Hydrogen: Contention-Aware Hybrid Memory for Heterogeneous CPU-GPU Architectures

Yiwei Li
Tsinghua University
liyw19@mails.tsinghua.edu.cn

Mingyu Gao
Tsinghua University
Shanghai Qi Zhi Institute
gaomy@tsinghua.edu.cn

***Abstract*—Integrating hybrid memories with heterogeneous processors could leverage heterogeneity in both compute and memory domains for better system efficiency. To ensure performance isolation, we introduce Hydrogen, a novel hardware architecture to optimize the allocation of hybrid memory resources to heterogeneous CPU-GPU systems. Hydrogen supports efficient capacity and bandwidth partitioning between CPUs and GPUs in both memory tiers. With the key observation that CPUs and GPUs exhibit distinct preferences to memory capacity and bandwidth, Hydrogen enables decoupled capacity and bandwidth allocation between CPUs and GPUs with flexible partitioning ratios. It also throttles overly excessive data migration from GPUs with a token-based mechanism. To effectively explore the large, multi-dimensional design space and support dynamically varying application behaviors, Hydrogen uses epoch-based online search for optimized configurations, and incorporates lightweight reconfiguration with reduced data movements. Combining these novel techniques, Hydrogen significantly outperforms existing designs by $1.16\times$ on average, and up to $1.31\times$.**

***Index Terms*—hybrid memory, DRAM cache, heterogeneous CPU-GPU system, fairness**

I. Introduction

Recent advances in big data, artificial intelligence, and in-memory database applications have driven the needs for modern high-performance microprocessors and memory systems [38], [39], [70]. Historically, performance enhancements primarily stemmed from transistor miniaturization. However, with Dennard scaling and Moore's Law approaching their limits [14], [61], there is a growing trend towards organizing transistors into heterogeneous components for better efficiency.

This trend manifests in two distinct forms: *compute-side* heterogeneity and *memory-side* heterogeneity. On the compute side, specialized hardware like GPUs is increasingly employed for data-parallel applications. Although traditionally dedicated GPUs have their independent memory systems, integrated GPUs [1], [21], [24] and other on-chip accelerators are now tightly integrated with the CPU cores and often share the same main memory. On the other hand, memory-side heterogeneity is emerging with technologies like high-bandwidth memory (HBM) [31] and non-volatile memory (NVM) [17], complementing traditional DRAM with higher bandwidth or greater density potentials, respectively. Hybrid memory architectures with multiple tiers aim to combine the advantages of various memory technologies, offering a unified, large, and fast memory abstraction for diverse applications [33], [60].

In the realm of high-performance computing (HPC), the advent of hybrid memory and heterogeneous computing paradigms has witnessed significant evolution, especially in the context of supercomputers. HPC systems, characterized by their demanding computational workloads such as large-scale simulation [7], scientific computing [18], and data analytics [43], increasingly leverage the synergy between heterogeneous computing resources (e.g., CPUs, GPUs, and accelerators) and hybrid memory architectures to achieve unparalleled performance and efficiency. As a representative example, the Frontier supercomputer [37] has over 9000 nodes, where each node uses 1 CPU and 4 GPUs, equipped with 512 GB DDR4 and 512 GB HBM2E, respectively. Each node further uses 4 TB NVM to match the significant memory footprints of HPC workloads. Hybrid memory systems offer a versatile solution to the diverse and memory-intensive requirements of typical HPC workloads, enabling them to benefit from both traditional DRAM and emerging memory technologies like HBM and NVM. This fusion of compute and memory heterogeneity not only addresses the scalability challenges faced by conventional architectures but also paves the way for innovative optimization strategies in supercomputing environments.

We envision that the future of hardware architecture performance growth lies in effectively harnessing both compute-side and memory-side heterogeneity. However, integrating hybrid memory architectures with heterogeneous CPU-GPU systems poses several non-trivial issues. In this work, we focus on the architectures with *integrated* GPUs, where both the CPUs and GPUs can access a unified memory system composed of hybrid technologies. We extensively analyze the importance of carefully managing all the critical resources in the hybrid memory, including *the capacity and bandwidth of both the fast and slow tiers of memory*. We find several key insights. For example, *CPU and GPU workloads have quite different preferences for the fast memory resources*. CPUs prefer larger fast memory capacity while GPUs require higher fast memory bandwidth. However, in traditional memory partitioning schemes, simply assigning different memory channels to CPUs and GPUs results in *coupled resource allocation*, which mismatches the above demands. In addition, the slow memory bandwidth is a crucial resource that affects both CPU and GPU performance. Because state-of-the-art hybrid memory prefers large data block granularities (e.g., 256 B) to exploit spatial locality [41], [65], *data migration between the two*

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

tiers would amplify slow memory traffic and waste bandwidth if the data migrated to the fast memory are not beneficial. Previous work [36], [40], [44], [49] has not fully captured this comprehensive design space or leveraged the above unique workload characteristics, and thus left significant performance potentials on the table. Actually, exploring such a large design space with multiple dimensions of resource management remains a challenging task, and would require not only *effective search strategies to identify optimized configurations*, but also *efficient runtime hardware support to alleviate reconfiguration overheads* for highly varying workload demands.

We propose Hydrogen (H_2), a novel hardware architecture to effectively partition the critical resources of hybrid memory for heterogeneous CPU-GPU systems, aiming to enhance performance isolation and fairness between CPUs and GPUs. Hydrogen accepts user-specified weights of CPU and GPU workloads, and optimizes the weighted throughput of the overall systems transparently to the user applications. Internally, Hydrogen identifies three key resources, namely *fast memory bandwidth*, *fast memory capacity*, and *slow memory bandwidth*, and defines a multi-dimensional design space for partitioning. It introduces novel and specialized partitioning schemes for each resource, catering to the unique characteristics of CPU and GPU workloads. Specifically, Hydrogen *decouples the bandwidth and capacity partitioning of the fast memory* using a novel mapping from cache sets/ways to memory channels, so that the CPUs and GPUs can obtain different ratios of bandwidth and capacity that best match their demands. Hydrogen also *utilizes token-based migration to allocate the slow memory bandwidth*, effectively throttling excessive data migration induced by the GPUs to reduce slow memory traffic amplification. Finally, to tackle the large, multi-dimensional design space, Hydrogen *adopts an epoch-based online sampling technique with hill climbing search* in order to find the best configuration for dynamically varying program behaviors. When updating the partitioning configurations, the system reconfiguration *utilizes consistent hashing mapping and lazy layout changes* to reduce the data movement/invalidation overheads.

Our evaluation across a range of memory-intensive CPU-GPU workload combinations demonstrates that Hydrogen significantly outperforms the best existing designs [36], [49], achieving speedups of $1.16\times$ on average and up to $1.31\times$. This owes to its comprehensive and efficient partitioning schemes for various memory resources as well as its effective adaptability to application behaviors. Overall, Hydrogen represents novel and contention-aware hybrid memory management for heterogeneous CPU-GPU architectures.

We make the following contributions in this paper.

- We present a comprehensive analysis of the unique demands and challenges of integrating hybrid memory architectures with heterogeneous CPU-GPU computing systems, highlighting the importance of carefully partitioning the capacity and bandwidth of both the fast and slow memories, in order to meet the distinct preferences of CPU and GPU workloads.
- We introduce Hydrogen, a novel hardware architecture designed to optimize the allocation of hybrid memory resources across CPUs and GPUs. Hydrogen uniquely decouples the bandwidth and capacity partitioning of the fast memory, and implements a token-based migration strategy for efficient slow memory bandwidth allocation, tailored to the specific needs of CPU and GPU workloads.
- We develop an epoch-based online sampling technique combined with a hill climbing search algorithm integrated into Hydrogen, enabling dynamic adaptation to the varying behaviors of applications over time. These techniques allow Hydrogen to efficiently explore a large, multi-dimensional design space for optimized memory resource partitioning.
- We demonstrate through extensive evaluation that Hydrogen significantly outperforms existing hybrid memory management approaches, achieving on average a $1.16\times$ speedup, and in certain cases up to $1.31\times$, across a variety of memory-intensive workloads. These improvements are attributed to Hydrogen's comprehensive and efficient partitioning schemes, as well as its adaptability to application-specific behaviors.

The rest of the paper is organized as follows: Section II introduces the background knowledge on hybrid memory systems and heterogeneous CPU-GPU architectures, highlighting the significant trends in both compute-side and memory-side heterogeneity. Section III presents the insights and challenges for combining hybrid memory and heterogeneous CPU-GPU systems, followed by a review of related work on memory resource partitioning. Section IV describes our proposed Hydrogen architecture in detail. Section V presents the evaluation methodology, which we use in Section VI to demonstrate the benefits of Hydrogen. We conclude the paper in Section VII.

II. Background

In this section, we introduce the background of system heterogeneity in both the memory and computing subsystems in modern high-performance hardware architectures.

A. Hybrid Memory Systems

Various memory technologies have emerged recently besides conventional DRAM. For example, high-bandwidth memory (HBM) utilizes 3D-stacking to achieve $4\times$ to $8\times$ bandwidth over traditional DRAM at comparable latencies [30], [31]. However, its capacity is constrained to up to 36 GB per stack due to thermal and cost considerations. On the other hand, non-volatile memories (NVM), e.g., Intel Optane memory [17], [25], provide substantial density advantages, but exhibit increased access latency and reduced bandwidth. *Hybrid memory* architectures aim to combine the benefits of fast access and large capacity through intelligently integrating multiple memory technologies [12], [13], [28], [32], [33], [42], [45], [50], [51], [54], [57], [60], [68]. The development and adoption of hybrid memory systems have garnered considerable interest within the industry. For example, Intel has introduced support for NVM and HBM in its

Cascade Lake and Sapphire Rapids processors, respectively. NVIDIA's Pascal architecture implemented HBM and DRAM tiering strategies. In our study, we mainly focus on the case with conventional DRAM as *slow memory* and HBM as *fast memory*, as this combination is widely deployed in modern CPU and GPU platforms [23], [27], [46]. Other heterogeneous memory technologies can be similarly adopted.

