Early Lessons from Characterizing Unified CPU–GPU Memory on DGX Spark

Seoyeong Lee and Xiaoyi Lu
Department of Electrical and Computer Engineering
University of Florida
{seoyeonglee,xiaoyilu}@ufl.edu

***Abstract*—NVIDIA DGX Spark integrates an Arm CPU and Blackwell GPU behind one coherent LPDDR5X memory pool. A roofline view captures throughput, but the shared-memory programming model can blur where CPU–GPU memory-policy costs are charged. We isolate three attribution factors: ownership state, lifecycle boundary, and timing window. Handoff tests show cold first touch reaches 35× steady reuse, with managed-family paths still 1.6–1.7× above steady at 256 MiB. Lifecycle tests show explicit and mapped host paths spend 264 ms and 246 ms in 256 MiB setup+teardown, versus 0.22–0.24 ms for managed-family paths. In a 400 MiB-per-array vector addition, end-to-end and steady-state timing change policy ranking. Unified physical memory simplifies data visibility, but it does not remove ownership, lifecycle, or timing-window cost attribution.**

***Index Terms*—DGX Spark, unified memory, CPU–GPU ownership handoff, first-touch latency, memory lifecycle, performance characterization**

I. Introduction and Motivation

Coherent CPU–GPU integration is changing host–accelerator memory in Grace Hopper and Grace Blackwell [1], [2]. DGX Spark brings this direction to a workstation-scale GB10 platform [3], [4]. Unlike coherent superchips with processor-local pools, GB10 places its Arm CPU and Blackwell GPU behind one LPDDR5X-backed physical pool [2], [3], [5]. (as shown in Fig. 1.)

A roofline-style view captures the CPU/GPU throughput gap (Fig. 2), but not what a unified-memory cost number includes [6], [7]. On DGX Spark, ownership state, lifecycle boundary, and timing window can change the apparent cost of the same memory policy.

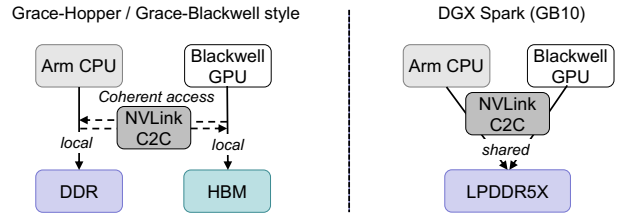


Fig. 1: Architectural comparison of separate and shared physical memory pools in coherent CPU–GPU systems. DGX Spark's GB10 uses one shared LPDDR5X-backed physical memory pool, unlike superchip designs with processor-local memory pools.

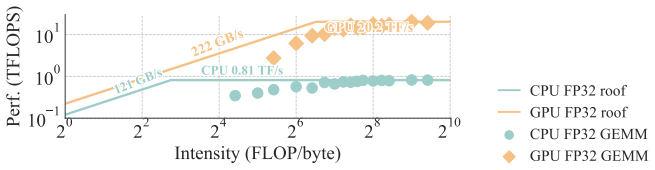


Fig. 2: Roofline-style FP32 view of DGX Spark using empirical bandwidth and compute roofs with measured GEMM points.

II. Experimental Approach

We compare five strategies: managed memory alone, managed+prefetch, managed+hints, mapped host memory for direct GPU access to pinned CPU pages, and explicit copies through pinned host buffers [8], [9]. Experiments run on one DGX Spark GB10: 20-core Arm CPU, integrated Blackwell GPU, 128 GB shared LPDDR5X memory, DGX OS 7, Linux 6.17.0-1008-nvidia, and CUDA 13.0 [3]. Each trial initializes on CPU, runs GPU access, then returns to CPU. We time cold first touch, repeated handoff, and setup/teardown. Factors 1 and 3 use n=10, Factor 2 uses n=30, and Hints aggregates three static hint variants [10].

III. Key Performance Factors and Early Findings

A. Factor 1: Shared Memory Does Not Mean Ownership-Free Access

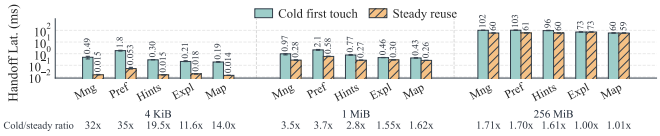


Fig. 3: Cold first-touch and steady-state same-buffer handoff latency across chunk sizes and memory policies. Bars show medians; error bars show IQR, except Hints, which spans the tested hint variants.

Fig. 3 compares cold first touch and steady same-buffer CPU–GPU handoff across chunk sizes. Across the shown sizes, mapped host memory is the lowest absolute-latency path in both regimes, but the high-cost case changes: managed+prefetch dominates small-chunk cold latency, and explicit copy becomes the highest-cost path for 256 MiB steady reuse. The cold/steady premium also changes with size. Managed+prefetch reaches the largest premium at 4 KiB, as high as 35×, while explicit and

mapped host reach near parity at 256 MiB and managed-family paths still retain a 1.6–1.7× premium.

