Community Detection for Large Graphs on GPUs with Unified Memory

Emre Dinçer
Computer Engineering
Izmir Institute of Technology
Izmir, Turkey
Email: emredincer@iyte.edu.tr

Işıl Öz
Computer Engineering
Izmir Institute of Technology
Izmir, Turkey
Email: isiloz@iyte.edu.tr

***Abstract*—While GPUs accelerate applications from different domains with different characteristics, processing large datasets gets infeasible on target systems with limited device memory. Unified memory support makes it possible to work with data larger than available GPU memory. However, page migration overhead for executions with irregular memory access patterns, like graph processing workloads, induces severe performance degradation. While memory hints help to deal with page movements by keeping data in suitable memory spaces, coarse-grain configurations can still not avoid migrations for executions having diverse data structures. In this work, we target the state-of-the-art CUDA implementation of the Louvain community detection algorithm and evaluate the impacts of the fine-grained unified memory hints on the performance. Our experimental evaluation shows that memory hints configured for specific data structures reveal significant performance improvements and enable us to work efficiently with large graphs.**

I. Introduction

Graphics Processing Units have been used successfully to accelerate graph applications [1], [2], [3], [4]. Louvain method proposes an efficient community detection technique by optimizing the modularity metric for large networks to identify highly connected subsets of nodes [5]. Real-world graphs are in sizes of hundreds of gigabytes, while GPU devices still have sizes of a few gigabytes. The existing work on community detection for GPUs deals only with graphs that fit in device memory [6], [7]. While there is some work on distributed multi-GPU graph processing [8], the working assumption is that the graph datasets fit in the collective memory of the GPUs.

CUDA Unified Memory (UVM) presents a virtual union view of memories to enable the developers to work with large data based on a demand paging mechanism by dynamically moving the data between CPU and GPU memories [9]. Memory oversubscription accommodates larger datasets than GPU device memory in exchange for performance loss. While memory oversubscription works better for applications with regular memory access patterns, irregular programs like graph processing suffer from frequent data movements [10]. UVM hints help to optimize data locality by providing guidance to the runtime and potentially eliminating data movements.

In this work, we target the state-of-the-art CUDA implementation of the Louvain community detection algorithm, namely *Rundemanen* [7], [11], and evaluate the impacts of the fine-grained unified memory hints on the performance. Our main contributions are as follows:

- We extend the Rundemanen to support unified memory execution by modifying the memory allocations and copy operations.
- We build a memory access tracker tool to analyze the access patterns and page fault behavior of the target application and to perform a fine-grained evaluation by considering individual data structures.
- We propose fine-grained memory hints for specific data structures to keep data in suitable memory spaces to avoid page migrations and performance degradation in case of memory oversubscription.
- Our experimental evaluation demonstrates that memory hints guiding unified memory page locations result in performance improvements for oversubscription scenarios when considering specific data structures in the target implementation. We achieve significant performance improvements with artificial oversubscription scenarios and datasets larger than available GPU global memory.

II. Background and Motivation

A. Louvain Community Detection Algorithm

Community detection presents a graph analysis that groups nodes within a graph based on their interactions with each other or according to specific properties [17]. Those groups, known as communities, help reveal the patterns of interactions. While several community detection techniques have been proposed, we focus on *Louvain* [5], which is an optimization-based method that tries to maximize *Modularity*, a metric quantifying how well-defined the communities are in a network. Well-defined communities refer to the situation in which the vertices in each community are more densely connected with each other than the other vertices outside of their community. *Rundemanen* [7], *cuGraph* [6], and *cuVite* [8] are state-of-the-art CUDA implementations, which employ certain optimization techniques for utilizing GPU resources. Table I presents the execution times for three codes with different graph datasets on different GPU devices. The results indicate that Rundemanen performs the best and uses memory

TABLE I: The execution times of the Louvain implementations on different GPU devices.

			Rundemanen[7]			cuGraph[6]			cuVite[8]		
dataset	**|V |**	**|E |**	**P4000**	**V100**	**A100**	**P4000**	**V100**	**A100**	**P4000**	**V100**	**A100**
soc-LiveJournal1[12], [13]	4.8M	69M	2.98	1.57	1.79	26.56	15.23	8.17	41.83	23.90	31.46
com-Orkut[12], [13]	3M	117M	7.64	4.06	4.39	-	62.77	38.13	-	44.04	45.31
it-2004[14], [15], [16], [13]	41M	1.1B	-	-	2.98	-	-	-	-	-	-
twitter7[12], [13]	41M	1.4B	-	-	10.49	-	-	-	-	-	-

more efficiently. While cuGraph and cuVite fail to work with larger datasets (given as **-** in table), Rundemanen completes its execution in the A100 device with 80GB global memory. In this work, we focus on Rundemanen and extend the implementation by supporting unified memory to enable the execution of larger graph datasets.

