SuperCut: Communication-Aware Partitioning for Near-Memory Graph Processing

Chenfeng Zhao, Roger D. Chamberlain, Xuan Zhang
McKelvey School of Engineering, Washington Univ. in St. Louis
St. Louis, Missouri, USA
{chenfeng.zhao,roger,xuan.zhang}@wustl.edu

ABSTRACT

The parallel execution of many graph algorithms is frequently dominated by data communication overheads between compute nodes. This bottleneck becomes even more pronounced in Near-Memory Processing (NMP) architectures with multiple memory cubes as local memory accesses are less expensive. Existing near-memory architectures typically use graph partitioning methods with a fixed vertex assignment, which limits their potential to improve performance and reduce energy consumption. Here, we argue that an NMP-based graph processing system should also consider the distribution of vertices onto memory cubes. We propose SuperCut, a framework for near-memory architectures to effectively reduce communication overheads while maintaining computational balance. We evaluate SuperCut via architectural simulation with 6 real-world datasets and 4 representative applications. The results show that it provides up to 1.8× total energy reduction and 2.6× speedup relative to current state-of-the-art approaches.

CCS CONCEPTS

• **Hardware → Memory and dense storage**; **Emerging architectures**; • **Computer systems organization → Multicore architectures**.

KEYWORDS

near-data processing, 3D-stacked memory, graph processing

ACM Reference Format:
Chenfeng Zhao, Roger D. Chamberlain, Xuan Zhang. 2023. SuperCut: Communication-Aware Partitioning for Near-Memory Graph Processing. In *20th ACM International Conference on Computing Frontiers (CF '23), May 9–11, 2023, Bologna, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3587135.3592209

1 INTRODUCTION

Due to their ability to capture the complex dependencies and relationships among individual data elements, graphs constitute an important data structure that have been widely used to represent social networks, citation networks, road networks, genome sequences, etc. The recent proliferation of graph processing applications, including machine learning [38], recommendation systems [25], and social network analysis [31], has heightened the need for efficiently processing graphs, both in terms of performance and energy consumption. Hence, a number of approaches have been proposed to efficiently process large-scale graphs [11, 22, 23, 27, 42].

The inherent properties of graph analytic applications pose challenges for conventional memory and communications systems, which in turn become performance bottlenecks. First, the operation of traversing neighbourhood vertices shows poor locality due to random memory accesses. Second, many graph algorithms have high memory bandwidth requirements because the node-level computation is relatively simple. Third, when executing in parallel, frequent data movement across the system puts pressure on the communications network.

Since the demand for higher memory bandwidth is an important part of accelerating large-scale graph processing, Near-Memory Processing (NMP) has been proposed to accelerate these tasks. The emergence of 3-D stacked memory technology, in which multiple DRAM chips are stacked on top of a single logic chip, has opened the door for the deployment of computation units near the physical DRAM. Instead of a single memory stack (often referred to as a *cube*), recent NMP architectures for large-graph parallel processing utilize multiple memory cubes [1–3, 40], which are able to provide both higher memory capacity and memory-capacity-proportional bandwidth. Figure 1 illustrates a general abstraction of near-memory graph processing systems. Real-world information is abstracted into graph data structures. Graph processing applications are deployed to computing units inside memory chips and executed in parallel. Interaction between memory chips is communicated via an interconnection network.

Executing graph processing applications on NMP architectures is distinct from traditional systems for a pair of reasons. First, the number of compute nodes can scale up substantially in a shared-memory paradigm since cache coherence is often not needed. E.g., Zhao et al. [41] show substantial performance gains in an NMP system primarily due to the greater parallelism achieved. Second, delivering information between compute nodes utilizes mechanisms that are substantially less heavy-weight than message-passing protocols.

Tesseract [1] proposes an NMP architecture for parallel graph processing with 16 cubes. While providing substantial performance gains over conventional DRAM-based architectures, its performance is ultimately limited by cross-cube communications. To alleviate such communication overhead, prior works [1, 40] tried METIS [16] to execute graph partitioning. However, the results were not promising. It was reported that there are several factors limiting the performance of METIS on such cases: (1) it leads to substantial variance between maximum and average communication; (2) it exacerbates intra-cube computational balance.

CF '23, May 9–11, 2023, Bologna, Italy

ACM ISBN 979-8-4007-0140-5/23/05.
https://doi.org/10.1145/3587135.3592209

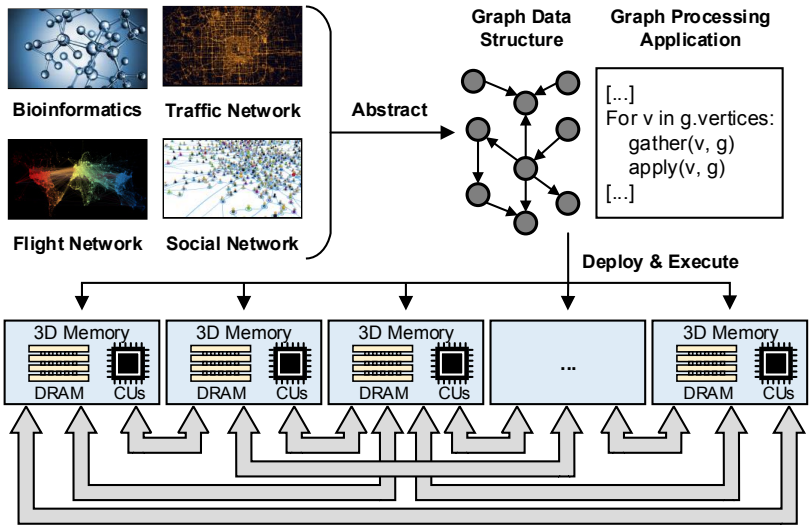

Figure 1: Near-memory graph processing system.

Subsequent works have proposed to diminish communication bottlenecks by alternative preprocessing of the graph [40] or by run-time adaptations [2, 3]. GraphP [40] proposes *source-cut* partitioning, in which replicas of the source vertex of each cross-cube edge are deployed in destination cubes, so that multiple cross-cube edges sent from a common source vertex to the same destination cube can be reduced to one. Therefore, lower cross-cube communication volume is required relative to Tesseract.

Despite the promising results from source-cut partitioning, there is still room for improvement. We observe that after performing source-cut partitioning, cross-cube communication still takes a significant portion of execution time (12%-78%) and energy consumption (14%-73%). This invites the open research question: *how effectively can partitioning algorithms reduce communications overheads while maintaining computational balance in an NMP system*?

