Graph Placement Optimization on a Heterogeneous Memory System

Qinzhe Wu
Electrical and Computer Engineering
University of Texas at Austin
Austin, United States
qw2699@utexas.edu

Snehil Verma
Electrical and Computer Engineering
University of Texas at Austin
Austin, United States
snehilv@utexas.edu

Alexander M. Taft
Electrical and Computer Engineering
University of Texas at Austin
Austin, United States
alexmtaft@utexas.edu

***Abstract*—Graph analytics continue to be an important and emerging field, and because their execution is data driven, numerous memory accesses are required. In an effort to ameliorate the growing divide between core processing power and memory bandwidth delivered, heterogeneous memory systems have begun to enter the stage of contemporary processor design. Heterogeneous memory can offer significantly higher bandwidth over conventional DRAM (∼5x on Intel's Knights Landing architecture using Multi-Channel Dynamic Random Access Memory (MCDRAM)). The benefits however can be stymied for graph processing applications as they are sensitive to memory layout change and small run-time adjustments can lead to a chain of updates, which incur additional run-time overhead and energy consumption.**

We propose a novel optimization technique that statically makes fine-grain placement decisions about which type of memory each vertex of a graph should reside in. This technique relies on the intuition that the natural properties of a graph (number of incoming/outgoing edges, topology, frontier composition) can be analyzed to make a decision about which vertices would most benefit from placement in MCDRAM. We modify a light-weight shared memory graph processing framework (Ligra) to use our optimization technique, evaluating the technique and showing it can be easily adopted. After exploring various heuristic strategies, our placement optimization shows increased performance can be achieved through partitioning, and at least in one case a 2x speedup can be obtained.

***Index Terms*—Graph Analytics, Heterogeneous Memory, Static Placement Optimization**

I. Introduction

To mitigate the growing disparity between delivered versus desired memory bandwidth plagued by conventional systems using DRAM, contemporary processors have begun to include on-package High-Bandwidth Memory (HBM). HBM can make up for starved bandwidth in data driven applications by acting as either a cache for off-chip memory, or as a separate NUMA node of flat addressable memory.

At the same time as HBM has become more readily available, graph applications have continued to grow in both size and popularity. Graph applications are inherently memory driven (rather than arithmetic) and may benefit from increased bandwidth of system memory - which contemporary processors employing HBM possess.

While dynamic partitioning algorithms have been proposed [1], [2], they require online profiling which adds overhead and complexity, and in some cases additional hardware to use it. We seek to investigate the natural characteristics of graphs (topology, incoming/outgoing edges, etc.) to make an offline and static partitioning of its vertices to achieve higher performance in a heterogeneous memory system.

The main contributions of this work are summarized as follows:

- We model the roofline for a heterogeneous memory system and show that there exists additional exploitable bandwidth.
- We propose a static graph placement optimization technique and implement it on top of a state-of-the-art graph processing framework.
- We explore and develop several different placement heuristics, of which some demonstrate performance improvement.

The rest of this paper is organized as follows: First, we motivate the study with a sanity check and prior work (§ II). We then propose a heuristic-based static graph placement optimization technique and illustrate it with several example decision-making strategies (§ III). The proposed optimization is then evaluated by running three common graph applications on four representative graphs, along with discussion of the results (§ IV). Finally, we conclude with future prospects and current limitations (§ V), and wrap-up our observations (§ VI).

II. Motivation and Related Work

As motivation for an optimization strategy, we first perform a sanity check to ensure large bandwidth differences on a real heterogeneous system can be observed. We then motivate our work from two bodies of research that explore heterogeneous memory systems for exploiting bandwidth, Dynamic Access Partitioning (DAP) [1] and ProfDP [2], and then state our intended users.

A. *Sanity Check*

To substantiate the need for a placement strategy we first conduct an empirical study using the Roofline Tool [3] developed by Berkley CS lab. We run the model at the Texas Advanced Computing Center (TACC) using Intel's Knights Landing (KNL) equipped with on-package MCDRAM and off-chip DRAM, which is the heterogeneous memory system

for our experimental setup. Since each type of memory is abstracted to a different NUMA node, we completely bind the roofline tool to each memory using the available $numactl$ command. Figure 1 exhibits the expected bandwidth differences between the two types of memory (84.0GB/s for DRAM and 396.1GB/s for MCDRAM). Hence, there is potential in exploiting the interplay between the two types of memory.