B. CUDA Unified Memory

Unified memory (UVM) [9] is a memory abstraction introduced by NVIDIA with CUDA compute capability 6.0, enabling the host and device to share a single address space and making memory management simple for CUDA developers. Moreover, its memory oversubscription support enables GPU devices to use more memory. Additionally, we can control the behavior of UVM page handling with UVM memory hints. CUDA offers an API call *cudaMemAdvise* that we can apply hints as follows:

- *cudaMemAdviseSetReadMostly*: A copy of the pages corresponding to the specified address range is created by the processor where the operation is performed, and read operations are carried out on this copy. When a write operation is performed, copies on all other processors become invalidated, excluding the one where the operation was performed.
- *cudaMemAdviseSetPreferredLocation*: If the processor seeking access has direct access to the preferred location, the operation is executed without moving the page; otherwise, the runtime migrates the page.
- *cudaMemAdviseSetAccessedBy*: It allows a specific processor to access the data without transferring the page containing it due to remote mapping. When the pages are migrated to the memory of another processor, the mapping is automatically reestablished.

In this work, we evaluate Rundemanen [7] for large graph datasets and propose fine granularity memory hints for its extended UVM version. We would like to answer the following research questions:

- What are the spatial and temporal characteristics of the specific data structures?
- What are the page fault rates for oversubscribed scenarios?
- How can we improve the execution performance by guiding the unified memory with fine-grained memory hints?

III. Data-Access Aware Community Detection

We build an analysis and profiling execution flow targeting our community detection program. Firstly, we modify Rundemanen to support unified memory, then identify all the variables by considering their role in the implementation and define object groups accordingly. Then, we develop a memory access tracker tool built upon CUDA program compilation and execution workflow. Additionally, we implement helper scripts to organize page fault data of the execution. Based on our object groups, fine-grained memory access, and page fault characteristics, we propose fine-grained object group-level memory hints to guide unified memory page migrations.

A. Data Structures

Since we focus on fine-granularity memory access analysis targeting specific data variables, we classify each vector-typed variable based on its functionality in the implementation and group them into four categories as follows:

- *Graph's CSR*: Represents the CSR representation of the graph being processed.
- *Hash Tables*: Stores aggregated weight from vertices to communities.
- *Community Information*: Includes several objects storing community-related information, such as the weights and sizes of the communities and the community IDs of the vertices.
- *Others*: Includes temporary objects for buckets, new graph information, and other bookkeeping data.

B. Memory Data Collection

We instrument our CUDA program to collect memory access information at runtime. We gather a set of metrics that specify the spatial and temporal characteristics of the object groups. Additionally, we track page faults for oversubscribed scenarios.

1) Memory Access Tracking: We build a memory access tracking framework on top of CUDA compilation and execution workflow. We design data structures and implement functionalities to handle memory accesses and modify PTX source files. Figure 1 presents our complete compilation and execution phases.

Within *Memory Accesses Handler*, we define variables holding essential data and implement a set of functions to perform data collection during program execution. Specifically, we define six variables: buffer holding the memory addresses, buffer holding the access times, current time (Unix time in nanoseconds), the number of elements the buffers hold (size), the buffers' capacity, and the number of operations recorded. The buffers and the current time variable are allocated to the host memory in zero-copy mode, and the others are allocated to the device memory. The current time is updated every 0.1

seconds by a separate host thread. The variable holding the number of operations is used to skip some buffer updates.

Since getting the information related to load and store operations is more practical, we create a Python script, *.ptx processor*, which modifies the target PTX code instead of CUDA kernel code directly. It inserts the directives that include the buffers and variables' definitions into the target code. Then, it reads the original PTX code and copies each line to the target PTX code. If the line contains memory load and store operation on GPU global memory, the processor inserts our additional instructions that handle the memory access tracking with the address and time information.

Our tool generates memory access information, with each entry including the address and the time of the corresponding memory operation, along with the execution logs. We determine which object's address ranges at its respective time.

