GCiM: A Near-Data Processing Accelerator for Graph Construction

Lei He*†, Cheng Liu*†, Ying Wang*†, Shengwen Liang*†, Huawei Li*†‡ and Xiaowei Li*†

*SKLCA, Institute of Computing Technology, Chinese Academy of Science, Beijing, China
†University of Chinese Academy of Science, Beijing, China
‡Peng Cheng Laboratory, Shenzhen, China
Email:{helei19g, liucheng, wangying2009, liangshengwen, lihuawei, lxw}@ict.ac.cn

***Abstract*—Graph is widely utilized as a key data structure in many applications like social network and recommendation systems. However, real-world graph construction typically involves massive random memory accesses and distance calculation, resulting in considerable processing time and energy consumptions on CPUs and GPUs. In this work, we present GCiM, a specialized processing-in-memory architecture for efficient graph construction and update. By directly deploying the computing units on the logic layer of the 3D stacked memory, GCiM benefits from memory-level parallelism and further improves the memory access efficiency with both optimized processing ordering and data layout. According to our experiments, GCiM shows 634.64X and 53.29X speedup while consuming 1470.7X and 442.56X less energy compared to CPU and GPU respectively.**

I. Introduction

Graph is a widely adopted data structure to interpret the non-linear entities in the fields of machine learning [1] and relational data processing [2], and it is becoming increasingly popular in modern computing systems. When large-scale correlated random data is converted into a graph, many powerful graph learning or processing methods can be applied to the complicated data, such as manifold embedding, semi-supervised graph learning, or relation-based data retrieval [3]. For example, in modern recommendation systems or social networks, the users, items, and their relations are described as large graphs of billion-scale nodes, so that the machine learning or mining can retrieve or index the high-dimensional data of interest more conveniently [4]. Therefore, converting large-scale data into graphs representation, including the vertices, edges, and the according property of high-dimension vector is a mandatory and common step before mining, indexing, or search operations on the large-scale data such as images and videos. The most popular graph construction framework is k-nearest neighbor (kNN) graph, where each node is connected to its k nearest neighbors. After applying such constructor, the similarities and correlations between objects can be kept, which enables highly-efficient accesses and retrieval operations upon the graph data entries.

However, billion-scale kNN graph construction and update is a non-trivial task, and poses a severe challenge to the computation strength, bandwidth, and power of state-of-the-art computing systems. Particularly, for the applications involving dynamic graphs that must be repetitively updated and reconstructed [5], how to perform energy and memory efficient graph construction is essential to the system efficiency in warehouse computers, which must be optimized in power and performance for low total cost of ownership (TCO).

Plenty of works from both software and hardware areas have been conducted to optimize the construction of kNN graphs, such as NN-descent [6], EFANNA [7] on CPU, Faiss on GPU [8], and specialized hardware architecture like Tigris [9] and QuickNN [10]. The large-scale graph construction and on-line update over billion-scale entities like users in social network and items in on-line C2C platforms [11] typically involve numerous random memory accesses, resulting in serious performance and power penalty to the system. Even though GPU solutions provide adequate computation parallelism [8], the irregular data accesses to large-scale and complicated graph data in memory still cause severe resource under-utilization in GPUs, aggravating the energy efficiency of power-consuming GPUs. Meanwhile, existing hardware designs for graph processing and graph learning mainly focused on tree search [12] and graph search [13], or specific applications such as 3D points [9] [10], which can be employed to construct small graphs of particular types, but they are also subject to the issue of memory bandwidth insufficiency and latency penalty encountered by large-scale graph construction.

In general, hardware solutions to graph construction for real-world applications suffer from three fold issues: 1) Graph construction essentially involves massive random memory accesses for the step of similarity establishment, thus requiring high bandwidth provision. 2) Prior methods do not fully exploit the data reusability in high-degree graph vertices, causing dramatic bandwidth waste in memory accesses. 3) Current graph construction solutions are designed to organize the whole dataset into a graph, but such strategy in dynamic graphs [5], where most points have no interaction with the newly inserted ones, will become extremely inefficient due to many redundant operations.