To further explore this question, we introduce a co-design framework for near-memory graph processing, called *SuperCut*, and evaluate its effectiveness. In this paper, we make several contributions:

- We propose a set of graph partitioning algorithms, containing: (1) a *mixed-cut* partitioning method which reduces communication volume by recognizing more cross-cube edge patterns, and (2) a *vertex-swapping-based greedy* algorithm to further reduce communication volume by iteratively changing the vertex distribution.
- We propose a *three-phase programming model* that is expressive for general vertex programs and explicitly handles computation and communication via user-defined functions along with a custom graph representation to bridge the software and hardware design while diminishing the irregularity of vertex traversal and communication.
- We generate specialized accelerators via high-level synthesis (HLS) and map them to FPGA resources on the logic layer of 3D-stacked memory cubes.

To evaluate our framework, we build a multi-cube, near-memory processing simulation platform with reconfigurable logic kernels as computing units by extending `gem5-SALAM` [30]. Our evaluation results show that SuperCut provides up to 1.8× total energy reduction and 2.6× speedup with 45% lower extra memory footprint relative to GraphP.

2 BACKGROUND

2.1 3-D Stacked Memory Technology

The basic principle of near-memory processing is to place computation units inside the memory device(s) to implement computation closer to the data. The emergence of 3-D stacked memory technology has provided a practical opportunity for realizing this vision [34]. These 3-D memories consist of multiple DRAM chips stacked on top of a single logic chip. The chips are connected by multiple vertical through-silicon vias (TSVs) so that the DRAM layers can be accessed with higher bandwidth and lower power than conventional off-chip memory channels. The underutilized logic layer has both area and power available for integrating compute functions [28, 41]. Commercial offerings include the early Hybrid Memory Cube (HMC) [13] as well as High Bandwidth Memory (HBM) [14]. While there are variations across the specific implementations, the core technology is common. To facilitate fair comparison with earlier work, we use technology parameters from HMC in our simulation models.

Figure 2 (on the left) shows the structure of a single memory cube. Each cube is divided into 32 vertical partitions called vaults and has 4 SerDes high-speed links to implement off-chip accesses. With each cube having a capacity of 8 GB, each vault has a capacity of 256 MB. The logic layer at the bottom of the stack consists of both interconnections and vault controller logic. Each vault can provide 10 GB/s bandwidth. Therefore, the internal bandwidth of each cube is 320 GB/s. For off-chip access implemented by the SerDes links, each link can provide a bandwidth of 120 GB/s. Each cube then has an external bandwidth of 480 GB/s. In addition, each cube has unused area on its logic layer. Previous works [1, 5] report that the spare area is about 60 mm^2, comprising 26.5% of the the total die

area (226 mm^2 per cube [32]). While much previous work has used small, in-order cores to implement the computing units within the logic layer [1, 28, 41], in this work we assume that the logic layer has a reconfigurable logic fabric [10, 33].

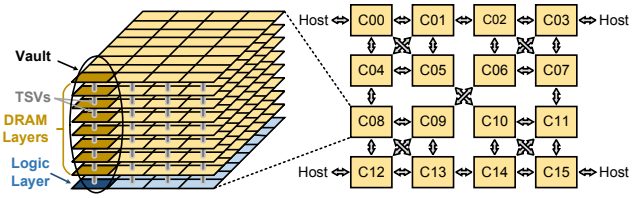

Figure 2: A single memory cube (left), and topology of the Dragonfly cube-to-cube interconnection network (right).

2.2 Memory Interconnection Network

Due to the flexibility of the SerDes links, various interconnect topologies can be considered. One prevalent topology is Dragonfly [17], which has high connectivity and a low network diameter. Figure 2 (on the right) shows a Dragonfly interconnection network with 16 cubes. Unused links are used to provide connectivity to host cores. We use the Dragonfly network topology in this work.

For the 16 cube system, the aggregated internal memory bandwidth is 5 TB/s while the bisection bandwidth of the interconnection network is only 480 GB/s, implying that inter-cube communications can easily become a performance limiter [8, 40]. In addition to performance, prior work [2] reports that cross-cube communication is also the primary source of energy consumption in graph processing applications, taking up 62% of the total.

2.3 Previous Near-Memory Architectures

There are several NMP architectures based on multiple memory cubes proposed to improve graph processing applications' performance and energy consumption. Tesseract [1] leverages the large internal bandwidth provided by 16 memory cubes connected in a Dragonfly topology. A single-issue in-order CPU and a prefetcher, serving as computation units, are deployed on the logic layer of each vault. Tesseract adopts Pregel [23] and provides a vertex-centric programming model. The authors report 9× speedups relative to a traditional multicore system using out-of-order cores. While this performance gain is substantial, Dai et al. [8] and Zhang et al. [40] indicate that Tesseract's overall memory bandwidth utilization is less than 40%, implying there are additional performance gains to be had. The reason for this bandwidth utilization limit is cross-cube memory accesses, which Tesseract did not try to optimize.

To reduce cross-cube communications, Zhang et al. [40] proposed a graph partitioning method, called *source-cut*. If two or more cross-edges share the same source vertex but have different destination vertices in a common cube, a replica of the source vertex is placed in the destination cube. Therefore, the data of the source vertex need only be transferred once. To realize the source-cut partitioning method, Zhang et al. proposed a *Two-Phase Vertex Programming* model. The `GenUpdate` phase generates the update for each replica and the `ApplyUpdate` phase updates each replica. Cross-cube communication will only happen before `ApplyUpdate`. To hide the remote latency of cross-cube communication, `GenUpdate` and communication are overlapped asynchronously. A barrier after each phase ensures that hardware cache coherence is not required.

Despite of the promising results of Zhang et al. [40], there are two inherent properties of source-cut partitioning that limits its potential: (1) Since only one cross-cube edge pattern is considered, it only considers a fraction of the cross-cube edges that might potentially be eliminated. (2) It adopts a fixed initial vertex distribution, which limits its options for reducing cross-cube communication overheads. To address these limitations, we introduce a novel software/hardware co-design framework for multi-cube NMP systems, called *SuperCut*, to effectively reduce cross-cube communication overheads while maintaining workload balance.

3 SUPERCUT FRAMEWORK

Here, we describe SuperCut, our co-design framework for near-memory graph processing. First, the graph dataset is pre-processed by graph partitioning algorithms (Sec. 3.1 to 3.3). Then the three-phase programming model (Sec. 3.4) is built with user-defined functions to express the graph processing applications. Next, NMP accelerators are generated via HLS (Sec. 3.5). The partitioned graph is stored using the custom graph representation (Sec. 3.6). Both the graph and the accelerator design is fed into the NMP simulation.

We repeatedly refer to Figure 3 to illustrate several points related to the graph partitioning. Figure 3(a) shows an example of a small synthetic graph. This graph has 8 cross-cube edges, each of which initially represents one cross-cube communication.

3.1 Mixed-Cut Partitioning

Our initial partitioning algorithm is called *mixed-cut*, which is a combination of source-cut partitioning and destination-cut partitioning. The source-cut pattern (described by Zhang et al. [40]) is illustrated by the blue dashed rectangle in Figure 3(a). After the transformation, the revised graph is shown in blue in Figure 3(b).