2) Page Faults: We utilize the NVIDIA CUDA Profiling Tools Interface (CUPTI) library [18] to monitor the addresses of pages experiencing faults. We develop a shared library to initialize and deinitialize the CUPTI library before and after the program's execution. During execution, the library logs activities related to unified memory counters to a file, specifically focusing on the migrations from host to device caused by data coherence.

The shared library is injected into the runtime using the *LD_PRELOAD* environment variable, ensuring it operates alongside the main program with no source code modification. After collecting the logs, a Python script parses the output logs and the execution logs generated by our memory access tracker. This script then associates each page fault with a specific object group, thereby providing a detailed analysis of fault occurrences.

C. Oversubscription Configurations

To evaluate the system performance under different oversubscription rates, we configure a set of oversubscription scenarios, where we pre-allocated memory space on the GPU device memory using *cudaMalloc* [19]. Specifically, we generate five configurations, in which varying percentages of the required space of the graph are pre-allocated on the global memory. The configurations are 0%(No), 10%, 30%, 50%, and 70% oversubscription. To calculate how much memory space we need to pre-allocate, we use the following formula:

$$(Mem.Size) - (1 - \frac{Oversubscription\%}{100}) * (Mem.Req.)$$

where $Mem.Size$ is the total GPU global memory size and $Mem.Req.$ is the maximum (peak) memory requirement of the application.

To estimate the maximum memory requirement of a graph, we first instrument the code by inserting print statements following each memory allocation, resizing, swapping, and freeing operation. This enables memory usage tracking during program execution. Subsequently, a script is executed that cumulatively calculates the memory usage while also recording the peak usage.

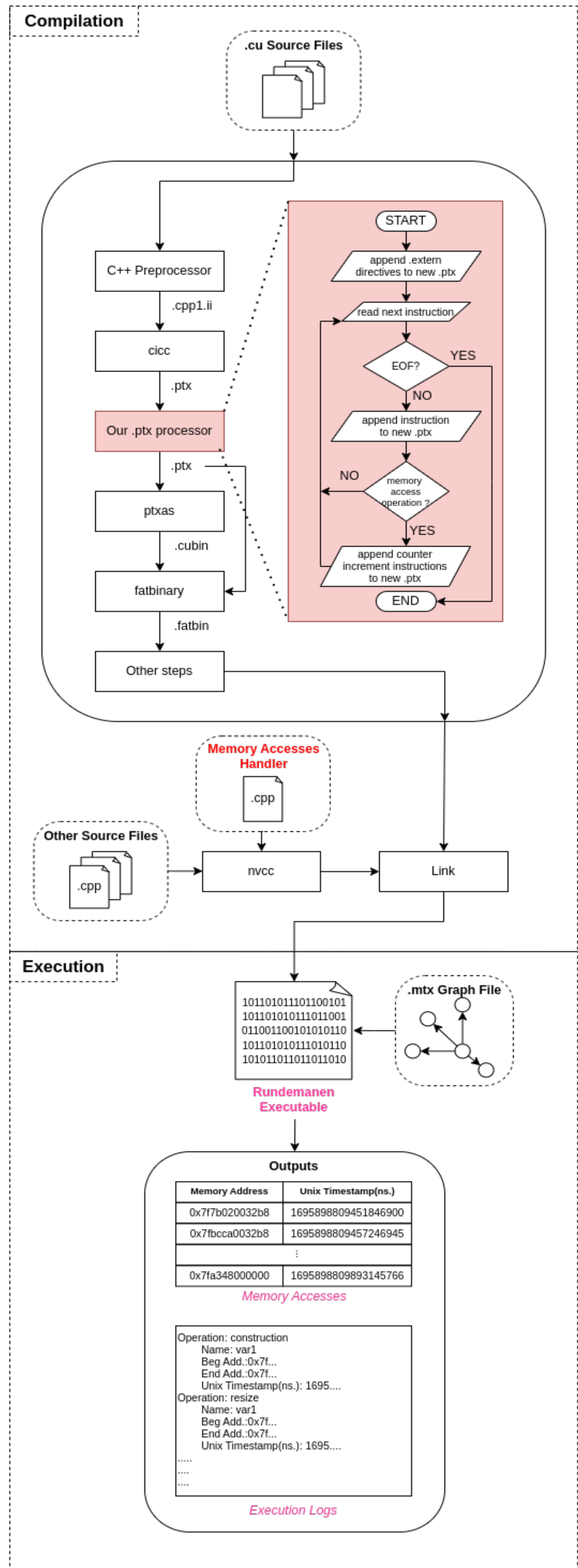

Fig. 1: Our memory access tracking framework flow.

