Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

A perceptual and predictive batch-processing memory scheduling strategy for a CPU-GPU heterogeneous system[*]

Juan FANG[‡], Sheng LIN, Huijing YANG, Yixiang XU, Xing SU

Faculty of Information Technology, Beijing University of Technology, Beijing 100124, China

E-mail: fangjuan@bjut.edu.cn; lins@emails.bjut.edu.cn; yangkx@emails.bjut.edu.cn; xuyx@emails.bjut.edu.cn; suxing@bjut.edu.cn

Received Oct. 11, 2022; Revision accepted Jan. 4, 2023; Crosschecked Feb. 23, 2023

Abstract: When multiple central processing unit (CPU) cores and integrated graphics processing units (GPUs) share off-chip main memory, CPU and GPU applications compete for the critical memory resource. This causes serious resource competition and has a negative impact on the overall performance of the system. We describe the competition for shared-memory resources in a CPU-GPU heterogeneous multi-core architecture, and a shared-memory request scheduling strategy based on perceptual and predictive batch-processing is proposed. By sensing the CPU and GPU memory request conditions in the request buffer, the proposed scheduling strategy estimates the GPU latency tolerance and reduces mutual interference between CPU and GPU by processing CPU or GPU memory requests in batches. According to the simulation results, the scheduling strategy improves CPU performance by 8.53% and reduces mutual interference by 10.38% with low hardware complexity.

Key words: CPU-GPU heterogeneous; Multi-core; Unified memory; Access scheduling
https://doi.org/10.1631/FITEE.2200449 **CLC number:** TP391.9

1 Introduction

With the growing demand for computing, heterogeneous computing has drawn extensive attention. Modern systems-on-a-chip usually consist of multiple types of cores with different functions (Chen et al., 2017). This approach is of interest because heterogeneous architectures can greatly improve computational efficiency. The widespread use of image processing units such as graphics processing units (GPUs) for image rendering and scientific computing makes CPU-GPU (here, CPU is short for central processing unit) heterogeneous architectures the most used heterogeneous computing systems in our lives. Today, high-end systems-on-a-chip include powerful CPU and GPU cores, both of which have significant memory system requirements.

CPU-GPU heterogeneous computing systems are classified as separated and integrated CPU-GPU heterogeneous architectures (Mittal and Vetter, 2015), as shown in Fig. 1. In the separated architecture, a large amount of data needs to be transmitted between the host and the device through a peripheral component interconnect express (PCIe) bus, but the transmission efficiency of the PCIe bus is limited by its own implementation mechanism and cannot reach the theoretical peak. This problem becomes a performance bottleneck in CPU-GPU heterogeneous computing systems. The integrated GPU and CPU share off-chip main memory, which is called a unified memory architecture (UMA). The architecture avoids the problem of memory copying and simultaneously reduces power consumption and extends endurance effectively. AMD's accelerated processing unit (APU) (Bouvier et al., 2014), Apple's

[‡] Corresponding author

[*] Project supported by the National Natural Science Foundation of China (Nos. 62276011 and 61202076) and the Natural Science Foundation of Beijing, China (No. 4192007)

ORCID: Juan FANG, https://orcid.org/0000-0002-4542-8727

M1 series, and others use this architecture.

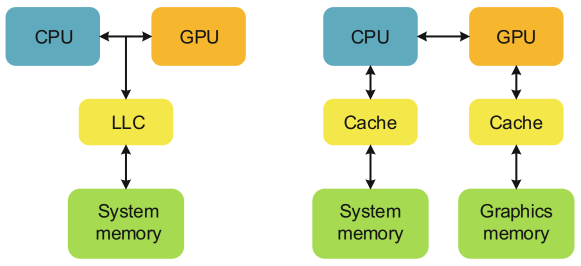

Fig. 1 Integrated and separated CPU-GPU heterogeneous architectures (CPU: central processing unit; GPU: graphics processing unit; LLC: last level cache)

It is found that while the CPU-GPU heterogeneous architecture with unified memory eliminates data transmission between components and reduces latency, UMA has serious data consistency and shared-resource contention issues (Power et al., 2013; Zhang et al., 2017; Hazarika et al., 2020). Today, unified memory is a critical shared resource in heterogeneous multi-core architectures, but memory requests from CPU and GPU applications on different cores are interleaved in the request buffer, which reduces the efficiency of existing memory scheduling. Applications with a large number of memory requests, particularly GPU applications, seriously interfere with the execution of memory-sensitive applications and reduce their performance. As the number of workloads increases in the CPU-GPU heterogeneous architecture with unified memory, the interference between CPU and GPU core memory requests becomes more severe. Unfortunately, most existing memory request scheduling algorithms require a relatively complex hardware implementation or a large and expensive request buffer to provide sufficient visibility of the entire global request stream, which is very difficult to implement on real machines. The first-ready first-come first-serve (FRFCFS) scheduling algorithm is still used on most real machines. Therefore, an access scheduling algorithm with low hardware complexity that can effectively reduce CPU and GPU interference is necessary.

To reduce CPU-GPU interference and satisfy the requirements of low hardware complexity, in this study we propose a perceptual and predictive batch-processing (PPBP) memory scheduling strategy, which has three main phases: (1) sensing CPU and GPU memory access requests in the request buffer, (2) estimating the GPU latency tolerance, and (3) using the FRFCFS scheduling algorithm to process CPU and GPU memory requests in batches.

This paper makes the following contributions:

1. We propose a simple and effective scheme to predict GPU latency tolerance. We predict GPU latency tolerance by accurately predicting the number of threads being executed in GPU by sensing the execution in the memory controller.

2. We propose the PPBP scheduling strategy. It reduces the interference between CPU and GPU memory requests and improves CPU performance without affecting GPU performance, while incurring low hardware cost and complexity.

3. We build a CPU-GPU heterogeneous system and perform a detailed comparison of PPBP with several other scheduling algorithms. Our results show that PPBP has better scalability, can effectively improve CPU performance by 8.53%, and can reduce CPU-GPU interference by 10.38%.

2 Related works

Currently, memory request scheduling research on the CPU homogeneous multi-core architecture, integrated GPU architecture, and separated GPU architecture is a popular research topic. Research on the CPU homogeneous multi-core architecture attempts to improve throughput and thread fairness in shared dynamic random-access memory (DRAM) systems (Mutlu and Moscibroda, 2008; Kim et al., 2010; Subramanian et al., 2015; di Sanzo et al., 2020). The shader core is the primary focus on GPU memory request scheduling research. Researchers are primarily concerned with minimizing the difference in latency between threads and increasing the core memory access speed with low latency tolerance (Jog et al., 2013; Fang et al., 2015; Wang HN and Jog, 2019; Lin et al., 2020; Wang QH et al., 2022).

