Adaptive Prefetch-Granularity Management for Locality Prediction in Unified Virtual Memory

Jeongha Lee
Dept.of Computer Engineering
Ewha University
Seoul, Republic of Korea
jeongha.lee@ewha.ac.kr

Kyungwoon Cho
Embedded Software Research Center
Ewha University
Seoul, Republic of Korea
cezanne@oslab.ewha.ac.kr

Hyokyung Bahn*
Dept.of Computer Engineering
Ewha University
Seoul, Republic of Korea
bahn@ewha.ac.kr

***Abstract*—Unified Virtual Memory (UVM) automates data migration between host and device memory through demand paging, but its tree-based prefetching heuristic frequently mischaracterizes spatial locality. As a result, both irregular and strongly sequential workloads suffer from excessive or insufficient page migrations, causing substantial performance loss. To address this limitation, we propose a dynamic prefetch-granularity adaptation framework that adjusts migration size at runtime based on the access patterns manifested in each page fault batch. The framework consists of two sequential stages: a coarse locality-guided controller that selects an initial granularity, followed by a fine outlier-based refinement step that suppresses sensitivity to transient irregularities. Experimental results across diverse sequential and random-access workloads show performance gains of 1.2×–1.9× in memory-sufficient settings and 1.1×–2.5× under oversubscription. The proposed method further achieves approximately 98% of the performance delivered by empirically optimal static granularity. These findings demonstrate that adaptive granularity control can substantially enhance UVM efficiency and stability for memory-intensive GPU workloads.**

Keywords—Unified Virtual Memory (UVM), GPU Memory Management, Prefetch Granularity, Adaptive Prefetching.

I. INTRODUCTION

GPUs provide massive parallelism and high on-device memory bandwidth, making them indispensable for data-intensive high-performance workloads such as deep learning, large scale graph analytics [1, 2], and scientific simulations [3]. However, the physical memory capacity of GPUs remains limited relative to modern dataset sizes, forcing developers to rely on explicit memory management—that is, manually orchestrating data movement using operations such as `cudaMemcpy()` and programmer-directed host–device transfers. This conventional model places substantial cognitive and engineering burdens on developers and often results in suboptimal data placement strategies that fail to adapt to dynamic access patterns. To address these limitations, NVIDIA introduced Unified Virtual Memory (UVM) [4], which provides a unified virtual address space and automatically manages data migration between CPU and GPU memories. By treating host memory as a backing store and enabling demand-based migration, UVM inherently supports memory-oversubscription and significantly improves programmability, allowing applications to operate on datasets that exceed the physical GPU memory capacity [5–7].

Although UVM abstracts memory movement away from the programmer, the underlying migration mechanisms create nontrivial performance challenges. UVM is designed around on-demand page migration, where a GPU access to a non-resident page triggers a page fault and initiates data transfer from host to device. This fault-handling latency stalls execution and can suspend all active warps dependent on the missing data. To alleviate these stalls, UVM incorporates proactive prefetching mechanisms as a fundamental component of its heterogeneous memory design. These mechanisms attempt to predict future access patterns and migrate pages ahead of time, typically using heuristic spatial locality assumptions that prefetch groups of adjacent pages. However, access patterns of modern AI workloads differ substantially from those of traditional workloads [8], causing actual access locality diverges from heuristics assumptions. Such mismatches can lead to unnecessary prefetch traffic, increased oversubscription pressure, and aggravated page thrashing, ultimately degrading overall performance.

While UVM incorporates proactive prefetching to mitigate the high latency of demand-driven migrations, its current implementation relies on a region-based, tree-structured heuristic that selects prefetch-granularity using a static residency threshold. When the fraction of resident pages within a subregion surpasses this threshold, UVM speculatively migrates the remaining pages of that region under the assumption that spatial locality will continue. However, because this decision is derived solely from local residency statistics within a narrowly defined address range, it lacks visibility into the broader and often phase-varying memory-access characteristics of real applications.