To cope with these issues, we proposed GCiM, a specialized processing-in-memory architecture to better accelerate the large-scale kNN-based graph construction and update in real-world applications. Firstly, we architect and design the GCiM accelerator directly on the logic layer of 3D stacked DRAM memory modules similar to Hybrid Memory Cube (HMC) to fully exploit both vault and bank level parallelism for better

978-1-6654-3274-0/21/$31.00 ©2021 IEEE

bandwidth utilization. Secondly, we propose a leaf-granularity access strategy in graph manipulation to increase the efficiency of data reuse and reduce data transmission. In addition, we design a dedicated data layout to balance the processing of IO intensive hardware modules for less random memory accesses. Finally, we utilize a tree-search module to insert the new points into the intentionally condensed sub-graphs, so that dynamic graph construction induces lower cost and maintains the quality of updated graph simultaneously.

In summary, we make the following contributions:

1) We proposed an in-memory graph construction architecture for kNN algorithm so that the massive irregular data features can be converted into weighted graphs in the memory without being frequently transmitted between the processor chips and memory modules.
2) The proposed architecture, GCiM, fully utilizes memory-level parallelism inside memory cubes and adopts a leaf granularity strategy with optimized data layout to reduce data transmission and improve the internal bandwidth utilization. Moreover, GCiM is the first hardware accelerator optimized to provide high throughput and low latency for online graph update.
3) We implemented GCiM with a cycle-accurate simulator, and evaluated the performance and energy-efficiency benefits brought to the graph processing and learning systems. The experiment shows that GCiM can speed up the graph construction operation by 634.64X and 53.29X, reduce the energy consumption by 1470.7X and 442.56X on average, when compared to the CPU and GPU baselines respectively.

II. Background and Motivation

A. Related Work on Graph Construction

Graph construction plays an important role in organizing large relational data as graphs for efficient machine learning and mining in many applications such as image retrieval and recommendation [1]- [4]. However, the time complexity of building a kNN graph is $O(n^2d)$ where n refers to the number of nodes and d represents the dimension of the node property. Thereby, it is time-consuming especially for large datasets and numerous efforts [6] [7] have been devoted to reduce the time complexity. Tree-based algorithms that split the large datasets into small sub datasets with a tree structure essentially divide the large global kNN calculation into smaller local kNN calculation. This approach greatly reduces the amount of the computation and the data movement, and it has become one of the main-stream solutions to large graph construction. Nevertheless, kNN on the sub datasets still requires massive irregular memory accesses and data movement as the nodes in the same sub datasets may not be located in continuous memory space, thus causing severe resource under-utilization and power consumption on general processors.

For higher performance, customized graph construction accelerators emerge recently. Tigris [9] and QuickNN [10] tailor typical graph construction accelerators for 3D perception-enabled applications using tree-based algorithms. QuickNN has the kNN calculation conducted sequentially across the dataset. However, there is little data reuse among the kNN calculation of the neighboring points because the kNN is performed within a leaf node and the neighboring points may likely be located in different leaf nodes. As a result, the same points may have to be repeatedly loaded and massive irregular memory accesses are required. Tigris has the tree building process performed on CPU, which requires additional data movement, thus aggravating the memory bandwidth bottleneck. Moreover, they target at graph construction on static data and fail to support the dynamic graph update. Thus, more efficient graph construction accelerator allowing dynamic update remains highly demanded.

Table I: Efficiency of EFFANNA.

	ST10K	ST1M	GT1M	ST100M
Arithmetic intensity (FLOPs/Byte)	0.0747	0.0674	0.0799	0.0364
L3 cache miss ratio	0.5441	0.9336	0.969	0.9551

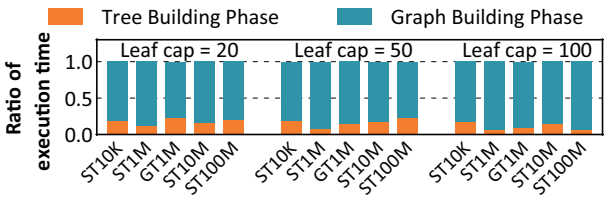

Fig. 1: Execution time breakdown of graph construction.

