Efficient Parallelization Of Irregular Applications On Gpu Architectures

Qihan Wang
William & Mary - Arts & Sciences, qwang19@wm.edu

Efficient Parallelization of Irregular Applications on GPU Architectures

Qihan Wang

Shijiazhuang, Hebei, China

Bachelor of Software Engineering, Beihang University, 2017

A Dissertation presented to the Graduate Faculty of
The College of William and Mary in Virginia in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

The College of William & Mary in Virginia
January 2024

APPROVAL PAGE

This Proposal is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

[illegible]

Qihan Wang

Approved by the Committee, January 2024

Bin Ren

Committee Chair
Bin Ren, Associate Professor, Computer Science
William & Mary

[illegible]

Gang Zhou, Professor, Computer Science
William & Mary

Jie Ren

Jie Ren, Assistant Professor, Computer Science
William & Mary

Robert Michael Lewis

Michael Lewis, Associate Professor, Computer Science
William & Mary

Jie Chen

Jie Chen, Senior Computer Scientist
Jefferson Lab

ABSTRACT

With the enlarging computation capacity of general Graphics Processing Units (GPUs), leveraging GPUs to accelerate parallel applications has become a critical topic in academia and industry. However, a wide range of irregular applications with the computation-/memory-intensive nature cannot easily achieve high GPU utilization. The challenges mainly involve the following aspects: ***first***, data dependence leads to coarse-grained kernel and inefficient parallelism; ***second***, heavy GPU memory usage may cause frequent memory evictions and extra overhead of I/O; ***third***, specific computation patterns produce memory redundancies; ***last***, workload balance and data reusability conjunctly benefit the overall performance, but there may exist a dynamic trade-off between them.

Targeting these challenges, this dissertation proposes multiple optimizations to accelerate two real-world applications: many-body correlation functions to simulate nuclear physics in a large-scale scientific system; the other is the eALS-based matrix factorization recommendation system.

To accelerate the calculations of many-body correlation functions, this dissertation presents three frameworks in GPU memory management and multi-GPU scheduling. **Firstly**, an optimized systematic GPU memory management framework, MemHC, utilizes a series of new memory reduction designs in GPU memory allocation, CPU/GPU communications, and GPU memory oversubscription. **Secondly**, an enhanced multi-GPU scheduling framework, MICCO, particularly by taking both data dimension (e.g., data reuse and data eviction) and computation dimension into account. MICCO designs a heuristic scheduling algorithm and a machine-learning-based regression model to generate the optimal settings of a proposed new concept to manage the trade-off. **Thirdly**, a locality-aware multi-GPU scheduling framework. This scheduler leverages pipeline batch generation with a looking-ahead strategy by building local dependency graphs for memory transfer reduction and better data reuse, achieving up to 79.92% memory cost reduction and 1.67x speedup.

To parallelize the eALS-based recommendation system, this dissertation proposes an efficient CPU/GPU heterogeneous recommendation system, HEALS. HEALS employs newly designed architecture-adaptive data formats to achieve load balance and good data locality on CPU and GPU. To mitigate the data dependence, HEALS presents a CPU/GPU collaboration model for both task parallelism and data parallelism with multiple kernel computation optimizations.

In summary, this dissertation efficiently accelerates two typical irregular applications on GPUs by building four frameworks, including CPU/GPU collaboration, GPU memory management, and multi-GPU scheduling.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

The following thesis and research are the culmination of guidance, encouragement, and moral support from many people throughout my graduate studies.

First, I sincerely thank my advisor, Prof. Bin Ren, for his thoughtful, patient, and diligent mentoring. Prof. Ren taught me a lot about how to conduct successful research. Under his detailed guidance and encouragement, I accumulated professional techniques and strengthened my determination. I will carry all the lessons I learned from these years with me in my future life.

During my PhD, I am fortunate to collaborate with Dr. Jie Chen from Jefferson Lab. I sincerely appreciate his patient guidance and support. During the three-year collaboration, his professional skills and work enthusiasm inspired me to keep learning and improving. I also would like to deeply thank Prof. Gang Zhou for his support and guidance in my PhD research work. It is my honor to have a collaboration opportunity with him to broaden my knowledge and improve my research skills.

I extend my gratitude to all my dissertation committee members, Bin Ren, Gang Zhou, Jiajia Li, and Robert M Lewis, for their generous support and attentive feedback.

I thank all other professors of William & Mary for supporting my graduate studies. I also thank the Computer Science administrative staff – Vanessa Godwin, Jacqulyn Johson, and Dale Hayes – for their caring and professional work. They gave me efficient help and support to deal with daily issues.

I extend my heartfelt thanks to my colleagues from the early stages of my career, especially my tech lead manager Yan Wang. During my six-year PhD journey, I dedicated one year to working as a full-time engineer in the industry. Yan's profound expertise and insights greatly contributed to my growth and professional development. His care, support, and guidance set a brilliant foundation for my entire career.

Finally, I would like to thank my group mates and friends for making my time at William & Mary truly memorable. I also appreciate my parents, my husband, and all my other family members for their solid support, care, and encouragement without which I would not be what I am today.

To my family.

LIST OF TABLES

LIST OF FIGURES

Efficient Parallelization of Irregular Applications on GPU Architectures

Chapter 1

Introduction

Graphics Processing Units (GPUs) support high-performance computing, such as accelerating real-world applications, particularly very large datasets. However, different from fine-grained applications, it is challenging to implement efficient parallel irregular applications on GPUs. Currently, the main challenges are provoked by both computation patterns and GPU memory limitations. More specifically, data dependence of the original sequential algorithm leads to coarse-grained computation. Meanwhile, dealing with large datasets requires heavy memory usage, which causes memory oversubscription and low GPU utilization. Additionally, how to manage the interplay between computation amount (load balance) and I/O reduction (data reuse) is also a critical challenge.

Targeting the challenges, we present three works to implement parallelism and acceleration of irregular applications on GPUs. Specifically speaking, we focus on exploring two real-world applications: an eALS-based recommendation system and many-body correlation function calculations. Based on these two applications, we propose three frameworks including CPU/GPU collaboration to achieve efficient concurrency, GPU memory management to eliminate redundancies, and a multi-GPU scheduler for the optimal trade-off between load balance and data reuse.

1.1 Thesis Statement

1.1.1 Parallel Recommendation System

A recommendation system is a fundamental building block of many real-world applications ranging from online shopping, and social networking, to short video sharing and media business, etc [13, 58]. Alternating Least Square (ALS) is a classic algorithm to solve matrix factorization [107, 70, 21, 85, 30, 53] widely used in recommendation systems. Existing efforts focus on parallelizing ALS on multi-/many-core platforms to handle large datasets. Recently, an optimized ALS variant called $eALS$ was proposed, and it yields significantly lower time complexity and higher recommending accuracy than ALS. However, it is challenging to parallelize eALS on modern parallel architectures (e.g., CPUs and GPUs) for the following reasons: ***First***, eALS' data dependence prevents it from fine-grained parallel execution, thus eALS cannot fully utilize GPU's massive parallelism; ***Second***, the sparsity of input data causes poor data locality and unbalanced workload; ***Third***, its large memory usage cannot fit into GPU's limited on-device memory, particularly for real-world large datasets.

1.1.2 Parallel Many-body Correlation Functions

Calculation of many-body correlation functions is one of the critical kernels utilized in many scientific computing areas, especially in Lattice Quantum Chromodynamics (Lattice QCD) [11, 15, 16, 17, 83, 6]. It is formalized as a sum of a large number of contraction terms each of which can be represented by a graph consisting of vertices describing quarks inside a hadron node and edges designating quark propagations at specific time intervals. Graph construction thus is defined as deleting one edge after another, which carries the computing cost of tensor contractions, until two connected hadron nodes are left. Due to its computation- and memory-intensive nature, real-world physics systems (e.g., multi-meson or multi-baryon systems) explored by Lattice QCD prefer to leverage GPUs.

Distinguished from general graph processing, many-body correlation function calculations show two specific features: a large number of computation-/data-intensive kernels and frequently repeated appearances of original and intermediate data. The former results in expensive memory operations such as tensor movements and evictions. The latter offers data reuse opportunities to mitigate the data-intensive nature of many-body correlation function calculations.

Existing optimizations [60, 79, 56, 2, 47, 12, 63, 46, 74] on many-body correlation mainly focus on individual tensor contractions (e.g., cuBLAS libraries and others). Compared with the prior works, this dissertation discovers a new optimization dimension, memory redundancy eliminations, for many-body correlation by exploring the optimization opportunities among tensor contractions. Additionally, current graph-based multi-GPU schedulers cannot capture the *data-centric* features claimed before, resulting in a sub-optimal performance for many-body correlation function calculations. Therefore, this dissertation explores a multi-GPU scheduling scheme to further accelerate many-body correlation calculations.

1.2 Contributions

First, this dissertation proposes an efficient CPU/GPU heterogeneous recommendation system based on fast eALS *for the first time* (called HEALS) that consists of a set of system optimizations. HEALS employs newly designed architecture-adaptive data formats to achieve load balance and good data locality on CPU and GPU. HEALS also presents a CPU/GPU collaboration model that can explore both task parallelism and data parallelism. HEALS also optimizes this collaboration model with data communication overlapping and dynamic workload partition between CPU and GPU. Moreover, HEALS is further enhanced by various parallel techniques (e.g., loop unrolling, vectorization, and GPU parallel reduction). Evaluation results show that HEALS outperforms other state-of-the-art baselines in both performance and recommendation quality. Particularly, HEALS achieves up to 5.75× better performance than a state-of-the-art ALS

GPU library. This work also demonstrates the possibility of conducting fast recommendations on large datasets with constrained (or relaxed) hardware resources, e.g, a single CPU/GPU node.

Second, this dissertation 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. MemHC 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, 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. Therefore, we implement targeting 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. 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$\times$ to 10.73$\times$. The proposed Pre-Protected LRU policy outperforms the original LRU policy by up to 1.36$\times$ improvement.

Third, this dissertation presents a multi-GPU scheduling framework, MICCO, to accelerate contractions for correlation functions particularly by taking the *data dimension* (e.g., data reuse and data eviction) into account. This work first performs a comprehensive study on the interplay of *data reuse* and *load balance*, and designs two new concepts: *local reuse pattern* and *reuse bound* to study the opportunity of achieving the optimal trade-off between them. Based on this study, MICCO proposes a heuristic scheduling algorithm and a machine-learning-based regression model to generate the optimal setting of reuse bounds. Specifically, MICCO is integrated into a real-world Lattice QCD system, Redstar, *for the first time* running on multiple GPUs. The evaluation demonstrates MICCO outperforms other state-of-art works, achieving up to 2.25× speedup in synthesized datasets, and 1.49× speedup in real-world correlation functions.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 introduces a parallel eALS matrix factorization-based recommendation system on CPU/GPU architectures. In Chapter 3, we present an optimized GPU memory management scheme to efficiently calculate many-body correlation functions. Chapter 4 further proposes an enhanced multi-GPU scheduling framework to accelerate many-body correlation functions. Finally, Chapter 5 summarizes this dissertation and discusses the ongoing and future research works.

Chapter 2

HEALS: A Parallel eALS Recommendation System on CPU/GPU Heterogeneous Platforms

2.1 Introduction

A recommendation system is a fundamental building block of many real-world applications ranging from online shopping, social networking, to short video sharing and media business, etc [13, 58]. With rapidly increasing data volumes, exploiting more efficient (and accurate) recommendation models has been attracting attention from the fields of information retrieval, machine learning, and high-performance computing fields [68, 99] etc.

Matrix factorization and its variants [107, 70, 21, 85, 30, 53] have been widely studied and among the most prevalent approaches for recommendation. In essence, the input data forms a sparse matrix whose elements either represent ratings (explicit), or implicit feedback from users (such as click or add-cart). The goal of the factorization aims to produce two (dense) matrices, a user matrix and an item matrix, whose product can be used to recover the sparse matrix. Especially, the missing elements in the sparse matrix can be then approximated by the multiplication between the user and item matrices. We can then recommend the top-rated missing items to each user.

There are different optimization methods for matrix-factorization based recommendation, and among them, **Alternating Least Square (ALS)** [32, 73, 35] is one of the state-of-the-art methods due to its simplicity, ease of implementation, and (recommendation) accuracy. It is also rather efficient if the data is not very large. Intuitively, ALS will fix one matrix (for instance, the user matrix), and then optimize the other matrix (for instance, the item matrix); then we will fix the latter matrix and optimize the first one. This alternating optimization will perform iterative until it converges.

However, when the data size becomes larger (i.e., the number of users and the number of items), ALS also requires computation efficiency and memory capacity. To efficiently train large datasets [61], various ALS-based parallel recommendation models are implemented on many-core architectures, such as LIBMF [21] and CuMF libraries [85, 84]. To further accelerate ALS algorithm, recently He's work [30] proposes an optimized ALS algorithm, fast element-wise Alternating Least Square (eALS) algorithm and shows its outperforming convergent speed (and recommendation accuracy) compared with other MF-based recommendation models.

However, it is challenging to parallelize eALS on modern parallel architectures like GPUs [59]: First, data dependence in the fast eALS algorithm prevents it from fine-grained workload partition, thus eALS cannot fully utilize GPU's massive parallelism. Second, implicit information constitutes large sparse matrices, resulting in poor data locality and workload unbalance. Third, addressing large datasets is challenging for GPUs with limited on-device memory.

Targeting these challenges, this work proposes a CPU/GPU heterogeneous recommendation framework (HEALS) based on implementing an efficient parallel fast eALS algorithm. HEALS employs a hybrid CPU/GPU collaboration model to alleviate the impact of data dependence. HEALS accomplishes not only data movement overlapping for distinct GPU kernels but concurrent kernels between CPU and GPU as well. Moreover, HEALS reorganizes both sparse and dense matrices to new architecture-adaptive formats to improve workload unbalance and data locality. HEALS partitions data into several chucks to handle large datasets. HEALS also leverages several classic system techniques to further accelerate recommendation. The key contributions are summarized as follows:

- We propose a CPU/GPU heterogeneous recommendation system, HEALS, based on an efficient parallel eALS-based matrix factorization *for the first time*.
- We design an architecture-adaptive data format for GPU and CPU to solve workload unbalance. Additionally, HEALS unfolds and packs dense matrices for better data locality.
- We present a hybrid CPU/GPU collaboration model incorporating data parallelism, task parallelism, and overlapped data transfer. It also adopts a data partition adjustment approach to balance workload between concurrent kernels.
- We apply crucial hardware-based accelerating techniques to further accelerate kernel computation, including loop transformation and GPU parallel reduction.

HEALS is extensively evaluated on four datasets by comparing with three other state-of-the-art works. To further validate its prediction accuracy and recommendation quality, HEALS is also evaluated on two metrics, Root Mean Square Error (RMSE) and Normalized Discounted Cumulative Gain (NDCG). Evaluation results demonstrate that HEALS outperforms other ALS-based parallel libraries. Particularly, it runs 5.75× faster than CuMF (a state-of-the-art GPU library), and 15.7× faster than LIBMF (a state-of-the-art CPU library), respectively. HEALS also demonstrates the possibility of performing fast recommendations on large datasets with constrained (or relaxed) hardware resources.

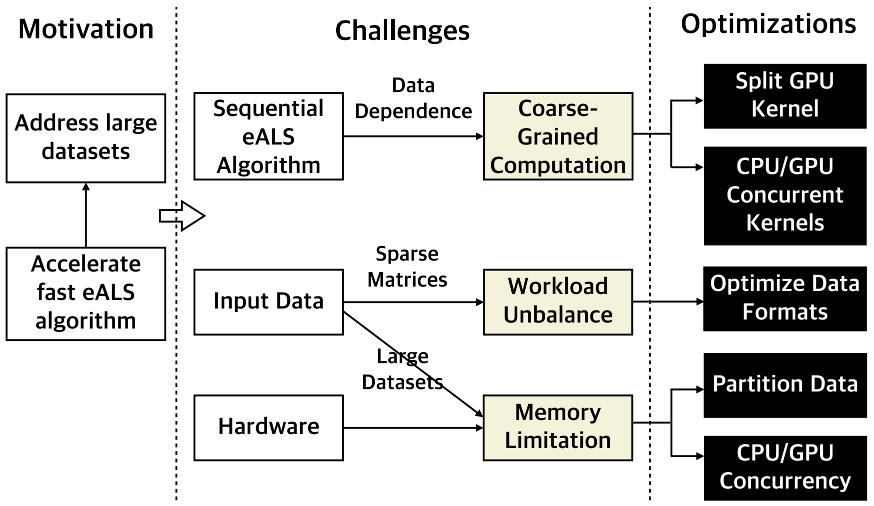


Figure 2.1: **Problem Statement: Motivation, Challenges, and Optimizations.**

2.2 Problem Statement

Fig. 3.4 illustrates the overview of the problem statement. The overall motivation is to accelerate recommendations for large datasets by designing a parallel fast eALS algorithm with the help of the superior computing capability of a heterogeneous CPU and GPU system. This section introduces three prerequisites of this work, explains the corresponding challenges, and summarizes some specific solutions. More specifically, it analyzes three sub-topics: coarse-grained computation, workload unbalance, and memory limitation.

2.2.1 Coarse-Grained Computation

As a variant of ALS, the sequential eALS algorithm[30] offers a theoretical guarantee of low computation cost and fast convergence, inspiring us to build our efficient parallel recommendation system based on this state-of-the-art algorithm. However, the original eALS algorithm leads to data dependence, which limits the performance of parallelism. Part of the kernel computation is prone to coarse-grained. More specifically, the kernel computation consists of three loops, and data dependence exists in the inner loops. We attempt to relax this inner-loop dependence by unrolling the inner loop, but this straightforward method significantly degrades the accuracy because it violates the theoretical guarantee. Therefore, this work focuses on utilizing a set of **system optimizations** to alleviate the influence of data dependence. This work splits the kernel computation into a CPU part and a GPU part and designs a CPU/GPU collaborative processing model.

2.2.2 Workload Unbalance

Workload unbalance is another bottleneck when calculating sparse matrices in eALS. The computation cost of different rows may vary dramatically for sparse matrices, directly leading to workload unbalance. A straightforward solution is to designate different numbers of threads to compute different rows; however, it will cause a significant overhead of threads scheduling. It is also time-consuming to quantify the workload of each row and decide which threads to use, especially for large datasets. Comparing with the above approach, it is more efficient to reorganize input data in advance. Thus, this work designs new data formats and divides the original data into groups with a more balanced workload. Threads are able to directly deal with reorganized groups, without any extra management in kernel computations.

2.2.3 Memory Limitation

Usually, large datasets cannot fit into the limited on-device memory of a single GPU. Two possible ways to solve this problem include scaling up to multiple GPUs and applying a CPU/GPU heterogeneous design. CuMF[84] selects the former to enlarge the whole memory size; however, it depends on the availability of hardware. In contrast, this work designs and implements a CPU/GPU concurrent execution model to leverage the large host memory efficiently **with relaxed hardware requirements**.

Table 2.1: Symbol Definitions

Name	Definitions
R	Sparse rating matrix
M	Number of users
N	Number of items
U	Dense user matrix
V	Dense item matrix
W	Sparse weight matrix
K	Number of factors
T	Predicted training matrix
L	Loss function
λ	Parameter to control the regularization

2.3 Algorithm Analysis

Before discussing the parallel implementation, this section explains the original eALS and its sequential implementation. It mainly explains eALS' theoretical definitions and analyzes its computation patterns. Tab. 2.1 illustrates the definitions of symbols used in the following sections.

2.3.1 Fast eALS Algorithm

Fast eALS algorithm aims to solve matrix factorization. Matrix factorization decomposes one matrix (usually a sparse matrix, e.g., a user-item rating matrix) into the product of two lower dimensional matrices (e.g., a user matrix and an item matrix). According to the Tab. 2.1, $R \in R^{M \times N}$, $U \in R^{M \times K}$, $V \in R^{N \times K}$. The matrix factorization can be defined as:

$$R = U \times V^T \tag{2.1}$$

A loss function L calculates the loss value between the predicted training matrix (T) and the ground truth matrix (R). Let r, u, and v represent elements of matrices R, U, and V. The loss function is calculated as:

$$L = \sum_{i=1}^{M}\sum_{j=1}^{N} w_{ij}(r_{ij} - \hat{r_{ij}})^2 + \lambda(\sum_{i=1}^{M} ||u_i||^2 + \sum_{j=1}^{N} ||v_j||^2) \tag{2.2}$$

To optimize the ALS algorithm, eALS (**e**lement-wise ALS) [30] introduces two new attributes, the element-wise learner and popularity-aware strategy. With this optimization, eALS separates the rated and unrated elements in matrix R, then assigns unrated elements a confidence value c. Here is the optimized loss function L:

$$L = \sum_{i,j \in R} w_{ij}(r_{ij} - \hat{r_{ij}})^2 + \sum_{i=1}^{M}\sum_{j \notin R_i} c_j \hat{r_{ij}})^2 + \lambda(\sum_{i=1}^{M} ||u_i||^2 + \sum_{j=1}^{N} ||v_j||^2) \tag{2.3}$$

The confidence c comes from the popularity fi that denotes the popularity of item i. The definition of c is:

$$c_i = c_0 \frac{f_i}{\sum_{j=1}^{N} f_j} \tag{2.4}$$

To minimize this loss function, we need to get the deviation and compute the updating rules of matrices U and V. The updating formulas of u_{if} and v_{jf} are defined as:

$$u_{if} = \frac{\sum_{j \in R}[w_{ij} r_{ij} - (w_{ij} - c_j)\hat{r}_{ij}^{f}]v_{jf} - \sum_{k \neq f} p_{ik} \times sv_{kf}}{\sum_{j \in R}(w_{ij} - c_i)v_{jf}^2 + sv_{ff} + \lambda} \tag{2.5}$$

$$v_{jf} = \frac{\sum_{i \in R}[w_{ij} r_{ij} - (w_{ij} - c_j)\hat{r}_{ij}^{f}]u_{if} - c_j \sum_{k \neq f} p_{ik} \times su_{kf}}{\sum_{j \in R}(w_{ij} - c_i)u_{if}^2 + c_j \times su_{ff} + \lambda} \tag{2.6}$$

As shown in Alg. 1, kernel computation is to update dense matrices U and V in formula (5) and (6). Take computing U as an example, after a matrix multiplication to compute S^v, the kernel consists of three *for* loops. The first loop is to compute every row of U, which is able to execute concurrently. In the middle *for* loop, every column is accessed in U, and the dependence exists in this loop. To be more specific, the sparse matrix R is updated before and after updating U. Then the updated R should be used in the next

Algorithm 1 Fast eALS Algorithm [30]

```
Require: $R$, $W$, $c$, $K$, $\lambda$
Ensure: Matrices $U$, $V$
1: Initialize $U$ and $V$ randomly
2: for $i, j \in R$ do
3:     $\hat{r_{ij}} = Eq.(1)$
4: end for
5: for Stopping criteria is not met do
6:     $S^v = \sum_{i=1^N} c_i v_i v_i^T$                          ▷ Update user factors
7:     for $i = 1; i \leq M; i++$ do
8:         for $f = 1; f \leq K; f++$ do
9:             for $j \in R_i$ do
10:                $\hat{r_{ij}^f} = \hat{r_{ij}} - u_{if} v_{jf}$
11:                $u_{if} = Eq.(5)$
12:            end for
13:            for $j \in R_i$ do
14:                $\hat{r_{ij}^f} = \hat{r_{ij}} + u_{if} v_{jf}$
15:            end for
16:        end for
17:    end for
18:    $S^u = U^T \times U$                                          ▷ Update item factors
19:    for $j = 1; j \leq M; j++$ do
20:        for $f = 1; f \leq K; f++$ do
21:            for $i \in R_j$ do
22:                $\hat{r_{ij}^f} = \hat{r_{ij}} - u_{if} v_{jf}$ $v_{jf} = Eq.(6)$
23:            end for
24:            for $i \in R_j$ do
25:                $\hat{r_{ij}^f} = \hat{r_{ij}} + u_{if} v_{jf}$
26:            end for
27:        end for
28:    end for
29: end forreturn $U$ and $V$
```

iteration to update the next element in one row of U. The data dependence limits the efficiency of the parallel fast eALS algorithm implementation.

Tab. 2.2 shows detailed comparisons between the original ALS implementation in CuMF_ALS[84] and the fast eALS algorithm. ALS consists of two main kernels, matrix multiplication (`get_hermitian_x`) and matrix inverse (`batch_solve`). No data dependence exists as partitioning the input matrices into tiles for both kernels. ALS's matrix inverse (`batch_solve`) consists of two solving functions, LU and Conjugate Gradient (CG). LU's complexity is K-time higher than eALS. CG is an estimated method and its iteration number is set to be 6 in the programs, so the time complexity of the CG solver is the same as the initial prediction in the fast eALS algorithm. If $(M + N)K$ is much larger than $|R|$, $(M + N)K^2$ is dominant in time complexity. In summary, the original ALS has obvious higher time complexity than fast eALS, while the optimized CG solution of ALS in CuMF

Table 2.2: Compare CuMF_ALS and fast_eALS: Time Complexity and Computation Patterns

Algorithm	Function	Time Complexity	Computation Pattern	Dependence
CuMF_ALS	`get_hermitian_x`	$\|R\|K^2$	Matrix multiplication	No
CuMF_ALS	`batch_solve`	$\frac{(M+N)K^3}{(M+N)K^2 * It\_CG}$	Matrix inverse Matrix multiplication	No
fast_eALS	`updateUser/updateItem`	$\|R\|K$	Irregular access	Yes
fast_eALS	`initialPredictions`	$(M+N)K^2$	Matrix multiplication	No

has the same time complexity as fast eALS. Besides time complexity, convergence speed is another critical factor to evaluate recommendation systems. In He *et al.*'s work[30], they conclude that the fast eALS algorithm has better recommendation quality than the ALS algorithm, which are compared in Section 2.8.