This restricted, region-local perspective introduces two systematic inefficiencies. First, in workloads with irregular or weak spatial locality, random-accesses may incidentally satisfy the residency threshold even though no meaningful locality exists. This causes UVM to prefetch pages that are never accessed, wasting PCIe or NVIDIA NVLink bandwidth and polluting GPU memory with irrelevant data. Such mispredicted migrations displace useful pages and exacerbate thrashing in oversubscribed environments. Second, in workloads exhibiting strong sequential or streaming locality, the localized threshold often underestimates the optimal migration size. Because the heuristic remains confined to the immediate subtree, it delays selecting a sufficiently large granularity capable of hiding transfer latency, thereby reducing achievable throughput. As a result, UVM's static, region-bound heuristic fails to distinguish between irregular and sequential phases and frequently produces suboptimal prefetch behavior.

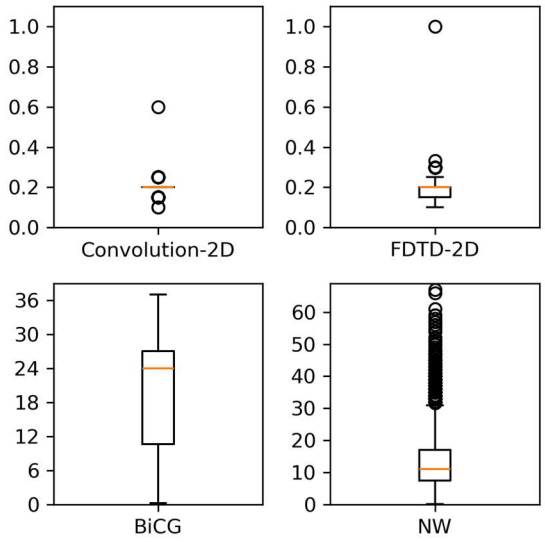


Fig. 1. Locality ratio R across representative workloads.

Prior studies have explored alternative directions, but each carries its own limitations. Choi et al. [6] use NVLink-connected multi-GPU memory as a fast extension of device memory and prefetch large 32MB regions, relying on high inter-GPU bandwidth rather than adapting to the workload's memory-access characteristics. Go et al. [5] adjust prefetch thresholds based on long-term fault statistics or memory pressure, but threshold tuning controls only when and how aggressively prefetching is triggered—not the migration granularity itself.

To overcome these limitations, we propose a dynamic prefetch-granularity adaptation framework that adjusts migration size at runtime based on the observed access pattern. Unlike prior approaches, our method exploits the spatial distribution of page faults within each fault-handling batch and operates in two stages to estimate locality more accurately:

1) Coarse-grained locality estimator: captures the global access trend across the batch; and

2) Fine-grained outlier suppression mechanism: filters transient fluctuations and prevents misclassification.

By combining global trend detection with robust noise filtering, the proposed framework selects aggressive prefetching only when locality is sustained, while constraining migration size in irregular phases. This enables high throughput in sequential regions and avoids unnecessary data movement in random-access workloads.

The primary contributions of this paper are summarized as follows:

- **Analysis of Existing UVM Prefetching**: We identify fundamental limitations of UVM's tree-based prefetching mechanism, demonstrating that its region-local locality estimation frequently mispredicts migration granularity under both regular and irregular memory-access patterns.
- **Dynamic Runtime Prefetch Control**: We introduce a lightweight, runtime mechanism that dynamically adjusts prefetch-granularity using locality signals extracted from each fault-handling batch, enabling adaptive, workload-aware migration decisions.
- **Empirical Performance Improvements**: Experimental results show that our method improves execution performance by 1.5× in memory-sufficient environments and 1.8× under memory-oversubscription, compared to the default UVM prefetching policy.

The remainder of this paper is organized as follows. Section II provides background on UVM's fault-handling and prefetching mechanisms, along with an analysis of their interaction with workload locality. Section III presents the proposed dynamic prefetch-granularity adaptation method. Section IV evaluates its effectiveness through extensive experiments under both memory-sufficient and oversubscribed conditions. Finally, Section V concludes the paper.