B. Characterization of the Graph Construction

To investigate the characteristics of the graph construction, we analyze the execution time of graph construction utilized in EFANNA [7] over a series of datasets. According to the execution time breakdown shown in Fig. 1, we observe that the *graph building phase* takes up the majority of the execution time on all the datasets. In addition, the time distribution of the graph construction is also affected by the leaf node sizes. The proportion of the graph building time raises from 82.44% to 89.46% on average when the leaf node size changes from 20 to 100 for higher graph construction quality.

We leverage the L3 cache miss ratio and arithmetic intensity (FLOPs/Byte) to further analyze the computing characterization of the *graph building phase*. As shown in Table I, the high cache miss ratio indicates that it operates on a large dataset with considerable random memory accesses. The arithmetic intensity is 0.058 FLOPs/Byte on average meaning that a single FLOP needs more than 10 Bytes memory accesses. The memory bandwidth requirement goes up to 320GB/s on a processor operating at 16 FLOPs/cycle with 2GHz clock, which makes it rather challenging to meet the computation requirements given large amount of irregular memory accesses. Thereby, we can conclude that graph construction is memory bound and still needs intensive optimizations.

In addition, we notice that many practical applications such as e-commerce and social networks have continuous incoming entities. For instance, Facebook's social network graph reports 86400 objects/second update in 2013 and Twitter has 143 thousand tweets per second to be updated [5]. These entities need to be updated at run-time to the constructed graph to ensure more accurate prediction. Thus, dynamic graph update is indispensable and common for graph construction,

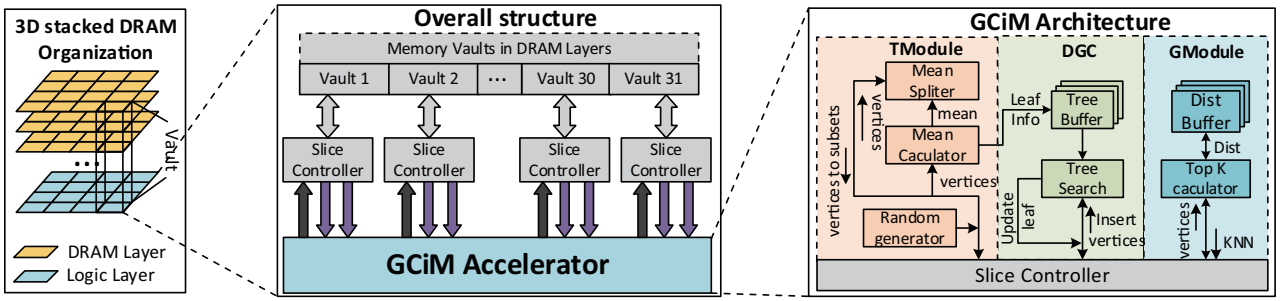

Fig. 2: GCiM hardware architecture.

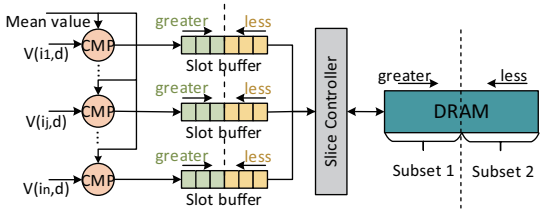

Fig. 3: Mean Spliter.

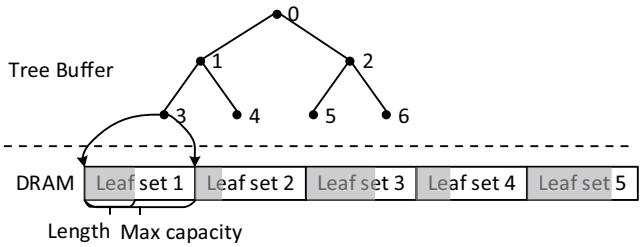

Fig. 4: Tree Buffer.

and it also needs to be addressed in the graph construction accelerator.