With the development of the CPU-GPU heterogeneous architecture with unified memory, there has been extensive research on shared-memory competition, and some studies have provided dedicated memory request schedulers for heterogeneous systems. GPU requests seriously interfere with CPU requests in shared memory, because GPU requests occupy most space in the request buffer and limit the analysis of CPU application memory access behaviors. To alleviate this problem, CPU and GPU requests need to be separated. To separate CPU and GPU requests and reduce interference, the segmented memory scheduler (SMS) (Ausavarungnirun et al., 2012) divides all requests in the request buffer into source-specific batches according to the request source. To accomplish the same goal as SMS, criticality-aware memory scheduling (CAMS) (Fang et al., 2020) establishes two request buffers in the memory controller to store CPU and GPU memory requests, separately.

More works tend to analyze GPU execution and then dynamically prioritize GPU memory requests. A dynamic priority scheduler adopts tile-based deferred rendering (TBDR) for dynamic progress estimation (Jeong et al., 2012). This scheduler gives GPU and CPU equal access priority when the GPU rendering rate lags behind the target rendering rate. During the last 10% of the rendered frames, GPU access takes precedence over CPU access. To estimate the dynamic progress of GPU workloads, deadline-aware memory scheduler for heterogeneous systems (DASH) scheduling (Usui et al., 2016) statically partitions the physical address space between CPU and GPU datasets and allocates two independent memory controllers to schedule access. CLAMS (Jog et al., 2016) is a new memory request scheduling algorithm for GPU memory access scheduling, which considers the latency tolerance of the cores that generate memory requests. The key idea of CLAMS is to use the proportion of critical requests in the memory request buffer to switch between critical and local optimization scheduling strategies.

3 Motivation

CPU and GPU architectures determine their memory access characteristics. GPU is used primarily for a large number of parallel computations because it has many computing cores. GPU schedules cyclically in multiple applications, which results in a large number of memory requests. CPU performs more logical computations than GPU and assigns a larger number of calculation workloads to GPU, which results in fewer memory requests of CPU. Figs. 2a and 2b show the CPU and GPU memory request intensities, respectively. According to our simulation results on the constructed CPU-GPU heterogeneous system model, the average number of GPU memory requests per unit time is 50 times the number of CPU memory requests. At the same time, due to program location, when the applications run only on the CPU or GPU systems, the row buffer hit rate of CPU applications is between about 40% and 80% and the row buffer hit rate of GPU applications is between about 45% and 85%. If the applications run on a CPU-GPU heterogeneous system with CPU and GPU interference, the row buffer hit rate of CPU applications is reduced by 23%–35%, as shown in Fig. 3a, and the row buffer hit rate of GPU applications is reduced by 11%–25%, as shown in Fig. 3b.

When CPU and GPU compete for shared resources, they are affected differently because of their differences in memory request characteristics. As the number of workloads increases, the effects become more severe. Figs. 4a and 4b present the performance of CPU and GPU benchmarks when executed with varying numbers of benchmarks, respectively.

Compared to individual execution, simultaneous execution of two benchmarks results in an average decrease of 3.72% in CPU performance and 1.12% in GPU performance. When four benchmarks are executed, the average performance drop is

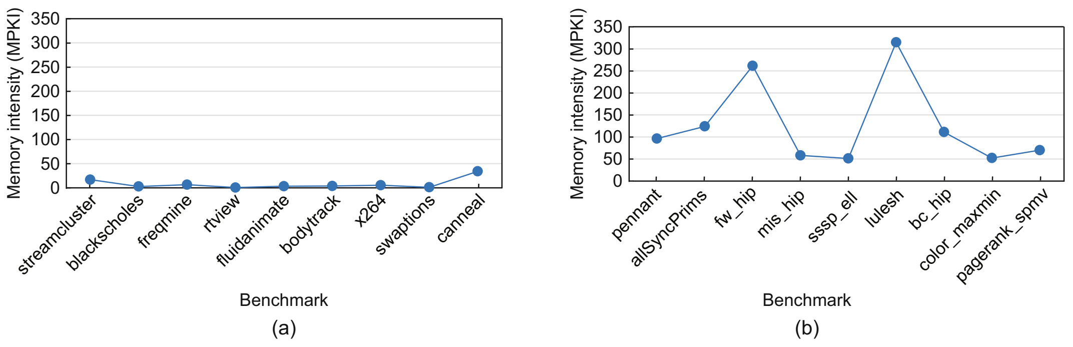

Fig. 2 Comparison of memory intensity between CPU (a) and GPU (b) applications (MPKI: misses per thousand instructions)

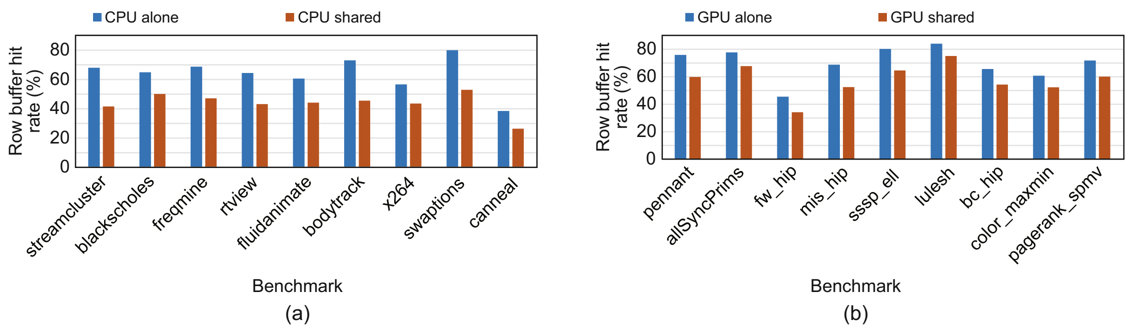

Fig. 3 Comparison of row buffer hit rate between CPU (a) and GPU (b) applications