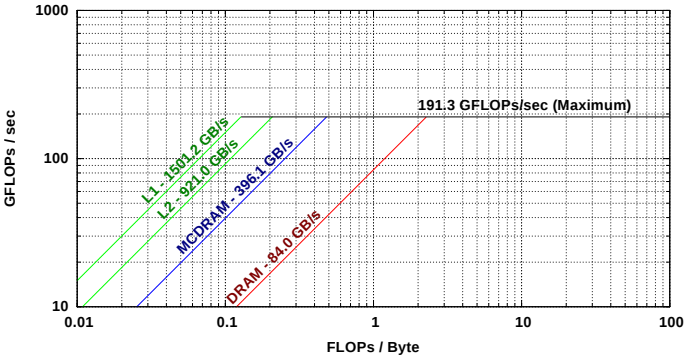


Fig. 1: Empirical roofline model for KNL flat-quadrant mode

B. *Dynamic Access Partitioning (DAP)*

Dynamic Access Partitioning [1] challenges traditional wisdom that higher hit rates in a memory side cache result in improved system performance. The authors state that hit rate in the memory side cache, beyond a certain point, can actually degrade system performance. This is said to be due to the overall delivered bandwidth becoming limited by that of HBM, which leaves underutilized bandwidth from DRAM on the table. By sacrificing memory side cache hit rate, multiple bandwidth sources can be utilized, leading to a greater aggregate bandwidth than just the HBM side cache alone. DAP uses an online profiler to identify the data that is most bandwidth critical, and explicitly sends the remaining to memory. They mention that their learning mechanism requires 16B of additional hardware.

Using the insight from DAP, we are motivated to research how multiple bandwidth sources could be utilized in a way other than just a memory side cache, namely, if offline partition decisions can be made for a graph to take advantage of multiple bandwidth sources. Explicitly this means using KNL in $flat\text{-}quadrant$ mode, rather than $cache$ mode.

C. *ProfDP*

Our second source of motivation is derived from a data placement guide in heterogeneous memory systems known as ProfDP [2]. In their work they assume that optimal data placement is achieved when all data is placed in fast memory (having the lowest latency or highest bandwidth). However, due to the challenges associated with placing all data in fast memory (capacity constraints or allocation capability), this kind of placement can not always be achieved. Even when there are no constraints on capacity or allocation capability, it may be better to have only some part of the data in fast memory to achieve better performance. ProfDP presents a light-weight offline profiler which is stated to achieve near-optimal data placement in a heterogeneous memory system. They achieve near-optimal performance (optimal being all data in fast memory) by making data placement decisions determined by a few concrete and abstract features of a data object - size, importance, latency sensitivity, bandwidth sensitivity.

We used the object features they highlight as motivation into exploring how the inherent qualities of a graph could be used in data placement decisions of either HBM or DRAM. We also challenge their statement that placing all data in fast memory is the optimal case and placement strategy, showing that data driven workloads (graph analytics in this case) can benefit from partitioning between heterogeneous memory even when all of the data can fit into HBM.

Moreover, ProfDP requires manual intervention from the programmers to explicitly perform the data placement and needs to run at least twice to compute a sensitivity metric and make placement decisions. On the other hand, we provide programmers a tool to relabel the vertices such that programmers do not have to make changes in their code. Additionally, we provide them with knobs to tune parameters corresponding to the placement strategy. We believe this would make the programmers' life easier and increase their productivity, as code bases are usually large.

D. *Users*

Our optimization strategy targets on those who run graph applications on heterogeneous memory systems and want to exert minimal effort in taking advantage of HBM. A user does not need to make modifications to their existing code as our optimization is statically done prior to their application running and also does not require additional hardware or online profiling. A user, if they desire, may also tune the performance of their application with different knobs made easily available to them (no code changes).

Additionally, other users may be those who can use our placement strategies as a building block for more advanced placement techniques, improving upon the ones that we propose.

III. Heuristic-based Static Graph Placement Optimization

To form placement optimization strategies, we rely on our intuition that good vertex placement policies can be derived from the topology of a graph. Once these characteristics are determined, we can relabel the vertices so that they reside with a contiguous address space in each type of memory (providing good spatial locality). By relabeling the vertices we have very fine-grain control on tuning which vertices reside in which memory.

To implement the strategy we modify a well known light-weight shared memory graph processing framework known as Ligra [4]. Modification to the framework involves analyzing an adjacency list, relabeling vertices to form a new adjacency list, and then allocating the vertices in either HBM or DRAM.

A. *Placement Strategies*

We develop several strategies and explain how they work using an example weighted directed graph shown in Figure 2. Some strategies take parameters (which can be tuned via knobs), which guide vertex placement in a slightly different way, but still follow the same fundamental placement algorithm.

