a graph. In this section, we discuss several approaches to scale the scratchpad architecture for larger graphs, where the *vtxProp* of their most-connected vertices does not fit into the scratchpads.

1) Storing *vtxProp* for the most-connected vertices. First of all, OMEGA's scratchpad architecture continues to provide significant benefits even when it provides storage for the *vtxProp* of <20% of the vertices. The key reason is that storing the most-connected vertices is the best investment of resources compared to re-purposing the same storage for conventional caches. However, as the size of the graph keeps increasing, there is a point where OMEGA's scratchpads would become too small to store a meaningful portion of *vtxProp*, and, consequently, the benefit would be negligible. Below, we discuss two approaches that enable OMEGA to continue providing benefits even in this scenario.

2) Graph slicing. [19], [45] proposed a technique called graph slicing/segmentation to scale the scratchpad/cache usage of their architecture. In this technique, a large graph is partitioned into multiple slices, so that each single slice fits in on-chip storage. Then, one slice is processed at a time and the result is merged at the end. While the same technique can be employed to scale the scratchpad usage for OMEGA, there are several associated performance overheads: 1) processing time required for partitioning a large graph to smaller slices; 2) combining the results of several slices. These overheads increase with the numbers of slices: the next approach address this challenge.

3) Graph slicing and exploiting power law. Instead of slicing graphs so that each slice fits in the scratchpads, slicing can be performed to fit just the *vtxProp* of the 20% most-connected vertices, which is sufficient to serve most *vtxProp* accesses. The approach significantly reduces the total number of graph slices by up to 5x, along with the associated overheads.

We evaluated of OMEGA based on the first approach. Evaluation of the other two options is left to future work.

VIII. Memory semantics

Cache coherence. Each *vtxProp* entry is mapped to and handled by only one scratchpad, without requiring access to the conventional caches, thus avoiding any sharing. All the remaining vertices and other data structures are managed by the conventional cache coherence protocol. Hence, there is no need to manage coherence among scratchpads or between scratchpads and conventional caches.

Virtual address translation. The local scratchpad controller maps the virtual address of an incoming request into a vertex ID, and it uses the ID to orchestrate scratchpad access, including identifying which scratchpad to access, whether local or remote. Hence, there is no need to incorporate virtual to physical address translation into the scratchpads.

TABLE III
Experimental testbed setup

Common configuration
Core: 16 OoO cores, 2GHZ, 8-wide, 192-entry ROB L1 I/D cache per core: 16KB, 4/8-way, private cache block size: 64 bytes Coherence protocol: MESI_Two_Level memory: 4xDDR3-1600, 12GB/s per channel interconnect topology: crossbar, 128-bits bus-width
Baseline-specific configuration
L2 cache per core: 2MB, 8-way, shared
OMEGA-specific configuration
L2 cache per core: 1MB, 8-way, shared SP per core: 1MB, direct, lat. 3-cycles SP access granularity: 1-8 bytes

Atomic operation. If OMEGA only employed distributed scratchpads as on-chip storage, execution of atomic operations would be handled by the cores. Since on-chip scratchpad accesses happen at word-level granularity instead of cache-line granularity, the cores lock only the required word address. Aspects of the implementation other than access granularity remain the same as in a conventional cache coherence protocol.

IX. Further Considerations with OMEGA

Dynamic graphs. OMEGA relies on an offline reordering algorithm to identify the 20% most popular vertices. With dynamic graphs, these set of vertices might vary, as new vertices/edges are added to or removed from the graph. By using a reordering algorithm to re-identify the popular vertices, as long as the high-level framework supports it, OMEGA can be adapted to continue to provide the same benefits as it does for static graphs. However, we defer a detailed discussion of this application of OMEGA to future work.

Locked cache vs. scratchpad. Locking cache lines allows programmers to load a cache line and disable its replacement policy [35]. This technique could be extended to capture the top popular vertices by locking their corresponding cache lines. While this technique might lower the amount of modifications to the architecture compared to OMEGA, it would still suffer from high on-chip communication overhead because data is inefficiently accessed on a cache-line granularity instead of word granularity.