For cross-cube updates with a common destination vertex, MessageFusion [2] dynamically merges these updates at the source cube before transferring them to the destination cube. Inspired by MessageFusion, we propose a static graph partitioning method, called *destination-cut*, to reduce cross-cube edges exhibiting this same pattern (multiple cross-cube edges which have the same destination vertex and distinct source vertices, all from the same cube). Figure 3(a) illustrates an example of the cross-cube edge pattern of destination-cut partitioning (marked in red). In this example, there are three cross-cube edges: $v_5 \rightarrow v_4$, $v_6 \rightarrow v_4$, and $v_7 \rightarrow v_4$ which have the same destination vertex, v_4.

In mixed-cut partitioning, we first implement source-cut partitioning as the initial partitioning method and then implement destination-cut partitioning as the secondary method. Figure 3(b) illustrates the results of mixed-cut partitioning on the example graph. Here, 6 out of 8 original cross-cube edges (75%) have been reduced, which is higher than source-cut partitioning alone (37.5%).

3.2 Vertex-Swapping Greedy Algorithm

The partitioning algorithms discussed so far do not consider moving vertices between cubes. The next element of SuperCut partitioning

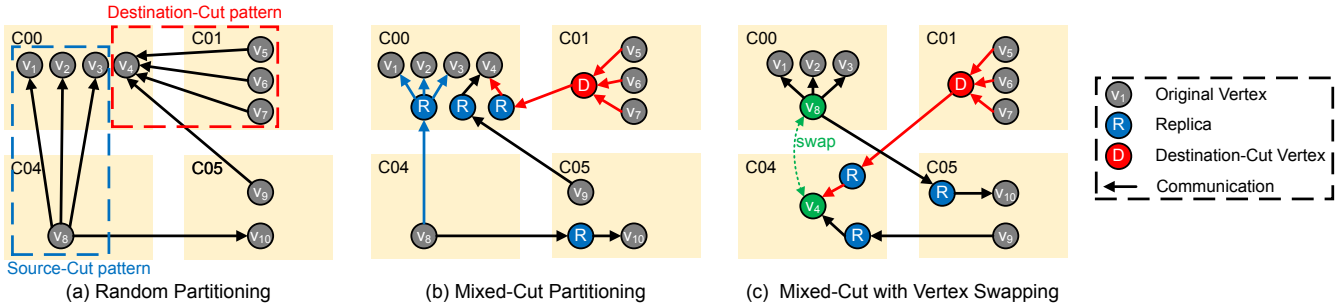

Figure 3: (a) A synthetic graph as an example to illustrate source-cut and destination-cut patterns. (b) The mixed-cut partitioned graph. (c) The mixed-cut example graph after swapping v_4 and v_8.

is a stochastic, greedy optimization algorithm that explicitly moves vertices across cube boundaries.

Inspired by the iterative placement algorithms of IC physical design [15], which continuously modify the placement of circuits by exchanging randomly-selected cells, we propose a greedy algorithm to diminish the cost of communication while maintaining workload balance across the cubes. In order to implement a greedy algorithm, a cost function is needed. The goal of the cost function is to capture both the execution time and energy consumption of cross-cube communication, described as follows:

$$cost(G) = \alpha_1 \times \max(cost_{comm}) + \alpha_2 \times \text{mean}(cost_{comm})$$

where $cost_{comm}$ is the cube-level communication cost in graph G calculated by multiplying the number of cross-cube edges and the number of SerDes links between each cube pair, and $\max(cost_{comm})$ is the maximum communication cost among all the cube pairs while $\text{mean}(cost_{comm})$ is the average communication cost among all the cube pairs. The first term represents the worst-case runtime of cross-cube data transfer and the latter term represents the aggregated energy consumption of cross-cube communications. Parameters α_1 and α_2 are adjusted to balance these two different goals into a single metric, which are set on the basis of the significance of performance and energy as required in different scenarios.

In order to avoid introducing workload imbalance, we implement a vertex-swapping strategy as the perturbation function in the greedy algorithm, shown in Algorithm 1. In this algorithm, G represents the original graph before the swapping operation, H represents the mixed-cut graph of the original graph before the swapping operation, and graph names with suffix $'$ represent graphs after the swapping operation (e.g., G' represents the original graph after the swapping operation) Initially, all the vertices are mapped into cubes using a hash function to get an initial vertex distribution (line 1), e.g., with a modulo function:

cube_index = vertex_index **modulo** total_number_of_cubes

which is widely used in prior works [8, 23, 40]. Next, initialize the cost value based on the mixed-cut graph H (line 2). In each iteration (lines 3-9) , the greedy algorithm will randomly swap two vertices in different cubes (line 4) . Then mixed-cut partitioning is applied to the graph and the cost is calculated based on the mixed-cut graph (line 5). If the cost increases, then undo the swap of the selected vertices (line 6); otherwise, keep the change (lines 7-9).

Algorithm 1: Vertex-Swapping Greedy Mixed-Cut Alg.

input : G: The original graph
output: H: The partitioned graph

```
1 assign vertices to cubes by a simple hash function;
2 H, cost_old = mixed_cut(G);
3 for i = 1; i < max_iterations; i++ do
4     G' = swap random pair of vertices in distinct cubes in G;
5     H', cost_new = mixed_cut(G', H);
6     if cost_old < cost_new then undo_swap(G) ;
7     else
8         cost_old = cost_new;
9         update H with H' and G with G';
```

Figure 3(c) illustrates the example graph after being processed by one iteration of the greedy algorithm. After swapping v_4 and v_8, the number of cross-cube edges is reduced from 4 to 3, compared to mixed-cut partitioning. Since the workload of each vertex is related to the vertex degree, the total degree of cube C04 remains unchanged and the total degree of C00 is reduced from 5 to 4. Therefore, by swapping a pair of vertices in different cubes instead of moving a single vertex cross cubes, the greedy algorithm can maintain some degree of workload balance.

3.3 Partial Graph Repartitioning

Since the execution time of the iterative optimization algorithm is proportional to the number of iterations, it can be slow when the iteration number is large. Even worse, since the mixed-cut partitioning and cost calculation is performed every iteration (lines 4-5 in Algorithm 1), the larger the scale of the graph, the slower the iterative algorithm. Therefore, implementing mixed-cut partitioning to the whole graph every time after swapping the selected node pair is inefficient, especially for large-scale graphs. Fortunately, we observe that only a portion of the graph is modified after exchanging a pair of vertices. This observation provides an opportunity to increase the efficiency of the vertex-swapping strategy by only processing the influence scope of the swapping operation, instead of the whole graph, in each iteration.