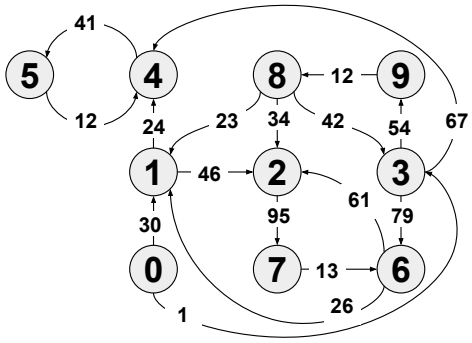
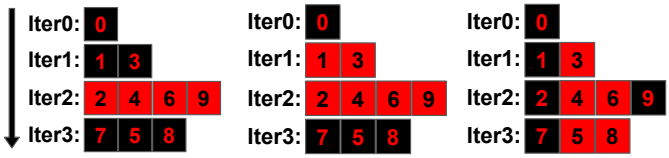


Fig. 2: Example Weighted Directed Graph



(a) Frontier(3) (b) Interleave(1,2) (c) Intraleave(1,2)

Fig. 3: Placement decisions made by traversal-based strategies on the example graph. Frontiers are listed as rows in the order of traversal. Red-base-**black-font** indicates the vertex is in HBM, and **black-base**-red-font means the vertex is in DRAM.

*1) **Degree**:* The Degree strategy uses two knobs to tune the placement decision: a threshold for out-degree, and a cut-off percentile for the list of children vertices in ascending order of in-degree.

First, the vertices with outgoing edges greater than the corresponding threshold are selected (vertices 3 and 8 in Table I). Then, we pick one of the selected vertices, examine it's child vertices, and sort them in ascending order with respect to the number of incoming edges. For all child vertices, we look at its percentile of the incoming edges, and if it is less than the cut-off percentile, we place it in HBM. This process is then repeated for all vertices (parent) initially picked.

The intuition behind this placement strategy is that a vertex with more number of outgoing edges would require more bandwidth while visiting its children, hence the children are chosen for placement. Additionally, if a child vertex has more incoming edges, it is more likely to be accessed often and would benefit from the lower latency of DRAM.

Table I shows the results of applying the Degree strategy on the example graph of Figure 2, with the out-degree threshold set to 2 and cut-off percentile set to 80.

TABLE I: Placement decision made by the Degree strategy. Out degrees in bold font are those exceeding the threshold (set to 2), and the bold vertices in adjacency list are those selected to be placed in HBM.

vertex ID	out degree	adjacency list (in degree)
0	2	1(3), 3(2)
1	2	2(3), 4(3)
2	1	7(1)
3	**3**	4(3), **6(2), 9(1)**
4	1	5(1)
5	1	4(3)
6	2	1(3), 2(3)
7	1	6(2)
8	**3**	1(3), **2(3), 3(2)**
9	1	8(1)

*2) **Frontier**:* This strategy analyzes the graph starting from the root node specified to determine the frontiers (waves) in a Breadth First Search (BFS) fashion. A frontier that has a number vertices beyond a controllable threshold will be placed in HBM. For example, the frontier of iteration 2 in Figure 3a includes four vertices (2, 4, 6, 9) using a threshold of three, hence it is placed into HBM. Conversely, if the number is less than the threshold it will be placed in DRAM. With this strategy, frontiers that would be accessed sequentially can be placed in HBM together.

The intuition here is that this may provide benefit to graphs which are highly connected and have very few frontiers. Frontiers with a large number of vertices may benefit from the high bandwidth, while those frontiers with a smaller number of vertices will not need the extra bandwidth.

*3) **Interleave**:* The interleave strategy is similar to the frontier strategy except that every frontier will alternate in placement between HBM and DRAM. A knob is provided for alternation ratio to control the number of frontiers per memory type. For example, Figure 3b shows if a ratio of 1-2 is chosen for the example graph (Figure 2), then the vertex in the first frontier goes to DRAM, the vertices in the following two frontiers (iteration 1 and 2) are placed in HBM, and finally the vertices in iteration 3 are placed in DRAM. If the ratio is 1-1, it means that every frontier will alternate in placement, so iteration 0 and 2 would be in DRAM, and iteration 1 and 3 woud be in HBM.

The intuition in this strategy is that a graph with a large number of vertices having small degree may benefit from placement in HBM, as the frontiers are small but there are many of them.

*4) **Intraleave**:* The intraleave strategy refines the interleave strategy by making the partition granule finer-grained. Rather than alternating per frontier, vertices alternate within the frontier. The knob presented to tune this strategy is again a ratio of DRAM to HBM, but the partition granule is vertices rather than frontiers.

