Overcoming the Memory Hierarchy Inefficiencies in Graph Processing Applications

Jilan Lin, Shuangchen Li, Yufei Ding, Yuan Xie
University of California, Santa Barbara, Santa Barbara 93117, USA
{jilan, shuangchenli, yufeiding, yuanxie}@ucsb.edu

***Abstract*—Graph processing participates a vital role in mining relational data. However, the intensive but inefficient memory accesses make graph processing applications severely bottlenecked by the conventional memory hierarchy. In this work, we focus on inefficiencies that exist on both on-chip cache and off-chip memory. First, graph processing is known dominated by expensive random accesses, which are difficult to be captured by conventional cache and prefetcher architectures, leading to low cache hits and exhausting main memory visits. Second, the off-chip bandwidth is further underutilized by the small data granularity. Because each vertex/edge data in the graph only needs 4-8B, which is much smaller than the memory access granularity of 64B. Thus, lots of bandwidth is wasted fetching unnecessary data.**

Therefore, we present G-MEM, a customized memory hierarchy design for graph processing applications. First, we propose a coherence-free scratchpad as the on-chip memory, which leverages the power-law characteristic of graphs and only stores those hot data that are frequent-accessed. We equip the scratchpad memory with a degree-aware mapping strategy to better manage it for various applications. On the other hand, we design an elastic-granularity DRAM (EG-DRAM) to facilitate the main memory access. The EG-DRAM is based on near-data processing architecture, which processes and coalesces multiple fine-grained memory accesses together to maximize bandwidth efficiency. Putting them together, the G-MEM demonstrates a 2.48× overall speedup over a vanilla CPU, with 1.44× and 1.79× speedup against the state-of-the-art cache architecture and memory subsystem, respectively.

I. Introduction

In many application domains, we use graphs to abstract the data of interests and excavate information from them, such as social networks, financial transactions, and knowledge databases [1], [2], [3], [4]. Graph processing thus emerges as an important workload. Although graph processing algorithms could vary depending on different application scenarios, they usually suffer from the same bottleneck: the dominated and expensive memory accesses.

We identify that there are two challenging inefficiencies in the existing memory hierarchy for graph processing: on-chip cache inefficiency and off-chip bandwidth inefficiency. (a) The **on-chip cache inefficiency** is caused by the poor locality in graph processing. Graph processing is widely known to be filled with random memory access, due to the unpredictable and sophisticated connections between vertices [5]. Since traditional cache and prefetcher architecture are usually designed for patterned accesses, they fail to capture the randomly accessed data desired by graph processing applications. Lots of work has reported the low cache hits of 10% 40% in graph applications [6], [7], since the traditional cache hierarchy is difficult to predict and capture the random access pattern. (b) The **off-chip bandwidth inefficiency** comes from the small data granularity in graph processing. First, traditional graph processing algorithms usually assign 4B/8B integers or floats to represent the vertex data [8], [9], which are reported to take more than 80% of the memory accesses [10]. However, current commercial DRAM memory generally supports the accessing granularity of 64B that aligns with the cacheline size [11], and the 4B/8B data appears to be all we need from the 64B cacheline due to the poor locality. Therefore, lots of off-chip memory bandwidth is wasted fetching unnecessary data.

Extensive efforts have been made toward improving the efficiency of memory hierarchy. From the on-chip cache side, researchers have found that nature graphs are well-known to be power-law distributed [12], [13], meaning few high-degree vertices are frequently accessed. Prior work managed to leverage this interesting property: OMEGA [14] designed a distributed scratchpad to store high-degree vertices, requiring a fixed data flow and programming model and thus losing the flexibility to process various graph applications; GRASP [10] customized the existing cache to enable the ability to identify and store such vertices, but the time-consuming graph reordering highly limits the performance. From the off-chip memory perspective, prior work sought system-level solutions by adopting specialized dataflows, which takes a more sequential access pattern to alleviate the fine-grained off-chip traffic. For instance, the X-STREAM [15] proposed an edge-centric processing flow that streams through all edges to update the source and destination vertices, and thus the off-chip accesses are streaming-patterned. However, the changed processing flow could introduce a large overhead of reading unnecessary data. Generally, not all edges are active during one iteration. Prior work has reported that there are only 10%-20% vertices remaining active [16], indicating lots of edges are not needed in the edge-centric processing.

To achieve both high flexibility and performance, we present G-MEM, a customized memory hierarchy design for graph processing. We design the G-MEM based on two insights: First, as specialized cache architectures could limit the application flexibility or bring significant hardware cost, a software-managed on-chip memory is leveraged to facilitate access to the hot vertices in a graph. Second, the coarse-grained data in the memory bus comes from multiple DRAM devices in a DRAM DIMM, we could expect finer-grained accesses with

This work was supported in part by NSF 1816833 and 1719160.

fewer DRAM devices. Specifically, we design a coherence-free scratchpad on-chip memory, since the power-law characteristic of graphs is hardly captured by the traditional cache hierarchy. To facilitate the use of the scratchpad memory, we further propose a graph-aware mapping strategy to manage the scratchpad from the software level. In addition, we design an elastic-granularity memory subsystem based on near-data processing (NDP) architecture that has been widely studied in prior work. We make up our elastic DRAM DIMM with off-the-shelf DRAM devices, and each DRAM device provides 8B data access. We equip the DIMM with a gather-scatter unit, which processes and coalesces multiple fine-grained memory accesses together to maximize the bandwidth efficiency.

We summarize the contributions of this paper as follows:

- We design a coherence-free scratchpad as the on-chip memory, coming with a proposed degree-aware vertex remapping strategy to exploit the graph's power low nature (Section III-B).
- We propose an off-chip memory composed of elastic-granularity DRAMs (EG-DRAMs) to coalesce fine-grained accesses and boost the bandwidth efficiency (Section III-C).
- We optimize the extension in the memory controller and the DDR4 protocol, to save the bandwidth demand and area overhead of the EG-DRAM (Section III-D).
- The Sniper-based [17] simulation demonstrates a 2.48× overall speedup over a vanilla CPU, with 1.44× and 1.79× speedup against the state-of-the-art cache architecture and memory subsystem, respectively.