2.3.2 Computation Pattern Analysis

Compared with the original ALS, the fast eALS has a more complicated computation kernel with data dependency. Fig. 2.2 shows the access patterns of the fast eALS with an example highlighted with a yellow background. Target matrix and dense matrix are two dense matrices, defined as U and V. The objective is to use R to estimate U and V. In the left part of Fig. 2.2, the goal is to update the red cross element (in row 1 and column 2) of the target matrix U. The first row of two sparse matrices (R and W) and the second column of the dense matrix (V) will be accessed to compute the selected target element (①). Next, the updated red element of the target matrix (U) and the second column of the dense matrix (V) are used to update the first row of the sparse matrix (R) (②), as shown in the right part of Fig. 2.2.

Updating each row of U raises data dependence among columns, while no dependence occurs between rows. Take updating one row of U into consideration. Before computing the second element (i.e., the red cross element in the first row of U), the first row of R is necessarily updated. However, this row of R is computed after updating the first element (i.e., the black cross one in the first row of U). More specifically, computing one

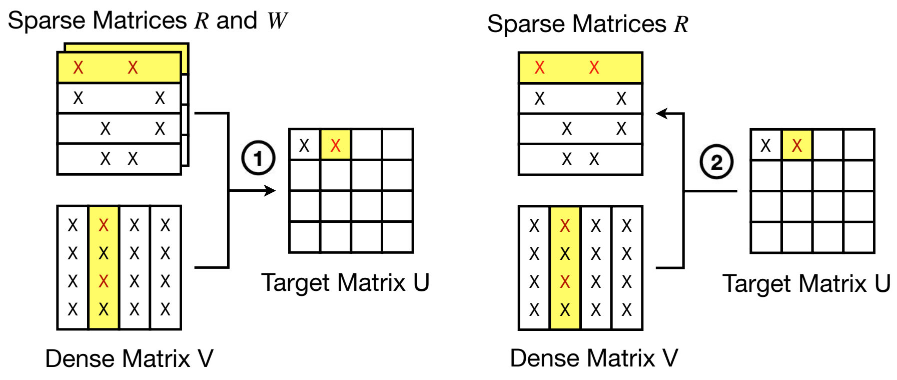


Figure 2.2: **Kernel Computation Pattern.** Kernel computation mainly involves two stages: one is to utilize R, W, and V to update the target matrix U (①); the other is to calculate R based on two dense matrices U and V (②). Highlighted elements participate in calculating one element of U. Updating the second element (red cross one) in the first row of U requires the first row of R, but this row of R is necessarily updated after computing the first element (black cross one) of U. **Dependence exists in the same row of the matrix** U, i.e., one element of U depends on the updated values of all previous elements in the same row.

element of U requires the associated row of R, and this row of R is calculated based on the updated previous elements of U. Computing one element of U has to wait until all previous elements accomplish updating. Thus, data dependence exists in updating the same row of U. Different rows of U can execute concurrently without any dependence.

2.4 System Overview

Based on the background and algorithm analysis, this work presents an efficient parallel eALS recommendation framework on CPU/GPU heterogeneous systems (called HEALS). Fig. 2.3 illustrates HEALS's overview that consists of two main parts: architecture-adaptive data format and hybrid CPU/GPU collaboration model.

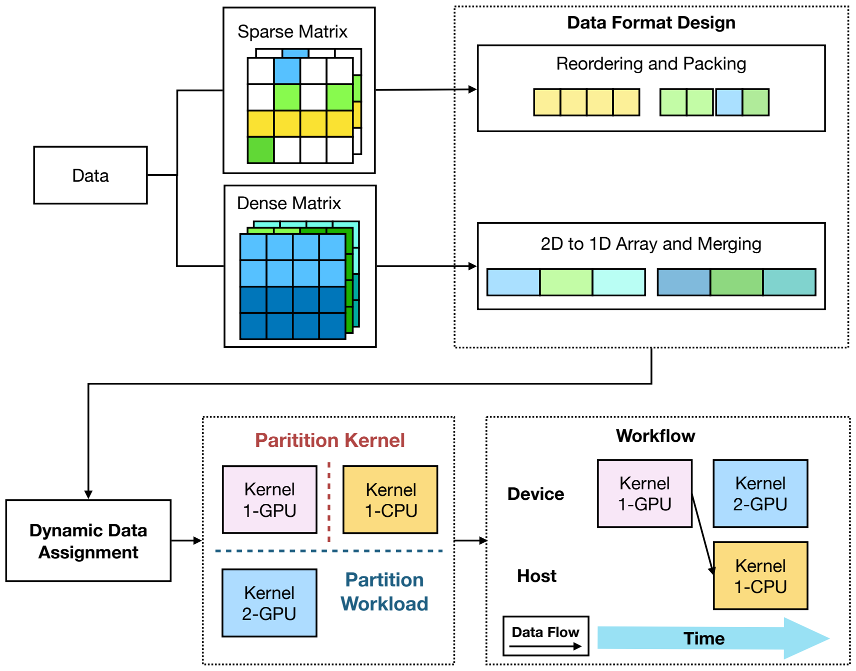


Figure 2.3: **System Overview: architecture-adaptive data format and hybrid CPU/GPU collaboration model.** Sparse matrices are reordered and packed into groups. Dense matrices are reorganized into a merged one-dimensional array. The optimized data are fed into the hybrid CPU/GPU collaboration model with a dynamic data assignment. Kernel computation includes two types: (a) **partitioning kernel into Kernel 1-GPU (partial computation kernel w/o dependence on GPU) and Kernel 1-CPU (dependence part on CPU)**; (b) **Kernel 2-GPU (complete kernel on different data)**.

For input data, sparse matrices and dense matrices have different optimizations. On the one hand, HEALS reorders sparse matrices and pack them into groups for better workload balance. The new data formats are designed based on well-known CSR [95, 33]. Moreover, new data formats are designed to be adaptive for specific architectures, based on hardware characteristics. On the other hand, in order to improve data locality, HEALS linearizes dense matrices from 2D to 1D array and merges multiple matrices to one array.

After pre-processing input data, the data will be assigned to the hybrid CPU/GPU collaboration model. HEALS **partitions kernel computation into Kernel 1-GPU**

(partial computation kernel without dependence on GPU) and Kernel 1-CPU (partial computation kernel with dependence on CPU). To further leverage concurrency, HEALS performs Kernel 2-GPU to hide the waiting latency of the device. **Kernel 2-GPU processes different data from Kernel 1-GPU/CPU and accomplishes the complete kernel computation**. Additionally, HEALS dynamically assigns data chunks with optimized proportion to achieve better workload balance.

2.5 architecture-adaptive data format

In order to reduce workload unbalance and improve data locality, HEALS presents architecture-adaptive data format for sparse matrices and dense matrices. Targeted workload unbalance, prior works propose various data format optimizations. For instance, Hong *et al.*'s paper[33] reorders the data and packs different rows. The data formats have two extra index arrays and a complicated data structure, which is suitable for simple kernel computation like matrix multiplication instead of matrix factorization. With respect to the kernel computation patterns, this work designs architecture-adaptive data format, including optimizing sparse matrices for GPU and CPU respectively, and dense matrix transformation.

2.5.1 Sparse Matrix

The sparse matrix optimization is shown in the upper part of Fig. 2.4. Before compressing the sparse matrix, HEALS reorders both rows and columns based on the number of rated elements. Then HEALS divides the whole sparse matrix into two parts: a **dense part** and a **sparse part**. Since GPU is beneficial for the computation-intensive kernel, the dense part is fed to GPU, while the sparse part to CPU.

As claimed in Section 5.1, workload unbalance relies on the varied row lengths of sparse matrices. To improve workload balance, the key idea is to divide the rows into groups and balance the number of elements in each group for each thread. HEALS utilizes the *greedy*

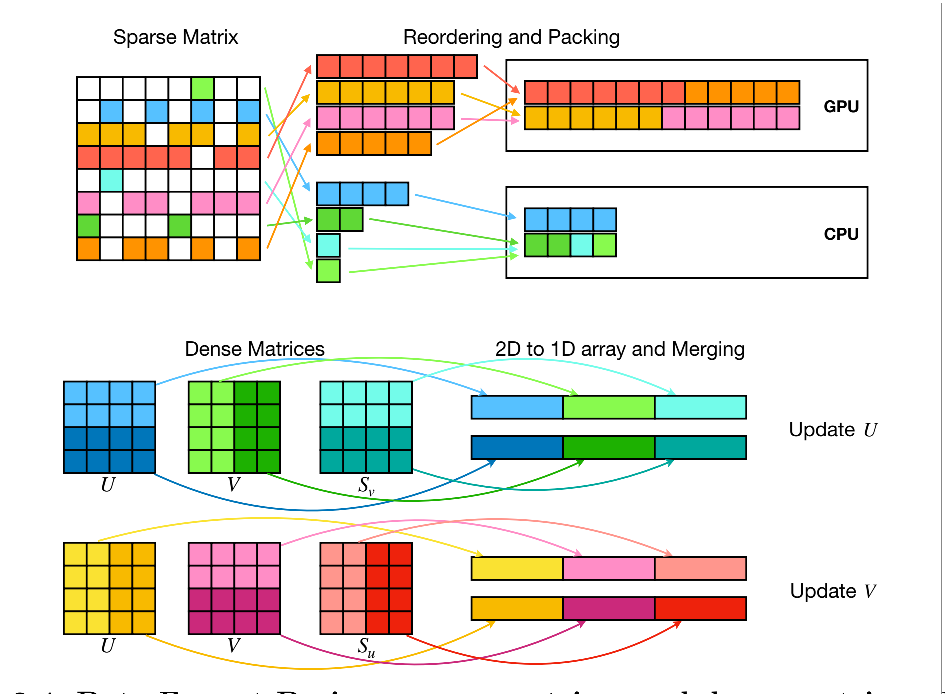


Figure 2.4: **Data Format Design: sparse matrices and dense matrices.** HEALS designs new architecture-adaptive data format to solve workload unbalance on GPU and CPU. Sparse matrices on GPU are reorganized with balanced group size, while on CPU adopt uncertain size groups but more even computation cost. For dense matrices, HEALS translates multiple matrices into a combined one-dimensional array for better data locality.

algorithm to distribute multiple rows into groups, and generate even groups. This new data format is named *Multiple Packed Compressed Sparse Row* (*MP-CSR*). The detailed steps are:

- **Step 1:** Reorder the matrix based on the row size.
- **Step 2:** Select the smallest row, and assign it to the current smallest group.
- **Step 3:** Continue selecting and assigning until all the rows are allocated. Each group contains an uncertain number of rows.

Since the data size of every group may be distinct, *MP-CSR* manages **an extra array** to record the first index position of every group. However, an extra array requires more memory space, resulting in extra access operations and more cache misses, especially for GPU. Thus, HEALS proposes another grouping approach, in which it assigns each group the same number of rows and balances the total number of elements in these rows. After reordering the rows, HEALS packs the longest one and shortest one together as a group. This new data format is named *Nested Packed Compressed Sparse Row* (*NP-CSR*). *NP-CSR* implies dividing multiple rows into even groups. The number of rows in the dense part is set to be even multiple numbers of the group. The number of groups depends on the input data size and GPU layouts. The detailed steps are:

- **Step 1:** Reorder the matrix based on the row size.
- **Step 2:** Select the largest row and smallest row, then assign them to one group.
- **Step 3:** Continue selecting and assigning until all the rows are allocated. Each group contains the same number of rows.

Compared with *NP-CSR*, *MP-CSR* has a better workload balance but a more complicated data structure. In contrast to GPU, CPU is more suitable to conduct an extra array with a larger cache and memory size. Therefore, **GPU employs *NP-CSR*, while CPU employs *MP-CSR***.

2.5.2 Dense Matrix

For dense matrices, HEALS linearizes a two-dimensional matrix into a one-dimensional array then merges dense matrices. When accessing data, the increased possibility of accessing near elements in different threads improves data locality. Packing three matrices together is also able to reduce data movements. For small datasets, dense matrices are easy to reorganize. For large dense matrices, HEALS partitions data to fit the limited global memory of GPU. As shown in Fig. 2.4, different color areas of dense matrices represent different partitions. Based on the access pattern in Fig. 2.2, light area in three matrices are loaded to deal with one partition of the target matrix. The partition organization depends on the input data size.

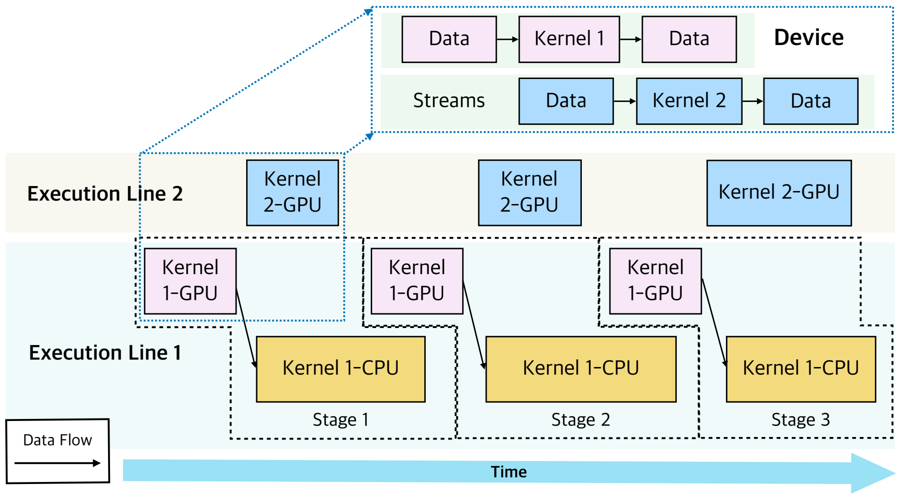


Figure 2.5: Framework Workflow: hybrid CPU/GPU collaboration model. Multi-level concurrency includes: (a) two execution lines constitute **data parallelism**; (b) Kernel 1-GPU and Kernel 1-CPU achieve **task parallelism**; (c) **Overlapped data transfer** hides latency between Kernel 1-GPU and Kernel 2-GPU. Additionally, computation is partitioned into several stages to handle very large datasets.

2.6 Hybrid CPU/GPU Collaboration Model

This section introduces hybrid CPU/GPU collaboration model in HEALS. Some related works implement CPU/GPU concurrent schemes to solve matrix factorization. Tsai *et al.*'s work [89] separates kernel computation to solve QR matrix factorization with a much longer execution time on CPU than GPU. Teodoro *et al.*'s work [86] designs a performance-aware method for multi-GPUs with extra scheduling overhead. In contrast to prior approaches, HEALS exploits a hybrid CPU/GPU collaboration model, combining *Workload Partition (WP)* and *CPU-GPU Pipeline (CGP)* collaboration models [82]. HEALS dynamically adjusts data partitioning proportion for balance workload and optimal overlapping.

2.6.1 Multi-level Concurrency Design

Fig. 2.5 illustrates the design of hybrid CPU/GPU collaboration model. To process large datasets, HEALS partitions the whole data into chunks. The scheduling framework solves chunks stage by stage. In each stage, kernel computation is divided into two execution lines: one executes on Kernel 1-GPU and Kernel 1-CPU, and another runs on Kernel 2-GPU. The two execution lines process independent data.

Kernel 1 represents the first execution line, including two kernels: kernel without data dependence and the rest computation with dependence. The no dependence computation is named Kernel 1-GPU, executing on GPU. The rest of kernel computation is Kernel 1-CPU on CPU. They are combined together to solve a complete kernel for the same data chunks. Kernel 1-CPU can only start after Kernel 1-GPU.

However, Kernel 1-GPU is more than five times faster than Kernel 1-CPU to process identical data chunks. The significant GPU waiting time affects the performance. To improve GPU utilization, HEALS applies another execution line, aiming to concurrently execute with Kernel 1-CPU. To avoid any conflicts, this execution line deals with separate data chunks on GPU. This kernel is named Kernel 2-GPU to implement the whole kernel computation. Only executing Kernel 2-GPU is not efficient enough due to the data dependence. Therefore, the optimal design is to incorporate these two execution lines, to fully utilize both GPU and CPU. If the dataset is large, the computation will be partitioned into several stages. Due to the sequential execution of kernels on the device, multiple stages constitute a pipeline model.

Based on Sun*et al.*'s work [82], HEALS combines two CPU/GPU collaboration models: **Workload Partition (WP)** and **CPU-GPU Pipeline (CGP)**. On one hand, Kernel 2-GPU and Kernel 1-CPU solve independent workload concurrently as a WP collaboration model. On other hand, the first execution line partitions kernel computation in multiple stages, formalized as a CGP collaboration model. Furthermore, HEALS achieves **multi-level concurrency**. As shown in Fig. 2.5, the concurrency consists of three major aspects:

task parallelism between Kernel 1-CPU and Kernel 1-GPU, *data parallelism* between two execution lines (i.e., Kernel 1 and Kernel 2), and *overlapped data transferring* between two GPU kernels.

2.6.2 Adjusting Data Partition Dynamically

This work designs a dynamic partition model to calculate the ideal proportion and adjusts it in the following iterations. According to Fig. 2.5, HEALS partitions the data into two independent execution lines, so it is critical to balance the workload between the two execution lines. When training the complete parallel fast eALS model, the iteration number is at least 50 to achieve a satisfying accuracy. After executing on the first iteration, HEALS obtains execution time and builds bivariate linear equations to compute the ideal partition ratio. Subsequently, the adjusted proportion is applicable in the following iterations.

Take one stage as an example to explain how to compute the ideal partition ratio. This work defines the execution time of Kernel 1-GPU to be t_1, Kernel 2-GPU to be t_2, and Kernel 1-CPU to be t_3. First, HEALS allocates θ tiles to solve. Tiles mean the groups of rows of the sparse matrices. Second, α tiles are allocated to Kernel 2-GPU and β to Kernel 1. The most overlapped case is $t_1 + t_2 = t_3$, between Kernel 1 and Kernel 2. Last, the equations are built as follows: $\theta = \alpha + \beta$, $\alpha \times t_2 + \beta \times t_1 = \beta \times t_3$.

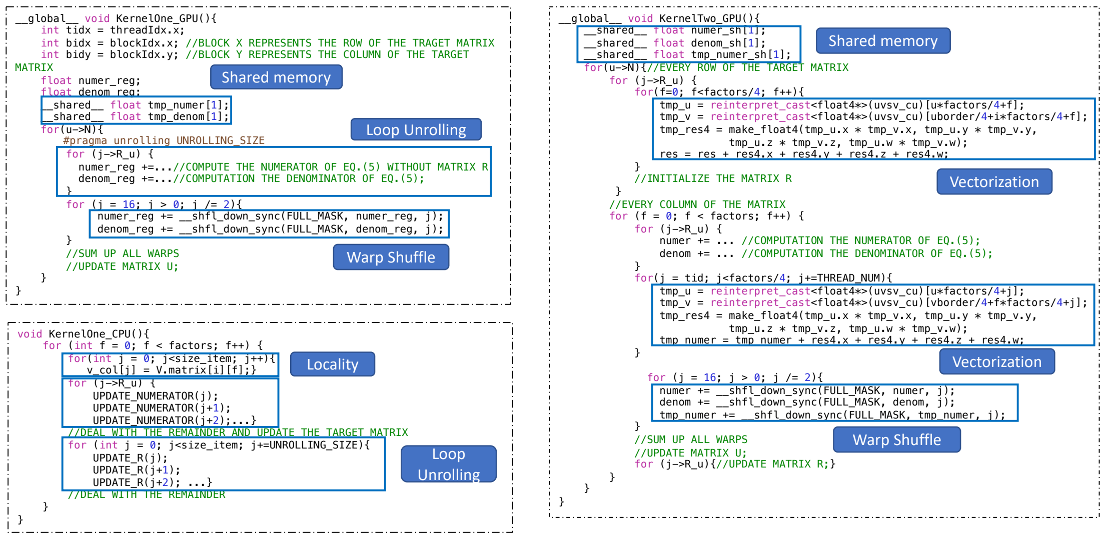
The partition ratio is set to be $\gamma = \frac{\alpha}{\beta}$. Execution time t_1, t_2, t_3, and the total number of tiles θ are given. The objective is to calculate γ, β and α. After solving the bivariate linear equations, results are shown as follows:

$$\gamma = (t_3 - t_1)/t_2 \tag{2.7}$$

$$\beta = \frac{t_2\theta}{t_3 - t_1 + t_2} \tag{2.8}$$

$$\alpha = \frac{(t_3 - t_1)\theta}{t_3 - t_1 + t_2} \tag{2.9}$$

If the data size is so large to be managed more than one stage, all stages will compute ratios individually after the first iteration and record them in a global array.



```
__global__ void KernelOne_GPU(){
    int tidx = threadIdx.x;
    int bidx = blockIdx.x; //BLOCK X REPRESENTS THE ROW OF THE TRAGET MATRIX
    int bidy = blockIdx.y; //BLOCK Y REPRESENTS THE COLUMN OF THE TARGET
MATRIX
    float numer_reg;
    float denom_reg;
    __shared__ float tmp_numer[1];
    __shared__ float tmp_denom[1];
    for(u->N){
       #pragma unrolling UNROLLING_SIZE
        for (j->R_u) {
          numer_reg +=...//COMPUTE THE NUMERATOR OF EQ.(5) WITHOUT MATRIX R
          denom_reg +=...//COMPUTATION THE DENOMINATOR OF EQ.(5);
        }
        for (j = 16; j > 0; j /= 2){
             numer_reg += __shfl_down_sync(FULL_MASK, numer_reg, j);
             denom_reg += __shfl_down_sync(FULL_MASK, denom_reg, j);
        }
        //SUM UP ALL WARPS
        //UPDATE MATRIX U;
    }
}
```

```
void KernelOne_CPU(){
    for (int f = 0; f < factors; f++) {
        for(int j = 0; j<size_item; j++){
           v_col[j] = V.matrix[i][f];}
        for (j->R_u) {
            UPDATE_NUMERATOR(j);
            UPDATE_NUMERATOR(j+1);
            UPDATE_NUMERATOR(j+2);...}
        //DEAL WITH THE REMAINDER AND UPDATE THE TARGET MATRIX
        for (int j = 0; j<size_item; j+=UNROLLING_SIZE){
            UPDATE_R(j);
            UPDATE_R(j+1);
            UPDATE_R(j+2); ...}
        //DEAL WITH THE REMAINDER
    }
}
```

```
__global__ void KernelTwo_GPU(){
    __shared__ float numer_sh[1];
    __shared__ float denom_sh[1];
    __shared__ float tmp_numer_sh[1];
    for(u->N){//EVERY ROW OF THE TARGET MATRIX
        for (j->R_u) {
            for(f=0; f<factors/4; f++){
                tmp_u = reinterpret_cast<float4*>(uvsv_cu)[u*factors/4+f];
                tmp_v = reinterpret_cast<float4*>(uvsv_cu)[uborder/4+i*factors/4+f];
                tmp_res4 = make_float4(tmp_u.x * tmp_v.x, tmp_u.y * tmp_v.y,
                           tmp_u.z * tmp_v.z, tmp_u.w * tmp_v.w);
                res = res + res4.x + res4.y + res4.z + res4.w;
            }
            //INITIALIZE THE MATRIX R
         }
        //EVERY COLUMN OF THE MATRIX
        for (f = 0; f < factors; f++) {
            for (j->R_u) {
                numer += ... //COMPUTATION THE NUMERATOR OF EQ.(5);
                denom += ... //COMPUTATION THE DENOMINATOR OF EQ.(5);
            }
            for(j = tid; j<factors/4; j+=THREAD_NUM){
                tmp_u = reinterpret_cast<float4*>(uvsv_cu)[u*factors/4+j];
                tmp_v = reinterpret_cast<float4*>(uvsv_cu)[vborder/4+f*factors/4+j];
                tmp_res4 = make_float4(tmp_u.x * tmp_v.x, tmp_u.y * tmp_v.y,
                           tmp_u.z * tmp_v.z, tmp_u.w * tmp_v.w);
                tmp_numer = tmp_numer + res4.x + res4.y + res4.z + res4.w;
            }

              for (j = 16; j > 0; j /= 2){
                numer += __shfl_down_sync(FULL_MASK, numer, j);
                denom += __shfl_down_sync(FULL_MASK, denom, j);
                tmp_numer += __shfl_down_sync(FULL_MASK, tmp_numer, j);
            }
            //SUM UP ALL WARPS
            //UPDATE MATRIX U;
            for (j->R_u){//UPDATE MATRIX R;}
        }
    }
}
```

Figure 2.6: Implementation in Kernel 1-GPU, Kernel 1-CPU and Kernel 2-GPU: applying hardware-based accelerating techniques. Loop transformation includes **loop unrolling** for a better locality and **vectorizd I/O** to leverage data parallelism. GPU parallel reduction is accelerated by **warp primitives (warp shuffle)**, which efficiently utilizes registers and shared memory for reduction.