The motivation for this strategy was to see how it compared to interleave and if a finer-grained partition provided any meaningful insight or improvements.

Figure 3c shows the results of applying Intraleave strategy

on the example graph with a ratio 1 to 2.

*5) **Random**:* A random strategy is simply that, random. A seed value is provided so that the distribution can be reproduced as well as a bias to determine the probability of being placed in HBM over DRAM. The use of a random strategy was to show how our derived placement strategies compared to randomly choosing vertices and if there was any merit in our approach.

B. Ligra Modification

To determine the characteristics of a vertex, the graph must first be pre-processed using its adjacency or weighted adjacency list. We have modified Ligra such that an intermediate step is taken prior to constructing the graph in which the adjacency list is analyzed to determine the number of incoming and outgoing edges to each vertex. Upon obtaining this information a placement policy is run, as described in III-A, which relabels the vertices to form a new adjacency list according to how they should be partitioned. In the new adjacency list there are 2 logical groups numbered from (0:D-1) and (D:N-1). D is the number of vertices going to DRAM, leaving N-D as the number of vertices going to HBM. The new adjacency list preserves the original topology of the graph (relabeling the vertices numbers only). The value of D is inserted at the top of the new adjacency list, which Ligra uses to know when to stop allocating vertices in DRAM and switch to allocation in HBM.

After relabeling, the new adjacency list is ready to be consumed by the framework and create the graph. We modified Ligra to use the $memkind$ library so that $hbw\_malloc()$ can be called to allocate memory in HBM. The framework allocates the first D vertices using $malloc()$ and the remaining N-D nodes with $hbw\_malloc()$. The graph is now ready to be processed by an application.

C. Persistent Topology

Relabeling the vertices does not change the topology of the graph in any way. To ensure that our remapping was done correctly, we run the Single Source Shortest Path (SSSP) application with both the original and relabeled adjacency lists, observing that the results produced are the same.

IV. Evaluation

In our evaluation we define the methodology used, reasoning for the selected graphs and applications, present the results in terms of execution time, and discuss the notable observations.

A. Methodology

To effectively evaluate our proposal, we run our modified Ligra framework on a real KNL node measuring execution time of 3 applications on 4 different graphs with varied topology (graphs are listed with basic properties in Table II and are characterized as Figure 4). The applications run are Breadth First Search (BFS), Single Source Shortest Path (SSSP), and Connected Components (CC). These applications were chosen because they are common graph traversal applications and native to the Ligra framework. The 4 graphs chosen were Road TX (TX), Twitter MPI (TW), Orkut (OK), and a synthetic graph known as Recursive MATrix (RMAT). These graphs were chosen to contrast size and connectivity, and our selection intended to represent how effective our optimization strategy is across graphs with varied topology. In other words, did our placement strategies benefit some applications/graphs while hurting others.

TABLE II: Graphs Dataset [5]–[7]

Abbr.	Name	$\|V\|$	$\|E\|$
TX	Road network Texas	1,393,383	3,843,320
OK	Orkut on-line social network	3,072,627	117,185,083
TW	Twitter (MPI)	52,579,683	1,963,263,821
RMAT	Synthesized RMAT graph	400,000,000	2,000,000,000

	Small Size (<= 16 GB)	**Large Size** (> 16 GB)
Small Degree (<= 10)	Road TX	RMAT2B
Large Degree (> 10)	Orkut	Twitter

Fig. 4: Characterization of Graphs used

For parallelism support, we use OPENMP rather than CILK since the number of threads as well as the affinity for them can be defined, resulting in more stable results. We run each test configured to use 68 threads with each tied to a physical core on KNL. We chose 68 threads because we observed that DRAM bandwidth became saturated at 64 threads using performance counters, as shown in § IV-C.