Hybrid memories can be organized in two modes: cache (or vertical) mode, and flat (or horizontal) mode. The cache mode designates the fast memory as another level of cache beyond the processor's cache hierarchy in front of the slow memory [13], [16], [20], [28], [32], [33], [42], [51], [63], [68]. In this mode, only the slow memory contributes to the physical memory capacity visible to the operating system (OS), while the fast memory is fully hardware managed. Alternatively, the flat mode exposes both memories to the OS, creating a large unified physical address space [12], [45], [50], [54], [57], [60], [64]. However, it incurs higher cost due to bidirectional data migration between the two tiers, which doubles data traffic and introduces write-after-read dependencies. Our design is compatible with both modes.

B. Heterogeneous CPU-GPU Architectures

This study envisions future heterogeneous systems that amalgamate GPUs and CPUs into a single package sharing a unified address space. This architecture not only simplifies programming but also enhances performance by enabling seamless data sharing and reducing memory transfer overheads. Integrated GPUs have been present since the advent of Intel Sandy Bridge [21] and AMD Fusion [1]. Within these architectures, the GPU incorporates a vast array of execution units (EUs) that are adept at performing massively parallel computations. As an example, Xe-LPG [24], Intel's recent integrated GPU architecture, features 16 EUs per GPU subslice. Each subslice is provisioned with dedicated caches for instructions and data. Six subslices constitute a GPU slice. The GPU can further scale to multiple slices, with each slice connected via an interconnect fabric that aggregates the total L3 cache capacity across the slices. This fabric also bridges the GPU and the CPU over a unified ring interconnect. Such an intricate connectivity paradigm culminates in achieving a peak throughput of 2.1 TFLOPs, $4\times$ higher than a datacenter CPU like Gold 5120. In addition, advancements in chiplet design paradigms [66] promise to escalate the capabilities of integrated GPUs. The ongoing development of integration technologies continuously narrows the gap between dedicated and integrated GPU solutions, and realizes more compact, energy-efficient, and performant computing platforms.

This heterogeneous CPU-GPU architecture is increasingly becoming a cornerstone in the development of upcoming supercomputers to achieve unprecedented efficiency for HPC. El Capital, the successor of Frontier, exemplifies this trend by utilizing the AMD MI300 accelerated computing unit as the basic building block [2]. Each MI300 combines a 24-core Zen4 CPU and an integrated GPU with 228 compute units into a single organic package, complemented with 128 GB HBM3 [4]. The ROCm framework and the Heterogeneous-compute Interface for Portability (HIP) further facilitate heterogeneous programming for HPC workloads, e.g., astrophysics, computational biology, machine learning, and more [3]. This architecture not only streamlines the data processing pipeline but also improves the data transfer speed between these computing units, which is crucial for HPC applications with frequent data exchange between CPUs and GPUs.

In addition, recent core heterogeneity encompasses more than just GPU integration. A notable trend is the asymmetric multi-core architecture [15] that stratifies compute-efficient "big" cores with energy-saving "small" cores. This stratification enables better thread scheduling across different core types and balances between processing performance and energy consumption. Architectures embodying this philosophy include ARM big.LITTLE [6], Apple M1 SoC [5], and the Intel Alder Lake processor family [26]. Another trend is to integrate specialized domain-specific accelerators into the same CPU chip. Intel's latest Sapphire Rapids CPUs have been fortified with Advanced Matrix Extensions (AMX) units, delivering a staggering 180 TFLOPs of BF16 throughput. This throughput is comparable to that of premier dedicated GPUs such as NVIDIA A100 [47]. While we mainly focus on integrated GPUs in this work, our proposed techniques could generalize to these similar heterogeneous architectures.

III. Motivation

This work aims to combine hybrid memory and heterogeneous CPU-GPU systems. We first describe the target architecture in Section III-A, and then discuss the design challenges and opportunities in Section III-B, which are not exploited by the prior work summarized in Section III-C.

A. Target Architecture

To balance with their substantial processing capabilities, heterogeneous CPU-GPU architectures demand better memory systems with fast speed and large capacity; more processing units require higher data access throughput, and are also capable of processing larger datasets. We thus believe that *hybrid memory architectures can substantially improve heterogeneous computing performance*. A well-orchestrated hybrid memory architecture can offer both high bandwidth and large capacity. For instance, HBM can deliver the necessary bandwidth for GPU operations, while conventional DDR4 DRAM can provide ample capacity for all processing cores.

Fig. 1 illustrates such an architecture, where a heterogeneous processor like Intel Xe-LPG is connected to hybrid HBM and DDR memories. We use HBM2E with 16 channels as the fast memory, and 4 channels of DDR4 as the slow memory, both behind the last-level cache (LLC) of the heterogeneous processor. We assume hardware-based hybrid memory management [12], [32], [33], [42], [50], [51], [57], [60]. We use the cache mode by default, but most of our designs directly apply to the flat mode as well, which is detailed in Section IV-F, with the main difference of swapping two blocks between the two memories rather than fetching a block only

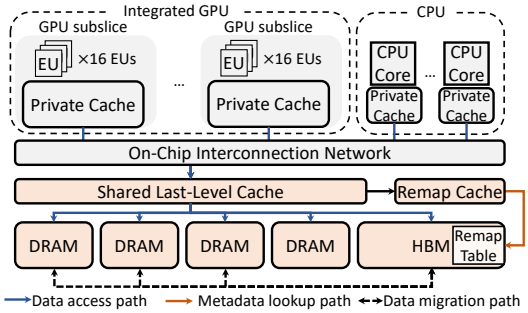


Fig. 1. Target architecture with heterogeneous CPUs and GPUs, and hybrid memories of HBM and DRAM.

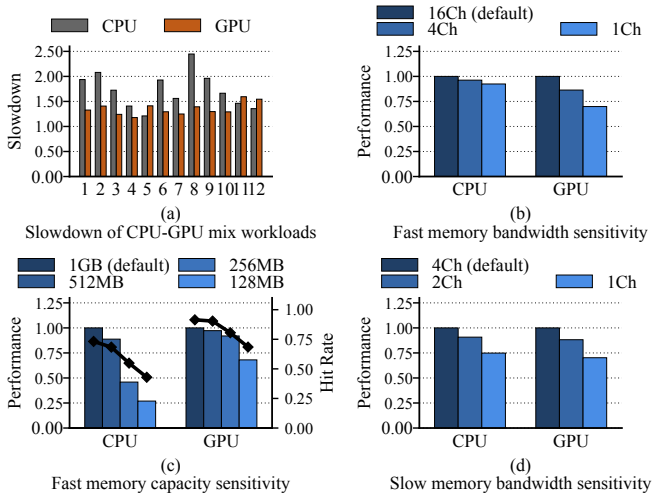


Fig. 2. Performance analysis of CPU and GPU workloads on hybrid memory. (a) shows the respective slowdown of CPU and GPU workloads when running them together in the system compared to running each alone. Larger numbers indicate higher slowdown. (b) to (d) show the performance sensitivity of CPU and GPU workloads in C1 when varying fast memory bandwidth, fast memory capacity, and slow memory bandwidth.

from slow to fast. The whole memory space is organized in a set-associative manner [32], [42], [57], [60]. Both the fast and slow memory spaces are divided into the same number of sets, and each set contains a few fast memory blocks (whose number is called the *associativity*) and some slow memory blocks. Caching only happens among fast/slow blocks within a set. When accessing data, a hardware-maintained *remap table* (stored in the fast memory) first translates the physical address into the actual *device address*, either in the fast memory (a cache hit) or in the slow memory (a cache miss). In the latter case the data block can be fetched into the fast memory based on the replacement policy. The remap table designs are widely used in state-of-the-art hybrid memory architectures to flexibly support not only the cache mode but also the flat mode [41], [50], [60], [65]. We further cache the remap table entries in an on-chip SRAM *remap cache* to speed up metadata probing. Following previous work, Hydrogen assumes data blocks of 256 bytes [41], [65] to balance metadata footprints and data migration cost. It uses a 4-way associativity (4 fast blocks per set) [41] to balance hit rates and metadata overheads. Fig. 11 investigates other associativity and block size choices.

B. Challenges and Opportunities

However, in the above architecture, hybrid memory is a shared resource between CPUs and GPUs, and thus may suffer from severe contention among data accesses from the two different processors. We quantitatively demonstrate such negative performance impact as the design challenges. We further analyze the distinct and complementary memory requirements of CPU and GPU workloads as the unique opportunities to motivate our novel optimizations.

We simulate a system following the configurations in Section III-A running the mixed CPU-GPU workloads in Table II (more details in Section VI). This baseline does not enforce any isolation between CPU and GPU memory accesses. First, Fig. 2(a) measures the performance slowdown when running the CPU and GPU workloads simultaneously on the system compared to running them alone. We observe that the CPU workloads typically experience more significant degradation compared to the GPU workloads. For instance, in C1, the CPU faces a 1.94× slowdown, while the GPU only suffers 1.33×. This can be attributed to the distinct natures of CPU and GPU workloads: CPUs are latency-sensitive and prefer larger fast memory capacity for more data hits, while GPUs, benefiting from thread-level parallelism, are more bandwidth-driven and can tolerate longer memory access latencies [19]. Therefore, to avoid contention between CPUs and GPUs, we would need to **carefully control the sharing of memory capacity and bandwidth (Challenge 1)**.