Optimizing access to the least-connected vertices. OMEGA achieves significant performance benefits even without accommodating the *vtxProp* of the least-connected vertices within scratchpads. However, the access latency to this portion of the *vtxProp*, mostly from off-chip memory, could potentially be the bottleneck to further performance improvements. In light of this observation, we note three potential directions to extend OMEGA's techniques to off-chip memory: 1) access to the *vtxProp* of the least connected vertices at word-level granularity; 2) offloading operations on the least-connected vertices to off-chip memory, leading to a hybrid PISC and PIM architecture; and 3) employing a hybrid close- and open-page policy: close-page for the least connected vertices as they lack spatial locality and open-page for the rest of the data structures including the *edgeList*. The focus of this work is on evaluating the benefits of on-chip storage and offloaded atomic operations, thus we defer the exploration of these directions to future work.

X. Experimental evaluation

Experimental setup. We modeled OMEGA on gem5 [11], a cycle-accurate simulation infrastructure. We ported the

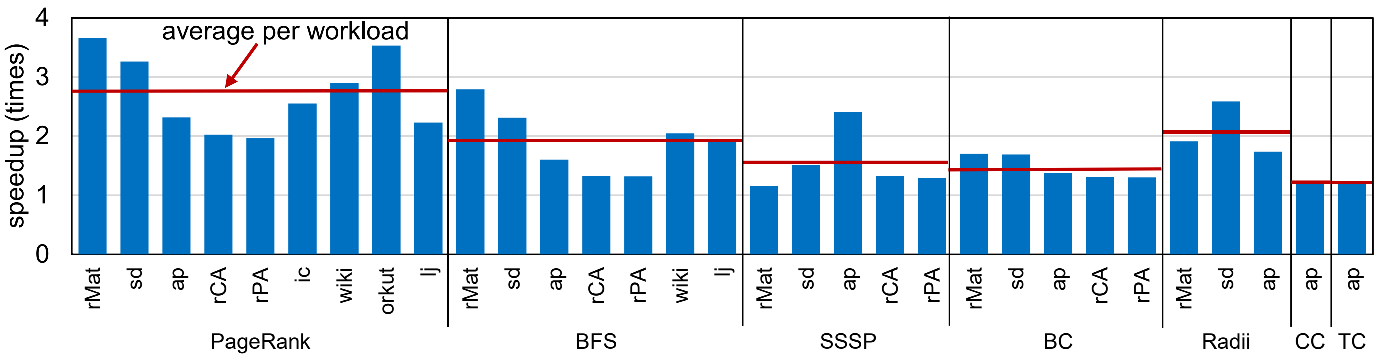


Fig. 14. **OMEGA performance speedup**. OMEGA provides 2x speedup, on average, over a baseline CMP running Ligra.

OpenMP implementation of the Ligra framework [38] to gem5 using "m5threads" and we carried out the simulation in "syscall" emulation mode. We compiled Ligra with gcc/g++ using the O3 optimization flag. Our simulation setup is summarized in Table III. Our baseline design is a CMP with 16, 8-wide, out-of-order cores with private 32KB of L1 instruction and data caches and shared L2 cache, with a total storage size matching OMEGA's hybrid scratchpad+cache architecture. For OMEGA, we keep the same parameters for the cores and L1 caches as the baseline CMP. Each scratchpad is augmented with a PISC engine. Communication with the caches takes place at cache-line granularity (64 bytes), and communication with the scratchpads occurs at word-size granularity, with entry sizes from 1 to 8 bytes, depending on the size of the *vtxProp* entry being accessed.

Workloads. We considered the workloads discussed in Section IV. Note that *TC* and *KC* present similar results on all our experiments, thus we report only the results for *TC*. *CC* and *TC* require symmetric graphs, hence, we run them on one of the undirected-graph datasets (*ap*). Because of the long simulation times of gem5, we simulate only a single iteration of *PageRank*. In addition, we simulate only the "first pass" of *BC* and, for *Radii*, we use a "sample size" of 16. Other algorithms are run with their default settings to their completion. Our selection of graph algorithms applied to the large graphs, such as *lj*, is motivated by the limited performance of gem5.