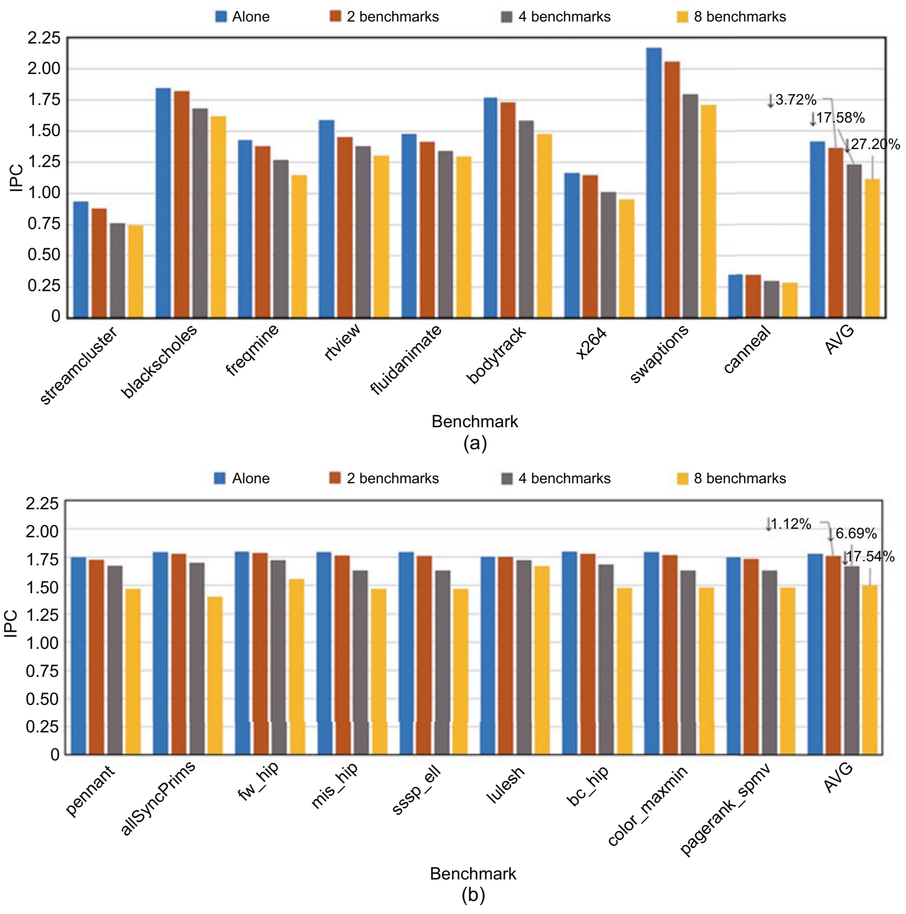

Fig. 4 CPU (a) and GPU (b) throughput when executing different numbers of benchmarks simultaneously (IPC: instructions per cycle)

17.58% for CPU and 6.69% for GPU. Similarly, when eight benchmarks are executed concurrently, the average performance decreases by 27.20% for CPU and 17.54% for GPU. Interference between CPU and GPU applications has a devastating impact on CPU performance when CPU and GPU share memory. So, CPU and GPU requests must be separated to reduce the interference between them.

GPU cores of different operating states have different latency tolerance for memory requests. By analyzing the GPU execution principle, when a core has a large number of warps that are stalled due to memory access, the latency tolerance of the core is low. In contrast, if a core has a large number of ready warps, the core can cyclically schedule ready warps to hide memory access latency. As a result, perceiving the numbers of ready warps and memory access warps per core can effectively estimate the latency tolerance of the core. By estimating the latency tolerance of all cores, we can dynamically assign different priorities to cores to improve the overall performance of the system. However, most of existing methods (Jog et al., 2016; Rai and Chaudhuri, 2017; Bitalebi and Safaei, 2023) are complicated and difficult to implement on hardware, so a simple and low-hardware-dependency method must be found.

In light of the problems discovered during the simulations, the following solutions are proposed in this study:

1. To address the problem that CPU and GPU memory requests interfere with each other and to greatly reduce CPU performance, we separate CPU and GPU requests by processing CPU and GPU memory requests in batches, respectively. FRFCFS is used to schedule and process only CPU or GPU requests over a period of time. This method eliminates the need to force the separation of CPU and GPU requests in the request buffer, reduces CPU and GPU interference, and decreases hardware complexity.

2. In view of the fact that existing methods for the delay tolerance estimation of the GPU core are too complicated, we propose a new calculation method to estimate the GPU delay tolerance. The new method predicts the number of execution threads in GPU by counting the number of GPU threads in the memory controller request buffer over a period of time. The memory access threads in CPU and GPU are threads that exist in the current request buffer. We estimate the latency tolerance of GPU through these two metrics.

3. CPU is more severely affected when CPU and GPU access the memory at the same time. In response to this issue, we appropriately increase the CPU memory request batch size to improve the CPU memory request priority within the range of GPU latency tolerance.

4 PPBP scheduling strategy

The PPBP scheduling strategy proposed in this study consists of three stages. In the first stage, the requests for CPU and GPU memory access in the request buffer are perceived, including the numbers of CPU and GPU requests and the number of GPU threads. In the second stage, the latency tolerance of GPU is estimated and the priorities of CPU and GPU memory access are adjusted by modifying the batch sizes of CPU and GPU memory requests. In the third stage, CPU and GPU requests are scheduled and processed in batches through the FRFCFS scheduling algorithm to reduce the interference between CPU and GPU requests.

4.1 Variables of PPBP

The first stage of the PPBP scheduling strategy needs to keep track of several variables. First, we introduce the concept of quantum, which represents a fixed period of system time. Second, we introduce the concept of cycle, and multiple quanta form a cycle. During each quantum, the PPBP scheduling strategy will count the number of CPU requests, the number of GPU requests, and the number of memory access threads generated by GPU in the request buffer. In the entire cycle, we calculate the total number of memory access threads generated by GPU. These data are used to dynamically estimate the latency tolerance and adjust the priorities of CPU and GPU memory access requests.

Based on the PPBP scheduling strategy, if a quantum is too long, CPU and GPU memory requests will be suspended for a long time, while if a quantum is too short, the performance of PPBP cannot be exerted, so we set the range of a quantum to 100 000 ticks. Because long or short cycles will increase the estimation error of the number of GPU threads and cause inaccurate estimation of the GPU delay tolerance, we set four quanta as one cycle after many simulations. Because CPU and GPU are affected differently, a higher priority is provided for CPU by default. The coefficient of the GPU priority is set to 0.5 based on the simulation results, and we denote it by symbol G'.