Furthermore, we also see that the degree of slowdown depends on the workload mixes, indicating that different workload combinations prefer different resource partitioning. Specifically, in C1 above, the CPU performance significantly degrades, so we should allocate more resources to the CPU. But in C5 the GPU exhibits slightly worse degradation than the CPU. This highlights the need for **a dynamic and adaptive approach to resource partitioning based on the workload characteristics (Challenge 2)**.

A more in-depth investigation reveals several novel design opportunities that we can leverage. We vary the numbers of fast and slow memory channels (and their bandwidth correspondingly) as well as the fast memory capacity, and measure how the performance of CPU and GPU workloads changes, respectively. We use C1 as the representative. First, when reducing the fast memory bandwidth in Fig. 2(b), the CPU workloads see little performance variation, while the GPU workloads slow down by up to 30%. This emphasizes the importance of **prioritizing fast memory bandwidth for bandwidth-sensitive GPU workloads (Insight 1)**. On the other hand, when reducing the fast memory capacity in Fig. 2(c), the trend is quite different. Now the CPU workloads suffer from sharper performance degradation, halved at 256 MB. However at the same capacity, the GPU workloads

are able to maintain 92% performance. This can be explained by their access patterns and the changes in fast memory hit rates. CPUs have more random accesses and rely on temporal and spatial locality enabled by a large fast-memory cache size. GPUs mostly issue streaming accesses and enjoy spatial locality even at small cache sizes. This result inspires us to **allocate more fast memory capacity to latency-sensitive CPU workloads (Insight 2)**. Finally we look at the slow memory bandwidth. In Fig. 2(d), both the CPU and GPU show notable slowdown, though the GPU is slightly more sensitive. As a result, it is necessary to **carefully partition slow memory bandwidth between CPUs and GPUs (Insight 3)**.

C. Related Work

Prior research has identified the need for performance isolation within the memory hierarchy. Initial approaches to cache partitioning in multi-core systems primarily focus on either hard partitioning along the way or set dimensions [11], [52], [55], or soft partitioning through adaptation in cache replacement policies [58], [67]. The Utility-based Cache Partitioning (UCP) strategy [52] aimed to allocate cache ways to applications in a manner that minimized cache misses. Nonetheless, the intrinsic limitations of cache associativity constrained way-partitioning to only a handful of coarse-grained partitions, significantly diminishing the effective associativity for each partition. Cho et al. [11] explored set-partitioning of cache space, a technique that, while innovative, necessitated complex modifications and incurred substantial overheads due to data copying or flushing during partition resizing, complicating its integration with the shared address space. PIPP [67] modified the replacement policy to provide control over the cache space allocation of different cores. Vantage [58] further enforced fine-grained dynamic partitioning based on statistical analysis. These designs do not differentiate CPU and GPU workload access patterns, but maximize the overall cache hit rate.

Other designs focus on cache partitioning for heterogeneous CPU-GPU architectures. To balance LLC use, TAP [40] introduced distinct cache insertion/partition strategies for CPUs and GPUs to leverage the fact that GPUs are less sensitive to data caching. Rai et al. [53] explored different read/write hit/miss policies for CPU and GPU requests. HeLM [44] opted for a bypass strategy for GPU workloads that could tolerate latency. OSCAR [69] proposed to substitute conventional SRAM with larger STT-RAM, coupled with interconnect traffic reconfiguration to minimize SRAM write latency and energy. However, a key difference between SRAM caches and hybrid memory systems is that, besides the capacity, the access bandwidth in the latter is also limited and must be carefully considered.

For hybrid memory architectures, APU [66] envisioned to integrate stacked memory and external memory for CPU-GPU heterogeneous systems using chiplets. Kim et al. [35] simply forced GPU workloads to the slow memory, only caching write-intensive blocks in the fast memory. HAShCache [49] prioritized CPU requests in the memory controller queue, partitioned the fast memory capacity, and supported slow memory bypassing. Profess [36] focused on general multi-process fairness for hybrid memory, adopting a bypass policy to ameliorate performance for those processes experiencing the most hit rate degradation or migration cost. Moreover, Intel's Resource Director Technology (RDT) [22] offered a mechanism to govern LLC way-partitioning and memory bandwidth distribution among different cores. None of these designs has comprehensively considered both capacity and bandwidth partitioning for both fast and slow memories as we do. We quantitatively compare our techniques with HAShCache and Profess in Section VI.

IV. Design

In this work, we propose Hydrogen, a novel architecture that effectively partitions critical resources of hybrid memory for heterogeneous CPU-GPU systems, in order to achieve robust performance isolation and improve fairness. It optimizes the weighted throughput (i.e., instructions per cycle, IPC) of CPU and GPU workloads, where the weights are specified by the user. The design of Hydrogen can be easily adapted to other optimization goals. Hydrogen is designed to operate at the hardware level, providing complete transparency to both the system and application software. We use the cache mode for simplicity, and briefly discuss the flat mode at the end.

Based on the insights in Section III-B, Hydrogen pinpoints three critical shared resources in hybrid memory: *fast memory bandwidth*, *fast memory capacity*, and *slow memory bandwidth*. It then employs a novel mapping scheme to decouple the bandwidth and capacity partitioning of fast memory (Section IV-A), and utilizes token-based migration to efficiently allocate the slow memory bandwidth (Section IV-B). To find the best configuration, Hydrogen adopts a dynamic, epoch-based online sampling technique (Section IV-C), with the help of efficient reconfiguration with consistent hashing mapping and lazy layout changes (Section IV-D). The overall workflow of Hydrogen is summarized in Section IV-E, and we discuss other design issues in Section IV-F.

A. Decoupled Partitioning for Fast Memory

We first discuss how Hydrogen partitions the fast memory bandwidth and capacity in a *decoupled* manner, so that the CPU gets enough capacity to ensure high hit rates, and the GPU utilizes sufficient bandwidth. Recall that we assume a 16-channel HBM as the fast memory, with each channel accessed in a 64 B cacheline granularity while the block size is 256 B. Therefore, we group 4 channels into a superchannel to supply one data block in each access, and there are 4 superchannels (simply called channels below) in the fast memory. Hydrogen can work with any number of (super)channels.

Since the fast memory is used as a cache, we can apply conventional cache partitioning schemes, such as way-partitioning [55] and set-partitioning [62]. Way-partitioning techniques isolate CPU and GPU data into separate cache ways, employing partition-conscious replacement policies that evict data from the CPU- or GPU-dedicated ways. Set-partitioning techniques, which utilize OS page coloring [11], distribute CPU and GPU data across different cache sets in

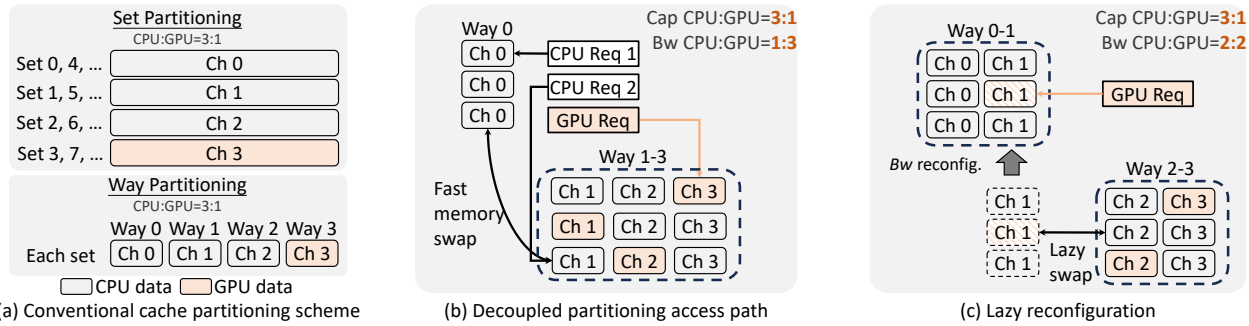


Fig. 3. Decoupled partitioning for fast memory capacity and bandwidth. (a) illustrates the conventional set/way-partitioning schemes by partitioning channels. (b) shows our novel decoupled partitioning scheme and the CPU/GPU data access paths. (c) depicts that minimum data blocks are swapped in a lazy manner, when switching the bandwidth ratio from 3:1 to 2:2.

a predetermined ratio. Fig. 3(a) shows two straightforward designs, where we map the partitioned way/set dimension to different channels, and partition the channels between the CPU and the GPU based on the desired ratio (e.g., CPU:GPU = 3:1; we discuss how to determine the best ratio in Section IV-C). While this approach is simple and ensures strong isolation by making the CPU and GPU access independent channels, it has a key drawback: the partitioning ratios of capacity and bandwidth are *coupled* and must be the same, corresponding to the number of allocated channels. This mismatches with Insights 1 & 2 in Section III-B, where the CPU should get more capacity and the GPU should get more bandwidth. Alternatively, we can map the partitioned dimension of ways/sets to each channel, and partition within each channel, i.e., sharing each channel between the CPU and GPU and internally applying partitioning. However, severe bandwidth interference would occur in the same channel, where GPU accesses may overwhelmingly dominate CPU accesses. Furthermore, partitioning within a channel requires significant hardware changes to the DRAM row buffer policy, the memory controller queue, and other hardware components of the channel.

