The Journal of Supercomputing (2020) 76:3129–3154
https://doi.org/10.1007/s11227-019-03135-7

A memory scheduling strategy for eliminating memory access interference in heterogeneous system

Juan Fang[1] · Mengxuan Wang[1] · Zelin Wei[1]

Published online: 10 January 2020

Abstract
Multiple CPUs and GPUs are integrated on the same chip to share memory, and access requests between cores are interfering with each other. Memory requests from the GPU seriously interfere with the CPU memory access performance. Requests between multiple CPUs are intertwined when accessing memory, and its performance is greatly affected. The difference in access latency between GPU cores increases the average latency of memory accesses. In order to solve the problems encountered in the shared memory of heterogeneous multi-core systems, we propose a step-by-step memory scheduling strategy, which improve the system performance. The step-by-step memory scheduling strategy first creates a new memory request queue based on the request source and isolates the CPU requests from the GPU requests when the memory controller receives the memory request, thereby preventing the GPU request from interfering with the CPU request. Then, for the CPU request queue, a dynamic bank partitioning strategy is implemented, which dynamically maps it to different bank sets according to different memory characteristics of the application, and eliminates memory request interference of multiple CPU applications without affecting bank-level parallelism. Finally, for the GPU request queue, the criticality is introduced to measure the difference of the memory access latency between the cores. Based on the first ready-first come first served strategy, we implemented criticality-aware memory scheduling to balance the locality and criticality of application access.

Keywords Heterogeneous multi-core · Shared memory · Memory scheduling

✉ Juan Fang
fangjuan@bjut.edu.cn

Mengxuan Wang
mengxuanw13@emails.bjut.edu.cn

[1] Faculty of Information Technology, Beijing University of Technology, Beijing 100124, China

1 Introduction

Increasing computing demand has caused more and more attention to heterogeneous computing in recent years. Although a GPU-accelerated system consisting of an independent CPU and an independent GPU can perform computational acceleration well for large loads, data transmission between the CPU and GPU is slow, and the advantages of GPU computing require expert-level programming to take advantage. In order to further leverage the computing power of GPUs, the introduction of GPUs on the same chip with multiple CPU cores, known as a heterogeneous multi-core architecture, has attracted researchers' attention [1]. When the core on the same processor shares processor resources (cache, memory, on-chip network resources [2], etc.), multiple applications running in parallel may interfere with each other. Memory resources are the most critical shared resources. As the number of cores increases, the memory access requests from different cores are intertwined, interfere with each other, compete for limited memory resources, and affect application and system performance.

Existing multi-core memory scheduling usually improves the system throughput and fairness by arbitrating memory accesses and prioritizing memory accesses [3–5]. The key to implementing an arbiter is to determine the priority of memory requests by analyzing application characteristics and memory access behavior. In addition, some researchers proposed to eliminate bank interference by modifying the address mapping scheme [6, 7], where bank is the smallest division of data read and written in parallel in the memory structure.

By using the CPU + GPU heterogeneous multi-core architecture, the competition for memory resources becomes more serious due to the completely different characteristics of the CPU and GPU. The starting point of GPU design is that GPUs are more suitable for compute-intensive and highly parallel applications. The CPU is sensitive to memory latency, and its memory requests need to be provided in a short amount of time. The GPU executes multiple threads by concurrently using a single instruction multiple data pipeline. A thread within a warp (GPU application execution scheduling unit) executes the same instruction with different data resources. When the executing warp memory instruction stalls, GPU hides the latency by switching to another warp to continue execution. Relative to CPU, each warp in GPU may have a large number of unfinished memory requests, and GPU memory requests will occupy a large portion of the request buffer. The existing multi-core memory scheduling determines the priority by analyzing the memory request in the request buffer. Due to the competition of GPU requests, there is less CPU memory in the request buffer. Previous memory scheduling strategies analyzed the memory access behavior of CPU applications based on the memory requests in the buffer. GPU requests affect the effectiveness of the memory strategy, which affects memory performance.

In addition, GPU-only memory access policies are often implicitly assumed to be equally important for memory requests from different cores. The GPU is insensitive to latency but sensitive to memory bandwidth. The existing access strategy focuses on improving bandwidth utilization and maximizing memory data throughput [8–10]. In fact, when executing a CUDA (Compute Unified Device Architecture) application, different cores have different numbers of stalled warps and different tolerances for delays. In particular, when the core contains more warps waiting for data to be returned from DRAM, it is unlikely to tolerate unfinished memory request delays, which means that these core requests are more important than others.

For the interference between memory access and different latency tolerance of the GPU core, we propose a step-by-step memory access strategy. The key idea is to take into account the problems encountered in memory access in the CPU + GPU heterogeneous architecture and solve them step by step. The strategy consists of three steps: (1) the interference of the access request between CPU and GPU. GPU memory requests occupy most of the memory request buffer, limiting the visibility of CPU application memory behavior. Create two memory request queues in the MC. When receiving memory requests, we put the access requests into different queues to isolate CPU access requests and GPU access requests according to the memory request source; (2) interference with access requests between multiple CPU cores. By analyzing the application's memory access behavior, including access intensity and row buffer hit ratio, we classify applications into different classes based on the application access behavior. We limited different classes of application access to different bank to eliminate interference from memory requests when multiple applications are executing in parallel; (3) different latency tolerances between GPU cores. We introduced criticality to represent the core latency tolerance, which is highly critical to cores that are unacceptable for latency, giving high priority to critical requests. The step-by-step memory access strategy can reduce the interference of GPU access requests on CPU access requests and bank conflicts, and at the same time improve the bank-level parallelism, and memory access latency differences between GPU cores. The main contributions of our work are the following:

- This paper introduced the challenges encountered by the memory access scheduling in heterogeneous multi-core systems due to the introduction of GPU. A large number of CPU memory requests limit the visibility of existing scheduling algorithms to CPU application access behavior and the difference in memory access latency among GPU cores.
- We proposed a step-by-step memory scheduling strategy, which consists of three parts and is progressive. First, we simply isolate CPU requests from GPU requests based on the memory request source in MC. For the CPU request, we combined with the different needs of each application for the number of banks according to the characteristics of multiple application access requests executed in parallel and develop a dynamic appropriate bank partitioning rule for them. For GPU requests, we introduced criticality to measure core latency tolerance and reduce access request latency based on criticality, which is used to determine the priority of the request.
- In the CPU + GPU heterogeneous system built by the gem5-gpu [11], we evaluated the memory access scheduling strategy and experimental results showing that the step-by-step memory scheduling strategy improves system performance.

The rest of this paper is organized as follows: Sect. 2 introduces the DRAM structure, which is used to analyze GPU and CPU access behavior and GPU core latency differences. Section 3 discusses modern memory scheduling related work. Section 4 describes the challenges encountered in heterogeneous multi-core memory systems. Section 5 introduces the dynamic bank partitioning strategy. Section 6 introduces core criticality and proposes a criticality-aware scheduling strategy. Section 7 analyzes the experimental results. Finally, Sect. 8 contains experimental conclusions and ideas for future work.

2 Background and motivation

2.1 Memory organization

DRAM usually contains one or more memory channels, and multiple different channels are executed in parallel. Each channel has an independent address, data, and instruction bus. Each memory channel contains one or more ranks, and all ranks in the same channel share resources. The MC can select a corresponding rank according to the chip select signal to give a corresponding response to the memory access command. Each rank contains multiple banks, and all banks in a rank share the command bus and address bus, reading and writing data in parallel at the same time. A bank is a two-dimensional structure containing rows and columns, as shown in Fig. 1. Generally, the line that selects the number of rows in the horizontal direction becomes the row, and the line that selects the transmission signal in the vertical direction becomes the column. Each bank has a row buffer, and when reading or writing data, it is required to put the contents of the row into the row buffer

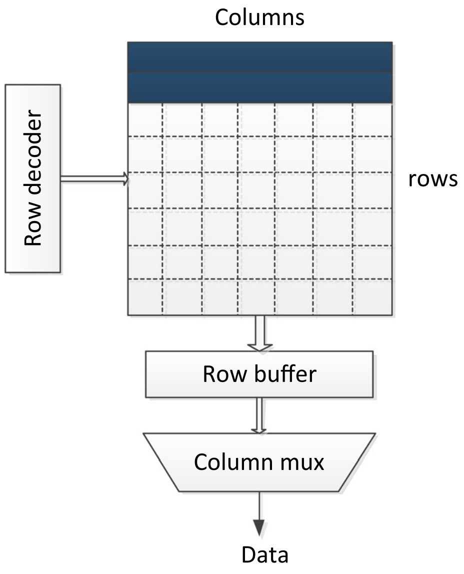

Fig. 1 Bank structure

for temporary storage. The row buffer is required because the behavior of reading or writing rows directly destroys the contents of the row. After the memory access request reaches the MC, the MC parses the physical address into a corresponding DRAM address including the channel id, rank id, bank id, row id, column id according to a certain address mapping rule. The data to be accessed are at a specific location in the bank. Figure 2 shows the mapping rules in the Intel Xeon server [12], which is a static address mapping that resolves a 32-bit physical address into a DRAM address. Dynamic address mapping is commonly used in modern DRAM.