D. Data-Access Aware UVM Hints

Based on memory access and unified memory characteristics of the execution, we choose UM migration policies in the granularity of object groups defined in Section III-A. Specifically, for each object group, i.e., *Graph's CSR*, *Hash Table*s, *Community Info*, and *Others*, we apply a specific memory hint using the *cudaMemAdviseSet* function, either *preferredLocation* or *accessedBy*. After analyzing our profiling-based data access characteristics given in Section IV-B, we define different policies for each object group and construct a set of configurations to evaluate in our performance analysis.

IV. Experimental Study

A. Experimental Setup

We conduct our experiments with various graph datasets on two different systems with different NVIDIA GPU architectures; *PASCAL*: two Intel Xeon Silver 4114 CPU, 32GB RAM, and NVIDIA Quadro P4000, *AMPERE*: Xeon E5-2609 v4 CPU, 64 GB RAM, NVIDIA RTX 3060 Ti GPU. We compile our target program versions with gcc 10.5.0 and CUDA 12.0 toolkit.

Firstly, we conduct a preliminary experimental study to analyze the memory utilization behavior of the program for target datasets. We run experiments to understand both GPU memory and unified memory utilization. Then, we define fine-grained memory hints for guiding data movement for oversubscription cases. While we run our preliminary memory utilization experiments in the *PASCAL* environment for the six datasets fitting in global memory, our unified memory evaluation includes larger graphs in the *AMPERE* system.

B. Preliminary Experiments

As part of our preliminary study, we conduct experiments to determine the memory access behavior of the object groups in our target program. We collect the number of dynamic memory accesses and page faults for each object group. In this way, we understand both temporal and spatial locality characteristics of the data structures, classify the object groups according to their memory utilization, and guide the unified memory for page migration.

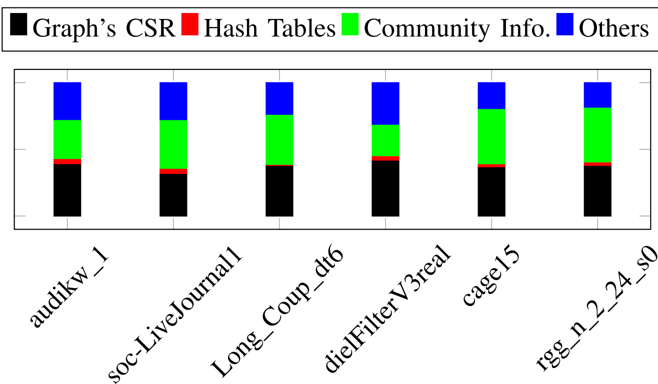

Fig. 2: Memory access ratios for each object group at PASCAL system.

Using our memory access tracker tool, we execute our program with the six datasets and gather memory access counts for each data structure. Figure 2 presents the ratio of the memory accesses for each object group across all datasets. For all datasets, *Hash Tables* structure exhibits the lowest memory accesses. On the other hand, *Graph's CSR* and *Community Info* data structures dominate the others with the highest memory accesses. We can understand that memory hints for unified memory page migration techniques need to be concentrated on the data structures within the given object groups with larger counts, mostly graph-related data.

In addition to memory access characteristics, to understand the fine-grained unified memory behavior, we collect the page fault rates. We artificially set the oversubscription rate to 30% and gather the page fault rates for data structures. Figure 3 presents the page fault rates for each object group. *Graph's CSR* exhibits the highest page fault rate, and we observe that this rate is higher for all datasets than its rate of memory accesses for the same datasets. While the chart does not exhibit a one-to-one correspondence with the data in Figure 2, we can clearly observe that *Graph's CSR* is the primary contributor to most page faults for all datasets. Moreover, we analyze the detailed time-space characteristics in the case of oversubscription, and we see similar effects caused by page migration and memory thrashing. Due to space limitations, we do not put those results in the current version of the paper.