A. Performance evaluation

Figure 14 shows the performance benefit provided by OMEGA compared to Ligra running on a baseline CMP. OMEGA achieved a significant speedup, over 2x on average, across a wide range of graph algorithms (see Table II) and datasets (see Table I). The speedup highly depends on the graph algorithm. OMEGA achieved significantly higher speedups for *PageRank*, 2.8x on average, compared to others. The key reason behind this is that Ligra maintains various data structures to manage the iterative steps of *PageRank*. In the case of *PageRank* and *TC*, all vertices are active during each iteration, hence, the per-iteration overhead of maintaining these data structures is minimal. Unfortunately, *TC*'s speedup remains limited because the algorithm is compute-intensive, thus random accesses contribute only a small fraction to execution time. In contrast, for the other algorithms, Ligra processes only a fraction of the vertices in each iteration, hence, maintaining the data structures discussed above negatively affects the overall performance benefit. However, even with these overheads, OMEGA managed to achieve significant speedups: for *BFS* and *Radii*, an average of 2x, and for *SSSP*, an average of 1.6x.

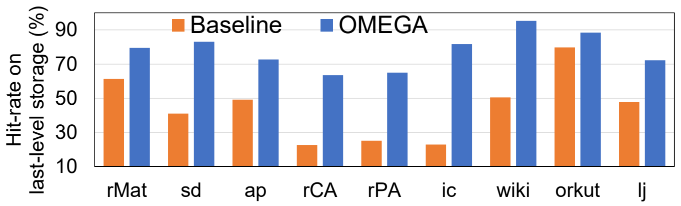


Fig. 15. **Last-level storage hit-rate in *PageRank***. OMEGA's partitioned L2 caches and scratchpads lead to a significantly larger hit-rate, compared to the baseline's L2 cache of the same size.

In addition to the impact of the algorithm, the speedup highly varies across datasets. OMEGA manages to provide significant speedup for datasets for which at least 20% of the *vtxProp* fits in the scratchpads, *e.g.*, *lj* and *rMat* (whose vtxProp fits completely). The key observation here is that, for a natural graph, OMEGA's scratchpads shall provide storage only for 20% of the *vtxProp* to harness most of the benefits provided by storing it completely in scratchpads. Despite not following a power-law distribution, *rCA* and *rPA* achieve significant performance gains since their *vtxProp* is small enough to fit in the scratchpads. However, compared to other datasets, they achieve a smaller speedup because their low out-degree connectivity (see Table I) makes the access to the *edgeList* more random compared to other graphs.

Cache/Scratchpad access hit-rate. In Figure 15, we compare the hit rate achieved by the last-level cache (L2) of the baseline design against that of OMEGA's partitioned storage (half scratchpad and half the L2 storage of the baseline). The plot reveals that OMEGA provides over 75% last-level "storage" hit-rate on average compared to a 44% hit-rate for the baseline. The key reason for OMEGA's higher hit rate is because most of the *vtxProp* requests for the graphs that follow the power law are served by scratchpads.

Using scratchpads as storage. To isolate the benefits of OMEGA's scratchpads, without the contributions of the PISC engines, we performed an experiment running *PageRank* on the *lj* dataset with this scratchpads-only setup. OMEGA achieves only a 1.3x speedup, compared to the >3x speedup when complementing the scratchpads with PISCs. The lower boost of the scratchpads-only solution is due to the foregoing of improvements in on-chip communication and atomic operation overheads. Computing near the scratchpads using the PISCs alleviates these overheads.

Off- and on-chip communication analysis. As noted by prior work [10], graph workloads do not efficiently utilize the available off-chip bandwidth. We measured our DRAM bandwitdh utilization on a range of datasets while running *PageRank* and report our findings in Figure 16. The plot indicates that OMEGA manages to improve the utilization of

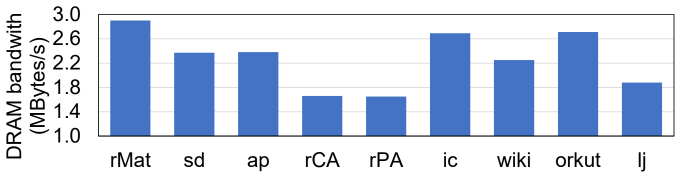


Fig. 16. **DRAM bandwidth utilization of *PageRank***. OMEGA improves off-chip bandwidth utilization by 2.28x, on average.