Based on this observation, we propose a method, called *partial graph repartitioning*, in which the scope of the input graph considered is reduced to a smaller-scale subgraph (except for the first iteration). This method is inspired by FPGA partial reconfiguration techniques [37] which change the logic for a particular region in an FPGA without impacting operation in areas outside this region. This method is illustrated in Figure 4. It consists of 4 steps: ❶ Find the influence scope of the swapping operation (a small-scale subgraph called G'_{sub}) from the graph G' in which a target vertex pair has been swapped. ❷ Find H_{sub}, the influence scope of the swap perturbation in H which is the mixed-cut graph of G. ❸ Extract G'_{sub} from G' and implement mixed-cut partitioning to the subgraph G'_{sub}. Then we can get H'_{sub}, the mixed-cut graph of G'_{sub}. ❹ Generate H', the mixed-cut graph of G', by removing H_{sub} from H and embedding H'_{sub} into H. The cost of H' can also be calculated in the same way. In this way, we process the small-scale subgraph representing the influence scope of the swap operation instead of processing the entire graph from scratch each iteration.

To find G'_{sub} and H_{sub}, we process all of the edges $e \in G'$ that are incident with at least one of the swapped vertices, enumerating all the possible cases. After checking all the possible scenarios, one of the key observations is that the boundary of the influence scope will not be expanded to the whole graph due to the fixed pattern of mixed-cut. Instead, the distance from any vertex in the influence scope to one of the swapped vertices is no more than 3. In other words, the scale of the influence scope is smaller than the whole graph for datasets with depth greater than 3.

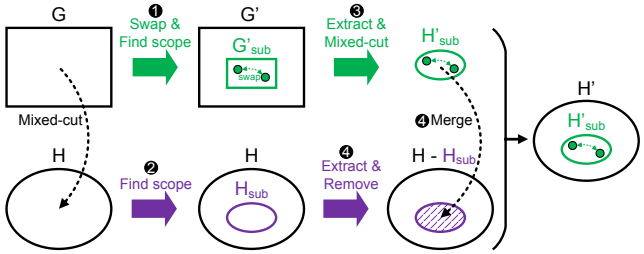

Figure 4: The partial graph repartitioning method. G_{sub} represents the influence scope of the target vertex pair before the swapping operation. H_{sub} is the mixed-cut graph of G_{sub}.

3.4 Three-Phase Programming Model

The two-phase programming model of GraphP has two limitations: (1) it supports source-cut partitioning, however it doesn't support destination-cut partitioning; (2) because the cross-cube data transfer procedure is scheduled by the operating system kernel rather than explicitly exposed to users, there is a lack of flexibility for optimizing and/or measuring cross-cube communication. In order to implement mixed-cut partitioning while maintaining compatibility with the source-cut method, we propose a new three-phase programming model shown in Algorithm 2, which has 3 steps:

Original Vertex Update (OVU) phase (lines 1-2): the original vertices are processed locally by collecting data from incoming neighbours and combining these data via computation operations packaged in the `gather_combine()` function which is customized

Algorithm 2: Pseudocode of Three-Phase Programming Model (one iteration).

```
input  : The SuperCut graph H and original graph G
output : Results of graph processing applications
1 for each original vertex v_org ∈ G do
2     gather_combine(v_org)
3 for each cross-cube edge e = (u, v) ∈ H do
4     update ← gather_combine(u); scatter(update)
5 for each original vertex v_org and replica v_r do
6     apply(v_org); apply(v_r)
```

to adapt to various graph applications. E.g., in the PageRank application, `gather_combine()` is an accumulation operation.

Remote Vertex Update (RVU) phase (lines 3-4): remote updates are generated and transferred across cubes. These updates are generated using user-defined function `gather_combine()` where: (1) destination-cut vertices are processed by combining data from incoming neighbors, and (2) each cross-cube edge starting from original vertices is traversed to get its source vertex data directly. After generating the updates, the user-defined function `scatter()` is invoked to transfer these updates across cubes.

Due to the inherent parallelism of graph applications, the OVU and RVU phase are executed in parallel so that the cross-cube communication latency is somewhat masked. Once OVU and RVU phase finish, all the updates are at their targets. It should be noted that cross-cube communication only happens during the RVU phase.

Apply phase (lines 5-6): In this phase, these updates are processed locally by the user-defined `apply()` function to generate the result for the current iteration, which also serves as the initial value of the next iteration.

Distinct from the two-phase programming model in GraphP, our programming model introduces the RVU phase for remote updates. If performing source-cut alone, the RVU phase is only responsible for data movement across cubes without the combining procedure. In this way, our programming model is not only suitable for mixed-cut partitioning but also compatible with source-cut. In addition, the cross-cube communication in our programming model is explicitly handled by the user-defined function scatter(), broadening the opportunity for communication functionality. A barrier before and after the Apply phase ensures that hardware cache coherence is not required.

3.5 Proposed Near-Memory System

To assess the benefits of SuperCut, we describe a near-memory system architecture that is similar, in many respects, to the near-memory systems of previous works. We use an HMC-like cube as our 3-D stacked memory with 8 GB DRAM capacity and 32 vaults per cube. Consistent with other multi-stack near-memory architectures, we utilize a Dragonfly topology (see Figure 2) to build a system with 16 memory cubes, in which each cube is connected to its neighbor cubes via SerDes links. We put FPGA resources on the logic layer of each cube, to which the 512 compute engines are mapped via HLS. These resources only take 0.26 mm^2 per cube (i.e., 0.12% of the total area), which is comparable to prior work.

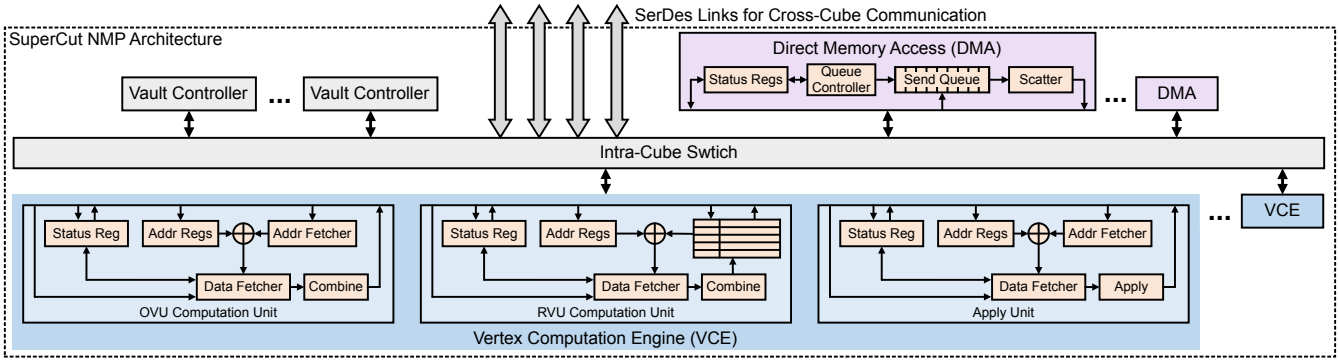

Figure 5: SuperCut near-memory processing architecture in the logic layer of each memory cube, composed of Vertex Computation Engines (VCEs) for intra-cube computation and DMAs for cross-cube communication. VCE consists of OVU Computation Unit, RVU Computation Unit, and Apply Unit.

