Go Local: Software/Hardware Co-design of 3D Manycore GPU Architectures for Accelerated Graph Computations

Manycore GPU architectures have become mainstay for accelerating graph computations. However, the main bottleneck of graph computations on manycore architectures is the data movement due to random main memory accesses. Poor data locality in real-world graphs exacerbates the bottleneck arising from random main memory accesses. Vertex reordering is a popular technique to improve locality by placing the vertices near their neighbors. However, as most real-world graphs are inherently irregular, different vertices have varying number of neighbors, which leads to uneven amount of memory requests. Therefore, node reordering schemes need to be complemented with efficient task allocation on manycore GPU architectures to reduce latency due to local cache misses. We introduce a software/hardware co-design framework for accelerating graph computation where an architecture-aware vertex reordering is coupled with a priority-based task allocation technique. As the task allocation aims to reduce on-chip latency and associated energy, the choice of Network-on-Chip (NoC) as the communication backbone in the manycore platform is an important parameter. By leveraging emerging three-dimensional (3D) integration technology, we propose design of a small-world NoC (SWNoC)-enabled manycore GPU architecture, where the placement of the links connecting the streaming multiprocessors (SM) and the memory controllers (MC) follow a power-law distribution. The proposed 3D SWNoC-enabled software/hardware co-design framework achieves 11.1% to 22.9% performance improvement and 16.4% to 32.6% less energy consumption depending on the dataset and the graph application when compared to the default order of dataset running on a conventional planner mesh architecture.

CCS CONCEPTS • Computer systems organization ~ Architectures ~ Other architectures ~ Special purpose systems

Additional Keywords and Phrases: Software/Hardware Co-design, Vertex Reordering, Graph Analytics, GPU Manycore, Small World NoC

1 INTRODUCTION

Graph analytics have become a central part of several data analytics and machine learning workflows used in many scientific and industrial application domains. Spurred by the advances in high-throughput technologies to generate data, graphs have become an effective way to represent relational data that are characterized by pairwise interactions or relationships (edges) between different entities (vertices). Graph theory is used to model and study a wide spectrum of complex networks from both the natural world and human-engineered systems, fueling their discovery pipelines. However, with the steep increases in both the volume of observable data and the diversity in applications, performance scalability of graph analytical pipelines has become a significant challenge. Consequently, the implementation of graph analytics on emerging manycore platforms is gaining momentum [1] [2] [3] [4] [5] .

Graph applications pose several inherent challenges when mapped on to a manycore platform. Real-world graphs can be large consisting of millions to billions of edges, making any computation on the graph objects memory intensive. This demand for large memory is exacerbated by the scale-free characteristics of real-world

networks [6] , which causes poor locality and therefore, a considerable degree of irregularity in data access patterns during computation. Poor locality in particular affects the performance of a majority of graph operations as they heavily rely on an ability to access vertex neighborhoods in a fast and efficient way. Consequently, preserving locality of graph structures is critical in reducing data movement overheads and improving application performance, particularly on modern day manycore platforms which have experienced increasing heterogeneity at the processing layers and deeper hierarchies at the memory layer.

Vertex (re)ordering is an effective way to improve locality of graph structures [6] . Given a graph *G(V,E)* with *n* vertices and *m* edges, the goal of vertex ordering is to compute an ordering of the *n* vertices in such a way that the average "gap" distance between an arbitrary vertex and any of its neighbors (along the ordering) is minimized. Intuitively, a better "packing" of neighboring vertices in nearby memory locations (i.e., spatial locality) could lead to reduced cache misses and therefore, reduced data movement. In iterative graph applications where the vertex neighborhoods are scanned repeatedly until a certain convergence criterion is met (e.g., PageRank [7] , community detection [8] , 1-distance coloring [3]), a better ordering could also lead to a better data reuse (i.e., temporal locality).

While numerous vertex ordering schemes have been independently proposed and evaluated for traditional manycore platforms [6] , relatively fewer efforts exist for evaluating and exploiting vertex reordering on emerging single chip manycore architectures. Manycore architectures present a host of additional challenges for graph applications. Reducing the on-chip data movement, both for *volume* and *latency*, could have a significant impact on both performance and energy overheads. While techniques such as vertex reordering can help with volume, latency is still very much a hardware trait. In other words, the performance reach of software-only solutions is limited by the physical characteristics of the underlying hardware. For instance, on a single chip manycore platform integrating a massive number of processing elements (conventional processors or GPUs) using an on-chip interconnect, the topology of the underlying network or how the different individual work units (or tasks) are allocated relative to the locations of the memory controllers, could all dictate data access latency. The gains achieved by running a smarter locality-maximizing scheme at the software level could be potentially lost or compromised by a slow network that increases latency of data movements when they occur (owing to cache misses). On the other hand, a straightforward mapping of an input graph (without taking advantage of its input characteristics or a better locality-preserving permutation) on to an otherwise highly optimized hardware, say with a fast low-latency network, could also lead to suboptimal performance outcomes. Consequently, we posit that a carefully designed software/hardware co-design is essential for boosting the performance and energy cost profiles of graph processing manycore accelerators.

In this paper, we present a software/hardware co-design framework for a graph accelerator. Our hardware platform is an optimized manycore GPU architecture that consists of streaming multiprocessors (SM) and memory controllers (MC) connected through a three-dimensional (3D) Network-on-Chip (NoC). While GPUs are becoming an increasingly popular choice for numerous application workloads, the NoC paradigm offers the network backbone required to integrate a massive number of such GPU SMs and MCs on a single chip. However, the combination of these two powerful paradigms has not been sufficiently evaluated or successfully demonstrated for graph processing. More specifically, several key questions remain to be answered for a successful deployment: (i) how to exploit input graph characteristics both at the software (algorithmic) and hardware (architectural) layers? (ii) how to improve locality of graph structures during computation? (algorithmic) (iii) how to design an efficient architecture that is best suited to execute irregular graph workloads?

(architectural) (iv) how to assign tasks on the chip in a hardware-aware manner so as to maximize concurrency while reducing communication latency and energy costs on the chip? (task scheduling).

Our software/hardware co-design approach for designing a graph accelerator addresses all the above questions. More specifically, the contributions of the paper are as follows:

- (software-level) We propose an efficient vertex reordering-based approach to map graph application workloads on to a target manycore GPU platform. The approach is geared toward exploiting input characteristics toward improving locality and reducing the overall volume of data movement.
- (runtime scheduling) We propose a novel priority-based runtime task scheduling scheme to allocate the individual thread work units (tasks) on to the SMs so as to reduce the overall data transfer latency.
- (hardware-level) We present the design of an optimized 3D NoC architecture that optimizes the placement of the SMs and MCs on chip, in a way that best fits the irregular graph application workloads and is capable of generating further reductions to communication latency on top of the gains provided in software.
- (evaluation) We present a thorough evaluation of our proposed software/hardware co-design on various real-world graph inputs with different characteristics, using three different graph operations – namely, PageRank, Single Source Shortest Path (SSSP), and Coloring (Color). Our results demonstrate that the proposed software/hardware co-design framework improves the performance by 11.1% to 22.9% and reduces energy consumption by 16.4% to 32.6% depending on different datasets and applications compared to a default order of graph dataset running on a 2D mesh architecture.

The rest of the paper is organized as follows. Section 2 presents the related work. In Section 3, we discuss the proposed software/hardware co-design framework for accelerating graph applications. Section 4 presents our experimental results and evaluation. Finally, in Section 5, we conclude the paper by summarizing the salient features of this work.

2 RELATED WORK

Designing specialized manycore architectures for graph analytics has been an area of active research in recent years. GPU-based manycore computing offers a promising direction toward accelerating graph applications. However, such platforms have deep memory hierarchies, which could exacerbate the costs in moving data for graph applications [1] . One possible way is to modify the organization of caches and partition these caches into multiple planar layers in a 3D structure to improve the cache hit rate with low access time [10] . Hardware architectures can also be customized for different vertex-centric applications by inserting application-level data structures and functions [11] . However, poor data locality and high memory bandwidth requirement in graph computation create significant amount of data movements. This data movement between processing and memory layer degrades the performance and energy consumption for graph-based applications in conventional architectures with external DRAM [12] [13] . Therefore, optimizing memory access is one of the primary objectives in designing graph accelerators. For example, Graphicionado [14] improves memory throughput by replacing random accesses with sequential accesses to scratchpad memory, whereas Tunao [15] devotes a dedicated on-chip buffer to store high degree vertices. Ozdal et al. propose an architecture specifically optimized for iterative graph applications with irregular access patterns and asymmetric convergence [11] . Utilizing DRAM-based Hybrid Memory Cube (HMC) is another way to enhance performance of graph accelerators [16] [1] . Resistive Random-Access Memory (ReRAM) based graph accelerators have been shown to significantly outperform CPU- or GPU-based systems both in terms of execution time and energy [17] [18] [19] [20] .