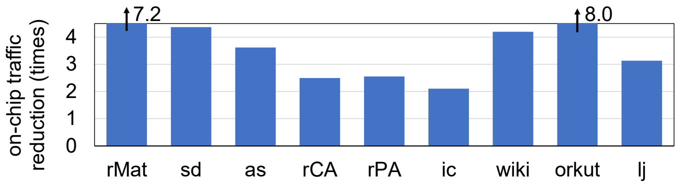


Fig. 17. **On-chip traffic analysis of *PageRank***. OMEGA reduces on-chip traffic by 3.2x, on average.

off-chip bandwidth by an average of 2.28x. Note that there is a strong correlation between the bandwidth utilization and the speedup reported in Figure 14 for *PageRank*. Indeed, the bandwidth improvement can be attributed to two key traits of OMEGA: 1) cores are freed to streamline more memory requests because atomic instructions are offloaded to the PISCs, and 2) since most of the random accesses are constrained to the OMEGA's scratchpads, the cores can issue more sequential accesses to the *edgeList* data structure. Furthermore, we found that graph workloads create lots of on-chip communication traffic. Our analysis measuring on-chip traffic volume, reported in Figure 17, show that OMEGA reduces this traffic by over 4x on average. OMEGA minimizes on-chip communication by employing word-level access to scratchpad data and offloading operations to the PISCs.

Non-power-law graphs. Figure 18 presents a speedup comparison for two large graphs: one for a power-law graph (*lj*) and another for a non-power-law graph (*USA*) on two representative algorithms: *PageRank* (graph statistics algorithm with no *active-list*) and *BFS* (graph traversal algorithm with *active-list*). OMEGA's benefit for *USA* is limited, providing a maximum of 1.15x improvement. The reason is that, since *USA* is a non-power-law graph, only approximately 20% of the *vtxProp* accesses hit the 20% most-connected vertices, compared to 77% for *lj*.

Scratchpad size sensitivity. In our analyses so far, OMEGA is configured with scratchpad-sizes that enables it to accommodate around 20% or more of the *vtxProp*. In this section, we present a scratchpad-size sensitivity study for *PageRank* and *BFS* on the *lj* dataset, over a range of scratchpad sizes: 16MB (our experimental setup), 8MB, and 4MB. We kept the size of the L2 cache the same as in our experimental setup (16MB) for all configurations. The results, reported in Figure 19, show that OMEGA managed to still provide a 1.4x speedup for *PageRank* and a 1.5x speedup for *BFS* even when employing only 4MB scratchpads, which accommodate only 10% of the *vtxProp* for *PageRank* and 20% of the *vtxProp* for *BFS*. As shown in the figure, 10% of the vertices are responsible for 60.3% of *vtxProp* for *PR*, and 20% of the vertices are responsible for 77.2% of the *vtxProp* for *BFS*. Note that this solution point entails significantly less storage than the total L2 cache of our baseline.

Scalability to large datasets. This study estimates the performance of OMEGA on very large datasets: *uk* and *twitter*. Since we could not carry out an accurate simulation, due to the limited performance of gem5, we modeled both the

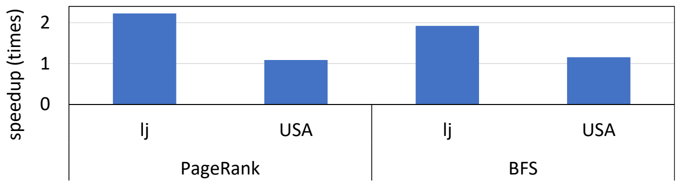


Fig. 18. **Comparison of power-law (*lj*) and non-power-law graphs (*USA*)**. As expected, OMEGA achieves only a limited speedup of 1.15x on a large non-power-law graph.

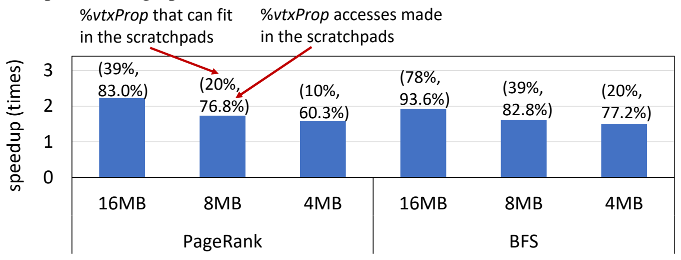


Fig. 19. **Scratchpad sensitivity study**. OMEGA provides 1.4x speedup for *PageRank* and 1.5x speedup for *BFS* with only 4MB of scratchpad storage.