The latency caused by a basic DRAM operation includes the transfer time of the memory request to the MC, the MC latency, and the DRAM bank operation latency. MC latency includes queuing, scheduling latency, and the latency of converting memory accesses into basic command. DRAM bank latency is determined by the row buffer state. When the buffer is open, the latency only includes column access operation latency (CAS), and when the row buffer is idle, the delay includes column access operation latency (CAS) and row activation operation latency (RAS). In the worst case, the row buffer is occupied, needs to be cleared by a precharge operation, latency includes precharge operation latency, row activation latency, and column access latency.

Column access: Read and write directly to the data in the target row in the row cache based on the column address.

Row activation: Activates the target row in the data area based on the row address and writes the entire row of the target row to the row cache. The row buffer must be guaranteed to be idle before writing data; otherwise, it must be precharged.

Precharge: Write the data in the line buffer back to the data area of the bank and clear the line buffer when the line buffer is idle.

2.2 CPU and GPU memory access behavior analysis

CPU and GPU applications have very different access characteristics [13]. GPU applications contain a large number of parallel threads. By switching the warp to hide the latency caused by memory stalls, GPU applications have more memory requests than CPU. The experiment specifically shows the difference in memory access behaviors between CPU and GPU applications through three metrics, namely memory intensity, row buffer hit ratio, and bank-level parallelism. Select CPU application from the SPEC CPU2006 benchmark [14] and select GPU application from the Rodinia benchmark [15].

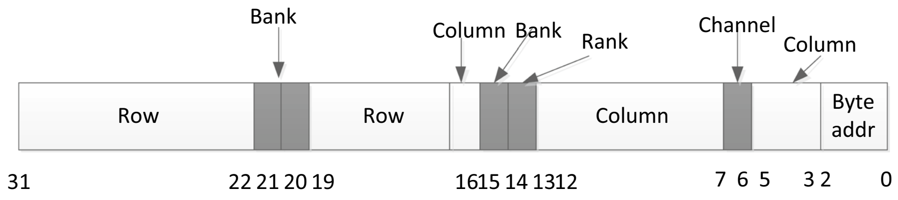

Fig. 2 Intel xeon5645 address mapping

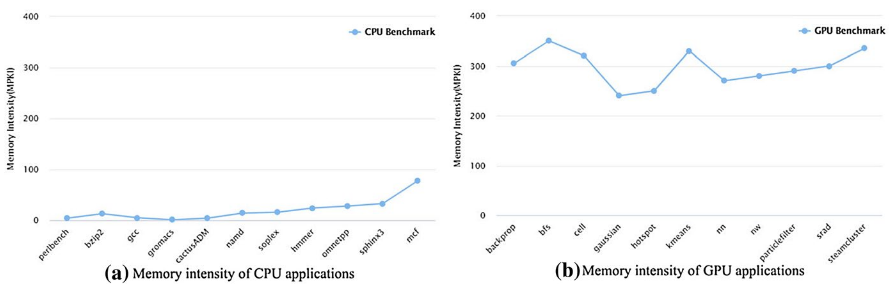

Fig. 3 CPU and GPU application memory intensity comparison

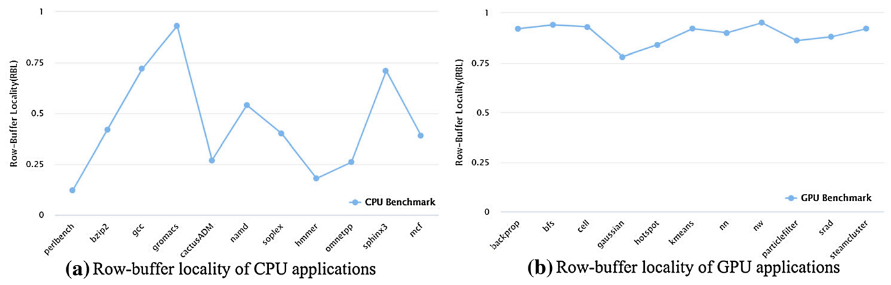

Fig. 4 CPU and GPU application row buffer hit ratio comparison

Figure 3 shows the comparison of memory intensity, defined as the number of memory requests per thousand instructions (MPKI). It is apparent from the figure that the memory intensity of the GPU application is much higher than that of the CPU application. Figure 4 shows the comparison of row buffer hit ratio. The row buffer hit ratio directly reflects the memory locality of the application. The figure shows that the GPU application row buffer hit ratio is always high, while that of the CPU is quite different. GPU applications have a high level of spatial locality and are typically access patterns associated with sequential memory access. Figure 5 shows the bank-level parallelism, which refers to the parallel execution of multiple access requests for different banks. Pipeline execution of multiple access requests can hide the latency caused by serial execution. As shown in Fig. 5, the GPU application bank-level parallelism is slightly higher than the CPU application. Compared with the low memory intensity and low row buffer hit ratio of CPU applications, GPU applications usually show high memory intensity and high row buffer hit ratio. When the CPU and GPU compete for shared memory resources, GPU applications will inevitably interfere with CPU applications. In the heterogeneous multi-core system built by gem5-gpu, the default memory access scheduling policy determines the priority based on the row buffer hit ratio, which seriously affects the memory access of the CPU application.

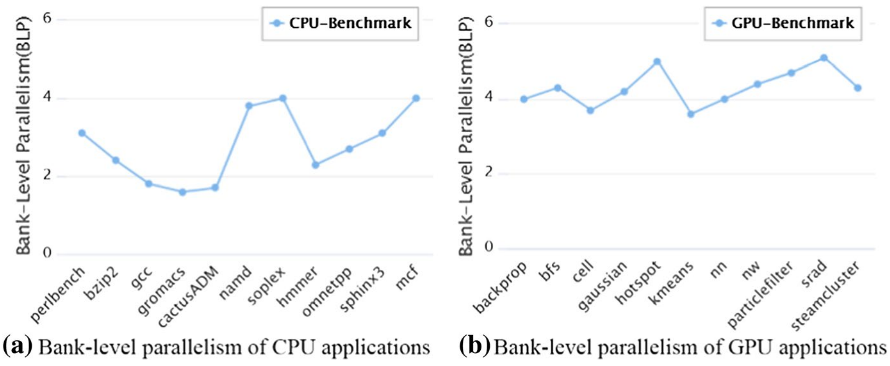

Fig. 5 CPU and GPU application bank-level parallelism comparison

2.3 Memory access latency differences between GPU cores

Adwait [16] is the first to propose a latency difference between GPU cores and improve the GPU memory scheduling strategy according to the core latency difference. Previous GPU access scheduling implicitly assumed that all requests from different cores were equally important, with the goal of increasing bandwidth utilization and maximizing system throughput. The GPU focuses on improving the overall performance of multiple concurrent execution threads by overlapping execution, rather than minimizing the latency of a particular request or core. Adwait observed that when a CUDA application is executed, the memory requests of different cores compete for shared memory, regardless of the latency difference between cores, resulting in uneven resource allocation. Different cores have different tolerances for latency. In particular, if there are more warps in the core that are waiting for data to be returned from DRAM, it is unlikely to tolerate unfinished memory request latency. These core requests are more important than others.

2.4 Motivation

The integration of multi-core CPU and GPU on the same chip is because while using the powerful computing power of the GPU, the impact of data communication has to be considered. However, the problems caused by integrating the CPU and GPU on the same chip cannot be ignored, especially the competition for memory resources. As analyzed in Sect. 2.2, there is a big difference between the memory access behavior of the CPU and GPU. This asymmetric memory behavior seriously affects the effect of the memory scheduling strategy applied by traditional multi-core CPU systems. When the GPU is released on the same chip with a CPU core, in order to maximize the computing power of the GPU, research on GPU memory request scheduling is also urgent. In Sect. 2.3, we discussed that latency differences between GPU cores can also affect system performance. Based on this, we have designed a new CPU-GPU heterogeneous memory scheduler.