II. Preliminaries

In this section, we introduce the preliminaries for future discussions, including graph processing basics, characterizations of graph workloads, and the related work.

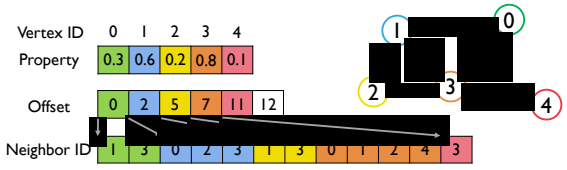

Fig. 1. A CSR graph formatted with 3 arrays: Property array denotes the value in each vertex; Offset array denotes the index of the vertex's neighbors, which locate at the Neighbor ID array.

A. Graph Processing Basics

Graph Data Layout: Graphs are usually stored vertex by vertex in a dense format, with neighbors attached to each vertex, since the edge connectivity between vertices is very sparse. For example, compressed Sparse Row (CSR) is a common-used data structure to represent a graph [18], as shown in Fig. 1. The property array stores the value of each vertex, for example, the PageRank score in the PageRank algorithm or the vertex depth in the Breadth-First-Search algorithm. The structural information is stored in 2 arrays: Offset array and Neighbor ID array. The offset array records the begin and end indices of the neighbors for each node, which can be found from the neighbor ID array.

Power-law Distribution: Natural graphs usually has a unique feature: power-law degree distribution [12], [13]. This means most vertices have relatively few neighbors while only a few have many neighbors (e.g., celebrities in a social network). A power-law degree probability distribution is formulated by:

$$P(d) \sim d^{-\alpha} \tag{1}$$

where the $P(d)$ denotes the probability that a vertex has a degree d, and the exponent α indicates "skewness" of the distribution (higher α means more vertices have low degrees).

Parallel Graph Processing: Graph workloads are traditionally processed in either a vertex-centric or an edge-centric manner [19], [20]. In vertex-centric processing, the property of a vertex is calculated in accordance with its neighbors' properties. Updating properties can be done by computing all vertices, or propagating from vertices that have their properties changed to neighbors. This process is usually iterative until convergence. Similarly, in edge-centric processing, the computation goes over edges instead of vertices, where the update is achieved through pushing the information from the source to the destination over a particular edge. Edge-centric processing brings more localities when streaming edges, but may also introduce extra data traffic as high-degree vertices can be read many times without cached.

Both vertex-centric and edge-centric processing can be accelerated by exploiting parallelism. We can spawn parallel threads to process individual vertices in the vertex-centric processing, or different edges in the edge-centric processing. Thus, lots of parallel graph processing frameworks and benchmarks are proposed accordingly, based on various parallel programming libraries such as OpenMP (multi-thread) and MPI (multi-machine) [21], [8], [19], [15].

B. Memory Hierarchy Inefficiencies

In this section, we introduce the characterizations of graph workloads to identify the memory hierarchy bottleneck.

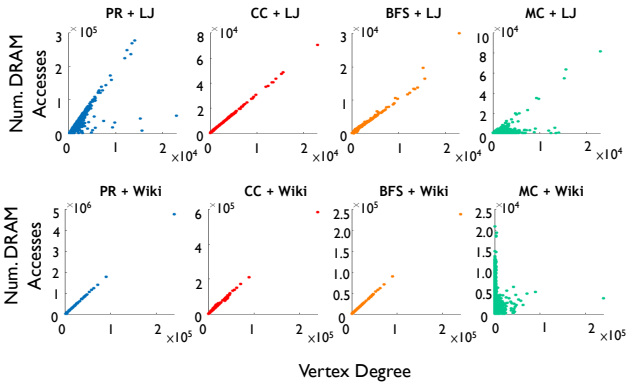

Fig. 2. The relationship between the vertex degree (shown in x-axis) and the number of memory accesses to it (shown in y-axis). Four graph applications are evaluated on two datasets: LiveJournal and Wiki.

First, *random access nature severely degrades the performance of on-chip caches.* Lots of work has reported the low

cache hits in graph applications [6], [10], since traditional caches are difficult to predict and capture the random access pattern. Typically, a cache hit ratio of 10%-20% is observed in the L2 cache, while a 30%-60% hit is observed in the LLC [6], [10]. Due to the power-law property, few high-degree vertices are frequently accessed. As shown in Fig. 2, we find the number of accesses to each vertex is literally linearly related to its degree. This gives us a hint that the on-chip memory should leverage the native feature from graph datasets.

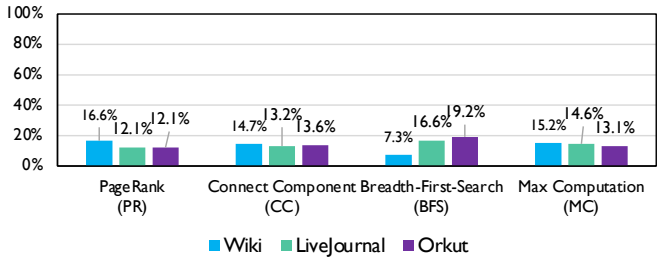

Fig. 3. The bandwidth efficiency on the 8 graph processing workloads, which is measured as average touched data in a cacheline.

Second, *a cacheline is underutilized due to the fine-grained memory accesses pattern.* Modern DDR4 DRAM is designed to comply with a 64B cacheline in caches. However, previous studies revealed that the granularity of memory accesses in graph applications can be as small as 4B or 8B [9], [7]. To quantitatively evaluate the exact granularity that graph applications demand, we investigate what percentage of the 64B cacheline is actually torched. As demonstrated in Fig 3, we extracted the memory traces from various graph applications and derived the percentage of used cacheline by averaging the touched data size within a time window. Besides, the vertex property data is set to be 4B. We find that even with a large memory window of 128, only less than 20% of 64B data are used. This means more than 80% of the memory bandwidth is wasted in fetching unnecessary data.