Locality in graph computations can help reduce random memory access requests. Vertex reordering is a technique that is often used to improve locality in graph computation [6] . However, since computing an optimal vertex reordering that minimizes linear gap measures is an NP-Hard problem [21] ,[22] multiple (efficient) heuristics are used in practice. The simplest of schemes use the degree information to order the vertices. This includes schemes such as Degree Sort that sorts all vertices by their degree, and schemes that focus only on "hub" vertices that have a large degree [23] [24] [25] . A related class of ordering schemes comes from the sparse linear algebra community. Referred to as fill-reducing techniques [26] , these schemes attempt to minimize the number of non-zeros (or fill) in the factorized matrix post-reordering. The Reverse Cuthill-McKee (RCM) method [27] , nested dissection [28] , and minimum degree methods [29] are examples of this class. Another class of techniques is window-based that use a sliding window of vertices to rearrange the vertices within each window. Gorder [30] is one such scheme that uses the number of common neighbors between any two vertices as a way to improve locality. Finally, there is a class of ordering schemes that are based on partitioning. More specifically, this approach first partitions the graph using partitioners (e.g., METIS [31]) or community detection tools (e.g., Grappolo [8]), and subsequently use the partitions to generate a linear ordering of the vertices. The idea is to use the partitions to identify highly connected parts of the graph, which in turn can aid in improving locality. A recent study [6] empirically evaluating these different classes showed the clear benefits of using partitioning- and fill-reducing-based schemes over simpler (and lighter-weight) degree-based and window-based schemes. However, these schemes have not been evaluated yet taking into account hardware architecture characteristics such as the on-chip network topology or using objectives that relate to latency and energy overheads.

In this paper, we propose a hardware-aware reordering scheme, which complements the priority-based task scheduling in a 3D NoC-enabled manycore GPU architecture. While improving the locality, the hardware-aware reordering scheme coupled with priority-based scheduling helps to reduce latency and energy overhead by generating a permuted array of vertices and assigning them to suitable SMs.

3 SOFTWARE/HARDWARE CO-DESIGN

We present a software/hardware co-design for accelerating graph computations on 3D manycore GPU architectures. Given an input graph, our co-design approach seeks to efficiently execute any arbitrary graph operation (e.g., PageRank [48] , Single Source Shortest Path [32] or any other computation that is defined on the graph). Our co-design framework has three major components, as illustrated in Fig. 1:

At the software layer, we present a vertex reordering-based approach (Sec. 3.1) as a way to improve the locality of graph structures for use in computation. With architecture-awareness, this step improves locality with the intent of eventually reducing the volume of on-chip data movement.

However, even with an ideal locality-preserving ordering, if the work units – fixed-size blocks of vertices assigned to each thread-block – are placed on-chip significantly far away from the memory banks, then the performance gains achieved through an optimal ordering could be potentially compromised. For this reason, at the scheduler level, we present a dynamic priority-based task allocation scheme (Sec. 3.2), which determines the optimal on-chip locations (processing elements) to execute each unit of work. This step uses information from both the data layer (specifically, the expected data needs of a block) and the hardware layer (specifically, the underlying architectural topology). The aim is to place each work unit as close as possible to the memory controllers that can supply dependent data so that the average latency of data movement is reduced.

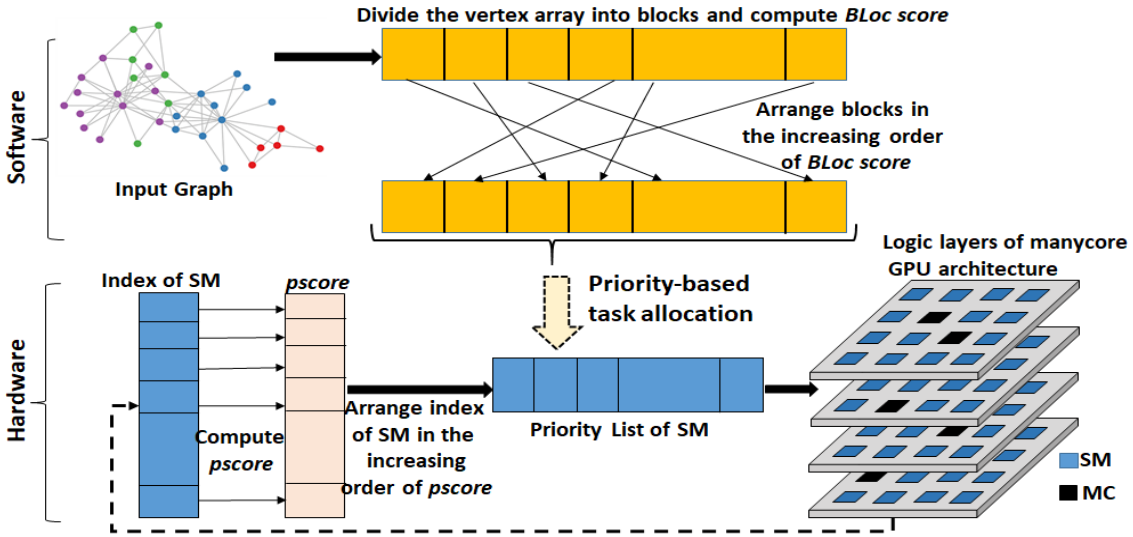

Fig. 1: Illustration of workflow in the proposed software/hardware co-design

Finally, in order to further reduce the latency and energy of data accesses on the chip, we present an optimized 3D manycore GPU architecture at the hardware layer (Sec. 3.3). In particular, we design a 3D Network-on-Chip (NoC)-based manycore GPU architecture with a small-world topology that is optimized to reduce the data movement latency and energy of graph computations.

This three-layered co-design, starting from the software to the scheduler to finally the hardware, makes for a crosscutting approach that is best equipped to complement the gains achieved in each layer. As we will show through experiments (Sec. 4), this multi-layered approach is capable of overcoming the limits of acceleration that can be achieved at any one particular layer.

3.1 Architecture-aware Vertex Reordering

3.1.1 Reordering Schemes and Quality Metrics

Vertex (re)ordering represents a class of techniques used to order the set of vertices in a graph in such a way that improves locality. Intuitively, the goal is to bring vertices that share an edge between them as near as possible in the ordering so that the probability of them co-locating in cache lines is high. More formally, let $G = (V, E)$ be an input graph with *n* vertices (in V) and *m* edges in E. As a convention we assume that the vertices are identified by integer labels from 1 through *n*. A *vertex ordering*, denoted by Π, of V represents a linear arrangement of V; i.e., a permutation $\Pi : i \rightarrow [1, n]$, for every vertex $i \in V$. The assignment $\Pi(i)$, is also called the *rank* of the vertex i.

Natural ordering: In practice, the user provides the input graph with a predefined order of the vertices. We refer to this input ordering as the *natural ordering* of V. In the natural ordering, $\Pi(i) = i$, for every vertex $i \in V$. Since an ordering scheme takes an input with this natural ordering, the process of generating a Π is also referred to as a "reordering".