4.2 Latency tolerance estimation of GPU

To reduce the complexity of existing scheduling strategies for the estimation of GPU core execution, PPBP dynamically estimates only GPU core execution by counting the number of memory access requests in the request buffer. In this study, the number of GPU memory access threads that exist in the request buffer is used to represent the number of threads that GPU establishes for memory access, and we denote it by $\text{GPUThread}_{\text{quantum}}$. The memory access threads follow a Pareto distribution, which means that the longer the workload runs, the longer it is expected to run. According to Pareto distribution, we predict the number of all threads in the current GPU by the number of all GPU memory access threads that have appeared in a cycle, which is denoted by $\mathrm{GPUThread}_{\mathrm{cycle}}$.

Fig. 5 shows the actual number of GPU threads, the predicted number of GPU threads, and the estimation error. According to the results, our method can achieve an estimation accuracy of >85%. We estimate the latency tolerance of GPU by predicting the numbers of executing threads and threads accessing memory in GPU. The method for calculating GPU latency tolerance is shown in Eq. (1):

$$\mathrm{LatencyTolerance} = \frac{\mathrm{GPUThread}_{\mathrm{cycle}}}{\mathrm{GPUThread}_{\mathrm{quantum}}}. \tag{1}$$

GPU latency tolerance decreases as the ratio of the number of memory access threads to the total number of threads increases. Based on the estimated GPU delay tolerance, the memory access priorities and the batch sizes of CPU and GPU memory requests are dynamically adjusted.

4.3 Batch size of memory requests

Two variables are considered when PPBP adjusts the batch sizes of CPU and GPU memory requests: the ratio of the number of GPU memory requests to the total number of memory requests and the estimated GPU latency tolerance:

$$\mathrm{GPU}_{\mathrm{PRO}} = G' \frac{\mathrm{numREQ}_{\mathrm{GPU}}}{\mathrm{numREQ}_{\mathrm{CPU}} + \mathrm{numREQ}_{\mathrm{GPU}}} \cdot \left(1 + \frac{1}{\mathrm{LatencyTolerance}}\right). \tag{2}$$

According to the equation, when the number of GPU memory requests increases and the GPU latency tolerance decreases, the memory access priority and the batch size of GPU memory requests increase.

PPBP controls the batch size by controlling the time to process CPU or GPU batches. Algorithm 1 shows the core code of the PPBP scheduling algorithm. We divide a quantum into two parts, one for processing CPU batches and the other for processing GPU batches. We dynamically adjust the batch sizes of two periods based on the GPU latency tolerance.

Algorithm 1 Perceptual and predictive batch-processing (PPBP)

Input: $\mathrm{numREQ}_{\mathrm{CPU}}$, $\mathrm{numREQ}_{\mathrm{GPU}}$, and LatencyTolerance

Output: Schedulability of the memory requests

```
1: if $\mathrm{numREQ}_{\mathrm{CPU}} \neq 0$ and $\mathrm{numREQ}_{\mathrm{GPU}} \neq 0$ then
2:     compute the GPU proportion $\mathrm{GPU}_{\mathrm{PRO}}$
3: else if $\mathrm{numREQ}_{\mathrm{CPU}} = 0$ then
4:     $\mathrm{GPU}_{\mathrm{PRO}} = 1$
5: else
6:     $\mathrm{GPU}_{\mathrm{PRO}} = 0$
7: end if
8: nextQuantum = Quantum
9: if isCPU and $\mathrm{GPU}_{\mathrm{PRO}} \neq 0$ then
10:    isCPU = false
11:    nextQuantum$*= \mathrm{GPU}_{\mathrm{PRO}}$
12: else if !isCPU and $\mathrm{GPU}_{\mathrm{PRO}} \neq 1.0$ then
13:    isCPU = true
14:    nextQuantum$*= 1 - \mathrm{GPU}_{\mathrm{PRO}}$
15: end if
```

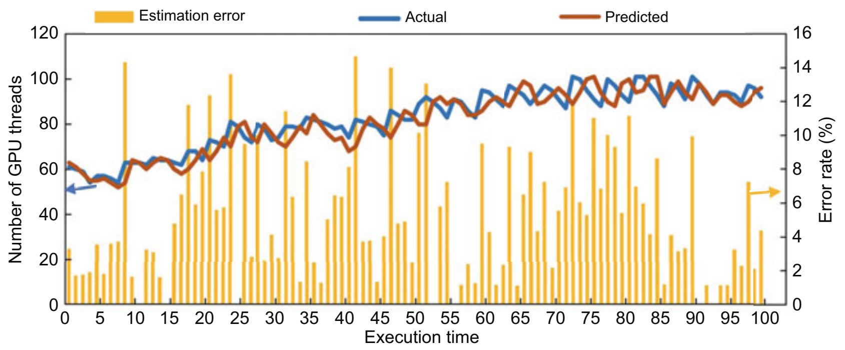

Fig. 5 Estimated number of GPU threads (Data are collected at an interval of five quanta, and one scale of the execution time spans 5×10^5 ticks)

4.4 Hardware implementation

PPBP abandons the GPU latency tolerance acquisition scheme, which is difficult to implement on hardware in existing studies, and estimates the GPU latency tolerance in a novel and simple way. The scheme sacrifices a little accuracy, but greatly reduces the complexity of hardware implementation. First, PPBP needs to count the numbers of CPU and GPU requests in the request buffer of the memory controller and the number of threads of GPU that are accessing memory in a quantum. To implement the quantum, each memory controller has a duration counter. Then, at the end of each quantum, the number of GPU access threads in a cycle and the above statistics are updated. The GPU latency tolerance is also calculated according to Eq. (1), and the priority of GPU is calculated according to Eq. (2). To maintain the above data indicators, a data table needs to be created in the memory controller. Depending on the data range of each metric, the data table requires only 32 bits of physical memory to meet the requirements. Next, the FRFCFS scheduling algorithm processes CPU or GPU requests within a quantum in batches, depending on the priority of GPU, and this step does not require additional hardware overhead. Table 1 shows the hardware cost of PPBP in detail, including the size of four registers and a continuous time counter. The consumption of the algorithm execution includes:

1. In each quantum, the numbers of CPU requests, GPU requests, and GPU memory access threads are counted.

2. At the end of the quantum, the number of GPU single-cycle memory access threads is updated, GPU latency tolerance is predicted, and GPU priority is calculated.

