MemHC: An Optimized GPU Memory Management Framework for Accelerating Many-body Correlation

QIHAN WANG, ZHEN PENG, and BIN REN, William & Mary, USA
JIE CHEN and ROBERT G. EDWARDS, Jefferson Lab, USA

The *many-body correlation function* is a fundamental computation kernel in modern physics computing applications, e.g., Hadron Contractions in Lattice quantum chromodynamics (QCD). This kernel is both computation and memory intensive, involving a series of tensor contractions, and thus usually runs on accelerators like GPUs. Existing optimizations on many-body correlation mainly focus on individual tensor contractions (e.g., cuBLAS libraries and others). In contrast, this work discovers a new optimization dimension for many-body correlation by exploring the optimization opportunities among tensor contractions. More specifically, it targets general GPU architectures (both NVIDIA and AMD) and optimizes many-body correlation's memory management by exploiting a set of *memory allocation and communication redundancy elimination* opportunities: first, *GPU memory allocation redundancy*: the intermediate output frequently occurs as input in the subsequent calculations; second, *CPU-GPU communication redundancy*: although all tensors are allocated on both CPU and GPU, many of them are used (and reused) on the GPU side only, and thus, many CPU/GPU communications (like that in existing Unified Memory designs) are unnecessary; third, *GPU oversubscription:* limited GPU memory size causes oversubscription issues, and existing memory management usually results in near-reuse data eviction, thus incurring extra CPU/GPU memory communications.

Targeting these memory optimization opportunities, this article proposes MemHC, an optimized systematic GPU memory management framework that aims to accelerate the calculation of many-body correlation functions utilizing a series of new memory reduction designs. These designs involve optimizations for GPU memory allocation, CPU/GPU memory movement, and GPU memory oversubscription, respectively. More specifically, first, MemHC employs duplication-aware management and lazy release of GPU memories to corresponding host managing for better data reusability. Second, it implements data reorganization and on-demand synchronization to eliminate redundant (or unnecessary) data transfer. Third, MemHC exploits an optimized Least Recently Used (LRU) eviction policy called Pre-Protected LRU to reduce evictions and leverage memory hits. Additionally, MemHC is portable for various platforms including NVIDIA GPUs and AMD GPUs. The evaluation demonstrates that MemHC outperforms unified memory management by 2.18× to 10.73×. The proposed Pre-Protected LRU policy outperforms the original LRU policy by up to 1.36× improvement.[1]

[1]New Article Not an Extension of a Conference Paper.

This work is partially supported by the NSF award CCF-2047516 (CAREER), and the US Department Of Energy, Office of Science, Offices of Nuclear Physics and Advanced Scientific Computing Research, through the SciDAC program under contract DE-AC05-06OR23177 under which JSA LLC operates and manages Jefferson Lab, and under the 17-SC-20-SC Exascale Computing Project.
Authors' addresses: Q. Wang, Z. Peng, and B. Ren, William & Mary, Williamsburg, VA; emails: {qwang19, zpeng01}@email.wm.edu, bren@wm.edu; J. Chen and R. G. Edwards, Jefferson Lab, Newport News, VA; emails: {chen, edwards}@jlab.org.

1544-3566/2022/03-ART24
https://doi.org/10.1145/3506705

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; **Redundancy**; • **Computing methodologies** → **Parallel algorithms**; • **Applied computing** → *Physics*; *Computer-aided design;*

Additional Key Words and Phrases: GPU memory management, memory redundancy elimination, memory oversubscription, many-body correlation function

ACM Reference format:
Qihan Wang, Zhen Peng, Bin Ren, Jie Chen, and Robert G. Edwards. 2022. MemHC: An Optimized GPU Memory Management Framework for Accelerating Many-body Correlation. *ACM Trans. Arch. Code Optim.* 19, 2, Article 24 (March 2022), 26 pages.
https://doi.org/10.1145/3506705

1 INTRODUCTION

Many-body correlation functions are widely used in scientific physics systems, such as Lattice **quantum chromodynamics (QCD)** [9–11]. Correlation function calculation is critical for physics observables (e.g., predicting properties of light nuclei [14]) and is broadly explored in Jefferson Lab (Jlab), Facility for Antiproton and Ion Research in Europe (FAIR), and Japan Proton Accelerator Research Complex (J-PARC) facilities [6]. A typical instance of many-body correlation is the *hadronic correlation function* in complex many-particle systems, involving quarks and gluons enclosed in mesons and baryons. Hadronic correlation calculation converts a series of quark-propagation-describing interactions among particles into many undirected graphs that have hadrons as vertices and gluons as edges, followed by performing a graph contraction on every graph that reduces graph edges one after another until only two vertices are left. Each reduction of an edge is a *tensor contraction* between hadron nodes, which is dubbed hadron contraction.

Computing many-body correlation functions is both computation and memory intensive because it involves not only a single hadron contraction but also a large number of hadron contractions with specific dependencies among them. Each hadron contraction can be formalized as a *batched matrix multiplication* in the meson system or a *batched tensor contraction* in the baryon system that is already computation intensive. In particular, the number of hadron contractions scales as the factorial of the number of quark degree of freedom, which makes computing many-body correlation functions memory intensive.

To overcome the significant expense of the calculations, a general solution is to leverage many-core architectures like GPUs [47, 49]. For example, the Redstar system [11], a well-known QCD simulation, *for the first time* calculates many-body correlation functions on many-core architectures. The Redstar system translates a statistic form of a many-body correlation function to an executable computation kernel, which consists of a series of hadron contractions. This work takes Redstar as an example and studies the new opportunities of optimizing many-body correlation on general GPU accelerators.

Many existing efforts that focus on optimizing tensor contractions [2, 7, 22, 23, 29, 31, 33, 39, 42] can be applied to many-body correlation; however, they usually result in sub-optimal performance. This is because all of these efforts focus on optimizing individual large tensor contractions, while many-body Lattice QCD correlation is featured with a great number of not large tensor contractions.

To address this issue, this work fully explores specific attributes of many-body correlation (i.e., a great many tensor contractions) and discovers new optimization opportunities. According to the physical observations and thorough programming analysis, redundant memory operations happen frequently in several areas: memory allocations, CPU/GPU memory communications, and memory oversubscriptions. First, in contrast to other many-body problems [15, 34, 43], overlapped

reduction paths among multiple contraction graphs result in a large number of repeated data (including initial and intermediate data), which brings memory allocation redundancy. Second, many-body correlation functions create many intermediate objects on CPU, and these objects also need to participate in computations on GPU. Accessing these intermediate objects from CPU and GPU interleavely causes frequent data movements between CPU and GPU. Finally, with the growing number of intermediate data, limited GPU memory on a single GPU inevitably leads to memory oversubscriptions [24, 27]. The randomly repeated data in hadron contractions easily cause evictions of previously used data that will be utilized very soon (near-reuse), raising redundant memory evictions and even data thrashing. Therefore, a general GPU memory optimization technique is required to control memory operations and accelerate hadron contractions.

This work proposes a memory management framework for many-body correlation on general GPU architectures (including NVIDIA GPUs and AMD GPUs) called MemHC to eliminate redundant memory operations. The novel optimizations mainly consist of three aspects. The first innovation is an integrated redundancy elimination mechanism to manage GPU memory from the host. Due to the large memory footprint of a single hadron node (37MB for a meson with 384 tensor size, 280MB for a baryon with 128 tensor size), current redundancy elimination techniques to operate within GPU registers [20, 37], cache [3, 13, 16], and shared memory [8, 20] are not practical in correlation functions. Targeting this challenge, MemHC leverages reusability to eliminate redundant memory allocation by applying duplication-aware management and overwriting lazy-released memory based on building mappings between CPU and GPU memory. Furthermore, MemHC explores and overcomes the limitations of Unified Memory management, which results in redundant CPU/GPU communications when passing references of GPU objects back to the host. The second new insight is a Pre-Protected eviction policy to minimize memory evictions by utilizing the specific storage formats, i.e., vectors, to predict data access and pre-protect all reusable data. Unlike other popular eviction policies [5, 19, 25, 38] to estimate the reuse distances of repeated data, the pre-protected eviction policy recognizes all repeated data in advance to completely avoid redundant evictions. The third contribution is the robust portability for general GPU architectures, especially for AMD GPUs using the ROCm framework, which currently does not support Unified Memory management.

The key contributions of this work can be summarized as follows:

- This work presents a GPU memory management framework, MemHC, to eliminate multiple memory redundancies. It efficiently facilitates reduction optimizations in memory allocations, CPU/GPU communications, and memory oversubscriptions.
- MemHC proposes memory reusability optimizations, including duplication-aware management and overwriting lazy-released memory, which yield benefits on allocation reductions.
- MemHC applies data reorganization based on contiguous memory locations and employs on-demand synchronization for CPU/GPU memory movement reductions, particularly overcoming memory communication redundancy produced by the Unified Memory management.
- MemHC exploits a novel **Least Recently Used (LRU)** eviction policy named Pre-Protected LRU eviction policy to protect reusable data in advance. This approach improves the memory hit rate and avoids data thrashing.
- MemHC illustrates robust portability on different platforms including NVIDIA GPUs and AMD GPUs.

This work evaluates general correlation functions based on synthesized random benchmarks with various parameters and data distributions. To further validate the practical performance, MemHC is extensively integrated into a real-world application and evaluated by three physical

correlation functions. The evaluation demonstrates that MemHC outperforms NVIDIA's Unified Memory management by **2.18**× to **10.73**× speedup. The proposed Pre-Protected eviction policy achieves up to **1.36**× higher GFLOPS than the original LRU eviction policy. Furthermore, the performance of real correlation functions is improved up to **6.12**× more than using Unified Memory management.

We organize the rest of the article as follows. Section 2 introduces the background of many-body correlation functions, the Redstar system, data characteristics, and computation patterns. Section 3 analyzes multiple memory redundancies and reusability opportunities. Section 4 illustrates an overview of the MemHC framework. Subsequently, we explain detailed techniques about memory reduction optimizations in Section 5. Section 6 demonstrates the evaluation observations and experimental results. Section 7 introduces related works, and Section 8 discusses future works. Finally, Section 9 concludes the article.

2 BACKGROUND

Computing many-body correlation functions, such as Lattice QCD, is a critical and challenging topic in the modern scientific field. Calculation of correlation functions is crucial for generating physics observables (e.g., predicting properties of light nuclei [14]) and is relevant to experiments planned for Jlab, FAIR, and J-PARC facilities [6]. Therefore, accelerating the many-body correlation function on GPU memory management has research and practical significance in nuclear physics.

Correlation function calculations are constituted by a large number of hadron contractions. One of the current popular lattice QCD systems, the Redstar system [9], focuses on solving correlation functions on many-core architectures efficiently. Taking accelerating hadron contraction as a critical user study about calculations of many-body correlation functions, this work is constructed and evaluated based on the Redstar system. This section mainly introduces the theoretical knowledge of correlation functions, analyzes the motivations to speed up hadron contraction on GPUs, explains the workflow of the Redstar system, and illustrates the kernel computation patterns of hadron contractions.