Linear gap: To measure the goodness of ordering, there have been several measures defined. The most widely used measure is the *average linear arrangement gap* [21] . Given an ordering Π of V, the *average linear*

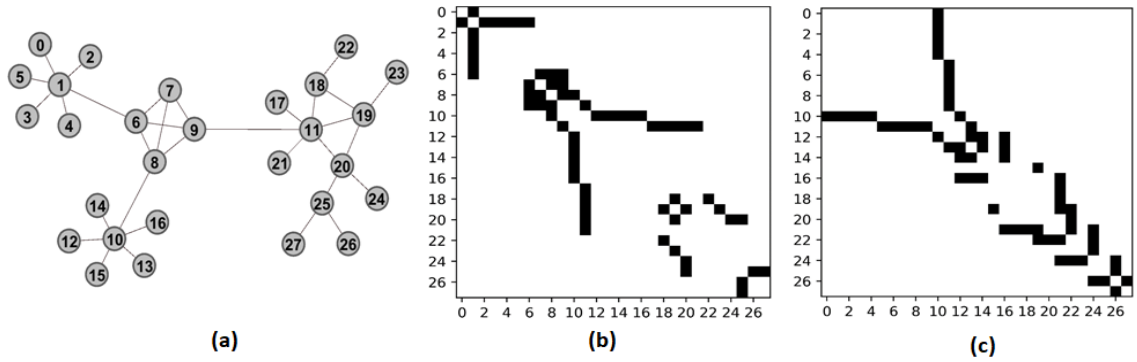

Fig. 2: Part (a) shows an example input graph in its natural order. Part (b) shows the natural order's adjacency matrix view of the non-zeroes (depicted as the dark cells). Part (c) shows the same graph input reordered using the RCM order. The RCM reordering provides a better clustering of the non-zeroes along the main diagonal, indicating better locality.

arrangement gap is defined as the absolute difference between the ranks of every adjacent pair of vertices, averaged over all edges. In other words, for each edge $(i,j) \in E$, the gap in Π between vertices *i* and *j* is given by: $\xi_{\Pi}(i,j) = |\Pi(i) - \Pi(j)|$. The *average linear arrangement gap* is then defined as:

$$\hat{\xi}(G,\Pi) = \frac{1}{m}\sum_{(i,j)\in E} \xi_{\Pi}(i,j) \quad (1)$$

Intuitively, smaller the value of $\hat{\xi}$, the better the quality of ordering. However, the problem of computing an ordering that minimizes the linear arrangement gap score is NP-Hard [22] . Consequently, various efficient heuristics are used to generate a reordering [32] . These schemes range from light-weight (degree-based) to more heavy-weight (window- and partitioning-based) schemes. A more detailed survey and empirical analysis of the different ordering schemes is available in [6] .

Ordering Schemes: In what follows, we briefly summarize the main ideas of four such node ordering schemes, which we evaluate in our software/hardware co-design framework.

1. **Degree Sort** uses the idea of sorting the vertices by their vertex degrees (defined as the number of edges incident on the vertex). Since the degree of a vertex is a number that ranges from $[0, n-1]$, we can use integer sorting methods to quickly generate a degree-based ordering. While this allows for a light-weight reordering, there are no particular locality properties that can be guaranteed through Degree Sort (or any of its variants [24]).
2. **Gorder** uses a window-based approach to minimize the number of cache misses expected to be generated due to the ordering [30] . Given the natural ordering of V, Gorder slides a window of a certain length over V and within each window, maximizes a score defined over the number of shared neighbors between any two vertices of that window. Intuitively, the goal is to improve the odds that if vertices i and j share a large fraction of neighbors in common, then they should also be co-located within the same window.
3. **Reverse Cuthill-McKee (RCM)** [27] is a type of fill-reducing strategy which tries to pack as many non-zeros as possible near the main diagonal of the adjacency matrix representation of the input graph. It works by performing an interleaved breadth-first search (BFS) and depth-first search (DFS) traversal

of the graph. It starts at a vertex with the smallest degree, subsequently visiting all its unvisited neighbors in the non-decreasing order of their degrees. This procedure is then repeated for each of the next level of neighbors.

4. **Grappolo** [8] is a partitioning-based scheme, which uses community detection by modularity optimization [33] to divide the graph into several communities (or disjoint clusters). The communities are then arbitrarily arranged such that the vertices originating from the same community are numbered in a contiguous manner in the output ordering. Since modularity optimization is aimed at identifying tightly-knit groups of vertices, this ordering is expected to preserve a high degree of locality.

As a representative example, Fig. 1 illustrates an input graph (a) along with the adjacency matrix profile of (b) natural order and (c) RCM order. We can see from Fig. 1 that how the adjacency matrix of the input graph is affected by RCM reordering scheme towards the goal of packing non-zeros closer to the diagonal.

3.1.2 Architecture-aware shuffling

In the GPU architecture, the main memory is divided into several banks, each of which is connected to one Memory Controller (MC). Different thread blocks process different subsets of vertices, and the vertex neighborhood determines the memory regions accessed. Ideally, we should map each thread block to a streaming multiprocessor (SM) such that the total number of communication hops between the SM and the MCs corresponding to the memory regions accessed by the thread block is minimized. While the exact thread block to SM mapping is vendor's proprietary information, simple models such as Round-robin distribution do not take advantage of the thread block-MC affinity. The exact per thread block-MC affinity is determined both by the application and the dataset.

In this work, we introduce a novel, lightweight, task allocation strategy that minimizes the average communication hops between SM and MC. Our solution is based on the key observation that the memory regions accessed while processing a vertex are dependent on its neighborhood – for instance, a vertex connected to, say five neighboring vertices typically tends to draw its updates from only those five neighbors. If the vertex ordering step was successful in placing those five neighbors in nearby indices (i.e., ranks in the output permutation Π), then with high probability those neighbor data can also be expected to be present within the same cache line. However, in scale-free graph inputs with a power-law degree distribution, this cannot always be guaranteed. If the dependent data is not in local cache, the data has to be fetched by the main memory, which requires SM-MC communication. This implies that we have to prioritize the placement of thread-blocks that process the vertices with a lot of non-local neighbors. To this end, we define a metric called the *block locality score* to estimate the expected communication overhead that would originate from a given thread block. The proposed algorithm then uses the *block locality score* metric to prioritize task allocation (i.e., a mapping of thread block to SM) such that the total number of hops is minimized.

Block locality score: We divide the entire set of vertices into uniform row panels according to the number of thread blocks. The size of each row panel (i.e., number of vertices per row panel) depends on the number of threads in a thread block. Henceforth, we refer to each such row panel as a *block.* Subsequently, we define a *block locality score* (abbreviated as the *BLoc* score), which is a qualitative measure for the locality within a block. More formally, we define the *BLoc* score to be the ratio of the number of intra-block edges (i.e., edges connecting any pair of vertices within that block) to the total number of edges incident on any vertex of the block.

Let i_0 and j_0 denote the beginning and ending indexes of a block *P(V',E')* in a graph G*(V,E)* and the delta function is defined as:

$$\delta(i,j) = \begin{cases} 1, & if(i,j) \in E \\ 0, & otherwise \end{cases} \quad (2)$$

Hence, the *BLoc score* is given by:

$$BLoc\ score = \frac{\sum_{i=i_0}^{j_0-1} \sum_{j=i+1}^{j_0} \delta(i,j)}{\sum_{k=i_0}^{j_0} d_k - \sum_{i=i_0}^{j_0-1} \sum_{j=i+1}^{j_0} \delta(i,j)} \quad (3)$$

where, d_k denotes the degree of vertex *k*. Therefore, if the *BLoc score* of a block has relatively high value, it signifies that associated vertices have higher number of neighbors inside the block and less number outside. In other words, most edges incident on the set of vertices in the current block is intra-block as opposed to inter-block. Hence, higher *BLoc score* signifies high data reuse, fewer local cache misses, and less main memory traffic. In contrast, blocks with high inter-block edges and low intra-block edges will have low *BLoc score* and will suffer from low data reuse, and high local cache misses.

Shuffling: The thread blocks must be mapped to SMs such that memory access latency and energy are minimized. Blocks with high global memory traffic should be executed on SMs that are close to MCs. Note that for a given NoC architecture, only a limited number SMs can be placed close to MCs. Also, note that assigning all the blocks to SMs towards the end of task allocation will result in load-imbalance and resource underutilization. We use *BLoc score* as a metric to determine the priority of each block. We rearrange the blocks in *increasing* order of their *BLoc scores*, and the blocks are scheduled for execution in this order. Thus, blocks at the beginning of reordered vertex array (i.e., with lower *BLoc score*) will receive a higher precedence in being assigned to SMs closely located to MCs. In what follows (Sec. 3.2) we elaborate on how this priority-based task allocation is achieved.