In Hydrogen, we adopt a *decoupled* partitioning approach enabled by a novel mapping scheme from ways/sets to channels, as shown in Fig. 3(b). Hydrogen associates ways to channels, and uses way-partitioning because it is easier to implement [55]. Nevertheless, our decoupled scheme also works with set-partitioning, as later discussed in Section IV-F. First, to ensure strong bandwidth isolation, we partition the channels to the CPU and GPU based on the bandwidth ratio (e.g., CPU:GPU = 1:3). Hence the CPU enjoys dedicated channels (way 0) without suffering from GPU access contention. Then, to offer more capacity to the CPU (capacity CPU:GPU $>$ bandwidth CPU:GPU as in Section III-B), we allocate space from the non-CPU-dedicated (i.e., shared) channels (ways 1 to 3) to the CPU, while ensuring that the GPU can still utilize all channels' bandwidth. To do so, we assign certain ways in each set to the GPU, but use different ways (channels) in different sets. Therefore, GPU accesses to different sets will go to different channels and enjoy full bandwidth. Although CPU accesses still consume some bandwidth from these channels, it is not a big issue due to two reasons. First, the GPU typically uses much more bandwidth, and the CPU bandwidth is a very small portion. Second, we let the CPU-dedicated channels and the CPU shares in the shared channels effectively form a two-level hierarchy, where the hottest CPU data are swapped into the dedicated channels and kept there (called *fast memory swap*), similar to standard cache replacement. This swap traffic is light, as quantitatively analyzed in Section VI-B.

B. Token-Based Migration for Slow Memory

Now we turn to the slow memory bandwidth. In hybrid memory systems, each miss to the slow memory would trigger a block-level replacement (for the cache mode) or swap (for the flat mode) to the fast memory, collectively referred to as *migration*. Migration can amplify traffic from 64 B to 256 B, thus heavily taxing the limited slow memory bandwidth if not bringing enough hit rate benefits. As depicted in Fig. 4, each memory access first checks whether it hits in the fast memory. If a miss occurs, both the demanding 64 B cacheline and the remaining 192 B are fetched from the slow memory, performing a refill operation in the granularity of 256 B blocks. Dirty victims also need writebacks, thereby consuming additional bandwidth. To conduct a full refill and writeback, almost two 256 B blocks are transferred from/to the slow memory, resulting in 7$\times$ data amplification over 64 B.

To address this, Hydrogen introduces a *token-based migration* mechanism to regulate slow memory migration bandwidth. With a hardware-friendly token counting design, Hydrogen determines whether the bandwidth consumption is acceptable to conduct a data migration. Note that Footprint Cache and other designs [33], [41] proposed optimizations that also save data migration cost and can be orthogonally applied here, but requiring more complex metadata management.

Recognizing that GPU workloads typically consume more bandwidth in both fast and slow memories (Section III-B), we choose to harness GPU-induced migrations in Hydrogen. The memory controller manages a hardware token counter. For each GPU-induced migration, some tokens are consumed (1 for each refill and 2 if there is also a dirty writeback or a data swap) and the counter is decremented. If the counter reaches zero, further migrations for GPU accesses would be suppressed. On the other hand, a token faucet mechanism

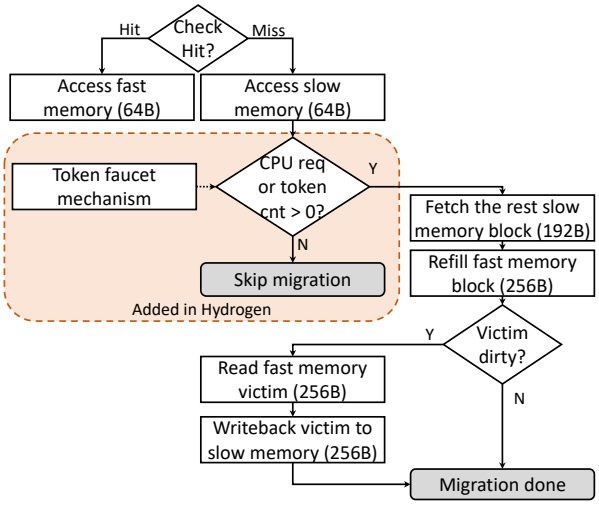


Fig. 4. Access flow with token-based migration in Hydrogen.

replenishes the counter by a certain amount periodically, e.g., every 1 million cycles. This amount specifies how many GPU-induced migrations are allowed in this period, and is adaptively adjusted to balance between CPU and GPU accesses, which is detailed in Section IV-C. Hydrogen uses a single counter for the entire slow memory bandwidth. We also tried separate per-channel counters, but there is negligible difference.

C. Epoch-Based Sampling

To enforce the end performance goal, e.g., maximizing the weighted IPC, Hydrogen adaptively determines three pivotal parameters reflecting the hybrid memory partitioning scheme: 1) `cap`, which controls the ratio of fast memory capacity allocated to the CPU and GPU; 2) `bw`, which controls the percentage of fast memory channels dedicated to the CPU, thus partitioning the fast memory bandwidth; 3) `tok`, which controls the amount of GPU-induced migrations, effectively partitioning the slow memory bandwidth. Note that `cap` and `bw` are decoupled in Hydrogen as discussed in Section IV-A.

To dynamically identify the optimal set of parameters, Hydrogen implements an online, epoch-based hill climbing algorithm. After each sampling *epoch* (e.g., 10M cycles), the hardware records the numbers of retired instructions in the CPU and GPU, respectively, and calculates the weighted IPC using the user-specified weights. Hydrogen keeps exploring different parameter choices, by iteratively selecting each of the three parameters to increase or decrease in the direction of improving the weighted IPC (i.e., hill climbing), until convergence. It then keeps this best parameter setting in the following epochs. To adapt to program phase changes, for every 500M cycles, Hydrogen starts a new parameter exploration *phase* to find the current best parameters. We evaluate the length choices of an epoch and a phase in Section VI-C.

Although previous work like TAP [40] suggested core sampling mechanisms, i.e., using different cores to simultaneously explore different competing policies, Hydrogen opts for the epoch-based sampling approach because of the much larger multi-dimensional search space, which contains many configurations and requires too many cores to effectively sample. Hill climbing places the sampling tasks along the temporal dimension, and focuses on the small regions close to the optimal point in the design space. We find that hill climbing is able to identify the optimal parameters after only a few epochs, and thus it is sufficient in our case.

D. Reconfiguration

System reconfiguration is triggered by changes in the resource partitioning parameters. Applying a new `tok` value is straightforward, done by the token faucet mechanism in the next epoch (Section IV-B). Changes to the `cap` and `bw` parameters, however, might have bigger impact involving excessive data relocation. To alleviate this cost, Hydrogen leverages *consistent hashing* [34], a method engineered to enhance the congruence between the previous and current mappings. This approach effectively reduces the requirement for extensive data movements or invalidations during the reconfiguration.

Consider a scenario where B out of N channels are allocated to the CPU, dictated by the `bw` parameter, and C ways in each N-way set are designated for CPU data, as determined by the `cap` parameter. For instance, $N = 4$, $B = 1$, and $C = 3$ in Fig. 3(b). Our problem is to select $N-C$ ways from the $N-B$ ways in the shared channels to store CPU data. This way selection is done by consistent hashing functions, so the number of reallocated ways is minimized, and the overlapped ways between the previous and current mappings are maximized to avoid relocating the data in them. These hashing functions use the set ID as a key, ensuring diverse way selection across different sets, in alignment with the requirements outlined in Section IV-A. For example, as illustrated in Fig. 3(c), adjusting the `bw` ratio from 1:3 to 2:2 necessitates only the invalidation or relocation of GPU blocks in way 1, while other ways are not impacted. Moreover, we could restrict the evaluation of these expensive hashing functions to only the reconfiguration period. After reconfiguration, the new allocations are recorded using just one extra `alloc` bit per way in the remap table entry to represent whether it is assigned to the CPU or GPU.

To reduce the reconfiguration overhead, adaptations to new `cap` and `bw` layouts are implemented in a *lazy* manner. For example, if a GPU block is reassigned from way 1 to way 2, the actual transfer of blocks is deferred, occurring off the critical path. A block is only moved or evicted when it is accessed and identified as misplaced, based on its `alloc` bit. As will be demonstrated in Section VI-B, this strategy ensures that the reconfiguration cost remains minimal.

E. Putting It All Together

With all the designs in Hydrogen, we now summarize its overall workflow. A memory access request from either the CPU or the GPU starts by looking up the remap metadata from the on-chip remap cache or the remap table in the fast memory. If the metadata indicate a hit in the fast memory, corresponding data are fetched from there. A fast memory

swap (Section IV-A) may be required in some cases, e.g., for a CPU request targeting a block outside the CPU-dedicated channels. If it is a fast memory hit but the `alloc` bit of the way mismatches with the actual request type, e.g., a GPU-requested block in a CPU-dedicated way, it indicates a need for lazy reconfiguration (Section IV-D), which invalidates the misplaced block after the access. If a miss occurs, we should access the slow memory and try to migrate data into the fast memory. If it is a GPU request, we need to first check and update the token counter (Section IV-B), and bypass the migration if there is no available token for the slow memory bandwidth. The token counter is replenished according to the token faucet mechanism. Finally, we periodically trigger epoch-based reconfiguration (Section IV-C), to determine whether to dynamically adjust the resource partitioning.