baseline and OMEGA in a high-level simulator. In the simulator, we keep the same number of cores, PISC units, and scratchpad sizes as in Table III. In addition, we make two key approximations. First, the number of DRAM accesses for *vtxProp* is estimated based on the average LLC hit-rate that we obtained by running each workload on the Intel Xeon E5-2630 v3 processor and using the Intel's VTune tool. The number of cycles to reach DRAM is set at 100 cycles and we also accounted for the LLC and scratchpad access latencies. Second, the number of cycles for a remote scratchpad access is set at 17 cycles, corresponding to the average latency of the crossbar interconnect. For the baseline solution, we configured the number of cycles required to complete an atomic operation execution to match the value we measured for the PISC engines: this is a conservative approach, as the baseline's CMP cores usually take more cycles than the PISC engines, being more complex units. Figure 20 reports our findings. In the figure, we also include the result from our gem5 evaluation for validation purposes. We note that the high-level estimates are within a 7% error, compared to the gem5 results. As shown in the Figure, OMEGA achieves significant speedup for the two very large graphs, even if they would benefit from much larger scratchpad resources. For instance, for *twitter*, OMEGA manages to provide a 1.7x speedup on *PageRank* even if only providing storage for 5% of the *vtxProp*. Note that 5% of the most-connected vertices are responsible for 47% of the total *vtxProp* accesses. This highly-skewed connectivity is the reason why OMEGA is able to provide a valuable speedup even with relatively small specialized storage.

B. *Area, power, and energy analysis*

We used McPAT [23] to model the core and Cacti [37] to model the scratchpads and caches. We synthesized PISC's

TABLE IV
PEAK POWER AND AREA FOR A CMP AND OMEGA NODE

Component	**Baseline CMP node**		**OMEGA node**	
	Power (W)	Area (mm^2)	Power (W)	Area (mm^2)
Core	3.11	24.08	3.11	24.08
L1 caches	0.20	0.42	0.20	0.42
Scratchpad	N/A	N/A	1.40	3.17
PISC	N/A	N/A	0.004	0.01
L2 cache	2.86	8.41	1.50	4.47
Node total	6.17	32.91	6.21	32.15

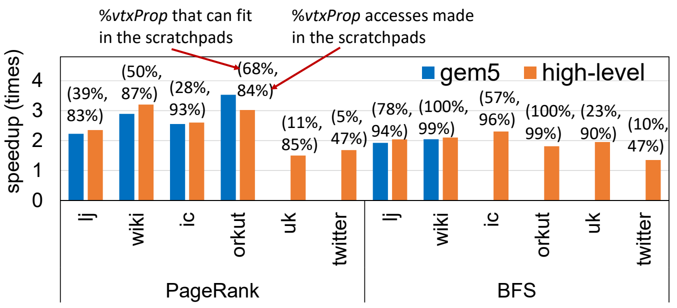


Fig. 20. **Performance on large datasets**. A high-level analysis reveals that OMEGA can provide significant speedups even for very large graphs: a 1.68x for *PageRank* runnning on *twitter*, our largest graph, when storing only 5% of *vtxProp* in scratchpads; and a 1.35x for *BFS*, storing only 10% of *vtxProp*.

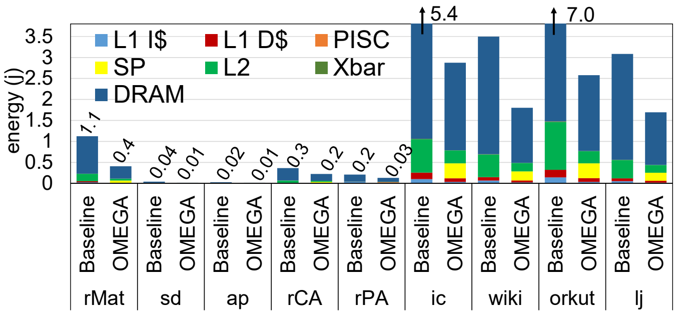


Fig. 21. **Comparison of energy spent in memory activities for *PageRank***. OMEGA requires less energy to complete the algorithm due to less DRAM traffic and shorter execution time. The energy efficiency of OMEGA's scratchpads over the caches contributes to the overall energy savings.

logic in IBM 45nm SOI technology. Note that the PISC's area and power is dominated by its floating-point adder. We referred to prior works for the crossbar model [3]. We used the same technology node, 45nm, for all of our components. We re-purposed half of the baseline's L2 caches space to OMEGA's on-chip scratchpads. Table IV shows the breakdown of area and peak power for both the baseline CMP and OMEGA. The OMEGA node occupies a slightly lower area (-2.31%) and consumes slightly higher in peak power (+0.65%) compared to the baseline CMP. The slightly lower area is due to OMEGA's scratchpads being directly mapped and thus not requiring cache tag information.