II. Unified Virtual Memory: Fault-Handling Behavior and Limitations of Tree-Based Prefetching

A. Batch-Based Fault-Handling Mechanism

UVM does not handle GPU page faults one by one; instead, it aggregates them and processes multiple faults together as a batch. A single batch can include up to 256 faults, and a fault-handling routine may process as many as 20 such batches sequentially [9]. This batch-based processing reduces CPU–GPU interrupts and amortizes the cost of fault-handling.

To understand how faults are grouped and migrated, it is important to consider UVM's internal memory management unit, the Virtual Address Block (VABlock). The UVM driver partitions the GPU-accessible virtual address space into fixed-size regions called VABlocks, each covering a 2MB virtual address range. All migration metadata and residency information are maintained at the granularity of these VABlocks. Thus, even though a batch aggregates multiple faults, the UVM driver decomposes the batch internally along VABlock boundaries: if a batch contains faults that belong to several different VABlocks, the driver processes each VABlock independently, typically issuing separate migration operations per VABlock.

Because batches are formed based on fault arrival order while VABlocks reflect the spatial layout of the virtual address space, the interaction between these two units provides direct insight into the spatial distribution of memory-accesses. During a single fault-handling routine, the handler may process multiple internal batches, and each batch may touch one or more VABlocks depending on how contiguous or dispersed the faulting addresses are. Such differences motivate the need for a simple metric that captures how broadly faults are distributed across VABlocks.

Let num_{batch} denote the number of batches processed in a single fault-handling routine, and num_{va} denote the number of distinct VABlocks involved across those batches. The ratio between these two quantities reflects the degree of spatial locality during batch processing. Workloads with strong locality tend to accumulate faults within a small number of VABlocks, while irregular or random-access workloads spread faults across many VABlocks. To quantify this behavior, we define the locality metric R in (1):

$$R = \frac{num_{va}}{num_{batch}} \qquad (1)$$

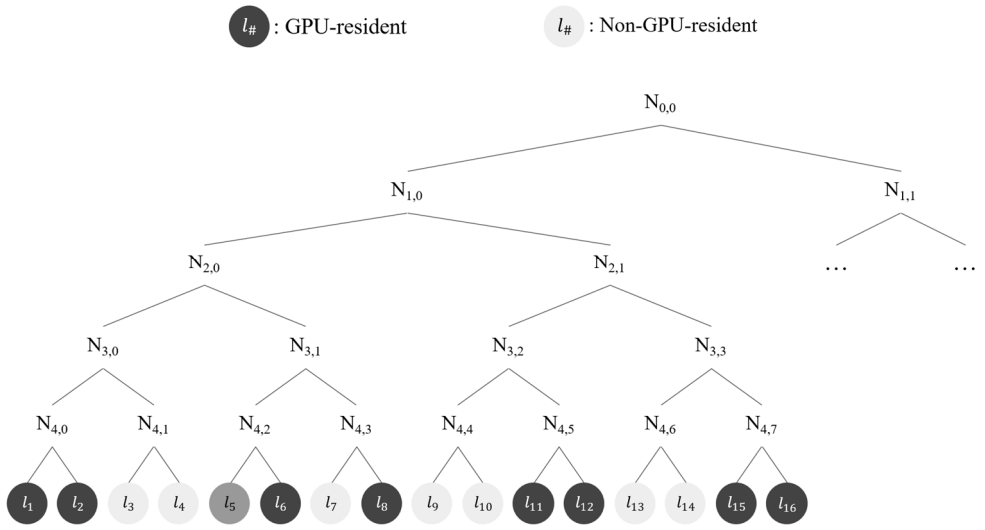


Fig. 2. Illustration of UVM's hierarchical prefetching process.

A lower value of R indicates that multiple faults within a batch map to the same VABlock, suggesting stronger spatial locality. Conversely, a higher value of R signifies that faults are distributed across many VABlocks, implying weak locality. Using this metric, we statistically analyzed differences in UVM batch composition under regular and random memory-access patterns.