2.7 Hardware-based Accelerating Techniques

This section presents hardware-based accelerating techniques used in HEALS, including *loop transformation* and *GPU parallel reduction*. Fig. 2.6 illustrates the implementing details in Kernel 1-GPU, Kernel 2-GPU and Kernel 1-CPU.

2.7.1 Loop Transformation

HEALS applies loop transformation including vectorization and loop unrolling. Vectorization is commonly used to leverage data parallelism, as one instruction manages multiple data (SIMD). On CPU, the compiler automatically applies SIMD optimizations in most loops when setting `-O3`. On GPU, vectorized I/O cannot be managed by `nvcc`, providing chances to improve the kernel performance. CUDA supports vectorization instructions, including 64 or 128-bit load and store operations. Vectorization is used to update R, a data-dependent part. Thus HEALS conducts vectorization in Kernel 2-GPU. Likewise, loop unrolling is applied in Kernel 1-CPU and Kernel 1-GPU to further improve data locality.

During loop transformation, HEALS leverages shared memory to store temporal data for each block. Based on the access pattern in Fig. 2.6, when updating one row of the target matrix, the whole dense matrix has to be accessed, which cannot fit in the limited shared memory. As for the sparse matrix, each thread processes one row, of which the size varies dramatically. Thus, HEALS mainly stores temporal parameter values instead of dense or sparse matrices in the shared memory, both in Kernel 1-GPU and Kernel 2-GPU. These parameters are extended to arrays, making each thread process one element without any conflicts.

2.7.2 Accelerating GPU Parallel Reduction

GPU parallel reduction techniques aim to summarize data among threads efficiently. Based on the eALS algorithm implementation in Section 2.3, summation operation is part of the kernel computation to calculate dense matrices. Therefore, HEALS performs efficient utilization of registers and shared memory to improve GPU parallel reduction.

GPU parallel reduction in HEALS is broadcast through *warp-level (registers)*, *block-level (shared memory)*, and *global-level (global memory).* To accomplish summation operations in Kernel 1-GPU and Kernel 2-GPU, HEALS takes advantage of registers based on warp-level primitives, `shfl_down_sync`. Warp shuffle directly fetches data from other threads' registers within one warp, which benefits broadcasting data without `__syncthreads()`. The detailed implementing steps are: *First*, HEALS utilizes primitive function `shfl_down_sync` to get the sum of one warp, and the first thread of every warp obtains the result; *Second*, HEALS employs `atomicAdd` operation to sum up all warps and store the result in shared memory; *Last*, HEALS gathers the values in the shared memory and stores the final result in the global memory. Overall, efficient GPU parallel reduction yields great benefits on performance improvements, which will be explained in Section 2.8.

Table 2.3: **Machine information**

Hardware	Descriptions
CPU	Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz
GPU	NVIDIA Corporation GP100GL Tesla P100 PCIe 16GB

Table 2.4: **Experiment datasets**

Name	Users	Items	Ratings	Sparsity
Yelp	25677	25815	698472	99.8947
Amazon	117176	75389	3790245	99.9571
YahooMusic	1127446	136736	431596064	99.7211
Frienderster	37551359	124836179	1768515776	99.9999

2.8 Evaluation

This section evaluates HEALS with a set of experiments. The evaluation objectives include: (a) demonstrating that HEALS outperforms other state-of-the-art related works; (b) evaluating the effects of proposed optimizations including architecture-adaptive data format, hybrid CPU/GPU collaboration model, and hardware-based accelerating techniques; (c) validating the benefits of HEALS on prediction accuracy and recommendation quality.

2.8.1 Experiment Settings

2.8.1.1 Environment

Experiments are conducted on a heterogeneous CPU/GPU platform with an Intel Xeon CPU with 40-cores, and an NVIDIA Tesla P100 GPU (as shown in Tab. 2.3). This platform is configured with Ubuntu 18.04.4, icpc (ICC) 19.1.0.166, and CUDA V10.1.105. All HEALS runs use all computing resources (i.e., 40 threads w/o hyper-threading and the whole GPU).

Table 2.5: **Overall Performance: Execution Time (s) Per Iteration**. HEALS is compared with original java implementation of eALS, LIBMF, and CuMF with CG and LU solvers. CuMF is measured with two factors: $f = 100$ and $f = 60$. 'Speedup' illustrates the minimum and maximum speedup of HEALS over all others. 'OOM' donates that CuMF cannot execute Friendster on a single GPU due to the limited on-device memory.

Datasets	Original eALS	LIBMF	CuMF CG,f=100	CuMF CG,f=60	CuMF LU,f=100	CuMF LU,f=60	HEALS f=64	Speedup
Yelp	0.284	0.18776	0.06942	0.04104	0.2902	0.1456	**0.0120**	**3.42X-23.6X**
Amazon	1.223	1.71973	0.4186	0.2852	0.8176	0.4102	**0.1192**	**2.39X-14.4X**
YahooMusic	89.811	70.905	19.55	12.4054	22.7412	13.3947	**8.3740**	**1.48X-10.7X**
Friendster	940.8	663.7	OOM	OOM	OOM	OOM	**78.6**	**8.4X-11.97X**

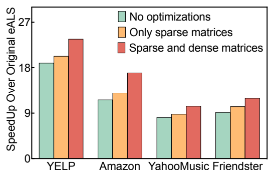


Figure 2.7: Speedup of the data formats

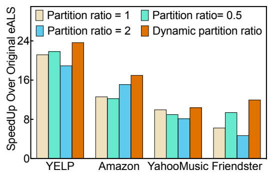


Figure 2.8: Speedup of the partition ratio

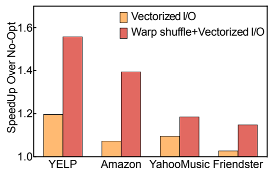


Figure 2.9: Speedup in the Kernel 1-GPU

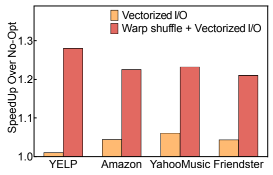


Figure 2.10: Speedup in the Kernel 2-GPU

2.8.1.2 Datasets

This work is tested on four datasets: Yelp, Amazon, YahooMusic, and Frienderster. Yelp and Amazon datasets come from the original eALS paper [30]. Dataset YahooMusic [24] includes the ratings of songs with artists and albums from the Verizon Media Labs. Frienderster is a sparse matrix dataset from the Stanford Large Network Dataset Collection. Frienderster contains 1.7 billion ratings to show HEALS's capacity of handling large data. It requires more than 56 GB memory for kernel computation, out of GPU's on-device memory. This original dataset is pre-processed to match our recommendation evaluation needs, including adding random rated values from one to five and eliminating all repeated edges and null elements. The number of users, items, and training ratings are shown in Tab. 2.4. This work randomly selects 90% data to train and 10% to test. Training datasets are used for measuring performance while testing datasets are used for measuring accuracy and recommendation quality.

2.8.1.3 State-of-the-art Works to Compare

HEALS is compared with three state-of-the-art implementations: **Original Java implementation** [30] is implemented in a sequential way. **LIBMF** [21] is a state-of-the-art matrix factorization library on CPU. This work compares with its ALS implementation (with all 40 CPU threads). **CuMF_ALS** [107, 84, 85] is a state-of-the-art library to solve matrix factorization on (multiple) GPUs. Each test runs 10 times. Because different runs do not vary significantly, this work only reports the average time for readability.

2.8.2 Overall Improvement

The overall performance improvement is evaluated in terms of the execution time per iteration (as shown in Tab. 2.5). CuMF library is evaluated with two different batch solver functions: CG and LU. Factor values (f) are commonly set to be 60 or 100 as suggested in Xie *et al.*'s work [107], so our experiments cover both factors. Tab. 2.5 shows the minimum and maximum speedup in the last column. `OOM` donates that CuMF cannot execute Friendster on a single GPU because of the limited on-device memory. Our evaluation results show that HEALS achieves significant speedups on all datasets, outperforming all other state-of-the-art works by a speedup from 1.48× to 23.6×. Particularly, comparing with the state-of-the-art CPU implementation (LIBMF with all 40 CPU threads), HEALS achieves 15.65×, 14.43×, 8.47×, and 8.44× speedup on Yelp, Amazon, YahooMusic, and Friendster, respectively. Comparing with the fastest version of the latest GPU implementation (CuMF), HEALS achieves 3.42×, 2.39×, and 1.48× speedup on Yelp, Amazon, and YahooMusic, respectively.

2.8.3 Performance Analysis: Optimization Breakdown

Fig. 2.7 to Fig. 2.10 illustrate the impact of different optimizations on performance improvements. The speedup is compared with the original eALS implementation in Fig. 2.7 and Fig. 2.8. The partition ratio in Fig. 2.8 represents the workload ratio of Kernel 2 over

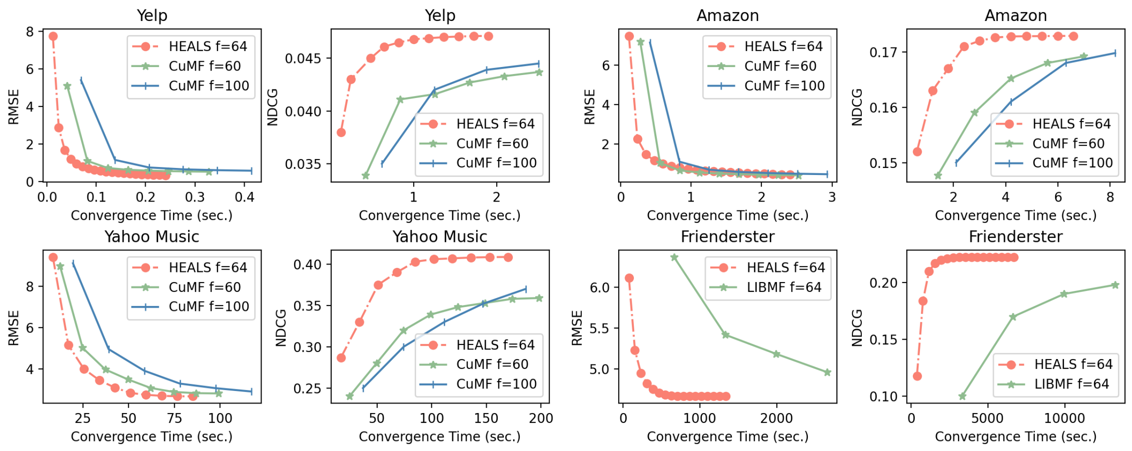


Figure 2.11: **Recommendation Efficiency: Convergence Speed of RMSE and NDCG.** Compare HEALS ($f = 64$) with CG Solver based CuMF ($f = 60$, $f = 100$) in datasets: Yelp, Amazon, and YahooMusic. Compare HEALS ($f = 64$) with LIBMF ($f = 64$) in Friendster.

Kernel 1. The baseline of Fig. 2.9 and Fig. 2.10 is the non-optimized kernel computation. The proposed hardware-based accelerating techniques are divided into `Vectorized I/O`, and `Warp shuffle + Vectorized I/O`.

Fig. 2.7 shows the impact of architecture-adaptive data format. Sparse matrix data format optimization brings 1.1× benefits on average while dense matrix optimization brings additional 1.2× gains, e.g., the speedup of Amazon improves from 13.01× with sparse matrix optimization only to 16.98× with both sparse and dense data format optimizations. Experiments demonstrate that dense matrix optimization yields slightly more benefits than the sparse matrix. Fig. 2.8 shows the benefits of dynamic partition ratio, achieving a 1.25× to 2.54× speedup over other selected fixed partition ratios. The improved results demonstrate great benefits of the hybrid CPU/GPU collaboration model. Fig. 2.9 and Fig. 2.10 show the speedup of hardware-based accelerating techniques compared with non-optimized GPU kernel implementation. Warp shuffle yields more benefits than loop transformation, e.g., the GPU vectorized I/O brings 1.09× performance gains on average while warp shuffle brings additional 1.23× benefits on average (in Fig. 2.9).

2.8.4 Recommendation Efficiency

This work evaluates recommendation efficiency in two major aspects: **matrix factorization precision** and **recommendation equality**. The performance metrics are Root Mean Square Error (**RMSE, the lower the better**) and Normalized Discounted Cumulative Gain (**NDCG, the higher the better**) [30]. RMSE represents the average accuracy of rated scores, illustrating matrix factorization precision. NDCG aims to measure the recommending quality without considering the values of rated scores. The observations involve not only *convergence point values* but *convergence speed* as well to measure the recommendation efficiency.

Fig. 2.11 shows the evaluation results. HEALS is measured with $f = 64$. HEALS is compared with CG solver-based CuMF with two factors ($f = 60$, $f = 100$) in three datasets. For Friendster, HEALS is compared with LIBMF ($f = 64$).

For RMSE, HEALS and CuMF achieve close factorization precision but HEALS obtains faster convergence speed. As shown in Fig. 2.11, HEALS achieves 1.45× convergence speedup over CuMF ($f = 60$) and 1.77× over CuMF ($f = 100$) on average, respectively. On Friendster, HEALS outperforms LIBMF, achieving 3.75× convergence speedup. Experiment results prove HEALS has better recommendation efficiency than other ALS-based parallel libraries.

For NDCG, HEALS yields benefits on both recommendation equality and convergence speed. For instance, HEALS obtains a 1.14× convergence point value over CuMF on YahooMusic. HEALS achieves a 2.78× convergence speedup over CuMF on Amazon. For Friendster, HEALS achieves a 1.12× NDCG absolute value and a 4.83× convergence speedup over LIBMF. In summary, HEALS shows great benefits on recommendation efficiency, outperforming others.

2.9 Related Work

Matrix factorization-based recommendation algorithms. Various matrix factorization-based algorithms have been applied in recommendation systems, e.g., Stochastic Gradient Descent (SGD) [13, 108], Cyclic Coordinate Descent (CCD) [70, 86], and ALS [30, 21, 84, 54]. He *et al.*'s eALS work [30] has proved that the optimized eALS-based recommendation model outperforms other matrix factorization-based approaches (e.g., SGD, CCD, and original ALS) in both computation cost and recommendation accuracy. That is why this work focuses on building *the first* efficient parallel recommendation system based on eALS.

Parallel ALS-based recommendation systems. Many existing research efforts focus on parallelizing ALS-based recommendation. LIBMF [21] offers an efficient ALS C++ library on CPUs. Chen *et al.* [20] also propose an optimized ALS algorithm based on Weighted-Regularization (WR) on CPU. Tan *et al.* [85, 84] employ iterative conjugate gradient solver to optimize parallel ALS, and present a parallel ALS implementation on GPU (CuMF). Closely related to HEALS, Teodoro *et al.* [86] propose a performance-aware CPU/GPU heterogeneous framework and optimize ALS by managing GPU layouts and partitioning. Chen *et al.* [18] implement a portable ALS framework to optimize original ALS by applying system techniques, e.g. thread batching techniques. Many other ALS-based parallel recommendation systems are implemented on distributed systems. Xie *et al.* [106] design a new loss function of ALS algorithm on Spark platform. Aljunid and Manjaiah [5] optimize conventional ALS algorithm based on Apache Spark. Winlaw *et al.* [96] propose an optimized ALS by using a nonlinear conjugate gradient (NCG) on Spark. Compared with all of these efforts, HEALS targets a more advanced algorithm (eALS) that is more challenging to be parallelized than the original ALS, and *for the first time* presents a set of new system optimizations on heterogeneous CPU/GPU systems.

Other parallel recommendation systems based on matrix factorization. Many other matrix factorization-based approaches are implemented on multi-core or many-core architectures. CuMF_SGD [107] scales up the SGD kernel computation on multiple GPUs, and Nisa *et al.* [70] present an optimized implementation of CCD++ on GPU. Li *et al.* [54] mainly focus on non-negative matrix factorization and present a multi-GPU implementation. All these efforts explore GPU optimizations without fully utilizing CPU resources or CPU/GPU collaboration. Zinkevich *et al.* [108] mainly optimize SGD in data parallelism. In contrast, HEALS exploits hybrid CPU/GPU collaboration model and combines data parallelism and task parallelism together. Again, different from all of these efforts, HEALS targets a more advanced eALS algorithm with better computation cost and recommendation accuracy while more challenges to be parallelized.

2.10 Summary

This work presents HEALS, an efficient CPU/GPU parallel recommendation system, *for the first time* building on top of fast eALS. To alleviate workload unbalance, HEALS employs a new architecture-adaptive data format for both GPU and CPU. HEALS is also equipped with a new hybrid CPU/GPU collaboration model with an adjustable partition ratio. HEALS supports multi-level concurrency including data parallelism, task parallelism, and overlapped data transferring. Moreover, HEALS takes advantage of various hardware-based accelerating techniques to further optimize kernel computation, including vectorized I/O, loop unrolling, and efficient GPU parallel reduction. HEALS outperforms other baselines by a speedup of 1.48$\times$ to 23.6$\times$. HEALS also achieves significantly better recommendation accuracy and quality, and faster convergence than other state-of-the-art libraries. In the future, we plan to extend HEALS to execute on multiple GPUs.

Chapter 3

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

3.1 Introduction

Many-body correlation functions are widely used in scientific physics systems, such as Lattice quantum chromodynamics (QCD) [17][16][15]. Correlation function calculation is critical for physics observables (e.g., predicting properties of light nuclei [28]), 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 [11]. A typical instance of many-body correlation is *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 which have hadrons as vertices and gluons as edges, followed by performing a graph contraction on every graph 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 are both computation- and memory-intensive because it involves not only a single hadron contraction but a large number of hadron contractions with specific dependencies among them. Each hadron contraction can be formalized as a *batched matrix multiplication* in meson system or a *batched tensor contraction* in baryon system that is already computational 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 [97, 101]. For example, Redstar system [17], a well-known QCD simulation, *for the first time* calculates many-body correlation functions on many-core architectures. 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 [60, 79, 56, 2, 47, 12, 63, 46, 74] 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 of other many-body problems [88, 65, 34], 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 growing number of intermediate data, limited GPU memory on a single GPU inevitably leads to memory oversubscriptions [52, 48]. 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 GPU 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, 280 MB for a baryon with 128 tensor size), current redundancy elimination techniques to operate within GPU registers [43, 71], cache [3, 23, 37], and shared memory [43, 14] 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 [72, 9, 50, 41] 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 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 avoid 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**× better than using Unified Memory management.

We organize the rest of the chapter as follows. Section 3.2 introduces the background of many-body correlation functions, the Redstar system, data characteristics, and computation patterns. Section 3.3 analyzes multiple memory redundancies and reusability opportunities. Section 3.4 illustrates an overview of the MemHC framework. Subsequently, we explain detailed techniques about memory reduction optimizations in Section 3.5. Section 3.6 demonstrates the evaluation observations and experimental results. Section 3.7 introduces related works, and Section 3.8 discusses future works. Finally, Section 3.9 concludes this chapter.

3.2 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 [28]), and is relevant to experiments planned for Jlab, FAIR, and J-PARC facilities [11]. Therefore, accelerating 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 [15] 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 GPU, explains the workflow of the Redstar system, and illustrates the kernel computation patterns of hadron contractions.

3.2.1 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:

$$C(t', t) =< \chi(t')\chi^{\dagger}(t) > \tag{3.1}$$

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

$$C(t, t') = \sum_k | < \chi|k > |^2 e^{-E_k(t'-t)} \tag{3.2}$$

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

$$C(t, 0) = Tr_{dist\ spin}[M^{12}(t)U^{23}(t, 0)M^{34}(0)D^{41}(0, t)] \tag{3.3}$$

In Equation (1), 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 Fig. 3.1 (a), and the edges between two vertices describe quark propagation. Another more complicated meson correlation function is shown in Fig. 3.1(b). The theoretical definition is:

$$C(t,0) = \sum_{p2,p3|p} c_{p2,p3} M^{12}(\vec{p},t) P^{25}(t,0) M^{56}(\vec{p_2},0) P^{63}(0,0) M^{34}(\vec{p_3},0) P^{41}(0,t) \tag{3.4}$$

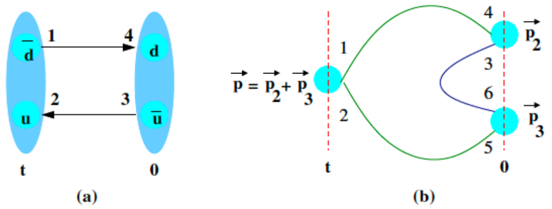


Figure 3.1: **Example of Correlation Functions.** Figure (a) describes quark propagation in a simple meson system; Figure (b) represents quark propagation in a complex meson system.

In 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 resource. 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.

3.2.2 Redstar System

The Redstar system is designed to evaluate many-body correlation functions on multi-core architectures including CPU and GPU. As shown in Fig. 3.2, the system consists of several stages including generating contraction graphs, producing multiple types of hadron

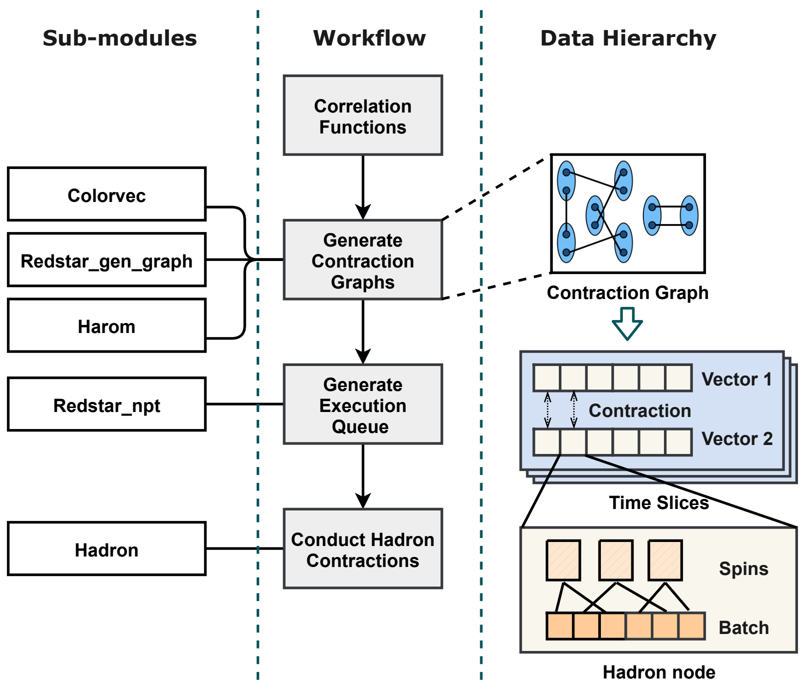


Figure 3.2: **Overview of the Redstar System: Sub-modules, Workflow and Hierarchical Data Structures.** The Redstar system involves several stages: contraction graph generation (`Redstar_gen_graph`), building hadron nodes (`colorvec` and `harom`) and hadron graph contraction solutions (`redstar_npt`, `hadron`). The `Hadron` package aims to accomplish hadron contractions. The Redstar system abstracts topologically contraction graphs from correlation functions, then reduce contraction graphs to various configurations. In one configuration, vector pairs cooperate to calculate contractions, which consists of independent hadron nodes. Hadron node includes multiple batches and associated spins.

nodes in distillation space, orchestrating 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 `harom` 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 `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.

3.2.3 Data Hierarchy

The Redstar system constructs hierarchical data structures. Fig. 3.2 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 3.2.1, 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 Fig. 3.2. 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 represent 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 examples, for two mesons (0, 0) is generated from these 4 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 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 to be 64, while the number of spins is 16. Four elements in the batch point to the same destination spin.

3.2.4 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, 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 Fig. 3.3. 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 4 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 4 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 4 temporary tensors is accumulated into a single tensor for one spin of the output.

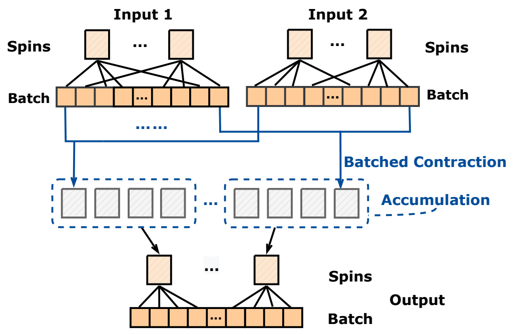


Figure 3.3: **Computation Patterns of an Individual Hadron Contraction.** Two input hadron nodes conduct a batched contraction, then accumulate to generate a batched output hadron node.

Allowing for many-body correlation functions, kernel calculation is not only computation-intensive but memory-intensive as well, 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 require 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.

3.3 Redundancy and Reusability Analysis

Based on the characteristics of computing correlation functions in Section 3.2, 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.

3.3.1 Memory Redundancy Analysis

Memory redundancies broadly exist in the following aspects: memory allocation, CPU/GPU memory communication, and memory oversubscription. *Firstly*, 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. *Secondly*, 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.