Figure 5 illustrates the SuperCut NMP architecture on the logic layer of each memory cube. For intra-cube computation and communication (via the existing intra-cube switch), we include one Vertex Computation Engine (VCE) per vault consisting of 3 components: OVU Computation Unit, RVU Computation Unit and Apply Unit. We also design DMAs to implement cross-cube communication.

OVU Computation Unit: The OVU computation unit consists of a status register, address registers, an address fetcher, a data fetcher and a combine module. The status register includes the trigger and status bits, while the address registers are used to store the starting address of input vectors. The data addresses calculated by summing starting addresses and offsets fetched by the address fetcher are fed to the data fetcher. Then the fetched data is combined, performing the `gather_combine()` function. E.g., To implement the PageRank application, the `gather_combine()` is defined by users as an accumulation operation. Thus the combine module is synthesized to be an accumulator by the HLS compiler.

RVU Computation Unit: The RVU computation unit implements remote update generation. Distinct from the OVU computation unit, the address fetcher is replaced with a hash table of cross-cube edge information, based on which remote updates are generated by the specialized data fetcher and combine module performing the `gather_combine()` function.

Updates are transferred to specialized DMAs, along with distinct destination addresses, through the intra-cube switch. Since the OVU and RVU phases are overlapped, the OVU and RVU computation units are triggered together each iteration.

DMA: The DMAs perform the cross-cube communication. We include a send queue in each DMA to which updates with destination addresses are sent. The enqueued updates are transferred to another cube by the specialized scatter module, the realization of the user-defined `scatter()` function in the RVU phase. By default, the `scatter()` function is defined as a copy function to directly transfer data across memory cubes. It could also be defined by users with other purposes to satisfy various functionality of graph applications.

Apply Unit: The function of the apply unit is to implement the `apply()` function of the Apply phase in which updates are fetched by the data fetcher and then processed to generate results for original vertices and replicas by the apply module.

3.6 Graph Representation

The representation of the graphs in memory is a key link bridging the software and hardware system. We propose a new graph representation with a customized data structure stored in memory. Figure 6 illustrates an example of the graph representation in Cube0. In our graph representation, original vertices and replicas are stored in CSR format while cross-cube edges information is stored in the form of a hash table where the key is edge ID and values are neighbors' IDs and destination addresses. The memory footprint of the hash table ranges from 0.17 MB to 189 MB, taking up 10%-13% of the overall memory footprint.

However, only considering the storage format is likely to introduce massive irregular memory accesses, which is more expensive than sequential memory accesses, when accessing and updating vertices in memory. To mitigate such irregularity, the order of vertices in the graph representation is rearranged during preprocessing. Original vertices and replicas are deployed within separate address ranges, so that these vertices can be accessed and updated sequentially in the appropriate phases of the programming model. In addition, since replicas in the same cube are updated by separate DMAs across cubes, to reduce irregularity of remote update, we also divide replicas into several address ranges, in the order of the index of the predecessor's memory cubes. In this way, replicas originating from the same cube can be updated contiguously.

Figure 6 illustrates an example of the intra-cube communication (inside Cube0) and cross-cube communication (from Cube0 to Cube1) in data layout view. ❶ In the OVU phase, data of original vertices and replicas from different address ranges are gathered and combined based on graph topology to generate updates for original vertices in Cube0. ❷ In the RVU phase, data of adjacent original vertices listed in the hash table is processed to generate remote updates. Separate from intra-cube communication, these updates are buffered in the send queue of the DMAs and then transferred to replicas originating from Cube0 in Cube1 using the destination addresses in the table. ❸ After the first two phases finish, updates are fetched from memory to apply for target vertices.

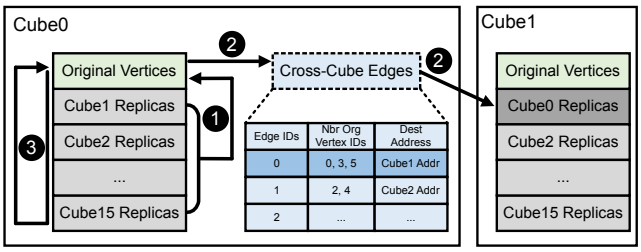

Figure 6: Diagram of graph representation in Cube0 and data communication (inside Cube0 and from Cube0 to Cube1).

4 EXPERIMENTAL METHODOLOGY

Simulation platform: We have adapted the `gem5-SALAM` [30] framework to build a bare-metal full-system NMP simulation platform. The host-side CPUs are based on the ARM ISA and the memory system consists of 16 HMC-like cubes to form a memory-centric network using a Dragonfly topology [17]. For our simulations, we use the standard distribution of `gem5` [4] that contains a stacked memory modeled after HMC and LLVM-based HLS accelerators to realize the computation units and programmable DMAs at 500 MHz.
Datasets: Table 1 shows the graph datasets used in our experiments. All these input graphs are collected from the Stanford Network Analysis Project (SNAP), a general-purpose graph library for network analysis and graph mining. These graphs have a wide range of types and fields, and are in the same scale range as prior works. In addition, Table 1 also shows maximum and average degree of graphs which have varying in-degree distributions, ranging from regular-like to powerlaw-like distributions.
Workloads: We code four popular graph processing applications in C using the proposed three-phase programming model. PageRank (PR) iteratively calculates the importance of web pages [6]. Average Teenage Follower (ATF) calculates the number of teenage followers of every user represented by vertices in the graph and the average number of teenage followers over K years old [12]. Breadth-First Search (BFS) searches a tree data structure, starting from a root vertex and traversing all the neighbours at the same depth iteratively. It is coded with a brute-force data parallel method to make it suitable for SIMD architecture [29]. Weakly Connected Components (WCC) finds a subgraph in which all the vertices are connected by some paths in which the direction of edges are ignored [35].
Evaluation methods: To evaluate the SuperCut framework, we simulate all the applications across all the graph datasets running on the NMP platform. We do the same for Tesseract and GraphP as well. Note that this implies we are comparing our proposed partitioning methods to the previously described Tesseract and GraphP on a common hardware platform (described in Section 3.5).

The preprocessing step is implemented with Python and NetworKit library [36]. Without any optimization, the execution time of the single-thread python version ranges from several minutes to multiple hours. Since the implications of preprocessing substantially vary among different implementations, we show the number of iterations the greedy algorithm takes for each graph in Table 1 (executed off-line). We hope this work would inspire follow-on studies for efficient implementations that would speedup this step.

In this work, we focus on exploring the on-line benefits of the partitioning methods. The parameters α_1 and α_2 in the cost function are set to $\alpha_1 = 0.2$ and $\alpha_2 = 0.8$ so as to emphasize energy savings somewhat over performance as the optimization goal.