Fig. 1 summarizes the distribution of R values across representative workloads. Regular-access workloads such as Convolution-2D and FDTD-2D tend to generate page faults within relatively narrow address ranges. Consequently, the value of R remains consistently low, with median values near 0.2 for both applications. These results indicate that faults within a batch typically map to only a small number of VABlocks, reflecting strong spatial locality and tightly clustered access ranges.

In contrast, irregular or random-access workloads exhibit markedly different behavior. As shown in the figure, both BiCG and NW produce substantially larger R values, demonstrating that faults are dispersed across a far broader set of VABlocks during batch-handling. The median R values rise to roughly 24 in BiCG and 11.1 in NW—two orders of magnitude larger than those of regular-access workloads. This sharp increase highlights the inherently weak locality of irregular workloads, where page faults are widely scattered throughout the virtual address space.

Overall, these findings confirm that R is an effective indicator of spatial locality during UVM fault-handling: low values correspond to compact fault-distribution patterns, whereas high values reflect widely dispersed and irregular-access behavior that spans many VABlocks within each batch-handling routine.

B. Tree-Structured Prefetching Mechanism of UVM

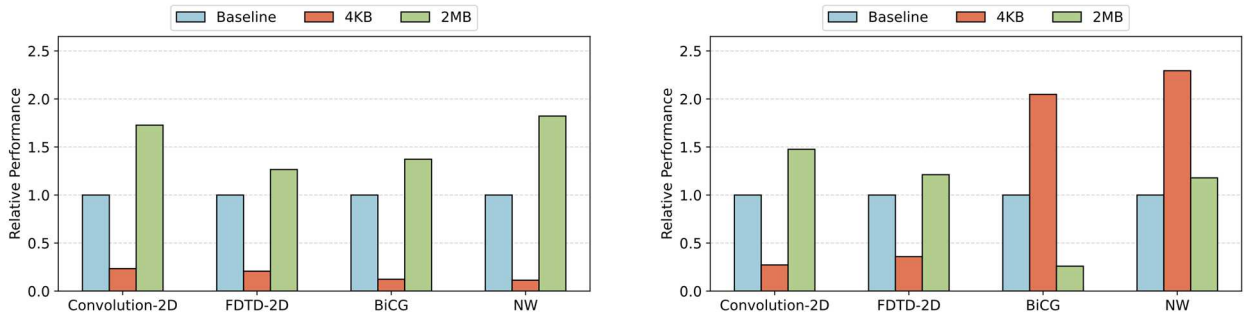
UVM determines prefetching decisions using a tree-structured organization of each VABlock. Internally, a VABlock is divided into 64KB leaf nodes, which are recursively grouped into larger subregions up to the VABlock root. When a page fault occurs, the driver begins at the leaf node corresponding to the faulting page and traverses upward toward the root, evaluating the GPU residency ratio at each level of the tree [5,9,10]. If the residency ratio of a subregion exceeds a preset threshold (50% by default), the driver speculatively migrates all remaining non-resident pages within that region into GPU memory.

Fig. 2 illustrates how the tree-based prefetch decision is applied in practice. In this example, 8 out of the 16 leaf nodes within the VABlock are already GPU-resident, and a page fault occurs at leaf node l_5. The driver begins its traversal from the leaf and evaluates the residency ratio at each ancestor node.

At the immediate parent $N_{4,2}$, the pages in the subregion corresponding to l_6 are already resident, so no prefetching is triggered at this level. Proceeding upward to $N_{3,1}$, the driver finds that 4 out of the 7 pages in this subregion are resident, exceeding the 50% threshold. Consequently, the remaining non-resident page l_7 becomes a prefetch candidate. At the next level, node $N_{2,0}$, 5 out of 8 pages are resident, again surpassing the threshold, which causes pages l_3 and l_4 to be flagged for prefetching. Finally, at node $N_{1,0}$, 9 out of 16 pages are resident, prompting the driver to add l_0 and l_1 to the prefetch list. As a result, multiple pages—l_3, l_4, l_7, l_9, l_{10}, l_{13}, and l_{14}—are ultimately migrated, making the entire subregion corresponding to $N_{1,0}$ resident in GPU memory.