3.3.2 Data Reusability Chances

Data hierarchy illustrates various repeated data from configurations to hadron nodes, shown in Fig. 3.2. For one original contraction graph, different configurations can be

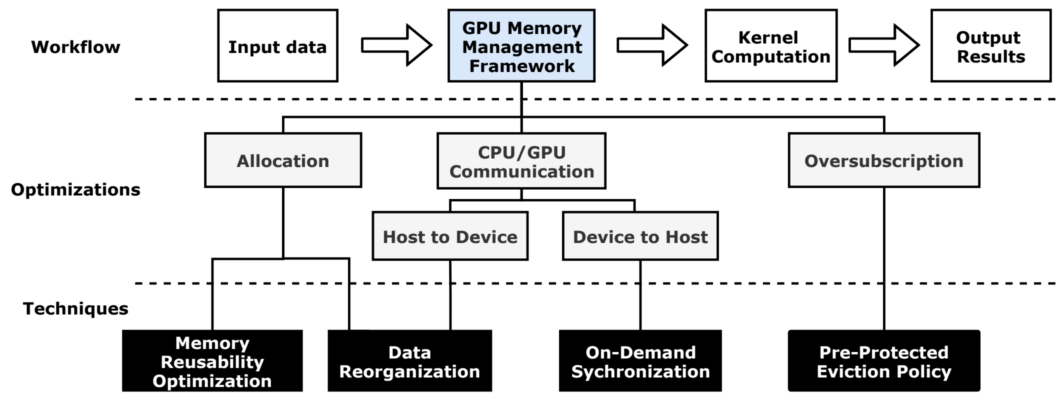


Figure 3.4: **System Overview: Optimizations and Associated Techniques.** MemHC facilitates memory reduction managements between loading input data and kernel computation. MemHC proposes reduction optimizations in three aspects: GPU memory allocation, CPU/GPU communications and GPU memory oversubscription. Techniques involve memory reusability optimizations, data reorganization, on-demand synchronization and the Pre-Protected eviction policy.

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.

3.4 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 [51], 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.

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 Fig. 3.4, 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, 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.

3.5 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: (a) memory reusability optimizations including *duplication-aware management* and *overwriting lazy-released memory*; (b) data reorganization based on contiguous memory locations; (c) performing on-demand synchro-

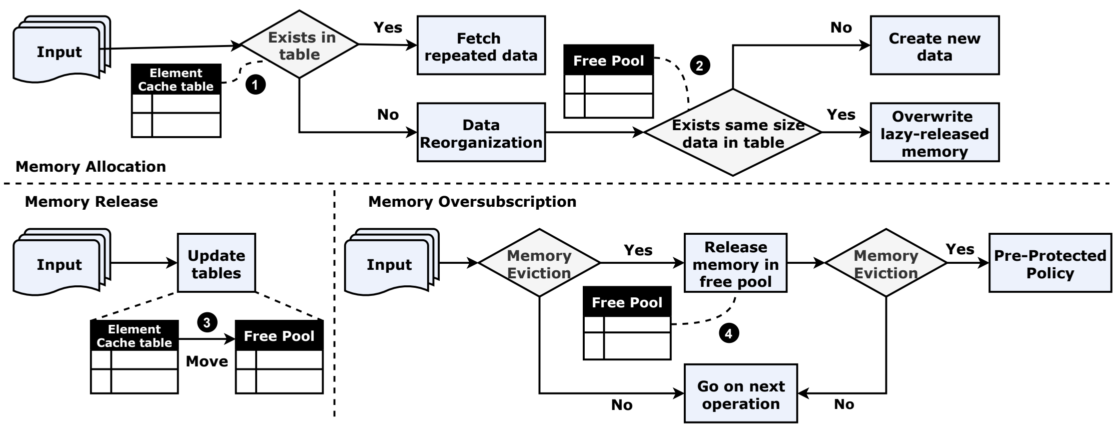


Figure 3.5: **Memory Reusability Optimizations: Duplication-aware Management and Overwriting Lazy-released Memory.** During memory allocation, MemHC checks the `element_cache` table to fetch duplicate data (❶). If the data is new, `free_pool` table will be checked to find lazy-released device memory with the same size (❷). If 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.

nization; (d) exploiting Pre-Protected eviction policy.

3.5.1 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 3.3. On the other hand, the allocating and releasing of 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 man-*

agement 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.

Fig. 3.5 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 device memory address of active data, while the `free_pool` table records device memory information about lazy-released data. When creating new memory, MemHC firstly checks the `element_cache` table to fetch the reusable data (❶). If the data is not recorded, MemHC will check `free_pool` table to find an allocated memory with the same memory size (❷). If 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 firstly 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 3.5.4.

3.5.2 Data Reorganization

Separated allocations of spins in one hadron node result in redundant memory allocations and memory movements. As claimed in Section 3.2, 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 sixteen. When solving one hadron node, the naive approach produces sixteen 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 [91, 75, 92, 39, 38], 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 sixteen, packing spins is able to reduce the number of allocating spins from sixteen to one. 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 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 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.

3.5.3 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 on the host. On other hand, calculations about intermediate data only occur onto the device. More specifically, some parameter manipulations require managing intermediate data from 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) [51, 57, 66, 76] cannot recognize this situation and produces redundant memory movements when passing references. Plenty of passing reference operations onto 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 on the host. MemHC handles the intermediate data to stay on GPU until released, without any CPU/GPU communications. More specifically, MemHC defines intermediate data 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, 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.

Algorithm 2 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 the head of contiguous memory then
5:         children ← cache_ptr → second.mem.children();              ▷ obtain children elements
6:         for $i \leftarrow 0 - children.size()$ do
7:             Free child_ptr                                          ▷ free memory of this child
8:             elem_cache_.erase(child_ptr);          ▷ remove this child from element_cache table
9:         end for
10:    end if
11:    update curr_mem_;      ▷ update memory information: the current available memory size
12:    lru_mem_.pop_back();                                           ▷ pop the back of the LRU queue
13:    elem_cache_.erase(cache_ptr);            ▷ remove the cache_ptr from element_cache table
14: end while
```

3.5.4 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, Pre-Protected LRU eviction policy to fully protect reusable data in advance based on vector forms of input data.

3.5.4.1 LRU Eviction Policy

To address memory evictions, prior efforts present many eviction algorithms, including Random Eviction, Most recently used (MRU), Least recently used (LRU), clock with adaptive replacement (CAR), Clock-Pro, and more complicated replacement methods [72, 9, 50].

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. Fig. 3.5 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 `element_cache` table to guide device memory eviction.

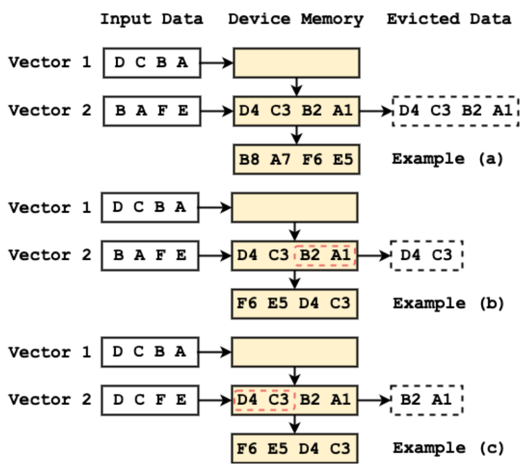


Figure 3.6: **Examples of Memory Oversubscription: Compare LRU and Pre-Protected LRU.** Input data are: (A, B, C, D) as a first vector and (E, F, A, B) or (E, F, C, D) as a second vector. Example (a) shows that LRU produces redundant evictions; Example (b) and (c) show that Pre-Protected LRU protect repeated data in advance, to avoid redundant evictions.

Detailed information about the LRU eviction algorithm is shown in Alg. 2. 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.

3.5.4.2 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. Fig. 3.6 Example (a) shows that LRU produces redundant evictions. Assume the input 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 eviciton policies, Clock-Pro [41] considers not only the recently referenced data but recently evicted data, which is better than LRU in one-time scan and large loop. CAR [9] 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 firstly 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 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.

Fig. 3.6 illustrates how Pre-Protected LRU policy works to eliminate redundant memory evictions. In this Example (b), after loading the second vector in the hadron package, A and B are found in 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 changing the repeated data or their positions, all the reusable data will be checked and protected in 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 3 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
```

Alg. 3 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 `element_cache` table, MemHC sets the flag to be true. When memory oversubscription happens, MemHC firstly 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.

3.6 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.

3.6.1 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 when memory oversubscriptions, and improved performance in 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 which appears previously to all the data. Repeated rate means the reusable data in one vector. For instance, 50% repeated rate represents half the data of each vector are repeated. 100% repeated rate represents 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 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 in a real-world system, Redstar system. Experiment results show great benefits of MemHC on real physics correlation functions, achieving up to 6.12× speedup.

3.6.2 Overall Performance Improvements (without Oversubscriptions)

Fig. 3.7 illustrates the overall improved performance of MemHC on NVIDIA. This work compares the unified memory management and optimized MemHC. Fig. 3.7 (a) and (b) show GFLOPS with varying repeated rate: 0%, 12.5%, 25%, 50%, 75% and 100%. Fig. 3.7 (c) and (d) represent GFLOPS with varying vector size: 1, 2, 4, 8, 16, 32. The evaluated tensor size is 384, as shown in Fig. 3.7 (a)(c) and 192 in Fig. 3.7 (b)(d). The evaluated results are calculated as an average result of ten execution times. Each execution time measures one hundred vectors which are randomly generated. The vector size is 16 in Fig. 3.7 (a) and (b). The repeated rate is 100% in Fig. 3.7 (d) and (e). The number of vectors is 100 in Fig. 3.7.

As shown in Fig. 3.7, MemHC outperforms unified memory management in all cases. when the tensor size is 384, the Speedup achieves from 2.18× to 3.62× in Fig. 3.7 (a) and ranges from 3.26× to 9.67× in Fig. 3.7 (b). These experiment results illustrate that the overall performance is sensitive to the repeated rate. When increasing repeated rate, the overall performance achieves more speedup, due to the increasing memory redundancy

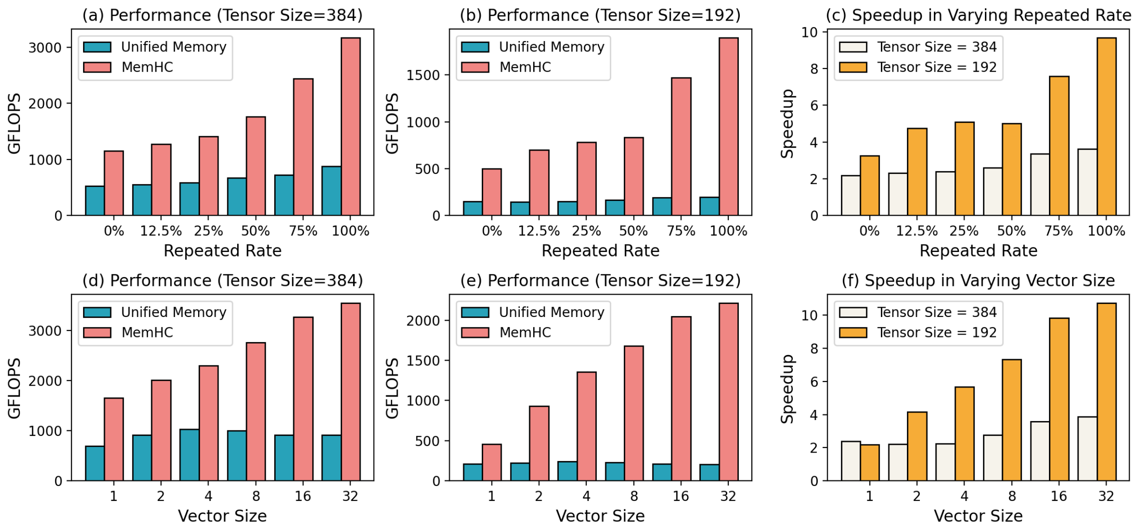


Figure 3.7: **Overall Performance: Comparing GFLOPS of Unified Memory Management and MemHC on NVIDIA.** Sub-figures (a) and (b) illustrate performance with varying repeated rate: 0%, 12.5%, 25%, 50%, 75% and 100%. Sub-figures (d) and (e) show performance with varying vector size: 1, 2, 4, 8, 16, 32. Sub-figures (c) and (f) imply the speed up of MemHC based on the unified memory. The sizes of evaluated tensors include 384 in sub-figures (a)(b) and 192 in sub-figures (d)(e).

opportunities.

MemHC yields more benefits when the tensor size is 192. The Speedup achieves from 2.39× to 3.86× in Fig. 3.7 (d) and 2.17× to 10.73× in Fig. 3.7 (e). This is because the latency of kernel computation is sensitive to tensor size. Tensor size is smaller, memory redundancies perform more impacts on the performance. More memory redundancies offer more optimization opportunities in MemHC, leading to more speedup shown in Fig. 3.7 (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 100% repeated rate, shown in Fig. 3.7 (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 repeated rate demonstrates high memory redundancy efficiency of MemHC.

When changing vector size, unified memory management does no show any improvements, while MemHC achieves obvious Speedup. Fig. 3.7 (d) shows that MemHC achieves GFLOPS from 1653.59G/s to 3541.42G/s and Fig. 3.7 (e) illustrates that GFLOPS improves from 454.53G/s to 2213.88G/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.

3.6.3 Performance Analysis in General Correlation Functions

3.6.3.1 Breakdown Analysis

To further explore the efficiency of MemHC, this section focuses on breakdown analysis. Fig. 3.8 evaluates the performance of the memory redundancy techniques separately and compares with the Unified memory management about both GFLOPS and Speedup. Unified memory management is evaluated by two cases: non-optimized implementation (`Unified Memory Naive`) and 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 repeated rate varies from 0%, to 100% in (b). The performance results are measured on NVIDIA P100.

Fig. 3.8 (a) and (c) show the impact of the repeated rate. When the repeated data is 0%, the Speedup of the `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

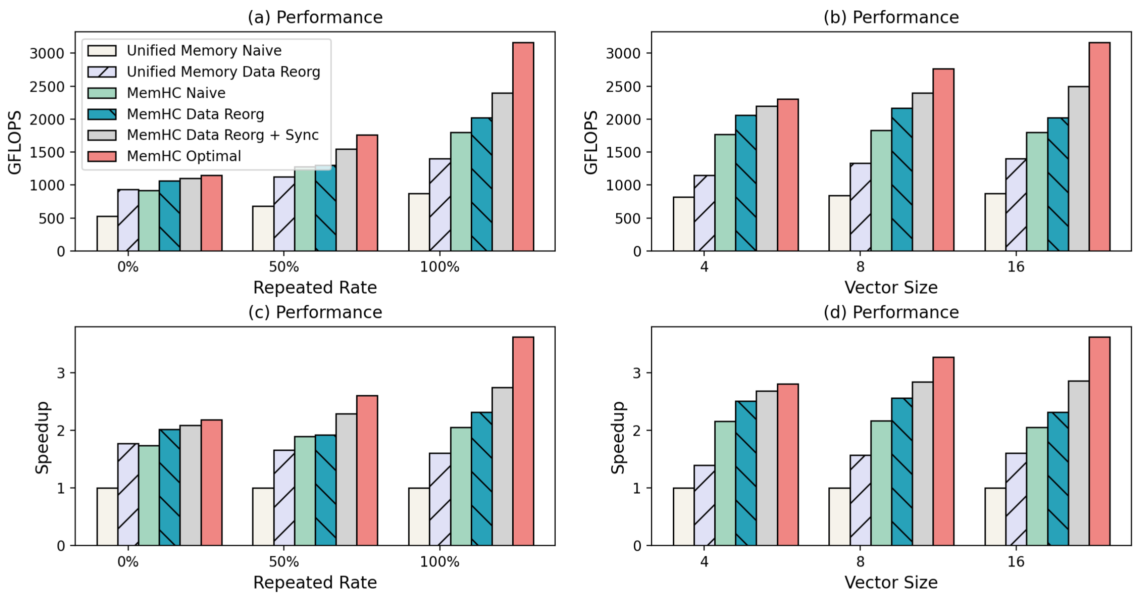


Figure 3.8: **Optimization Breakdown: Compare Unified Memory Management and MemHC about GFLOPS and Speedup.** Unified memory management is evaluated by two cases: Unified Memory Naive and Unified Memory Data Reorg. MemHC is evaluated by four cases: naive (MemHC Naive), only optimized by data reorganization (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.

achieves from 2.5× to 2.8× in (a) and 2.1× to 2.4× in (c). Fig. 3.8 (b) and (d) illustrate the influence of the vector size. Take 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× speed up 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 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

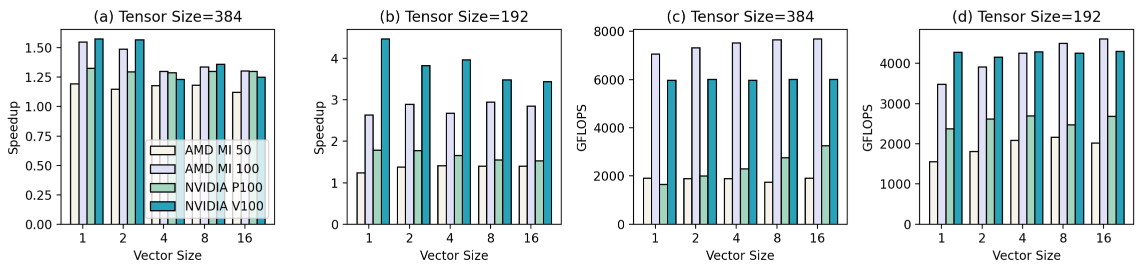


Figure 3.9: **Exploring Portability: GFLOPS and Speedup on AMD MI50, MI100 and NVIDIA P100, V100.** Tensor sizes are 384 and 192. Vector sizes are 1, 2, 4, 8, 16. Figure (a) and (b) show speedup based on non-opmized explicit implementation. Figure (c) and (d) illustrate GFLOPS.

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 overlapped 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 internal structure, has an impact on reorganization overhead, further influencing the data reorganization improvements.

3.6.3.2 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.

Fig. 3.9 (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 size 192 and 384, NVIDIA V100 can achieve average 3.8× and 1.3× speedup, respectively.

Fig. 3.9 (c) and (d) illustrate the MemHC's GFLOPS on 4 different GPUs, including

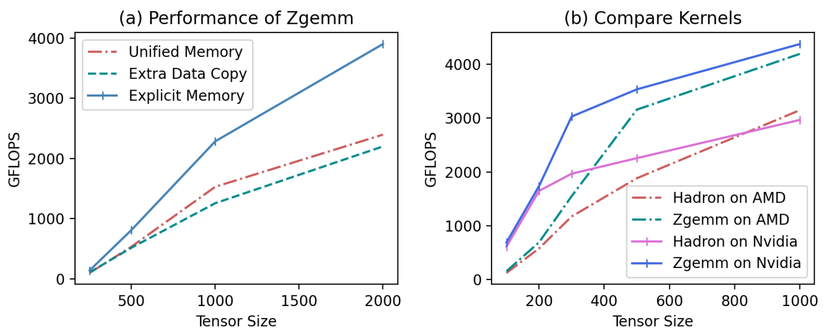


Figure 3.10: **Exploring Kernel Computation on AMD MI50 and NVIDIA P100.** Sub-figure (a) explores the limitation of Unified Memory. Extra Data Copy donates adding redundant data transfer in explicit memory implementation. Sub-figure (b) compares the performance of `zgemm` in cuBLAS and hadron contraction kernels on MI50 and P100.

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.

3.6.3.3 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 abut CPU/GPU communications in unified memory management.

Current NVIDIA GPUs provide Unified Memory [51, 57], which supports a virtual single address both accessible for 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 [66, 76]. When evaluating unified memory management, this work sets `cudaMemAdviseSetReadMostly` flag and performs `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 on `zgemm` kernel of cuBLAS. In Fig. 3.10 (a), there are three lines illustrate the performance of `zgemm`. The blue line simulates the explicit implementation of `zgemm`. The red line represents the unified memory managed 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 Fig. 3.10 (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.

3.6.3.4 Exploring Hadron Contraction Kernel

To explore the efficiency of 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. Fig. 3.10 (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 3.2, this work implements batched tensor contraction kernel based on the `zgemm`. When the tensor size is smaller than 200, the performance of `zgemm` kernel and hadron contraction achieve close performance. Besides `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 per-

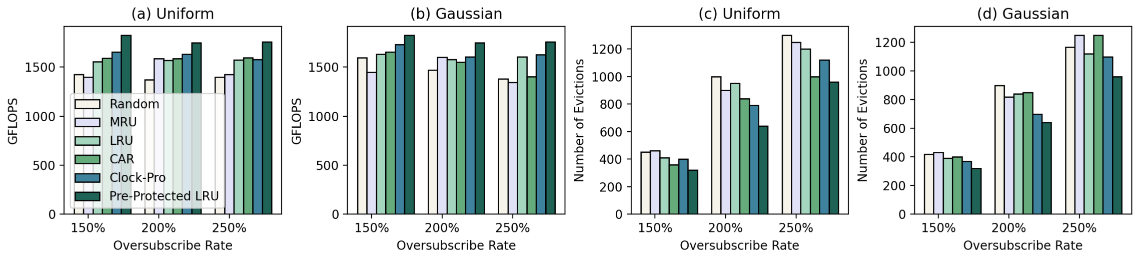


Figure 3.11: **Comparing multiple eviction policies.** Pre-protected eviction policy is compared with Random policy, MRU, LRU, CAR and Clock-Pro policies. Measure metrics include GFLOPS and the number of evictions. Oversubscribe rate changes from 50% to 150%. The tensor size is 384. Vector size is 64. Repeated rate is 50%.

formance. 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.

3.6.4 Memory Oversubscriptions in General Correlation Functions

This section evaluates 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. Fig. 3.11 illustrates the comparison between Pre-Protected LRU and other eviction policies, including Random, MRU, LRU, CAR, and Clock-Pro. The experiments measure GFLOPS in Fig. 3.11 (a) and (b), and the sum of ten iterations of the number of evictions in Fig. 3.11 (c) and (d). The data distribution includes Uniform distribution and Gaussian distribution, when the oversubscribe rate ranges from 150% to 250%. Fig. 3.11 (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 mean GFLOPS over other eviction policies in both datasets using various oversubscribe rates. Fig. 3.11 (c) and (d) show the number of evictions of these eviction policies. The evaluation results illustrate

Table 3.1: **Performance of Memory Eviction with Varying Vector Size: Hit Rate, GFLOPS on NVIDIA P100.** The results are calculated as the average values per vector of ten 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.

Experiment Results Per Iteration (Oversubscription Rate=50%, Repeated Rate=1/2)								
Vector Size	**Eviction Policy**	**Allocate**	**Evicts**	**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%**	**1062.84**	**1.36×**
16	LRU_UNIFORM	48	12	35	61	36.45%	1453.61	1×
	LRU_GAUSSIAN	47	12	36	60	37.50%	1501.1	1.03×
	Pre-Protected LRU	**44**	**8**	**39**	**57**	**40.63%**	**1907.89**	**1.31×**
32	LRU_UNIFORM	89	22	74	118	38.54%	1571.02	1×
	LRU_GAUSSIAN	87	20	76	116	39.58%	1605.04	1.02×
	Pre-Protected LRU	**84**	**16**	**79**	**113**	**41.15%**	**1943.48**	**1.23×**

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 in 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 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 Tab. 3.1 3.2 3.3, which are calculated as the average value of ten execution loops.

The Pre-Protected LRU achieves improvements from 1.12× in Tab. 3.2 to 1.36× in Tab. 3.1 GFLOPS of the LRU eviction policy. The average improvement is 1.21×. Hit rate improvements achieve from ten percent to thirty percent better than original LRU policy. As the vector size increases from 8 to 64, the improved performance decreases from

Table 3.2: **Performance of Memory Eviction with Varying Oversubscription Rate: Hit Rate, GFLOPS on NVIDIA P100.** The results are calculated as the average values per vector of ten 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.

Experiment Results Per Iteration (Vector size=64, Repeated Rate=1/2)								
Oversub-rate	**Eviction Policy**	**Allocate**	**Evicts**	**Hit**	**Miss**	**Hit Rate**	**GFLOPS**	**Speedup**
50%	LRU_UNIFORM	178	46	145	239	37.76%	1554.54	1×
	LRU_GAUSSIAN	171	40	152	232	39.58%	1630.69	1.05×
	Pre-Protected LRU	**164**	**32**	**159**	**225**	**41.4%**	**1825.28**	**1.17×**
100%	LRU_UNIFORM	216	120	235	341	40.79%	1568.06	1×
	LRU_GAUSSIAN	213	82	238	338	41.31%	1578.86	1.01×
	Pre-Protected LRU	**196**	**64**	**255**	**321**	**44.2%**	**1750.12**	**1.12×**
150%	LRU_UNIFORM	251	120	328	440	42.71%	1575.25	1×
	LRU_GAUSSIAN	243	112	336	432	43.75%	1606.90	1.01×
	Pre-Protected LRU	**228**	**96**	**351**	**417**	**45.7%**	**1759.57**	**1.12×**

Table 3.3: **Performance of Memory Eviction with Varying Repeated Rate: Hit Rate, GFLOPS on NVIDIA P100.** The results are calculated as the average values per vector of ten 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.