3. FRFCFS processes CPU or GPU access requests in batches in one quantum.

5 Simulations

5.1 Implementation of the simulations

On the gem5 (Binkert et al., 2011) simulator, we built a heterogeneous computing system with eight CPU cores and 16 GPU computing cores to evaluate the performance of the PPBP scheduling strategy. The gem5 simulator is a modular platform for computer architecture research, which encompasses system-level architecture as well as processor microarchitecture. The gem5 simulator supports two distinct system architectures; system simulation (SE) and full system (FS) modes are available. The SE mode can simulate the majority of operating-system-level services, and also provide a good functional simulation speedup. The FS mode can accurately simulate system time and the overhead by simulating a complete system, including the operating system, thread scheduling in user and kernel modes, and various devices.

In the simulations, we evaluated the performance of the PPBP scheduling strategy in various scenarios by running a mixture of CPU and GPU benchmarks in different numbers. The CPU benchmarks were mounted on the specified CPU, where CPUs with the benchmark could access memory and CPUs without the benchmark were idle during the simulations, so they had no effect on other CPUs. Like CPU benchmarks, GPU benchmarks were mounted on the specified CPU, and CPU scheduled GPU to execute parallel computations. We evaluated the performance of PPBP on architectures with eight CPUs and one GPU, and compared PPBP with existing scheduling algorithms: one CPU benchmark and one GPU benchmark, two CPU benchmarks and two GPU benchmarks, and four CPU benchmarks and four GPU benchmarks. To prevent interference between CPU or GPU cores from becoming the main cause of system

Table 1 Additional state (over FRFCFS) required for a possible PPBP implementation

Component	Description/Purpose	Size (number of additional bits)
$\text{numREQ}_{\text{cpu}}$ (register)	Number of CPU requests in the request buffer	$\log_2 \text{numREQ}_{\text{CPU}}$ (6)
$\text{numREQ}_{\text{gpu}}$ (register)	Number of GPU requests in the request buffer	$\log_2 \text{numREQ}_{\text{GPU}}$ (12)
$\text{GPUThread}_{\text{cycle}}$ (register)	Number of GPU memory access threads in a cycle	$\log_2 \text{GPUThread}_{\text{cycle}}$ (8)
$\text{GPUThread}_{\text{quantum}}$ (register)	Number of GPU memory access threads in a span	$\log_2 \text{GPUThread}_{\text{quantum}}$ (8)
Time counter of a quantum (time counter)	Count the execution time and update the range	

performance degradation, a same number of CPU and GPU benchmarks were conducted for each simulation. Also, to ensure that the simulation covered the possibilities as much as possible, we randomly mixed an equal number of CPU and GPU benchmarks and ensured that each CPU benchmark and each GPU benchmark were combined with each other at least once. All simulations were conducted several times and the average performance was obtained. Using the mechanisms described above, we evaluated every possible PPBP execution under the heterogeneous architecture consisting of eight CPUs and one GPU, and guaranteed the accuracy of the simulation data.

We used the FS mode for the simulation to ensure the simulation accuracy, and strictly controlled the execution time of each workload to ensure that the CPU and GPU workloads were executed and terminated at the same time. So, the interference between CPU and GPU applications can be fully observed.

5.2 System configuration

The CPU was an X86 architecture with a two-level private cache, and the GPU was an AMD APU architecture. To fully use the GPU performance, the GPU had only 16 computing units (CUs), and each CU had a private first level cache. The CPU and GPU shared the last level cache and main memory. Table 2 provides the details of our simulated heterogeneous architecture.

Table 2 Configuration of the simulation system

Component	Configuration
CPU	8 cores, 2.3 GHz, X86, out-of-order
GPU	16 computing units, 800 MHz, X86, out-of-order
LLC	512 KB shared, 128-bit line, 8-way, LRU
DRAM	8 GB; channel/rank/bank: 1/2/8; row buffer size: 2 KB

5.3 Workloads

We chose the typical 12 benchmarks in PARSEC3.0 (Zhan et al., 2016) as CPU benchmarks, and 12 benchmarks provided in gem5-resources (Jamieson and Chandrashekar, 2022), such as lulesh, pennant, and DNNMark, as GPU benchmarks (Table 3). PARSEC3.0 is a benchmark suite consisting of multi-thread programs that focus on a new type of workload, covers a variety of domains, and is intended to represent a shared-memory program for the next-generation chip multiprocessors. The GPU benchmark program is an official set of stable and gem5-compatible GPU benchmark programs provided by gem5 that is extremely stable and offers experimental repeatability. Table 3 shows the memory intensities (the number of memory accesses per 1000 instructions) and the row buffer hit rate of each benchmark. As the memory intensity of the benchmark becomes higher, the interference with other cores will spring up more frequently during memory accesses. Therefore, we randomly mixed each workload to ensure that each CPU benchmark and each GPU benchmark were combined with each other once to cover all possible combinations of memory intensities. To reduce the complexity of the algorithm and improve the hardware implementation, we considered only the mutual interference between GPU and CPU in our simulations, so we did not distinguish between memory-intensive and sensitive benchmarks. In total, we used 24 workloads to evaluate the performance of PPBP.

5.4 Metrics

Our quantitative analysis of the PPBP scheduling strategy is presented in terms of CPU-GPU interference and system performance. We used the weighted_speedup metric to evaluate the performance of CPU and GPU, which is shown in Eq. (3). We used a commonly used weighted_speedup metric to measure the system throughput, comparing the performance degradation between executing a single benchmark and executing multiple benchmarks simultaneously. The GPU load was also mounted on the CPU cores in the integrated CPU and GPU heterogeneous architecture. When the GPU code was executed, CPU scheduled GPU to perform parallel computations, so the weighted_speedup was used to measure the GPU performance.

$$\text{weighted\_speedup} = \sum_{i=1}^{n} \frac{\text{IPC}_i^{\text{shared}}}{\text{IPC}_i^{\text{alone}}}. \qquad (3)$$

We used the average_slowdown to measure the level of the interference between CPU and GPU memory requests. The average_slowdown refers to the average performance degradation across all applications. The larger the average deceleration, the greater the interference between the memory requests. The average deceleration is calculated in Eq. (4):

$$\text{average\_slowdown} = \frac{1}{n}\sum_{i=1}^{n}\frac{\text{IPC}_i^{\text{alone}}}{\text{IPC}_i^{\text{shared}}}. \tag{4}$$

6 Evaluation and analysis

We compared the performance and anti-jamming performance of PPBP, FRFCFS, FRFCFS-Cap, and BLISS. The FRFCFS scheduling algorithm uses row buffers in a row buffer hit-first fashion, and old requests take precedence over new requests. Because GPU applications generate a large number of requests to access the same row in the CPU-GPU heterogeneous architecture, FRFCFS unfairly gives GPU applications a higher priority, which seriously affects CPU performance. FRFCFS-Cap is an FRFCFS modification that limits the number of consecutive row buffer hit requests that an application can make. BLISS is a memory request scheduling algorithm in a homogeneous architecture proposed by Subramanian et al. (2015). The algorithm dynamically divides applications into two types, disturbed and undisturbed, and gives disturbed applications a higher priority.