Our energy analysis on *PageRank* running a wide-range of datasets reveals that OMEGA provides 2.5x energy saving, on average, compared to a CMP-only baseline. Since an OMEGA node consumes roughly the same peak power as a baseline CMP node, it is natural to expect that OMEGA's performance benefits translate to energy saving. Since OMEGA's modifications to the baseline are limited to the memory hierarchy, in Figure 21, we provide a breakdown of the energy consumption only for the memory system, including the DRAM. Across the different datasets, OMEGA provides significant energy savings in the memory subsystem, as well as in the off-chip memory system. As shown in the figure, OMEGA's scratchpads consume less energy compared to caches. OMEGA also uses less DRAM energy because most of the accesses to *vtxProp* are served from on-chip scratchpads.

XI. Related work

Table V compares OMEGA against prior works. OMEGA is better than previous architectures primarily because it exploits the power-law characteristics of many natural graphs to identify the most-accessed portions of the *vtxProp* and it utilizes scratchpads for their efficient access. Furthermore, most of the atomic operations on *vtxProp* are executed on lightweight PISC engines instead of CPU cores. OMEGA is easily deployable in new graph frameworks and applications because of it maintains the existing general-purpose architecture, and simply provide some small and effective additional components. We believe that OMEGA is the first architecture to address both on- and off-chip communication costs by exploiting the power-law characteristics of many natural graphs, without requiring changes in the application.

Optimizations on general-purpose architectures. [10] characterizes graph workloads on Intel's Ivy Bridge server. It showed that locality exists in many graphs, which we leverage in our work. [14] minimizes the overhead of synchronization operations for graph applications on a shared-memory architecture, by moving computation to dedicated threads. However, dedicated threads for computation would provide a lower performance/energy efficiency compared to our lightweight PISC architecture. [45] proposes both graph reordering and segmentation techniques for maximizing cache utilization. The former provides limited benefits for natural graphs, and the latter requires modifying the framework, which we strive to avoid. [46] proposes domain-specific languages, thus trading off application's flexibility for higher performance benefits.

Near-memory processing and instruction offloading. [6], [28] propose to execute all atomic operations on *vtxProp* in the off-chip memory because of the irregular nature of graph applications. However, our work shows that the locality within the structure of many natural graphs can be exploited using on-chip scratchpads. Scratchpads have lower latency and energy consumption per-access, compared to off-chip memory.

Domain-specific and specialized architecture. Application-specific and domain-specific architectures, including those that utilize scratchpads [5], [19], [33], [39], have recently flourished to address the increasing need of highly efficient graph analytics solutions. However, the focus is on performance/energy efficiency, foregoing the ability to be amenable to modifications in frameworks and applications. In addition, these solutions do not fully exploit the locality in the structure of many natural graphs.

GPU and vector processing solutions. GPU and other vector processing solutions, such as [4], [32], [42], have been increasingly adopted for graph processing, mostly in the form of sparse matrix-vector multiplication. However, the diverse structure of graphs limits the viability of such architectures.

Heterogeneous cache block size architecture. [20] employed variable cache-line sizes based on the runtime behavior of an application. Similarly, OMEGA provides variable storage access sizes (cache-line-size for accessing caches and word-size for accessing scratchpads) depending on the type of data structure being accessed.