Experiment Results Per Iteration (Oversubscription Rate=50%, Vector Size = 64)								
Repeated Rate	**Eviction Policy**	**Allocate**	**Evicts**	**Hit**	**Miss**	**Hit Rate**	**GFLOPS**	**Speedup**
1/8	LRU_UNIFORM	196	64	127	257	33.07%	1346.61	1×
	LRU_GAUSSIAN	194	62	129	255	33.59%	1362.25	1.01×
	Pre-Protected LRU	**188**	**56**	**135**	**249**	**35.15%**	**1595.54**	**1.18×**
1/4	LRU_UNIFORM	192	60	131	253	34.11%	1400.95	1X
	LRU_GAUSSIAN	190	58	133	251	37.23%	1427.83	1.02×
	Pre-Protected LRU	**180**	**48**	**143**	**241**	**34.64%**	**1720.16**	**1.23×**
3/4	LRU_UNIFORM	151	20	172	212	44.79%	1940.44	1×
	LRU_GAUSSIAN	149	18	174	210	45.31%	1982.20	1.02×
	Pre-Protected LRU	**148**	**16**	**175**	**209**	**45.57%**	**2257.93**	**1.16×**

1.36× in Tab. 3.1 to 1.17× in Tab. 3.2. The experiment results show that the original LRU policy is sensitive to the vector size. In Tab. 3.1, 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 makes less probability to be evicted. In other cases, memory oversubscription performance is not influenced by data distribution. Additionally, Tab. 3.2 illustrates stable performances of three eviction policies with varying oversubscription rate. 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. Tab. 3.3 explores the influence of the repeated rate. On one hand, the performance improvement is relatively stable, achieving about 1.2×. On 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 3.5.4. The Pre-Protected LRU eviction policy always outperforms 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.

3.6.5 User Case: Evaluation in Redstar System

In order to evaluate in practical scenarios, this work measures three real physics correlation functions in Redstar system. As claimed in Section 3.2, calculating one correlation function includes multiple configurations with multiple time intervals. In a typical practical scenario, one correlation function produces about four to five hundred configurations, then one configuration executes through sixty-four 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 measure by the average executing results of one configuration of the correlation functions with a single time interval. All these three correlation functions belong to multi-meson system computations. The detailed information of correlation functions is shown in Tab. 3.4.

This work designs a set of experiments to integrate into the Redstar system. Execution time and GFLOPS are measured to evaluate performance Improvements. Tab. 3.5 illustrates the experiment results. Execution time implies the average of ten execution loops at a single time interval of one configuration. GFLOPS are calculated based on

Table 3.4: **Information of Real Correlation Functions** 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

Function Name	Tensor Size	#Total Nodes	Memory (GBytes)	#Contractions
`a1_rhopi`	128	106	0.44	68
`f0d2`	256	2173	36.28	1968
`f0d4`	256	2173	36.32	1970

Table 3.5: **Performance of Real Correlation Functions: Execution Time and GFLOPS on NVIDIA P100.** Speedup means the times of the accelerated performance of MemHC.

Function	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**×

all the memory operations and the execution time. According to the Tab. 3.5, 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.

3.7 Related Work

Tensor contraction optimization works. Prior works focus on implementing a general method [60, 79, 56, 2, 47, 12, 63, 46, 74] 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[60], symmetry[46], and high rank[79]. Kim *et al.*'s work [47] proposes a GPU code generator of tensor contractions to leverage data reuse in a high dimensional loop. Another work of Kim *et al.* [46] aims to optimize CCSD kernel computation for specific applications. Ma *et al.*'s work [62] implements a code generator to translate tensor expressions to optimized CUDA codes. Nelson *et al.*'s work [69] presents 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. Graphics processing units (GPUs) is famous for dramatically speeding up the computation of various practical applications, including deep learning [100, 104, 102, 93] and scientific computing. Many existing research efforts aim to optimize GPU memory management. Li *et al.*'s work [55] presents a set of hybrid implicit or explicit data movement frameworks to optimize GPU unified memory. Dashti *et al.*'s work [22] discusses various popular memory management methods in heterogeneous systems including HSA, NVIDIA, and AMD hardware. Ausavarungnirun *et al.*'s work [8] proposes 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 [52] classifies applications as regular and irregular and provides three memory oversubscription mitigation techniques. Kim *et al.*'s work [45] also focuses 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 [43, 71] eliminate redundant operations in registers. Other elimination redundancy works focus on GPU cache [3, 23, 37] and shared memory [43, 14]. Unlike these efforts, one hadron node requires about 37M memory cost in a single meson system with 384 tensor size, which can not be optimized in GPU register, cache or shared memory. Unified Memory management [51] jointly manages host and device memory, and provides memory allocation elimination techniques. However, as analyzed in Sec. 3.6, 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 evict reusable data and leverage memory hit rate. Popular eviction policies include Random Eviction, Most recently used (MRU), Least recently used (LRU), clock with adaptive replacement (CAR), Clock-Pro, and more complicated policies [72, 9, 50, 41]. Clock-Pro [41] considers not only the recently referenced data but recently evicted data, which is better than LRU in one-time scan and large loop. CAR [9] 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.

3.8 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 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.

3.9 Summary

In the paper, 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, including 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 complicated hadronic systems.

Chapter 4

MICCO: An Enhanced Multi-GPU Scheduling Framework for Many-Body Correlation Functions

4.1 Introduction

Calculation of many-body correlation functions is a key kernel widely used in many scientific physics systems (such as Lattice Quantum Chromodynamics(QCD)) [11, 15, 16, 17, 83, 6]. *Hadronic correlation function* in complex multi-meson and multi-baryon systems is a typical example of many-body correlation function, which involves quarks enclosed in mesons and baryons. Calculating hadronic correlation functions converts a series of quark propagations describing interactions among hadrons into many undirected graphs that have quarks of the hadrons as vertices and quark propagations as edges, followed by performing a graph contraction on every graph that reduces graph edges one after another until only two hadrons are left. Each reduction of an edge is a *tensor contraction* between hadron nodes which is dubbed hadron contraction.

Calculation of many-body correlation functions is computation and memory-intensive because it usually involves many thousands even millions of contractions resulting in extremely large numbers of tensor contractions. Graph contractions also generate a large amount of intermediate data, requiring significant memory resources. Thus, real-world physics systems commonly rely on high-end computing devices like many-core GPUs to compute many-body correlation functions. Specifically, due to the limited memory size of a single GPU, multi-GPU systems are preferred.

However, accelerating the calculation of many-body correlation functions on multi-GPUs is challenging. In contrast to general graph-based applications that process a huge graph on multi-GPUs [10, 19, 40], many-body correlation function calculations are featured with two specific characteristics: First, the entire calculation consists of many computation-/data-intensive kernels that are represented by graph edges. Second, repeated hadron nodes appear frequently because of overlapped reduction paths among multiple contraction graphs. The former shifts the scheduling bottleneck from optimizing graph partition and reducing partition synchronization (as shown frequently in general graph processing) to improving the GPU assignment of these computation kernels to avoid expensive memory operations such as tensor evictions in memory oversubscription situations, or tensor movements. The latter offers unique (and many) data reuse opportunities that potentially mitigate the data-intensive nature of many-body correlation function calculations. Unfortunately, existing multi-GPU scheduling frameworks [94, 10, 4, 36, 44, 80] mainly focus on workload balance without considering the above *data dimension* that is critical to the execution performance of many-body correlation, thus resulting in sub-optimal system performance if they are adopted directly.

To address this issue, this work presents a new multi-GPU scheduling framework, MICCO, to accelerate calculating many-body correlation functions. The key innovation of MICCO is that it brings the *data dimension* into the whole scheduling picture, particularly by studying the impact of a *data reuse-load balance* interplay on the scheduling and leveraging this interplay to find the optimal scheduling scheme. The key insight of this study is that data reuse and load balance form a trade-off relationship in scheduling scheme exploring and multiple factors affect this trade-off, rendering it very challenging to find a global optimal scheduling solution within a practical time budget for real-world systems.

Fortunately, this study demonstrates that it is possible to create a highly effective local optimal scheduling with the help of two newly designed concepts including *local reuse pattern* and associated simplified but effective mapping analysis, and *reuse bound* that characterizes the allowed level of load imbalance when exploring data reuse opportunities. Based on both new concepts, this work proposes a heuristic scheduling algorithm that toggles between leveraging data reuse and pursuing load balance particularly by taking memory evictions into account, and designs a machine-learning-based regression model to determine the optimal setting of reuse bounds. MICCO is integrated into a well-known Lattice QCD system, Redstar [15, 16, 17], *for the first time* running it on multiple GPUs.

The main contributions can be summarized as follows:

- For the first time performing a comprehensive study on the interplay between data reuse and load balance in multi-GPU scheduling of many-body correlation function calculations, particularly introducing two new concepts that are critical in multi-GPU scheduler design, *local reuse pattern* and *reuse bound.*
- Based on the previous study, presenting a multi-GPU scheduling framework, MICCO, to accelerate the calculation of many-body correlation functions that consists of a heuristic scheduling algorithm and a regression model to generate optimal settings to balance the impact of data reuse and load balance, particularly by considering memory oversubscription situations.
- Integrating MICCO into a real-world Lattice QCD system, Redstar, and for the first time running it on a multi-GPU environment.

MICCO is extensively evaluated with both synthesized datasets and real-world datasets with varied settings. The evaluation demonstrates that MICCO outperforms other state-of-art works in all situations, achieving up to 2.25× speedup.

4.2 Background

4.2.1 Many-body Correlation Function

Hadronic correlation functions are the central quantities to be calculated when determining the properties and interactions of quarks directly from Lattice QCD simulations. Calculation of correlation functions is crucial for generating physics observables and is relevant to experiments planned for Jlab, FAIR, and J-PARC facilities [11, 83, 6]. However, the computational cost of constructing such correlators is, however, known to be exceptionally enormous. The reason for such a high cost comes from computing all required quark propagation diagrams [15] resulting from *Wick contractions* [15, 16, 17]. The number of such diagrams grows factorially as the number of quarks and the total number of freedom of the hadronic systems under consideration increase. A quark propagation diagram can be represented as a graph consisting of a set of hadron nodes each of which has vertices (V) representing the quarks inside a hadron node and undirected edges (E) describing quark propagations at specific time intervals. Especially, the number of unique graphs can be potentially huge approaching in the order of $500,000$. The graph contraction of a graph, which is defined as deleting one edge after another, consists of a series of hadron contractions involving batched matrix multiplications for a meson system or batched tensor contractions for a baryon system[1]. A large number of contraction graphs on many time-slices and the size of matrices/tensors ($\approx 100s$) associated with the hadron nodes present extreme computing challenges. It is paramount to utilize modern computing accelerators such as GPUs to speed up the calculations of hadron contractions.

[1]This paper uses *tensor* in the following discussion to refer to both two-dimensional *matrix* and higher dimensional *tensor*.

4.2.2 Topological Representations

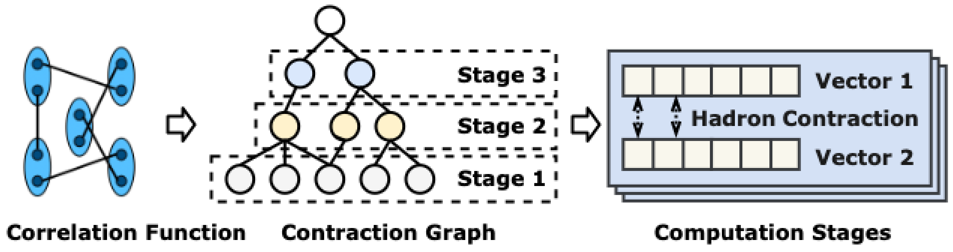


Figure 4.1: **Topology Representations of many-body Correlation.** Correlation functions are represented as multiple contraction graphs. Each contraction graph consists of multiple computation stages. Each stage consists of two vectors of independent hadron nodes. Each pair of hadron nodes conducts hadron contractions.

To translate the statistic definition into a formalized computational problem, Fig.4.1 illustrates many-body correlation function calculations as a topological representation, in the form of contraction graphs. It is worth noting that a many-body correlation may involve **thousands of contraction graphs** while this figure only shows one for simplicity. In each contraction graph, vertices represent hadron nodes, while edges describe the interactions between hadron nodes. Hadron nodes are formalized as batched matrices or tensors, with different ranks of tensors representing different types of hadron nodes (e.g., matrices in meson systems and three-dimensional tensors in baryon systems, respectively). The associated interactions between two hadron nodes are formalized as matrix multiplications or tensor contractions.

A well-known Lattice QCD system, Redstar [15, 16, 17] first translates each correlation function into a set of unique contraction graphs, and then produces a sequence of hadron contractions from the generated contraction graphs. One correlation function can produce many thousands of contraction graphs. Each graph undergoes a graph contraction process during which one edge after another is reduced until only two nodes are left. Again, each reduction of an edge corresponds to a matrix multiplication or tensor contraction.

To leverage the concurrency of many-body correlation calculations, pre-processing, based on dependency analysis is used to partition the computation into several stages (e.g., stages 1, 2, and 3 in Fig.4.1) with these stages executing sequentially. Each stage contains two vectors and each vector contains independent hadron nodes. Each pair of associated hadron nodes in these two vectors accomplishes hadron contractions. Since the hadron nodes are independent, the hadron contractions can execute concurrently.

4.2.3 Challenges and Opportunities

Many-body correlation calculation introduces multiple new (and interesting) challenges to multi-GPU scheduling due to its unique computation patterns:

Calculation consists of many computation-intensive kernels. In contrast to conventional graph processing applications (e.g., BFS, PageRank, and Shortest Path) [94, 81], many-body correlation comprises a large number of small contraction graphs that construct a backbone computation structure, and the overall correlation function consists of many computation-intensive kernels, e.g., matrix multiplications or tensor contractions that are represented by edges of these contraction graphs. The multi-GPU scheduling bottleneck is shifted from graph partition and partition synchronization reduction to proper GPU assignment of these computation-intensive kernels to avoid frequent memory oversubscription and intensive tensor movement.

Contraction graphs may share hadron nodes. A hadron node may appear multiple times in more than one contraction graph in a random manner. The tensors belonging to this hadron participate in multiple computations if this hadron is shared by multiple contraction graphs. This key observation demonstrates that many-body correlation offers many *data reuse* opportunities during its computation. It is critical to take data reuse into account during multi-GPU task allocation. This work particularly studies the interplay between data reuse and load balance and proposes an enhanced scheduler based on this study.

System cannot afford a heavy scheduler. Finding the optimal scheduling scheme for the entire many-body correlation computation is time-consuming because it consists of thousands of kernel computations that involve many matrices and tensors. The resulted

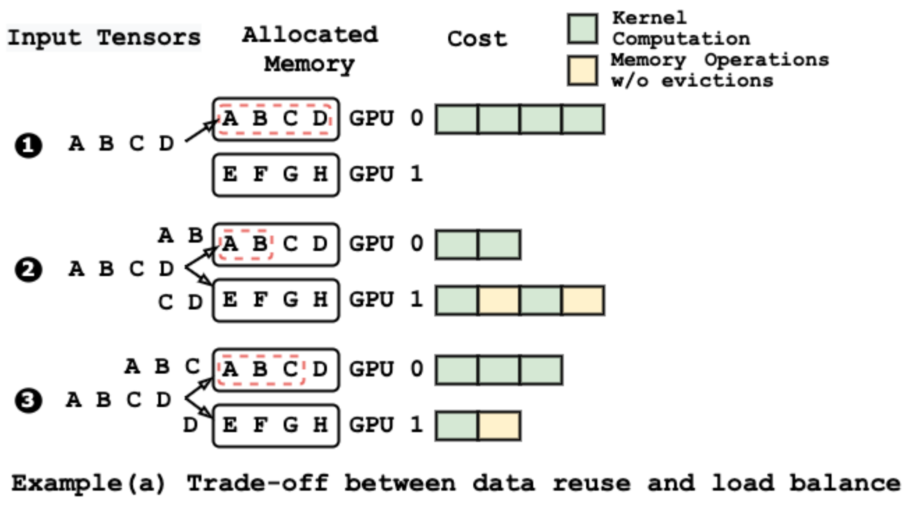


Figure 4.2: **Example (a): Trade-off between data reuse and load balance.** Input tensors are A, B, C, and D. Case ❶ only considers data reuse; Case ❷ only cares about load balance; Case ❸ trades off data reuse and load balance. Red dotted frames label reused data. The green bars mean kernel computation cost, and the yellow bars mean memory operation cost (allocation and communication) without memory evictions.

searching space is huge. However, due to the computation- and memory-intensive nature, the real-world systems cannot afford a heavy scheduling mechanism. A lightweight approach with a reduced scheduling search space and the limited cost is desired.

4.3 Interplay between Data Reuse and Load Balance

This section carefully studies the interplay between two scheduling metrics, data reuse and load balance, and their effects on multi-GPU scheduling of many-body correlation calculations. This section further analyzes the impact of multiple key factors on this interplay. This study aims to guide the design of MICCO.

4.3.1 Data Reuse and Load Balance Trade-off Analysis

Although improving load balance and data reuse can both lead to better multi-GPU system performance, the multi-GPU scheduler may not be able to achieve optimal for both, simultaneously, e.g., optimizing data reuse may result in unbalanced computation.

An interesting trade-off relationship exists between these two metrics. Fig.4.2 illustrates a detailed example. Assume input data are four tensors (A, B, C, and D) in a vector. If at present time GPU 0 has fetched a copy of these tensors from CPU and GPU 1 stores another set of tensors (E, F, G, and H). In the next step, if only considering data reuse, all input tensors should be assigned to GPU 0 (as shown in case ❶); while if only caring about workload balance, GPU 0 and GPU 1 should fetch the identical amount of tensors (as shown in case ❷). However, both cases result in sub-optimal system performance. Case ❶ only keeps GPU 0 busy, while case ❷ incurs extra memory operations for two tensors (C and D), including two tensor allocations and two tensor movements from CPU to GPU. In contrast to both cases, we point out case ❸ specifically that trades off data reuse and load balance, i.e., assigning three tensors (A, B, and C) to GPU 0 and one tensor (D) to GPU 1. This case results in the best system performance among three schedule schemes.

Fig.4.3 shows that concerning memory oversubscription, both data reuse (Example (b)) and load balance (Example (c)) are able to reduce memory evictions. Leveraging data reuse decreases the total new memory allocations to avoid memory oversubscription. In Example (b), assume each GPU memory can hold up to four input tensors. Both scheduling cases have balanced workloads, but case ❷ does not reuse the repeated tensors and causes two extra memory operations (including two memory allocations and two tensor movements), and two memory evictions for each GPU. In Example (c), assume each GPU memory can hold two more output tensors. Example (c) compares two cases to show load balance can also help oversubscription: case ❶ has better reusability with three reused tensors, and case ❷ has only two reused tensors but better workload balance. In case ❶, a memory eviction occurs when tensor C results a new output tensor. Case ❷ achieves no evictions and better performance than case ❶.

Remarks: A proper trade-off between data reuse and workload balance results in the optimal task allocation and helps avoiding memory evictions in GPU oversubscription situations that frequently happen in large-scale scientific computations with memory-

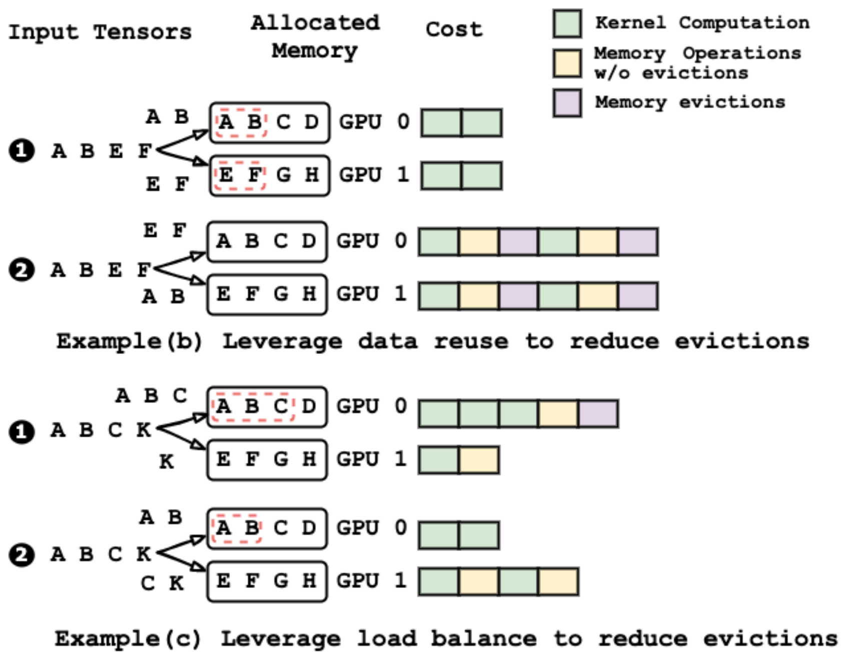

Input Tensors
Allocated Memory
Cost
Kernel Computation
Memory Operations w/o evictions
Memory evictions
GPU 0
GPU 1
Example(b) Leverage data reuse to reduce evictions
Example(c) Leverage load balance to reduce evictions

Figure 4.3: **Examples: Trade-off between data reuse and load balance regarding memory evictions.**

intensive kernels like many-body correlation.

4.3.2 Factors Impacting the Data Reuse-Load Balance Trade-off

The execution of many-body correlation function calculation consists of three main parts: kernel computation, memory allocation, and data communication (i.e., data movement between CPU and GPU or between two GPUs). The latter two are referred to as memory operations in this paper. Data reuse mainly reduces memory operation cost, while workload balance is critical to kernel computation performance. Our study discovers that multiple factors influence this data reuse-load balance trade-off, and our multi-GPU scheduler design can benefit from a careful study of them.

4.3.2.1 The Impact of Local Reuse Pattern on the Trade-off

Theoretically, if the scheduler can capture all data reuses and conduct an exhaustive search by targeting the best data reuse-load balance combination, it can find the optimal

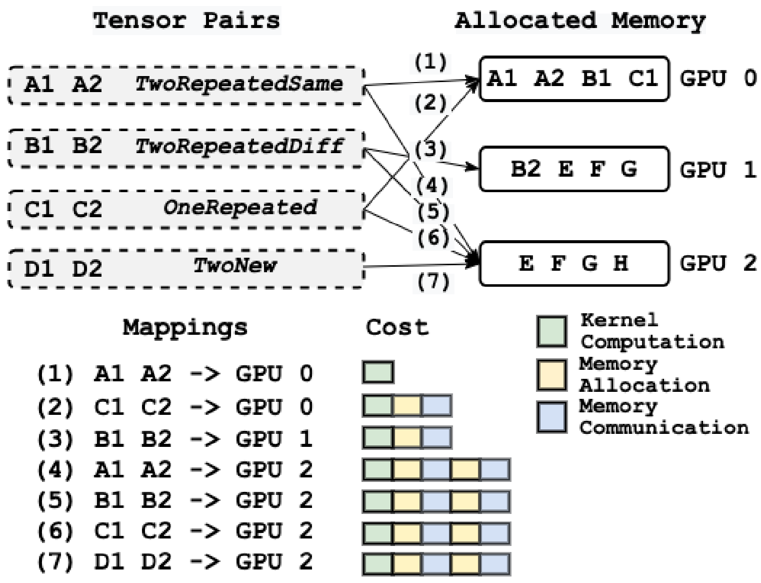


Figure 4.4: **Example: Local reuse patterns and task assignments.** Classify tensor pairs based on four local reuse patterns: *TwoRepeatedSame*, *TwoRepeatedDiff*, *OneRepeated*, and *TwoNew*. Mappings between tensor pairs and GPUs can be categorized into seven cases. Mapping (1) represents two reused tensors, assigned to the re-utilized GPU with the least overhead. Mappings (2) and (3) contain one reused tensor, and the rest four mappings have two new tensors, resulting in the most expensive cost.

task scheduling scheme. However, two major issues exist: First, it assumes the global knowledge of all contraction graphs that may not be available for many cases, particularly when (partial) contraction graphs are generated dynamically. Second, the search space is too large and this exhaustive search is easy to be proved an NP problem as other task scheduling problems. To address this issue, this work proposes to leverage *local reuse pattern* information to dynamically search the local optimal scheduling scheme based on a key study as follows.

Each tensor contraction involves two tensors. The tensor pair of each (incoming) tensor contraction can be categorized into one of four local reuse patterns (Fig.4.4 shows an example):

- ***twoRepeatedSame***: Both tensors in this pair already exist in the current memory of the same GPUs. $A1$ and $A2$ already exist in the memory of GPU 0 when the new tensor pair (with $A1$ and $A2$) comes.

- ***twoRepeatedDiff***: Two tensors exist in the current memory of different GPUs. $B1$ and $B2$ already exist on GPU 0 and 1, respectively.
- ***oneRepeated***: One tensor of this tensor pair exists in current GPU memory. $C1$ presents in GPU 0 and $C2$ is a new tensor.
- ***twoNew***: Neither of the two tensors of the tensor pair exist in current GPUs' memory. $D1$ and $D2$ are two new tensors.

Based on this *local reuse pattern* classification, Fig.4.4 also demonstrates and analyzes the cost of seven typical task assignments/mappings[2]: Mapping (1) assigns both tensors to the GPU that stores $A1$ and $A2$, previously. Mappings (2) and (3) assign only one reused tensor in the tensor pair to the GPU with this tensor before, producing one memory allocation and one memory communication. Mappings (4) - (7) incur the most expensive cost: two memory allocations and two memory communications.

Remarks: Although it is challenging to find the global optimal scheduling scheme, it is possible to create a local optimal one with our insights on the *local reuse patterns* and *mapping study* aforementioned, particularly by designing a heuristic approach (introduced in Section 4.4). This approach has demonstrated its high efficacy in our evaluation.