If a subsequent page fault occurs in a leaf belonging to the sibling region $N_{1,1}$, the driver repeats the same hierarchical evaluation. Because the parent VABlock root $N_{0,0}$ now exhibits a residency ratio of 17 out of 32 pages—still above the 50% threshold—the entire subregion of $N_{1,1}$ is marked for prefetching, even if only one of its pages has faulted. This behavior is a key characteristic of the tree-based heuristic: residency in one portion of the VABlock can trigger aggressive migration of large, independent regions, solely because their higher-level ancestor nodes have aggregated residency ratios that exceed the threshold.

Although the tree-based prefetching heuristic in UVM is generally considered effective for workloads with strong spatial locality [10], our analysis reveals several inherent limitations. Even in regular-access workloads, the heuristic's reliance on subregion-level residency can delay the selection of an appropriately large migration granularity. Because the decision is made hierarchically from smaller to larger



(a) Memory-Sufficient

(b) Memory-Oversubscription

Fig. 3. Relative performance under different prefetch granularities. Baseline corresponds to UVM's default tree-based heuristic.

subregions, prefetching may be triggered only after multiple fault events have already occurred, reducing the mechanism's ability to fully hide migration latency.

The limitations become more pronounced for irregular or sparse-access workloads. When accessed leaf nodes are widely dispersed across the VABlock, the aggregated residency ratios at higher-level nodes can exceed the threshold despite the absence of true spatial contiguity. This causes the prefetch mechanism to infer locality where none exists, leading to speculative migration of pages whose address ranges do not correspond to the actual access pattern. Such mispredictions introduce unnecessary data transfers across the interconnect and, under memory-oversubscription, exacerbate GPU memory pressure, often resulting in additional evictions and reduced overall performance.

Fig. 3 compares the execution time of representative workloads under different prefetch granularities. The baseline corresponds to UVM's default tree-based heuristic with a 50% residency threshold. For comparison, we evaluate two fixed migration granularities: traditional 4KB page-level migration and 2MB VABlock-level migration.

Fig. 3(a) shows the results in a memory-sufficient environment where GPU memory can fully accommodate the working set. In this setting, aggressive prefetching generally improves performance for both sequential and irregular-access patterns. Using only 4KB page migration—effectively disabling prefetching—causes an average slowdown of 0.2× relative to the baseline, consistent with the expectation that the tree-based mechanism reduces fault-handling overhead. However, all workloads benefit even more from the 2 MB fixed granularity, achieving an average speedup of approximately 1.5×. This indicates that the default heuristic does not escalate migration granularity quickly enough, even for regular-access workloads where larger prefetch regions would be advantageous.

In contrast, memory-oversubscription introduces substantially different behavior, as illustrated in Fig. 3(b). For regular-access workloads, aggressive 2MB migration remains beneficial and yields an average improvement of 1.3×, mirroring the trend observed in the memory-sufficient case. Random-access workloads, however, exhibit the opposite trend: here, 4KB page-level migration outperforms the baseline by an average of 2.2×. Under memory pressure, the tree-based heuristic frequently prefetches pages that are never used, inflating migration traffic and causing rapid turnover of GPU memory. These unnecessary transfers exacerbate contention and lead to thrashing-like behavior, ultimately degrading performance.

III. Adaptive Prefetching UVM

Motivated by the limitations of existing UVM prefetching—particularly its reliance on localized residency thresholds that fail to distinguish sequential from random-access patterns—we propose a runtime mechanism that dynamically adapts prefetch-granularity based on the evolving fault behavior of the workload. The proposed mechanism operates in two stages. The first stage, pattern-based coarse adjustment (Section III-A), establishes an initial granularity based on the observed access regularity. The second stage, outlier-aware fine adjustment (Section III-B), refines this choice by reacting to short-term deviations in the distribution of page faults. Together, these two stages enable responsive and stable control of the migration granularity, improving prefetch efficiency across diverse workload patterns.