3 Related work

3.1 Memory partitioning

Muralidhara et al. [17] proposed an application-based memory channel partitioning (MCP). According to the memory access behavior of different threads, different memory channels are allocated for each application, and the memory access interference between threads is reduced. Relatively speaking, the MCP partition granularity is too large. Since the growth rate of the number of threads in the system is much higher than the increase of the number of memory channels, there will still be a situation of shared memory channels, and the interference problem of the memory access requests between the threads still exists. Some scholars [18, 19] proposed to use the bank as the granularity, the memory access requests of different threads are mapped to different banks, and the interference between multiple memory access requests is truly eliminated. Our work uses a dynamic bank partitioning strategy to develop the best bank partitioning rules for each thread, taking into account the locality and parallelism of the memory access request.

3.2 Multi-core memory scheduling

Mutlu et al. [20] proposed a parallel scheduling of memory access requests based on parallelism (PAR-BS). PAR-BS mainly consists of two steps, batch scheduling and memory access request parallelism. Batch scheduling refers to grouping memory access requests according to the order in which the memory access requests arrive. Parallelism refers to the parallel execution of access requests that access different rows of different banks. The size of the group directly affects system performance. PAR-BS does not consider the access requirements of different threads, and the effect of improvement is limited.

3.3 GPU memory scheduling

Wang [21] proposed a GPU DRAM scheduling program that utilizes the last level cached inter-core location information detected in the MSHR. The main reason for the inter-core location is that multiple cores access shared read-only data in the same cache line, and more threads resume execution by prioritizing memory requests at high core locations. Our work leverages the core criticality concepts presented in [16] to describe the difference in latency between cores, reducing core latency without affecting access locality.

3.4 Heterogeneous multi-core memory scheduling

Most of the scheduling algorithms analyze the request flow in the request buffer and analyze the application characteristics to determine the core priority [22]. Rachata [23] proposes that the GPU request seriously interferes with the CPU request, because the GPU request occupies the request buffer. Most of the space limits the analysis of memory access behavior for CPU applications. In order to solve this problem, we need to isolate CPU and GPU requests.

4 Memory scheduling challenge

4.1 GPU interference with CPU memory requests

As discussed in Sect. 2.2, GPU applications always maintain high memory intensity and high row buffer hit ratios. The high memory access intensity shown by the GPU makes GPU memory requests in the row buffer much more than CPU memory requests. In the multi-core architecture, the memory scheduling policy is directed to scenarios where there are only CPU memory requests in the row buffer. Usually, the memory request scheduling policy determines the memory characteristics of the application by analyzing the request flow of the request buffer, thereby determining the priority of the core or memory request. Figure 6a shows that in a multi-core architecture, the row buffer contains only CPU access requests, and the MC analyzes the information of these requests to determine the memory behavior of the application. Figure 6b shows that under the CPU + GPU heterogeneous architecture, due to the memory access density of the GPU application, a large number of requests from the GPU occupy most of the MC request buffer, and there are few CPU access requests. The CPU access request provides insufficient information to analyze the memory behavior of the CPU application by the MC. This is one of the specific manifestations of GPU interference with CPU memory access. On the other hand, due to the high row buffer hit ratio of the GPU, according to the locality, the memory scheduling policy with the priority of the row buffer hit rate prioritizes the GPU request and also interferes with the CPU memory access. In order to solve this problem, we simply created two request memory queues in the MC. According to the request source, we divide it into CPU memory requests and GPU memory requests and place them in two different request queues to isolate CPU and GPU memory

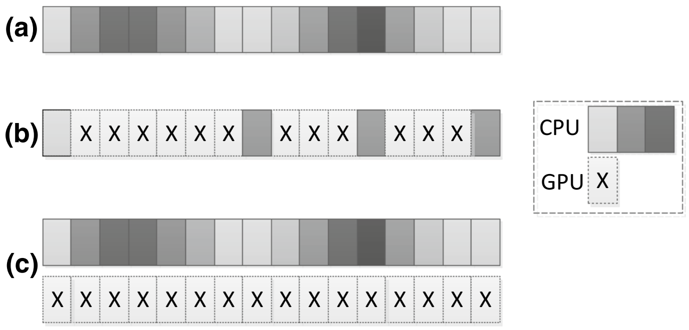

Fig. 6 **a** CPU-only case, **b** MC's visibility when CPU and GPU are combined, **c** Isolate CPU and GPU memory requests

requests, as shown in Fig. 6c. For CPU request queue and GPU request queue, we use different memory scheduling strategies, which are dynamic bank partitioning in Sect. 5 and core critical scheduling in Sect. 6.

At runtime, the current processing queue needs to be selected. Our principle is to prioritize CPU memory requests while ensuring that GPU applications can still make reasonable progress. Therefore, we use a simple method to decide whether the memory request is currently selected from the CPU queue or the GPU queue for processing. When both queues have memory requests, we determine based on a configurable threshold *T–P* and a randomly generated probability *rp* within the period. If *rp* is less than *T–P*, then select the CPU queue, otherwise select the GPU queue. To ensure that the CPU request has a high priority, the threshold *T–P* selection is important. After the memory request is selected, different scheduling policies are adopted according to the request type.

4.2 Memory access request interference between CPU cores

Even if the CPU and GPU memory requests are isolated, the memory access requests between multiple CPU cores still interfere with each other. Memory systems mainly rely on locality principles to increase bandwidth and reduce access latency. When multiple CPU cores share memory, memory access requests from different cores are intertwined and interfere with each other to compete for limited memory resources. Request interference seriously destroys the characteristics of the original memory access request, making the randomness of the memory access request increase. Therefore, it is difficult to optimize the memory system by using the principle of locality of memory access requests or improving the parallelism of memory access requests. The locality of *lbm* is very high, and the row buffer hit rate is as high as 98% when running alone. When 4 *lbm* is executed in parallel, the row buffer hit rate is reduced to 50%, which verifies that the memory access interference between multiple threads destroys the locality of the application. Figure 7 is an example of a memory access request interference between multiple cores. When a single thread is executed in Fig. 7a, because the two memory accesses are in the same row, the second memory access can directly perform column access operations. In Fig. 7b, there are memory access interferences from other threads. Memory access requests that are not in the same row need to be precharged, row activated, and column accessed to read and write data. The lack of locality leads to an increase in latency and in system power consumption that affects system performance.

The primary problem with memory scheduling is how to effectively eliminate interference from access requests between applications and threads. The use of resource partitioning to eliminate access interference is an effective solution. By dividing the granularity of bank, the memory access requests between different cores are isolated by mapping the memory access requests of different cores to different banks, and the original memory access request characteristics of each core are retained to the greatest extent. The problem with keeping the local isolation bank is to reduce the bank-level parallelism of the thread or application. In this experiment, when scheduling queue requests for CPU memory, balancing locality and bank-level

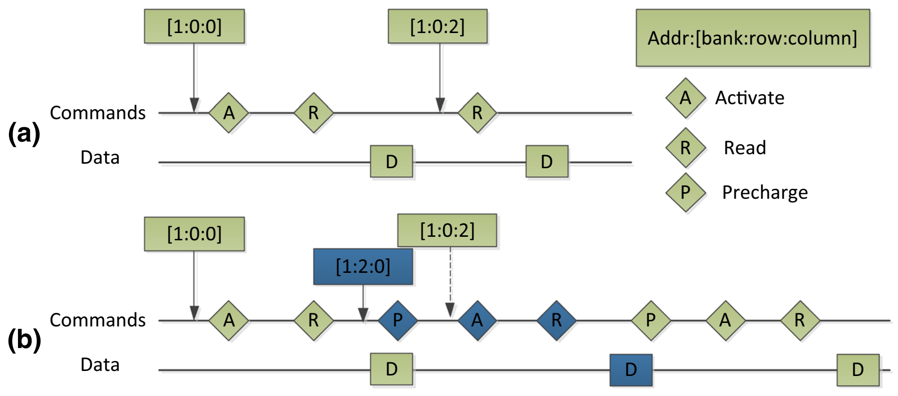

Fig. 7 Example of a memory access request interference between multiple cores

parallelism are problems that needs to be solved by using a dynamic bank partitioning strategy.