4.3.2.2 The Impact of Reuse Bounds on the Trade-off

Another key factor that impacts the data reuse-load balance trade-off is *the level of allowed load imbalance*, i.e., the scheduler allows a certain level of load imbalance to leverage the potential data reuse. This work defines this factor as a special term called **reuse bound**. For example, assume assigning eight tensors to two GPUs. If the reuse bound is zero, each GPU must receive four tensors (i.e., with a perfect load balance). If the reuse bound is two, each GPU can receive up to six tensors, i.e., each GPU allows to exceed the average allocation by two.

[2]The costs of other mappings that are not shown in this figure have been covered by these cases.

Table 4.1: **Description of Reuse Bounds.** Reuse bounds manage different tensor pairs and mappings, representing the allowed level of load imbalance.

Name	Tensor Pairs	Mappings
`Reuse_bound_1`	*TwoRepeatedSame*	(1)
`Reuse_bound_2`	*TwoRepeatedDiff*,*OneRepeated*	(2) (3)
`Reuse_bound_3`	*TwoNew*	(4)-(7)

Table 4.2: **Definition and Impact of Data Characteristics.**

Data Characteristics	Description	Impact on Performance
Tensor Size	Dimension length of a tensor	Computation, Allocation
Vector Size	Number of tensors in one vector	Computation, Allocation Communication
Data Distribution	Repeated data follows biased or unbiased distribution	Allocation Communication
Repeated Rate	Ratio of the repeated data by total data per vector	Allocation Communication

Considering the tensor pairs with different local reuse patterns and mappings impact the schedule differently (e.g., a tensor pair with *twoRepeatedSame* and mapping type (1) brings more data reuse benefits while others bring less), this work specifically maintains three *reuse bounds* according to the local reuse patterns and mappings of incoming tensor pairs. Table 4.1 explains these reuse bounds in detail.

Besides local reuse patterns (and mappings), multiple data characteristics also influence the setting of reuse bounds. Table 4.2 characterizes them in detail, particularly specifying their performance impact on either computation and/or memory operations. Because of these factors, it is challenging to set a uniform set of reuse bound values. An auto-tuning or machine learning approach is desired for finding reuse bound values.

To further support the above claim, this work leverages Spearman's rank correlation coefficient [77], a widely used approach to explore the relationships among data characteristics, three reuse bounds, and performance. The Spearman correlation unveils the correlation (whether linear or not) between two variables. All seven factors have positive impacts on the GFLOPS, as shown in Fig.4.5. *Data Distribution* and *Repeated Rate* benefit

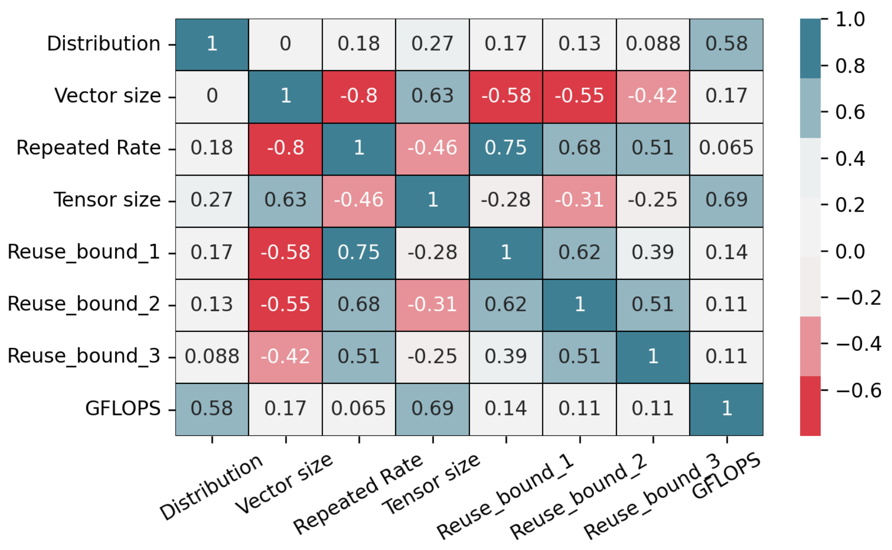


Figure 4.5: **Heatmap of the Spearman correlation coefficients.** The correlation coefficients are among data characteristics (*Data Distribution*, *Vector Size*, *Repeat Rate*, and *Tensor Size*), three reuse bounds, and GFLOPS.

data reuse to improve the GFLOPS. Larger *Vector Size* and *Tensor Size* bring more kernel computations, resulting in higher GFLOPS. Reuse bounds represent better data reuse and workload unbalance. The positive coefficients of reuse bounds illustrate that data reuse is slightly more important than workload balance. *Data Distribution* and *Repeated Rate* have positive coefficients with reuse bounds, due to the benefits of data reuse. *Vector Size* and *Tensor Size* are sensitive with workload imbalance, having negative coefficients with reuse bounds.

<u>Remarks:</u> Reuse bounds are critical to trade-off data reuse benefits and load imbalance costs; however, many factors influence the setting of reuse bounds. This fact guides us to design an approach to find the optimal reuse bounds efficiently (e.g., our regression model in Section 4.4.3).

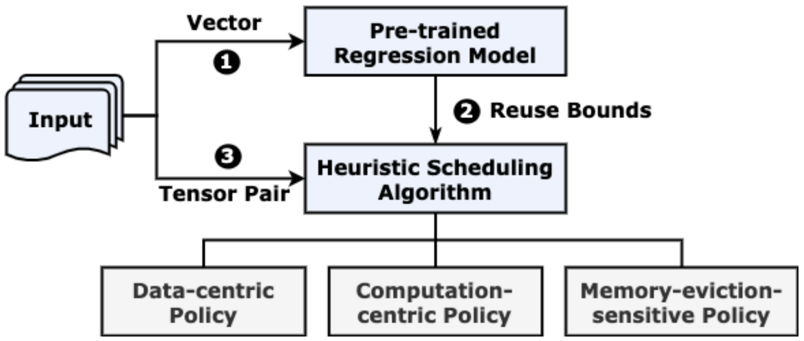

Input
Vector
❶
Pre-trained Regression Model
❷ Reuse Bounds
❸
Tensor Pair
Heuristic Scheduling Algorithm
Data-centric Policy
Computation-centric Policy
Memory-eviction-sensitive Policy

Figure 4.6: **System overview of MICCO.** Input data is tensors in vectors. MICCO dynamically handles vectors and generates GPU assignments for each vector. MICCO consists of a regression model and a heuristic scheduling algorithm. MICCO extracts data characteristics of each vector to the regression model (❶). The regression model generates optimal reuse bounds (❷). The heuristic algorithm classifies tensor pairs (❸) and jointly manages three policies.

4.4 Multi-GPU Scheduling Framework

This section introduces the design and optimization of our multi-GPU scheduling framework, MICCO. MICCO's design focuses on these aspects: 1) exploring data reuse opportunities for repeated tensors, 2) improving load balance to keep GPUs busy, and 3) achieving optimal data reuse-load balance trade-off with considerations of memory evictions.

4.4.1 System Overview

Fig. 4.6 shows an overview of MICCO that mainly consists of two components: a heuristic scheduling algorithm, and a pre-trained lightweight regression model. Fig. 4.6 also illustrates the workflow that MICCO calculates many-body correlation functions. In the first step, MICCO fetches input vectors from the upstream module of a scientific application (e.g., Lattice QCD) and feeds each vector to the pre-trained regression model (❶). In the second step, the pre-trained regression model prepares for its online inference input (e.g., the data characteristics of tensor pairs in a given vector), conducts an online inference, and outputs a set of reuse bounds for this vector (❷). Because this regression model is small, this step is lightweight incurring negligible overhead. In the third step, the heuristic

Table 4.3: Definitions of Variables.

Variable Name	Descriptions
$Tensor1$,$Tensor2$	Input tensor of one tensor pair
$reuseBd$	A vector of three reuse bounds
$tensorsGPU$	A pair between tensors and GPU
$mapGPUTensor$	GPU-Tensor pairs mappings
$mapGPUCom$	GPU-Computation cost mappings
$mapGPUMem$	GPU-Memory cost mappings
$numGPU$	The number of GPUs
$numTensor$	The number of tensors
$balanceNum$	$\lfloor numTensor\ /\ numGPU \rfloor$
$candiQueue$	A queue of candidate GPUs
$GPUMaxMemory$	The maximal memory size of GPU

scheduling algorithm takes each tensor pair (❸) in a given vector and the associated reuse bounds to assign the related tensor contraction (and its tensor pair) to a specific GPU. Particularly, the heuristic scheduling algorithm toggles among three policies to assign tensors: *data-centric policy*, *computation-centric policy*, and *memory-eviction-sensitive policy.*

4.4.2 Heuristic Scheduling Algorithm

To trade-off data reuse and load balance, this heuristic algorithm toggles among three scheduling policies. ***Data-centric policy*** emphasizes data reuse, and assigns tensors based on the aforementioned tensor pair local reuse pattern classification and mapping strategy. ***Computation-centric policy*** emphasizes workload balance, and ensures each GPU handles the identical number of tensor pairs. ***Memory-eviction-sensitive policy*** emphasizes reducing memory cost to avoid evictions.

As claimed in Sec. 4.3, data reuse is the principal factor to alleviate expensive memory operations, thus benefiting the data-intensive nature of many-body correlation functions. Therefore, the data-centric policy first dominates MICCO's scheduling to find *available* GPUs that hold the incoming tensors already, and then MICCO stores these GPUs' IDs in a queue (*candiQueue*). To decide if a GPU is available, MICCO compares a GPU's computing utilization with the reuse bounds from the regression model, i.e., if assigning

Algorithm 4 Heuristic Scheduling Algorithm

```
Require: Tensor $Tensor1$, $Tensor2$, Vector $reuseBd$, Map $mapGPUTensor$, $mapGPUCom$,
    $mapGPUMem$, Integer $balanceNum$
Ensure: A pair $tensorsGPU$
 1: Initialize $candiQueue$
 2: $GPUsofTensor1 = mapGPUTensor.find(Tensor1)$
 3: $GPUsofTensor2 = mapGPUTensor.find(Tensor2)$
 4: if $(GPUsofTensor1 \cap GPUsofTensor2) \neq NULL$ then
 5:     for $it1 : GPUsofTensor1$ do
 6:         if $it1 \in GPUsofTensor2 \cap mapGPUTensor.at(it1).size() < reuseBd[0] + balanceNum$ then
 7:             Add $it1$ to $candiQueue$
 8:         end if
 9:     end for
10: end if
11: if $candiQueue = NULL \cap (GPUsofTensor1 \neq NULL \cup GPUsofTensor2 \neq NULL)$ then
12:     for $it1 : GPUsofTensor1$ do
13:         if $mapGPUTensor.at(it1).size() < reuseBd[1] + balanceNum$ then
14:             Add $it1$ to $candiQueue$
15:         end if
16:     end for
17:     for $it2 : GPUsofTensor2$ do
18:         if $mapGPUTensor.at(it2).size() < reuseBd[1] + balanceNum$ then
19:             Add $it2$ to $candiQueue$
20:         end if
21:     end for
22: end if
23: if $candiQueue = NULL$ then
24:     for $it = 1; it \leq numGPU; it++$ do
25:         if $mapGPUTensor.at(it).size() < reuseBd[2] + balanceNum$ then
26:             Add $it$ to $candiQueue$
27:         end if
28:     end for
29: end if
30: Call Alg.5 to determine $tensorsGPU$
31: Update $mapGPUTensor$, $mapGPUCom$, $mapGPUMem$ return $tensorsGPU$
```

the incoming pair to a given GPU results in severe load imbalance, this GPU is *unavailable*. Next, the computation-centric policy dominates MICCO's scheduling to select the GPU with least computation from *candiQueue* to further balance workload. If the former scheduling causes any oversubscription of a GPU in *candiQueue*, the memory-eviction-sensitive policy kicks in to select the GPU with the most available memory in *candiQueue* to avoid data evictions.

Alg.4 illustrates the designed heuristic scheduling algorithm that processes tensor pairs one after another. Alg.5 shows the generation of an assignment between a tensor pair and a GPU by designed scheduling policies. Tab. 4.3 explains the variables in both algorithms.

The heuristic algorithm is *greedy* with $O(n^2)$ time complexity, where n is the number of tensor pairs. The outer n loop is to traverse all tensor pairs, while the inner n loop is to check previous tensor pairs in $mapGPUTensor$. The main steps of this algorithm are clarified as follows:

- ***Step-I***: Alg.4 figures out the local reuse pattern of an incoming tensor pair by checking $mapGPUTensor$ (line 2-3). If the pair belongs to $twoRepeatedSame$ (line 4), Alg.4 finds all available GPUs that hold this pair already and puts their IDs in *candiQueue* (line 5-10).
- ***Step-II***: If the $twoRepeatedSame$ pair cannot find any available GPUs or the pair belongs to $twoRepeatedDiff$ or *oneRepeated* (line 11), Alg.4 fills in *candiQueue* with the available GPUs containing one tensor of the pair (line 12-22). Otherwise, Alg.4 fills in *candiQueue* with all available GPUs (line 23-29).
- ***Step-III***: Alg.5 monitors memory cost and recognizes memory oversubscription (line 2-6). If no memory eviction occurs (line 7), Alg.5 selects the GPU with the least computation from the *candiQueue* (line 8-12). If memory eviction appears, Alg.5 selects the GPU with the most memory capacity from the *candiQueue* (line 14-19).
- ***Step-IV***: Alg.5 creates a tensor assignment (*tensorGPU*) by combining the selected GPU ID with this tensor pair (line 22), and passes it to Alg.4 (line 30). Alg.4 dynamically updates $mapGPUTensor$ after handling each tensor pair (line 31).

4.4.3 Regression Model

To determine the optimal setting of the three reuse bounds, MICCO builds a regression model to explore the correlation between data characteristics and reuse bounds. Input (feature variables) is data characteristics and output (response labels) is optimal reuse bound setting. Data characteristics include vector size, tensor size, data distribution, and repeated rate. Vector size and tensor size are given variables by input data. Data

Algorithm 5 Tensor Assignment Algorithm

```
Require: Map $mapGPUCom$, $mapGPUMem$, $mapGPUTensor$, Vector $candiQueue$, Tensor $Tensor1$,
    $Tensor2$, Integer $GPUMaxMemory$
Ensure: A pair $tensorsGPU$ Initialize vector $GPUSelect$, Integer $GPUID$, Bool $evictFlag$
 1: Integer $candidateNum = candiQueue$.size()
 2: for $id = 1; id \leq candidateNum; id++$ do
 3:     if $mapGPUMem$.at($id$).size() > GPUMaxMemory then
 4:         $evictFlag$ = TRUE
 5:     end if
 6: end for
 7: if $evictFlag \neq TRUE$ then
 8:     $GPUSelect$.add(min ($mapGPUCom$.at($id$).size(), $id \in candiQueue$))
 9:     if $GPUSelect$.size()>1 then
10:         $GPUID$ =random (min ($mapGPUMem$.at($id$).size(), $id \in candiQueue$))
11:     else
12:         $GPUID = GPUSelect$.at(0)
13:     end if
14: else
15:     $GPUSelect$.add (min ($mapGPUMem$.at ($id$).size(), $id \in candiQueue$))
16:     if $GPUSelect$.size()>1 then
17:         $GPUID$ =random(min ($mapGPUCom$.at($id$).size(), $id \in candiQueue$))
18:     else
19:         $GPUID = GPUSelect$.at(0)
20:     end if
21: end if
22: $tensorsGPU$ = make_pair ($Tensor1$, $Tensor2$, $GPUID$) return $tensorsGPU$
```

distribution is judged to be uniform or biased. Repeated rate is calculated dynamically for each vector. The regression model is offline trained once in the beginning. In offline training, the total data size is 300, 20% of which is test data to evaluate the prediction. For each set of feature variables, we measure GFLOPS of all possible values of reuse bounds and set the optimal reuse bounds to be the response labels. Reuse bounds range from 0 to $numTensor - balanceNum$ (i.e., assigning all data to one GPU). During online scheduling, MICCO extracts data characteristics of each vector and executes the inference of the pre-trained regression model to generate optimal reuse bound values.

Tab. 4.4 compares the precision of three regression models [67], including Linear Regression, Gradient Boosting, and Random Forest. R^2 Score [42] is a well-known statistical metric to measure the regression predicting quality. R^2 Score is closer to 1, the regression model is more accurate. The results in Tab. 4.4 illustrate that the correlation among data characteristics, reuse bounds, and GFLOPS is non-linear. It is also difficult to predict

Table 4.4: **R^2 Score of Regression Models**

Model Name	*LinearRegression*	*GradientBoosting*	*RandomForest*
R^2 Score	0.57	0.91	0.95

the optimal reuse bound setting by a policy-based approach. Thus, building a non-linear regression model is necessary. MICCO selects Random Forest [78] as its regression model because of its high accuracy (95%). Here are more details about this model: the learning rate of Gradient Boosting and Random Forest is 0.1, the number of boosting stages in Gradient Boosting is 150, and the number of trees in Random Forest is 150.

4.5 Evaluation

This section aims to evaluate MICCO, particularly with the following objectives: (1) proving MICCO outperforms state-of-art schedulers with varied vector sizes and data repeated rates for both uniform and non-uniform data distributions; (2) exploring the impact of reuse bounds and demonstrating MICCO obtains stable improvements with varied numbers of GPUs (scalability), tensor size, and memory oversubscription rate; (3) showing that MICCO can be integrated into a real-world system, Redstar [15, 16, 17], and yields obvious benefits on real problem sizes and datasets.

4.5.1 Experiment Setup

Platforms. MICCO is evaluated on eight AMD MI100 GPUs, each with 32G GPU memory[3]. The compiler is Rocm-4.3.0 based on clang 13.0.0. These GPUs are connected to an AMD EPYC 7502 32-Core CPU Processor.

Baseline and optimized versions. This evaluation compares MICCO with a state-of-art work, `Groute` [10], a popular and efficient multi-GPU scheduling framework. `Groute`

[3]Although MICCO is evaluated on the latest AMD GPUs, it can also run on other GPUs like other generations of AMD GPUs and NVIDIA GPUs because its design is general, and not bound with specific GPU hardware implementations.

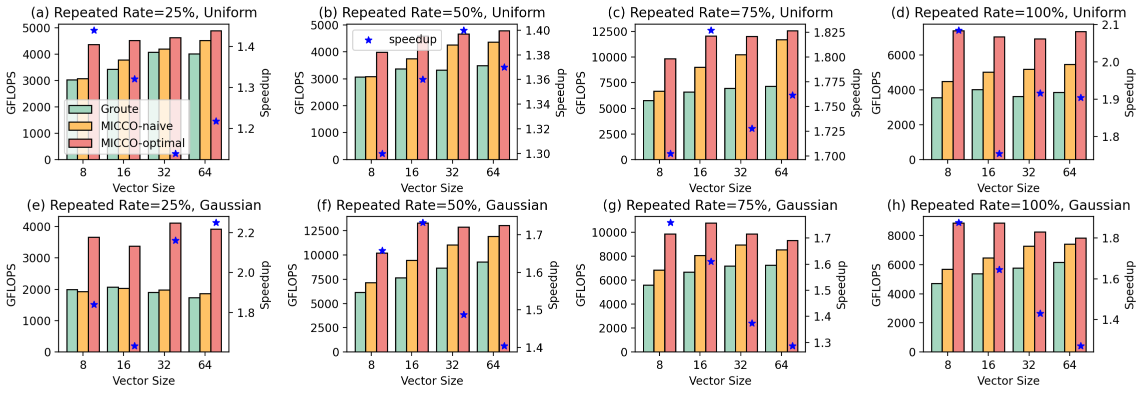


Figure 4.7: **Overall Performance.** Two distributions: Uniform (a)-(d) and Gaussian (e)-(h). **Blue stars denote speedup of `MICCO-optimal` / `Groute`.** Repeated rate varies from 25% to 100%. Vector size varies from 8 to 64. Tensor size is 384. The utilized GPU number is eight.

assigns jobs and associated data on the earliest available device to achieve good load balance similar to many other frameworks [40, 19] Two versions of MICCO are evaluated including `MICCO-naive` and `MICCO-optimal`. `MICCO-naive` does not benefit from reuse bounds (by setting these values as zero) while `MICCO-optimal` leverages reuse bounds produced from the regression model.

Evaluation setups. Our experiments extensively evaluate MICCO by changing multiple data characteristics including vector size, repeated rate, tensor size, and memory oversubscription rate. MICCO is also evaluated with varied numbers of GPUs for scalability. To evaluate the impact of data distribution, our experiments synthesize both unbiased and biased datasets. The selection of repeated data from the previous data follows two distributions: Uniform and Gaussian.

Real-world system and datasets. To further validate the practical performance, MICCO[4] is also integrated into Redstar and evaluated on three real-world correlation functions.

Table 4.5: **Execution Time (ms).** Tensor size is 384. Vector size is 64. Repeated rate is 50%. Sum of 10 vectors.

Distribution	Scheduling Overhead	Total Time
Uniform	8.27	4925.73
Gaussian	8.52	1550.88

4.5.2 Overall Performance Evaluation

Fig.4.7 illustrates the overall performance improvements by comparing MICCO with `Groute` in two distributions: Uniform and Gaussian. We measure four vector sizes from 8 to 64, and the tensor size is 384. The speedup of MICCO over `Groute` is also shown in Fig.4.11, labeled as blue stars.

Experiment results demonstrate that MICCO outperforms `Groute` in all cases, achieving up to 2.25× speedup. Fig.4.7 (a)-(d) show throughput in Uniform distribution. The optimal version of MICCO (`MICCO-optimal`) is able to achieve 1.57× geometric mean speedup than `Groute`. Fig.4.7 (e)-(h) show throughput in Gaussian distribution, and the geometric mean speedup is 1.65× than `Groute`. Compared with `MICCO-naive`, `MICCO-optimal` achieves up to 1.89× speedup. These results show the great benefits of MICCO's heuristic scheduling algorithm and regression model.

One interesting observation is growing repeated rate cannot keep improving performance, further validating the trade-off between data reuse and load balance. The best performance appears with 75% repeated rate in Uniform, and 50% repeated rate in Gaussian. Please note that repeated rate describes initial characteristics of input data rather than real reused data in calculations. Considering load balance, some repeated data has to be assigned to new GPUs for the optimal performance according to our heuristic scheduling, so improving repeated rate does not necessarily mean more data reuse.

When comparing data distributions, the following two observations also support that data reuse and load balance jointly impact the performance. One is that to reach the best

[4]`https://github.com/JeffersonLab/hadron`.

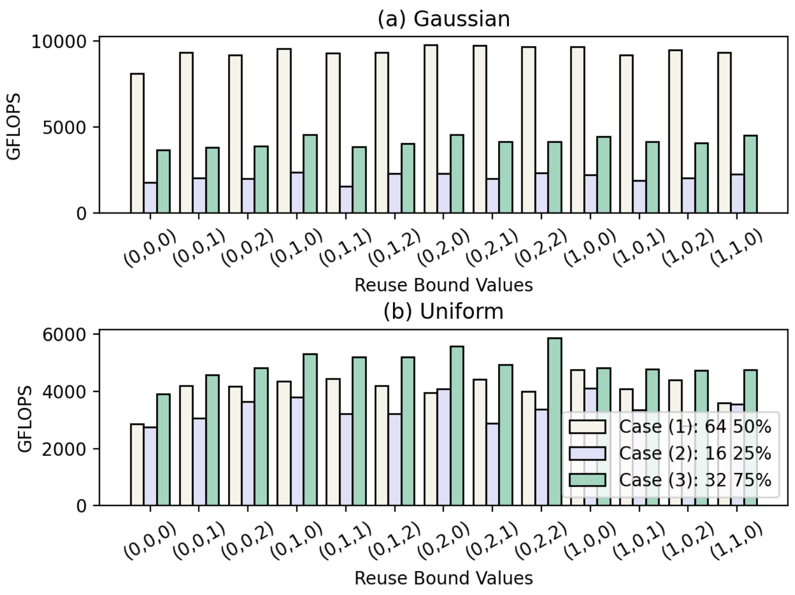


Figure 4.8: **Impact of Reuse Bounds.** Case (1) vector size = 64, repeated rate = 50%; Case (2) vector size = 16, repeated rate = 25%; Case (3) vector size = 32, repeated rate = 75%; Tensor size is 384. 13 sets of three reuse bounds are measured, and the ranging from 0 to 2.

throughput, the repeated rate of Uniform is higher than Gaussian. This is because biased distribution (like Gaussian that determines the selection of repeated data) leads to more load imbalance than Uniform distribution, and this load imbalance particularly increases with the growth of biased degree. Another observation is that a larger vector size may degrade the performance of Gaussian (as shown in Fig.4.7 (g) and (h)). This is because the increasing vector size and large repeated rate (more than 50%) produce many biased repeated data and cause increasingly severe load imbalance.

Tab.4.5 demonstrates the extremely **low scheduling overhead** of MICCO (`MICCO-optimal`), particularly compared with the total executing time (5.4‰ and 1.6‰).

4.5.3 Performance analysis

This section further studies MICCO's performance from multiple aspects. Please notice that `MICCO` in this section denotes `MICCO-optimal`.