XII. Conclusion

In this work, we optimize the memory subsystem of a general-purpose processor to run graph frameworks without requiring significant additional changes from application developers. We expose the inherent locality contained in natural graphs to our design, providing significantly more on-chip accesses for irregularly accessed data. In addition, we augment the on-chip distributed scratchpads with atomic operation processing engines, providing significant performance gains. Our solution achieves on average a 2x boost in performance and a 2.5x energy savings, compared to a same-sized baseline CMP running a state-of-the-art shared-memory graph framework. The area and peak power needs of our solutions are

TABLE V
COMPARISON OF OMEGA AND PRIOR RELATED WORKS

	CPU	**GPU**	**Locality Exists [10]**	**Graphicionado [19]**	**Tesseract [5]**	**GraphIt [46]**	**GraphPIM [28]**	**OMEGA**
leveraging power law	limited	limited	**yes**	no	no	limited	no	**yes**
memory subsystem	cache	cache	cache	**scratchpad**	cache	cache	cache	**cache & scratchpad**
logic for non-atomic-operation	**general**	**general**	**general**	specialized	**general**	**general**	**general**	**general**
logic for atomic-operation	general	general	general	**specialized**	general	general	**specialized**	**specialized**
on-chip communication granularity	cache-line	cache-line	cache-line	**word**	**word**	cache-line	**word**	**cache-line & word**
offloading target for atomic operation	N/A	N/A	N/A	**scratchpad**	off-chip memory	N/A	off-chip memory	**scratchpad**
compute units for atomic operation	CPU	GPU	CPU	**specialized**	CPU	CPU	**specialized**	**CPU & specialized**
framework independence&modifiability	**yes**	**yes**	**yes**	limited	**yes**	limited	**yes**	**yes**
propose software-level optimizations	**yes**	**yes**	partially	no	no	**yes**	no	no

comparable to that of the baseline node, as we trade cache storage for equivalently sized scratchpads.

ACKNOWLEDGMENTS

This work was supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

REFERENCES

[1] “9th DIMACS Implementation Challenge ,” http://www.dis.uniroma1.it/challenge9/, May 2018.

[2] “WebGraph,” http://webgraph.di.unimi.it/, May 2018.

[3] N. Abeyratne, R. Das, Q. Li, K. Sewell, B. Giridhar, R. G. Dreslinski, D. Blaauw, and T. Mudge, “Scaling towards kilo-core processors with asymmetric high-radix topologies,” in *Proc. HPCA*, 2013.

[4] M. Ahmad and O. Khan, “GPU concurrency choices in graph analytics,” in *Proc. IISWC*, 2016.

[5] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A scalable processing-in-memory accelerator for parallel graph processing,” in *Proc. ISCA*, 2015.

[6] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, “PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture,” in *Proc. ISCA*, 2015.

[7] A. Airola, S. Pyysalo, J. Björne, T. Pahikkala, F. Ginter, and T. Salakoski, “All-paths graph kernel for protein-protein interaction extraction with evaluation of cross-corpus learning,” *BMC bioinformatics*, 2008.

[8] R. Albert, H. Jeong, and A.-L. Barabási, “Internet: Diameter of the world-wide web,” *nature*, 1999.

[9] A.-L. Barabási and R. Albert, “Emergence of scaling in random networks,” *science*, 1999.

[10] S. Beamer, K. Asanovi, and D. Patterson, “Locality exists in graph processing: Workload characterization on an ivy bridge server,” in *Proc. IISWC*, 2015.

[11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. Hill, and D. Wood, “The gem5 simulator,” *ACM SIGARCH Computer Architecture News*, 2011.

[12] D. Chakrabarti, Y. Zhan, and C. Faloutsos, “R-MAT: A recursive model for graph mining,” in *Proc. ICDM*, 2004.

[13] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, “One trillion edges: Graph processing at facebook-scale,” *Proceedings of the VLDB Endowment*, 2015.

[14] H. Dogan, F. Hijaz, M. Ahmad, B. Kahne, P. Wilson, and O. Khan, “Accelerating graph and machine learning workloads using a shared memory multicore architecture with auxiliary support for in-hardware explicit messaging,” in *Proc. IPDPS*, 2017.

[15] V. M. Eguiluz, D. R. Chialvo, G. A. Cecchi, M. Baliki, and A. V. Apkarian, “Scale-free brain functional networks,” *Physical review letters*, 2005.

[16] M. Faloutsos, P. Faloutsos, and C. Faloutsos, “On power-law relationships of the internet topology,” in *ACM SIGCOMM computer communication review*, 1999.

[17] A. V. Goldberg and C. Harrelson, “Computing the shortest path: A search meets graph theory,” in *Proc. SODA*, 2005.