C. Related Work

Cache and Prefetcher Architectures: Prior work has broadly discussed the opportunity of graph-specialized caches and prefetchers to improve the data locality for on-chip memory. OMEGA [14] is a distributed scratchpad memory to store high-degree vertices, requiring a fixed data flow and programming model to process the vertex mapping and communication; GRASP [10] extended the existing cache to enable the ability to identify and manage these hot vertices, but it requires the time-consuming graph reordering. On the other hand, indirect memory prefetcher (IMP) [22] is designed for applications that exhibit pointer-chasing behavior. But IMP could prefetch lots of unnecessary data. DROPLET [6] further improved IMP with dedicated units to identify the property, offset, and neighbor ID data explicitly, and thus fetches those vertices associated with currently cached vertices more precisely. HATS [5] also speculates the vertex data processed in the core and issues prefetches over the entire community of these vertices. Both DROPLET and HATS require a fixed data structure in the application, such that the hardware could directly recognize and operate on the graph data and generate prefetching commands.

	OMEGA [14]	GRASP [10]	IMP [22]	DROPLET [6]	HATS [5]	**G-MEM (ours)**
Flexiblity	×	✓	✓	×	×	✓
Performance	✓	×	×	✓	✓	✓

TABLE I
COMPARISON OF G-MEM'S ON-CHIP MEMORY WITH PRIOR WORK.

G-MEM differs from the prior work by achieving both high flexibility and performance, as shown in Table I. We put a general-purpose scratchpad as the on-chip memory, which does not require one particular data layout and data flow and could fit into various application scenarios. Instead, we design a data allocation strategy based on a lightweight reordering algorithm to manage the scratchpad from the software level.

Memory Subsystem Design: There are relatively fewer studies on a specialized off-chip memory to accelerate graph processing. Dynamic-Granularity Memory Subsystem (DGMS) [23] and Adaptive-Granularity Memory Subsystem (AGMS) [24] leveraged narrowed memory bus to achieve fine-grained memory accesses. As a memory channel contains both command/address (C/A) bus and data bus, such designs only narrow the width of data bus but still need the same C/A bus. Therefore, the C/A bus then becomes a huge overhead considering the limited pin fan-out from the chip. We will explain this in detail in Section 3.2. On the other hand, Gather-Scatter DRAM [25] designs a DRAM memory that is able to perform strided gather/scatter memory access. However, as the conventional strided prefetcher does not work for the random memory accesses, this fix-pattern gather-scatter DRAM is not preferable for graph processing workloads.

G-MEM leverages the recent near-data processing (NDP) technology (specifically DRAM-based NDP technology), which puts a special-function unit on the DRAM DIMM to operate data near the memory and meet different application demands [26], [27], [28]. The key idea is to pack the sophisticated and fine-grained memory requests together on the DRAM DIMM and send them back to the processor. Therefore, G-MEM boosts the bandwidth efficiency without introducing overhead to the system bandwidth.

III. G-MEM ARCHITECTURE

In this section, we present the architecture design of G-MEM. We first give an overview of G-MEM, followed by the coherence-free scratchpad design, degree-aware vertex remapping scheme, and the NDP-based elastic DRAM memory.

A. Overview

Fig. 4 presents an overview of the G-MEM architecture, which includes the on-chip coherence-free scratchpad (CF-scratchpad) and off-chip elastic-granularity DRAM (EG-DRAM). First, we make the scratchpad memory coherence-free by having it share the memory space with DRAM. Therefore, each core can access it with a designated address space without maintaining the coherence between the scratchpad and the DRAM. We further propose a degree-aware vertex remapping scheme to manage the scratchpad from the software

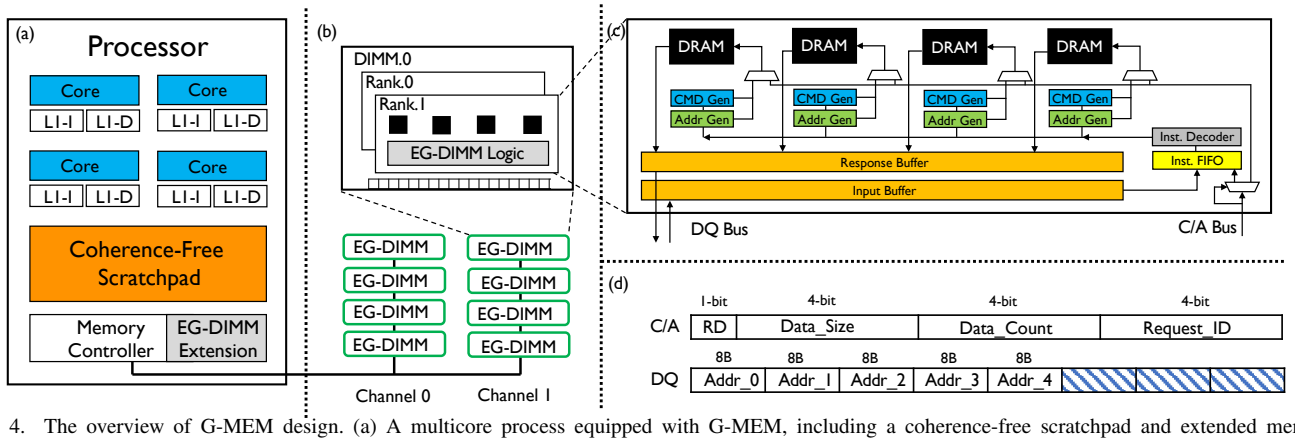

Fig. 4. The overview of G-MEM design. (a) A multicore process equipped with G-MEM, including a coherence-free scratchpad and extended memory controller. (b) The elastic-granularity (EG) memory channels connected to the processor. (c) The microarchitecture of the elastic-granularity DRAM, which achieves fine-grained access by separating DRAM devices. (d) The instruction format of EG-DRAM, with packing multiple memory requests together.

level. Second, we design the elastic-granularity DRAM as the off-chip memory. We leverage the near-data processing (NDP) technology to access the on-DIMM DRAM devices individually and achieve fine-grained data access. We also design the instruction format by leveraging the unused address line and data line in the PRECHARGE command to ensure compatibility with the commodity DDR4 interface.