2.1 Correlation Functions

Based on the nature of the complicated many-particle systems, calculating many-body correlation functions is very important for generating physical observables. Different physics scenarios require different types of correlation functions. To be more specific, correlation function computation consists of many wick-contractions, which in turn can be turned into matrix multiplication in meson systems or tensor contraction in baryon systems. The rank of the tensors depends on the number of quarks in a hadronic node.

A correlation function between annihilation and creation operator χ at Euclidean t and t' is able to define the energy of an eigenstate of the Hamiltonian of a quantum field theory. The definition of the correlation function is:

$$C(t', t) = \langle \chi(t')\chi^{\dagger}(t) \rangle. \tag{1}$$

When inserting $\hat{H}|k\rangle = E_k|k\rangle$, as a complete set of eigenstates of the Hamiltonian, the correlation function represents an accumulation of all states:

$$C(t, t') = \sum_k |\langle \chi|k\rangle|^2 e^{-E_k(t'-t)}. \tag{2}$$

Each state has the same quantum numbers as the source operators. Take a two-point meson correlation function as an example; the definition is:

$$C(t, 0) = Tr_{dist\ spin}[M^{12}(t)U^{23}(t, 0)M^{34}(0)D^{41}(0, t)]. \tag{3}$$

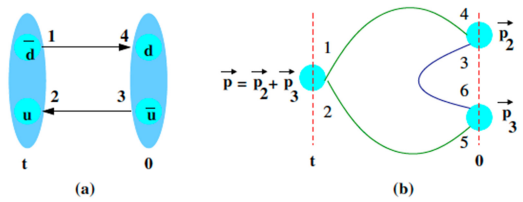

Fig. 1. Example of Correlation Functions. Figure (a) describes quark propagation in a simple meson system; Figure (b) represents quark propagation in a complex meson system.

In Equation (2), calculating the correlation function includes evaluating the quark field path-integral, inserting the out-product of the distillation operators, and keeping track of smearing labels as indices. The correlation function can be abstracted in Figure 1(a), and the edges between two vertices describe quark propagation. Another more complicated meson correlation function is shown in Figure 1(b). The theoretical definition is:

$$C(t,0) = \sum_{p2,p3|p} c_{p2,p3} M^{12}(\vec{p},t) P^{25}(t,0) M^{56}(\vec{p_2},0) P^{63}(0,0) M^{34}(\vec{p_3},0) P^{41}(0,t). \tag{4}$$

In practical physics scenarios, calculating many-body correlation functions on an ensemble of gauge fields has a high time cost to solve a great number of hadron contractions in physical observations. For instance, the hadron contraction number achieves more than 10,000 in a two-meson f0 system, while a baryon system is able to generate more than 100,000 hadron contractions. The computation cost of computing correlation functions grows rapidly due to the intermediate data. Producing intermediate data continuously also takes up significant memory resources. Moreover, the current advanced calculations of multi-meson correlation functions need about 10M core-hours for one ensemble in the gauge field. Therefore, improving the calculation functions is challenging and has great practical benefits in real-world scientific applications.

2.2 Redstar System

The Redstar system is designed to evaluate many-body correlation functions on multi-core architectures including CPU and GPU. As shown in Figure 2, the system consists of several stages including generating contraction graphs, producing multiple types of hadron nodes in distillation space, operating hadron contractions, and calculating correlation function results. The input of the Redstar system is a list of correlation functions. `Redstar_gen_graph` package translates the physical correlation functions to a contraction graph. In the contraction graph, vertices represent hadron nodes with various quarks. Then edges describe the interactions between hadron nodes. Subsequently, the system classifies different types of hadron nodes, relying on their physical definitions, then constructs hadron nodes to complete contraction graphs by using sub-modules `hadrom` and `colorvec`.

Another critical package, `redstar_npt`, computes the contraction graphs on multiple time slices and produces execution queues to guide hadron contraction computations. Some graph reorganization operations are applied to improve the correlation function computation. Moreover, this package conducts evaluations to measure the graph-level optimizations. The sub-module `redstar_npt` results in computation configurations and a queue of hadron contractions, which will be the input of the `hadron` package. Hadron contractions are generated in a fixed execution order. Sub-module `redstar_npt` is mainly implemented on CPU using parallel techniques such as OpenMP. The `hadron` package manages vectors to carry out hadron contractions. Hadron contractions take advantage of general GPU architectures.

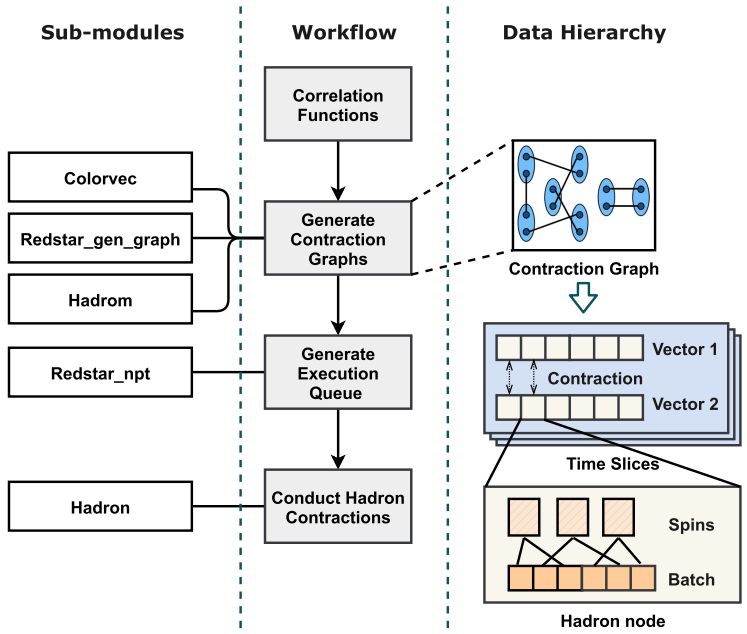

Fig. 2. Overview of the Redstar System: Sub-modules, Workflow and Hierarchical Data Structures. The Redstar system involves several stages: contraction graph generation (`Redstar_gen_graph`), building hadron nodes (`colorvec` and `hadrom`), and hadron graph contraction solutions (`redstar_npt`, `hadron`). The `hadron` package aims to accomplish hadron contractions. The Redstar system abstracts topologically contraction graphs from correlation functions, then reduces contraction graphs to various configurations. In one configuration, vector pairs cooperate to calculate contractions, which consists of independent hadron nodes. The hadron node includes multiple batches and associated spins.

2.3 Data Hierarchy

The Redstar system constructs hierarchical data structures. Figure 2 illustrates the whole picture of the multi-level data. When computing correlation functions, the Redstar system produces a sequence of hadron contractions from contraction graphs. Every single graph undergoes a graph contraction process during which one edge after another in the graph is reduced until two nodes are left. Each reduction of an edge corresponds to a tensor contraction.

Based on the definitions in Section 2.1, calculating correlation functions can be abstracted to compute a series of contractions generated from all the graphs on multiple time slices. All the graphs are topologically the same across different time slices but with different hadron nodes as their vertices. Thus, all the hadron contractions are the same types of calculations with different hadron nodes for different time slices.

One time slice includes various vectors. The order of vectors is determined by the execution queue, which is generated by `redstar_npt`. Contractions occur between two associated elements in a pair of vectors, like the first elements of Vector 1 and Vector 2 in Figure 2. A pair of vectors incorporate to accomplish contraction calculations.

Each vector consists of multiple independent hadron nodes, and a hadron node can be formalized as a tensor $T^{(abr)}(ijk)$, where abr represents spin and ijk represents distillation space. A contraction happens on both spin and spatial indices, and each spin component itself is a spatial tensor. To carry out a contraction between two hadron nodes, a single resulting spin of the destination nodes comes from multiple spins of these two hadron nodes. For example, for two

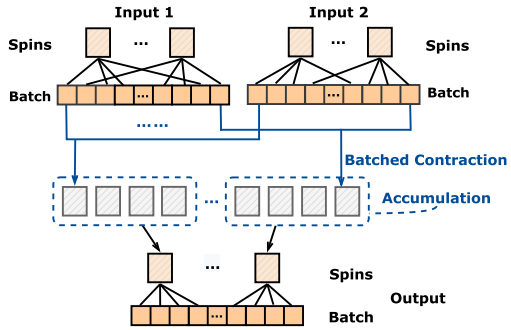

Fig. 3. Computation Patterns of an Individual Hadron Contraction. Two input hadron nodes conduct a batched contraction, then accumulate to generate a batched output hadron node.

mesons (0, 0) is generated from these four pairs of spins: (0, 0) (0, 0), (0, 1) (1, 0), (0, 2) (2, 0) and (0, 3) (3, 0). Therefore, a hadron contraction can be expressed as a sequence of matrix multiplications or a sequence of tensor contractions. From here we refer to these arrays of tensor contractions as batched multiplications. Furthermore, a single spin component from the source of a contraction can appear multiple times in the batched multiplications, e.g., (1, 0) can go to (0, 0) and (1, 0) to (1, 1). We define these duplicate relations between batch and spins to be *overlapped batch-spin mappings*. Take meson systems as an example; the batch size is often 64, while the number of spins is 16. Four elements in the batch point to the same destination spin.

2.4 Kernel Computation Analysis

Compared with conventional graph-based applications including Sparse matrix-vector multiplication, BFS, and PageRank, there are two specific characteristics in contraction graphs: (1) the entire calculation consists of a large number of small computation kernels (dense matrix multiplication or tensor contraction), representing each edge of contraction graphs; (2) the repeated appearance of the input data and intermediate output data, because of overlapped reduction paths among multiple contraction graphs to compute one correlation function.

It is well known that matrix multiplication or tensor contraction is already computation intensive. However, the expensive computation cost of many-body correlation results from a large number of hadron contractions, which leads to memory-intensive kernel computation. This section mainly explains the computation patterns of individual hadron contraction and analyzes the kernel computation of many-body correlation functions.

Take a meson system as an example to illustrate the computation pattern of an individual hadron contraction in Figure 3. The hadron node is two-dimensional, consisting of 16 spins, and the batch size will be 64. In the batch layer, each element points to one spin, and four elements point to the identical spin. Hadron contraction can be formalized as a batched tensor contraction and an accumulation operation. Input 1 and input 2 represent a pair of hadron nodes as input data. The mappings between batch and spins are probably different in the two input data. Two hadron nodes accomplish batched tensor contraction, generating 64 temporal tensors with 16 groups of four tensors being mapped to a single spin of the output hadron node. The library cuBLAS [1] is applied to conduct a batched contraction. Subsequently, every single group of the four temporary tensors is accumulated into a single tensor for one spin of the output.

Allowing for many-body correlation functions, kernel calculation is not only computation intensive but also memory intensive, due to the small size of spins in one hadron node and a large number of hadron contractions. On the one hand, the rank of the tensor is two and the tensor

size is often not more than 384 in meson systems. The computation cost of an individual hadron contraction is not heavy, which is 37MB in size for a single meson. On the other hand, the large number of hadron nodes requires significant memory capacity. Particularly, the data type is necessarily set to be double complex so as to guarantee the computation precision. Compared with the limited computation cost of a single kernel, it is more critical to focus on optimizing memory management for a series of hadron contractions.