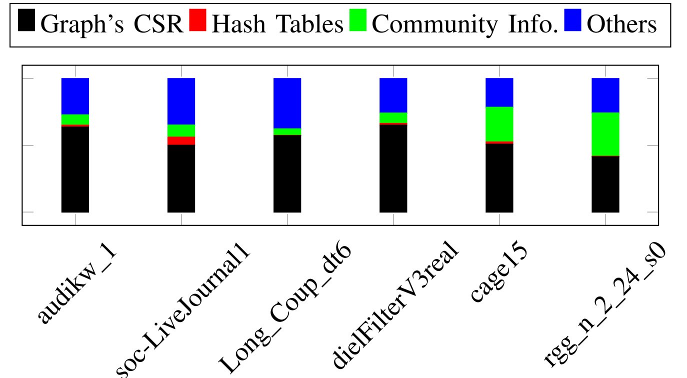

Fig. 3: Page fault ratios for each object group with 30% oversubscription at PASCAL system.

C. Memory Hint Experiments

Considering the memory access behavior and page fault rates, we define UVM hints based on different data structures. Table II presents our configurations, i.e., hints for the specific target data structures.

We collect the execution times at PASCAL and AMPERE systems for several datasets with memory hints at 0%, 10%, 30%, 50%, and 70% oversubscriptions. Figure 5 displays the results along with Table III, presenting the average performance gain of each memory hint over the *base* configuration (no memory hints).

base, *adv3*, and *adv4* interchangeably exhibit the worst performance at 50% and 70%. Having memory hints only to

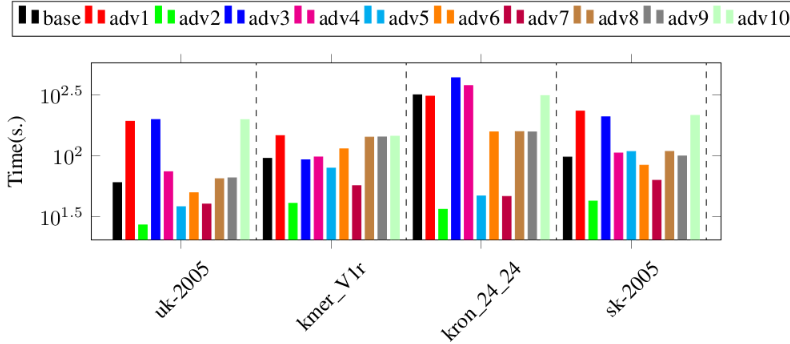

Fig. 4: The running times for non-fitting datasets at AMPERE.

TABLE II: Configurations for memory hints specific to object groups.

Name	preferredLocation CPU	accessedBy GPU
base	-	-
adv1	All object groups	All object groups
adv2	Graph's CSR	Graph's CSR
adv3	Hash Tables	Hash Tables
adv4	Community Info.	Community Info.
adv5	Others	Others
adv6	Graph's CSR, Community Info.	Graph's CSR, Community Info.
adv7	Graph's CSR, Others	Graph's CSR, Others
adv8	Community Info., Others	Community Info., Others
adv9	Graph's CSR, Community Info., Others	Graph's CSR, Community Info., Others
adv10	Hash Tables, Community Info., Others	Hash Tables, Community Info., Others

Hash Tables as in *adv3* does not provide performance benefits over *base* because it does not contribute significantly to the page faults. As expected, when there is no oversubscription, *base* shows the best performance, without any migration and guidance.

We observe that *adv7* has the most performance improvement over *base* at 70% oversubscription. At 10% and 30% oversubscriptions, *adv7*, if not the best, exhibits decent performance. At 50% oversubscription, either *adv2* or *adv7* shows the largest improvement. Furthermore, at 10% and 30% oversubscription, *adv2*, if the best is not *base*, outperforms the performance of other memory hints. *adv6*, *adv8*, *adv9*, and *adv10* adversely affect performance at smaller oversubscriptions for *soc-Livejournal*, *Long_Coup_dt6*, and *kron_g500-logn21*; at higher oversubscriptions, their performance gain falls somewhere in the middle compared to other memory hints. The performance of *adv5* is insignificant for target datasets.

Table III presents average normalized (to *base*) execution times. As the oversubscription rate increases, all hints' average performance gains also increase (except for *adv3* and *adv4* at 30% oversubscription).

TABLE III: The normalized average performance improvements for all hints.