4.3 Latency difference between GPU cores

In Sect. 2.3, there is a difference in memory access latency between GPU cores. When there is a large amount of stalling warp in a core, it will take more time to wait for data to be returned from memory due to the lack of warp switching to hide the memory latency. FR-FCFS [10] prioritizes locally, does not distinguish between memory request sources, treating different core requests equally, so that the problems caused by latency differences between cores cannot be solved. When considering the difference in latency between cores, a simple idea is that the more stalling warp exists in a core, the higher the core memory latency will be, and we give the core higher priority. This idea breaks the access locality of the application when considering the latency difference. How to consider the access locality and core latency difference of an application at the same time is an urgent problem to be coped with.

5 Dynamic bank partitioning strategy

Bank partitioning refers to limiting the access of different applications to different banks and isolating memory access between multiple cores. In order to satisfy the bank-level parallelism of the application while ensuring the locality of the program, for the CPU request queue, we implemented a dynamic bank partitioning strategy.

According to the memory behavior analysis of the application in Sect. 2.2, different CPU applications have different memory intensity, row buffer hit ratio, and bank-level parallelism. Regardless of the application memory characteristics, each application is divided into equal banks, and the memory partition cannot be used. Applications with high memory access are most affected by the performance of the memory system. Applications with fewer access requests also have less memory bandwidth and do not interfere too much with other applications with high memory access intensity. The experiment focused on workloads with high memory access intensity.

For access-intensive applications, we considered row buffer hit ratios. Applications with high buffer hit ratios have the greatest impact on receiving row buffer local interference, which means that they benefit the most from bank partitioning. For applications with high memory intensity and high row buffer hit ratios, an exclusive bank is allocated to isolate interference. On the other hand, applications with low row buffer hit ratios rely on bank-level parallelism, allowing such applications to share multiple banks, stream access to different banks, and reduce access latency for serial execution. According to the two key characteristics of application memory intensity and row buffer hit ratio, different partitioning rules are dynamically applied for different applications, and the locality of the memory and the parallelism of the bank level is balanced.

5.1 Dynamic bank partition strategy design

The dynamic bank partitioning strategy is a cyclical dynamic strategy. During an interval, the application access behavior information during the interval is first collected, including the row buffer hit ratio and the number of memory accesses. At the beginning of the next period, the application is grouped according to the memory access information collected in the previous period, and different bank partitioning rules are allocated for each type of application according to the grouping situation, and the execution is continued according to the partitioning rule in this period, as shown in Fig. 8.

The feasibility of setting the bank partitioning rule in the next period is the phase of the application accessing behavior based on the application access behavior information in the previous. Although the memory access of the application is difficult to predict, experimental analysis shows that many programs are executed as phases, and the memory behavior of each phase may be very different from other phases, but in one phase, the memory behavior is similar. Figure 9 selects the similarity of the four CPU application verification phases. The partitioning rules are discussed in

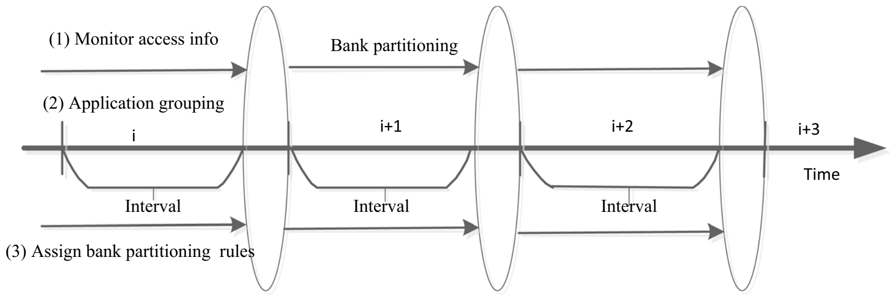

Fig. 8 Dynamic bank partition periodic process

Sect. 5.1.2 and are divided into three categories. According to the application memory access behavior in the previous period, it is effective to set the bank partitioning rule in the next period, and then, it will be wrong when the program classification changes. In particular, this error will be corrected in the next period.

5.1.1 Monitoring application access behavior

Firstly, differentiate the request source and add a core id field for each memory request. When the core issues a memory request, the memory request id field is assigned and packaged and passed to the MC according to the core. The MC sets two counters for each bank, a memory access counter and a row address counter, which are, respectively, recorded in an interval, the number of memory access requests received by each bank, and the row address of the last memory access. Thus, the number of accesses and the hit ratio of each application during this period are calculated.

5.1.2 Application grouping

Group applications are based on collected application access counts and row hit ratio information. We set two thresholds, access intensity threshold (T-AI) and row hit ratio threshold (T-RH). First, according to the T-AI, the application is divided into two types: memory intensive and non-intensive. For access non-intensive applications, since there is little memory access for such applications, we do not need to consider their hit ratio. For memory-intensive applications, it is divided into two categories according to the T-RH: high hit ratio and low hit ratio. Finally, the CPU applications that are executed in parallel are divided into three groups, that is, the memory is non-intensive(NI), the memory access is intensive and hit rate is low(ILH), the access intensive and the hit rate is high (IHH). The thresholds for the access intensity and the row hit ratio are all set according to a large number of experimental results (Table 1).

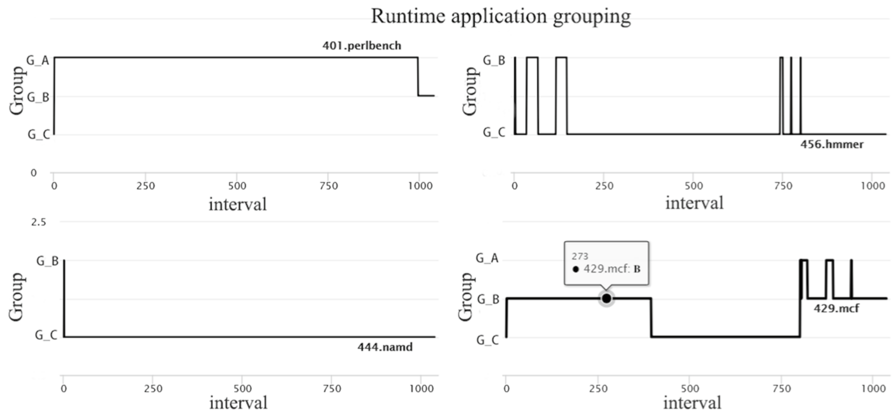

Fig. 9 Runtime program grouping

5.1.3 Bank partition rules

The bank partitioning rules for each type of application vary according to the access characteristics of each group. Equation 1 defines a bank partition unit, BPU. In the memory hierarchy in Eq. 1, the system has N_{channel} channels, each channel has N_{rank} ranks, and each rank has N_{bank} memory banks in the memory system, and $N_{\text{Intensive}}$ represents the number of applications that access memory intensive.

$$\text{BPU} = \frac{N_{\text{channel}} * N_{\text{rank}} * N_{\text{bank}}}{N_{\text{Intensive}}} \tag{1}$$

The partitioning rules are reflected in the different BPUs assigned by different applications.

1. Access non-intensive (NI), since this type of application has fewer access requests, it will not interfere with other applications. At the same time, the number of banks is limited, so that such applications are not processed, and all banks in the memory can be directly accessed.
2. The access intensive and the hit rate is low (ILH), and every two applications in the group share 2 BPU. Although there are many application accesses in this group, the row buffer conflict ratio is still high. The memory access latency causes increasing memory access requests to be blocked in the memory request buffer, and even the memory access request is starved. For this type of application, if multiple applications access different banks, they can be executed in a pipeline, making full use of bank-level parallelism to reduce the delay caused by serial execution access. In this experiment, every two applications in the group share 2 BPU.
3. The access intensive and the hit ratio (IHH), and each application in the group is assigned a BPU. The mutual interference between the applications in the group

Table 1 The rule of application grouping

```
Definition: T-AI and T-RH are two thresholds        N: the number of applications
Begin
    for (i = 1 to N)
        if intensity_i < T-AI
            Applications i is assigned to the non-intensive (NI)
        else
            if hitRatio_i < T-RH
                Application i is assigned to low hit ratio (ILH)
            else
                Application i is assigned to high hit ratio (IHH)
End
```

is serious, and the locality of the single application is also the highest. Therefore, each application in the group is allocated a BPU to isolate the memory access request between application requests, taking advantage of memory locality to improve memory power consumption and access latency.