6.1 Performance analysis

Fig. 6 shows the normalized weighted speedup provided by four algorithms on nine representative CPU workloads when four CPU benchmarks and four GPU benchmarks were executed simultaneously. Compared with FRFCFS, PPBP improved the system performance by 14.64% at maximum and 8.53% on average. Compared with the previously implemented BLISS with the best performance, the performance of PPBP was improved by about 5%.

Fig. 7a shows the performance of algorithms

Table 3 Characteristics of CPU and GPU benchmarks running on the baseline

No.	Benchmark		MPKI		Row buffer hit rate (%)	
	CPU	GPU	CPU	GPU	CPU	GPU
1	streamcluster	pennant	16.83	96.35	68.00	75.94
2	blackscholes	allSyncPrims	2.68	123.98	64.97	77.82
3	freqmine	fw_hip	6.47	261.43	68.82	45.54
4	rtview	mis_hip	0.54	58.17	64.48	68.81
5	fluidanimate	sssp_ell	3.31	51.27	60.62	80.29
6	bodytrack	lulesh	3.76	314.77	73.11	84.08
7	x264	bc_hip	5.29	111.17	56.72	65.68
8	swaptions	color_maxmin	1.08	52.27	80.04	60.78
9	canneal	pagerank_spmv	34.02	70.00	38.52	71.90
10	facesim	DNNMark	13.64	143.53	70.10	75.64
11	ferret	ForceTreeTest	1.31	88.57	62.27	69.56
12	vips	square	4.28	63.45	66.52	75.59

MPKI: misses per thousand instructions

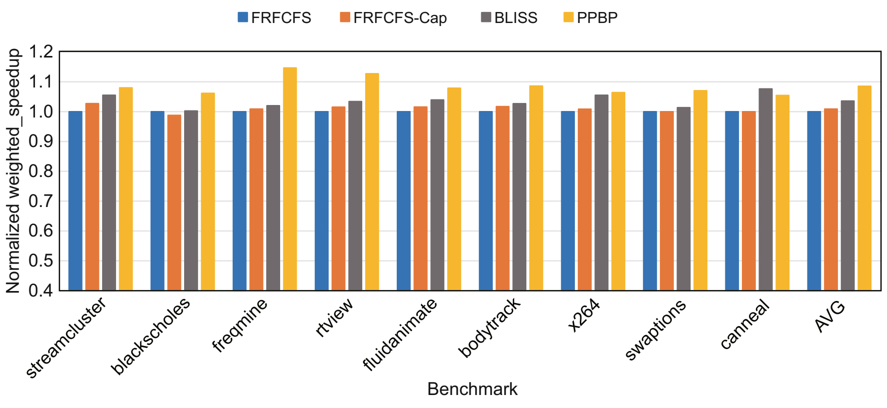

Fig. 6 CPU performance of PPBP and three other algorithms

when they executed one CPU benchmark and one GPU benchmark simultaneously. Fig. 7b shows the performance of algorithms when they executed two CPU benchmarks and two GPU benchmarks simultaneously. Fig. 7c shows the performance of algorithms when they executed four CPU benchmarks and four GPU benchmarks simultaneously. In the case of running a CPU benchmark and a GPU benchmark, PPBP does not show its significant advantage compared to other algorithms. However, as the number of workloads increased, PPBP gradually showed its advantages. In the case of four CPU benchmarks and four GPU benchmarks, the performance of PPBP increased by 8.53% compared with FR-FCFS, and the performance of PPBP was much better than that of BLISS. When more workloads were executed concurrently, the number of threads that accessed memory concurrently increased, whereas BLISS classified these threads, which increased the system latency and hardware complexity. PPBP simply separated memory requests into CPU and GPU memory requests. Its complexity did not increase with an increase in the number of workloads, so its adaptability and stability were strong.

However, on the GPU side, the performance of PPBP was not very good, and it can only guarantee that GPU performance did not decline. There are multiple computation cores in GPU. As the number of workloads increased, the number of cores activated increased, and there was mutual interference between different cores as well. PPBP considers only the interference between CPU and GPU memory requests, and does not consider the interference between memory requests from different GPU cores, which results in poorer GPU performance. In the future, we will focus on the interference between GPU cores, to enable PPBP to further improve system performance.

6.2 Interference between CPU and GPU

Fig. 8 shows the performance of algorithms in dealing with the interference between CPU and GPU memory requests under different numbers of CPU and GPU workloads. Obviously, PPBP was superior to the three other algorithms in handling mutual interference, and this advantage became more obvious as the number of workloads performed increases. Compared with FRFCFS, PPBP reduced the mutual interference between CPU and GPU by 10.38%. At the same time, BLISS demonstrated excellent performance in a CPU homogeneous architecture, but it is unsuitable to be applied to CPU-GPU heterogeneous architectures.

By comparing the above two performance metrics, we conclude that the PPBP scheduling strategy can improve the performance of CPU applications and reduce the interference between CPU and GPU memory requests based on low hardware cost and complexity.