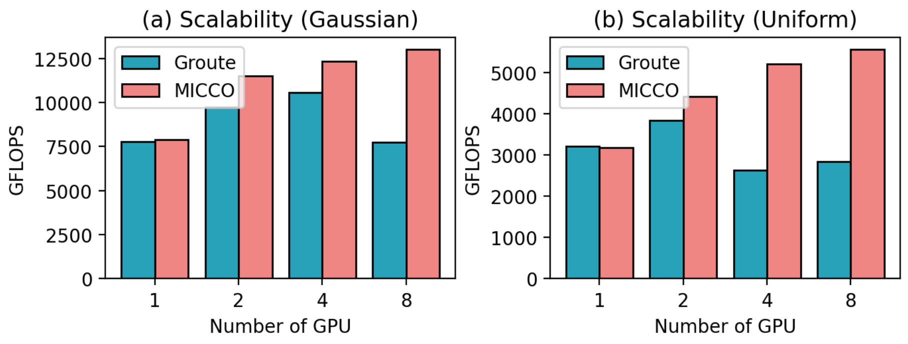


Figure 4.9: **Scalability.** Tensor size is 384. Vector size is 64.

Exploring the impacts of reuse bounds. Fig.4.8 shows the impact of changing reuse bounds on the performance. The experiments include three cases: Case (1) vector size = 64, repeated rate = 50%; Case (2) vector size = 16, repeated rate = 25%; Case (3) vector size = 32, repeated rate = 75%. This work measures thirteen values of three reuse bounds. The reuse bound values of the best performance vary when changing vector size, repeated rate, and data distribution. In Case (1) of Fig.4.8 (a), the best performance is 9753 GFLOPS with (0,2,0), but in Case (3) of Fig.4.8 (b), the best performance is 5869 GFLOPS with (0,2,2). The evaluation results explicitly show that multiple data characteristics influence data reuse-load balance trade-off and the optimal values of reuse bounds, which are hard to predict by a policy-based approach or linear regression. This observation is consistent with Tab. 4.4, and further supports the necessity of building a non-linear regression model to generate optimal reuse bounds.

Exploring scalability. This work compares MICCO with `Groute` and changes the number of GPUs from 1 to 8 in Uniform and Gaussian distributions. As shown in Fig.4.9, MICCO outperforms `Groute`, achieving up to 1.96× speedup. One observation is the slow growth of GFLOPS with an increasing number of GPUs. e.g., GFLOPS increases from 7877 GFLOPS on 1 GPU, to 13043 GFLOPS on 8 GPUs in Fig.4.9 (a). One reason is when computing small tensors (tensor size is 384), memory operation impacts more than computation on GFLOPS. Another reason is more GPUs bring more computation capacity

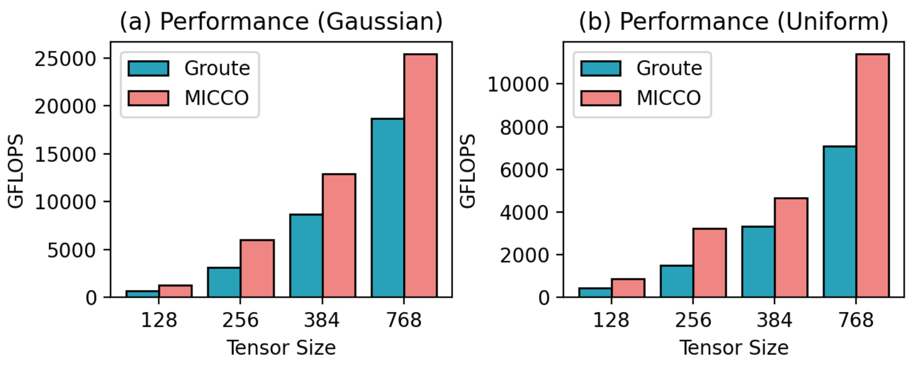


Figure 4.10: **Impact of Tensor Size.** Tensor size varies from 128 to 768. Vector size is 64. Repeated rate is 50%.

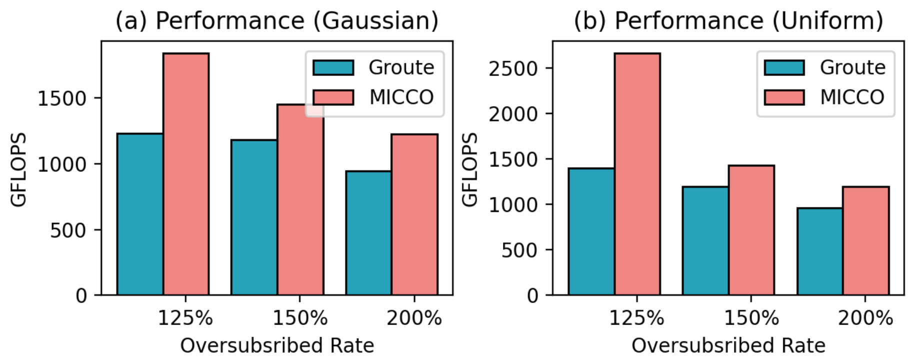


Figure 4.11: **Memory Oversubscription.** Oversubscription rate increases from 125% to 200%. Vector size is 64. Tensor size is 384. Repeated rate is 50%.

but make data reuse harder. One GPU can reuse all repeated tensors, while multiple GPUs cannot achieve full data reuse concerning load balance. The speedup improves from 1.18× on 2 GPUs to 1.68× on 8 GPUs, showing MICCO yields great benefits on leveraging data reuse and reducing memory operations.

Exploring the impact of tensor size. This work compares `Groute` and MICCO with varying tensor sizes including 128, 256, 384, 768 in two distributions. As shown in Fig.4.10, MICCO outperforms `Groute`, achieving speedup from 1.35× to 1.92×. The performance is sensitive to the tensor size, which determines the kernel computation cost. Overall, MICCO obtains better performance than `Groute` in all cases as tensor size varies.

Exploring memory oversubscription. The experiments measure two data distributions when vector size is 64, tensor size is 384, and the repeated rate is 50%. MICCO achieves a speedup up to 1.9× over `Groute`. The GFLOPS decreases with the increasing memory oversubscription. For instance, GFLOPS decreases from 1841 to 1224 in Gaussian and 2663 to 1194 in Uniform as the subscription rate increases from 125% to 200%. This observation shows that the performance is sensitive to memory evictions. The geometric mean speedup of MICCO over `Groute` is 1.4× in Gaussian and 1.2× in Uniform. Memory evictions have slightly more impacts on Uniform than Gaussian distribution.

4.5.4 Case Study: Real-world Datasets in Redstar System

Table 4.6: **Real Many-body Correlation Functions.** Total memory represents the total device memory about input and intermediate output data. 'Speedup' is based on the `Groute`.

Function	**Tensor Size**	**Memory Cost**	**Speedup**
`a1_rhopi`	128	56.05G	**1.49x**
`f0d2`	256	4645.12G	**1.41x**
`f0d4`	256	4064.48G	**1.36x**

In order to evaluate practical scenarios, this work measures three real physics correlation functions in the Redstar system. The correlation functions are `a1_rhopi` in a_1 system, and `f0d2` and `f0d4` in f_0 system. All of them belong to meson systems and consist of two-particle and single-particle constructions. Their tensor size and total device memory cost are shown in Tab. 4.6. The memory cost is the sum of sixteen time slices, including initial input data and intermediate output data. The utilized number of GPUs is eight. Vector size, repeated rate, and data distribution vary dynamically. Compared with `Groute`, the speedup achieves up to 1.49×. The experiment results demonstrate the practical significance of MICCO.

4.6 Related Works

Recently, many multi-GPU scheduling frameworks have been developed to support different types of applications [25, 31, 10, 27, 36, 49, 105, 103, 99, 98, 101]. The efforts closely related to this work include some general data-aware multi-GPU schedulers, graph processing schedulers, schedulers that support other irregular computations, streaming task schedulers aiming to process a sequence of computation kernels, and schedulers for machine learning models [102].

General data-aware Multi-GPU schedulers Although the current general data-aware GPU schedulers (as [7, 26, 87]) that consider data locality have shown efficacy on many applications based on large matrices (e.g., Matrix Multiplication, LU, etc.), it is not easy to directly apply them to our application due to multiple unique features of many-body correlation calculations. Augonnet *et al.* [7] mainly focus on reducing transfer latency and overlapping with kernel computation, while reducing data movement counts brings more significant performance gains for our application because of the large number of kernels (with reusable data). Gonthier *et al.* [26] assume the knowledge of all tasks and dependencies, but our application requires online scheduling for dynamic graphs. This work also mainly focuses on single-GPU scheduling. Teodoro *et al.* [87] mainly focus on optimizing CPU-GPU transfers instead of multi-GPUs.

Multi-GPU schedulers in graph applications and other irregular applications. Many multi-GPU graph processing schedulers [10, 19, 40] adopt greedy-based strategies that assign jobs to the earliest available devices and mainly consider workload balance. Ben *et al.* [10] present an asynchronous and runtime multi-GPU programming model for graph and irregular applications. Chen *et al.* [19] propose a task-based dynamic load-balancing solution for both single- and multi-GPU systems. These efforts mainly focus on optimizing workload balance to improve the GPU utilization without considering data reusability and the interplay between load balance and data reusability. More close to our work, Kim *et al.* [44] develop CODA that enables co-placement of compute and data for fine-grained interleaved memory with a low-cost method. Although this work takes data placement into account, it pays more attention to data locations rather than reusing data.

Multi-GPU schedulers for streaming tasks and machine learning models. Huynh *et al.* [36] propose a code generation framework mapping streaming applications onto a multi-GPU system. Melot *et al.* [64] present a crown scheduling to improve the efficiency of energy utilization. Udupa *et al.* [90] propose an efficient technique to execute stream programs on GPUs. These efforts do not consider data reuse-load balance trade-off in terms of memory oversubscription as MICCO. Many multi-GPU schedulers for machine learning workloads have been proposed recently [105, 49, 27, 29]. Gandiva [105] is a domain-specific scheduler, accelerating deep learning models by packing jobs on multiple GPUs. CROSSBOW [49] proposes a multi-GPU scheduler for deep learning with small batch sizes. These efforts have different focuses and do not study the trade-off of load balance and data reuse as MICCO, either. Additionally, many works, e.g., Tiresias [27] and Marble [29], apply preemptive scheduling approaches, which are not suitable for this work, due to the heavy overhead of suspending and resuming in many kernel computations.

4.7 Summary

This work presents MICCO, a multi-GPU scheduling framework to accelerate calculating many-body correlation functions, integrated in a real-world Lattice QCD system, Redstar system. This work extensively studies the data reuse-load balance interplay, and further brings up *local reuse pattern* and *reuse bounds.* MICCO proposes a heuristic scheduling algorithm toggling data reuse and load balance regarding memory oversubscription. Moreover, MICCO builds a regression model to predict optimal reuse bounds. In evaluation, MICCO achieves up to 2.25× speedup in synthesized datasets and 1.49× speedup in real correlation functions. In the future, we plan to extend the design of MICCO to a multi-node cluster with GPUs and implement it on an NVIDIA GPU cluster upon the availability of these devices.

Chapter 5

Locality-aware Multi-GPU Scheduling for Many-Body Correlation Functions

5.1 Problem Statement

The locality-aware multi-GPU scheduling framework has been augmented within the MICCO context. This enhancement primarily targets the alleviation of substantial memory transfer costs that arise during the pipeline input generation phase, especially when dealing with exceedingly large real-world datasets.

5.1.1 Pipeline Batch Generation

In the computation of many-body correlation functions, the Redstar system initially generates contraction graphs, subsequently partitioning them into an execution queue. As discussed in the previous sections, the nature of this execution queue is inherently unpredictable. Our frameworks have been developed to accommodate computing kernels that manage tensor pairs corresponding to each vector. To elaborate, computations are conducted on a vector-by-vector basis, wherein each vector is processed individually. All input vectors manifest in a seemingly random fashion, rendering the execution queue akin to runtime streaming data. Contrarily, in the most recent restart system, input vectors are not produced sequentially. Instead, they are generated through a pipeline process for very large dataset, implying that the real system provides a batch of vectors as input at any given instance.

5.1.2 Motivation

In the pipeline input-generation system, we identify several avenues for optimization in multi-GPU scheduling due to the grouping of vectors. Firstly, unlike a wholly unpredictable input file, this system allows for preliminary checking some vectors in advance of their scheduled processing time. Secondly, the actual input vectors exhibit variability in batch size (vector size), differing from the fixed sizes commonly assumed in theoretical models. In practice, batch sizes may reduce. Even though the system routinely receives more than one vector at a time, employing the same scheduling tactics as before is suboptimal, as variations in batch size can lead to underutilization of computational resources. Thirdly, there exists a complex interplay between the overheads of pipeline batch loading and the advantages gained from this looking-ahead activity. As the number of vectors we choose to look ahead increases, both the overhead and the algorithmic complexity escalate, complicating the task at hand. Concurrently, looking-ahead more data enhances the system's ability to schedule tensor pairs efficiently, fostering better data reuse and diminishing the need for memory transfers between GPUs.

Sections 3 and 4 introduce two frameworks: MemHC, which focuses on optimizing kernel computations on a single GPU, and MICCO, which is principally utilized to handle fixed batch sizes and process vectors sequentially. Unfortunately, these existing frameworks fall short in addressing the novel input characteristics presented by pipeline generation, varied batch sizes, and the delicate balance between cost and benefit.

5.2 Multi-GPU Scheduling

5.2.1 Interplay between Cost and Benefit of Pipeline Batching

There exists a trade-off between the batching pipeline cost and the benefits of the looking-ahead strategy. The waiting time for generating the execution queue within the Redstar system accumulates as we increase the number of vectors to preview, subsequently escalating the complexity of the scheduling algorithm. Conversely, checking more data in advance enhances the framework's ability to allocate recurring tensor pairs to the same GPU, thereby capitalizing on reusability and optimizing data use. This, in turn, facilitates a reduction in memory transfers among GPUs.

To navigate this complexity, it is imperative to formulate a performance model that meticulously balances the trade-off between the costs and benefits of pipeline batching. The overarching objective is to minimize the total overhead, taking into account the intricate interplay of costs and benefits. The relationship among overhead, costs, and benefits can be encapsulated through the regression function F, which lends itself to linear regression analysis. As costs escalate, the total overhead concurrently accumulates, with the parameter λ assuming a positive value, and μ taking on a negative value, reflecting the intricate dynamics between these variables.

$$A^{ij}_{\alpha\beta}B^{jk}_{\beta\gamma}$$

$$A^{ij}_{\alpha\beta}B^{jkl}_{\beta\gamma\delta}$$

$$A^{ijk}_{\alpha\beta\gamma}B^{klm}_{\gamma\delta\varepsilon}$$

$$Overhead = F(Cost, Benefits)$$

$$Overhead = \lambda Cost + \mu Benefits(\lambda > 0, \mu < 0)$$

The incurred costs are predominantly influenced by the quantity of batches or vectors, which not only prolongs the waiting time required for generating input tensor pairs but also amplifies the complexity of the scheduling algorithm. To accurately encapsulate this, we characterize the cost in terms of the number of tensor pairs, denoted as N, and the number of kernels, represented by K. On the flip side, the benefits derived from this process can be quantified through the enhancement in data reusability, measurable by the increased reuse ratio, R, and the diminished memory transfer ratio, M. We can approximate these relationships utilizing linear regression models, as illustrated below.

$$Cost = \lambda_1 N + \lambda_2 K$$

$$Benefits = \mu_1 R + \mu_2 M$$

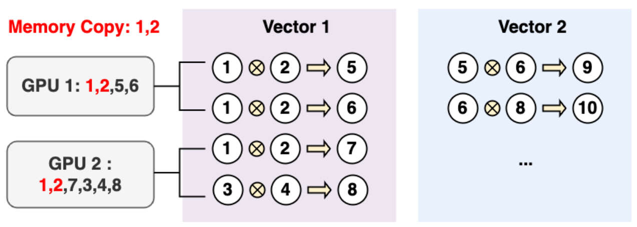
In summary, the formulas above are used to determine the optimal amount of vectors which contains the input tensor pairs.

5.2.2 Instance Analysis: locality Graph and Vector Reorganization

Having established the optimal quantity of vectors to preview for enhanced scheduling, the subsequent task involves strategizing the reorganization of these vectors to maximize data reuse. The principal methodology employed here is locality analysis.

To elucidate the intricacies of our scheduling design, we provide a practical example demonstrating the process of conducting locality analysis. Through this example, we will illustrate the steps involved in constructing local dependency graphs, a crucial endeavor for optimizing data reusability and minimizing superfluous memory copies. This helps reorganize the input tensors and ensures a more efficient utilization of resources, contributing significantly to the overall performance of the system.

In the illustrative example depicted in Fig. 5.1, we focus on the execution of two vectors, namely vector 1 and vector 2, with the intention of allocating them across two GPUs: GPU 1 and GPU 2. Vector 1 encompasses four tensor pairs scheduled for execution,



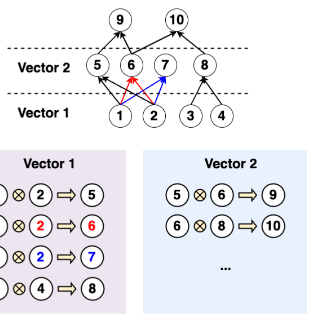


Figure 5.1: **GPU assignment instance.** Each original vector contains four tensor pairs to compute four kernels. Here are two GPUs to schedule.

Figure 5.2: **Generating local dependency graph.**

three of which involve tensors 1 and 2, commonly repeated and computed with varying indexes in real-world scenarios. The fourth tensor pair comprises tensors 3 and 4. The resulting output tensors from these operations are tensors 5, 6, 7, and 8. In the interest of achieving a balanced workload distribution, tensors 1 and 2 will be duplicated across both GPU 1 and GPU 2, resulting in two memory copies between the GPUs.

A dependency exists between vector 1 and vector 2, necessitating the execution of vector 2 subsequent to vector 1, as tensors 5, 6, and 8 (output tensors of vector 1) are prerequisites for vector 2. This dependency can be visually represented using a graph, wherein nodes symbolize tensors and edges denote dependencies between input and output tensors. Employing dependency analysis, a local dependency graph can be constructed to visualize all look-aheaded tensors, as showcased in Fig. 5.2. This graphical representation facilitates a comprehensive understanding of the inter-tensor dependencies and serves as a valuable tool for optimizing tensor allocation across GPUs, ultimately enhancing data reusability and reducing redundant memory transfers.

When considering all vectors that have been previewed as part of the look-ahead process, the scenario is illustrated in Fig. 5.3. In this specific group of preloaded inputs, each vector comprises four tensor pairs requiring computation. Vectors 1 and 2 are consistent with the configurations described in Fig. 5.1 and Fig. 5.2, respectively.

Additionally, vectors 4 and 5 are introduced in this example. For vector 4, there

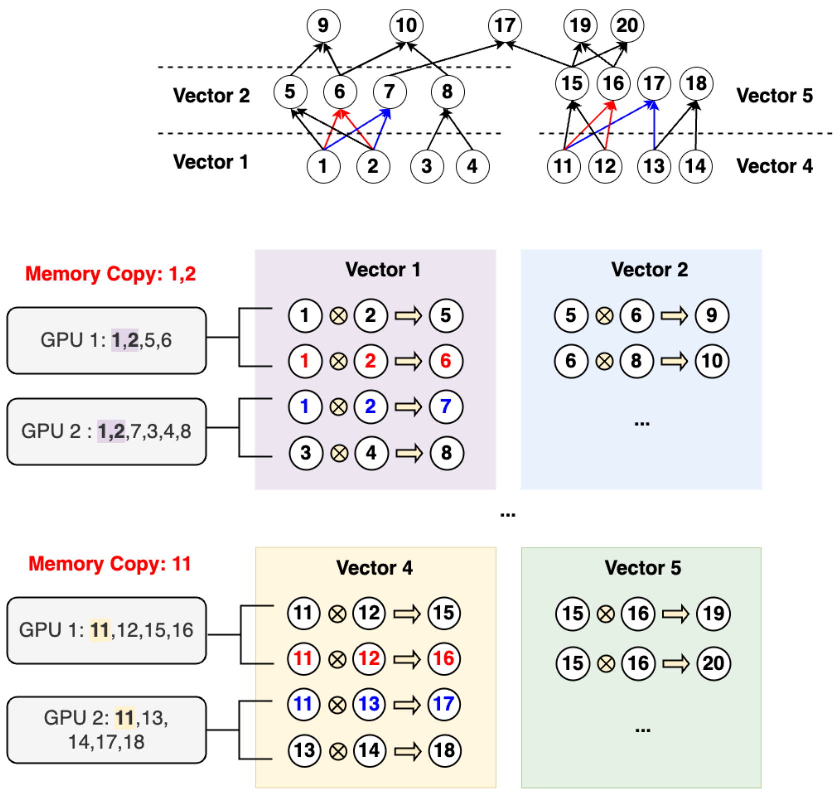


Figure 5.3: **Sub-graph generation from vectors.** Vector 1 brings two redundant memeory copies and vector 4 brings one redundnat memory copy.

are four kernel computations, which, if assigned to GPUs on a per-vector basis, would necessitate one memory copy, labeled as 11. However, it is crucial to note that the current input consists of a group of vectors rather than isolated vectors. This grouping presents a unique opportunity to reorganize these vectors through combination and reordering, potentially leading to more efficient resource utilization and reduced memory transfers.

Moreover, a local dependency graph is constructed, as depicted at the top of Fig. 5.3. This graph highlights the dependencies between vectors 1 and 2, as well as between vectors 4 and 5. Notably, there are no dependencies between vectors 1 and 4, indicating that these vectors can be processed independently of each other.

This comprehensive understanding of inter-vector dependencies, facilitated by the local dependency graph, enables a more informed and strategic approach to vector assignment

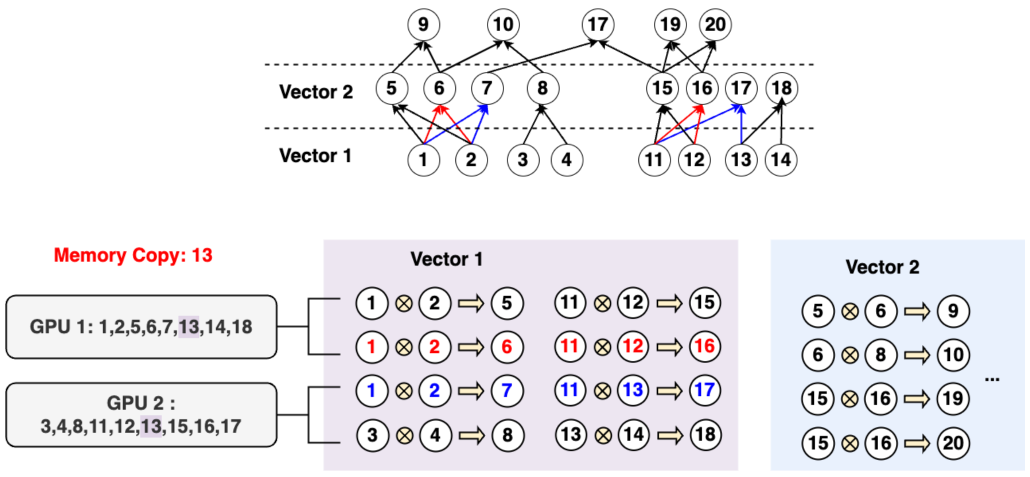


Figure 5.4: **Vector reorganization.** Generated local graphs can help reorder tensor pairs and form new organized vectors.

and scheduling on the available GPUs. By exploiting the lack of dependencies between certain vector pairs, we can optimize the scheduling process, ultimately leading to enhanced data reuse and reduced memory copy operations.

Fig. 5.4 depicts the outcome of reorganizing the vectors in the provided example. Given that there is no dependency between vector 1 and vector 4, these two vectors can be merged, allowing for the deferral of the original kernel calculations in vector 2. This fusion of vectors results in a distinctive scheduling approach.

Upon combining vectors 1 and 4, it becomes feasible to allocate four tensor pairs to each GPU to maintain a balanced workload. Consequently, the first three tensor pairs, all of which share the same input tensors 1 and 2, can be assigned to GPU 1. This strategic assignment notably reduces the redundant memory copies from three (tensors 1, 2, and 11) to just one (tensor 13).

In conclusion, the process of locality analysis encompasses evaluating reuse distances to ascertain the optimal combination size, as well as reordering the tensor pairs based on the construction of a local dependency graph. The example provided elucidates how locality analysis significantly contributes to leveraging data reusability and minimizing memory transfers, ultimately enhancing the overall efficiency of the system. Through this strategic

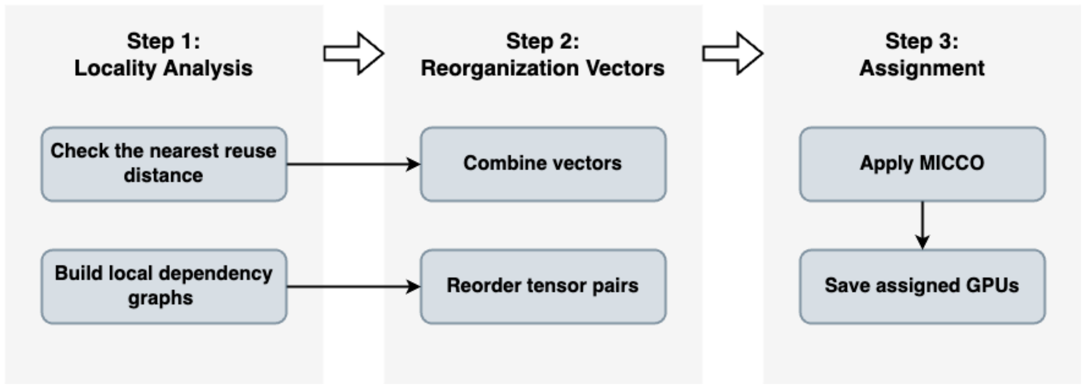


Figure 5.5: **Scheduling Algorithm Overview**

approach, we are able to optimize resource utilization and reduce overhead, paving the way for more efficient and effective GPU scheduling.