III. GCiM Architecture

A. GCiM Overview

In order to accelerate the memory-bound graph construction, we take advantage of the near-data processing technique to reduce the data movement across the memory hierarchy, and build a specialized accelerator GCiM to fully utilize high-bandwidth 3D DRAM. The overview of GCiM is shown in Fig. 2. It is built on a logic layer over a die-stacked memory architecture composed of multiple DRAM layers. The stacked DRAM dies are partitioned into a series of vertical vaults and there are 32 vaults in our memory module. Each vault operates at 10GB/s and the entire DRAM has a total bandwidth of 320GB/s. To fully exploit the vault-level parallelism, GCiM also needs to be highly parallelized and accesses the different vaults with a series of slice controllers.

For the purpose of general graph construction acceleration including online graph update, GCiM divides the whole process into three phases: *tree building phase*, *graph building phase*, and *graph update phase* correspondingly. The *tree building phase* is essentially to build a tree structure that can characterize the general spatial locality of the points in the dataset. First of all, it randomly selects a dimension yielded by a random generator and calculates the mean over the dataset in this dimension with a mean calculator. Then it separates the points into two subsets with a Mean spliter according to the result of the Mean calculator. By recursively dividing the points in the dataset, the dataset can be split into multiple partitions and organized as a tree structure where points in the same leaf node are more likely to be close to each other. The generated tree structure will be stored in the tree buffer, which facilitates the search of neighbors of any given point during the *graph update phase*.

Graph building phase is triggered right after the tree building phase with which the tree structure is determined. With the tree structure, k nearest neighbors in a leaf node can also be considered as k nearest neighbors of the entire dataset, so we can conduct the kNN algorithm with the granularity of a leaf node in the tree structure instead of the entire large dataset. In addition, the distance calculation between points in the same leaf node can also be reused for k nearest neighbor search for all the different points in the same leaf node. Thereby, we have a distance buffer to record the calculated distance between points to avoid repeated distance computation. When the distances are calculated, we have a top K sorter to decide the k nearest neighbors.

The last stage is *graph update phase*, which has the newly incoming points updated to the graph. The first step is to distribute the incoming points to the leaf nodes of the tree structure, which is essentially a series of comparison. This step is done in the tree search module. After the tree search, *graph building phase* is invoked for only the updated leaf nodes which may affect the structure of the resulting graph. Particularly for the top K calculation, distances between the original points are stored in the Dist Buffer and they can be reused for the new Top K processing. The graph update eventually depends on the change of the top K processing results.

B. GCiM Micro-architecture

1) Tree Building Module(TModule): As the Mean Calculation is performed on the according point partition instead of the entire dataset, the Mean Spliter can only start after the Mean Calculator in the same layer of the tree structure. The dependency dramatically hinders the pipeline processing. Nevertheless, the point partitions obtained from the Mean Spliter can be immediately utilized to retrieve the corresponding features and conduct the Mean Calculation of the next layer in a pipelined manner with random dimension generated beforehand, which also avoids repeatedly loading the point partitions. As for the Mean Calculator, it essentially retrieves the specified feature dimensions of the corresponding point

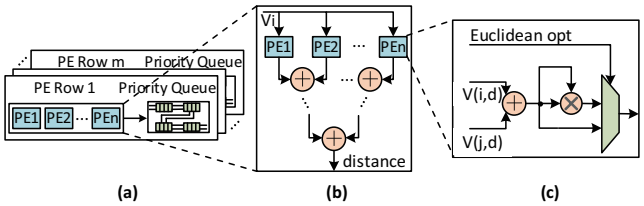

Fig. 5: Top K calculator details. (a)Top K calculator. (b)PER design. (c)single PE structure.

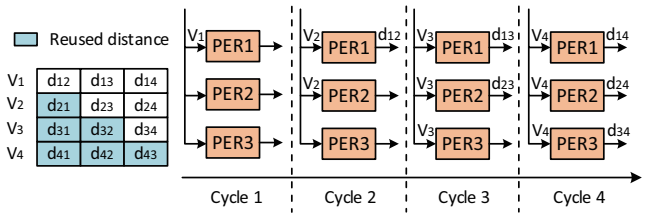

Fig. 6: Example of Top K Calculator dataflow.

partition streamed from the Mean Spliter and conducts the aggregation with an adder tree to match the feature retrieval throughput. For the Mean Spliter, it loads a specified feature dimension of a point partition and splits them into two subsets based on the mean value calculated in the previous stage. The outputs will be streamed back to the external memory. To ensure higher bandwidth utilization, the indices are gathered in buffers and written back in batches as shown in Fig. 3.