3 REDUNDANCY AND REUSABILITY ANALYSIS

Based on the characteristics of computing correlation functions in Section 2, this work figures out various memory redundancies, which offers optimization opportunities to accelerate many-body correlation. In particular, there exist multiple types of duplicate data, raising data reusability chances.

3.1 Memory Redundancy Analysis

Memory redundancies broadly exist in the following aspects: memory allocation, CPU/GPU memory communication, and memory oversubscription. *First*, a great many intermediate data are repeatedly created and released during the calculations. The naive approach is to create new memory for each input pair. However, allocate and release operations about identical data are frequently repeated. Likewise, some new data obtain the same memory size as the released data. Thus, a number of memory allocations are redundant, which brings high time cost. *Second*, although the intermediate data references are created on CPU to determine the executing order, all the kernel computations exist on GPU. When manipulating parameters on CPU, only the references of intermediate data are passed. More specifically, no access operations of data values are performed, such as readings or writings. Under this situation, memory movements between CPU and GPU are unnecessary and result in memory redundancy in CPU/GPU communications. *Finally*, as the number of hadron nodes increases, more data are repeated in an uncertain order, leading to near-reuse memory evictions. Therefore, to solve the memory redundancy, a systematic memory optimization technique is desired for accelerating many-body correlation functions.

3.2 Data Reusability Chances

Data hierarchy illustrates various repeated data from configurations to hadron nodes, shown in Figure 2. For one original contraction graph, different configurations can be considered as different execution iterations. Their computations are consistent, while the data values are updated at a different time interval. When calculating one contraction graph, data occur repeatedly, which can be classified as three types: *duplicate initial data*, *repeated intermediate data*, and *overlapped batch-spin mappings*. For the appearance order of the repeated data, all the three types execute in an uncertain order. As for repeat frequency, initial data and intermediate data obtain random repeat frequency. The frequency of *overlapped batch-spin mappings* is a fixed number, relying on the input requirements (definitions of many-body correlation functions). Overall, repeated data provide multi-level data reusability chances, which inspires optimizations to fully utilize these repeated data and improve correlation function calculations.

4 SYSTEM OVERVIEW

According to the previous analysis, repeated data appearances cause broad memory redundancies and bring data reusability opportunities. The limitations of Unified Memory management [26], including redundant memory movements and lacking portability in general architectures (e.g., AMD GPUs), inspire an optimized memory redundancy mechanism for many-body correlation functions.

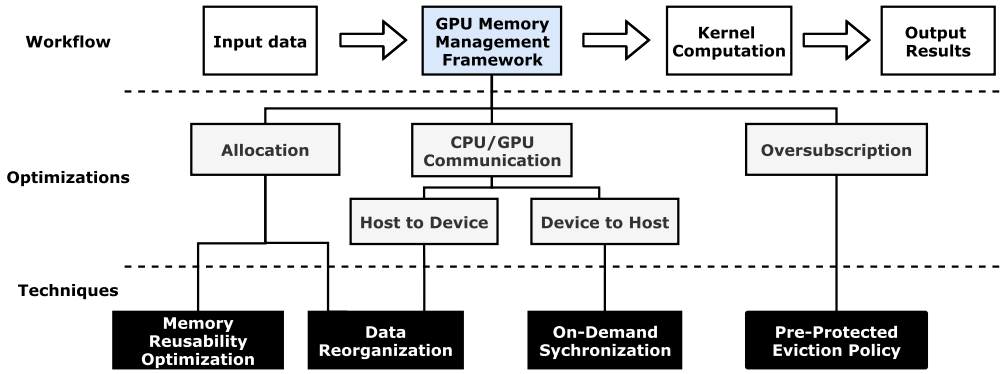

Fig. 4. System Overview: Optimizations and Associated Techniques. MemHC facilitates memory reduction managements between loading input data and kernel computation. MemHC proposes reduction optimizations in three aspects: GPU memory allocation, CPU/GPU communications, and GPU memory oversubscription. Techniques involve memory reusability optimizations, data reorganization, on-demand synchronization, and the Pre-Protected eviction policy.

Therefore, this work proposes MemHC, a GPU memory management framework, which efficiently facilitates a series of memory reduction optimizations to accelerate correlation function calculations. As shown in Figure 4, this work presents three optimized memory managements, including memory reusability optimizations for memory allocation, data reorganization and synchronization for CPU/GPU memory communications, and the Pre-Protected eviction for memory oversubscription, in order to eliminate redundant memory operations and enhance data reusability. *First*, MemHC conducts memory reusability optimizations. The optimizations involve *duplication-aware management* for repeated data and *overwriting lazy-released memory* for new intermediate data. *Second*, data reorganization is beneficial for both memory allocation and memory movement from the host to the device. *Third*, to decrease the latency of CPU/GPU memory communications, MemHC implements on-demand synchronization to efficiently manage data movements from device to host. *Last*, MemHC exploits a novel eviction policy, the Pre-Protected eviction policy, in order to avoid redundant evictions and data thrashing. Overall, multi-level memory redundancies motivate the corresponding memory reduction optimizations. The proposed GPU memory management framework, MemHC, adopts various techniques to leverage data reusability, eliminate redundant memory operations, and accelerate many-body correlation functions.

5 MEMORY REDUCTION OPTIMIZATIONS

Memory redundancies exist broadly in memory allocations, CPU/GPU communications, and memory oversubscription. Targeting these redundancy opportunities, this section mainly introduces a set of associated techniques: (1) memory reusability optimizations including *duplication-aware management* and *overwriting lazy-released memory*, (2) data reorganization based on contiguous memory locations, (3) performing on-demand synchronization, and (4) exploiting the Pre-Protected eviction policy.

5.1 Memory Reusability Optimization

Memory reusability optimizations involve enhancing the reusability of duplicate data and reducing redundant allocations of new intermediate data. On the one hand, identical data appear repeatedly through calculations, as explained in Section 3. On the other hand, the allocating and releasing of

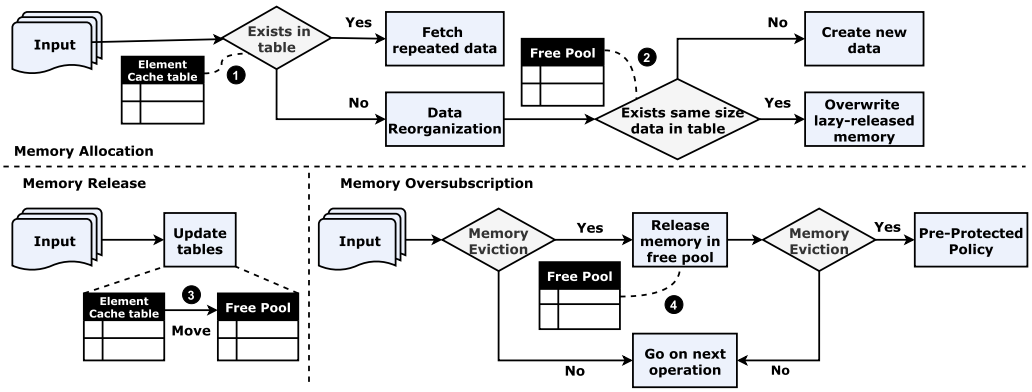

Fig. 5. Memory Reusability Optimizations: Duplication-aware Management and Overwriting Lazy-released Memory. During memory allocation, MemHC checks the `element_cache` table to fetch duplicate data (❶). If the data is new, the `free_pool` table will be checked to find lazy-released device memory with the same size (❷). If it exists, MemHC overwrites the device memory by the new data. When releasing memory, recordings are moved from the `element_cache` table to the `free_pool` (❸). When memory is oversubscribed, the recorded data in `free_pool` are released (❹) before eviction.

intermediate data occur frequently, which brings a large number of redundant memory. These two types of allocation redundancies inspire MemHC for memory reusability opportunities.

MemHC exploits two targeted memory reusability optimizations for both repeated data and new intermediate data:

- **Duplication-aware management.** This work employs *duplication-aware management* to reuse duplicate input data. MemHC records the mappings between host objects and associated device memory locations. The *duplication-aware management* recognizes repeated input data by checking the mappings. The allocated device memory is directly fetched for kernel calculations, without any redundant memory allocations.
- **Overwriting lazy-released memory.** To further leverage the reusability of intermediate data, another optimization is *overwriting lazy-released memory*. Memory release operations are delayed to be reused by new intermediate data, which obtain the same memory size as the allocated memory. This approach efficiently reduces both memory allocations and memory release operations.

Figure 5 illustrates the workflow of memory reusability optimizations. MemHC designs two memory tables to help implement *duplication-aware management* and *overwriting lazy-released memory*: `element_cache` table and `free_pool` table, respectively. The `element_cache` table mainly records mapped host objects and the device memory address of active data, while the `free_pool` table records device memory information about lazy-released data. When creating new memory, MemHC first checks the `element_cache` table to fetch the reusable data (❶). If the data is not recorded, MemHC will check the `free_pool` table to find an allocated memory with the same memory size (❷). If it exists, the memory will be reused and overwritten by new data. When managing memory release, MemHC erases data in the `element_cache` table and then adds it to the `free_pool` table (❸). Additionally, MemHC first releases the data in the `free_pool` (❹) during memory oversubscription. If memory is still not enough, the Pre-Protected eviction policy will start addressing memory evictions. In summary, the usage of two memory tables efficiently leverages data reusability and eliminates redundant memory operations. The two memory reusability

optimizations yield significant benefits on memory allocation reduction. Detailed optimizations about memory oversubscription will be explained in Section 5.4.

5.2 Data Reorganization

Separated allocations of spins in one hadron node result in redundant memory allocations and memory movements. As claimed in Section 2, each spin represents a matrix or tensor. Although the batch layer combines spins together, these spins are created and allocated individually on the host. Take an example in a typical meson system. The number of spins is set to be 16. When solving one hadron node, the naive approach produces 16 allocations. Furthermore, calculating a large number of hadron nodes leads to expensive time cost.

Therefore, MemHC performs data reorganization to reduce memory allocations and movements. Compared with other data reorganization works [17, 18, 40, 44, 45], MemHC mainly focuses on reorganizing the internal structure of the hadron node by packing spins together. As for the previous example, if the number of spins is 16, packing spins is able to reduce the number of allocating spins from 16 to 1. The number of transferring spin values from host to device can also be reduced. Thus, the objective is to pack spins together into contiguous memory locations. MemHC applies contiguous data storage formats on both input and output data. After reorganizing the data structure, the latency of data allocations and transfer can be significantly reduced.

Particularly, reorganizing the hadron node structure requires more batch manipulations than common batched tensors, due to the *overlapped batch-spin mappings*. To cover general correlation functions, these mappings are considered randomly produced. Managing the *overlapped batch-spin mappings* is the main challenge of employing contiguous spin memory locations.

To overcome this issue, MemHC records the *overlapped batch-spin mappings* in advance and rebuilds the mappings before and after kernel computations. Kernel `zgemm` in cuBLAS [1] requires the input data structure to be a two-dimensional array. MemHC translates the allocated contiguous one-dimensional array to a two-dimensional array, in order to construct a formal input of the `zgemm` kernel. The *overlapped batch-spin mappings* of input data are given as an unpredictable structure, but the mappings of output data are fixed. MemHC builds the batch-spin mappings of the output data on the device after accumulation operations.