5.2.3 Locality-aware Scheduling Algorithm

The scheduling algorithm depicted in Fig. 5.5 encompasses three integral steps, each contributing to the optimal assignment of tensor pairs to GPUs.

Step 1: Locality Analysis

In this initial phase, the algorithm scrutinizes all the input vectors, evaluating the nearest reuse distance of output tensors. Absence of reuse within the group indicates no dependencies, halting further vector combinations. Conversely, vectors exhibiting dependencies are amalgamated until such dependencies become apparent. Subsequently, the algorithm constructs local dependency graphs based on the analyzed dependencies within this vector group. This enables the identification of vectors lacking dependencies, yet remaining uncombined due to their original loading order.

Step 2: Vector Reorganization

Following the locality analysis, the algorithm proceeds to reorganize the vectors. This involves reevaluating reuse distances to facilitate vector combinations, as well as utilizing the local dependency graphs to reorder or postpone certain tensor pairs between vectors. This step is crucial as it enhances data reusability and reduces memory transfers, optimizing the overall efficiency of the system.

Step 3: Tensor Pair Scheduling

The final step employs the MICCO framework to meticulously schedule the tensor pairs. The objective is to maximize the reuse of repeated tensors while maintaining a tolerable level of workload imbalance across GPUs. The MICCO framework plays a pivotal role in achieving this balance, ensuring optimal utilization of resources. The resulting assignment is then saved, providing a reusable solution for future instances involving the same input file configurations.

In summary, this three-step algorithm synergizes locality analysis, vector reorganization, and tensor pair scheduling to optimize GPU resource utilization, enhance data reusability, and reduce memory transfers. The ability to save and reuse assignment results for recurrent input configurations further contributes to the efficiency and effectiveness of the scheduling process.

5.3 Evaluation

To evaluate the performance improvements of the multi-GPU scheduler in data reusability, three real-world datasets are applied to measure the reduced memory transfers and leveraged data reuse. The baseline is to schedule a single vector each time. The memory transfer ration is calculated by the number of reduced memory copied divided by the total number of redundant memory copies produced by the baseline. The improved data reusability ratio is the number of the increased reused tensors divided by the total number of tensors. The Table 5.1 shows the detailed information.

Table 5.1: **Improvements of Real Correlation Functions.**

Function	**Assignment on 4 GPUs**		**Assignment on 8 GPUs**	
	Memory Transfers	**Reusability**	**Memory Transfers**	**Reusability**
`a0.111_A1M`	**79.93%**	**50.86%**	77.79%	61.14%
`f0.000_A1pP`	62.12%	36.20%	61.75%	44.02%
`roper.000_G1g`	70.84%	31.57%	**83.15%**	**68.57%**

Table 5.2: **Real Many-body Correlation Functions.** 'Ratio' means the impact of the baseline memory transfer on the total execution time. 'OOM' means the performance is hard to measured due to the mixed computation.

Function	**Memory Impact**	**Reduced Memory**	**Ratio**	**Speedup**
`a0.111_A1M`	41.48%	114.7 GB	79.93%	**1.49x**
`f0.000_A1pP`	64.67%	146.72 GB	62.12%	**1.67x**
`roper.000_G1g`	OOM	1.77 TB	70.84%	OOM

According to the Table 5.1, the locality-aware multi-GPU scheduling framework can achieve up to 79.93% reduced redundant memory copies in 4 GPUs and 83.15% in 8 GPUs. The scheduling framework can also achieve obvious improved data reusabiliy ratio, up to 50.86% in 4 GPUs and 68.57% in 8 GPUs.

The Table 5.2 shows the memory impact, which means the percentage of the memory transfer time of the total execution time of the baseline. The memory transfer takes up 64.67% time of the total execution in the correlation function `f0.000_A1pP`. Table 5.2 also claims the reduced memory size of the scheduler based on the baseline. The scheduler can achieve up to 1.67x times speeup.

5.4 Summary

In summary, the multi-GPU scheduler has been successfully extended to accommodate the novel input characteristics arising from the pipeline batching generation. This enhanced approach enables the looking-ahead strategy, allowing for executing a group of vectors rather than adhering to a vector-by-vector computation model. Based on the locality analysis, the scheduler constructs local dependency graphs. These graphs play a crucial role in reorganizing the input vectors, culminating in improved data reusability and a reduction in memory transfers between GPUs. This optimized process not only enhances efficiency but also contributes to a more streamlined and effective GPU scheduling experience.

Chapter 6

Conclusion

This dissertation introduces four innovative frameworks designed to expedite two distinct irregular applications across general GPU architectures, encompassing both AMD and NVIDIA GPUs.

Application 1: Parallel Recommendation System

The first application revolves around a parallel recommendation system characterized by a computation-intensive kernel. The challenges herein include data dependence, load imbalance, and the handling of large datasets. To address these issues, this dissertation unveils a CPU/GPU heterogeneous framework, meticulously crafted to implement an efficient parallel eALS-based matrix factorization recommendation system.

Application 2: Many-Body Correlation Functions

The second application pertains to the calculation of many-body correlation functions, a task that is both computation-intensive and memory-intensive. The optimization strategies for this application, therefore, necessitate a careful balance between computational load and I/O reduction. The dissertation's contributions to accelerating these correlation functions are threefold. **First**, a GPU memory management framework dedicated to eliminating multiple instances of memory redundancy, thereby optimizing memory usage. **Second**, an enhanced multi-GPU scheduling framework which aims to strike the optimal balance between load balancing and data reuse, ensuring efficient GPU utilization. **Third**, a locality-aware multi-GPU scheduler. By constructing local dependency graphs to reorganize input tensors, this scheduler enhances data reusability and minimizes memory transfers between GPUs.

These three frameworks have been seamlessly integrated into a real-world scientific system, demonstrating their practicality and effectiveness in optimizing GPU-accelerated computations. Together, these frameworks encapsulate a comprehensive solution to the challenges posed by the two aforementioned irregular applications, showcasing the dissertation's significant contribution to the field of GPU computing.

Bibliography

[1] http://docs.nvidia.com/cuda/cublas/. *Nvidia, 2015.*

[2] Ahmad Abdelfattah, Marc Baboulin, Veselin Dobrev, Jack Dongarra, Christopher Earl, Joel Falcou, Azzam Haidar, Ian Karlin, Tz Kolev, Ian Masliah, et al. High-performance tensor contractions for gpus. *Procedia Computer Science*, 80:108–118, 2016.

[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 International Symposium on High Performance Computer Architecture (HPCA)*, pages 494–506. IEEE, 2016.

[4] Ahmad Al Badawi, Bharadwaj Veeravalli, Jie Lin, Nan Xiao, Matsumura Kazuaki, and Aung Khin Mi Mi. Multi-gpu design and performance evaluation of homomorphic encryption on gpu clusters. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2020.

[5] Mohammed Fadhel Aljunid and DH Manjaiah. An improved als recommendation model based on apache spark. In *International Conference on Scientific Computation and Statistics ICSCS), year=2018, organization=Springer.*

[6] Y Aoki, T Blum, N Christ, C Dawson, K Hashimoto, T Izubuchi, JW Laiho, L Levkova, M Lin, R Mawhinney, et al. Lattice qcd with two dynamical flavors of domain wall fermions. *Physical Review D*, 2005.

[7] Cédric Augonnet, Jérôme Clet-Ortega, Samuel Thibault, and Raymond Namyst. Data-aware task scheduling on multi-accelerator based platforms. In *2010 IEEE 16th International Conference on Parallel and Distributed Systems*, pages 291–298. IEEE, 2010.

[8] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J Rossbach, and Onur Mutlu. 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*, pages 136–150, 2017.

[9] Sorav Bansal and Dharmendra S Modha. Car: Clock with adaptive replacement. In *USENIX Conference on File and Storage Technologies (FAST)*, volume 4, pages 187–200, 2004.

[10] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An asynchronous multi-gpu programming model for irregular computations. *The ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, 2017.

[11] Evan Berkowitz, Thorsten Kurth, Amy Nicholson, Bálint Joó, Enrico Rinaldi, Mark Strother, Pavlos M Vranas, and André Walker-Loud. Two-nucleon higher partial-wave scattering from lattice qcd. *Physics Letters B*, 765:285–292, 2017.

[12] Alina Bibireata, Sandhya Krishnan, Gerald Baumgartner, Daniel Cociorva, Chi-Chung Lam, P Sadayappan, J Ramanujam, David E Bernholdt, and Venkatesh Choppella. Memory-constrained data locality optimization for tensor contractions. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 93–108. Springer, 2003.

[13] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *COMPSTAT*. Springer, 2010.

[14] Guoyang Chen, Yufei Ding, and Xipeng Shen. Sweet knn: An efficient knn on gpu through reconciliation between redundancy removal and regularity. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 621–632. IEEE, 2017.

[15] Jie Chen, Robert Edwards, and Frank Winter. Graph-based contractions with optimal evaluation strategies. *ADSE03-LatticeQCD Application Strategy WBS 1.2.1.03*, (Milestone ADSE03-7), 2017.

[16] Jie Chen, Robert Edwards, and Frank Winter. Performance enhancement to the graph-based contraction calculations. *ADSE03-LatticeQCD Application Strategy WBS 1.2.1.03*, (Milestone ADSE03-7), 2018.

[17] Jie Chen, Robert Edwards, and Frank Winter. Enabling graph based contraction calculations for multi-nucleon systems. *ADSE03-LatticeQCD Application Strategy WBS 1.2.1.03*, (Milestone ADSE03-14), 2019.

[18] Jing Chen, Jianbin Fang, Weifeng Liu, Tao Tang, and Canqun Yang. clmf: A fine-grained and portable alternating least squares algorithm for parallel matrix factorization. *Future Generation Computer Systems*, 2018.

[19] Long Chen, Oreste Villa, Sriram Krishnamoorthy, and Guang R Gao. Dynamic load balancing on single-and multi-gpu systems. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[20] Maosen Chen, Tun Chen, and Qianyun Chen. An efficient implementation of the als-wr algorithm on x86 cpus. In *BMO*. Springer, 2019.

[21] Wei-Sheng Chin, Bo-Wen Yuan, Meng-Yuan Yang, Yong Zhuang, Yu-Chin Juan, and Chih-Jen Lin. Libmf: a library for parallel matrix factorization in shared-memory systems. *JMLR*, 2016.

[22] Mohammad Dashti and Alexandra Fedorova. Analyzing memory management methods on integrated cpu-gpu systems. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, pages 59–69, 2017.

[23] 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)*, pages 237–244. IEEE, 2016.

[24] Gideon Dror, Noam Koenigstein, Yehuda Koren, and Markus Weimer. The yahoo! music dataset and kdd-cup'11. In *KDD Cup 2011*. PMLR, 2012.

[25] Trilce Estrada, David A Flores, Michela Taufer, Patricia J Teller, Andre Kerstens, and David P Anderson. The effectiveness of threshold-based scheduling policies in boinc projects. In *e-Science'06*. IEEE, 2006.

[26] Maxime Gonthier, Loris Marchal, and Samuel Thibault. *Locality-Aware Scheduling of Independant Tasks for Runtime Systems*. PhD thesis, Inria, 2021.

[27] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeong-jae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A {GPU} cluster manager for distributed deep learning. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation {NSDI}*, pages 485–500, 2019.

[28] Feng-Kun Guo, Christoph Hanhart, Ulf-G Meissner, Qian Wang, Qiang Zhao, and Bing-Song Zou. Hadronic molecules. *Reviews of Modern Physics*, 90(1):015004, 2018.

[29] Jingoo Han, M Mustafa Rafique, Luna Xu, Ali R Butt, Seung-Hwan Lim, and Sudharshan S Vazhkudai. Marble: A multi-gpu aware job scheduler for deep learning on hpc systems. In *CCGRID*. IEEE, 2020.

[30] Xiangnan He, Hanwang Zhang, Min-Yen Kan, and Tat-Seng Chua. Fast matrix factorization for online recommendation with implicit feedback. In *SIGIR*, 2016.

[31] Stephen Herbein, Dong H Ahn, Don Lipari, Thomas RW Scogland, Marc Stearman, Mark Grondona, Jim Garlick, Becky Springmeyer, and Michela Taufer. Scalable i/o-aware job scheduling for burst buffer enabled hpc clusters. In *Proceedings of the International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, 2016.

[32] Balázs Hidasi and Domonkos Tikk. Fast als-based tensor factorization for context-aware recommendation from implicit feedback. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML KDD)*. Springer, 2012.

[33] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2019.

[34] Chaofeng Hou, Ji Xu, Peng Wang, Wenlai Huang, and Xiaowei Wang. Efficient gpu-accelerated molecular dynamics simulation of solid covalent crystals. *Computer Physics Communications*, 184(5):1364–1371, 2013.

[35] Yifan Hu, Yehuda Koren, and Chris Volinsky. Collaborative filtering for implicit feedback datasets. In *ICDM*. Ieee, 2008.

[36] Huynh Phung Huynh, Andrei Hagiescu, Weng-Fai Wong, and Rick Siow Mong Goh. Scalable framework for mapping streaming applications onto multi-gpu systems. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012.

[37] 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)*, pages 258–271. IEEE, 2019.

[38] John Jenkins, James Dinan, Pavan Balaji, Tom Peterka, Nagiza F Samatova, and Rajeev Thakur. Processing mpi derived datatypes on noncontiguous gpu-resident data. *IEEE Transactions on Parallel and Distributed Systems*, 25(10):2627–2637, 2013.

[39] John Jenkins, James Dinan, Pavan Balaji, Nagiza F Samatova, and Rajeev Thakur. Enabling fast, noncontiguous gpu data movement in hybrid mpi+ gpu environments. In *2012 IEEE International Conference on Cluster Computing*, pages 468–476. IEEE, 2012.

[40] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. A distributed multi-gpu system for fast graph processing. *Proceedings of the VLDB Endowment*, 11(3):297–310, 2017.

[41] Song Jiang, Feng Chen, and Xiaodong Zhang. Clock-pro: An effective improvement of the clock replacement. In *USENIX Annual Technical Conference, General Track*, pages 323–336, 2005.

[42] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 2015.

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

[44] Hyojong Kim, Ramyad Hadidi, Lifeng Nai, Hyesoon Kim, Nuwan Jayasena, Yasuko Eckert, Onur Kayiran, and Gabriel Loh. Coda: Enabling co-location of computation and data for multiple gpu systems. *ACM TACO*, 2018.

[45] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. Batch-aware unified memory management in gpus for irregular workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1357–1370, 2020.

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

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

[48] Marcin Knap and Paweł Czarnul. 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):7625–7645, 2019.

[49] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter Pietzuch. Crossbow: scaling deep learning with small batch sizes on multi-gpu servers. *arXiv preprint arXiv:1901.02244*, 2019.

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

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

[52] Chen Li, Rachata Ausavarungnirun, Christopher J Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. A framework for memory oversubscription management in graphics processing units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–63, 2019.

[53] Hao Li, Kenli Li, Jiyao An, and Keqin Li. Msgd: A novel matrix factorization approach for large-scale collaborative filtering recommender systems on gpus. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2017.

[54] Hao Li, Kenli Li, Jiwu Peng, and Keqin Li. Cusnmf: A sparse non-negative matrix factorization approach for large-scale collaborative filtering recommender systems on multi-gpu. In *ISPA/IUCC*. IEEE, 2017.

[55] Lingda Li and Barbara Chapman. 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*, pages 1–16, 2019.

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

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

[58] Ao Liu, Qiong Wu, Zhenming Liu, and Lirong Xia. Near-neighbor methods in random preference completion. In *the Association for the Advancement of Artificial Intelligence (AAAI)*, 2019.

[59] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. Why gpus are slow at executing nfas and how to make them faster. In *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[60] Jiawen Liu, Dong Li, and Jiajia Li. Athena: High-performance sparse tensor contraction sequence on heterogeneous memory. In *International Conference on Supercomputing (ICS)*, 2021.

[61] Peng Lu and Maalla Allam. Hybrid collaborative filtering recommendation algorithm for als model based on a big data platform. In *IEEE Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, 2021.

[62] Wenjing Ma, Sriram Krishnamoorthy, Oreste Villa, Karol Kowalski, and Gagan Agrawal. Optimizing tensor contraction expressions for hybrid cpu-gpu execution. *Cluster computing*, 16(1):131–155, 2013.

[63] Devin A Matthews. High-performance tensor contraction without blas. *SIAM Journal on Scientific Computing*, 40, 2016.

[64] Nicolas Melot, Christoph Kessler, Jörg Keller, and Patrick Eitschberger. Fast crown scheduling heuristics for energy-efficient mapping and scaling of moldable streaming tasks on manycore systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2015.

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

[66] Alok Mishra, Lingda Li, Martin Kong, Hal Finkel, and Barbara Chapman. Benchmarking and evaluating unified memory for openmp gpu offloading. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–10, 2017.

[67] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[68] Paritosh Nagarnaik and A Thomas. Survey on recommendation system methods. In *IEEE International Conference on Electronics Circuits and Systems (ICECS)*. IEEE, 2015.

[69] Thomas Nelson, Axel Rivera, Prasanna Balaprakash, Mary Hall, Paul D Hovland, Elizabeth Jessup, and Boyana Norris. Generating efficient tensor contractions for gpus. In *2015 44th International Conference on Parallel Processing*, pages 969–978. IEEE, 2015.

[70] Israt Nisa, Aravind Sukumaran-Rajam, Rakshith Kunchum, and P Sadayappan. Parallel ccd++ on gpu for matrix factorization. In *General-Purpose Computation on Graphics Processing Units (GPGPUs)*. 2017.

[71] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. Patdnn: Achieving real-time dnn execution on

mobile devices with pattern-based weight pruning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 907–922, 2020.

[72] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[73] István Pilászy, Dávid Zibriczky, and Domonkos Tikk. Fast als-based matrix factorization for explicit and implicit feedback datasets. In *The ACM Recommender Systems Conference (RecSys)*, 2010.

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

[75] Shane Ryoo, Christopher I Rodrigues, Sara S Baghsorkhi, Sam S Stone, David B Kirk, and Wen-mei W Hwu. 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*, pages 73–82, 2008.

[76] John E Savage and Mohammad Zubair. Evaluating multicore algorithms on the unified memory model. *Scientific Programming*, 17(4):295–308, 2009.

[77] Philip Sedgwick. Spearman's rank correlation coefficient. *Bmj*, 2014.

[78] Tao Shi and Steve Horvath. Unsupervised learning with random forest predictors. *Journal of Computational and Graphical Statistics*, 2006.

[79] 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), pages 193–202. IEEE, 2016.

[80] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. In *Euro-Par*. Springer, 2014.

[81] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. Whippletree: Task-based scheduling of dynamic workloads on the gpu. *ACM Transactions on Graphics (TOG)*, 2014.

[82] Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David Kaeli. Hetero-mark, a benchmark suite for cpu-gpu collaborative computing. In *IISWC*. IEEE, 2016.

[83] SN Syritsyn, JD Bratt, MF Lin, HB Meyer, JW Negele, AV Pochinsky, M Procura, M Engelhardt, Ph Hägler, TR Hemmert, et al. Nucleon electromagnetic form factors from lattice qcd using 2+ 1 flavor domain wall fermions on fine lattices and chiral perturbation theory. *Physical Review D*, 2010.

[84] Wei Tan, Liangliang Cao, and Liana Fong. Faster and cheaper: Parallelizing large-scale matrix factorization on gpus. In *Proceedings of the International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, 2016.

[85] Wei Tan, Shiyu Chang, Liana Fong, Cheng Li, Zijun Wang, and LianGLiang Cao. Matrix factorization on gpus with memory optimization and approximate computing. In *2018 International Conference on Parallel Processing (ICPP)*, 2018.

[86] George Teodoro, Tahsin M Kurc, Tony Pan, Lee AD Cooper, Jun Kong, Patrick Widener, and Joel H Saltz. Accelerating large scale image analyses on parallel, cpu-gpu equipped systems. In *IEEE Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2012.

[87] George Teodoro, Tony Pan, Tahsin M Kurc, Jun Kong, Lee AD Cooper, Norbert Podhorszki, Scott Klasky, and Joel H Saltz. High-throughput analysis of large microscopy image datasets on cpu-gpu cluster platforms. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 103–114. IEEE, 2013.

[88] Andres Tomas, Chia-Chen Chang, Richard Scalettar, and Zhaojun Bai. Advancing large scale many-body qmc simulations on gpu accelerated multicore systems. In *Proceedings of the 31st IEEE International Parallel Distributed Processing Symposium (IPDPS)*, pages 308–319, 2012.

[89] Yaohung M Tsai, Weichung Wang, and Ray-Bing Chen. Tuning block size for qr factorization on cpu-gpu hybrid systems. In *International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, 2012.

[90] Abhishek Udupa, R Govindarajan, and Matthew J Thazhuthaveetil. Software pipelined execution of stream programs on gpus. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2009.

[91] Pedro Valero-Lara, Ivan Martínez-Pérez, Raül Sirvent, Xavier Martorell, and Antonio J Pena. Nvidia gpus scalability to solve multiple (batch) tridiagonal systems implementation of cuthomasbatch. In *International Conference on Parallel Processing and Applied Mathematics*, pages 243–253. Springer, 2017.

[92] 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*, pages 308–316. IEEE, 2011.

[93] Qihan Wang, Wei Niu, Li Chen, Ruoming Jin, and Bin Ren. 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)*. IEEE, 2021.

[94] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2016.

[95] Brian Wheatman and Helen Xu. Packed compressed sparse row: A dynamic graph representation. In *2018 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2018.

[96] Manda Winlaw, Michael B Hynes, Anthony Caterini, and Hans De Sterck. Algorithmic acceleration of parallel als for collaborative filtering: Speeding up distributed big data recommendation in spark. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2015.

[97] Q Wu, C Brinton, Z Zhang, M Cucuringu, A Pizzoferrato, and Z Liu. Equity2vec: End-to-end deep learning framework for cross-sectional asset pricing. In *2nd ACM International Conference on AI in Finance*, 2021.

[98] Qiong Wu, G. Brinton Christopher, Zhang Zheng, Pizzoferrato Andrea, LIU Zhenming, and Cucuringu Mihai. Equity2vec: End-to-end deep learning framework for cross-sectional asset pricing. *International Conference on AI in Finance*, 2021.

[99] Qiong Wu, Adam Hare, Sirui Wang, Yuwei Tu, Zhenming Liu, Christopher G Brinton, and Yanhua Li. Bats: A spectral biclustering approach to single document topic modeling and segmentation. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2021.

[100] Qiong Wu, Adam Hare, Sirui Wang, Yuwei Tu, Zhenming Liu, Christopher G Brinton, and Yanhua Li. Bats: A spectral biclustering approach to single document topic modeling and segmentation. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 12(5):1–29, 2021.

[101] 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 IEEE 13th International Conference on Semantic Computing (ICSC)*, pages 279–286. IEEE, 2019.

[102] Qiong Wu and Zhenming Liu. Rosella: A self-driving distributed scheduler for heterogeneous clusters. *arXiv preprint arXiv:2010.15206*, 2020.

[103] Qiong Wu, Felix M Wong, Yanhua Li, Zhenming Liu, and Varun Kanade. Adaptive reduced rank regression. *Advances in Neural Information Processing Systems*, 33:4103–4114, 2020.

[104] Qiong Wu, Felix Ming Fai Wong, Zhenming Liu, Yanhua Li, and Varun Kanade. Adaptive reduced rank regression. *arXiv preprint arXiv:1905.11566*, 2019.

[105] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In {*OSDI*}, pages 595–610, 2018.

[106] LI XIE, WENBO ZHOU, AND YAOSEN LI. Application of improved recommendation system based on spark platform in big data analysis. *Cybernetics and Information Technologies*, 2016.

[107] XIAOLONG XIE, WEI TAN, LIANA L FONG, AND YUN LIANG. Cumf_sgd: Parallelized stochastic gradient descent for matrix factorization on gpus. In *Proceedings of the International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, 2017.

[108] MARTIN ZINKEVICH, MARKUS WEIMER, LIHONG LI, AND ALEX J SMOLA. Parallelized stochastic gradient descent. In *Proceedings of the International Conference on Neural Information Processing Systems (NIPS)*, 2010.

VITA

Qihan Wang

Qihan Wang is a PhD Candidate in the Department of Computer Science at The College of William & Mary. Her PhD advisor is Prof. Bin Ren. Her research interests mainly include High Performance Computing, GPU architecture, and machine learning. Her PhD research works have been accepted by IPDPS 2022, TACO 2021, HiPC 2021 and Smart Health 2020. Previously, she received her Bachelor of Software Engineering at Beihang University in 2017.