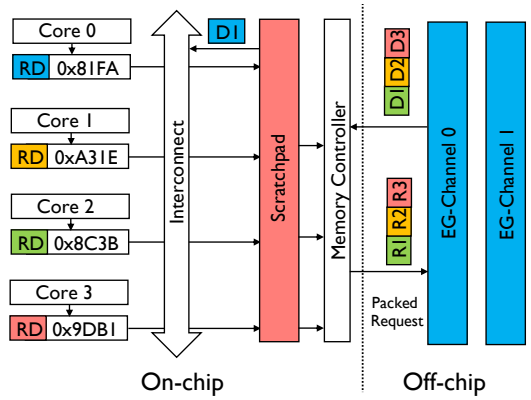

Fig. 5. The data flow in the G-MEM hierarchy. Four read requests are issued from cores, where one is served by the on-chip scratchpad and the other three go through the off-chip EG-memory.

Fig. 5 gives an illustrative data flow in G-MEM. Four read requests to different fine-grained data are issued from cores, which transit through the interconnection between core and scratchpad. The scratchpad identifies that the address of request #1 locates within itself and sends the desired data back. Then, the other three requests are forwarded to the memory controller. With the EG-DRAM extension unit, the memory controller packs those requests together and sends them to the off-chip EG-DRAM. Thus, three reads could be served with only one memory walk to boost the bandwidth efficiency.

B. Coherence-Free Scratchpad

In this section, we introduce our coherence-free scratchpad that locates in the same memory address space with DRAM, We then present how is the scratchpad memory managed at the software level.

1) Opportunity of using scratchpad memory: Concerning that some higher-degree vertices are accessed much more frequently than other vertices and hardware prefetchers are unlikely to predict such memory access pattern, a scratchpad memory that is managed at the software level is more appealing to store these high-degree vertices. Also, compared with a large shared cache, the scratchpad memory brings the benefits of easier hardware implementations and no tag access/hierarchy traverse time, meaning a smaller area overhead and lower access latency. A similar idea has been exploited in OMEGA [14], which equips each core with a scratchpad memory to store high-degree vertices by reducing the cache size. However, we here focus on a more general and flexible design and minimizing the traffic between scratchpad and main memory.

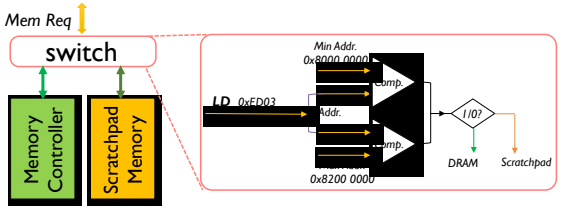

Fig. 6. Coherence-Free Scratchpad

2) Avoiding coherency by memory space partitioning.: To eliminate the coherence between scratchpad and DRAM and reduce the bandwidth demand, we propose a heterogeneous memory subsystem consisting of both scratchpad and DRAM, meaning the scratchpad shares the same address space with DRAM and works as a fast main memory.

As shown in Fig. 6, first, when allocating the memory addresses, the memory management unit records a reserved memory space that is as large as the scratchpad memory. For instance, a 32MB scratchpad memory takes the address from 0x80000000 to 0x82000000, and any data allocated within this address belongs to the scratchpad. Then, when a memory request comes down to the memory controller, a switch determines to either route the request to the scratchpad or bypass it to the DRAM controller only according to the address of this request. Therefore, we make the scratchpad

transparent to the program at run-time, and no special control of different data flows is required.

3) Graph-Aware Scratchpad Management: The system-level management of the scratchpad memory such as memory allocating mentioned above can be achieved by the memory management unit(MMU) [29], [30] where the scratchpad can be treated as another piece of memory in a similar way as the non-uniform memory access (NUMA) in multi-socket CPU [31]. We would like to explain more in detail about the software-level managing for the scratchpad memory. For large-scale graphs, we can only expect a small portion of vertices can fit into the on-chip scratchpad memory. As discussed in Section II-B, this small portion is preferable to be the high-degree vertices that are quite frequently accessed. However, the problem is these vertices are not stored continuously and also randomly exist in the node array.

Key idea: We propose a degree-aware vertex remapping scheme to tackle the issue of random distributed high-degree vertices. The scheme manages to exchange high-degree vertices to the front tail of the vertex array, such that they would locate in sequential address space in the scratchpad. For example, for a graph with 100 vertices, the vertex 0-9 are almost high-degree vertex and locate in the scratchpad while vertex 10-99 with relative lower degrees remain in the DRAM. In addition, since we do not essentially need to sort the vertices but just separate them instead, the algorithm only needs to scan over the vertex array and identify a high-degree vertex by a threshold, so heavy reordering is not required.

As illustrate in Algorithm 1, we maintain 2 vertex pointers, p_{low} and p_{high}, for recording the current low-degree vertex and searching the next high-degree vertex, respectively. A degree threshold is used to determine whether a vertex is high-degree or low-degree. As we find the expected p_{low} within the scratchpad region and one p_{high} outside the scratchpad, we switch the properties of these 2 vertices and pair them into a map. The algorithm terminates when we go to the end of the scratchpad or the total number of vertices, indicating that we already have enough hot vertices in the scratchpad or we cannot find any more hot vertices. In terms of the algorithm complexity, in the worst case, it takes at most $O(N) + O(E \cdot S)$ time under the condition that we switch out all the vertices in the scratchpad. However, since it is unlikely to face such many high-degree vertices and the size of the scratchpad is relatively small, the computation complexity in normal cases is around $C \cdot O(E)$. Note that, existing pre-processing frameworks usually provide techniques like vertex sorting [8]. Our proposed vertex ID remapping strategy can be easily embedded into the pre-processing step.