5.3 On-demand Synchronization

One of the frequent operations in correlation function calculation is to manipulate intermediate GPU objects by passing their references on the host. On one hand, the execution order of intermediate data should be guided by the manager onto the host. On the other hand, calculations about intermediate data only occur on the device. More specifically, some parameter manipulations require managing intermediate data from the host, but these operations are passing references without accessing values. Thus, it is unnecessary to update host data values during computations. However, current management frameworks (e.g., unified memory management of NVIDIA) [26, 30, 35, 41] cannot recognize this situation and produce redundant memory movements when passing references. Plenty of passing reference operations on the host incur significant CPU/GPU communication redundancies.

Targeting this specific situation, this work accomplishes on-demand synchronization to eliminate CPU/GPU communication redundancy. Synchronizations occur when releasing device memory or accessing associated host data values instead of passing references onto the host. MemHC handles the intermediate data to stay on GPU until released, without any CPU/GPU communications. More specifically, MemHC defines intermediate data as a new data type, *GPU-only object*, and avoids redundant memory movements about this type of data. As a complementary to the CPU/GPU communication management, MemHC also implements a synchronization function,

ALGORITHM 1: LRU Eviction Policy

Require: *curr_mem_*, *lru_mem_*, *elem_cache_*, *gpu_only_objs_*

```
1: while curr_mem_ is not enough for new data do
2:    mem ← lru_mem_.back(); {obtain the host address of the least used object in LRU queue}
3:    cache_ptr ← elem_cache_.find(mem); {obtain the object from the element_cache table}
4:    if cache_ptr is GPU-only then
5:       objptr ← gpu_only_objs_.find(cache_ptr); {obtain the device address of the evict object}
6:       objptr->sync(); {update associated data on the host}
7:    end if
8:    if cache_ptr is the head of contiguous memory then
9:       children ← cache_ptr->second.mem.children(); {obtain children elements}
10:      for i ← 0 − children.size() do
11:         Free child_ptr {free memory of this child}
12:         elem_cache_.erase(child_ptr); {remove this child from element_cache table}
13:      end for
14:   end if
15:   update curr_mem_; {update memory information: the current available memory size}
16:   lru_mem_.pop_back(); {pop the back of the LRU queue}
17:   elem_cache_.erase(cache_ptr); {remove the cache_ptr from element_cache table}
18: end while
```

which makes data copy controllable for user requirements. Data transfer occurs from device to host only if the synchronization function is called to update host values.

5.4 Memory Oversubscription: Pre-Protected Eviction

With respect to the limited memory of GPU (e.g., 16GB memory in NVIDIA), memory oversubscription is a critical topic in GPU memory management. To eliminate memory eviction redundancy, MemHC designs a novel algorithm, the Pre-Protected LRU eviction policy, to fully protect reusable data in advance based on vector forms of input data.

5.4.1 LRU Eviction Policy. To address memory evictions, prior efforts present many eviction algorithms, including Random Eviction, **Most recently used (MRU)**, LRU, **clock with adaptive replacement (CAR)**, Clock-Pro, and more complicated replacement methods [5, 25, 38].

The LRU eviction strategy is based on the **First In First Out (FIFO)** algorithm. The main idea of the LRU policy is to evict the least recently used elements first. Figure 5 illustrates the pre-process of the memory evictions. The lazy-released memory in `free_pool` is freed when device memory is oversubscribed. MemHC manages an LRU memory queue to contain all active host memory addresses and makes use of the `element_cache` table to guide device memory eviction.

Detailed information about the LRU eviction algorithm is shown in Algorithm 1. First, MemHC checks the current memory size and determines the number of data to evict. Next, it fetches the back element of the LRU memory queue and finds the mapped value of this fetched host memory address in `element_cache` table. Based on the contiguous data formats, if the evicted element is the head of the contiguous memory, MemHC frees all its children elements. Last, MemHC updates the `element_cache` table and the LRU memory queue.

5.4.2 Pre-Protected LRU Eviction Policy. Although LRU is efficient to deal with common GEMM kernel computations, LRU may cause near-reuse memory evictions when calculating correlation functions. Figure 6 Example (a) shows that LRU produces redundant evictions. Assume the input

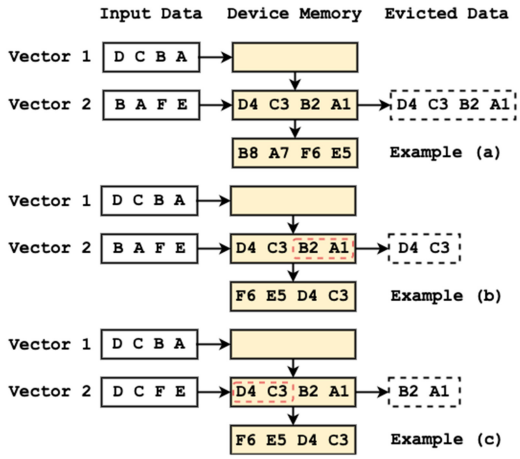

Fig. 6. Examples of Memory Oversubscription: Compare LRU and Pre-Protected LRU. Input data are (A, B, C, D) as a first vector and (E, F, A, B) or (E, F, C, D) as a second vector. Example (a) shows that LRU produces redundant evictions; Examples (b) and (c) show that Pre-Protected LRU protects repeated data in advance, to avoid redundant evictions.

data are two vectors. The first vector includes (A, B, C, D) and the second one is (E, F, A, B). The memory size is four. The number with the letter means the order of loading and storing in the LRU queue. LRU policy selects the data with the smallest number to evict. Data A, B, C, and D are pushed into the LRU queue in order. When solving the second vector, A and B are evicted first, then allocated again. The total number of memory evictions is four. As the number of repeated data grows, it may incur data thrashing.

As for other eviction policies, Clock-Pro [19] considers not only the recently referenced data but also recently evicted data, which is better than LRU in one-time scan and large loop. CAR [5] is self-tuned and theoretically more efficient than LRU. These eviction policies implement different techniques to reduce the redundant evictions but cannot avoid them completely.

This work first implements LRU, as one of the most popular and fundamental algorithms, to assist many-body correlation calculations. Based on this policy, MemHC designs a Pre-Protected LRU eviction policy to avoid redundant memory evictions of repeated data by utilizing the vector form of hadron nodes. The vector form means loading a sequence of data in advance, capturing all repeated data to completely avoid redundant memory evictions. More specifically, this approach takes the reuse distance of data within one vector into consideration. The pre-protected data have the least reused distances. Limiting the prediction range into vector size aims to balance the managing overhead and eviction reductions.

Figure 6 illustrates how the Pre-Protected LRU policy works to eliminate redundant memory evictions. In Example (b), after loading the second vector in the hadron package, A and B are found in the LRU memory queue and pre-protected to avoid evictions since they are in both the first and the second vector. When E comes in, since A and B are protected, the unprotected least recently used data, C, will be evicted. After solving the second vector, only two data, C and D, are evicted. Compared with the original LRU policy, our method reduces the evictions from four to two and increases two memory hits. Example (c) further shows that by changing the repeated data or their positions, all the reusable data will be checked and protected in the LRU queue in advance, without producing redundant memory evictions. When the second vector (E, F, C, D) comes in, C and D will be protected and only A and B are evicted.

ALGORITHM 2: Pre-Protected LRU Eviction Policy

Require: *vector*, *curr_mem_*, *lru_mem_*, *elem_cache_*, *gpu_only_objs_*

```
1: if curr_mem_ is not enough for new data then
2:    for src in vector do
3:       if src in elem_cache_ {check each element of vector exists in element_cache table} then
4:          src.flag_protected ← TRUE; {protect the reusable data}
5:       end if
6:    end for
7: end if
8: while curr_mem_ is not enough for new data do
9:    mem ← lru_mem_.back(); {obtain the host address of the least used object in LRU queue}
10:   cache_ptr ← elem_cache_.find(mem); {obtain the object from the element_cache table}
11:   if cache_ptr is protected then
12:      lru_mem_.erase(cache_ptr);
13:      lru_mem_.push_front(cache_ptr); {avoid evicted in the next iteration}
14:      src.flag_protected ← FALSE; {avoid over-protection}
15:   end if
16:   Same statements in LRU Eviction
17: end while
```

Algorithm 2 shows the Pre-Protected eviction policy. In the beginning, MemHC checks the LRU memory queue and finds all the repeated data to pre-protect. It adds a flag to label pre-protection. If the data exists in the `element_cache` table, MemHC sets the flag to be true. When memory oversubscription happens, MemHC first checks a pre-protected flag. We design the flag to distinguish the protected data from other input data. If the data is protected, move it from the end to the front of the LRU memory queue. In most cases, the GPU memory size is much larger than the data size of one vector, and unprotected data are enough to evict.

6 EVALUATION

This work aims to accelerate many-body correlation functions based on the optimized GPU memory management. We build the GPU memory management framework, MemHC, which efficiently eliminates multiple memory redundancies in calculating correlation functions. The experiments broadly cover evaluating general correlation functions and real-world physics correlation functions with varying factors on NVIDIA and AMD GPUs.

6.1 Experiment Methodology

Evaluation Setup. To measure the performance for general architectures, MemHC executes on NVIDIA Pascal P100, NVIDIA Volta V100, AMD MI50, and AMD MI100. P100 has 16GB GPU memory, while V100, MI50, and MI100 have 32GB GPU memory. Kernel computation is compiled by CUDA 10.2 on NVIDIA and ROCm 4.3.0 on AMD.

Experiment Design. This work designs three series of experiments, including general correlation functions with fitted memory, general correlation functions with memory oversubscriptions, and improved performance in the Redstar system.

To evaluate general many-body correlation functions, this work applies a set of synthesized benchmarks. The benchmarks broadly cover multiple tensors with varying tensor size, repeated rate, and vector size. The vector size means the number of independent hadron nodes in one vector. Repeated rate means the ratio of the data that appears previously to all the data. Repeated rate

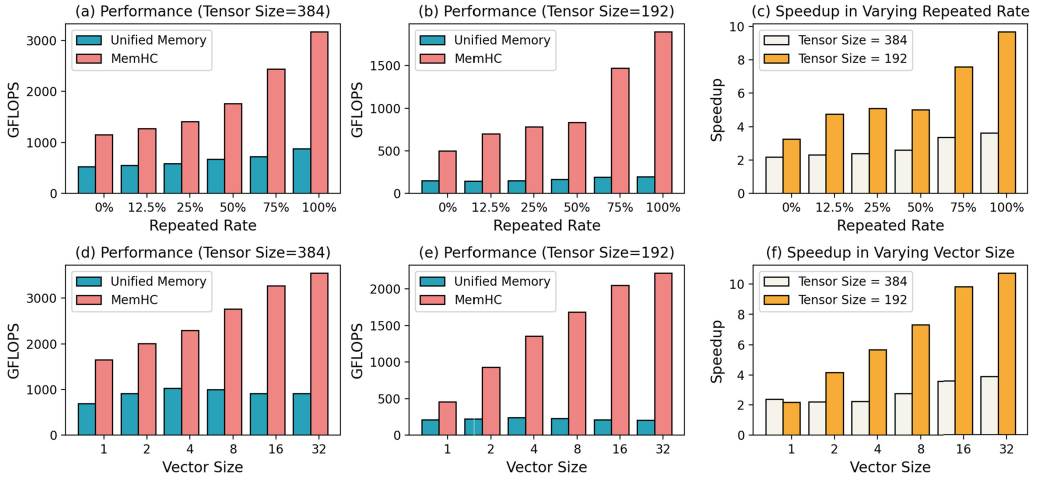

Fig. 7. Overall Performance: Comparing GFLOPS of Unified Memory Management and MemHC on NVIDIA. Figures (a) and (b) illustrate performance with varying repeated rate: 0%, 12.5%, 25%, 50%, 75%, and 100%. Figures (d) and (e) show performance with varying vector size: 1, 2, 4, 8, 16, 32. Figures (c) and (f) imply the speedup of MemHC based on the unified memory. The sizes of evaluated tensors include 384 in Figures (a) and (b) and 192 in Figures (d) and (e).

means the reusable data in one vector. For instance, a 50% repeated rate represents that half the data of each vector are repeated; a 100% repeated rate represents that all the data in one vector appear previously. Each vector has unique data. Particularly, the vector size implies the number of streams managed by OpenMP on CPU, aiming to reduce the latency of parameter manipulations. Oversubscription rate implies the proportion of the oversubscribed data size to the memory capacity.