Advise	Oversubscription				
	No	10%	30%	50%	70%
adv1	0.098	1.642	1.742	4.194	18.934
adv2	0.470	**7.201**	**8.583**	9.274	17.004
adv3	0.278	1.351	0.783	1.069	1.333
adv4	0.171	1.323	0.754	1.319	1.629
adv5	0.312	5.687	6.266	6.854	3.260
adv6	0.149	1.833	2.518	4.790	16.512
adv7	0.284	3.778	5.702	**13.020**	**62.516**
adv8	0.131	2.129	2.387	4.478	5.429
adv9	0.117	2.047	2.290	5.272	27.527
adv10	0.094	1.808	1.749	3.715	6.255

The artificial oversubscription scenarios may not completely represent reality. Therefore, to ensure that the results are reliable, we also test some datasets that already do not fit into the GPU global memory in the AMPERE system. Figure 4 demonstrates the results for those datasets. As in artificial oversubscription scenarios above, *adv2* and *adv7* have the shortest running times. In contrast to the results of the artificial oversubscription scenarios, *adv2* overscores *adv7* for all datasets.

Our results demonstrate that fine-grained UVM memory hints configured for specific object groups yield significant performance improvements over other configurations. Specifically, *adv2* (*Graph's CSR*) and *adv7* (*Graph's CSR + Others*) outperforms both the versions without memory hints or with memory hints for other data structures.

V. Conclusion

We evaluate the memory access pattern of Rundemanen code, the state-of-the-art Louvain community detection algorithm implementation for GPU systems. Our analysis reveals that the target code exhibits specific temporal and spatial characteristics for different object groups. We observe a similar page fault behavior for oversubscription scenarios. Based on our observations, we define fine-grained memory hints to help UVM runtime, potentially avoid page migrations, and perform better. Our configurations targeting memory hints for graph-related data result in significant performance improvements over the executions with no hints or other configurations.