3.2 Data Movement-aware Priority-based Task Allocation

As mentioned in Section *3.1.2*, blocks with low *BLoc scores* (i.e., poor intra-block locality) should be mapped to SMs located close to those MCs that are most likely to cater to the block's data requests, thereby reducing the total execution time and energy. To determine this mapping, we need to characterize the closeness of SMs to MCs. A simple metric to use can be the average distance, which is the average number of hops between a given SM and all MCs. However, this metric assumes that the memory traffic between a given SM and MCs is uniform, which is generally not the case for most real-world graphs – i.e., skewed degree distributions could result in skewed SM to MC traffic patterns as well. Hence, we need to assign weights proportional to the memory traffic between a given SM and a given MC. The memory traffic volume between any SM and an MC is dictated by the number of edge dependencies between the vertices mapped to that SM and the vertices covered by that MC. Therefore, we define a weight W_p for the p^{th} MC to denote the fraction of the overall traffic that can be attributed to this MC (from any SM). This weight W_p is given by:

$$W_p = \frac{\sum_{t=1}^{T} M_{tp}}{\sum_{m=1}^{N} \sum_{t=1}^{T} M_{tm}} \quad (4)$$

where *N* is the total number of MCs, *T* is the total number of thread blocks. M_{tp} denotes the number of edges connecting the set of vertices in t^{th} thread block to the vertices covered by the p^{th} MC. We introduce a *proximity*

score (abbreviated as the *pscore*) to characterize the weighted closeness on-chip of an SM to all its dependent MCs, as follows:

$$pscore(q) = \sum_{i=1}^{N} W_p * H_{pq} \quad (5)$$

where H_{pq} denotes the number of hops between p^{th} MC and q^{th} SM on the chip. An SM with a low *pscore* value signifies less long-range (multi-hop) communication and is therefore a better choice for thread blocks with low *Bloc score* (poor locality). An SM priority list is generated by sorting the SMs in the increasing order of the *pscore* value (the lower the *pscore* value, the higher the priority). The thread blocks are selected in the increasing order of *Bloc scores* and mapped in a Round-robin fashion to the SMs based on the SM priority list. Thus, thread blocks with low *Bloc score* have a higher probability of getting mapped to an SM with a low *pscore* and thereby minimizing the total number of communication hops.

3.3 Optimized NoC-based Architecture Design

In the priority-based scheme, the priority of SM is determined by the *pscore* parameter defined in the previous section. As the *pscore* for each SM depends on the number of hops between SM and MC, choice of NoC architecture influences the *pscore*. The number of SMs is generally larger than that of MCs. Hence, graph operations mapped onto GPU-based manycore architectures are expected to predominantly give rise to many-to-few and/or few-to-many traffic patterns. Henceforth for convenience, we simply refer to this pattern as "many-to-few". This is owing to fact that the SMs process vertices in parallel. Consequently, SMs assigned to process high degree vertices will generate more traffic to MCs than the others. The many-to-few type of data exchange gives rise to long-range traffic patterns as it will be impractical to assume all communicating SMs and MCs can be placed in close proximity on-chip. As long-range data movement requires multiple hops; it increases latency and energy overhead. Therefore, on-chip traffic pattern needs to be analyzed to design suitable NoC for graph-based applications. Hence, we studied the on-chip traffic pattern through the hop count distributions of three graph applications: PageRank, Color, and SSSP taken from the Pannotia suite [34]. We considered five different

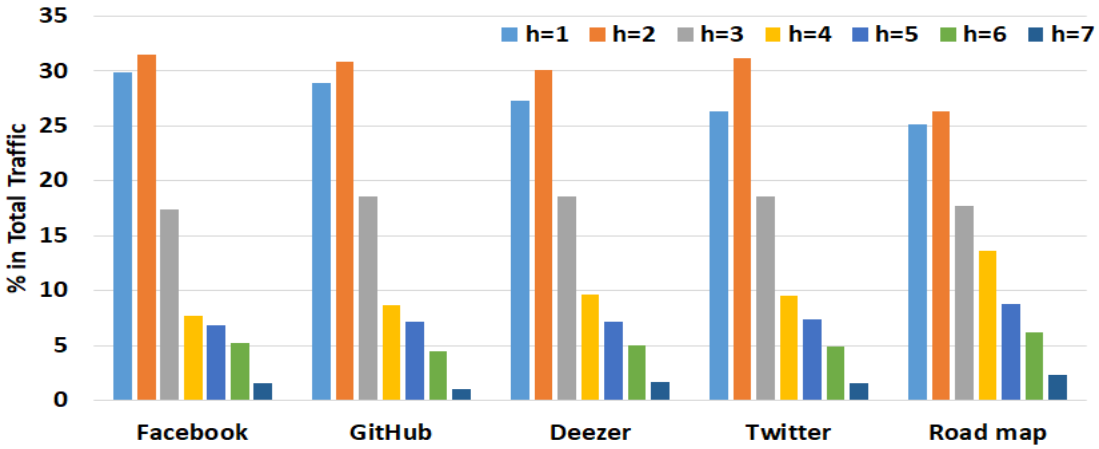

Fig. 3 Hop count distribution in 2D mesh NoC for PageRank with different graph datasets (hop counts are given by *h* in the legend).

graph datasets, viz., Facebook [35] , GitHub [36] , Deezer [37] , Twitter [38] and Road map [39] respectively and map the graph applications on to a traditional 2D mesh-based NoC architecture. Fig. 3 shows the traffic distribution using the PageRank application as an example; similar patterns (not shown) were observed with the other two graph applications. It is evident from Fig. 3 that for all the datasets, there is a significant amount of traffic exchange between SMs and MCs separated by more than two hops. We call the traffic with more than two hops as a "long-range" traffic. Hence, all the graph inputs considered here generated a significant amount of long-range traffic. These traffic distributions suggest that designing a small world NoC (SWNoC) is suitable for the graph applications under consideration. It has been already shown that either by inserting long-range shortcuts in a regular mesh to induce small-world effects or by adopting power-law based small-world connectivity, we can achieve significant performance gain and lower energy dissipation compared to traditional multi-hop mesh networks [40] [41] [42] [43] [44] .

In the proposed SWNoC architecture, SMs and MCs are connected using a small-world interconnection network, where the links are established following the power law distribution. More precisely, if Euclidean distance between core (SM/MC) *u* and *v* is d_{uv}, the probability *P(u,v)* of establishing a link between these two cores is proportional to the distance d_{uv}, raised to a finite power. We can represent the probability *P(u,v)* as follows:

$$P(u,v) = \frac{d_{uv}^{-\alpha}}{\sum_{\forall c}\sum_{\forall d} d_{cd}^{-\alpha}} \quad (5)$$

Here, parameter *α* governs the nature of connectivity. A larger *α* means a locally connected network with a few or even no long-range links. On the other hand, a zero value of *α* generates an ideal small-world network following the Watts-Strogatz model [43] - one with long-range shortcuts that are virtually independent of the distance between the cores. It has been shown that the average hop count is minimum with a fixed wiring cost for the value α as 1.8 [45] . It should be noted that when a small-world network is implemented in a planar (2D) structure, there will be multiple physically long wires connecting the largely separated cores. Ultimately, this will give rise to high timing and energy overheads. However, when a small-world NoC is implemented using 3D integration, the largely separated cores in a 2D structure can be placed in different planar dies and connected using vertical links. As the vertical links are much smaller in length, the 3D SWNoC reduces the timing and

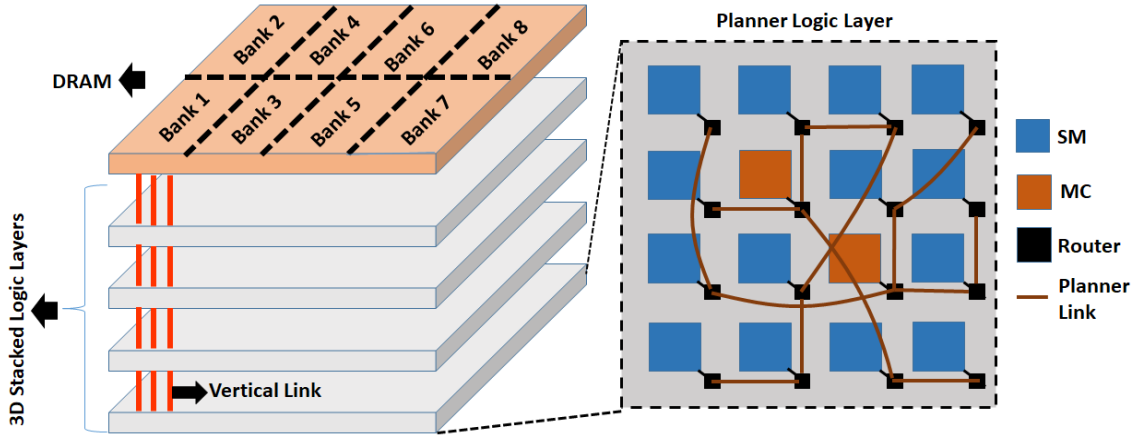

Fig. 4: Proposed 3D SWNoC based manycore GPU architecture.

energy costs [44] . Therefore, in this work, we use the manycore GPU architecture interconnected via the 3D SWNoC designed using the power law model of (5) as the computing substrate for the graph applications under consideration.