F. Discussion

Hardware cost. The implementation of Hydrogen necessitates only minor hardware changes and incurs moderate overheads. Specifically, the hardware uses several registers to maintain the current `cap`, `bw`, and `tok` configurations and the system throughput. The hill climbing algorithm further requires the storage for another set of configurations to compare the current and the previous performance, along with some logic to conduct exploration and optimization. To facilitate decoupled partitioning of the fast memory, the hardware also records both the dedicated and shared channels. All the above changes only need some registers and simple logic. For lazy reconfiguration, one extra `alloc` bit is added for each block in the remap table in the fast memory. This would add a tiny 0.049% metadata storage overhead.

Decoupled set-partitioning. It is also possible to implement a decoupled set-partitioning scheme, analogous to the proposed way-partitioning scheme in Hydrogen. In this design, cache sets are statically interleaved across different fast memory channels. Similar to Fig. 3(b), the sets in a certain number of channels (e.g., 4 out of 16 channels) are dedicated to CPU data, while the rest are shared between the CPU and GPU with an interleaving scheme that spreads GPU data across all shared channels. The OS and the GPU runtime regulate memory allocation requests from the CPU and GPU with the page coloring technique [11], so that CPU and GPU data are correspondingly mapped into the designated sets. Although this design supports set-partitioning, it also inherits the typical drawbacks such as high repartitioning overheads and OS-level modifications.

Flat mode support. Different from the cache mode, the blocks are initially placed in the fast memory until the fast memory is used up, i.e., the first touch policy. For each data migration, we swap the slow memory block and a victim block in the fast memory, within the same set. Since we adopt the same set-associative layout, the decoupled fast memory partitioning works as in Section IV-A. We always decrement the token counter by 2 as each swap contains read and write. This makes the system more cautious about data migration.

TABLE I
SYSTEM CONFIGURATIONS.

CPU	8 cores
CPU L1	8-way, 64 kB per core, 64 B cachelines, LRU
CPU L2	8-way, 1 MB per core, 9-cycle latency, LRU
GPU	96 Execution Units
GPU L1	128 kB per 16 units
Shared LLC	16-way, 16 MB shared, 38-cycle latency, LRU
Fast memory	HBM2E, 16 channels × 1 rank × 16 banks; 1600 MHz, RCD-CAS-RP: 23-23-23; RD/WR: 6.4 pJ/bit, ACT/PRE: 15 nJ
Slow memory	DDR4-3200, 4 channels × 2 ranks × 16 banks; 1600 MHz, RCD-CAS-RP: 22-22-22; RD/WR: 33 pJ/bit, ACT/PRE: 15 nJ

V. METHODOLOGY

Simulated configuration. We use zsim [59] to evaluate Hydrogen on an 8-core CPU + 96-EU GPU system as detailed in Table I. We use HBM2E and DDR4 as the fast and slow memories, respectively. The specifications of HBM2E and DDR4 are extracted from recent literature [9], [29], [30]. We set the slow memory capacity to cover the workload footprint so no application suffers from page faults. The fast memory has 1/8 of the slow memory capacity, as in previous hybrid memory designs [41], [65]. We use CACTI 7.0 [8] to model SRAM structures. We use a 256 B block size and 4-way cache mode by default, and evaluate other granularity and associativity configurations in Section VI-C.

Workloads. We randomly combine CPU and GPU workloads into 12 combinations listed in Table II. The CPU workloads are chosen from the memory-intensive workloads in SPEC CPU2017 [48]. We fast-forward over the initialization phase and simulate the memory access traces of 5 billion instructions when executed in the rate mode with 2 copies for each CPU workload. The GPU workloads include several Rodinia benchmarks [10] and a BERT model inference task from MLPerf [56]. We only simulate the memory accesses from the GPU kernels. The default IPC weights for our performance goal are CPU:GPU = 12:1, following their core count ratio, to make their accesses equally important. We evaluate other weights in Fig. 10.

Baselines. Our baseline follows Fig. 1 without any partitioning. We also compare with HAShCache [49] and Profess [36], two recent partitioning designs for hybrid memory. HAShCache adopts a direct-mapped cache organization and uses chaining to support pseudo-associativity. We modify Profess to support the cache mode and 4-way associativity, and port it to the same HBM+DDR hybrid memory configuration as the other systems for fair comparison. We also implement a simple way-partitioning scheme, denoted as WayPart, which does not use our decoupled technique, and dedicates 75% of the ways to the CPU workloads. All designs use the same block size and a 256 kB on-chip remap cache in the memory controller.

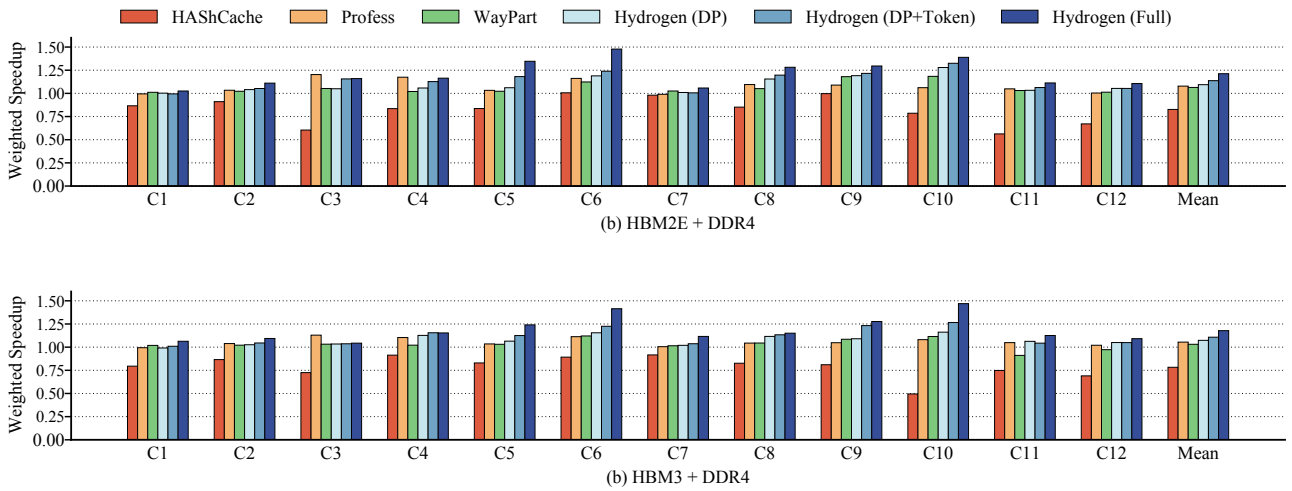


Fig. 5. Performance comparison between HAShCache, Profess, WayPart, and several Hydrogen variants. Normalized to the baseline without any partitioning.

TABLE II
WORKLOAD COMBINATIONS.

	CPU Workloads	GPU Workloads
C1	gcc-mcf-lbm-roms	backprop
C2	omnetpp-lbm-gcc-xz	backprop
C3	roms-mcf-deepsjeng-cactusBSSN	hotspot
C4	lbm-fotonik3d-deepsjeng-omnetpp	lud
C5	roms-lbm-deepsjeng-fotonik3d	streamcluster
C6	omnetpp-xz-roms-deepsjeng	pathfinder
C7	bwaves-gcc-xz-fotonik3d	needle
C8	fotonik3d-gcc-omnetpp-deepsjeng	bfs
C9	mcf-cactusBSSN-roms-deepsjeng	srad
C10	deepsjeng-xz-roms-bwaves	pathfinder
C11	omnetpp-gcc-fotonik3d-lbm	bert
C12	mcf-gcc-cactusBSSN-omnetpp	bert

VI. EVALUATION

We first present the overall comparison results between Hydrogen and the baseline designs. We then conduct a detailed analysis to illustrate the benefits and overheads of Hydrogen. Finally, we extend our evaluation to more configurations to demonstrate the generality of Hydrogen.

A. Overall Comparison

Fig. 5(a) presents the overall performance of different designs, using HBM2E and DDR4 as the fast and slow memory tiers. Hydrogen (Full) outperforms the non-partitioned baseline by $1.24\times$ on average, and up to $1.48\times$. We analyze the other variants of Hydrogen in Section VI-B. HAShCache does not perform well, mainly due to the limited hit rates from its direct-mapped organization, which stress the slow memory bandwidth. Profess works well for certain workloads (e.g., C3 and C4), but it only prevents improper migration, without decoupling the fast memory resource partitioning to match the different requirements of CPU and GPU workloads. These workloads are less sensitive to the partitioning scheme, and their access patterns are friendly to Profess' MDM replacement policy. If we use LRU in Profess, Hydrogen would perform better. Similarly, WayPart also does not use decoupled partitioning and only shows limited benefits. Also its fixed resource partitioning ratio between the CPU and GPU is suboptimal for some workload combinations. Moreover, previous designs do not iteratively optimize their policies as Hydrogen does. Compared to Profess, the best one among these baselines, Hydrogen is $1.16\times$ faster on average and up to $1.31\times$. Compared to HAShCache, Hydrogen has a speedup of $1.47\times$ on average and up to $1.98\times$.