Internal nodes of the tree mainly include the child node indices and the partition information. The tree structure is relatively small, so it can be fully accommodated in the on-chip buffer i.e. Tree Buffer shown in Fig. 4. The leaf node contains the actual point indices, so its size far exceeds the on-chip memory and it is usually stored in DRAM. To ensure parallel accesses without conflicts during the *tree building phase*, each leaf node is reserved with a fixed amount of memory space. There will be multiple point partitions in a layer of the tree structure when the processing moves on. While the point partitions in the same layer of the tree are independent, they can be processed in parallel and 32 processing pipelines are implemented for both the inter and intra partitions for both higher processing throughput and memory bandwidth utilization as shown in Fig. 3.

2) Graph Building Module(GModule): The core of *GModule* is a Top K calculator. It conducts the top K calculation in the granularity of a leaf node. The structure of the Top K calculator is shown in Fig. 5(a). It consists of a 2D array of processing elements (PEs) for the distance calculation and a priority queue in each row of PEs utilized for the top K minimum distance searching as proposed in [14]. In addition, each row of the PEs is equipped with an adder tree to enable pipelined Euclidean distance calculation of two points according to Fig. 5(b). Since the feature dimension of a point is usually larger than the number of PEs in a row (PER), the PEs in PER can maintain high utilization when performing the distance calculation of a pair of points. On top of the PER, we further exploit the feature reuse among the different PERs and illustrate the data flow with an example as shown in Fig. 6. Suppose there are four points in a leaf node and the distance calculation of the points can be mapped to three PERs. In cycle 1, V_1 is loaded to PER_1 and cached in PEs. In cycle 2, V_2 is broadcast to PER_1 and PER_2, cached in PER_2. Meanwhile, d_{12} is calculated. In cycle 3, V_3 is broadcast to the three PERs, and cached in PER_3, then we can obtain d_{13} and d_{23}. In cycle 4, we broadcast V_4 to all these PERs and obtain d_{14}, d_{24}, and d_{34} accordingly. At the same time, the output distance in each cycle will be streamed to the following Priority Queue and Dist Buffer for further reuse. Given more distance calculation, the PERs can be fully utilized. While the number of PERs is usually smaller than the number of points in a leaf node, the nodes in a leaf node need to be further partitioned based on the row of PERs to fit the computing array and the point partitions will be processed sequentially. As we need to obtain the Top K of a leaf node instead of a leaf node partition, the Priority Queue will be reused across the processing of the different partitions and flushed only when a new leaf node is processed.

3) Dynamic Graph Construction (DGC): Dynamic graph construction that has the newly incoming points updated to the built graph is an important functionality of GCiM. The new points are streamed to the *Tree Search Module* to go through a depth-first search (DFS) of the tree such that the new points can be distributed to the corresponding leaf nodes of the tree structure by comparing to the mean value in each node of the tree. For the DFS, only one comparison is required in each layer of the tree and the tree structure is stored in on-chip buffer i.e. *Tree Buffer*, so the tree search is fast. When the new nodes are updated, we will invoke the Top K calculation on only the updated leaf nodes. Since the distances between nodes in the original leaf nodes are already calculated and stored in memory, they can be reused directly and we only calculate distance relevant to the new points. Essentially, DGC reuses the major modules in *GModule*.

C. Data Layout Optimization

The data layout greatly affects the memory access pattern of GCiM and has substantial influence on the performance and energy consumption of the memory-bound graph construction on GCiM. Thus, we attempt to optimize the data layout especially the features of the datasets for efficient graph construction in this subsection. The majority of the memory accesses are from the Mean Calculator and the Top K Calculator of the accelerator according to previous discussion. The Mean Calculator needs to read value of a specific feature dimension based on the indices of the points located in a leaf node of the tree structure. Hereby, it prefers to have the point features stored with dimension-major layout (DML) and then split into different vaults to enable parallel accesses. Different from the Mean Calculator, the Top K calculator is mainly determined by the large number of the distance calculations and needs the entire features of the points. Thereby, vertex-major layout (VML) that has the entire point feature stored sequentially in the same vault and different point features distributed across vaults is more efficient. To achieve higher overall performance of the graph construction, we have a

Table II: ANN Datasets.