To further validate the practical performance, this work measures performance improvements in three real physical correlation functions, in which MemHC is integrated to the Redstar system as a user case evaluation. Unified Memory management is evaluated as the baseline for both general and real-world correlation functions. To evaluate the performance of the Pre-Protected eviction policy, MemHC compares with the original LRU eviction policy. When GPU memory is oversubscribed, the performance is sensitive to data distribution. Thus, this work evaluates two data distributions including uniform distribution and Gaussian distribution.

Evaluation Objectives. Evaluation aims to achieve the following objectives: *First,* this work demonstrates that MemHC outperforms unified memory management of NVIDIA, achieving up to 10.73× speedup in general correlation functions. The performance of the Pre-Protected eviction policy is improved up to 1.36× compared with LRU. *Second,* this work presents the generality of proposed optimizations with varying tensor size, vector size, repeated rate, and oversubscription rate. *Third,* this work illustrates the robust portability and widely compares the performance on multiple platforms, including NVIDIA P100, NVIDIA V100, AMD MI50, and AMD MI100. *Last,* MemHC is evaluated by applying it in a real-world system, the Redstar system. Experiment results show great benefits of MemHC on real physics correlation functions, achieving up to 6.12× speedup.

6.2 Overall Performance Improvements (without Oversubscriptions)

Figure 7 illustrates the overall improved performance of MemHC on NVIDIA. This work compares the unified memory management and optimized MemHC. Figures 7(a) and (b) show GFLOPS with

varying repeated rate: 0%, 12.5%, 25%, 50%, 75%, and 100%. Figures 7(c) and (d) represent GFLOPS with varying vector size: 1, 2, 4, 8, 16, 32. The evaluated tensor size is 384 as shown in Figures 7(a) and (c) and 192 in Figures 7(b) and (d). The evaluated results are calculated as an average result of 10 execution times. Each execution time measures 100 vectors, which are randomly generated. The vector size is 16 in Figures 7(a) and (b). The repeated rate is 100% in Figures 7(d) and (e). The number of vectors is 100 in Figure 7.

As shown in Figure 7, MemHC outperforms unified memory management in all cases. When the tensor size is 384, the speedup achieved is from 2.18× to 3.62× in Figure 7(a) and ranges from 3.26× to 9.67× in Figure 7(b). These experimental results illustrate that the overall performance is sensitive to the repeated rate. When increasing the repeated rate, the overall performance achieves more speedup, due to the increasing memory redundancy opportunities.

MemHC yields more benefits when the tensor size is 192. The speedup achieved is from 2.39× to 3.86× in Figure 7(d) and 2.17× to 10.73× in Figure 7(e). This is because the latency of kernel computation is sensitive to tensor size. Tensor size is smaller, and memory redundancies have more impacts on the performance. More memory redundancies offer more optimization opportunities in MemHC, leading to more speedup shown in Figures 7(c) and (f).

One observation is that MemHC is sensitive to the repeated rate. When the repeated rate is 50%, the performance achieves less than half of that in the case of a 100% repeated rate, shown in Figure 7(b). This is because the ratio of repeated data affects the amount of memory redundancies. More repeated data provide more redundant opportunities. Particularly, to eliminate memory redundancy, one of the critical optimizations in MemHC is to leverage reusability of the repeated data. Therefore, the sensitivity of the repeated rate demonstrates high memory redundancy efficiency of MemHC.

When changing vector size, unified memory management does not show any improvements, while MemHC achieves obvious speedup. Figure 7(d) shows that MemHC achieves GFLOPS from 1,653.59G/s to 3,541.42G/s, and Figure 7(e) illustrates that GFLOPS improves from 454.53G/s to 2,213.88G/s. This is because a larger vector size produces more redundancies. As explained before, all cases apply a 100% repeated rate and the number of vectors is fixed. A larger vector size brings more repeated data in each vector. Compared with unified memory management, experiment results show great benefits of MemHC on data reusability.

6.3 Performance Analysis in General Correlation Functions

6.3.1 Breakdown Analysis. To further explore the efficiency of MemHC, this section focuses on breakdown analysis. Figure 8 evaluates the performance of the memory redundancy techniques separately and compares with the Unified memory management both GFLOPS and speedup. Unified memory management is evaluated by two cases: non-optimized implementation (`Unified Memory Naive`) and an optimized version by data reorganization (`Unified Memory Data Reorg`). MemHC is evaluated in the following four cases: (1) `MemHC Naive` without any optimizations but explicit implementation; (2) `MemHC Data Reorg`, which is only optimized by data reorganization; (3) `MemHC Data Reorg + Sync` combining data reorganization and on-demand synchronization; (4) the optimal implementation, `MemHC Optimal`, with all memory redundancy techniques. Tensor size is set to be 384, and the vector size varies from 4 to 16 in (a) and the repeated rate varies from 0%, to 100% in (b). The performance results are measured on NVIDIA P100.

Figures 8(a) and (c) show the impact of the repeated rate. When the repeated data is 0%, the speedup of `MemHC Optimal` is about 1.25× based on the `MemHC Naive`. When the repeated rate is 50%, Explicit `MemHC Optimal` achieves 2.6× speedup. When applying both data reorganization and on-demand synchronization, the geometric mean speedup achieved is from 2.5× to 2.8× in (a) and 2.1× to 2.4× in (c). Figures 8(b) and (d) illustrate the influence of the vector size. Take

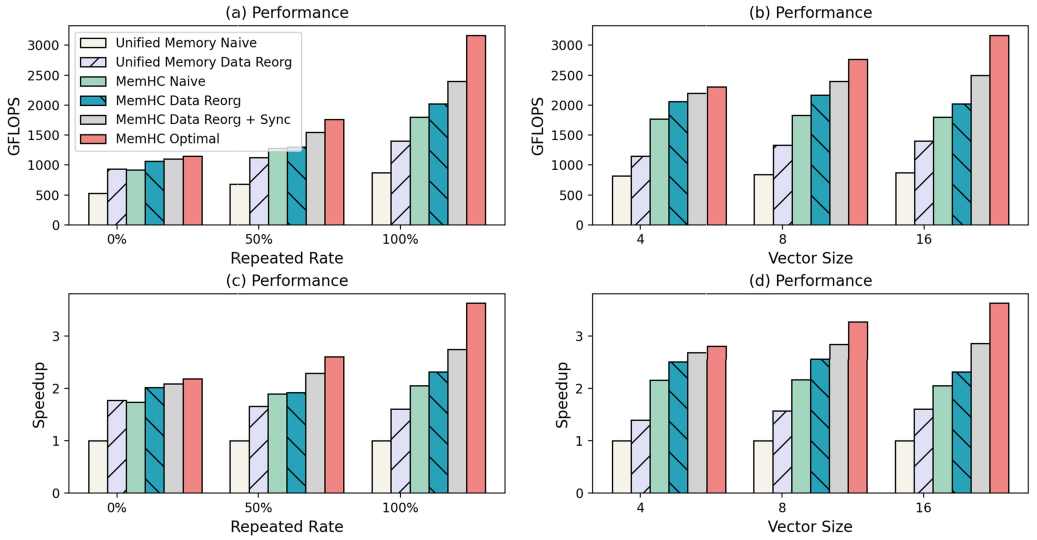

Fig. 8. Optimization Breakdown: Compare Unified Memory Management and MemHC about GFLOPS and Speedup. Unified memory management is evaluated by two cases: Unified Memory Naive and Unified Memory Data Reorg. MemHC is evaluated by four cases: naive (MemHC Naive), only optimized by data reorganization (MemHC Data Reorg), optimized by data reorganization and on-demand synchronization (MemHC Data Reorg + Sync), and the optimal implementation (MemHC Optimal). Tensor size is 384. The vector size varies from 4 to 16 and repeated rate varies from 0% to 100%. The performance results are measured on P100.

comparing vector sizes 4 and 16 as an example. The speedup is 1.42× of the `MemHC Optimal`. `Unified Memory Data Reorg` achieves 1.22× speedup and `Unified Memory Naive` achieves 1.06× speedup. Data reorganization improves both unified memory management and MemHC. Memory reusability optimizations further enhance performance in MemHC.

As for data reorganization, when the repeated rate is 50%, the performance is not obviously improved by the `MemHC Naive`. This is because data reorganization reduces the redundant allocations and communications no matter whether the data are new or repeated. The reorganization is to change the data structure in each hadron node, so as to reduce the number of memory operations from the batch size to one. Additionally, the repeated rate has an impact on the overhead of data reorganization. If the repeated rate is 50%, half of the hadron nodes are repeated and the batch-spin mappings will be complicated to extract and rebuild, leading to trivial improvements by `MemHC Naive`. If the repeated rate is 0%, applying data reorganization has more improvements than that of the 50% repeated rate, since there is no overlap in the batch-spin mappings. If the repeated rate is 100%, all the hadron nodes will be the same and the reorganization is simple. Therefore, the repeated rate, which determines the complexity of the hadron node's internal structure, has an impact on reorganization overhead, further influencing the data reorganization improvements.

6.3.2 Exploring Portability on NVIDIA and AMD GPUs. To further confirm the robust portability of this work, the performance of MemHC is broadly evaluated on different GPU architectures.

Figures 9(a) and (b) illustrate the speedup of MemHC over the non-optimized explicit implementation on those four GPU architectures. In most cases, NVIDIA V100 can achieve the best speedup among others. When tensor size is 192, NVIDIA V100 can achieve up to 4.4× speedup when vector size is 1. For tensor sizes 192 and 384, NVIDIA V100 can achieve average 3.8× and 1.3× speedups, respectively.