A. Pattern-Based Coarse Adjustment

As discussed in Section II, the degree of locality can be quantified by the ratio R, defined in (2).

$$R = \frac{num_{va}}{num_{batch}} \leq 1 \qquad (2)$$

A smaller value of R indicates that faulting addresses remain concentrated within a narrow virtual address region. In our empirical analysis, regular-access workloads such as Convolution-2D and FDTD-2D showed median R values around 0.2, which is consistent with highly sequential behavior, in which each batch accesses only a small number of distinct VABlocks.

However, R may fluctuate across batches even for sequential workloads. To accommodate this variation while still distinguishing strong locality, we adopt a relaxed criterion and treat $R \leq 0.3$ as the threshold for sequential patterns. We maintain an internal state variable A, which represents an accumulated estimate of the access regularity, and update it according to (3):

$$\begin{cases} A \leftarrow A_{max} & (\text{if } R \leq 0.3) \\ A \leftarrow A + 1 & (\text{else if } R \leq 1 \text{ and } A < A_{max}) \\ A \leftarrow A - 1 & (\text{else if } R > 1 \text{ and } A > A_{min}) \end{cases} \qquad (3)$$

Here, A_{max} and A_{min} denote the upper and lower bounds corresponding to strongly sequential and strongly random

TABLE I. SUMMARY OF BENCHMARK WORKLOADS USED FOR TRACE COLLECTION AND EVALUATION

Workload	Memory-Access Type	Application Domain
Convolution-2D	Sequential	Computer Vision
FDTD (Finite Difference Time Domain)-2D	Sequential	Scientific Computing
BiCG (Biconjugate Gradient)	Random	Linear Algebra, Scientific Computing
NW (Needleman-Wunsch)	Random	Bio-informatics

TABLE II. EXPERIMENTAL HARDWARE AND SOFTWARE CONFIGURATION

Component	Specification
CPU	Intel Xeon E5-2630 v4 @ 2.20GHz
System Memory	28GB
GPU	NVIDIA TITAN V (12GB Memory)
Operating System	Linux Kernel 5.4.0-100-generic
NVIDIA Driver / CUDA	Driver 545.23.06 / CUDA 12.3

behavior, respectively. Prefetch-granularity is adjusted only when A reaches either boundary and remains unchanged while A stays within the intermediate range. In this way, the coarse adjustment stage provides a stable directional signal—toward larger granularity under sequential access, and smaller granularity under irregular-access—without overreacting to short-term fluctuations in R.

B. Outlier-Based Fine Adjustment

If the coarse stage does not trigger a granularity update, we apply a finer adjustment based on the distribution of page faults within each batch. Let f_b denote the number of page faults generated by VABlock b. Let B denote the set of VABlocks within a batch, and $b \in B$ denote an individual VABlock. The mean μ and standard deviation σ of the fault counts within the batch are given by (4) and (5), respectively:

$$\mu = \frac{1}{|B|}\sum_{b\in B} f_b \qquad (4)$$

$$\sigma = \sqrt{\frac{1}{|B|}\sum_{b\in B}(f_b-\mu)^2} \qquad (5)$$

The z-score of each VABlock z_b is then computed using (6):

$$z_b = \frac{f_b-\mu}{\sigma} \qquad (6)$$

We classify a VABlock as an outlier when the absolute value of its z-score exceeds 2 (i.e., $|z_b| > 2$). The fraction of outliers in the batch is defined in (7):

$$frac_{outlier} = \frac{|\{b \in B||z_b| > 2\}|}{|B|} \qquad (7)$$

If $frac_{outlier} > 0.1$, we interpret this as a sign of increased variability in memory-access behavior and increment the density score D. Once D reaches 1, the granularity is increased. Conversely, if $frac_{outlier} \leq 0.1$, we decrement D; and when $D = 0$, the granularity is reduced. This fine-adjustment mechanism filters out short-term fluctuations while allowing persistent changes in the fault distribution to influence granularity decisions. As a result, the system avoids oscillatory behavior yet remains sensitive to meaningful shifts in access variability.

IV. EVALUATION

To evaluate the effectiveness of the proposed adaptive prefetching method, we conduct a series of experiments using representative benchmarks that exhibit diverse memory-access characteristics. In this evaluation, we first describe the workloads and trace-collection methodology, followed by the experimental setup and performance results.