Dataset	ST10K	ST1M	GT1M	ST10M	ST100M
Vectors	10,000	1,000,000	1,000,000	10,000,000	100,000,000
Dimension	128	128	960	128	128

mixed data layout to balance the processing of the two I/O-intensive hardware modules. The basic idea is to divide both the feature and points into partitions and distribute them into the 3D stacked memory vaults. The exact partition strategy depends on the overall memory access efficiency which can be approximated with the amount of memory accesses of the two hardware modules and the corresponding memory access efficiency under different memory access granularity. Note that memory access efficiency can be obtained with offline testing.

IV. Evaluation

A. Experimental Setup

We built a cycle-accurate simulator to measure the performance of GCiM accelerator and a RTL design synthesized with Synopsys Design Compiler (DC) in TSMC 14nm process technology. The accelerator is implemented on the logic layer stacked on a 3D memory module and works at 400 MHz. The DRAM module is divided into 32 vaults with a total internal bandwidth of 320GB/s. Finally, the power of the 3D memory is estimated with the simulator proposed in [15].

Due to the lack of general graph construction accelerator, we compared GCiM with state-of-art implementations including EFANNA [7] and Faiss [8] on CPU and GPU respectively. EFANNA utilizes the same tree-based algorithm while Faiss adopts k-means for similarity establishment. CPU platform is equipped with Intel Xeon Gold 5217@3.0GHz processor and 256GB DDR4. GPU platform is equipped with NVIDIA Telsa V100 SXM2 and 32 GB HBM2. The datasets utilized are listed in Table II and their sizes range from 10K to 100M [16].

B. Experimental Results

1) Power & Area: We have GCiM configured with 32 processing pipelines in *TModule*. *GModule* is equipped with 64 PERs each of which has 32 PEs. 32bit fixed point MAC is utilized in each PE. Each Slot buffer in the Mean Spliter contains at most 512 points and the total capacity of the slot buffers is 128KB. Tree Buffer is determined by the number of nodes in the tree structure and it is set to be 128KB to meet the requirement of the largest dataset utilized in the experiments. Dist Buffer is mainly utilized as a write buffer to enable batch write to the memory and it is set to be 4KB. In summary, the total amount of on-chip buffer is 260KB. The data path of *TModule* and *GModule* with a large processing array dominates the chip area. Total chip area is $1.928mm^2$ and the power consumption is 1.527W in TSMC $14\,\mathrm{nm}$ technology.

2) Performance: We compare the performance of GCiM with the baselines including EFANNA and Faiss in CPU and GPU respectively. The number of points in each centroid for Faiss and leaf size for EFANNA and GCiM is 1000. K is set to be 10. For EFANNA, it is accelerated with both AVX instructions and multi-thread optimizations. The performance comparison is shown in Fig. 7(a). It reveals that

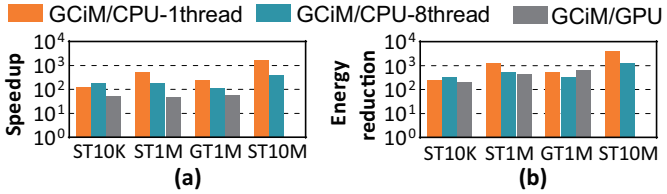

Fig. 7: (a) Performance speedup over CPU and GPU. (b) Energy reduction over CPU and GPU.