As shown in Fig. 10, the system channel = 1, rank = 2, bank = 8, and 4 applications are executed in parallel, where bench0 and bench1 are ILH, bench2 is IHH, and bench3 is NI. N(channel) = 1, N(rank) = 2, N(bank) = 8, N(intensive) = 3, BPU = 5. Bench2 has one BPU, bench1 and bench2 share two BPUs, and bench3 is unconstrained and is able to access all banks.

5.2 Workload

The bank partitioning strategy proposed by the experiment is effective for memory-intensive applications and does not specifically deal with non-intensive applications. Obviously, when multiple applications are executed in parallel, the more non-intensive applications are accessed, the less obvious the experimental results are. To implement this, we need to choose the right application to verify the policy. We chose CPU application from SPEC CPU 2006 [14], and according to the memory intensity, it is classified into two types: memory access intensity and memory access non-intensive. When the memory intensity is higher than 4, the memory access is dense; otherwise, the memory access is non-intensive. Table 2 shows the CPU benchmark classification, Class A indicates that the memory is intensive, and Class B indicates that the memory is non-intensive.

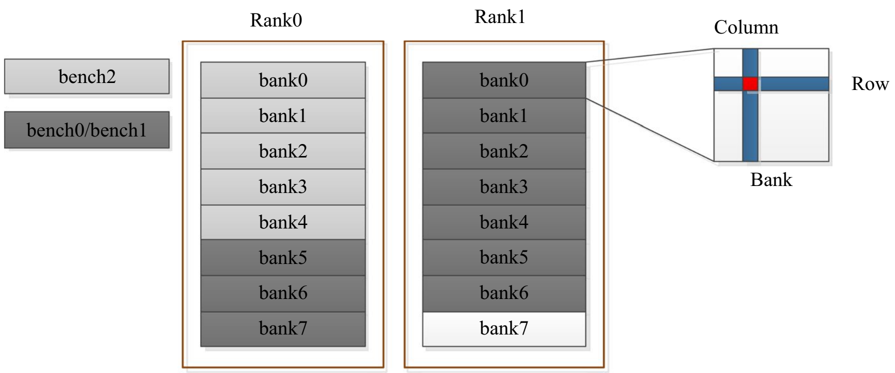

Fig. 10 Bank partition example

Table 2 CPU benchmark classification

Application	MPKI	Class	Application	MPKI	Class
400.perlbench	0.08	B	450.soplex	13.21	A
401.bzip2	4.10	A	454.calculix	0.04	B
410.bwaves	5.21	A	456.hmmer	2.82	B
416.games	0.12	B	458.sjeng	0.37	B
429.mcf	74.35	A	459.GemsFDTD	24.70	A
435.gromacs	1.12	B	462.libquantum	26.24	A
436.cactusADM	4.74	A	464.h264ref	1.22	B
437.leslie3d	14.79	A	470.lbm	28.30	A
444.namd	0.11	B	473.astar	5.19	A
445.gobmk	0.51	B	481.wrf	0.12	B

6 Core criticality-aware memory scheduling

6.1 Core criticality

We introduced core criticality to describe the difference in latency between cores, and after storing CPU and GPU requests in different request queues, we implemented core criticality-aware scheduling for GPU requests. When the core contains more stalling warps waiting for data to be returned from DRAM, it is difficult to tolerate unfinished memory request delays. We define the core tolerance for memory latency as core tolerance. The quantification of core tolerance is divided into two steps. First, when the MC receives a memory request, it distinguishes the source of the memory request and counts the number of memory requests per core. When there is no staling warp in the core, in other words, the warp in this core is executing the calculation instruction or the required data already exist in the private cache, the core has no pending memory request. There is also no memory request from the core in the memory request queue. It is expected that this core can provide latency tolerance for memory. On the other hand, when the core has a long-waiting warp, the memory request issued by the core remains in the memory request queue. The next step is to, for each core, periodically calculate the ratio of memory requests issued by the core to the total number of memory requests in the memory request queue. The memory request issued by all cores is equal to the number of requests in the memory request queue, so the ratio takes a value between 0 and 1. We use this ratio to measure core latency tolerance. The higher the ratio, the more memory requests the core is waiting to process, the lower the core latency tolerance.

Next, we need to determine if the core is critical based on core latency tolerance. The latency tolerance is a fraction from 0 to 1, which is quantized into 8 equal parts for more convenient representation. For example, if the ratio is greater than 7/8, indicating that the core has a large number of pending memory requests, then the core is considered to be the most critical, and set a criticality level of 0 for the core. Ratio less than 1/8 means that the core has few memory requests to be processed and the core has high latency tolerance. It is also considered that this core is the least critical, and the criticality level is set to 7. The core criticality level is from 0 to 7, divided into 8 levels. Whether a core is critical is determined by the level and a Criticality rank Threshold (T-CR). When the T-CR is set to 4, the core is considered critical only if the core criticality level is less than or equal to 4, and the core memory request is a critical request.

6.2 Criticality request percentage

After determining the value of T-CR, the memory requests in the memory request buffer queue are divided into critical requests and non-critical requests. Use Criticality Request Percentage (PCR) to indicate the percentage of critical requests in the memory request buffer. High PCR means that most of the requests in the current request buffer are considered critical and need to be prioritized. The competition between critical requests is still fierce, and the scheduler will prioritize critical requests for a long time, seriously affecting the locality of the application. A low PCR value means that only a small percentage of requests in the current request buffer are considered critical, and critical memory-based requests are of little use. If the PCR value is 100% or 0%, the core criticality will become less important. What's more, the value of PCR is very important since it is used to distinguish between critical and non-critical requests and improve the effectiveness of critical-based memory scheduling strategies.

(1) PCR changes with T-CR

The value of the T-CR determines the value of the PCR. In order to visually show the effect of T-CR on PCR, Fig. 11 is represented by an example. The current core number is 4, and Fig. 11a shows the criticality rank of the four cores, which are 2, 4, 5, and 7. Figure 11b shows that under the condition of Fig. 11a, the value of T-CR is from 0 to 7, and the PCR also changes with T-CR. When $TCR_a > TCR_b$, $[PCR(TCR_a) \geq PCR(TCR_b)]$.

(2) PCR is related to the application

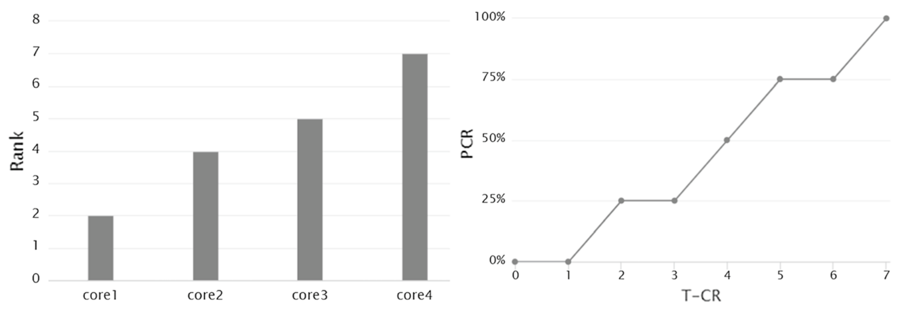

Fig. 11 PCR changes with T-CR

The difference in latency between cores is different from applications. If the value of T-CR is fixed during the scheduling process, regardless of the change in core latency difference, the appropriate PCR cannot be obtained, so that the critical request and other requests coexist. We chose the Rodinia benchmark [15] to test the difference in criticality rank of different applications. As shown in Fig. 12, the critical rank difference "diff" is defined as the range of core critical rank in the current MC. Diff-7 indicates that the current highest core criticality rank is 7 and the lowest is 0, which also means that the criticality differences of the current core are very large, and it is more suitable to use criticality to improve performance. Diff-0 means that the current critical rank of all cores is the same, so that the core criticality is not important. The ordinate of Fig. 12 is the percentage of DRAM cycles, which represents the percentage of the cycle of difference in the different critical levels during the entire execution of the application. Combining with the definition of diff, the more DRAM cycle ratio of diff-7, the more obvious the latency difference between cores in the whole execution process, and the effect based on criticality memory scheduling will be more obvious. When the proportion of DRAM cycles of diff-0 is more, it means that the core latency difference of this application is trivial, and the memory scheduling effect based on the locality of memory access will be good.