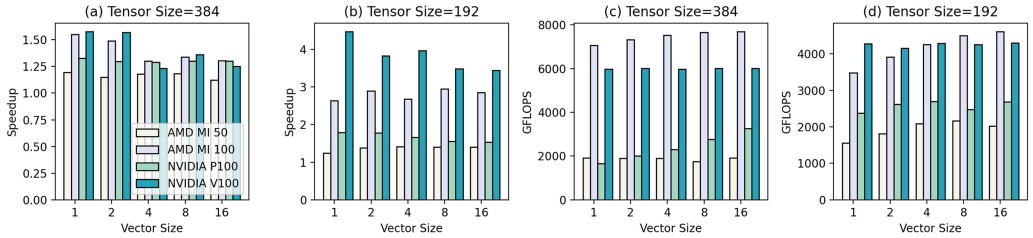

Fig. 9. Exploring Portability: GFLOPS and Speedup on AMD MI50, MI100 and NVIDIA P100, V100. Tensor sizes are 384 and 192. Vector sizes are 1, 2, 4, 8, 16. Figures (a) and (b) show speedup based on non-optimized explicit implementation. Figures (c) and (d) illustrate GFLOPS.

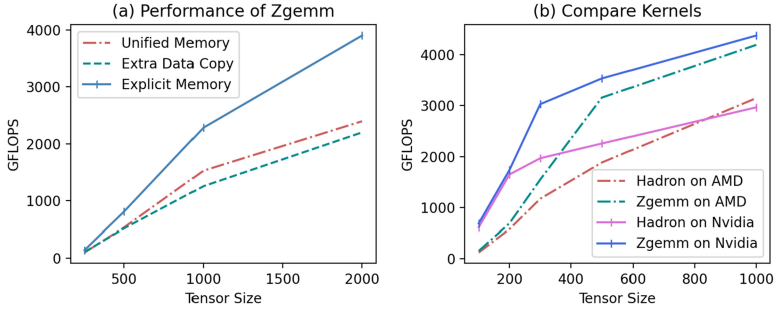

Fig. 10. Exploring Kernel Computation on AMD MI50 and NVIDIA P100. Figure (a) explores the limitation of Unified Memory. Extra Data Copy denotes adding redundant data transfer in explicit memory implementation. Figure (b) compares the performance of zgemm in cuBLAS and hadron contraction kernels on MI50 and P100.

Figures 9(c) and (d) illustrate the MemHC's GFLOPS on four different GPUs, including AMD MI50, AMD MI100, NVIDIA P100, and NVIDIA V100, with various settings of vector sizes and tensor sizes. When the vector size increases from 1 to 16, MemHC increases GFLOPS on all four architectures. Among them, AMD MI100 and NVIDIA V100 have better GFLOPS than AMD MI50 and NVIDIA P100. When the tensor size increases from 192 to 384, the absolute GFLOPS gap between them increases.

6.3.3 Exploring CPU/GPU Communications in Unified Memory. Compared with unified memory management, one critical benefit of MemHC is improving CPU/GPU communications. This section mainly analyzes the limitations about CPU/GPU communications in unified memory management.

Current NVIDIA GPUs provide Unified Memory [26, 30], which supports a virtual single address accessible for both CPU and GPU. This feature is convenient for users to extend CPU codes to CUDA codes without caring about data locations. Recent machines (e.g., Tesla P100) support hardware page faulting and on-demand migration [35, 41]. When evaluating unified memory management, this work sets the `cudaMemAdviseSetReadMostly` flag and performs the `cudaMemPrefetchAsync` function to manage data locations.

To further validate the limitations of unified memory, this work conducts a series of experiments to explore its bottleneck. Experiments are set up by using the `zgemm` kernel of cuBLAS. In Figure 10(a), there are three lines illustrating the performance of `zgemm`. The blue line simulates the explicit implementation of `zgemm`. The red line represents the unified memory managed

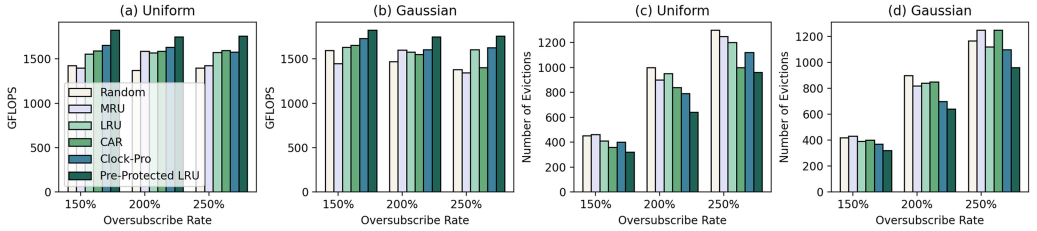

Fig. 11. Comparing Multiple Eviction Policies. Pre-Protected eviction policy is compared with Random policy, MRU, LRU, CAR, and Clock-Pro policies. Measure metrics include GFLOPS and the number of evictions. Oversubscribe rate changes from 50% to 150%. The tensor size is 384. Vector size is 64. Repeated rate is 50%.

by the GPU driver. Another implementation is the explicit memory management with extra data copy from device to host when passing references of device objects. As shown in Figure 10(a), the unified memory management achieves from 61.4% to 70.5% performance of the explicit memory management. The green line is very close to the red line, which means the extra data copy in explicit memory management can simulate the case of the unified memory implementation.

In conclusion, the GPU driver produces redundant CPU/GPU communications when passing references in the unified memory. The observation further supports the necessary and high efficiency of defining GPU-only objects to keep intermediate data always on device. Compared with unified memory management, MemHC efficiently eliminates redundant CPU/GPU communications.

6.3.4 Exploring Hadron Contraction Kernel. To explore the efficiency of the hadron contraction kernel, this work compares the performance between hadron contractions and `zgemm`. The `zgemm` kernel is well optimized in a widely used library, considered as the peak performance of computing batched tensor contractions. Figure 10(b) shows the experimental results. In NVIDIA, MemHC achieves 61.4% to 94.8% performance of `zgemm`. The average percentage is 73.6%. In AMD, MemHC is able to achieve 59.6% to 83.7% and the average percentage is 74.5%.

As explained in Section 2, this work implements batched tensor contraction kernel based on the `zgemm`. When the tensor size is smaller than 200, the performance of the `zgemm` kernel and hadron contraction achieve close performance. Besides the `zgemm` kernel, the execution latency of hadron contractions includes parameter manipulations on CPU and accumulating operations among batches on GPU. As the tensor size becomes larger, the extra overhead increases in the beginning and generates a stable impact on the whole performance. Both NVIDIA P100 and AMD MI50 have similar trends, while P100 achieves a stable trend earlier than MI50. The experiment results illustrate the high efficiency of our hadron contraction implementation.

6.4 Memory Oversubscriptions in General Correlation Functions

This section evaluates the LRU eviction policy in two synthesized datasets including uniform distribution and Gaussian distribution. The evaluation involves two aspects: comparing with multiple eviction policies and exploring the performance broadly with different situations.

Comparing with multiple eviction policies. Figure 11 illustrates the comparison between Pre-Protected LRU and other eviction policies, including Random, MRU, LRU, CAR, and Clock-Pro. The experiments measure GFLOPS in Figures 11(a) and (b) and the sum of 10 iterations of the number of evictions in Figures 11(c) and (d). The data distribution includes Uniform distribution and Gaussian distribution, when the oversubscribe rate ranges from 150% to 250%. Figures 11(a) and (b) show the GFLOPS results of these eviction policies. Among them, Pre-Protected LRU shows the best performance in all cases. Overall, Pre-Protected LRU can achieve 1.1× geometric

Table 1. Performance of Memory Eviction with Varying Vector Size: Hit Rate, GFLOPS on NVIDIA P100

Experiment Results per Iteration (Oversubscription Rate = 50%, Repeated Rate = 1/2)								
Vector Size	**Eviction Policy**	**Allocate**	**Evictions**	**Hit**	**Miss**	**Hit Rate**	**GFLOPS**	**Speedup**
8	LRU_UNIFORM	27	8	16	32	33.33%	780.16	1×
	LRU_GAUSSIAN	25	6	18	30	37.5%	934.27	1.19×
	Pre-Protected LRU	**24**	**4**	**19**	**29**	**39.58%**	**1,062.84**	**1.36×**
16	LRU_UNIFORM	48	12	35	61	36.45%	1,453.61	1×
	LRU_GAUSSIAN	47	12	36	60	37.50%	1,501.1	1.03×
	Pre-Protected LRU	**44**	**8**	**39**	**57**	**40.63%**	**1,907.89**	**1.31×**
32	LRU_UNIFORM	89	22	74	118	38.54%	1,571.02	1×
	LRU_GAUSSIAN	87	20	76	116	39.58%	1,605.04	1.02×
	Pre-Protected LRU	**84**	**16**	**79**	**113**	**41.15%**	**1,943.48**	**1.23×**

The results are calculated as the average values per vector of 10 execution loops. The vector size varies from 8 to 32. Each vector contains half repeated data. Uniform distribution and Gaussian distribution are applied to evaluate LRU. Pre-Protected LRU protects all repeated data, so data distributions have no impact. Oversubscribed memory is half of the available memory size. Improvement means the times of the GFLOPS improvements.

Table 2. Performance of Memory Eviction with Varying Oversubscription Rate: Hit Rate, GFLOPS on NVIDIA P100

Experiment Results per Iteration (Vector Size = 64, Repeated Rate = 1/2)								
Oversubscription	**Eviction Policy**	**Allocate**	**Evictions**	**Hit**	**Miss**	**Hit Rate**	**GFLOPS**	**Speedup**
50%	LRU_UNIFORM	178	46	145	239	37.76%	1,554.54	1×
	LRU_GAUSSIAN	171	40	152	232	39.58%	1,630.69	1.05×
	Pre-Protected LRU	**164**	**32**	**159**	**225**	**41.4%**	**1,825.28**	**1.17×**
100%	LRU_UNIFORM	216	120	235	341	40.79%	1,568.06	1×
	LRU_GAUSSIAN	213	82	238	338	41.31%	1,578.86	1.01×
	Pre-Protected LRU	**196**	**64**	**255**	**321**	**44.2%**	**1,750.12**	**1.12×**
150%	LRU_UNIFORM	251	120	328	440	42.71%	1,575.25	1×
	LRU_GAUSSIAN	243	112	336	432	43.75%	1,606.90	1.01×
	Pre-Protected LRU	**228**	**96**	**351**	**417**	**45.7%**	**1,759.57**	**1.12×**

The results are calculated as the average values per vector of 10 execution loops. The oversubscribed memory rate varies from 50% to 150%; 50% means half of the available memory size is oversubscribed. The vector size is 64. Each vector contains half repeated data.

mean GFLOPS over other eviction policies in both datasets using various oversubscribe rates. Figures 11(c) and (d) show the number of evictions of these eviction policies. The evaluation results illustrate that Pre-Protected LRU has the lowest number of evictions than others in both synthesized datasets. When the oversubscribe rate is 150%, 200%, and 250%, Pre-Protected LRU only has a geometric mean 77.2%, 71.6%, and 82.1% number of evictions of other methods in the uniform dataset, and 79.7%, 78.1%, and 81.6% in the Gaussian dataset, respectively.

Exploring performance in varying situations. This work compares the performance between the Pre-Protected LRU and original LRU policy with three crucial factors: vector size, oversubscription rate, and repeated rate. For the Pre-Protected eviction policy, all the repeated data are protected. Values and positions of the random repeated data have no influence on the performance. Performance metrics include memory hit rate and GFLOPS. Detailed experiment results are shown in Tables 1, 2, and 3, which are calculated as the average value of 10 execution loops.