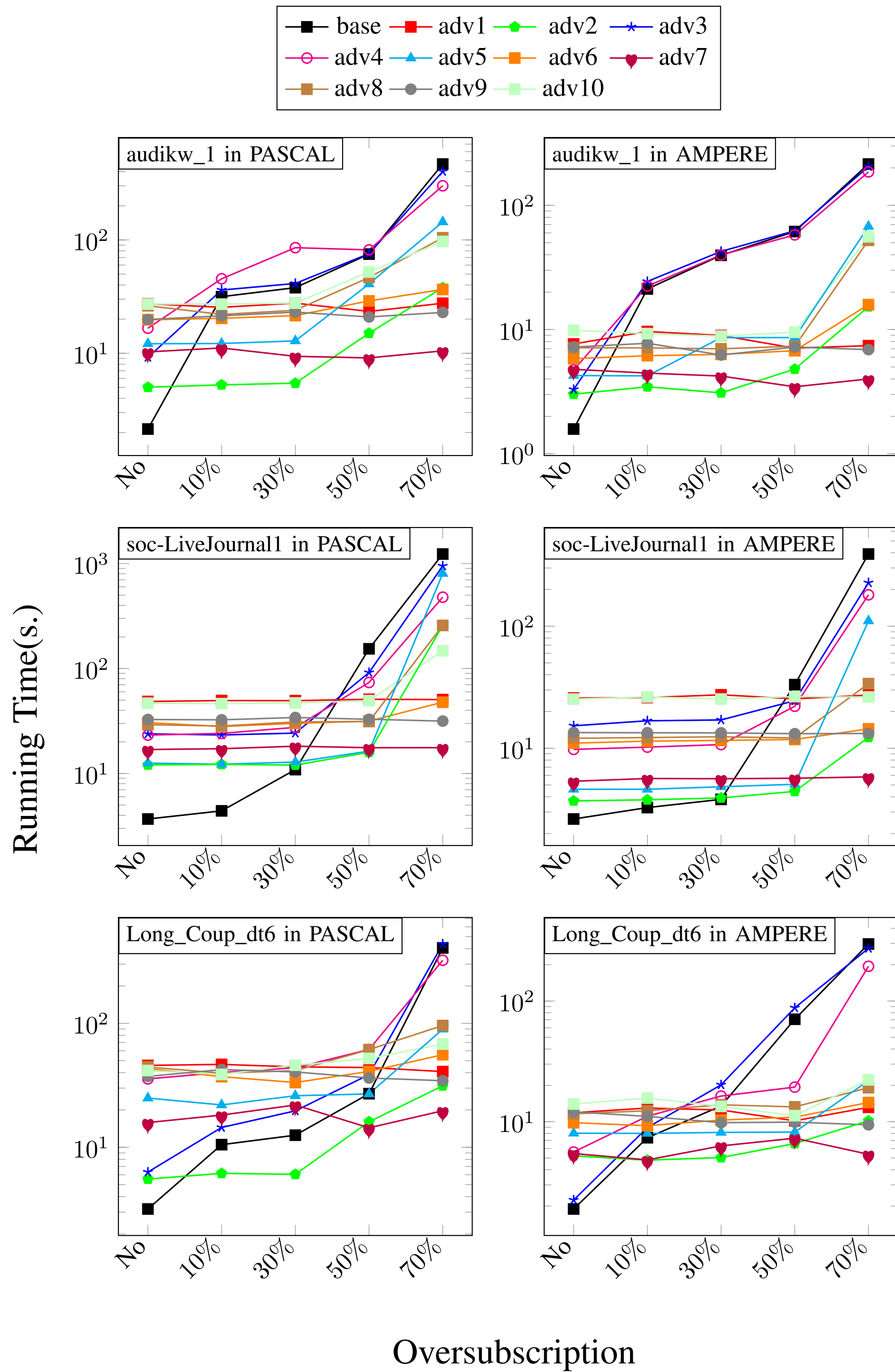

Fig. 5: Execution times for memory hint configurations at different oversubscription rates.

ACKNOWLEDGMENT

The authors gratefully acknowledge the HPC RIVR consortium (www.hpc-rivr.si) and EuroHPC JU (eurohpc-ju.europa.eu) for funding this research by providing computing resources of the HPC system Vega at the Institute of Information Science (www.izum.si). This work was supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK), Grant No: 122E395.

REFERENCES

[1] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *High Performance Computing – HiPC 2007*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 197–208.

[2] C. Yang, A. Buluç, and J. D. Owens, "Graphblast: A high-performance linear algebra-based graph framework on the gpu," *ACM Trans. Math. Softw.*, vol. 48, no. 1, feb 2022.

[3] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens, "Gunrock: Gpu graph analytics," *ACM Trans. Parallel Comput.*, vol. 4, no. 1, aug 2017.

[4] v. E. N. J. Traag V. A., Waltman L., "From louvain to leiden: guaranteeing well-connected communities," *Scientific Reports*, vol. 9, no. 5233, 2019.

[5] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, oct 2008.

[6] "cugraph - rapids graph analytics library," https://github.com/rapidsai/cugraph, 2022.

[7] M. Naim, F. Manne, M. Halappanavar, and A. Tumeo, "Community detection on the gpu," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 625–634.

[8] N. Gawande, S. Ghosh, M. Halappanavar, A. Tumeo, and A. Kalyanaraman, "Towards scaling community detection on distributed-memory heterogeneous systems," *Parallel Computing*, vol. 111, p. 102898, 2022.

[9] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in cuda," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, 2014, pp. 1–6.

[10] C.-H. Chang, A. Kumar, and A. Sivasubramaniam, "To move or not to move? page migration for irregular applications in over-subscribed gpu memory systems with dynamap," in *Proceedings of the 14th ACM International Conference on Systems and Storage*, ser. SYSTOR '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3456727.3463766

[11] A. K. Mahantesh Halappanavar, Howard (Hao) Lu and S. Ghosh, "grappolo," https://github.com/ECP-ExaGraph/grappolo, 2020.

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

[13] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[14] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[15] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th international conference on World Wide Web*, S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, Eds. ACM Press, 2011, pp. 587–596.

[16] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: A scalable fully distributed web crawler," *Software: Practice & Experience*, vol. 34, no. 8, pp. 711–726, 2004.

[17] X. Su, S. Xue, F. Liu, J. Wu, J. Yang, C. Zhou, W. Hu, C. Paris, S. Nepal, D. Jin, Q. Z. Sheng, and P. S. Yu, "A comprehensive survey on community detection with deep learning," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–21, 2022.

[18] "Nvidia cuda profiling tools interface (cupti) - cuda toolkit," 2024. [Online]. Available: https://developer.nvidia.com/cupti

[19] C. Shao, J. Guo, P. Wang, J. Wang, C. Li, and M. Guo, "Oversubscribing gpu unified virtual memory: Implications and suggestions," in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 67–75. [Online]. Available: https://doi.org/10.1145/3489525.3511691