B. Factor 2: Lifecycle Boundary Can Dominate Policy Cost

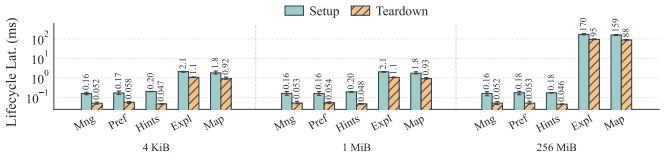
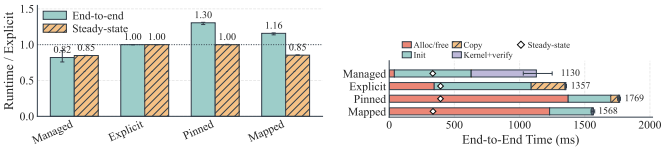


Fig. 4: Setup and teardown latency by buffer size and memory policy. Paired bars show medians; error bars show IQR, except Hints, which spans the tested hint variants.

Fig. 4 separates setup and teardown from the recurring handoff path. At 256 MiB, explicit and mapped host spend about 264 ms and 246 ms in setup+teardown, while managed-family paths stay near 0.22–0.24 ms. Thus the apparent large-buffer penalty of explicit and mapped host can be mostly allocation, mapping, and free work rather than recurring handoff cost. Policy comparisons should state whether lifecycle work is charged or excluded.

C. Factor 3: Timing Window Changes Policy Ranking



(a) Normalized policy ranking (b) End-to-end phase breakdown

Fig. 5: Timing-window comparison for the 400 MiB-per-array vector addition benchmark. **(a)** Runtime normalized to explicit copy under end-to-end and steady-state timing. **(b)** Absolute end-to-end phase breakdown; diamonds mark the steady-state total used in **(a)**.

We run a 400 MiB-per-array vector addition benchmark and compare end-to-end versus steady-state timing windows, then decompose the end-to-end runs to identify which phases drive the ranking shift. Policy ranking changes with the timing window: pinned and mapped are slower end-to-end, but pinned becomes near-explicit and mapped becomes near-managed in steady state. The shift comes from what the timed window charges: pinned and mapped expose 1.2–1.4 s of alloc/free, while explicit exposes a 260 ms copy term; steady-state totals compress to 334–394 ms.

IV. Conclusion and Future Work

On DGX Spark, shared LPDDR5X does not collapse CPU–GPU memory behavior into one cost: managed-family first touch retains a premium, while explicit and mapped setup and teardown can exceed recurring handoff cost. In our evaluations, mapped host has the lowest handoff latency but trails managed and explicit copy end-to-end in vector addition; steady-state timing brings it close to managed as alloc/free is excluded. Thus, neither roofline throughput nor handoff latency alone establishes an application-style policy ranking. In future work, we hope to evaluate these distinctions under CPU–GPU overlap, placement changes, inter-node RoCE transfers and NCCL collectives, and full application and runtime workloads.

References

[1] *NVIDIA GH200 Grace Hopper Superchip*, NVIDIA, 2026, [Online]. Available: https://www.nvidia.com/en-us/data-center/grace-hopper-superchip/. Accessed: Aug. 1, 2026.

[2] *NVIDIA GB200 NVL72*, NVIDIA, 2026, [Online]. Available: https://www.nvidia.com/en-us/data-center/gb200-nvl72/. Accessed: Aug. 1, 2026.

[3] *DGX Spark User Guide: Hardware Overview*, NVIDIA, 2026, [Online]. Available: https://docs.nvidia.com/dgx/dgx-spark/hardware.html. Accessed: Aug. 1, 2026.

[4] *DGX Spark Porting Guide: System Overview*, NVIDIA, 2026, [Online]. Available: https://docs.nvidia.com/dgx/dgx-spark-porting-guide/overview.html. Accessed: Aug. 1, 2026.

[5] N. Fujita, T. Boku, T. Yoshida, T. Shirai, and M. Tsuji, "GPU-CPU Shared Memory Performance Analysis on NVIDIA GH200," in *2025 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*, 2025, pp. 1–2.

[6] 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.

[7] Q. Yu, B. Childers, L. Huang, C. Qian, and Z. Wang, "A Quantitative Evaluation of Unified Memory in GPUs," *The Journal of Supercomputing*, vol. 76, pp. 2958–2985, 2020.

[8] *CUDA C Programming Guide: Programming Model*, NVIDIA, 2026, [Online]. Available: https://docs.nvidia.com/cuda/cuda-programming-guide/01-introduction/programming-model.html. Accessed: Aug. 1, 2026.

[9] *CUDA C Programming Guide: Unified and System Memory*, NVIDIA, 2026, [Online]. Available: https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/understanding-memory.html. Accessed: Aug. 1, 2026.

[10] *CUDA C Programming Guide: Asynchronous Execution*, NVIDIA, 2026, [Online]. Available: https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/asynchronous-execution.html. Accessed: Aug. 1, 2026.