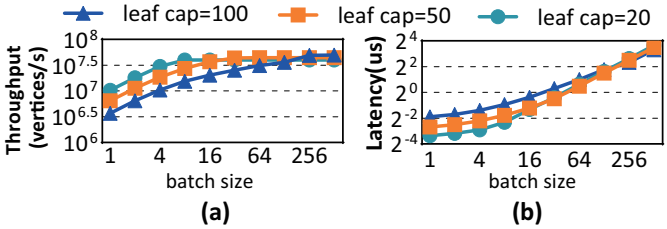

Fig. 8: (a) Throughput in DGC. (b) Graph update latency.

the average performance speedups of GCiM are 634.64X, 215.02X, and 53.29X over CPU-1thread, CPU-8threads, and GPU implementations respectively. In addition, GCiM shows higher performance speedup on the largest dataset i.e. ST10M. In contrast, Faiss fails due to the out of memory error on ST10M in GPU. EFANNA with multi-thread optimization prefers larger datasets and the performance even drops on smaller datasets because of the muti-thread synchronization overhead, Nevertheless, EFANNA with 8 threads achieves only 2.95X performance speedup compared to the single thread implementation, which shows limited scalability. Unlike EFANNA and Faiss, GCiM is less sensitive to the datasets and the scalability of GCiM will be further investigated in the following subsection.

3) Energy Consumption: The power consumption of CPU and GPU is obtained from Likwid and NVPROF respectively, and the power consumption of GCiM is estimated using Synopsys DC. The energy consumption comparison is shown in Fig. 7(b). GCiM achieves about 1470.7X, 619.72X and 442.56X less energy consumption compared to CPU-1thread, CPU-8thread and GPU. On top of the execution advantage, the great energy consumption reduction is mainly attributed to the unified customized GCiM accelerator with much less computation and memory access redundancy that benefits from the computing order optimization and data layout optimization. These optimizations will be evaluated in the following subsection.

4) Dynamic Graph Construction: As for the dynamic graph construction, it concerns both the graph update latency and graph update throughput. The update latency shows how fast a new data can be updated to the graph while the update throughput exhibits how efficient new nodes can be updated to the graph. The two metrics can be contradictory and the requirements of the two metrics may vary under different scenarios. They can be adjusted with update batching as shown in the Fig. 8, where we randomly inserted 1K points into the ST10K dataset. When a set of points are batched for the update, the inspection of the new points with the tree structure can be pipelined. Meanwhile, with the batch processing, multiple new points may be located in the same leaf node

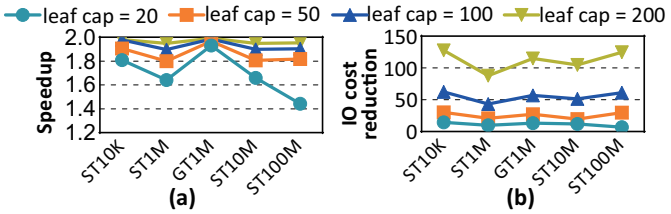

Fig. 9: Comparison between ISO and LGO over performance and io reduction.

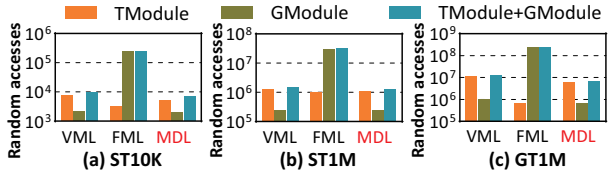

Fig. 10: # of random memory accesses issued from the major modules of GCiM under different data layouts and datasets.

of the tree. In this circumstance, the feature of old points can be reused to perform top-K calculation of the corresponding leaf node especially the distance calculation and thus acquire high computation unit utilization. The update throughput can be improved accordingly and gets saturated when PERs are fully utilized as shown Fig. 8(a). In contrast, the graph update latency increases with the batch size but much more smoothly before the computing resources especially the PERs are used up at batch 64. In addition, we observe that the leaf node size also affects the graph update latency and throughput. Generally, smaller leaf node size requires less distance calculation and thus exhibits lower latency and higher throughput.

C. Optimizations

1) Top K processing order comparison: An intuitive approach of the top K calculation is to process the points in index sequential order (ISO). In this work, we propose to conduct the top K calculation in leaf granularity order (LGO) and the points in the same leaf node will be processed together for the sake of better data reuse. The two different processing approaches are compared in Fig. 9. It can be observed that LGO achieves averagely 1.86X performance speedup compared to ISO. The main reason is that ISO needs to load the same feature of a point multiple times when the point is a potential neighbor of different nodes with indices far from each other. In contrast, LGO has the points in the same leaf node processed in parallel. As these points are potential neighbors and are required by each other for the distance calculation, the processing has much better data locality and less memory accesses accordingly as presented in Fig. 9. In addition, we observe that the size of the leaf node also greatly affects the reduction of the memory accesses and larger leaf nodes offer more optimization space in general.