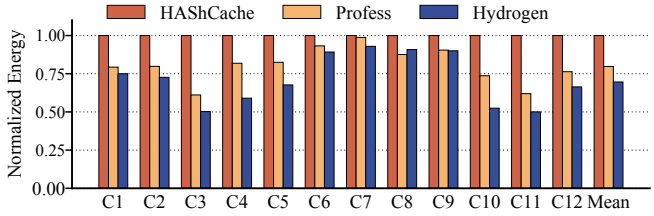
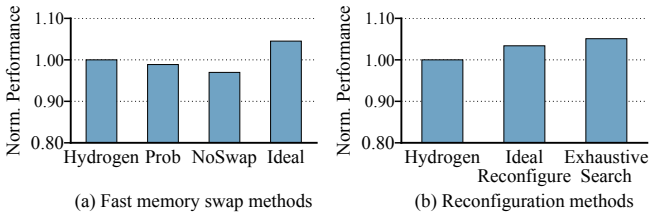


Fig. 6. Memory energy comparison between HAShCache, Profess, and Hydrogen. Normalized to HAShCache.

Furthermore, we also show the performance results in Fig. 5(b) when increasing the fast memory bandwidth by adopting the more recent HBM3 memory [31], with doubled bandwidth and scaled timing parameters. Hydrogen outperforms the best baseline, Profess, by $1.12\times$ on average. The speedups are smaller with HBM3, because higher fast memory bandwidth makes bandwidth partitioning less critical, while capacity partitioning remains important. If we have more CPU/GPU cores or use more bandwidth-intensive workloads, the speedups would become more significant.

Fig. 6 further illustrates the memory energy consumption. Overall Hydrogen achieves on average 31% energy reduction compared to HAShCache, and up to 50% on C11, thanks to the optimized memory partitioning schemes that both improve



(a) Fast memory swap methods (b) Reconfiguration methods

Fig. 7. Performance impact of (a) fast memory swap methods (b) reconfiguration overheads. Geomeans of all workloads are shown.

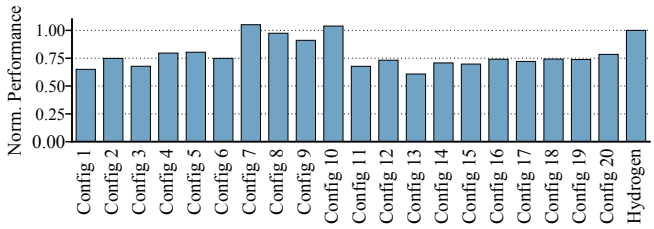
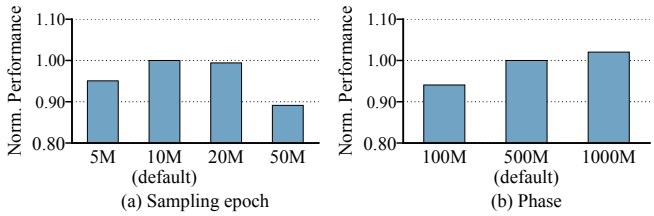


Fig. 8. Performance of some of the exhaustive search configurations and the one found by Hydrogen, with C5. Normalized to Hydrogen.



(a) Sampling epoch (b) Phase

Fig. 9. Sensitivity studies on (a) phase length and (b) sampling epoch length. Geomeans of all workloads are shown.

performance and reduce expensive slow memory accesses. In C11, the significant energy saving matches the high performance gain in Fig. 5, where the 30% speedup translates to 26% static DRAM energy reduction.

B. Analysis of Benefits and Overheads

Performance breakdown. We further evaluate the individual techniques used in Hydrogen by separating their performance contributions in Fig. 5(a). Due to space limits we will primarily analyze the HBM2E results. Hydrogen (DP) only enables decoupled partitioning for the fast memory but with a fixed partitioning scheme that allocates 75% fast memory bandwidth and 25% fast memory capacity to GPU workloads. It achieves an average 1.10× speedup. A more detailed investigation shows that CPU workloads show similar performance in Hydrogen (DP) and simple WayPart. This is expected as WayPart uses 75% capacity for CPU workloads, the same as Hydrogen (DP). However, Hydrogen (DP) has the benefit of offering sufficient bandwidth to GPU workloads. For example in C10, compared with the baseline, the GPU performance decreases to 23% in WayPart, while keeping at 87% in Hydrogen (DP).

Then, Hydrogen (DP+Token) further enables the token-based migration for the slow memory. Here we allow a fixed ratio of 15% requests to migrate in each epoch, where the threshold is heuristically set based on the bandwidth ratio of the fast and slow memories. Regulating migration gives us another 4.4% speedup over Hydrogen (DP). It is particularly effective for those combinations seeing excessive memory accesses from GPUs, e.g., a 12% improvement for C5 with `streamcluster`.

So far all the resource partitioning ratios are manually and heuristically set. Finally, we add the hill climbing algorithm to automatically explore the design space and select the optimized configurations. This technique shows another 8.6% speedup. For some workloads, e.g., C5, the manually specified configurations are suboptimal due to workload-specific memory access patterns. After our systematic optimization, C5 shows a 14% improvement.

Fast memory swap overheads. Fig. 7(a) compares several design choices for fast memory swaps, i.e., how to migrate data from non-CPU-dedicated channels to CPU-dedicated channels (Section IV-A). The *Ideal* design assumes zero cost for the migration, but it only brings a 4.5% speedup over our method. This is because there are only 11.9% of the CPU accesses requiring fast memory swaps, given that most CPU hot data are well cached in the dedicated channels. We also try another probability-based method *Prob* that bypasses half of the fast memory swaps. This technique decreases the end-to-end performance by 1.2% due to higher contention in the non-CPU-dedicated channels. Not doing fast memory swaps at all (*NoSwap*) brings even more degradation, with 4% on average and up to 5.1%. These results show that fast memory swaps are generally beneficial to the overall performance, and have only minor overheads even compared to the ideal case.

Reconfiguration overheads and sampling effectiveness. From Fig. 7(b) we see that the reconfiguration in Hydrogen causes only 3.2% performance loss compared to an ideal reconfigure design that can instantly switch to the new configuration. This demonstrates the effectiveness of our consistent hashing mapping and lazy layout updates (Section IV-D). Moreover, our epoch-based sampling is effective. If we exhaust all configurations offline and choose the best one, it only outperforms our online hill climbing algorithm by 5.1%, including the reconfiguration overheads. We show some of the exhaustive search results for C5 as an example in Fig. 8. Different configurations affect performance, with the optimal one being 73% faster than the median. In addition, Hydrogen effectively chooses a good configuration, within 96.1% of the optimal choice.

C. Sensitivity Studies

Sampling epoch lengths and phase lengths. Fig. 9 shows the effects of using different sampling epoch length and phase length values (Section IV-C). Using 10 million cycles for a sampling epoch balances the sampling cost and the adaptability. If the sampling epoch is too short, the reconfiguration overheads will become non-negligible, reducing the end-to-

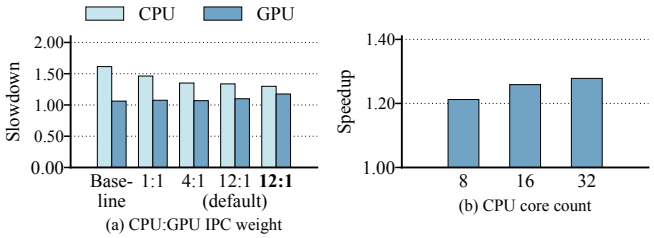


Fig. 10. (a) Impact of different CPU:GPU IPC weights, with C6. Lower slowdown means better performance. (b) Impact of CPU core counts. Speedups are normalized to the baseline without any partitioning.

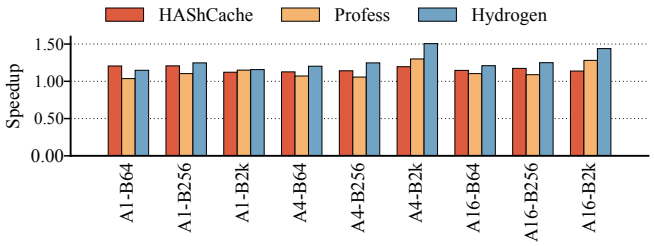


Fig. 11. Impact of different associativities (A) and block sizes (B). Normalized to the baseline without any partitioning of the same configuration.

end performance by over 5%. If the sampling epoch is too long, the reconfiguration opportunities are limited, and the performance is also sub-optimal. We empirically find that 20 optimization steps are required before convergence. For the phase length, our evaluated workloads do not have frequent behavior changes, so short phases incur unnecessary reconfigurations. The OS can optionally turn off phase reconfiguration if it knows the workloads are stable. To retain flexibility, we set a default value of 500 million cycles.

IPC weights. Fig. 10(a) uses different CPU:GPU IPC weights for the end performance goal, from 1:1 to 32:1 on C6. Higher ratios prioritize CPUs, and reduce the CPU slowdown from 1.61× to 1.30×. But the GPU slowdown slightly increases from 1.06× to 1.18×. Users can flexibly set different weights according to their needs. All settings outperform the non-partitioned baseline.

Core counts. Fig. 10(b) increases the CPU cores while keeping 96 GPU cores, with the IPC weights also changing according to the core count ratio. Increasing CPU cores emphasizes the importance of bandwidth and capacity partitioning. However, more CPU cores reduce the GPU's impact, so partitioning more resources to the CPU yields less significance, similar to the trend in Fig. 10(a).

Block sizes and associativities. Fig. 11 scales the associativity (A) from 1 to 16, and the block size (B) from 64 B to 2 kB. The chaining technique in the original HAShCache design enables pseudo-associativity to cache more data blocks, but it only works with the direct-mapped organization. To scale HAShCache to higher associativities, we disable chaining and add corresponding tag access latencies. Hydrogen exhibits consistent speedups across different configurations, except A1-B64. In direct-mapped scenarios, HAShCache slightly outperforms Profess and Hydrogen, because of its chaining optimization mentioned above. Hydrogen outperforms Profess by properly partitioning the fast memory to leverage more associativities. For large block sizes with high data migration cost, Hydrogen outperforms Profess by effectively controlling the migration rate when the bandwidth resources are limited.