Choice of degree threshold: According to prior study on power-law distribution, nature graphs usually have a skewness factor $\alpha \approx 2$ [32]. This means we can pick a fixed degree threshold for most graphs with such prior knowledge. For a specific scratchpad size and graph statistics (number of vertices), we can decide what percentage of vertices can be loaded into the scratchpad. And thus, the threshold is selected according to Eq. (1). On the other hand, the choice of threshold is not necessarily precise, since we only need to cache part of the hot vertices into the scratchpad.

Algorithm 1: Degree-Aware Vertex Remapping

INPUT: Graph $\mathcal{G}\{V, E\}$, degree threshold θ_{low} and θ_{high} , number of vertices in the scratchpad S ($S < |V|$).
begin
  Initialize: $N = |V|$; Vertex pointers p_{low} = v_0, p_{high} = v_S; Vertex map V_{map};
  while $p_{low} < v_S$ **do**
    if $p_{low}.degree() > \theta_{low}$ **then**
      p_{low}++;
      continue;
    end
    while $p_{high} < V_N$ **do**
      if $p_{high}.degree() < \theta_{high}$ **then**
        p_{high}++;
        continue;
      end
      $\&p_{temp} = \&p_{high}$;
      $\&p_{high} = \&p_{low}$;
      $\&p_{low} = \&p_{high}$; // Switch p_{low}, p_{high}
      $V_{map}.insert(p_{low}, p_{high})$;
      break;
    end
  end
  # pragma omp parallel for
  for e *in* E **do**
    if $e.source$ *in* V_{map} **then**
      $e.source = V_{map}[e.source]$
    end
    if $e.dest$ *in* V_{map} **then**
      $e.dest = V_{map}[e.dest]$
    end
  end
end

C. Elastic-Granularity DRAM

In this section, we introduce our design for the elastic-granularity DRAM (EG-DRAM). Additionally, we design the instruction format for accessing multiple fine-grained data in a packed request.

1) Challenge and opportunity in conventional DRAM: Given that on average 14.0% of one 64B cache line (around 9B) is actually accessed in graph applications as characterized in Section II-B, a memory access granularity of 8B seems to be more reasonable and efficient. However, the current DDR4 DRAM is burst-oriented and designed to fit with the 64B cache line size [11]. This is then the smallest granularity we can expect per transaction. Even though the DDR4 offers configurable burst length (for example, setting burst length from 8 to 2 to access 16B data each time), but this only affects memory controller settings on the processor side but not the DRAM. As a result, the DRAM still sends 16B data and another 48B invalid data back to the controller, meaning that we cannot actually have bandwidth gain from changing the burst.

The opportunity inspiring our EG-DRAM comes from the DRAM's internal hierarchy. Generally, a DRAM DIMM is

organized as $rank$, $chip$, $bank$, row and $column$. When accessing DRAM, we have to specify the addresses and choose which $rank$, $bank$, row, and $column$ the data is located in, but all the $chips$ within the same rank share these addresses. Each DRAM $chip$ (or DRAM device) outputs 4- or 8-bit data (what is called x4 chip or x8 chip) every clock edge. With 8 dual-edge clocks of bursting, 4B or 8B fine-grained data could be expected from a single chip.

2) Elastic-Granularity (EG) DRAM design.: The key idea of EG-DRAM is cutting down the number of DRAM chips for each individual memory access. As shown in Fig. 4(c), the EG-DRAM mainly consists of the instruction FIFO, instruction decoder, command (CMD)/address (Addr) generator, and input/response buffer. Multiplexers are also used to bypass the EG-DIMM logic and serve regular memory requests.

Instruction FIFO & instruction decoder: The instruction FIFO receives packed memory requests from both the C/A bus and DQ bus (which is first cached at the input buffer). The multiplexer determines if the coming request is an EG-instruction, such that it pushes the request to the instruction FIFO or directly forwards it to the DRAM devices. The instruction decoder reads from the FIFO and dispatches the request to different DRAM devices separately, such that finer-grained data could be achieved.

Command & address generators: Each DRAM device is equipped with individual command/address generators. They are finite-state machines that follow the standard DDR4 protocol and work as signal generators to activate the DRAM device. To minimize their area overhead, we simplify and optimize the states within them, which will be discussed in Section III-D.

Input & response buffer: Since the regular DDR4 channel is synchronized but our EG-instruction is processed asynchronously, the two buffers are used to cache the data from/to the DQ bus. The input buffer stores the instruction from the DQ bus, while the response buffer cache the data for a packed request.

3) EG-instruction format.: The goal of designing EG-instruction is the compatibility with DDR4 protocol, such that the memory controller can communicate with EG-DRAM through standard DDR4 memory channels. Inspired by FIR-DRAM [33], we issue ENMC instructions from the memory controller with PRECHARGE command combining special addresses and data. For example, according to the DDR4 JEDEC specification [11], for a 4Gb DIMM with 8 $\times$8 DRAM chips, the row address space consumes 14 bits, i.e., A0-A13 in the C/A bus, and the data bus is 64-bit. Normal PRECHARGE command sets all the row address bits to be low, since no row information is needed. Therefore, an ENMC instruction could be accommodated by sending a PRECHARGE command but turning on the row address signals.

Given this insight, we design the ENMC instruction formatted in 13-bit command (line A0-A12) and 64B data (DQ bus). As shown in Fig. 4 (d), in the command line, we use 1 bit to denote request type (read or write), while the data size, data count, and request ID are specified with 4 bits respectively. On the other hand, the 64B (under bursting) in the DQ bus are used for the addresses for these requests. It is mandatory that the multiple requests packed in one instruction are the same type and accessing the same data size, such that the returned data could be formatted in a strided pattern. Note that for a write instruction, the DQ bus needs 2 bursts to send both the addresses and data.

D. Memory Controller Extension and Optimization

In this section, we introduce the extension to the existing memory controller for EG-DRAM. We then optimize the DDR4 commands for better C/A bus efficiency.

1) Memory controller extension.: To encode and issue the EG-instruction, we need to extend the memory controller. The main task of the extension unit is to scan the memory request queue through a fixed window and coalesce the proper requests together as a packed instruction. It leverages the address mapping unit to identify those requests within the same DRAM rank, such that they could be packed simultaneously. Note that, the data type of vertex properties in a graph are fixed for one workload, i.e., they are either *float*, *int*, or *long*. Therefore, finding data of the same size is not difficult.