A. Workloads and Trace Collection

We employ a set of open-source micro-benchmarks [11–13] that collectively cover both sequential and random memory-access patterns. To ensure realistic memory footprints, each benchmark is configured to allocate approximately 8GB of managed memory. Furthermore, to emulate GPU memory-oversubscription conditions, we pre-allocate a portion of the GPU memory at application startup using `cudaMalloc()`, thereby reducing the effective GPU memory available during execution. Table 1 summarizes the characteristics of the benchmarks used in this study, including their access patterns, primary operations, and application domains.

B. Experimental Setup

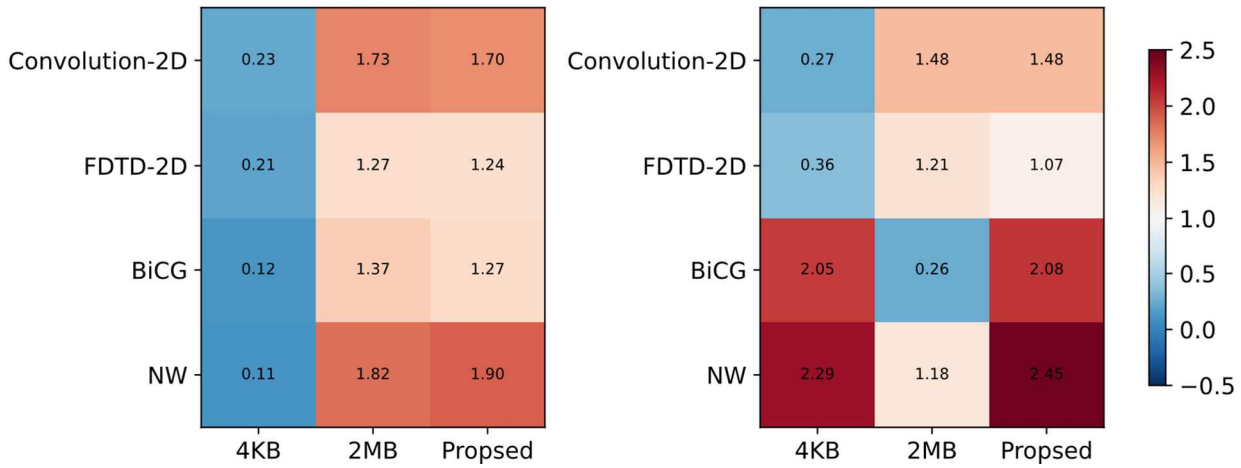
The evaluation compares the proposed method against the conventional UVM tree-based heuristic as well as fixed migration configurations with different granularities. Specifically, we consider 4KB page-level migration, and NVIDIA UVM's 2MB VABlock migration. All experiments described in this section are executed on the hardware and software platform summarized in Table 2.

C. Performance Comparison

Fig. 4 presents the normalized execution time of each benchmark relative to the baseline UVM prefetching policy. Overall, the proposed method consistently improves performance across all workloads and memory conditions. Under memory-sufficient configurations, it achieves a speedup ranging from 1.24× to 1.90× compared with the baseline. In memory-oversubscription scenarios, it maintains robust benefits, improving performance by 1.07× to 2.45×.

A notable observation is that the proposed method closely tracks the performance of the best manually selectable prefetch-granularity—that is, the optimal granularity that would be chosen if the workload's access pattern and memory availability were known beforehand. For instance, random-access workloads such as NW and BiCG perform best with 4KB granularity under oversubscription, while regular-access workloads such as Convolution-2D and FDTD-2D benefit from 2MB granularity. The proposed approach automatically converges toward these choices without prior profiling or offline tuning, achieving on average approximately 98% of the performance attainable by such oracle-like optimal configurations.

In summary, the proposed technique performs aggressive prefetching when memory is abundant and dynamically adjusts its migration granularity when GPU memory becomes



(a) Memory-Sufficient (b) Memory-Oversubscription

Fig. 4. Normalized execution time of benchmark workloads under memory-sufficient and memory-oversubscription conditions, comparing 4KB, 2MB, and the proposed adaptive prefetching method.

constrained. This behavior enables the method to deliver consistent and substantial performance gains over the baseline, while also maintaining performance comparable to that of an ideally tuned configuration that has full knowledge of workload behavior.