4 EXPERIMENTAL RESULTS

4.1 Experimental Setup

We use GPGPU-Sim [46] to simulate the performance of the proposed manycore GPU architecture in presence of the software/hardware co-design framework. We use Booksim [47] for implementing the various NoC architectures considered in this work. For profiling energy consumption, we utilize GPUWattch [48] . The GPUs are based on the NVIDIA Volta architecture and here we have considered 56 SMs and 8 MCs to give rise to a system with 64 cores where SMs and MCs, both are described as cores. In 2D mesh, 64 cores are arranged in an 8x8 grid on a planner layer. For 3D SWNoC, 64 cores are equally partitioned into four planar layers which are connected through vertical links with each other. Within each layer, 16 cores are placed in a 4x4 grid pattern. The memory unit is equally divided into 8 memory banks and each of them is connected to one MC. We distribute the SMs and MCs among the four planar layers to achieve high throughput. The configuration with MCs uniformly distributed among four planar layers (each layer has 2 MCs) provided the highest throughput. Fig. 4 illustrates the proposed 3D SWNoC based manycore GPU architecture. For all experiments, we considered DRAM-based memory.

For full-system performance evaluation, we consider three well-known graph applications, viz. PageRank, Color, and SSSP taken from the Pannotia suite [34] . These three were chosen due to their varying algorithmic properties. PageRank is a classic case of iterative vertex-centric graph computation where each vertex' state is affected by its neighbors iteratively until convergence. "Color" implements graph coloring, which is also vertex-centric albeit an exemplar of a greedy class of approaches that feature in independent set problems. SSSP is single source shortest path, which performs a sweep of the full graph (much like a BFS). Five different datasets (shown in Table 1) were considered including two social networks with skewed degree distribution, and a road map network with largely uniform vertex degree distribution.

4.2 Impact of Node Reordering on Performance

In this section, we analyze the performance of different vertex reordering schemes when they are mapped to a 2D mesh-based manycore system. We use a simple mesh in order to quantify the performance gains that can be achieved just using software-level reordering schemes. Fig. 5 shows the execution time of the PageRank

Table 1: Input statistics of the graph datasets used in our experiments.

Input graph (label)	No. vertices	No. edges
ego-Facebook(Facebook) [35]	4,038	88,234
musae_Github(GitHub) [36]	37,699	289,003
gemsec-Deezer(Deezer) [37]	41,773	125,826
ego-Twitter(Twitter) [38]	81,305	1,768,149
road_luxembourg-osm (Road map) [39]	114,598	119,667

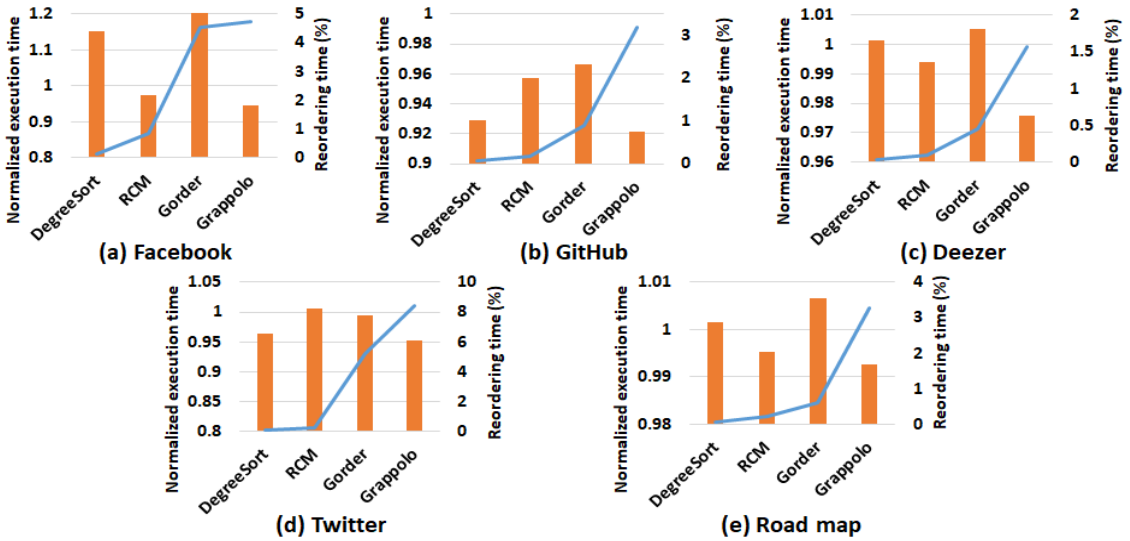

Fig. 5: The orange bars show normalized execution time (reordering time + processing time) of PageRank with different reordering schemes, normalized with respect to the execution time observed using the natural (input) ordering. The blue line curve shows the % fraction of the total time taken by the reordering step.

application with the above-mentioned datasets by incorporating various node ordering schemes. We show both the normalized execution times obtained under each reordering scheme (relative to the natural ordering) and the fraction of time taken by the reordering step (as a measure of the preprocessing overhead). Similar trends were observed with the other two graph applications and hence were omitted due to space considerations. It should be noted that the reordering schemes add a pre-processing cost to the overall execution time. We measure and report this preprocessing overhead in Fig. 5. As can be observed, the reordering costs are higher for Gorder and Grappolo – this is to be expected since these two schemes are heavier in computation cost than degree-based reordering. However, the overall runtime of the PageRank application is the least for the Grappolo-based reordering (but this is not true for Gorder). This suggests that the effort spent in reordering by Grappolo pays off from an application standpoint. As mentioned above, as Grappolo uses community detection by modularity optimization aimed at identifying tightly-knit groups of vertices, it is able to keep related vertices together and thereby it preserves a high degree of locality. Hence, Grappolo inherently enhances the overall performance by reducing cache misses and increasing data reuse. The relatively higher cost in reordering for Grappolo arises due to multiple iterations on the graph before compacting communities to their coarser levels [8] . As the overall processing times obtained under Grappolo were consistently the fastest, we choose Grappolo-based reordering as our default option for the remainder of our evaluation.

4.3 Full System Performance and Energy Evaluation

In this section, we present the performance and energy consumption of the manycore GPU by incorporating the architecture-aware Grappolo-based reordering, on a set of progressively sophisticated manycore GPU NoC architectures (2D mesh, 3D mesh, and 3D SWNoC). Though Grappolo is the best performing vertex reordering scheme, it is still oblivious to the task allocation and the NoC architecture. To implement architecture-aware

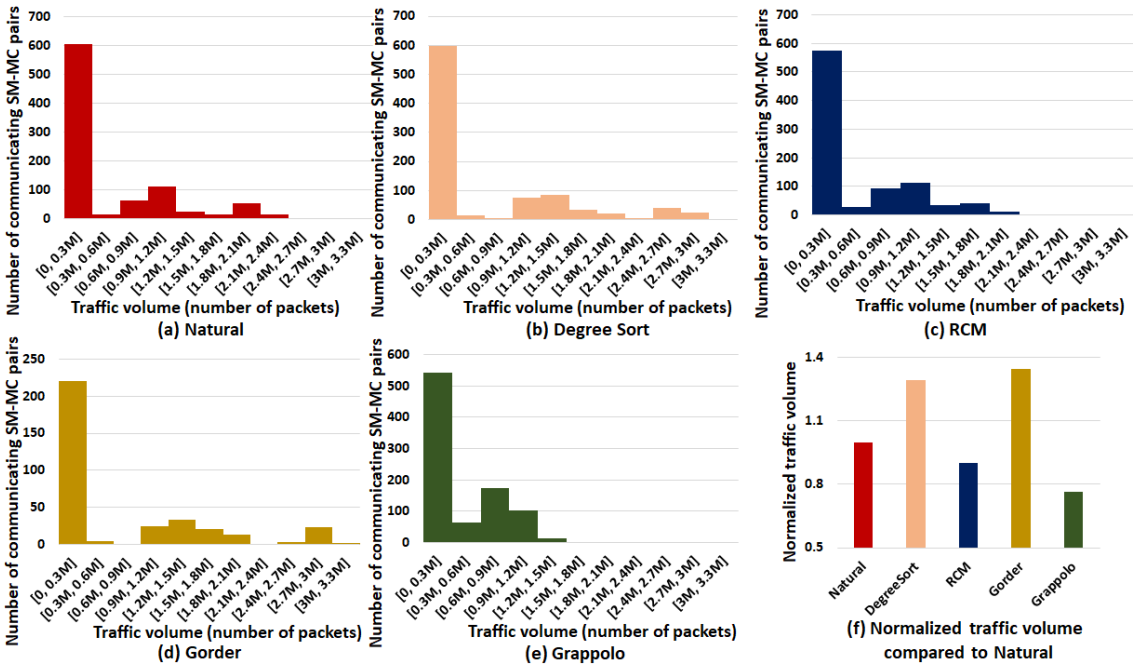

Fig. 6: Distribution of traffic exchanged between different SM-MC pairs (parts a-e) for PageRank with different reordering of Facebook dataset, along with the normalized volume of traffic (part f) compared to Natural.