6.3 Balanced access locality and criticality

In the process of selecting the appropriate T-CR to allow critical requests and other requests in the current memory request queue to coexist, we still need to balance the locality and core criticality of the memory. We introduced a scheduling mode threshold (T-MS), which dynamically switches based on the critical-based memory scheduling policy and the local-based memory scheduling policy based on T-MS. This dynamic switching scheduling strategy becomes a criticality-aware memory scheduling (CAMS). As shown in Eq. 2, when the PCR is lower than the T-MS, the critical-based memory scheduling strategy is selected. The use of critical ordering

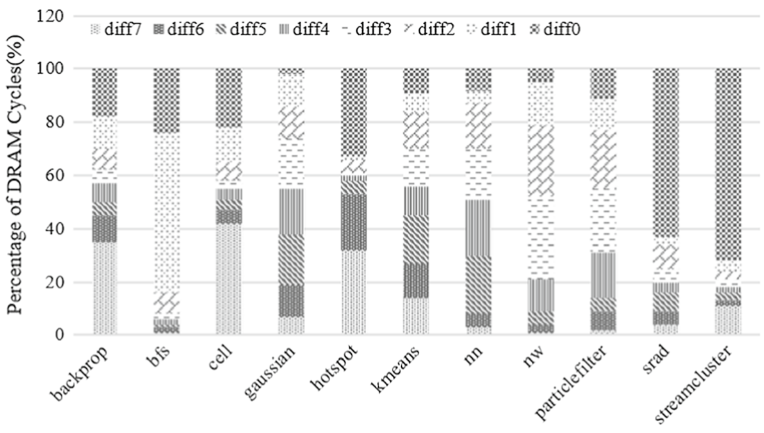

Fig. 12 Core latency difference for different applications

of memory requests gives priority to critical cores and their requests, resulting in core priority services with low latency tolerance. When the PCR is higher than the T-MS, it means that there are too many critical requests that are prioritized over other requests, and the request is no longer important. Switching to the memory scheduling policy FR-FCFS maximizes the row buffer hit ratio.

$$\mathrm{PCR}\begin{cases} \leq \mathrm{TMS}, & \text{criticality} \\ > \mathrm{TMS}, & \text{locality} \\ = 0, & \text{criticality} = \text{locality} \end{cases} \tag{2}$$

6.4 Algorithm design

The CAMS focus on the setting of T-CR and T-MS, which directly affects the performance of the scheduling strategy. One of the simplest algorithms is to set an initial value for the T-CR and T-MS at the beginning of the run and fix it during execution, called static-CAMS. According to the discussion in Sect. 6.3, we know that the latency differs from applications. The latency difference of the same application during the execution is also changed, so the T-CR needs to dynamically change according to the currently executed application. The T-MS is fixed, and the T-CR dynamic update memory scheduling strategy is called semi-CAMS. According to the discussion in Sect. 6.2, it is understood that T-MS dynamically changes with T-CR according to the CAMS. Table 3 is criticality-aware memory scheduling pseudo-algorithm.

7 Simulations and results

7.1 Experimental methodology

7.1.1 Experimental environment

We used the gem5-gpu [11] and gem5-gpu internal DRAM simulators to evaluate our proposed solution. gem5-gpu is a simulator that models tightly integrated CPU-GPU systems. It builds on gem5 [24], a modular full-system CPU simulator, and

Table 3 Criticality-aware memory scheduling pseudo-algorithm

Semi-CAMS	**CAMS** T-SMinit = 40%

T-CR=8.
for k ∈ {1…7} **do**
    if (0 < P-CR(k) ≤ T-MS < P-CR(k+1) **then**
        T-CR = k.
        end if; end for
return T-CR
	T-MS = T-SMinit; T-CR = 8.
	for k ∈ {1…7} **do**
	    if (0 < P-CR(k) ≤ T-MS < P-CR(k)) **then**
	        T-CR = k; T-MS = P-CR(T-CR).
	        end if; end for
	if T-CR = 8 **then** T-MS = 0%. **end if**
	return (T-CR, T-MS)

Table 4 Simulated system parameters

Parameter	Setting
CPU processor	4 cores, 3.25 GHZ, X86, out-of-order
GPU processor	4SM, 800 MHZ, X86, out-of-order
$L1$ Cache	32 KB private, 128b line, 2 way, LRU
$L2$ Cache	512 KB shared, 128b line, 8 way, LRU
DRAM	8 gb, channel/rank/bank: 1/2/8, Row buffer size: 2 KB

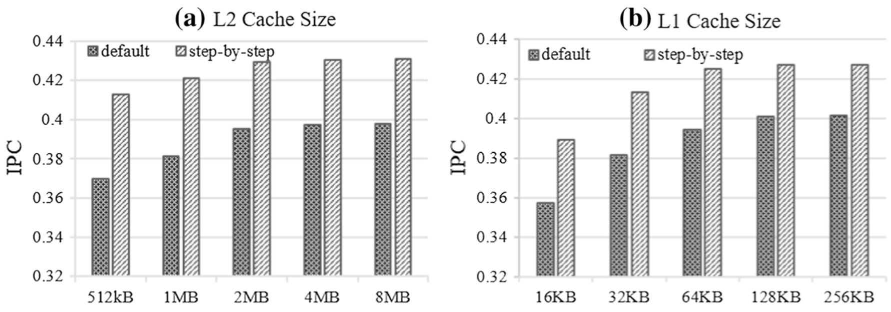

Fig. 13 Impact of $L2$ cache size and $L1$ Cache size on system performance

GPGPUSim [25], a detailed GPGPU simulator. Gem5-gpu is a configurable emulator. We model a quad-core CPU, and a GPU heterogeneous multi-core system. The system configuration is shown in Table 4. The simulation system is a multi-level cache structure with a first-level cache ($L1$ Cache) and a second-level cache ($L2$ Cache). $L1$ Cache is fast but has high manufacturing cost and limited capacity. $L2$ Cache is equivalent to the buffer of $L1$ Cache. It stores data that the processor needs but cannot hold in $L1$ Cache.

The size of $L2$ and $L1$ has a great impact on performance and will also affect the memory scheduling effect, as shown in Fig. 13. In Fig. 13a, the $L2$ size increases and the performance increases. However, when the $L2$ size increases from 2 to 4 M, it can be seen that the increase in performance is not obvious. Obviously, the $L2$ size increases to a certain extent, and the performance no longer changes significantly. In addition, when the $L2$ size is set to 512 KB, our step-by-step scheduling strategy has the highest performance improvement compared to the default policy. When $L2$ increases, the performance gap between the two decreases. The effect of $L1$ size in Fig. 13b is similar to $L2$ size. Based on the above discussion, we configured both $L1$ and $L2$ size with smaller values, 32 KB and 512 KB, respectively.

7.1.2 Performance metrics

We use the speedup to measure system throughput. The formula is calculated as 3,IPC$_{shared}$ and IPC$_{alone}$ refer to the average number of instructions executed per cycle(IPC)of the parallel application and the separate application, respectively. In

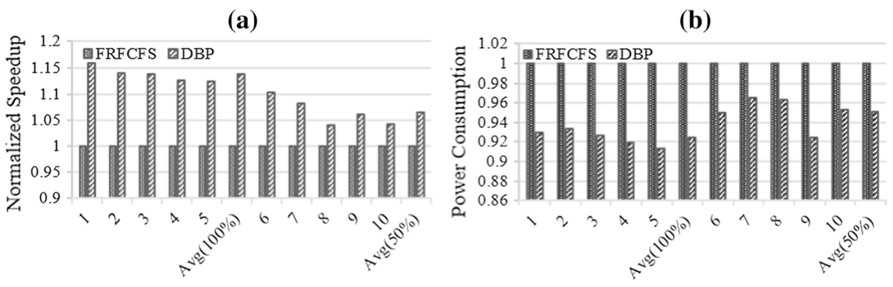

Fig. 14 Performance and power of different schemes with 4 CPU applications

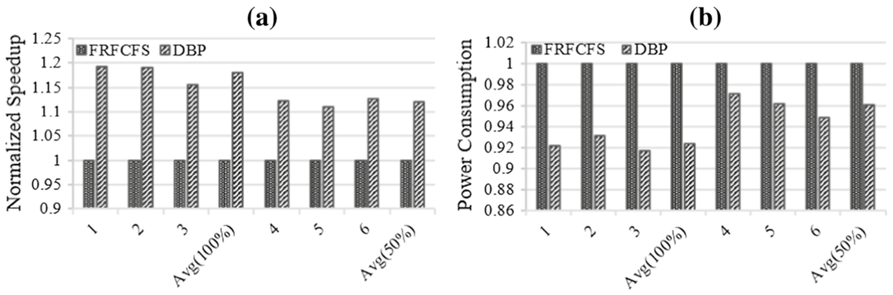

Fig. 15 Performance and power of different schemes with 8 CPU applications

gem5-gpu, IPC can be obtained in stat.txt generated after the application is executed. By integrating the McPAT power model and the GPUWattch model into the gem5-gpu, it is also possible to collect system power consumption in the application execution.