2) DDR4 protocol optimization.: Two design considerations motivate the optimization for the existing protocol: First, since we currently need the C/A bus to send both regular DDR4 commands (PRECHARGE, ACTIVATE, READ, etc) and EG-instruction, the memory controller may encounter C/A bandwidth bottleneck during the execution. Second, the limited on-DIMM area requires us to minimize the size of command/address generator, since they are copied individually for each DRAM device and could be a large overhead. Therefore, the goal of the optimization is to reduce the number of commands in the DDR4 protocol, which could save both the bandwidth and area.

Opportunity: We find that there are generally 3 steps to access the DRAM: $precharge$ (write back the current opened row), $row\ activate$ (open another row), and $column\ access$ (read/write the data), all of which have corresponding commands in the DDR4 protocol. Since the command bus is not double-rate, meaning 3 cycles are needed compared with the 4 data cycles. However, as the graph applications are filled with random memory accesses, we may hardly gain much row buffer hit and the current opened row appears to write back again and again. This means letting the data wait at the row buffer does not bring much benefit and we could save one cycle by precharging the row buffer automatically after the column read/write operations.

Therefore, we re-design the finite-state machine in both the memory controller and command/address generator, by replacing all READ/WRITE commands with READ_AP/WRITE_AP. Thus, the C/A bus or signal generators only need to send 2 commands for one access. This eliminates the $precharge$ command, and only 2 cycles in the command bus are occupied in accessing 4-cycle data, leaving half of the command bandwidth being idle.

IV. Evaluation

In this section, we present the evaluation results of our G-MEM. We first clarify our experiment methodology. Second, we present the comparison of G-MEM to prior work, along with the sensitivity studies. Finally, we discuss the area and power overhead in the EG-DRAM architecture.

A. Methodology

Evaluation Tools: We evaluate the G-MEM based on trace-based and cycle-accurate simulations through the Sniper multi-core simulator [17], with modifications on the memory hierarchy and DRAM subsystem. With the interface provided in Sniper, we model the power consumption and area breakdown with McPAT [34].

Configurations: We configure a 32-thread system with four memory channels, while each thread has 32KB L1-D and 32KB L1-I caches. The scratchpad size is set as 32MB, considering we do not enable L2 cache. Each memory channel consists of 4 DDR4-2666 ranks, and each rank has 8×8 DRAM chips that add to a total capacity of 8Gb. In addition, we synthesize our EG-logic with TSMC 28nm technology, running on the frequency of 400MHz. The input buffer and response buffer have the size of 512B respectively, aligned with the size of maximal requests.

Baselines: Since we targeting general-purpose graph processing, we take the performance of a vanilla 32-thread CPU as the main evaluation baseline. For the overall performance, an NVIDIA Tesla V100 is also used for comparison, which has 5120 threads and 960GB/s memory bandwidth. We emphasize that this may not be a fair comparison since our G-MEM is designed specifically for the memory hierarchy in CPU, and it only has about 32 threads and 80GB/s bandwidth.

On the other hand, we compare the coherence-free scratchpad (CF-scratchpad) with the GRASP, a recent domain-specific cache design for graph analytics [10]. Then, we choose the Gather-Scatter DRAM (GS-DRAM) [25] as the baseline for EG-DRAM. We clarify that GS-DRAM is not designed specifically for graph processing, and the strided gather/scatter may not be preferable for random accesses. However, GS-DRAM is still closely related to EG-DRAM, as we both customize DRAM to collect multiple requests in one transaction.

Datasets	Type	Nodes	Edges
wiki-topcast (Wiki)	Hyperlink	1,791,489	28,511,807
soc-LiveJournal1 (LJ)	Social Network	4,847,571	68,993,773
Orkut	Communities	3,072,441	117,185,083
Friendster	Communities	65,608,366	1,806,067,135

TABLE II
FOUR DATASETS THAT ARE EVALUATED AMONG FOUR ALGORITHMS.

Graph Workloads: We use 2 algorithms: PageRank (PR), connected component (CC) from GAP benchmarks [8] (for static graphs), and 2 algorithms: bread-first searching (BFS) and max computation (MC) from SAGA benchmarks [35] (for streaming graphs). These two benchmarks have no fixed programming model and well-optimized codes for multithreading, and thus are considered as state-of-the-art benchmarks. Four datasets are used from the SNAP dataset collection [36], as shown in Table II. Moreover, GAP processes datasets into the CSR format, while SAGA stores them as adjacency lists. For graph algorithms running on GPU, we use the GunRock framework [37].

B. Performance

In this section, we demonstrate the performance results of G-MEM, including the performance speedups compared with the CPU baseline and prior work.

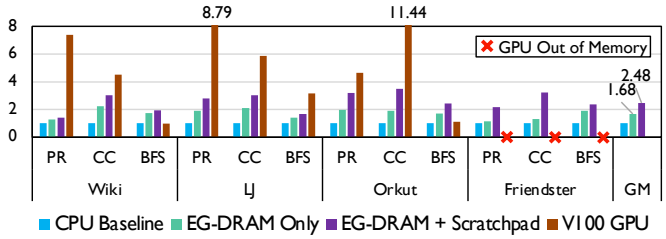

Fig. 7. The overall performance of G-MEM compared with the CPU baseline. We configure the G-MEM without/with the coherence-free scratchpad.

1) Overall Performance.: Fig. 7 shows G-MEM's overall performance results compared with the CPU and GPU baseline, where two configurations (without/with the scratchpad) are used to present the performance breakdown of on-chip scratchpad and off-chip DRAM. First, G-MEM without/with the scratchpad achieves 1.68/2.48× speedup on average over the CPU baseline respectively. This indicates both the coherence-free scratchpad and elastic-granularity DRAM offers considerable speedups to the system. Second, benefiting from large thread-count and higher bandwidth, GPU provides an excellent performance than CPU with 4.08× speedup. However, the main drawback of GPU is its capacity. As shown, GPU fails to run large-scale graphs such as Friendster and thus is not the perfect platform for graph processing.