reordering, the array of reordered vertices is divided into several blocks according to the number of thread blocks and subsequently those blocks are arranged in the increasing order of their *BLoc score* (eqn. (3)). Intuitively, improvement in data locality should reduce the volume of communication (i.e., increasing *BLoc score*). To demonstrate the variation in communication volume with node reordering, in Fig. 6 we show the amount of traffic exchanged between SM-MC pairs for PageRank with the Facebook dataset. Fig. 6(f) also illustrates the normalized volume of traffic for the reordering schemes compared to natural. It is evident that Grappolo does not create any traffic towards the tail part of the distribution. It reduces the communication volume the most among all the ordering schemes (23% less than natural). It is also interesting to see that the schemes degree sort and Gorder do not help reduce the traffic volume (relative to natural) – only RCM and Grappolo achieve a reduction. We do not repeat the communication volume plot with other graph applications and datasets for brevity. We also compare the distribution of *BLoc score* in natural- and Grappolo-ordered graph datasets. Fig. 7 illustrates the distribution of *BLoc scores* for Grappolo and natural order on all the inputs. We can see from Fig.6 that, on average, the values for the *BLoc score* are much higher for Grappolo than for the natural order. Hence, blocks formed due to Grappolo are expected to generate significantly less data movement compared to the blocks in natural order. However, the improvement of *BLoc score* due to Grappolo is data dependent. It is clear from Fig. 7 that the improvement of *BLoc score* for Road map is much lower than the other social media datasets (e.g., Facebook, GitHub, Deezer, Twitter). This is because the Road map dataset's natural ordering already had a good locality to start with.

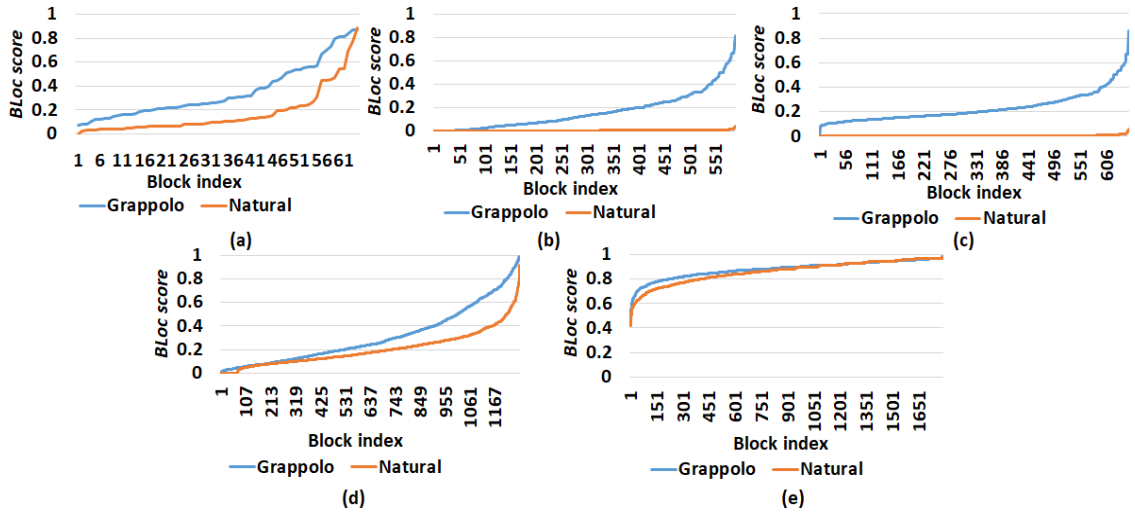

Fig. 7: Distribution of *BLoc score* of different blocks for Grappolo and natural order of (a)Facebook, (b)GitHub, (c)Deezer, (d)Twitter and (e)Road map datasets.

When the graph application is mapped to the manycore system, a reduction in communication is expected to affect the overall performance improvement. This is corroborated in our results shown in Fig. 5. Grappolo is able to achieve 4.9% to 9.6% improvement in execution time for social media datasets, where it is only 0.75% for the Road map. However, it should be noted that the performance improvement achieved by incorporating only the existing vertex reordering schemes is limited. The reordering schemes need to be enhanced with architecture-aware task allocation policy. Hence, while executing the graph applications, task allocation for assigning blocks to SMs follows the proposed priority-based scheme where the blocks with lower *BLoc score* are prioritized to be assigned to SMs with lower *pscore* (eqn. (5)). However, as the *pscore* for each SM depends on the number of hops between SM and MC, a suitable choice of NoC architecture is important for improving the full-system performance.

Next, we consider the full system execution time and energy consumption of the proposed software/hardware co-design framework by incorporating architecture-aware reordering in the manycore platform. For exhaustive performance analysis, we consider five configurations: natural ordering on a 2D mesh (N_2DMesh), Grappolo on 2D mesh (G_2DMesh), Grappolo priority-based task allocation using 2D mesh, 3D mesh, and the proposed 3D SWNoC (G_2DMesh_P, G_3DMesh_P, and G_3DSWNoC_P respectively). Figs. 8 (a), (b) and (c) illustrate the normalized execution time of the proposed software/hardware co-design framework for PageRank, Color and SSSP respectively. It is evident from Fig. 8 that G_3DSWNoC_P achieves the lowest execution time among all the configurations considered here. It achieves 12.6% to 22.9% performance improvement depending on the dataset compared to the N_2DMesh configuration.

Figs. 8 (a), (b), and (c) also show the contributions to the net performance gain from each software and hardware component (reordering, priority-based scheme, and NoC) for the graph applications tested. The contribution of each software and hardware component towards the overall performance improvement varies with the dataset and the graph application. We can see from Fig. 8 that the contribution in performance improvement due to reordering for social network inputs (Facebook, Twitter) varies from 17.3% to 42.1%

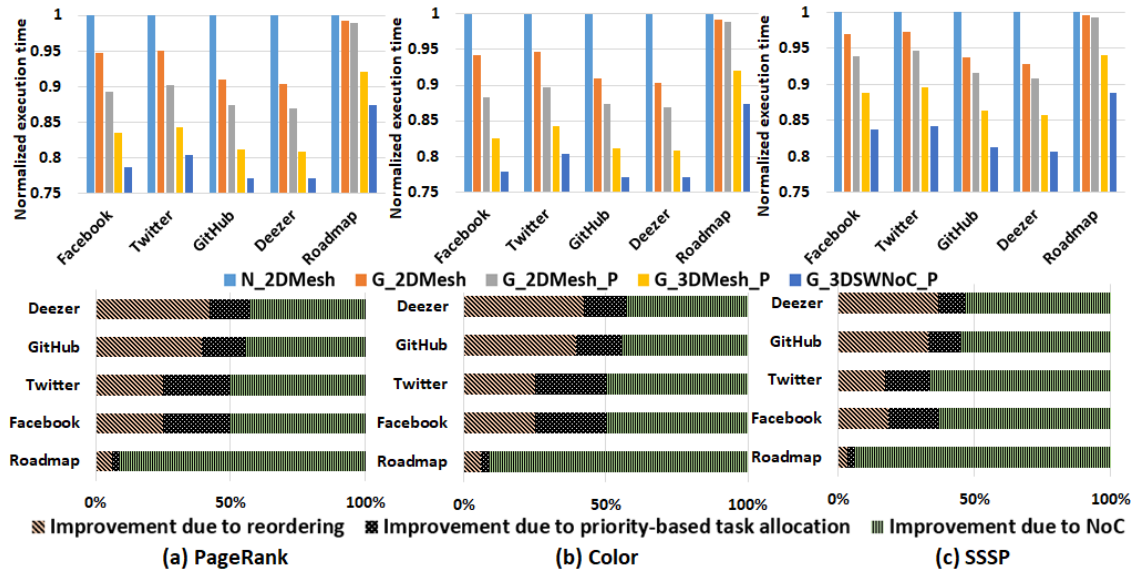

Fig. 8: The top part of the figure shows the normalized execution times of 2D mesh with natural (N_2DMesh), 2D mesh with Grappolo (G_2DMesh), 2D mesh with Grappolo and priority-based task allocation (G_2DMesh_P), 3D mesh with Grappolo and priority-based task allocation (G_3DMesh_P) and 3D SWNoC with Grappolo and priority-based task allocation (G_3DSWNoC_P) compared to N_2DMesh. The bottom part of the figure shows the corresponding %contributions to the improvement achieved by: reordering (left), priority-based task allocation (middle), and NoC (right).