$$\text{Speedup} = \sum_{i=1}^{n} \frac{\text{IPC}_i^{\text{shared}}}{\text{IPC}_i^{\text{alone}}} \tag{3}$$

7.2 Individual CPU performance

First, after implementing a dynamic bank partitioning strategy for the CPU request queue, we compare it with the FR-FCFS memory scheduling strategy. The bank partitioning strategy proposed is only effective for the memory-intensive application. By currently running 4 CPU applications and 1 GPU application at the same time, the proportion of memory-intensive applications is 100% and 50%, respectively. 4 CPU applications, which are all memory-intensive applications, hold 100%. 4 CPU applications running at the same time hold 50%, with 2 memory-intensive applications, and 2 non-intensive applications. The experimental results are shown in Figs. 14 and 15.

Figure 14a shows the impact of dynamic bank partitioning strategies on performance. Workloads 1–5, 4 memory-intensive CPU applications are executed in parallel, and competing for shared memory is fierce. DBP isolates memory access requests from different applications without affecting bank-level parallelism, eliminating memory access interference between cores. In this case, performance increased by up to 15.8% and the average increase of 13%. The applications of workloads 6–10 executed in parallel have non-intensive applications. The experimental results are not obvious, and the performance is improved by 6%. The change in system energy consumption is shown in Fig. 14b in which workloads 1–5 reduce power consumption by up to 9%. The proportion of memory-intensive applications is different, as well as the effect of the strategy. We continued to select 8 CPU applications and 1 GPU application to execute in parallel, and application competed for shared memory more intensively. In this case, performance is increased by up to 19%, with an average increase of 17%, and the power consumption goes down by 8%.

7.3 Individual GPU performance

After implementing a core criticality-aware memory scheduling for the GPU request queue, we compare it with the FR-FCFS memory scheduling strategy. As shown in Fig. 16, we compared four scheduling strategies, static-cams, semi-cams, cams, and FR-FCFS. In fact, the first three scheduling strategies are improved on the basis of FR-FCFS. The benchmark was taken from Rodinia benchmark [15], and the experiment executed a GPU application separately. For static-cams, we set the T-CR to 4 and the T-MS to 20%. Performance increased by 3% in average, with the upper bound of to 1. For application *nn*, although performance did not improve, there is in line with expectations. For semi-cams, we fixed the T-MS to 40%,

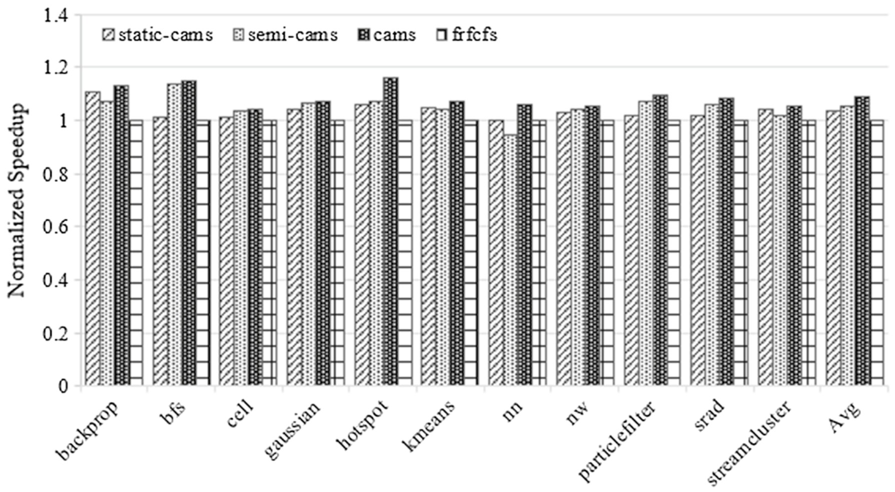

Fig. 16 System throughput of different schemes

and the performance increased by 5% in average, and up to 14%. Compared with static-cams, the effect has been improved. Finally, the observation of cams maintains the link between T-CR and T-SM, with an average performance increase of 9% and upper bound of 16%, and the effect is also improved compared with semi-cams. Compared with FR-FCFS, cams sacrificed partial access locality in order to reduce access latency for specific cores. We used the row buffer hit ratio to represent the locality of the application, and the core average latency represents a critical scheduling effect, as shown in Fig. 17. Figure 17a collects the row buffer hit ratio of the application. It can be seen that comparing with FR-FCFS, cams determine the impact on the locality of the application memory access. At the same time, it is observed from Fig. 17b that cams reduce the average core latency of the application, which proves that cams balance the locality and criticality of the application.

7.4 Combined CPU-GPU performance

Sections 7.3 and 7.4, respectively, select different memory scheduling strategies for isolated CPU request queues and GPU request queues to improve their performance. The step-by-step memory scheduling strategy combines the two strategies, and the results are shown in Fig. 18. Four memory-intensive applications and one GPU application form a workload. For example, workload1 refers to four CPU applications and a GPU application *streams*. Performance increased by an average of 17% and increased by up to 19%. Dynamic bank partitioning strategies for CPU queues and core criticality-aware memory strategies effects for GPU queues can be superimposed.

7.5 Hardware overhead

Set access count counter for each core to record the number of fetches per application. In addition, we need to set the hit rate counter for the CPU core to record the hit rate of the application. In order to achieve bank partitioning, two counters need to be set for each bank. In order to implement criticality awareness strategies, the critical to each core needs to be preserved. The biggest overhead is to create a memory request queue for isolating CPU and GPU fetches (Table 5).

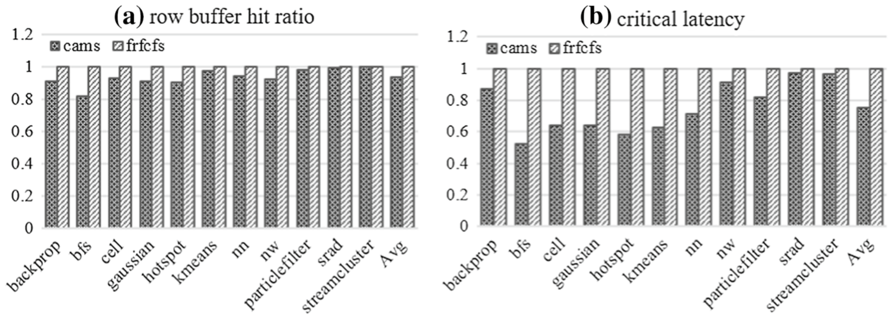

Fig. 17 DRAM row buffer hit ratio and memory latency of critical requests

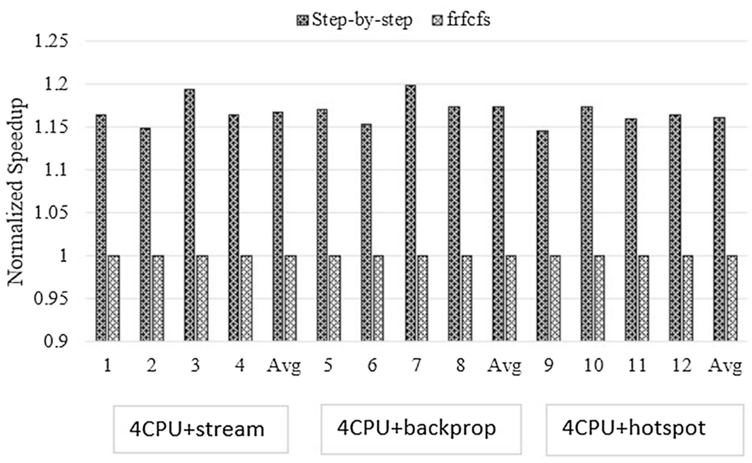

Fig. 18 Impact of step-by-step scheduling strategy on system performance

Table 5 Hardware overhead

Name	Function	Size
Access-counter	Periodically calculate the number of memory access requests	$(N_{\text{cpucore}} + N_{\text{gpucore}}) * \log_2 \text{access}_{\max}$
hitRate-counter	Periodically calculate hit rate	$N_{\text{cpucore}} * \log_2 \text{hitRate}_{\max}$
Bank pre-address	Bank address of previous memory access	$N_{\text{ch}} * N_{\text{rank}} * N_{\text{bank}} * \log_2 N_{\text{bank}}$
Bank access	Bank access per interval	$N_{\text{ch}} * N_{\text{rank}} * N_{\text{bank}} * \log_2 N_{\text{bank}}$
mc buffer size	Request a copy of the buffer, isolating the CPU and GPU request queue	300
Rank-counter	Periodically calculate the critical level	$N_{\text{cpucore}} * \log_2 \text{rank}_{\max}$