6.3 Analysis of PPBP

Figs. 9a and 9b show the trends of CPU and GPU performances with different quantum sizes when four CPU benchmarks and four GPU benchmarks were executed, respectively. The simulations

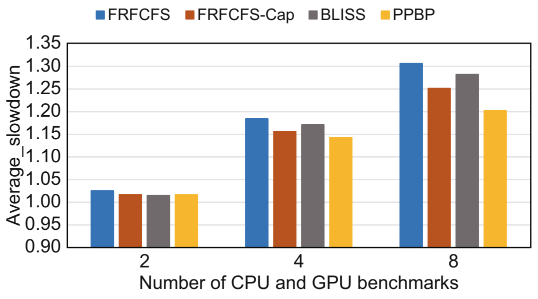

Fig. 8 Average_slowdown at different numbers of CPU and GPU applications

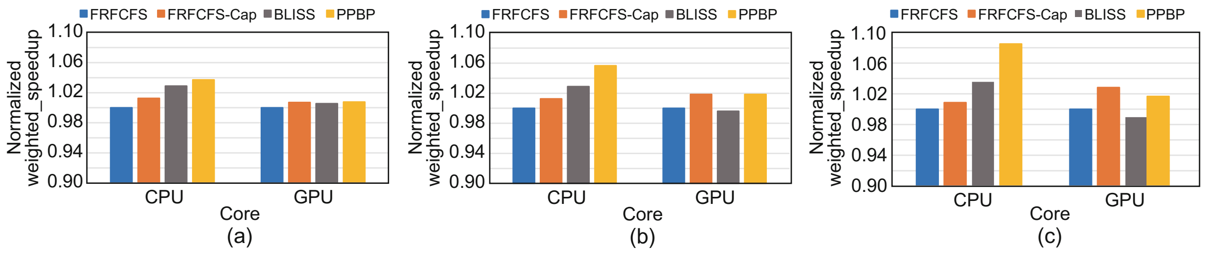

Fig. 7 Performance of PPBP and three other algorithms with one (a), two (b), and four (c) CPU and GPU benchmarks

indicated only how the size of the quantum from 1×10^3 to 1×10^7 affected PPBP. When the quantum size was too small, the number of CPU or GPU requests processed in batches became insufficient. PPBP was equivalent to FRFCFS when the quantum size was infinitely small. As a result, the quantum was too narrow to benefit from PPBP. When the quantum was too wide, the number of CPU or GPU requests processed in batches was too large, and the switching time between CPU and GPU requests was too long, which seriously affected CPU and GPU performance. At the same time, because the number of GPU memory requests was much larger than the number of CPU memory requests, CPU was more severely affected. As shown in Fig. 9, the performance of CPU was the best when the quantum size was 1×10^5, and the performance of GPU was the best when the quantum size was 1×10^7. Because the main purpose of PPBP is to improve CPU performance and reduce the impact of GPU on CPU, and the improvement in GPU performance was small, we set the quantum size to 1×10^5.

Figs. 10a and 10b indicate the trends in CPU and GPU performances with different GPU priority factors, respectively. When the GPU priority factor was smaller, the CPU priority was higher. CPU performance gradually decreased when the GPU priority coefficient was increased. However, in 0.30–0.55, the CPU performance did not change significantly. This is because when the CPU priority is higher, the interference of memory requests between CPU and GPU becomes the most important factor that affects the CPU performance. However, if the GPU priority coefficient is too small, too much time is spent in processing CPU memory requests, and GPU memory requests need to wait for a long time, which seriously affects GPU performance. So, after simulations the best GPU priority coefficient was 0.50.

7 Conclusions

To handle the problem of CPU performance degradation caused by the severe competition of shared memory in the CPU-GPU heterogeneous architecture, a PPBP scheduling strategy is proposed in this paper, which aims to reduce the destructive impact on CPU memory access with low hardware complexity. First, PPBP dynamically estimates the GPU latency tolerance by perceiving the conditions of CPU and GPU memory requests in the request

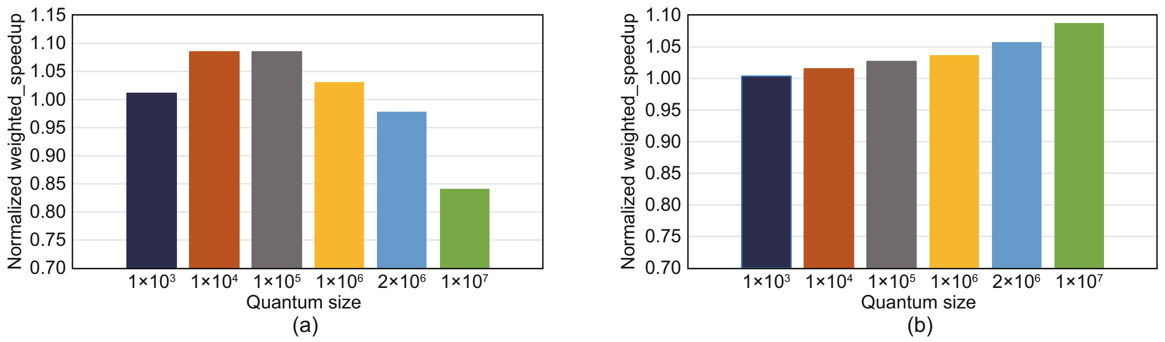

Fig. 9 PPBP performance with varied PPBP quantum sizes: (a) CPU; (b) GPU

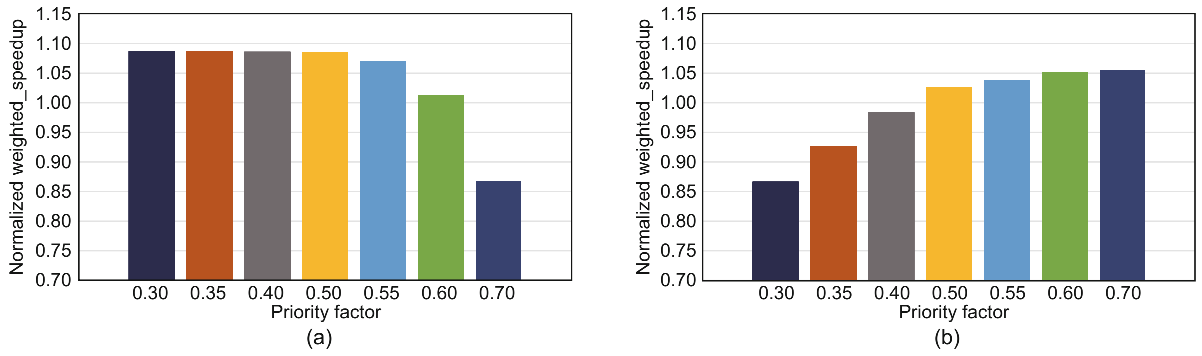

Fig. 10 PPBP performance with different GPU priority factors: (a) CPU; (b) GPU

buffer, to appropriately increase the memory access priority of CPU applications. Then, CPU or GPU memory requests are scheduled and processed in batches based on the FRFCFS scheduling algorithm, to reduce the interference between CPU and GPU. The evaluation results show that PPBP has better performance in mixed execution with different numbers of CPU and GPU benchmarks. Compared with previous memory schedulers, PPBP reduces hardware complexity, improves the possibility of hardware implementation, and provides better performance and fairness for CPU without reducing GPU performance. In the future, we will consider the interference between GPU and CPU cores in PPBP, to further improve system performance.