whereas the improvement for Road map is only 3.3% to 5.9%. This happens due to the fact that the improvement of the *BLoc score* from natural to Grappolo is much higher for the social network inputs than for Road map. Along with data dependency, performance improvement due to reordering varies with the graph applications too. In the vertex-centric applications like PageRank and Color, the state of each vertex is updated by its neighbors iteratively until convergence. However, SSSP computes shortest paths from vertices beyond their neighborhood (effectively computing a breadth-first search or BFS sweep from each possible source vertex). This reduces the potential for reuse, compared to iterative operations such as PageRank or coloring where neighborhood could be accessed repeatedly across multiple iterations. Therefore, the locality in graph structures matters more for PageRank and Color than it does for SSSP. Hence, performance improvement due to reordering for PageRank and Color is more than that of SSSP.

Similar to vertex reordering, the contribution of our priority-based task allocation scheme to the full-system performance improvement is also data dependent. Priority-based task allocation prioritizes blocks with higher communication overhead to be assigned to SMs placed near the communicating MCs. Hence, the adopted task allocation scheme is beneficial when the blocks have varying communication overhead (i.e., *BLoc score*). As shown in Fig. 7, the variation of *BLoc score* in social network inputs is much higher than for Road map. Therefore, when blocks with an almost equal value of *BLoc score* in Road map are mapped to the manycore GPU architecture, they are prone to generate almost even amount of memory access requests to MCs. Hence, there is not much difference in performance even when some blocks that do not have high priority are assigned

to SMs near communicating MCs. Hence, we can see from Fig. 8 that task allocation contributes 10.3% to 26.2% to overall performance improvement for social network inputs, whereas the corresponding values for Road map vary from 2.7% to 5% depending on the specific graph application.

3D SWNoC improves performance by reducing the latency associated with on-chip data movement for all the datasets. As all the datasets have a significant amount of long-range traffic, the SWNoC is equally effective for all. Hence, the contribution of 3D SWNoC in total performance improvement is comparatively higher when the reordering scheme and the priority-based task allocation do not contribute significantly to the total execution time reductions. The contribution from NoC in overall performance improvement for Road map (91.4% to 94%) is significantly higher than that of the social network inputs (43% to 66.5%).

Along with the performance evaluation, energy consumption of the overall architecture needs to be analyzed. Figs. 9 (a), (b), and (c) illustrate the full-system energy consumption of the proposed software/hardware co-design framework for PageRank, Color, and SSSP respectively. It is evident from Fig. 9, G_3DSWNoC_P reduces energy consumption from 16.4% to 32.6% depending on the dataset when compared to N_2DMesh. Figs. 9 (a), (b) and (c) also show the contributions from each software and hardware component (reordering, priority-based scheme, and NoC) for PageRank, Color and SSSP respectively in the overall reduction of full-system energy consumption. Similar to the execution time, the contributions of each software and hardware component in reducing energy consumption vary with the dataset. By analyzing the distribution of *BLoc score,* it is evident that reordering and priority-based task allocation for Road map is less effective in reducing on-chip data volume than social media datasets. Therefore, the contribution from reordering and priority-based task allocation to overall energy reduction for Road map (17.64% to 26.8%) is less than that of other social network inputs (43.1% to 64.8%). On the other hand, 3D SWNoC improves energy efficiency by reducing long-range traffic. As all datasets have a significant amount of long-range traffic, the SWNoC is equally effective in all datasets. Hence, as reordering scheme and priority-based task allocation for Road map is not contributing

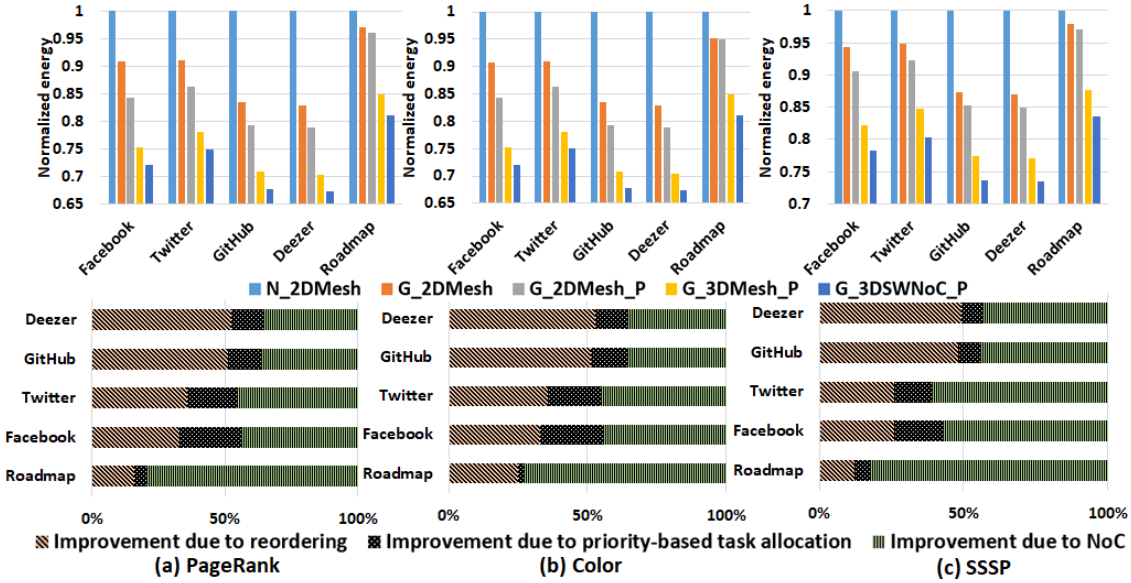

Fig. 9: Normalized energy of 2D mesh with natural (N_2DMesh), 2D mesh with Grappolo (G_2DMesh), 2D mesh with Grappolo and priority-based task allocation (G_2DMesh_P), 3D mesh with Grappolo and priority-based task allocation (G_3DMesh_P) and 3D SWNoC with Grappolo and priority-based task allocation (G_3DSWNoC_P) compared to N_2DMesh. The bottom part of the figure shows the corresponding %contributions to the improvement achieved by: reordering (left), priority-based task allocation (middle), and NoC (right).

significantly to reduce the total energy consumption, contribution of 3D SWNoC is comparatively higher for Road map (73.1% to 82.3%) than that of social network inputs (35.1% to 61%).

5 CONCLUSION

In this work, we present a software/hardware co-design framework for accelerating graph computations on 3D manycore GPU architectures. Each layer in the co-design is designed to deliver specific improvements in performance and energy. At the software level, an architecture-aware vertex reordering-based approach is proposed to improve the locality of graph structures and thereby reduce the on-chip data movement volume. This vertex reordering technique is complemented with a novel priority-based task allocation scheme to prioritize vertices with high communication overhead to be assigned to SMs placed near the communicating MCs for reducing the overall communication latency and energy consumption. As the NoC choice influences task allocation, we propose the design of a 3DSWNoC-enabled manycore GPU architecture, where the placement of the links connecting the SMs and MCs follow a power-law distribution. The proposed 3D SWNoC-enabled software/hardware co-design framework achieves 11.1% to 22.9% performance improvement and 16.4% to 32.6% less energy consumption depending on the dataset and the application when compared to the natural order of graph dataset running on a conventional planner mesh architecture. We have also demonstrated that the contributions from each software and hardware component vary with the datasets. For social network inputs, vertex reordering, priority-based task allocation, and NoC architecture all contribute noticeably to the overall performance and energy improvement. However, for Road Map, the contributions from the reordering and priority-based task allocation are very limited. Most of the performance benefit and reduction in energy consumption in Road map comes from the low latency SWNoC architecture. This dichotomy in the results goes to show that the input characteristics (particularly degree distributions and the initial ordering of vertices in the graphs) could have a pronounced impact on what can be achieved through software-hardware co-design methodologies.