8 Conclusion

Introducing the GPU into a multi-core system, and being able to take advantage of the computing power of the GPU to improve system performance, has brought new challenges, especially resource competition between cores. In order to solve the problem of shared memory contention, we proposed a step-by-step memory access scheduling strategy. Our strategy also considered the memory interference of GPU applications on CPU applications, the memory interference between multiple CPU applications, and further considers the delay differences between GPU cores. Firstly, the MC isolates CPU and GPU memory requests while receiving memory requests depending on the source of the request. For the CPU memory request, the dynamic bank partitioning strategy is adopted to dynamically allocate different banks for different types of applications, effectively eliminating the memory request interference between applications without affecting the bank-level parallelism of the application. Secondly, for the GPU request, considering the latency difference between the application cores, we introduced the core criticality, maintained a balance between the two goals of maximizing the row buffer hit rate and reducing the core access latency, and designed the criticality-aware scheduling strategy. CAMS balances application access locality and criticality. The experiment was done in a heterogeneous multi-core system constructed by gem5-gpu. As for future work, we need to consider how to apply this solution on real machines, since the experimental results show that, compared with the default FR-FCFS strategy of gem5-gpu, the step-by-step scheduling strategy improves the performance by an average of 17%, and the maximum is 19%. We conclude that our method can effectively provide system performance and the introduction of core criticality in heterogeneous memory systems is a creative idea and a promising approach to improving the performance of heterogeneous systems in the future.

Acknowledgments This work is supported by Beijing Natural Science Foundation (4192007) and supported by the National Natural Science Foundation of China (61202076), along with other government sponsors. The authors would like to thank the reviewers for their efforts and for providing helpful suggestions that have led to several important improvements in our work. We would also like to thank all teachers and students in our laboratory for helpful discussions.

References

1. Lee JH, Shi W, Gil JM (2018) Accelerated bulk memory operations on heterogeneous multi-core systems. J Supercomput 74(12):6898–6922
2. Fang J, Lu Y, Liu S, Lu J, Chen T (2015) KL_GA: an application mapping algorithm for mesh-of-tree (MoT) architecture in network-on-chip design. J Supercomput 71(11):4056–4071
3. Kim Y, Han D, Mutlu O, Harchol-Balter M (2010) ATLAS: a scalable and high-performance scheduling algorithm for multiple memory controllers. In: HPCA—16 2010 the Sixteenth International Symposium on High-Performance Computer Architecture, Bangalore, pp 1–12
4. Subramanian L, Lee D, Seshadri V, Rastogi H, Mutlu O (2014) The blacklisting memory scheduler: achieving high performance and fairness at low cost. IEEE 32nd International Conference on Computer Design (ICCD), Seoul, pp 8–15
5. Kim Y, Papamichael M, Mutlu O, Harchol-Balter M (2010) Thread cluster memory scheduling: exploiting differences in memory access behavior. In: 43rd Annual IEEE/ACM International Symposium on Microarchitecture, Atlanta, GA, pp 65–76
6. Liu L, Cui Z, Xing M, Bao Y, Chen Y, Wu C (2012) A software memory partition approach for eliminating bank-level interference in multicore systems. In: 21st International Conference on Parallel Architectures and Compilation Techniques (PACT), Minneapolis, MN, pp 367–375
7. Wei M, Feng X, Xue J, Jia Y (2014) Software-hardware cooperative dram bank partitioning for chip multiprocessors. In: Ifip International Conference on Network and Parallel Computing. Springer
8. Jog A, Kayiran O, Nachiappan NC, Mishra AK, Kandemir M, Mutlu O, Iyer R, Das C (2013) OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance. In:

Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. ACM

9. Lakshminarayana NB, Lee J, Kim H, Shin J (2012) DRAM scheduling policy for GPGPU architectures based on a potential function. IEEE Comput Archit Lett 11(2):33–36
10. Rixner S, Dally WJ, Kapasi UJ, Mattson P, Owens JD (2000) Memory Access Scheduling. ACM Sigarch Comput Archit News 28(2):128–138
11. Power J, Hestness J, Orr MS, Hill MD, Wood DA (2015) gem5-gpu: a heterogeneous CPU–GPU simulator. IEEE Comput Archit Lett 14(1):34–36
12. Gao K, Fan D, Wu J, Liu Z (2015) Decoupling contention with victim row-buffer on multicore memory systems. In: 2015 IEEE international parallel and distributed processing symposium workshop, Hyderabad, pp 454–463
13. Fang J, Zhang X, Liu S, Chang Z (2019) Miss-aware LLC buffer management strategy based on heterogeneous multi-core. J Supercomput 2019:1–10
14. SPEC CPU2006. http://www.spec.org/spec2006
15. Che S, Boyer M, Meng J, Tarjan D, Sheaffer WJ, Lee S, Skadron K (2009) Rodinia: a benchmark suite for heterogeneous computing. In: 2009 IEEE International Symposium on Workload Characterization (IISWC), Austin, TX, pp 44–54
16. Jog A, Kayiran O, Pattnaik A, Kandemir MT, Mutlu O, Iyer R, Das C (2016) Exploiting core criticality for enhanced GPU performance. In: The 2016 ACM SIGMETRICS International Conference. ACM
17. Muralidhara SP, Subramanian L, Mutlu O, Kandemir M, Moscibroda T (2011) Reducing memory interference in multicore systems via application-aware memory channel partitioning. In: 44th annual IEEE/ACM international symposium on microarchitecture (MICRO), Porto Alegre, pp 374–385
18. Jeong MK, Yoon DH, Sunwoo D, Sullivan M, Lee I, Erez M (2012) Balancing DRAM locality and parallelism in shared memory CMP systems. In: IEEE International Symposium on High-Performance Comp Architecture, New Orleans, LA, pp 1–12
19. Xie M, Tong D, Huang K, Cheng X (2014) Improving system throughput and fairness simultaneously in shared memory CMP systems via dynamic bank partitioning. In: 20th International Symposium on High Performance Computer Architecture (HPCA), Orlando, FL, pp 344–355
20. Mutlu O, Moscibroda T (2008) Parallelism-aware batch scheduling: enhancing both performance and fairness of shared DRAM systems. In: 2008 International Symposium on Computer Architecture, Beijing, pp 63–74
21. Li D, Tor MA (2016) Inter-core locality aware memory scheduling. IEEE Comput Archit Lett 15(1):25–28
22. Wang H, Singh R, Schulte M, Kim NS (2014) Memory scheduling towards high-throughput cooperative heterogeneous computing. In: 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT), Edmonton, AB, pp 331–341
23. Ausavarungnirun R, Chang KKW, Subramanian L, Loh GH, Mutlu O (2012) Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In: 39th Annual International Symposium on Computer Architecture (ISCA), Portland, OR, pp 416–427
24. Binkert N, Beckmann B, Black G, Reinhardt SK, Saidi A, Basu A, Hestness J, Hower DR, Krishna T, Sardashti S, Sen R, Sewell K, Shoaib M, Vaish N, Hill MD, Wood DA (2011) The gem5 simulator. ACM SIGARCH Comput Archit News 39(2):1–7
25. GPGPU-Sim. http://www.gpgpu-sim.org. Accessed 25 Jan 2018