Since the HLS accelerators are triggered and run in parallel, we use the maximum execution time across the HLS accelerators as the execution time for each iteration. The energy is computed by summing the dynamic energy consumption contributions from the local computation phases and the cross-cube communication phase. The total energy consumption of the HLS accelerators in each phase is modeled by `gem5-SALAM`. The energy consumption of the SerDes links, memory accesses to DRAM layers, and other modules on the logic layer are drawn from prior works [13, 28, 41].

5 EVALUATION

5.1 Energy Consumption and Performance

We first quantitatively examine the energy consumption benefits of SuperCut, comparing SuperCut with 2 baselines: Tesseract and GraphP. Figure 7 shows the normalized energy consumption breakdown into computation, local memory accesses, and cross-cube communication relative to Tesseract. Focusing first on the energy consumption reduction of cross-cube communication, we observe that all the applications benefit from cross-cube communication reduction. The energy consumption reduction of cross-cube communication for each application ranges from 3.12× to 7.23× relative to Tesseract. Compared with GraphP, the energy consumption reduction of cross-cube communication ranges from 1.32× to 3.09×. This is because SuperCut incorporates the aggregated cross-cube communication volume as one of the optimization targets. Due to the energy reduction of cross-cube communication, overall energy consumption is also reduced. The overall energy consumption reduction ranges from 1.1× to 3.09× and 1.06× to 1.84× relative to Tesseract and GraphP, respectively.

We next examine performance improvement by showing the overall speedup, defined as the execution time of the four graph applications relative to Tesseract, in Figure 8. Examining the last bar of each application, we observe that all the applications improve over both Tesseract and GraphP. Particularly, compared with GraphP (i.e., the state-of-the-art work), the geometric mean speedup is 1.59×, 1.64×, 1.24×, 1.33× for PageRank, ATF, BFS and WCC, respectively. We conclude that due to lower cross-cube communication volume and a balanced computational load, the performance of SuperCut is strong for all of the applications and all of the graph datasets.

Turning our attention to how the energy consumption and performance benefit varies across applications, we observe a common relationship for both of them between applications with cross-cube communication and local memory access ratios. Figure 9 illustrates the average energy delay product (EDP) of each application across all the graph datasets with cross-cube communication ratio calculated as the average fraction of the data volume transferred across cubes to the overall data access volume. Here, we observe that high vertex activity applications (i.e., PageRank and ATF) with higher cross-cube communication ratio show more significant EDP reduction. This is consistent with a large communication volume within these applications. Since all the vertices in these applications are active in each iteration, the communication-to-computation ratio is

Table 1: Graph dataset.

Graph Name	Graph Type	Vertex Count	Edge Count	Description	Iteration Count	Maximum Degree	Average Degree
Wiki-Vote (WV)	Directed	7.1K	103.7K	Wikipedia who-votes-on-whom network [19]	200K	893	14.6
ego-Twitter (TT)	Directed	81K	1.8M	Social circles from Twitter [24]	500K	1205	21.7
Amazon0302 (AZ)	Directed	262.1K	1.2M	Amazon product co-purchasing network [18]	3M	5	4.7
Com-Amazon (AU)	Undirected	334.9K	925.9K	Amazon product network [39]	5M	168	2.8
Com-DBLP (DU)	Undirected	317.1K	1M	DBLP collaboration network [39]	5M	306	3.3
soc-LiveJournal1 (LJ)	Directed	4.8M	69M	LiveJournal online social network [20]	20M	20293	14.2

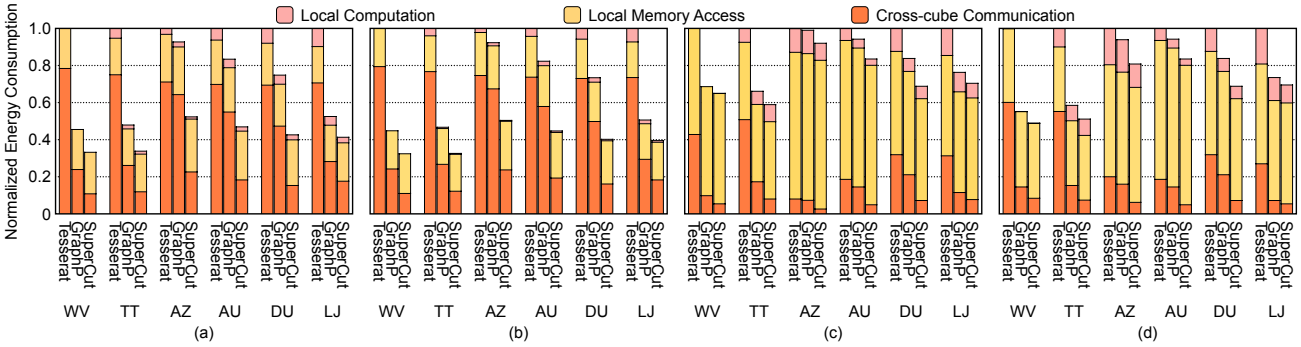

Figure 7: Normalized energy consumption breakdown of (a) PageRank, (b) ATF, (c) BFS, and (d) WCC applications, normalized to Tesseract. WV, TT, AZ, AU, DU and LJ are individual graphs.

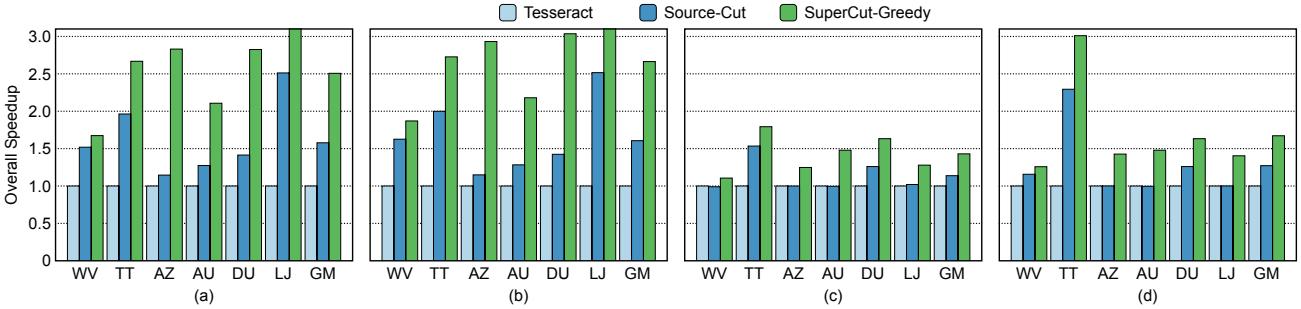

Figure 8: Overall speedup of (a) PageRank, (b) ATF, (c) BFS, and (d) WCC applications, normalized to Tesseract. WV, TT, AZ, AU, DU and LJ are individual graphs, GM is the geometric mean.