The Pre-Protected LRU achieves improvements from 1.12× in Table 2 to 1.36× in Table 1 in GFLOPS of the LRU eviction policy. The average improvement is 1.21×. Hit rate improvements achieved are from 10% to 30% better than the original LRU policy. As the vector size increases from 8 to 64, the improved performance decreases from 1.36× in Table 1 to 1.17× in Table 2. The experiment results show that the original LRU policy is sensitive to the vector size. In Table 1,

Table 3. Performance of Memory Eviction with Varying Repeated Rate: Hit Rate, GFLOPS on NVIDIA P100

Experiment Results per Iteration (Oversubscription Rate = 50%, Vector Size = 64)								
Repeated Rate	**Eviction Policy**	**Allocate**	**Evictions**	**Hit**	**Miss**	**Hit Rate**	**GFLOPS**	**Speedup**
1/8	LRU_UNIFORM	196	64	127	257	33.07%	1,346.61	1×
	LRU_GAUSSIAN	194	62	129	255	33.59%	1,362.25	1.01×
	Pre-Protected LRU	**188**	**56**	**135**	**249**	**35.15%**	**1,595.54**	**1.18×**
1/4	LRU_UNIFORM	192	60	131	253	34.11%	1,400.95	1×
	LRU_GAUSSIAN	190	58	133	251	37.23%	1,427.83	1.02×
	Pre-Protected LRU	**180**	**48**	**143**	**241**	**34.64%**	**1,720.16**	**1.23×**
3/4	LRU_UNIFORM	151	20	172	212	44.79%	1,940.44	1×
	LRU_GAUSSIAN	149	18	174	210	45.31%	1,982.20	1.02×
	Pre-Protected LRU	**148**	**16**	**175**	**209**	**45.57%**	**2,257.93**	**1.16×**

The results are calculated as the average values per vector of 10 execution loops. Repeated rate varies from 1/8 to 3/4. The vector size is 64. Oversubscribed memory is half of the available memory size.

when the tensor size is 8, the performance of Gaussian distribution is obviously better than the uniform case. This is because Gaussian distribution produces less least-used data than a uniform distribution. When the tensor size is small, the repeated data have more probability existing in the center positions, which means there is less probability to be evicted. In other cases, memory oversubscription performance is not influenced by data distribution. Additionally, Table 2 illustrates stable performances of three eviction policies with varying oversubscription rates. More specifically, both GFLOPS and improvements are close in these three cases. It is concluded that the oversubscription rate has a trivial impact on memory eviction performance. Table 3 explores the influence of the repeated rate. On one hand, the performance improvement is relatively stable, achieving about 1.2×. On the other hand, values of memory hits and GFLOPS increase as the repeated rate is higher. This is because more repeated data bring more reusability opportunities. The Pre-Protected eviction policy yields stable benefits on eliminating redundant memory evictions with varying oversubscription rate and repeated rate. In summary, evaluation results are consistent with our theoretical analysis in Section 5.4. The Pre-Protected LRU eviction policy always outperforms the LRU eviction policy in memory hits and GFLOPS. The Pre-Protected LRU eviction policy is able to avoid data thrashing and obviously eliminate redundant memory evictions.

6.5 User Case: Evaluation in Redstar System

In order to evaluate in practical scenarios, this work measures three real physics correlation functions in the Redstar system. As claimed in Section 2, calculating one correlation function includes multiple configurations with multiple time intervals. In a typical practical scenario, one correlation function produces about 400 to 500 configurations, and then one configuration executes through 64 time intervals. Each configuration obtains the identical computation and different data in all time intervals. Different configurations also represent the same computations. Therefore, the performance is measured by the average executing results of one configuration of the correlation functions with a single time interval. All three of these correlation functions belong to multi-meson system computations. Detailed information of correlation functions is shown in Table 4.

This work designs a set of experiments to integrate into the Redstar system. Execution time and GFLOPS are measured to evaluate performance improvements. Table 5 illustrates the experiment results. Execution time implies the average of 10 execution loops at a single time interval of one configuration. GFLOPS are calculated based on all the memory operations and the execution time. According to Table 5, the improved performance ranges from 3.56× to 6.12× in execution time and 3.56× to 6.08× in GFLOPS. These results support the significant benefits of MemHC on accelerating hadron contractions in real-world applications.

Table 4. Information of Real Correlation Functions

Function Name	Tensor Size	#Total Nodes	Memory (GBytes)	#Contractions
a1_rhopi	128	106	0.44	68
f0d2	256	2173	36.28	1,968
f0d4	256	2173	36.32	1,970

Basic information of three correlation functions including the tensor (spin) size, the number of initial and unique hadron nodes, the theoretical needed memory, and the number of hadron contractions.

Table 5. Performance of Real Correlation Functions: Execution Time and GFLOPS on NVIDIA P100

Function Name	Execution Time (s)			GFLOPS		
	Unified Memory	MemHC	Speedup	Unified Memory	MemHC	Speedup
a1_rhopi	0.61	0.17	**3.56**×	59.87	212.87	**3.56**×
f0d2	16.19	2.67	**6.06**×	134.52	815.22	**6.06**×
f0d4	16.33	2.67	**6.12**×	133.44	810.72	**6.08**×

Speedup means the times of the accelerated performance of MemHC.

7 RELATED WORK

Tensor contraction optimization works. Prior works focus on implementing a general method [2, 7, 22, 23, 29, 31, 33, 39, 42] to optimize an individual tensor contraction instead of a number of tensors. Some other related works optimize tensor contraction kernel for specific tensor patterns, such as sparsity [31], symmetry [22], and high rank [42]. Kim et al. [23] propose a GPU code generator of tensor contractions to leverage data reuse in a high-dimensional loop. Another work of Kim et al. [22] aims to optimize CCSD kernel computation for specific applications. Ma et al. [32] implement a code generator to translate tensor expressions to optimized CUDA codes. Nelson et al. [36] present a machine-learning-based approach to find the optimal GPU codes for tensor contraction. Different from all these efforts, MemHC targets a large number of tensor contractions on efficient memory management and redundancy eliminations.

GPU memory management frameworks. GPU is famous for dramatically speeding up the computation of various practical applications, including deep learning [46, 48, 50, 51] and scientific computing. Many existing research efforts aim to optimize GPU memory management. Li and Chapman [28] present a set of hybrid implicit or explicit data movement frameworks to optimize GPU unified memory. Dashti and Fedorova [12] discuss various popular memory management methods in heterogeneous systems including HSA, NVIDIA, and AMD hardware. Ausavarungnirun et al. [4] propose Mosaic, a new GPU memory manager that efficiently supports multiple page sizes. Compared with these works, MemHC mainly targets specific attributes, multiple memory redundancies, in many-body correlation. Other prior works mainly explore the memory oversubscription based on unified memory management. A framework ETC [27] classifies applications as regular and irregular and provides three memory oversubscription mitigation techniques. Kim et al. [21] also focus on dealing with memory oversubscription on unified memory including **Thread Oversubscription (TO)** and **Unobtrusive Eviction (UE)**. In contrast, MemHC exploits a novel eviction policy, the Pre-Protected eviction, based on an explicit and optimized GPU memory management for many-body correlation.

Memory redundancy elimination techniques. Existing work about memory redundancy elimination techniques cannot address correlation function efficiently. For instance, many efforts about accelerating neural networks [20, 37] eliminate redundant operations in registers. Other elimination redundancy works focus on GPU cache [3, 13, 16] and shared memory [8, 20]. Unlike these

efforts, one hadron node requires about 37M memory cost in a single meson system with 384 tensor size, which cannot be optimized in GPU register, cache, or shared memory. Unified Memory management [26] jointly manages host and device memory and provides memory allocation elimination techniques. However, as analyzed in Section 6, Unified Memory management produces redundant CPU/GPU memory communications when passing references of GPU objects from the host, which is a frequent operation in computing many-body correlation functions. Therefore, Unified Memory management and other conventional redundancy elimination techniques are not suitable for many-body correlation calculations.

Memory eviction policies. Currently many efforts aim to avoid evicting reusable data and leverage the memory hit rate. Popular eviction policies include Random Eviction, MRU, LRU, CAR, Clock-Pro, and more complicated policies [5, 19, 25, 38]. Clock-Pro [19] considers not only the recently referenced data but also recently evicted data, which is better than LRU in one-time scan and large loop. CAR [5] is self-tuned and theoretically more efficient than LRU. These eviction policies implement different techniques to reduce the redundant evictions but cannot avoid them completely. Compared with these efforts, the pre-protected policy utilizes the specific data structure (vector form of hadron nodes) to predict data access in advance, which can help pre-protect repeat data and avoid redundant memory evictions.

8 DISCUSSION

We discuss two future works about optimizing many-body correlation functions. On one hand, we plan to scale up the current work to multiple GPUs. The challenges include high efficiency of multi-GPU scheduling for many-body correlation calculations and memory operation reductions, especially memory communication among GPUs. On the other hand, there exist complicated correlation function systems, like tetra systems based on four-dimensional tensors. High-dimensional tensors make contraction much more complex (e.g., tensor permutations), both in memory utilization and in computation expense. In the future, we will extend the framework to address more types of hadronic systems and further optimizations on high-rank tensor contractions.

9 CONCLUSION

In the article, we present an efficient GPU memory management framework MemHC to eliminate broad memory redundancies. The redundant memory operations involve memory allocations, CPU/GPU memory communications, and oversubscription. MemHC exploits associated reduction optimizations including memory reusability optimizations, data reorganization, and on-demand synchronization. Memory reusability optimizations include duplication-aware management and overwriting lazy-released memory for duplicate data and new intermediate data. Additionally, this work designs a novel Pre-Protected LRU eviction policy to avoid redundant memory evictions and data thrashing. In evaluation, MemHC outperforms the unified memory management in general correlation functions and three real-world physics correlation functions. The improvements are able to achieve from 2.17× to 10.73× higher GFLOPS. MemHC is also widely evaluated in four architectures, NVIDIA P100, NVIDIA V100, AMD MI50, and AMD MI100, to demonstrate the robust portability and generalization. Furthermore, although this framework is built for many-body correlation functions, some new insights, like the Pre-Protected LRU eviction, are potentially helpful for other workloads. For instance, the neural network models (particularly DNN training tasks that require a significant amount of GPU memory) have pre-defined data structures to reuse intermediate results and model weights when bypassing computational graphs. The Pre-Protected LRU eviction method can help pre-protect the reusable intermediate data in advance to eliminate redundant memory evictions for large datasets. Another example is large time-evolving graph

processing that generates dynamic graph structures and provides data reuse opportunities for the repeated part of the graphs. Many current efforts focus on predicting temporal graph behaviors, which allows the Pre-Protected LRU eviction policy to pre-protect the reusable nodes and eliminate redundant memory evictions. In the future, we will explore more optimizations for multiple GPU scheduling and accelerating of complicated hadronic systems.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for making innumerable helpful suggestions and comments.