VII. Conclusions

This paper presents Hydrogen, a hybrid memory management scheme for heterogeneous CPU-GPU systems. It effectively partitions fast memory capacity, fast memory bandwidth, and slow memory bandwidth to increase system-level performance isolation. It achieves on average 1.16× and up to 1.31× speedups over state-of-the-art designs.

Acknowledgment

The authors thank the anonymous reviewers and the shepherd for their valuable suggestions, and the Tsinghua IDEAL group members for constructive discussion. This work was supported by the National Natural Science Foundation of China (62072262). Mingyu Gao is the corresponding author.

References

[1] AMD, "Fusion," 2011, http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx.

[2] AMD, "AMD Delivers Leadership Portfolio of Data Center AI Solutions with AMD Instinct MI300 Series," 2023, https://www.amd.com/en/newsroom/press-releases/2023-12-6-amd-delivers-leadership-portfolio-of-data-center-a.html.

[3] AMD, "AMD Infinity Hub," 2023, https://www.amd.com/en/developer/resources/infinity-hub.html.

[4] AMD, "AMD Instinct MI300A Accelerators," 2023, https://www.amd.com/en/products/accelerators/instinct/mi300/mi300a.html.

[5] Apple, "Apple Unleashes M1," 2020, https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/.

[6] Arm, "Arm big.LITTLE technology," 2011, https://www.arm.com/technologies/big-little.

[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS Parallel Benchmarks - Summary and Preliminary Results," in *1991 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 1991, pp. 158–165.

[8] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 14:1–14:25, 2017.

[9] N. Chatterjee, M. O'Connor, D. Lee, D. R. Johnson, S. W. Keckler, M. Rhu, and W. J. Dally, "Architecting an Energy-Efficient DRAM System for GPUs," in *23rd International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2017, pp. 73–84.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, 2009, pp. 44–54.

[11] S. Cho and L. Jin, "Managing Distributed, Shared L2 Caches through OS-Level Page Allocation," in *39th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2006, pp. 455–468.

[12] C. Chou, A. Jaleel, and M. K. Qureshi, "CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache," in *47th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2014, pp. 1–12.

[13] C. Chou, A. Jaleel, and M. K. Qureshi, "BATMAN: Techniques for Maximizing System Bandwidth of Memory Systems with Stacked-DRAM," in *3rd Internatinoal Symposium on Memory Systems (MEMSYS)*. ACM, 2017, pp. 268–280.

[14] H. Esmaeilzadeh, E. R. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *38th International Symposium on Computer Architecture (ISCA)*. ACM, 2011, pp. 365–376.

[15] A. Garcia-Garcia, J. C. Saez, and M. Prieto, "Contention-Aware Fair Scheduling for Asymmetric Single-ISA Multicore Systems," *IEEE Transactions on Computers*, vol. 67, no. 12, pp. 1703–1719, 2018.

[16] N. D. Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan, "Bi-Modal DRAM Cache: Improving Hit Rate, Hit Latency and Bandwidth," in *47th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2014, pp. 38–50.

[17] F. T. Hady, A. P. Foong, B. Veal, and D. Williams, "Platform Storage Performance With 3D XPoint Technology," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1822–1833, 2017.

[18] M. T. Heath, *Scientific Computing: an Introductory Survey*. SIAM, 2018.

[19] J. Hestness, S. W. Keckler, and D. A. Wood, "A Comparative Analysis of Microarchitecture Effects on CPU and GPU Memory System Behavior," in *International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, 2014, pp. 150–160.

[20] C. Huang and V. Nagarajan, "ATCache: Reducing DRAM Cache Latency via a Small SRAM Tag Cache," in *23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2014, pp. 51–60.

[21] Intel, "Sandy Bridge," 2011, http://software.intel.com/en-us/articles/sandy-bridge/.

[22] Intel, "Intel Resource Director Technology (Intel RDT)," 2016, https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html.

[23] Intel, "Intel Xeon Phi Coprocessor x200 Product Family," 2017, https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-phi-coprocessor-x200-family-datasheet.pdf.

[24] Intel, "Intel Iris Xe GPU Architecture," 2020, https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2023-0/intel-iris-xe-gpu-architecture.html.

[25] Intel, "Intel Optane DC Persistent Memory," 2020, https://builders.intel.com/docs/networkbuilders/intel-optane-dc-persistent-memory-telecom-use-case-workloads.pdf.

[26] Intel, "Intel Hybrid Architecture (code name Alder Lake)," 2022, https://www.intel.com/content/www/us/en/developer/articles/technical/hybrid-architecture.html.

[27] Intel, "4th Gen Intel Xeon Processor Scalable Family, sapphire rapids," 2023, https://www.intel.com/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.html.

[28] H. Jang, Y. Lee, J. Kim, Y. Kim, J. Kim, J. Jeong, and J. W. Lee, "Efficient Footprint Caching for Tagless DRAM Caches," in *22nd International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2016, pp. 237–248.

[29] JEDEC, "DDR4 SDRAM Standard," 2021, https://www.jedec.org/standards-documents/docs/jesd-79-4a.

[30] JEDEC, "High Bandwidth Memory (HBM2) DRAM," 2021, https://www.jedec.org/standards-documents/docs/jesd235a.

[31] JEDEC, "High Bandwidth Memory (HBM3) DRAM," 2023, https://www.jedec.org/standards-documents/docs/jesd238a.

[32] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache," in *47th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2014, pp. 25–37.

[33] D. Jevdjic, S. Volos, and B. Falsafi, "Die-Stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache," in *40th International Symposium on Computer Architecture (ISCA)*. ACM, 2013, pp. 404–415.

[34] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," in *29th Annual ACM Symposium on the Theory of Computing (ToC)*. ACM, 1997, pp. 654–663.

[35] D. Kim, S. Lee, J. Chung, D. Kim, D. H. Woo, S. Yoo, and S. Lee, "Hybrid DRAM/PRAM-Based Main Memory for Single-Chip CPU/GPU," in *49th Annual Design Automation Conference (DAC)*. ACM, 2012, pp. 888–896.

[36] D. Knyaginin, V. Papaefstathiou, and P. Stenström, "ProFess: A Probabilistic Hybrid Main Memory Management Framework for High Performance and Fairness," in *24th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2018, pp. 143–155.

[37] O. R. N. Laboratory, "Update on Frontier Exascale System and Early Science," 2022, https://science.osti.gov/-/media/ascr/ascac/pdf/meetings/202203/ASCAC_202203-Geist.pdf.

[38] A. Labrinidis and H. V. Jagadish, "Challenges and Opportunities with Big Data," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 2032–2033, 2012.

[39] Y. LeCun, Y. Bengio, and G. E. Hinton, "Deep Learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[40] J. Lee and H. Kim, "TAP: A TLP-Aware Cache Management Policy for a CPU-GPU Heterogeneous Architecture," in *18th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2012, pp. 91–102.

[41] Y. Li and M. Gao, "Baryon: Efficient Hybrid Memory Management with Compression and Sub-Blocking," in *29th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 137–151.

[42] G. H. Loh and M. D. Hill, "Efficiently Enabling Conventional Block Sizes for Very Large Die-Stacked DRAM Caches," in *44th International Symposium on Microarchitecture (MICRO)*. ACM, 2011, pp. 454–464.

[43] P. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi, "S12 - The HPC Challenge (HPCC) Benchmark Suite," in *2006 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM Press, 2006, p. 213.

[44] V. Mekkat, A. Holey, P. Yew, and A. Zhai, "Managing Shared Last-Level Cache in a Heterogeneous Multicore Processor," in *22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE Computer Society, 2013, pp. 225–234.

[45] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-Stacked and Off-Package Memories," in *21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2015, pp. 126–136.

[46] NVIDIA, "NVIDIA Pascal Architecture," 2016, https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture/.

[47] NVIDIA, "NVIDIA Ampere Architecture," 2020, https://www.nvidia.com/en-us/data-center/ampere-architecture/.

[48] R. Panda, S. Song, J. Dean, and L. K. John, "Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon?" in *24th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2018, pp. 271–282.

[49] A. Patil and R. Govindarajan, "HAShCache: Heterogeneity-Aware Shared DRAMCache for Integrated Heterogeneous Systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 4, pp. 51:1–51:26, 2017.

[50] A. Prodromou, M. R. Meswani, N. Jayasena, G. H. Loh, and D. M. Tullsen, "MemPod: A Clustered Architecture for Efficient and Scalable Migration in Flat Address Space Multi-level Memories," in *23rd International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2017, pp. 433–444.

[51] M. K. Qureshi and G. H. Loh, "Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design," in *45th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2012, pp. 235–246.

[52] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *39th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2006, pp. 423–432.

[53] S. Rai and M. Chaudhuri, "Exploiting Dynamic Reuse Probability to Manage Shared Last-level Caches in CPU-GPU Heterogeneous Processors," in *30th International Conference on Supercomputing (ICS)*. ACM, 2016, pp. 3:1–3:14.

[54] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page Placement in Hybrid Memory Systems," in *25th International Conference on Supercomputing (ICS)*. ACM, 2011, pp. 85–95.