REFERENCES

[1] K. Duraisamy, H. Lu, P. Pande and A. Kalyanaraman. 2016. High-Performance and Energy-Efficient Network-on-Chip Architectures for Graph Analytics. ACM Transactions on Embedded Computing Systems 66.

[2] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. 2016. Gunrock: a high-performance graph processing library on the GPU. In Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16). Association for Computing Machinery, New York, NY, USA, Article 11, 1–12.

[3] A. Kalyanaraman and P. P. Pande. 2019. A Brief Survey of Algorithms, Architectures, and Challenges toward Extreme-scale Graph Analytics. 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), Florence, Italy, pp. 1307-1312.

[4] A. Lenharth, D. Nguyen, and K. Pingali. 2016. Parallel graph analytics. Commun. ACM 59, 5 (May 2016), 78–87.

[5] A. Addisie and V. Bertacco. 2020. Centaur: hybrid processing in on/off-chip memory architecture for graph analytics. In Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference (DAC '20). IEEE Press, Article 158, 1–6.

[6] R. Barik, M. Minutoli, M. Halappanavar, N. R. Tallent and A. Kalyanaraman. 2020. Vertex Reordering for Real-World Graphs and Applications: An Empirical Evaluation. 2020 IEEE International Symposium on Workload Characterization (IISWC), pp. 240-251.

[7] L. Page, S. Brin, R. Motwani, and T. Winograd. 1999. The pagerank citation ranking: Bringing order to the web. Technical Report SIDL-WP-1999- 01204, Stanford Univerisity.

[8] H. Lu, M. Halappanavar, and A. Kalyanaraman. 2015. Parallel heuristics for scalable community detection. Parallel Comput. 47, C (August 2015), 19–37.

[9] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. 2016. Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics. In 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). 1–13.

[10] A. A. Maashri, G. Sun, X. Dong, V. Narayanan and Y. Xie. 2009. 3D GPU architecture using cache stacking: Performance, cost, power and thermal analysis. IEEE International Conference on Computer Design, Lake Tahoe, CA. 254-259.

[11] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk. 2016. Energy Efficient Architecture for Graph Analytics Accelerators. In 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA) (2016-06). 166–177.

[12] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. 2017. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA).

[13] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das. 2016. Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities. In Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16). Association for Computing Machinery, New York, NY, USA, 31–44.

[14] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. 2016. Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics. In 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 1–13.

[15] J. Zhou et al. 2017. TuNao: A High-Performance and Energy-Efficient Reconfigurable Accelerator for Graph Processing. 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID).

[16] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. 2017. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA).

[17] D. Fujiki, S. Mahlke, and R. Das. 2017. In-memory Data Flow Processor. 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT).

[18] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen. 2018. GraphR: Accelerating Graph Processing Using ReRAM. 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA). DOI: https://doi.org/10.1109/hpca.2018.00052.

[19] G. Dai, T. Huang, Y. Wang, H. Yang, and J. Wawrzynek. 2019. GraphSAR. Proceedings of the 24th Asia and South Pacific Design Automation Conference.

[20] L. Zheng et al. 2020. Spara: An Energy-Efficient ReRAM-Based Accelerator for Sparse Graph Analytics Applications. 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS).

[21] J. Petit. 2004. Experiments on the minimum linear arrangement problem. ACM J. Exp. Algorithmics 8, Article 2.3 (2003), 33 pages.

[22] M. R. Garey, D. S. Johnson, and L. Stockmeyer. 1974. Some simplified NP-complete problems. In Proceedings of the sixth annual ACM symposium on Theory of computing (STOC '74). Association for Computing Machinery, New York, NY, USA, 47–63.

[23] Y. Zhang, V. Kiriansky, C. Mendis, M. Zaharia, and S. Amarasinghe. 2016. Optimizing cache performance for graph analytics. arXiv preprintarXiv:1608.01362.

[24] V. Balaji and B. Lucia. 2018. When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs. In 2018 IEEE International Symposium on Workload Characterization, pages 203–214.

[25] U. Kang and C. Faloutsos. 2011. Beyond 'Caveman Communities': Hubs and Spokes for Graph Compression and Mining. IEEE 11th International Conference on Data Mining, Vancouver, BC, Canada, 2011, pp. 300-309.

[26] T. Davis, S. Rajamanickam and W. Sid-Lakhdar. 2016. A survey of direct methods for sparse linear systems. Acta Numerica, 25, 383-566.

[27] E. Cuthill and J. McKee. 1969. Reducing the bandwidth of sparse symmetric matrices. In Proceedings of the 1969 24th national conference (ACM '69). Association for Computing Machinery, New York, NY, USA, 157–172.

[28] A. George. 1973. Nested dissection of a regular finite element mesh. SIAM Journal on Numerical Analysis, 10(2):345–363.

[29] A. George and J. Liu. 1989. The evolution of the minimum degree ordering algorithm. SIAM Review, 31(1):1–19.

[30] H. Wei, J. X. Yu, C. Lu, and X. Lin. 2016. Speedup Graph Processing by Graph Ordering. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16). Association for Computing Machinery, New York, NY, USA, 1813–1828.

[31] G. Karypis and V. Kumar.1999. Kumar, V.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM Journal on Scientific Computing 20(1), 359-392. Siam Journal on Scientific Computing. 20.

[32] I. Safro, D. Ron, and A. Brandt. 2009. Multilevel algorithms for linear ordering problems. ACM J. Exp. Algorithmics 13, Article 4 (2009), 20 pages.

[33] V. Blondel, J. Guillaume, R. Lam-biotte, and E. Lefebvre. 2008. Fast unfolding of commu-nities in large networks. Journal of Statistical Mechanics:Theory and Experiment, 2008(10):P10008.

[34] S. Che, B. M. Beckmann, S. K. Reinhardt and K. Skadron. 2013. Pannotia: Understanding Irregular GPGPU Graph Applications. IEEE International Symposium on Workload Characterization (IISWC), Portland, OR, 185-195.

[35] http://snap.stanford.edu/data/ego-Facebook.html; date accessed: April 2021

[36] http://snap.stanford.edu/data/github-social.html; date accessed: April 2021

[37] http://snap.stanford.edu/data/gemsec-Deezer.html; date accessed: April 2021

[38] http://snap.stanford.edu/data/ego-Twitter.html
[39] http://networkrepository.com/road-luxembourg-osm.php
[40] P. Wettin et al. 2014. Design Space Exploration for Wireless NoCs Incorporating Irregular Network Routing. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 33, no. 11, pp. 1732-1745.
[41] A. Ganguly, K. Chang, S. Deb, P. P. Pande, B. Belzer and C. Teuscher. 2011. Scalable Hybrid Wireless Network-on-Chip Architectures for Multicore Systems. IEEE Transactions on Computers, vol. 60, no. 10, pp. 1485-1502.
[42] S. Deb et al. 2013. Design of an Energy-Efficient CMOS-Compatible NoC Architecture with Millimeter-Wave Wireless Interconnects. IEEE Trans. Comput. 62, 12 (December 2013), 2382–2396.
[43] S. Das, J. R. Doppa, D. H. Kim, P. P. Pande and K. Chakrabarty. 2015. Optimizing 3D NoC design for energy efficiency: A machine learning approach. 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Austin, TX, USA, 2015, pp. 705-712.
[44] S. Das, J. R. Doppa, P. P. Pande and K. Chakrabarty. 2016. Design-Space Exploration and Optimization of an Energy-Efficient and Reliable 3D Small-world Network-on-Chip. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD).
[45] T. Petermann and P. De Los Rios. 2005. Spatial small-world networks: A wiring cost perspective. arXiv: Condmat/0501420v2.
[46] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong and T. M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. 2009 IEEE International Symposium on Performance Analysis of Systems and Software, Boston, MA. 163-174.
[47] N. Jiang et al. 2013. A detailed and flexible cycle-accurate Network-on-Chip simulator. 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Austin, TX. 86-96.
[48] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. 2013. GPUWattch: enabling energy optimizations in GPGPUs. SIGARCH Comput. Archit. News 41, 3 (June 2013), 487-498.