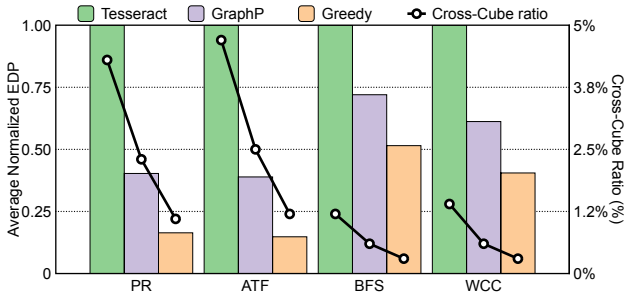

Figure 9: Average energy delay product and cross-cube communication ratio.

high, leading to greater potential benefits achievable by SuperCut. In contrast, the property of low vertex activity applications (i.e., BFS and WCC), that only a portion of vertices participate each iteration, leads to a lower communication ratio. Thus, SuperCut achieves lower EDP reduction on these applications. We conclude that SuperCut is most beneficial for high vertex activity applications with a larger cross-cube communication ratio.

5.2 Mixed-Cut Partitioning

As mentioned in Section 3, SuperCut incorporates both the mixed-cut partitioning method and the greedy algorithm, illustrated in Figure 3(b) and (c) respectively. To understand how different components contribute to the benefits of SuperCut in terms of performance and energy consumption, we implement mixed-cut partitioning

without greedy in SuperCut. Since all the graphs have similar tendency, here we take AZ as an example. Figure 10(a) shows the energy of all four applications on AZ. We have two observations: First, mixed-cut partitioning reduces the overall energy consumption by generating less communication volume than GraphP on all the applications, validating our assumption that recognizing more edge patterns is beneficial to communication reduction. Second, the greedy algorithm combined with mixed-cut partitioning further reduces cross-cube communication volume by optimizing the vertex distribution. Figure 10(b) illustrates the overall speedup. From the figure, we can draw the same conclusions about mixed-cut partitioning in terms of performance. Note that the high vertex activity applications benefit the most from the inclusion of the greed algorithm.

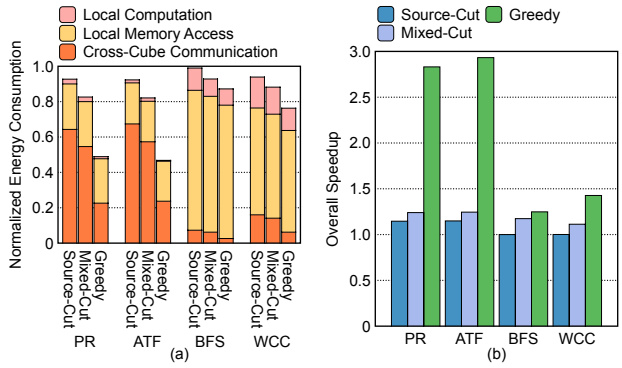

Figure 10: (a) Energy consumption breakdown and (b) overall speedup of mixed-cut partitioning on Amazon0302 (AZ), normalized to Tesseract.

5.3 Memory Footprint

Since SuperCut adopts the replica mechanisms of GraphP, both these works introduce extra memory footprint. In addition, SuperCut also generates destination-cut vertices during partitioning, the topological information of which is stored in memory. To assess the feasibility of SuperCut in terms of memory usage, we quantitatively examine the extra memory footprint of GraphP and SuperCut.

The evaluation results show that the extra memory footprint of SuperCut is 48%-75% of GraphP. This benefit comes from 3 facts: (1) SuperCut is better than GraphP at reducing the aggregated cross-cube communication volume (i.e., it introduces fewer replicas); (2) destination-cut vertices are only added for the pattern with multiple cross-cube edges, which guarantees that SuperCut has a lower memory footprint than GraphP; and (3) data for destination-cut vertices is buffered in the queue of DMAs instead of in memory.

6 RELATED WORK

6.1 Near-Memory Graph Processing Systems

Besides the comparison baselines of Tesseract and GraphP, there are other NMP architectures designed to accelerate large-scale graph processing. GraphPIM [26] proposes an instruction offloading mechanism to computation units on the logic layer of a single HMC device instead of a network consisting of multiple cubes. MessageFusion [2] proposes an NMP architecture to reduce cross-cube communication in transit by coalescing multiple cross-cube messages before reaching the same destination vertex. We take inspiration from this technique in our destination-cut static partitioning algorithm. GraphVine [3] explores another way to reduce HMC network congestion at runtime using multicast techniques. Both these works failed to optimize the distribution of vertices, limiting their efficiency. GraphH [8], GraphQ [43] and GraphRing [21] tried to regularize communication overhead by proposing reconfigurable HMC interconnection, a vertex reordering mechanism, and a ring-structured memory network, respectively. None of them directly reduced communication volume or considered graph distribution.

6.2 General Graph Partitioning Strategies

For general distributed graph processing systems, graph partitioning strategies also play a vital role in communication optimization and workload balance, which can be classified [9] into edge-cut and vertex-cut. PowerGraph [11] and PowerLyra [7] adopt vertex-cut to minimize vertex numbers across partitions by assigning edges to replicas in different machines. Although vertex-cut shows good load balance for skewed graphs, it is not suitable for near-memory graph processing because it leads to higher communication cost and requires more complicated implementation mechanisms. Therefore, the partitioning algorithms designed for near-memory graph processing, including the algorithms proposed in this work, are edge-cut [22, 23, 42] in which vertices of the graph are evenly assigned to minimize the number of edges across partitions. Pregel [23] is an early distributed graph processing system which adopts random edge-cut partitioning and provides the message-passing mechanism to deliver updates between machines. Tesseract adopts this approach. The partitioning proposed in GraphP [40] is also an edge-cut method in essence, in which out-going edges across memory cubes are partitioned. The basic principle of destination-cut partitioning as an edge-cut method where edges sharing a destination are combined has been adopted for traditional systems [11, 42]. In this work, we are interested in its effectiveness on near-memory systems, in which the overheads of a cross-cube data transfer are very different than a message-passing send/receive pair.

7 CONCLUSIONS

For many graph processing applications, especially those with high vertex activity, cross-cube communication is a performance bottleneck on multi-cube NMP architectures. Here, we propose SuperCut, a framework for near-memory architectures to effectively reduce communication overheads while maintaining computational balance. We evaluate SuperCut on an NMP architecture based on reconfigurable logic using 4 representative graph applications and 6 real-world graphs. Results show that it provides up to 1.8× total energy consumption reduction and 2.6× speedup with 45% lower extra memory footprint relative to the current state-of-the-art.

ACKNOWLEDGMENTS

This work is supported by NSF under grants CNS-1739643 and CNS-1763503.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proc. of 42nd International Symposium on Computer Architecture*. ACM, New York, NY, USA, 105–117.