[55] P. Ranganathan, S. V. Adve, and N. P. Jouppi, "Reconfigurable Caches and their Application to Media Processing," in *27th International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 2000, pp. 214–224.

[56] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "MLPerf Inference Benchmark," *arXiv preprint arXiv:1911.02549*, Nov 2019.

[57] J. H. Ryoo, M. R. Meswani, A. Prodromou, and L. K. John, "SILC-FM: Subblocked InterLeaved Cache-Like Flat Memory Organization," in *23rd International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2017, pp. 349–360.

[58] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-grain Cache Partitioning," in *38th International Symposium on Computer Architecture (ISCA)*. ACM, 2011, pp. 57–68.

[59] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *40th International Symposium on Computer Architecture (ISCA)*. ACM, 2013, pp. 475–486.

[60] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent Hardware Management of Stacked DRAM as Part of Memory," in *47th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2014, pp. 13–24.

[61] T. N. Theis and H. P. Wong, "The End of Moore's Law: A New Beginning for Information Technology," *Computing in Science and Engineering*, vol. 19, no. 2, pp. 41–50, 2017.

[62] K. Varadarajan, S. K. Nandy, V. Sharda, B. Amrutur, R. R. Iyer, S. Makineni, and D. Newell, "Molecular Caches: A Caching Structure for Dynamic Creation of Application-Specific Heterogeneous Cache Regions," in *39th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2006, pp. 433–442.

[63] E. Vasilakis, V. Papaefstathiou, P. Trancoso, and I. Sourdis, "Decoupled Fused Cache: Fusing a Decoupled LLC with a DRAM Cache," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 4, pp. 65:1–65:23, 2019.

[64] E. Vasilakis, V. Papaefstathiou, P. Trancoso, and I. Sourdis, "LLC-Guided Data Migration in Hybrid Memory Systems," in *33rd IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 932–942.

[65] E. Vasilakis, V. Papaefstathiou, P. Trancoso, and I. Sourdis, "Hybrid2: Combining Caching and Migration in Hybrid Memory Systems," in *26th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 649–662.

[66] T. Vijayaraghavan, Y. Eckert, G. H. Loh, M. J. Schulte, M. Ignatowski, B. M. Beckmann, W. C. Brantley, J. L. Greathouse, W. Huang, A. Karunanithi, O. Kayiran, M. R. Meswani, I. Paul, M. Poremba, S. Raasch, S. K. Reinhardt, G. Sadowski, and V. Sridharan, "Design and Analysis of an APU for Exascale Computing," in *23rd International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2017, pp. 85–96.

[67] Y. Xie and G. H. Loh, "PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches," in *36th International Symposium on Computer Architecture (ISCA)*. ACM, 2009, pp. 174–183.

[68] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas, "Banshee: Bandwidth-Efficient DRAM Caching via Software/Hardware Cooperation," in *50th International Symposium on Microarchitecture (MICRO)*. ACM, 2017, pp. 1–14.

[69] J. Zhan, O. Kayiran, G. H. Loh, C. R. Das, and Y. Xie, "OSCAR: Orchestrating STT-RAM Cache Traffic for Heterogeneous CPU-GPU Architectures," in *49th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2016, pp. 28:1–28:13.

[70] H. Zhang, G. Chen, B. C. Ooi, K. Tan, and M. Zhang, "In-Memory Big Data Management and Processing: A Survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 7, pp. 1920–1948, 2015.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

The artifact contains the implementation of Hydrogen, a set of approaches for contention-aware hybrid memory management for CPU-GPU heterogeneous systems. We make the following contributions in this paper.

- C_1: We present a comprehensive analysis of the unique demands and challenges of integrating hybrid memory architectures with heterogeneous CPU-GPU computing systems, highlighting the importance of carefully partitioning the capacity and bandwidth of both the fast and slow memories, in order to meet the distinct preferences of CPU and GPU workloads.
- C_2: We introduce Hydrogen, a novel hardware architecture designed to optimize the allocation of hybrid memory resources across CPUs and GPUs. Hydrogen uniquely decouples the bandwidth and capacity partitioning of the fast memory, and implements a token-based migration strategy for efficient slow memory bandwidth allocation, tailored to the specific needs of CPU and GPU workloads.
- C_3: We develop an epoch-based online sampling technique combined with a hill climbing search algorithm integrated into Hydrogen, enabling dynamic adaptation to the varying behaviors of applications over time. These techniques allow Hydrogen to efficiently explore a large, multi-dimensional design space for optimized memory resource partitioning.
- C_4: We demonstrate through extensive evaluation that Hydrogen significantly outperforms existing hybrid memory management approaches, achieving on average a $1.16\times$ speedup, and in certain cases up to $1.31\times$, across a variety of memory-intensive workloads. These improvements are attributed to Hydrogen's comprehensive and efficient partitioning schemes, as well as its adaptability to application-specific behaviors.

Moreover, we detail the methodologies for obtaining or generating the datasets utilized in our study, along with the scripts for replicating the experiments described in the paper.

B. Computational Artifacts

We use A_1 as our computational artifact to this paper.

The following table outlines the artifacts in relation to the paper's contributions and specifies the elements within the paper that can be reproduced using each artifact.

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Tables I-II and Figure 2
	C_2-C_4	Figures 5-11

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

The artifact A_1 contains two components related to the contributions. A_1 includes a trace-based simulator framework for an integrated CPU-GPU system, and trace generators for the SPEC CPU2017, Rodinia, and MLPerf benchmarks. This provides the analysis in C_1 and the baseline performance in C_4. Furthermore, A_1 modifies the memory controller in the simulator to incorporate C_2 and C_3, showcasing the performance advantages of Hydrogen over existing hybrid memory partitioning designs HAShCache and Profess.

Expected Results

Hydrogen with C_2 and C_3 designs should have higher weighted speedups than the baseline designs. By applying the specified hardware configurations in the simulation, Hydrogen outperforms the non-partitioned baseline by $1.24\times$ on average, and up to $1.48\times$. Compared to the Profess baseline, Hydrogen is $1.16\times$ faster on average and up to $1.31\times$. Compared to the HAShCache baseline, Hydrogen is $1.47\times$ faster on average and up to $1.98\times$.

Expected Reproduction Time (in Minutes)

The artifact's execution time varies by workloads, generally spanning 20 to 40 minutes for trace generation and hundreds of minutes for complete simulation, offering a practical window for replication and analysis.

Artifact Setup (incl. Inputs)

Hardware: A_1 requires Intel CPUs for execution. We specifically use dual 2.20 GHz Intel Xeon Gold 5120 CPUs with 28 logical cores per CPU, and 256 GB RAM. The trace generators for Rodinia and MLPerf require NVIDIA GPUs. NVIDIA GeForce RTX 3090 is used in this paper.

Software:

- Run-time environment: The experiments in this paper use Ubuntu 22.04.3 LTS for all tools, and CUDA Toolkit version 11.3 for Rodinia and MLPerf trace generation.
- Dependencies: A_1 requires Intel Pin 3.11, scons 3.1.1, libconfig 1.7.1, libhdf5 1.10.5, and zlib 1.2.11.

Datasets / Inputs:

- We use SPEC CPU2017 as the CPU workloads. It is *not available* for download. Official website.
- We use Rodinia 3.1 and MLPerf Inference Benchmark v3.0 as the GPU workloads. They are available here and here.

Installation and Deployment:

- Compilation: The experiments in the paper use gcc 8.4.0 to compile the code. Use `make -C traces/` and `make -C sims/` to build the trace generators and the simulator, respectively.

Artifact Execution

We evaluate the end-to-end performance of Hydrogen using real-world CPU and GPU workloads. This includes three tasks. T_1 generates the workload traces, which are used as the input to the simulation task T_2. The output of T_2 is processed by another task T_3, which produces CSV tables as the final results.

T_1: We generate the workload traces and configure the simulated hardware. The `./traces/generate_overall_spec_workload` script generates all SPEC CPU2017 workload traces. We skip the initialization phase and simulate the memory access traces of 5 billion instructions. The `./traces/generate_overall_rodinia_workload` script generates all Rodinia GPU workload traces. The `./traces/generate_overall_mlperf_workload` script generates all MLPerf GPU workload traces. For Rodinia and MLPerf, we only simulate the memory access traces from the GPU kernels.

T_2: The simulation configuration templates are provided in the `sims/` directory, containing `baseline`, `hashcache`, `profess`, and `hydrogen`. We specify the trace file paths in `sys.logic.cpu.traceFilePrefix` and `sys.logic.gpu.traceFilePrefix`, and simulate different designs using commands like `sims/build/opt/zsim sims/baseline/zsim.cfg`.

T_3: After simulation, we gather performance statistics from the simulator, e.g., the execution time and the cache hit rate. The command `./extract_performance.py sims/` lists all experiments in T_2, and extracts all critical metrics, including CPU and GPU performance, into a final CSV table `perf.csv`.

Artifact Analysis (incl. Outputs)

The overall performance is summarized in the `perf.csv` file in the CSV format. Each entry shows the CPU and GPU cycles of one workload combination under one design. We calculate the CPU and GPU speedups by dividing the cycles with the baseline cycles. Then the overall speedup is weighted summed using the CPU and GPU speedups. Each entry has one weighted speedup, which corresponds to one bar in Figure 5. We are supposed to see that Hydrogen outperforms the baseline and other state-of-the-art designs in terms of the weighted speedups. The results correspond to the numbers reported in Figure 5. Figure 5 is the most important result, containing the performance breakdown and performance comparison between Hydrogen and other designs. Numbers in other figures can be generated similarly.