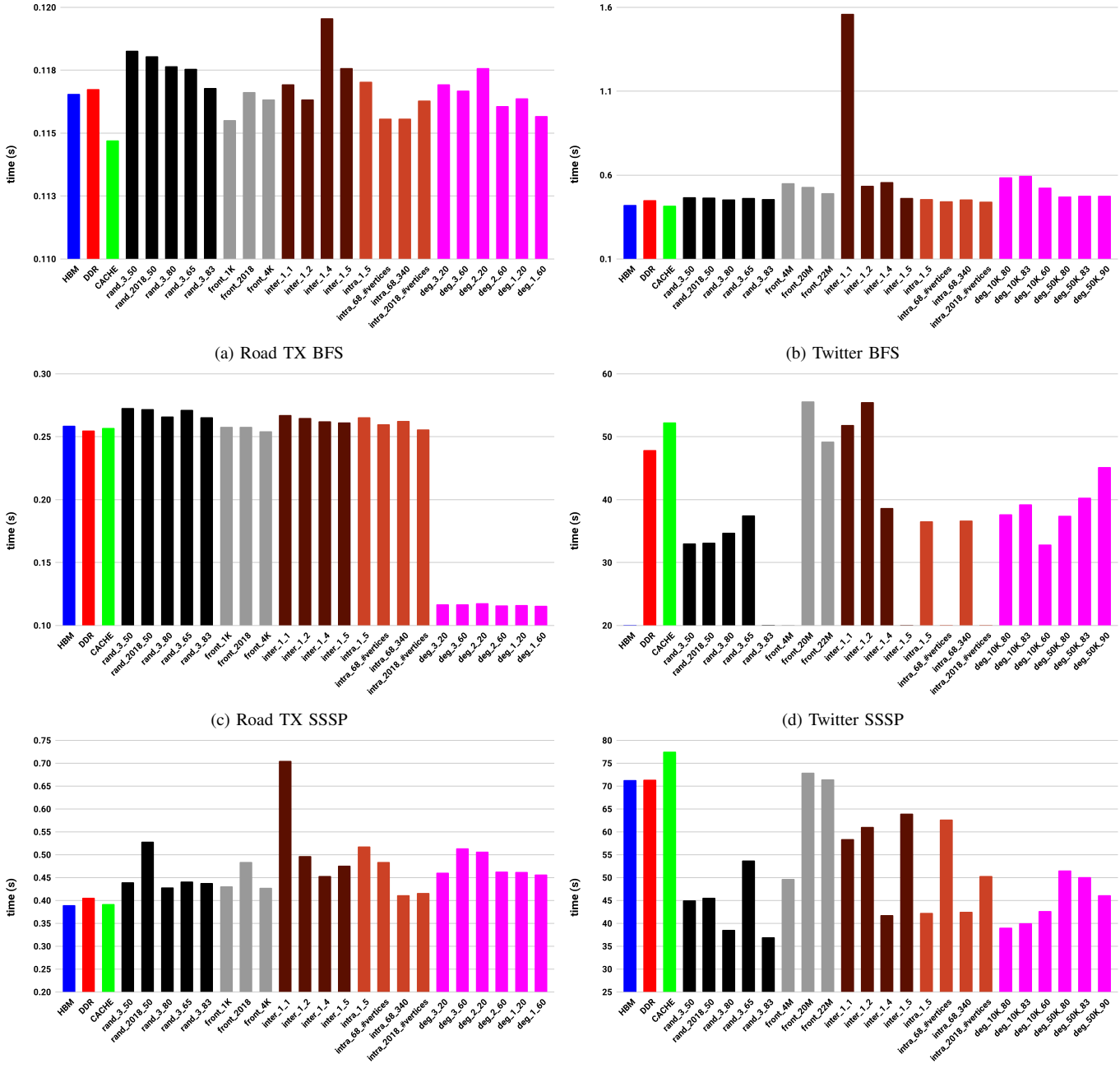
We measure execution time and attempt to collect performance counters for insights into the bandwidth achieved. We compare the execution time of these applications in various memory configurations (DRAM only, HBM only, HBM as a cache, and using our placement strategy) and compare it to the baselines of all in DRAM or all in HBM. Each experiment was repeated 50 times to improve the reliability of results, and execution time is presented as the geometric mean of the 50 runs.

B. Results

The overall performance of our tests were mixed with no single placement strategy having a clear across the board advantage over the others, as shown in Figures 5 and 7 (with the legend mentioned in Figure 6).

The knobs chosen for placement strategies were either random or purposefully chosen by looking at the distribution of incoming and outgoing edges of the vertices, the number of vertices visited in an iteration i.e. the frontier size, or the number of physical cores on KNL (68).

To motivate discussion we identify notable observations below and attempt to provide a rational behind them. Some runs are omitted because they encountered segmentation faults during execution (empty bars in the graph). The segmentation faults are the result of having more vertices to place in HBM than it can accommodate.