REFERENCES

[1] NVIDIA. 2015. http://docs.nvidia.com/cuda/cublas/.
[2] Ahmad Abdelfattah, Marc Baboulin, Veselin Dobrev, Jack Dongarra, Christopher Earl, Joel Falcou, Azzam Haidar, Ian Karlin, Tz Kolev, Ian Masliah, et al. 2016. High-performance tensor contractions for GPUs. *Procedia Computer Science* 80 (2016), 108–118.
[3] Neha Agarwal, David Nellans, Eiman Ebrahimi, Thomas F. Wenisch, John Danskin, and Stephen W. Keckler. Selective GPU caches to eliminate CPU-GPU HW cache coherence. In *2016 IEEE HPCA*.
[4] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 2017. Mosaic: A GPU memory manager with application-transparent support for multiple page sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 136–150.
[5] Sorav Bansal and Dharmendra S. Modha. 2004. CAR: Clock with adaptive replacement. In *FAST*.
[6] Evan Berkowitz, Thorsten Kurth, Amy Nicholson, Bálint Joó, Enrico Rinaldi, Mark Strother, Pavlos M. Vranas, and André Walker-Loud. 2017. Two-nucleon higher partial-wave scattering from lattice QCD. *Physics Letters B* (2017).
[7] Alina Bibireata, Sandhya Krishnan, Gerald Baumgartner, Daniel Cociorva, Chi-Chung Lam, P. Sadayappan, J. Ramanujam, David E. Bernholdt, and Venkatesh Choppella. 2003. Memory-constrained data locality optimization for tensor contractions. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 93–108.
[8] Guoyang Chen, Yufei Ding, and Xipeng Shen. Sweet KNN: An efficient KNN on GPU through reconciliation between redundancy removal and regularity. In *2017 IEEE ICDE*.
[9] Jie Chen, Robert Edwards, and Frank Winter. 2017. Graph-based contractions with optimal evaluation strategies. *ADSE03-LatticeQCD Application Strategy WBS 1.2.1.03* Milestone ADSE03-7 (2017).
[10] Jie Chen, Robert Edwards, and Frank Winter. 2018. Performance enhancement to the graph-based contraction calculations. *ADSE03-LatticeQCD Application Strategy WBS 1.2.1.03* Milestone ADSE03-7 (2018).
[11] Jie Chen, Robert Edwards, and Frank Winter. 2019. Enabling graph based contraction calculations for multi-nucleon systems. *ADSE03-LatticeQCD Application Strategy WBS 1.2.1.03* Milestone ADSE03-14 (2019).
[12] Mohammad Dashti and Alexandra Fedorova. 2017. Analyzing memory management methods on integrated CPU-GPU systems. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*. 59–69.
[13] Kelu Diao, Ioannis Papapanagiotou, and Thomas J. Hacker. HARENS: Hardware accelerated redundancy elimination in network systems. In *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 237–244.
[14] Feng-Kun Guo, Christoph Hanhart, Ulf-G. Meißner, Qian Wang, Qiang Zhao, and Bing-Song Zou. 2018. Hadronic molecules. *Reviews of Modern Physics* 90, 1 (2018), 015004.
[15] Chaofeng Hou, Ji Xu, Peng Wang, Wenlai Huang, and Xiaowei Wang. 2013. Efficient GPU-accelerated molecular dynamics simulation of solid covalent crystals. *Computer Physics Communications* 184, 5 (2013), 1364–1371.
[16] Mohamed Assem Ibrahim, Hongyuan Liu, Onur Kayiran, and Adwait Jog. Analyzing and leveraging remote-core bandwidth for enhanced performance in GPUs. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 258–271.
[17] John Jenkins, James Dinan, Pavan Balaji, Tom Peterka, Nagiza F. Samatova, and Rajeev Thakur. 2013. Processing MPI derived datatypes on noncontiguous GPU-resident data. *IEEE Transactions on Parallel and Distributed Systems* 25, 10 (2013), 2627–2637.
[18] John Jenkins, James Dinan, Pavan Balaji, Nagiza F. Samatova, and Rajeev Thakur. 2012. Enabling fast, noncontiguous GPU data movement in hybrid MPI+ GPU environments. In *2012 IEEE International Conference on Cluster Computing*. IEEE, 468–476.
[19] Song Jiang, Feng Chen, and Xiaodong Zhang. 2005. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *USENIX Annual Technical Conference, General Track*. 323–336.

[20] Hyeonjin Kim, Sungwoo Ahn, Yunho Oh, Bogil Kim, Won Woo Ro, and William J. Song. 2020. Duplo: Lifting redundant memory accesses of deep neural networks for GPU tensor cores. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'20)*. IEEE, 725–737.

[21] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. 2020. Batch-aware unified memory management in GPUs for irregular workloads. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[22] Jinsung Kim, Aravind Sukumaran-Rajam, Changwan Hong, Ajay Panyala, Rohit Kumar Srivastava, Sriram Krishnamoorthy, and Ponnuswamy Sadayappan. 2018. Optimizing tensor contractions in CCSD(T) for efficient execution on GPUs. In *Proceedings of the 2018 International Conference on Supercomputing*. 96–106.

[23] Jinsung Kim, Aravind Sukumaran-Rajam, Vineeth Thumma, Sriram Krishnamoorthy, Ajay Panyala, Louis-Noël Pouchet, Atanas Rountev, and Ponnuswamy Sadayappan. 2019. A code generator for high-performance tensor contractions on GPUs. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'19)*. IEEE, 85–95.

[24] Marcin Knap and Paweł Czarnul. 2019. Performance evaluation of unified memory with prefetching and oversubscription for selected parallel CUDA applications on NVIDIA Pascal and Volta GPUs. *The Journal of Supercomputing* 75, 11 (2019), 7625–7645.

[25] Swadhesh Kumar and P. K. Singh. 2016. An overview of modern cache memory and performance analysis of replacement policies. In *2016 IEEE International Conference on Engineering and Technology (ICETECH'16)*. IEEE, 210–214.

[26] Raphael Landaverde, Tiansheng Zhang, Ayse K. Coskun, and Martin Herbordt. 2014. An investigation of unified memory access performance in CUDA. In *2014 IEEE High Performance Extreme Computing Conference (HPEC'14)*. IEEE, 1–6.

[27] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. 2019. A framework for memory oversubscription management in graphics processing units. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[28] Lingda Li and Barbara Chapman. 2019. Compiler assisted hybrid implicit and explicit GPU memory management under unified address space. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16.

[29] Rui Li, Aravind Sukumaran-Rajam, Richard Veras, Tze Meng Low, Fabrice Rastello, Atanas Rountev, and Ponnuswamy Sadayappan. 2019. Analytical cache modeling and tilesize optimization for tensor contractions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[30] Wenqiang Li, Guanghao Jin, Xuewen Cui, and Simon See. 2015. An evaluation of unified memory technology on NVIDIA GPUs. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 1092–1098.

[31] Jiawen Liu, Dong Li, and Jiajia Li. 2021. Athena: High-performance sparse tensor contraction sequence on heterogeneous memory. In *ICS*.

[32] Wenjing Ma, Sriram Krishnamoorthy, Oreste Villa, Karol Kowalski, and Gagan Agrawal. 2013. Optimizing tensor contraction expressions for hybrid CPU-GPU execution. *Cluster Computing* 16, 1 (2013), 131–155.

[33] Devin A. Matthews. 2016. High-performance tensor contraction without BLAS. *SIAM Journal on Scientific Computing* 40 (2016).

[34] Alexander S. Minkin, Andrey A. Knizhnik, and Boris V. Potapkin. 2017. GPU implementations of some many-body potentials for molecular dynamics simulations. *Advances in Engineering Software* 111 (2017), 43–51.

[35] Alok Mishra, Lingda Li, Martin Kong, Hal Finkel, and Barbara Chapman. 2017. Benchmarking and evaluating unified memory for OpenMP GPU offloading. In *Proceedings of the 4th Workshop on the LLVM Compiler Infrastructure in HPC*. 1–10.

[36] Thomas Nelson, Axel Rivera, Prasanna Balaprakash, Mary Hall, Paul D. Hovland, Elizabeth Jessup, and Boyana Norris. Generating efficient tensor contractions for GPUs. In *2015 ICPP*. IEEE.

[37] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. 2020. PatDNN: Achieving real-time DNN execution on mobile devices with pattern-based weight pruning. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. 907–922.

[38] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Record* 22, 2 (1993), 297–306.

[39] Roman Poya, Antonio J. Gil, and Rogelio Ortigosa. 2017. A high performance data parallel tensor contraction framework: Application to coupled electro-mechanics. *Computer Physics Communications* 216 (2017), 35–52.

[40] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. 2008. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

[41] John E. Savage and Mohammad Zubair. 2009. Evaluating multicore algorithms on the unified memory model. *Scientific Programming* (2009).
[42] Yang Shi, Uma Naresh Niranjan, Animashree Anandkumar, and Cris Cecka. Tensor contractions with extended BLAS kernels on CPU and GPU. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. 193–202.
[43] Andres Tomas, Chia-Chen Chang, Richard Scalettar, and Zhaojun Bai. 2012. Advancing large scale many-body QMC simulations on GPU accelerated multicore systems. In *IPDPS*. 308–319.
[44] Pedro Valero-Lara, Ivan Martínez-Pérez, Raül Sirvent, Xavier Martorell, and Antonio J. Pena. 2017. NVIDIA GPUs scalability to solve multiple (batch) tridiagonal systems implementation of cuthomasbatch. In *International Conference on Parallel Processing and Applied Mathematics*. Springer.
[45] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Xiangyong Ouyang, Sayantan Sur, and Dhabaleswar K. Panda. Optimized non-contiguous MPI datatype communication for GPU clusters: Design, implementation and evaluation with MVAPICH2. In *2011 IEEE International Conference on Cluster Computing*. IEEE, 308–316.
[46] Qihan Wang, Wei Niu, Li Chen, Ruoming Jin, and Bin Ren. 2021. HEALS: A parallel eALS recommendation system on CPU/GPU heterogeneous platforms. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC'21)*. IEEE.
[47] Q. Wu, C. Brinton, Z. Zhang, M. Cucuringu, A. Pizzoferrato, and Z. Liu. 2021. Equity2vec: End-to-end deep learning framework for cross-sectional asset pricing. In *ICAIF*.
[48] Qiong Wu, Adam Hare, Sirui Wang, Yuwei Tu, Zhenming Liu, Christopher G. Brinton, and Yanhua Li. 2021. Bats: A spectral biclustering approach to single document topic modeling and segmentation. *ACM TIST* (2021).
[49] Qiong Wu, Wen-Ling Hsu, Tan Xu, Zhenming Liu, George Ma, Guy Jacobson, and Shuai Zhao. Speaking with actions-learning customer journey behavior. In *2019 ICSC*. IEEE.
[50] Qiong Wu and Zhenming Liu. 2020. Rosella: A self-driving distributed scheduler for heterogeneous clusters. *arXiv preprint arXiv:2010.15206* (2020).
[51] Qiong Wu, Felix Ming Fai Wong, Zhenming Liu, Yanhua Li, and Varun Kanade. 2019. Adaptive reduced rank regression. *arXiv preprint arXiv:1905.11566* (2019).

Received July 2021; revised November 2021; accepted December 2021