[18] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, “Powergraph: Distributed graph-parallel computation on natural graphs,” in *Proc. OSDI*, 2012.

[19] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, “Graphicionado: A high-performance and energy-efficient accelerator for graph analytics,” in *Proc. MICRO*, 2016.

[20] K. Inoue, K. Kai, and K. Murakami, “Dynamically variable line-size cache architecture for merged DRAM/Logic LSIs,” *IEICE Transactions on Information and Systems*, 2000.

[21] A. Kyrola, G. E. Blelloch, and C. Guestrin, “Graphchi: Large-scale graph computation on just a pc.” USENIX, 2012.

[22] J. Leskovec and A. Krevl, “SNAP Datasets: Stanford large network dataset collection,” http://snap.stanford.edu/data, May 2018.

[23] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures,” in *Proc. MICRO*, 2009.

[24] Y. Lim, U. Kang, and C. Faloutsos, “Slashburn: Graph compression and mining beyond caveman communities,” *IEEE Transactions on Knowledge and Data Engineering*, 2014.

[25] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, “MOSAIC: Processing a Trillion-Edge Graph on a Single Machine,” *EuroSys*, 2017.

[26] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: a system for large-scale graph processing,” in *Proc. SIGMOD*, 2010.

[27] A. Mukkara, N. Beckmann, and D. Sanchez, “Cache-Guided Scheduling: Exploiting Caches to Maximize Locality in Graph Processing,” *1st International Workshop on Architecture for Graph Processing*, 2017.

[28] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, “GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks,” in *Proc. HPCA*, 2017.

[29] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, “GraphBIG: understanding graph computing in the context of industrial solutions,” in *High Performance Computing, Networking, Storage and Analysis*, 2015.

[30] M. E. Newman, “Power laws, Pareto distributions and Zipf’s law,” *Contemporary physics*, 2005.

[31] D. Nicolaescu, X. Ji, A. Veidenbaum, A. Nicolau, and R. Gupta, “Compiler-Directed Cache Line Size Adaptivity,” in *Intelligent Memory Systems*, 2001.

[32] A. H. Nodehi Sabet, J. Qiu, and Z. Zhao, “Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing,” in *Proc. ASPLOS*, 2018.

[33] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, “Energy efficient architecture for graph analytics accelerators,” in *Proc. ISCA*, 2016.

[34] L. Page, S. Brin, R. Motwani, and T. Winograd, “The PageRank citation ranking: Bringing order to the web,” Stanford InfoLab, Tech. Rep., 1999.

[35] I. Puaut and C. Pais, “Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison,” in *Proc. DATE*, 2007.

[36] A. Roy, I. Mihailovic, and W. Zwaenepoel, “X-stream: Edge-centric graph processing using streaming partitions,” in *Proc. SOSP*, 2013.

[37] P. Shivakumar and N. P. Jouppi, “Cacti 3.0: An integrated cache timing, power, and area model,” 2001.

[38] J. Shun and G. E. Blelloch, “Ligra: a lightweight graph processing framework for shared memory,” in *ACM Sigplan Notices*, 2013.

[39] S. G. Singapura, A. Srivastava, R. Kannan, and V. K. Prasanna, “OSCAR: Optimizing SCrAtchpad reuse for graph processing,” in *Proc. HPEC*, 2017.

[40] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, “Graphmat: High performance graph analytics made productive,” *Proceedings of the VLDB Endowment*, 2015.

[41] A. V. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji, “Adapting cache line size to application behavior,” in *Proc. ICS*, 1999.

[42] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, “Gunrock: A high-performance graph processing library on the GPU,” in *ACM SIGPLAN Notices*, 2016.

[43] A. Yasin, “A top-down method for performance analysis and counters architecture,” in *Proc. ISPASS*, 2014.

[44] K. Zhang, R. Chen, and H. Chen, “NUMA-aware graph-structured analytics,” *ACM SIGPLAN Notices*, 2015.

[45] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, “Making caches work for graph analytics,” in *Proc. Big Data*, 2017.

[46] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, “GraphIt-A High-Performance DSL for Graph Analytics,” *arXiv preprint arXiv:1805.00923*, 2018.

[47] X. Zhu, W. Han, and W. Chen, “GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning,” in *USENIX Annual Technical Conference*, 2015.