(a) Road TX BFS

(b) Twitter BFS

(c) Road TX SSSP

(d) Twitter SSSP

(e) Road TX CC

(f) Twitter CC

Fig. 5: Execution time on Road TX and Twitter graphs for three applications (BFS, SSSP, CC)

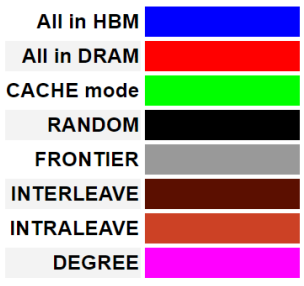
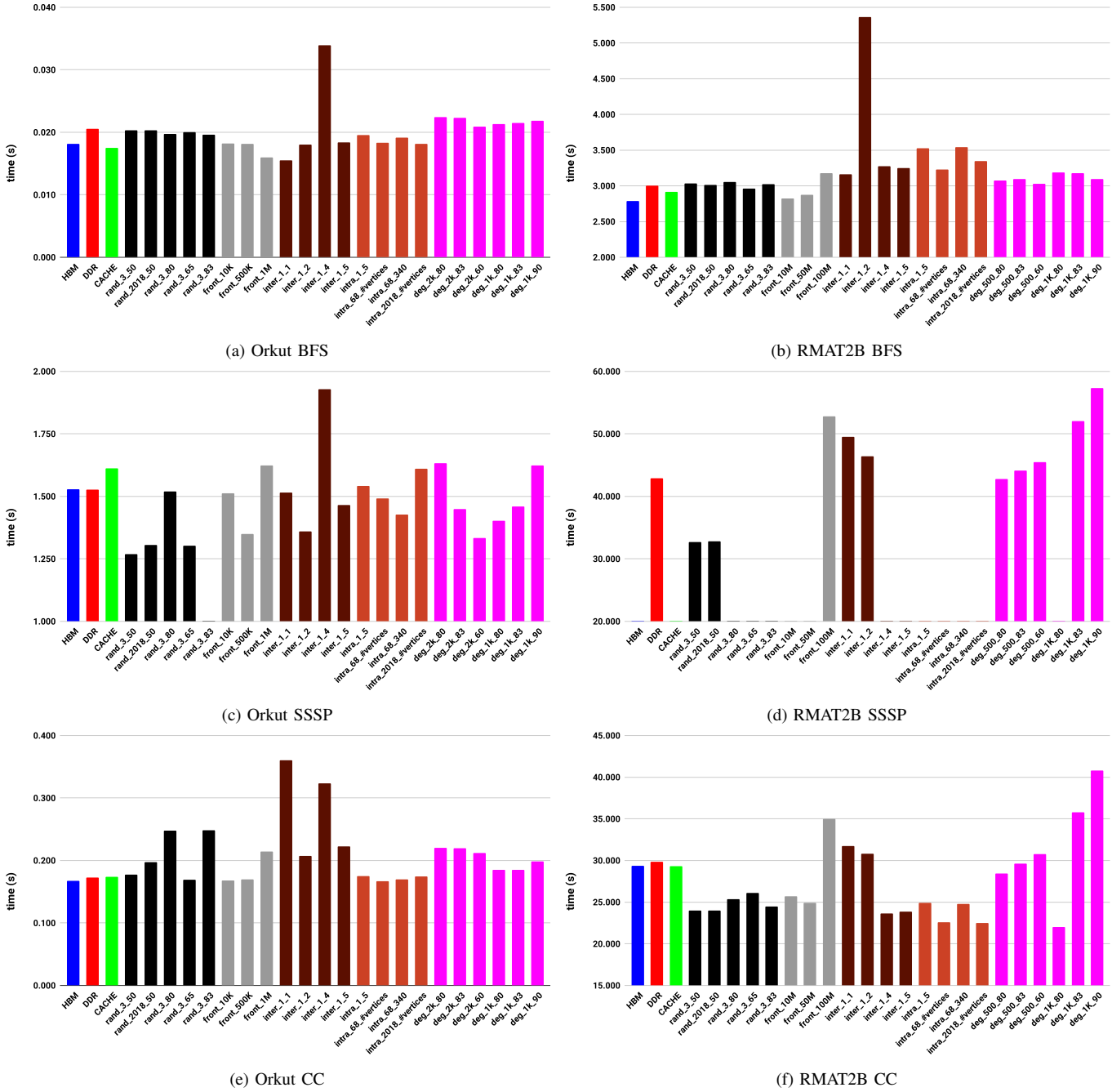


Fig. 6: Legend for the figures

1) 2X Speedup with Degree SSSP: The most notable observation is the 2x speedup in Figure 5(c) for SSSP on Road TX with all variants of the degree strategy performing well. We think Road TX is ideally suited for the Degree strategy because it prioritizes incoming edges of the children. Since Road TX has many vertices with few edges, and the time complexity of the SSSP algorithm (Bellman-Ford) is $O(m.n)$, prioritizing a vertex that has less incoming edges to it would benefit most from being placed in HBM.