Contributors

Juan FANG and Sheng LIN designed the research. Sheng LIN and Yixiang XU processed the data. Sheng LIN, Huijing YANG, and Xing SU drafted the paper. Juan FANG and Xing SU helped organize the paper. Sheng LIN and Xing SU revised and finalized the paper.

Compliance with ethics guidelines

Juan FANG, Sheng LIN, Huijing YANG, Yixiang XU, and Xing SU declare that they have no conflict of interest.

Data availability

The data that support the findings of this study are openly available in PARSEC3.0 at https://parsec.cs.princeton.edu/parsec3-doc.htm.

References

Ausavarungnirun R, Chang KKW, Subramanian L, et al., 2012. Staged memory scheduling: achieving high performance and scalability in heterogeneous systems. Proc 39th Annual Int Symp on Computer Architecture, p.416-427. https://doi.org/10.1109/ISCA.2012.6237036

Binkert N, Beckmann B, Black G, et al., 2011. The gem5 simulator. *ACM SIGARCH Comput Archit News*, 39(2):1-7. https://doi.org/10.1145/2024716.2024718

Bitalebi H, Safaei F, 2023. Criticality-aware priority to accelerate GPU memory access. *J Supercomput*, 79(1):188-213. https://doi.org/10.1007/s11227-022-04657-3

Bouvier D, Cohen B, Fry W, et al., 2014. Kabini: an AMD accelerated processing unit system on a chip. *IEEE Micro*, 34(2):22-33. https://doi.org/10.1109/MM.2014.3

Chen W, Ray S, Bhadra J, et al., 2017. Challenges and trends in modern SoC design verification. *IEEE Des Test*, 34(5):7-22. https://doi.org/10.1109/MDAT.2017.2735383

di Sanzo P, Pellegrini A, Sannicandro M, et al., 2020. Adaptive model-based scheduling in software transactional memory. *IEEE Trans Comput*, 69(5):621-632. https://doi.org/10.1109/TC.2019.2954139

Fang J, Yu L, Liu ST, et al., 2015. KL_GA: an application mapping algorithm for mesh-of-tree (MoT) architecture in network-on-chip design. *J Supercomput*, 71(11):4056-4071. https://doi.org/10.1007/s11227-015-1504-y

Fang J, Wang MX, Wei ZL, 2020. A memory scheduling strategy for eliminating memory access interference in heterogeneous system. *J Supercomput*, 76(4):3129-3154. https://doi.org/10.1007/s11227-019-03135-7

Hazarika A, Poddar S, Rahaman H, 2020. Survey on memory management techniques in heterogeneous computing systems. *IET Comput Dig Tech*, 14(2):47-60. https://doi.org/10.1049/iet-cdt.2019.0092

Jamieson C, Chandrashekar A, 2022. gem5 GPU accuracy profiler (GAP). Proc 4th gem5 Users Workshop, p.44.

Jeong MK, Erez M, Sudanthi C, et al., 2012. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC. Proc Design Automation Conf, p.850-855.

Jog A, Kayiran O, Nachiappan NC, et al., 2013. OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance. *ACM SIGPLAN Not*, 48(4):395-406. https://doi.org/10.1145/2499368.2451158

Jog A, Kayiran O, Pattnaik A, et al., 2016. Exploiting core criticality for enhanced GPU performance. Proc ACM SIGMETRICS Int Conf on Measurement and Modeling of Computer Science, p.351-363. https://doi.org/10.1145/2896377.2901468

Kim Y, Han D, Mutlu O, et al., 2010. ATLAS: a scalable and high-performance scheduling algorithm for multiple memory controllers. Proc 16th Int Symp on High-Performance Computer Architecture, p.1-12. https://doi.org/10.1109/HPCA.2010.5416658

Lin CH, Liu JC, Yang PK, 2020. Performance enhancement of GPU parallel computing using memory allocation optimization. Proc 14th Int Conf on Ubiquitous Information Management and Communication, p.1-5. https://doi.org/10.1109/IMCOM48794.2020.9001771

Mittal S, Vetter JS, 2015. A survey of CPU-GPU heterogeneous computing techniques. *ACM Comput Surv*, 47(4):69. https://doi.org/10.1145/2788396

Mutlu O, Moscibroda T, 2008. Parallelism-aware batch scheduling: enhancing both performance and fairness of shared DRAM systems. Proc Int Symp on Computer Architecture, p.63-74. https://doi.org/10.1109/ISCA.2008.7

Power J, Basu A, Gu JL, et al., 2013. Heterogeneous system coherence for integrated CPU-GPU systems. Proc 46th Annual IEEE/ACM Int Symp on Microarchitecture, p.457-467. https://doi.org/10.1145/2540708.2540747

Rai S, Chaudhuri M, 2017. Using criticality of GPU accesses in memory management for CPU-GPU heterogeneous multi-core processors. *ACM Trans Embed Comput Syst*, 16(5s):133. https://doi.org/10.1145/3126540

Subramanian L, Lee D, Seshadri V, et al., 2015. The blacklisting memory scheduler: balancing performance, fairness and complexity. https://arxiv.org/abs/1504.00390v1

Usui H, Subramanian L, Chang KKW, et al., 2016. DASH: deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators. *ACM Trans Archit Code Optim*, 12(4):65. https://doi.org/10.1145/2847255

Wang HN, Jog A, 2019. Exploiting latency and error tolerance of GPGPU applications for an energy-efficient DRAM. Proc 49th Annual IEEE/IFIP Int Conf on Dependable Systems and Networks, p.362-374. https://doi.org/10.1109/DSN.2019.00046

Wang QH, Peng Z, Ren B, et al., 2022. MemHC: an optimized GPU memory management framework for accelerating many-body correlation. *ACM Trans Archit Code Optim*, 19(2):24. https://doi.org/10.1145/3506705

Zhan XS, Bao YG, Bienia C, et al., 2016. PARSEC3.0: a multicore benchmark suite with network stacks and SPLASH-2X. *ACM SIGARCH Comput Archit News*, 44(5):1-16. https://doi.org/10.1145/3053277.3053279

Zhang F, Zhai JD, He BS, et al., 2017. Understanding co-running behaviors on integrated CPU/GPU architectures. *IEEE Trans Parall Distrib Syst*, 28(3):905-918. https://doi.org/10.1109/TPDS.2016.2586074