2) Data layout optimization: In this experiment, we compare the proposed mix data layout (MDL) with other two traditional layouts i.e. vertex-major layout (VML) and dimension-major layout (DML) as described in Sec III-C. When the data in the memory are sequentially accessed, the memory bandwidth is fully utilized despite the data layouts. In contrast, the number of random memory accesses is the direct reason that leads to the different memory bandwidth utilization or memory access efficiency. Hereby, it can be used as the metric to compare the efficiency of the three graph layouts. As shown in Fig. 10, VML is mainly favored by the *GModule* that needs to read the entire point feature for distance calculation while DML is preferred by *TModule* that frequently read a specific dimension of the features for the mean calculation. Nevertheless, they may cause considerable random memory accesses on the other module of GCiM. Different from the two baseline layouts, GCiM has a hybrid data layout to balance the preference of both modules with total 28.5% and 96.7% random memory accesses reduction compared to VML and DML respectively.

V. Conclusion

In this paper, we proposed an in-memory graph construction architecture GCiM for the first time, which can be scaled to large real-world applications. The architecture fully exploits the in-memory parallelism of a 3D stacked memory and investigates the computation reuse and data locality with data layout optimizations. In addition, it also supports quick online graph construction for evolving graphs. Experiment shows that GCiM achieves 634.64X and 53.29X performance speedup and 1470.7X and 442.56X energy efficiency improvement on average compared to CPU and GPU respectively.

Acknowledgement

This paper is supported in part by the National Key Research and Development Program of China under grant 2020YFB1600201, and the National Natural Science Foundation of China (NSFC) under grant No.(62090024, 61876173, 61902375). Lei He and Cheng Liu made equal contributions. The corresponding authors are Ying Wang and Huawei Li.

References

[1] W. Liu *et al.*, "Large graph construction for scalable semi-supervised learning," in *ICML*, 2010.
[2] Kemelmacher-Shlizerman *et al.*, "Exploring photobios," *TOG*, 2011.
[3] S. Arya *et al.*, "Approximate nearest neighbor queries in fixed dimensions," in *SODA*, 1993.
[4] C. Fu *et al.*, "Fast approximate nearest neighbor search with the navigating spreading-out graph," *arXiv preprint arXiv:1707.00143*, 2017.
[5] D. Sengupta *et al.*, "Graphin: An online high performance incremental graph processing framework," in *Eur-Par*. Springer, 2016.
[6] W. Dong *et al.*, "Efficient k-nearest neighbor graph construction for generic similarity measures," in *WWW*, 2011.
[7] C. Fu *et al.*, "Efanna: An extremely fast approximate nearest neighbor search algorithm based on knn graph," *arXiv:1609.07228*, 2016.
[8] J. Johnson *et al.*, "Billion-scale similarity search with gpus," *arXiv preprint arXiv:1702.08734*, 2017.
[9] T. Xu *et al.*, "Tigris: Architecture and algorithms for 3d perception in point clouds," in *IEEE Micro*, 2019.
[10] R. Pinkham *et al.*, "Quicknn: Memory and performance optimization of kd tree based nearest neighbor search for 3d point clouds," in *HPCA*. IEEE, 2020.
[11] A. Ching *et al.*, "One trillion edges: Graph processing at facebook-scale," *PVLDB*, 2015.
[12] S. H *et al.*, "A hardware processing unit for point sets," in *HPG*, 2008.
[13] V. Lee *et al.*, "Application-driven near-data processing for similarity search," *arXiv preprint arXiv:1606.03742*, 2016.
[14] S. Moon *et al.*, "Scalable hardware priority queue architectures for high-speed packet switches," *TC*, 2000.
[15] F. Schuiki *et al.*, "A scalable near-memory architecture for training deep neural networks on large in-memory datasets," *TC*, 2018.
[16] A. Laurent *et al.*, http://corpus-texmex.irisa.fr/.