(a) Orkut BFS

(b) RMAT2B BFS

(c) Orkut SSSP

(d) RMAT2B SSSP

(e) Orkut CC

(f) RMAT2B CC

Fig. 7: Execution time on Orkut and RMAT graphs for three applications (BFS, SSSP, CC)

2) BFS Outlier: Across the board the BFS application performed the most consistently with any strategy and graph. This was surprising as we thought this application would be the easiest one to exploit with either our frontier or intraleave strategy, and would show the most improvement to the baseline.

Both the frontier and intraleave strategies perform close to the baseline however, in each of the BFS runs there is a clear outlier with the interleave strategy that is substantially worse than the others (sometimes by $2x$). On large graphs such as Twitter that are highly connected with few frontiers (<10), we believe it does so poorly because a frontier with a large number of vertices is placed in DRAM rather than HBM. The performance is then hindered by the bandwidth that DRAM can provide. Conversely, the frontier strategy is not as susceptible to this because the size of each frontier is taken into account and no large frontiers are placed into bandwidth limited memory.

3) Execution Time Differences: With BFS each vertex is visited once while with other applications and corresponding

algorithms, each vertex is visited more than once and hence have longer execution time. Moreover, it seems that the longer the execution time, the more chance there is of boosting performance.

4) Random: We observe that randomly placing data across the heterogeneous memory system provides significant improvements when the size of the graph or the application is large. As in Twitter and RMAT, a random strategy achieves up to 2x and 1.3x speedup (wrt HBM only) respectively. Furthermore, for Orkut SSSP it provides 1.2x speedup while not being able to improve upon the other applications. We believe this behavior is because SSSP takes weighted graphs as an input, and hence consumes more memory, while BFS and CC take unweighted graphs. There seems to be a correlation between the size of the graph and how well the random strategy does.

5) Advantageous to Partition: In every graph there is at least one strategy that performs better than placing all vertices in either HBM or DRAM. Our results show that graphs can benefit from partitioning between the two types of memory and that the optimal is not when all data is placed in fast memory, as ProfDB assumes. Although we do not discover a strategy that is all encompassing or the best, we do show that it is possible to get better performance through at least some method of partitioning a graph.

C. *Performance Counter Pitfalls*

To evaluate the delivered bandwidth during application execution we attempted to query the performance counters available using Linux's $perf$ tool. Our hope was that we would be able to analyze the bandwidth delivered from each type of memory and extract insights about how a placement strategy impacted bandwidth utilization. To sanity check our use of the perf tool and performance counters we first developed a microbenchmark which ran parallel load streams to sequential cache lines (64B). We used $pthreads$ to parallelize the code and the $memkind$ library to allocate the array in HBM. We expected that as we doubled the number of threads we would see bandwidth delivered double until the maximum possible bandwidth was reached from each type of memory.

We used the performance counters in Table III and sampled them every 100ms using perf. Bandwidth delivered was then calculated through the following equation:

$$GB/s = \frac{(counter\_value * 64B)}{0.1s}$$

TABLE III: Performance Counters

Counter Name	Description
offcore_response.demand_data_rd.mcdram	number of MCDRAM memory reads
offcore_response.demand_data_rd.ddr	number of DRAM memory reads

The spec'ed maximum bandwidth of DRAM on KNL is ~90GB/s and ~400 GB/s for MCDRAM. As shown in Figure 8, as the number of threads is doubled bandwidth delivered doubles. However, at 16 threads the bandwidth has surpassed the theoretical maximum bandwidth DRAM can provide. We made many attempts to identify the cause of this misrepresentation of bandwidth but were unsuccessful in finding it. We used a simple time-of-day (TOD) calculation to determine that the bandwidth during the microbenchmark was not actually exceeded (# of loads and time taken to complete was known).

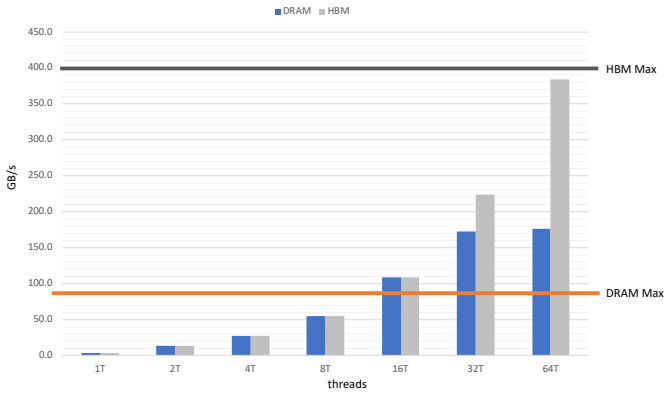


Fig. 8: Bandwidth derived via Performance Counters

Ultimately, due to our inability to identify the culprit of the inflated bandwidth results using perf, we abandoned the use of performance counters in our evaluation, relying on execution time as our main source for performance evaluation.