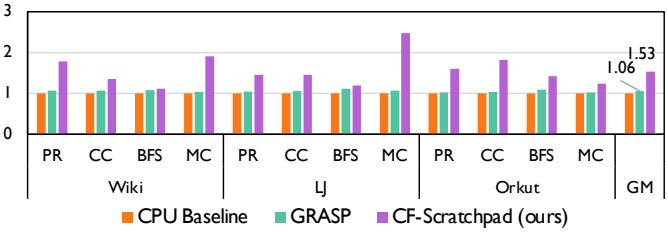

Fig. 8. The end-to-end performance of CF-scratchpad compared against the GRASP, where both the graph reordering in GRASP and our vertex remapping are included in the performance.

2) Compared with prior work.: As shown in Fig. 8, we compare the end-to-end performance of CF-scratchpad with the GRASP, the state-of-the-art cache design for graph analytics [10]. We achieve 1.44× average speedup against the GRASP. Specifically, we find GRASP only achieves 1.06× speedup over the CPU baseline (indeed, GRASP reported 1.04× speedup originally in the paper). This is caused by the large preprocessing overhead included in the end-to-end performance. As GRASP relies on heavy graph reordering algorithms to fit the high-degree vertices into caches, the

exhausting reordering time migrates the performance gaining from the hardware. Different from GRASP, our degree-aware vertex remapping is threshold-based and lightweight, such that incurring low overhead to the performance.

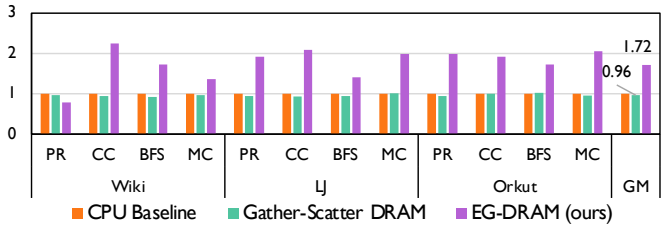

Fig. 9. The performance of EG-DRAM compared against the GS-DRAM. The near-baseline performance of GS-DRAM is because the strided gather/scatter are not preferred in graph processing.

Fig. 9 shows the comparison between GS-DRAM and EG-DRAM, and we achieved 1.79× speedup over the GS-DRAM. As we mentioned before, GS-DRAM may not be suitable for graph workloads. It only explores the strided gather and scatter, and hardly coalescing requests among the randomly distributed data. Therefore, the performance of GS-DRAM is expected to be similar to regular DRAM in the random-access scenario. Our EG-DRAM overcomes this issue by exploiting more complicated on-DIMM logic to serve an individual request from each DRAM device, so the bandwidth is easily saturated.

C. Sensitivity Study

In this section, we analyze the sensitivity of G-MEM with different hardware configurations, including the scratchpad size and the DRAM device width.

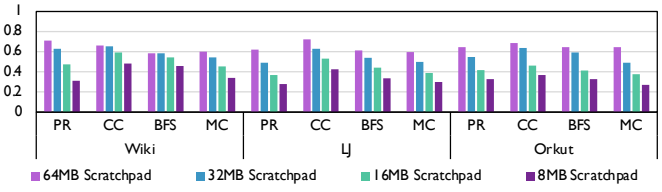

Fig. 10. The scratchpad hit ratio under different scratchpad size settings, varying from 8MB to 64MB.

1) Sensitivity to different scratchpad size. : We use the scratchpad hit ratio to demonstrate the efficiency of scratchpad, as higher hit ratio leads to reduced off-chip memory traffic. As shown in Fig. 10, we vary the size of scratchpad from 8MB to 64MB and compare their hit ratio. We find that larger scratchpad always outperforms the smaller ones. With increasing the capacity by 2×, we observe an higher hit ratio increased by 10.3%, 11.5% and 7.4% for 16MB, 32MB and 64MB respectively. This indicates that we cannot expect a linear increase of the hit ratio from enlarging the capacity. Considering the trade-off between area and efficiency, we take the size of 32MB as our system configuration.

2) Sensitivity to different DRAM configurations. : We configure the DRAM device on the EG-DIMM to ×8 chip and ×16 chip, which results in 8B and 16B accessing granularity

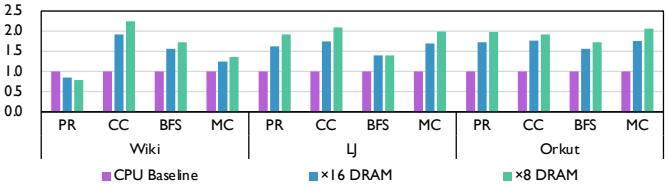

Fig. 11. The performance results under two DRAM device configurations.

for each transaction. As shown in Fig. 11, we find that except for the PR on Wiki, the ×8 DRAM demonstrates a better performance than the ×16 DRAM by 11%. For the PR on Wiki, the ×8 DRAM appears less efficient than the ×16 DRAM, which aligns with our explanation in Section IV-B1: this workload explores more sequential accesses, resulting in a better performance for a coarser-grained DRAM.

D. Power and Area Breakdown

	Area (mm^2)	Power (mW)		Area (mm^2)	Power (mW)
Inst. FIFO	0.017	14.2	**Inst. Decoder**	0.002	2.3
Input Buffer	0.034	28.4	**Response Buffer**	0.034	28.4
CMD/Addr Gen.	0.159	149.6	**MC Extension**	0.020	17.7
EG-DRAM – Area 0.266mm^2; Total Power 240.6mW					
CF-Scratchpad – Area 28.94mm^2; Total Power 1.32W					

TABLE III
AREA AND POWER ESTIMATION OF EG-DRAM'S OVERHEAD AND CF-SCRATCHPAD MEMORY.