[2] Leul Belayneh, Abraham Addisie, and Valeria Bertacco. 2019. MessageFusion: On-path message coalescing for energy efficient and scalable graph analytics. In *Proc. of IEEE/ACM International Symposium on Low Power Electronics and Design*. IEEE, 6 pages.

[3] Leul Belayneh and Valeria Bertacco. 2020. GraphVine: Exploiting multicast for scalable graph analytics. In *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 762–767.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[5] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. 2018. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *Proc. of 23rd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, USA, 316–331.

[6] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems* 30, 1-7 (1998), 107–117.

[7] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing* 5, 3 (2019), 13:1–13:39.

[8] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, et al. 2018. GraphH: A processing-in-memory architecture for large-scale graph processing. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 38, 4 (2018), 640–653.

[9] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, et al. 2018. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proc. of 39th Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 752–768.

[10] Mingyu Gao and Christos Kozyrakis. 2016. HRL: Efficient and flexible reconfigurable logic for near-data processing. In *Proc. of IEEE Int'l Symp. on High Performance Computer Architecture*. IEEE, 126–137.

[11] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. of 10th USENIX Symp. on Operating Systems Design and Implementation*. USENIX, 17–30.

[12] Sungpack Hong, Semih Salihoglu, Jennifer Widom, and Kunle Olukotun. 2014. Simplifying scalable graph processing with a domain-specific language. In *Proc. of IEEE/ACM Int'l Symp. on Code Generation and Optimization*. ACM, New York, NY, USA, 208–218.

[13] Joe Jeddeloh and Brent Keeth. 2012. Hybrid memory cube new DRAM architecture increases density and performance. In *Proc. of Symposium on VLSI Technology (VLSIT)*. IEEE, 87–88.

[14] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. 2017. HBM (high bandwidth memory) DRAM technology and architecture. In *Proc. of IEEE International Memory Workshop (IMW)*. IEEE, 4 pages.

[15] Andrew B Kahng, Jens Lienig, Igor L Markov, and Jin Hu. 2011. *VLSI Physical Design: From Graph Partitioning to Timing Closure*. Springer.

[16] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392.

[17] Gwangsun Kim, John Kim, Jung Ho Ahn, and Jaeha Kim. 2013. Memory-centric system interconnect design with hybrid memory cubes. In *Proc. of 22nd Int'l Conf. on Parallel Arch. and Compilation Techniques*. IEEE, 145–155.

[18] Jure Leskovec, Lada Adamic, and Bernardo Huberman. 2007. The dynamics of viral marketing. *ACM Trans. on the Web* 1, 1 (2007), 5:1–5:39.

[19] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Signed networks in social media. In *Proc. of SIGCHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 1361–1370.

[20] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

[21] Zerun Li, Xiaoming Chen, and Yinhe Han. 2022. GraphRing: an HMC-ring based graph processing framework with optimized data movement. In *Proc. of 59th ACM/IEEE Design Automation Conference*. ACM, New York, NY, USA, 1063–1068.

[22] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012), 716–727.

[23] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proc. of ACM Int'l Conf. on Management of Data*. ACM, New York, NY, USA, 135–146.

[24] Julian McAuley and Jure Leskovec. 2012. Learning to Discover Social Circles in Ego Networks. In *Proc. of 25th International Conference on Neural Information Processing Systems*. Curran Associates, Inc., 539–547.

[25] Batul J Mirza, Benjamin J Keller, and Naren Ramakrishnan. 2003. Studying recommendation algorithms by graph analysis. *Journal of Intelligent Information Systems* 20, 2 (2003), 131–160.

[26] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks. In *Proc. of IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 457–468.

[27] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. 2016. Energy Efficient Architecture for Graph Analytics Accelerators. In *Proc. of 43rd International Symposium on Computer Architecture*. IEEE, 166–177.

[28] Seth H Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. 2014. NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 190–200.

[29] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. MachSuite: Benchmarks for accelerator design and customized architectures. In *Proc. of Int'l Symp. on Workload Characterization*. IEEE, 110–119.

[30] Samuel Rogers, Joshua Slycord, Mohammadreza Baharani, and Hamed Tabkhi. 2020. gem5-SALAM: A System Architecture for LLVM-based Accelerator Modeling. In *Proc. of 53rd IEEE/ACM Int'l Symp. on Microarchitecture*. IEEE, 471–482.

[31] Alessandra Sala, Lili Cao, Christo Wilson, Robert Zablit, Haitao Zheng, and Ben Y Zhao. 2010. Measurement-calibrated graph models for social network experiments. In *Proc. of 19th Int'l Conf. on World Wide Web*. ACM, New York, NY, USA, 861–870.

[32] Manjunath Shevgoor, Jung-Sik Kim, Niladrish Chatterjee, Rajeev Balasubramonian, Al Davis, and Aniruddha N Udipi. 2013. Quantifying the relationship between the power delivery network and architectural policies in a 3D-stacked memory device. In *Proc. of 46th IEEE/ACM Int'l Symp. on Microarchitecture*. IEEE, 198–209.

[33] Gagandeep Singh, Mohammed Alser, Damla Senol Cali, Dionysios Diamantopoulos, Juan Gómez-Luna, Henk Corporaal, and Onur Mutlu. 2021. FPGA-based Near-Memory Acceleration of Modern Data-Intensive Applications. *IEEE Micro* 41, 4 (2021), 39–48.

[34] Gagandeep Singh, Lorenzo Chelini, Stefano Corda, Ahsan Javed Awan, Sander Stuijk, Roel Jordans, Henk Corporaal, and Albert-Jan Boonstra. 2019. Near-memory computing: Past, present, and future. *Microprocessors and Microsystems* 71 (2019), 102868.

[35] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2014. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *Proc. of IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 550–559.

[36] Christian L Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. 2016. NetworKit: A tool suite for large-scale complex network analysis. *Network Science* 4, 4 (2016), 508–530.

[37] Kizheppatt Vipin and Suhaib A Fahmy. 2018. FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *Comput. Surveys* 51, 4 (2018), 72:1–72:39.

[38] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE Trans. on Neural Networks and Learning Systems* 32, 1 (2020), 4–24.

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

[40] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing communication for PIM-based graph processing with efficient data partition. In *Proc. of Int'l Symp. on High Performance Computer Architecture*. IEEE, 544–557.

[41] Chenfeng Zhao, Xuan Zhang, and Roger D Chamberlain. 2022. Executing Data Integration Effectively and Efficiently Near the Memory. *IEEE Design & Test* 29, 2 (2022), 65–73.

[42] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *Proc. of USENIX Symp. on Operating Systems Design and Implementation*. USENIX, 301–316.

[43] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. GraphQ: Scalable PIM-based graph processing. In *Proc. of 52nd IEEE/ACM International Symposium on Microarchitecture*. ACM, New York, NY, USA, 712–725.