V. Current Limitations and Future Prospects

The following describe limitations of our current implementation and evaluation that may affect other users. We also discuss areas for improvement that can be pursued in the future.

A. *Threads*

The results we attained were with 68 threads on KNL. Other users would be impacted by their core configuration and thread availability. Fewer number thread configurations may not exceed the bandwidth available from HBM, and find partitioning a moot point with heterogeneous memory.

B. *HBM Capacity Constraints*

A limitation in our modification to the framework was that a maximum size of 16GB could be used for HBM (size available on KNL). Other users may be impacted by this limitation when they have more than 16GB available to use. Currently a warning is given and a segmentation fault would occur if the application were run. Future work should handle running out of memory more gracefully.

C. *Memory Hierarchy*

The memory hierarchy can have a significant impact on the overall performance of a system and the individual applications that run on it. Our optimization, in its current form, does not take into consideration the memory model or

cache hierarchy of different microarchitectures, solely trying to optimize partitioning at the lowest level of memory. Users may be impacted by their specific memory hierarchy and future work should involve investigation into how memory capacity and hierarchy influence the performance of graphs, fueling refinement to the placement heuristics we propose.

D. *Robustness*

As seen from the results in Section IV, none of our five strategies gain considerable performance improvement in all cases. In future work, we would like to try other placement strategies based on different heuristics. Hopefully, a strategy can be found to uniformly improve performance across the board.

E. *Runtime Information*

Allowing duplicated vertices in DRAM and HBM could be another direction of future work. This modification would not require a big change to the framework and can be implemented similar to the *relabel* tool. The intent is to mimic cache and involve some runtime information to introduce dynamic behavior in the placement strategy while having zero-overhead, which contrasts with cache.

F. *Existing Work Comparison*

Due to time constraints, our main focus was on modifying the Ligra framework, devising heuristics for graph placement strategies, and evaluating their performance. As such, we were not able to make 1-to-1 comparisons of how our placement strategies compare to those such as DAP and ProfDB. Future work could encompass direct comparisons to partitioning strategies such as these to quantify the merit of our optimization strategy.

VI. Conclusion

In this paper we propose a fine-grain static placement optimization with a few heuristic-based strategies. We show that the optimal data placement in a heterogeneous memory system can not be assumed by placing all the data in fast memory, and that statically partitioning a graph can provide performance improvements. We extend an existing lightweight shared memory graph processing framework known as Ligra, and provide programmers a few knobs to control the placement of graphs without changing their code base. The evaluation experiments with three different graph applications on four representative graphs to show the effectiveness of our proposed optimization strategies. Some strategies provided up to 2x improvements (w.r.t. HBM only), however, there is no golden strategy among the ones we evaluated that can achieve the best performance in all cases.

In addition, we learned that in order to prevent inflated or misleading results it is important to do a sanity check of your observations. Failure to do so can cause both wasted effort and negligent assertions to be made about the effectiveness of a finding or proposal.

References

[1] J. Gaur, M. Chaudhuri, P. Ramachandran, and S. Subramoney, "Near-optimal access partitioning for memory hierarchies with multiple heterogeneous bandwidth sources," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 13–24.

[2] S. Wen, L. Cherkasova, F. X. Lin, and X. Liu, "Profdp: A lightweight profiler to guide data placement in heterogeneous memory systems," in *Proceedings of the 2018 International Conference on Supercomputing*, ser. ICS '18. New York, NY, USA: ACM, 2018, pp. 263–273. [Online]. Available: http://doi.acm.org/10.1145/3205289.3205320

[3] T. Ligocki, "Empirical roofline tool," https://bitbucket.org/berkeleylab/cs-roofline-toolkit, 2016.

[4] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. New York, NY, USA: ACM, 2013, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/2442516.2442530

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

[6] J. Kunegis, "Konect: The koblenz network collection," in *Proceedings of the 22Nd International Conference on World Wide Web*, ser. WWW '13 Companion. New York, NY, USA: ACM, 2013, pp. 1343–1350. [Online]. Available: http://doi.acm.org/10.1145/2487788.2488173

[7] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable simd-efficient graph processing on gpus," in *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '15, 2015, pp. 39–50.