Table III shows the breakdown area and power estimation of EG-DRAM overhead. The total area of EG logic is $0.247mm^2$, and the total power is $224.1mW$, which are quite insignificant considering the area and power of DRAM DIMM are in the order of hundred mm^2 and W [38]. Specifically, the command and address generators take 64.3% of the total area and 66.8% of the total power. The buffers compose 27.5% of the total area and 25.3% of the total power. Moreover, the control logic, including instruction FIFO and instruction decoder take 7.7% of the total area and 7.4% of the total power. Finally, the area and power of the CF-scratchpad are about $28.94mm^2$ and $1.32W$, respectively. This is acceptable considering CPU usually has an area of hundreds of mm^2.

V. CONCLUSION

In this paper, we present G-MEM, a customized memory hierarchy design for graph processing applications. First, we propose a coherence-free scratchpad as the on-chip memory, which leverages the power-law characteristic of graphs and only stores those hot data that are frequent-accessed. We equip the scratchpad memory with a graph-aware mapping strategy to better manage it for various applications. On the other hand, we design an elastic-granularity memory subsystem based on near-data processing (NDP) architecture, which processes and coalesces multiple fine-grained memory accesses together to maximize the bandwidth efficiency. Putting them together, the G-MEM demonstrate a 2.48× overall speedup over a vanilla CPU, with 1.44× and 1.79× speedup against the state-of-the-art cache architecture and memory subsystem, respectively.

REFERENCES

[1] E. Agirre, A. Barrena, and A. Soroa, "Studying the wikipedia hyperlink graph for relatedness and disambiguation," *arXiv preprint arXiv:1503.01655*, 2015.

[2] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow, "The anatomy of the facebook social graph," *arXiv preprint arXiv:1111.4503*, 2011.

[3] X. Shao, J. Xie, T. Hong, and A. Jost, "System and method for identity-based fraud detection through graph anomaly detection," Jul. 21 2009, uS Patent 7,562,814.

[4] J. Pujara, H. Miao, L. Getoor, and W. Cohen, "Knowledge graph identification," in *International Semantic Web Conference*. Springer, 2013, pp. 542–557.

[5] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 1–14.

[6] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, "Analysis and optimization of the memory hierarchy for graph processing workloads," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 373–386.

[7] S. Eyerman, W. Heirman, K. D. Bois, J. B. Fryman, and I. Hur, "Many-core graph workload analysis," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018, pp. 22:1–22:11.

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

[9] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[10] P. Faldu, J. Diamond, and B. Grot, "Domain-specialized cache management for graph analytics," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 234–248.

[11] JEDEC, "Jesd79-4 - jedec," https://www.jedec.org/category/technology-focus-area/main-memory-ddr3-ddr4-sdram, 2017.

[12] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. 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)*, 2012, pp. 17–30.

[13] L. A. Adamic and B. A. Huberman, "Power-law distribution of the world wide web," *science*, vol. 287, no. 5461, pp. 2115–2115, 2000.

[14] P. Sakarda, T. Brandt, and H. H. Wu, "Memory manager for heterogeneous memory control," Aug. 4 2009, uS Patent 7,571,295.

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

[16] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, L. 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*, 2019, pp. 615–628.

[17] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.

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

[19] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[20] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, p. 25, 2015.

[21] Z. Meng, A. Koniges, Y. H. He, S. Williams, T. Kurth, B. Cook, J. Deslippe, and A. L. Bertozzi, "Openmp parallelization and optimization of graph-based machine learning algorithms," in *International Workshop on OpenMP*. Springer, 2016, pp. 17–31.

[22] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "Imp: Indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 178–190.

[23] D. H. Yoon, M. K. Jeong, M. Sullivan, and M. Erez, "The dynamic granularity memory system," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2012, pp. 548–560.

[24] D. H. Yoon, M. K. Jeong, and M. Erez, "Adaptive granularity memory systems: A tradeoff between storage efficiency and throughput," in *Proceedings of the 38th annual international symposium on Computer architecture*, 2011, pp. 295–306.

[25] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Gather-scatter dram: In-dram address translation to improve the spatial locality of non-unit strided accesses," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 267–280.

[26] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee *et al.*, "Recnmp: Accelerating personalized recommendation with near-memory processing," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 790–803.

[27] Y. Kwon, Y. Lee, and M. Rhu, "Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 740–753.

[28] ——, "Tensor casting: Co-designing algorithm-architecture for personalized recommendation training," *arXiv preprint arXiv:2010.13100*, 2020.

[29] J. E. Zolnowsky, C. L. Whittington, and W. M. Keshlear, "Memory management unit," Sep. 25 1984, uS Patent 4,473,878.

[30] B. Egger, J. Lee, and H. Shin, "Scratchpad memory management for portable systems with a memory management unit," in *Proceedings of the 6th ACM & IEEE International conference on Embedded software*. ACM, 2006, pp. 321–330.

[31] J. S. Kimmel, R. A. Alfieri, A. Miles, W. K. McGrath, M. J. McLeod, M. A. O'connell, and G. A. Simpson, "Operating system for a non-uniform memory access multiprocessor system," Aug. 15 2000, uS Patent 6,105,053.

[32] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. 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)*, 2012, pp. 17–30.

[33] Y.-C. Kwon, S. H. Lee, J. Lee, S.-H. Kwon, J. M. Ryu, J.-P. Son, O. Seongil, H.-S. Yu, H. Lee, S. Y. Kim *et al.*, "25.4 a 20nm 6gb function-in-memory dram, based on hbm2 with a 1.2 tflops programmable computing unit using bank-level parallelism, for machine learning applications," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64. IEEE, 2021, pp. 350–352.

[34] 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 *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2009, pp. 469–480.

[35] A. Basak, J. Lin, R. Lorica, X. Xie, Z. Chishti, A. Alameldeen, and Y. Xie, "Saga-bench: Software and hardware characterization of streaming graph analytics workloads," IEEE, pp. 12–23, 2020.

[36] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[37] 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*, vol. 51, no. 8. ACM, 2016, p. 11.

[38] Micron, "3-dimensional stack (3ds) ddr4 sdram," https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/16gb_32gb_x4_x8_3ds_ddr4_sdram.pdf, 2019.