V. Conclusions

This paper examined the limitations of UVM's tree-based prefetching mechanism, showing that its locality inference is often misaligned with actual workload behavior, resulting in both under- and over-migration across diverse access patterns. To address this gap, we introduced a lightweight runtime technique that adaptively adjusts prefetch-granularity according to the access characteristics manifested in each fault-handling batch. By integrating a coarse locality estimator with a fine-grained outlier-filtering step, the proposed method continuously aligns migration size with evolving access behavior, without requiring offline profiling or prior workload knowledge. Our evaluation demonstrates that this adaptive strategy consistently improves performance, achieving up to 1.9× speedup in memory-sufficient conditions and up to 2.5× under oversubscription, while reaching approximately 98% of the performance attainable by an oracle-level static granularity. These results highlight that adaptive granularity control offers a practical and effective path toward more robust, efficient, and workload-aware UVM operation, particularly in memory-constrained GPU environments.

Acknowledgment

This work was supported in part by the National Research Foundation of Korea (NRF) under Grant RS-2024-00461678and RS-2025-02214322 funded by Korean Government (MSIT).

References

[1] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: a high-performance graph processing library on the GPU," SIGPLAN Notices, Vol. 51, Iss. 8, No. 11, pp.1-12, 2016.

[2] H. Liu and H. H. Huang, "Enterprise: breadth-first graph traversal on GPUs," In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15), No.68, pp.1-12, 2015.

[3] Y. Liu, A. Wirawan, and B. Schmidt, "CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions," BMC Bioinformatics, Vol. 14, No. 117, 2013.

[4] NVIDIA, "Unified Memory in CUDA 6," Accessed: Nov. 29, 2025, [Online]. Available: https://developer.nvidia.com/blog/unified-memory-in-cuda-6/

[5] S. Go, H. Lee, J. Kim, J. Lee, M. K. Yoon and W. W. Ro, "Early-Adaptor: An Adaptive Framework forProactive UVM Memory Management," Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 248-258, 2023.

[6] S. Choi, S. Choi, T. Kim, J. Jeong, R. Ausavarungnirun, M. Jeon, Y. Kwon, and J. Ahn, "Memory Harvesting in Multi-GPU Systems with Hierarchical Unified Virtual Memory Memory Harvesting in Multi-GPU Systems with Hierarchical Unified Virtual Memory," Proceedings of USENIX Annual Technical Conference (ATC), 2022.

[7] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, "Batch-Aware Unified Memory Management in GPUs for Irregular Workloads," In Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), pp.1357-1370, 2020.

[8] J. Lee, S. Lim, and H. Bahn, "Analyzing Data Access Characteristics of AIoT Workloads for Efficient Write Buffer Management," IEEE Internet of Things Journal, vol. 12, no. 15, pp. 31601–31614, 2025.

[9] T. Allen and R. Ge, "In-depth analyses of unified virtual memory system for GPU accelerated computing," Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), No. 64, pp.1–15, 2021.

[10] D. Ganguly, Z. Zhang, J. Yang and R. Melhem, "Adaptive Page Migration for Irregular Data-intensive Applications under GPU Memory Oversubscription," Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 451-461, 2020.

[11] D. Ganguly, "gpgpu-sim_UVMSmart," Accessed: Nov. 29, 2025, gitHub repository. [Online]. Available: https://github.com/DebashisGanguly/gpgpu-sim_UVMSmart

[12] S. Grauer-Gray, W. Killian, J. Cavazos, R. Searles, and L.Xu, "PolyBench-ACC," Accessed: Nov. 29, 2025, gitHub repository. [Online]. Available: https://github.com/cavazos-lab/PolyBench-ACC

[13] H. Yu, "gpu-rodinia," Accessed: Nov. 29, 2025, gitHub repository. [Online]. Available: https://github.com/yuhc/gpu-rodinia