CacheC: LLM-based GPU Cache Management to Enhance Kernel Concurrency

Mengyue Xi[0009−0003−5711−3110], Jingyi He[0009−0001−8672−9817], and Xianwei Zhang✉[0000−0003−3507−4299]

Sun Yat-sen University, Guangzhou, China
ximy@mail2.sysu.edu.cn, hejy268@mail2.sysu.edu.cn,
zhangxw79@mail.sysu.edu.cn

Abstract. Each new generation of GPUs significantly enhances the resources available for diverse applications, with kernel concurrency playing a crucial role in maximizing utilization and boosting performance. However, existing kernel concurrency strategies usually tend to neglect cache contention, where concurrent kernels potentially target the same cache levels. Traditional cache management methods are inadequate for addressing this issue, as they focus on individual kernels without heavily considering inter-kernel interactions. To overcome these challenges, we propose `CacheC`, a method that utilizes large language models (LLMs) to analyze cache affinity at the granularity of individual load instructions. For each kernel pair, `CacheC` extracts detailed features of all loads, evaluates their cache affinity across levels, and scores their suitability for concurrency. Based on these scores, `CacheC` not only selects kernel pairs with appropriate cache compatibility but also formulates load-specific cache bypassing strategies to enhance utilization. By iteratively scheduling kernel pairs and adjusting their cache policies, `CacheC` dynamically optimizes cache utilization and reduces cache contention during concurrent kernel execution. Experiments on off-the-shelf GPUs demonstrate that `CacheC` achieves a 19.67% reduction in turnaround time and a 24.48% improvement in throughput. It also delivers an average speedup of 1.337× across scheduled kernel pairs, showcasing its effectiveness in alleviating cache contention and enhancing kernel concurrency performance.

Keywords: GPU cache management · Concurrent kernel execution · Kernel pairing · Large language models.

1 Introduction

Graphics Processing Units (GPUs) are known for their exceptional computational throughput, thanks to their thousands of threads and efficient thread-switching techniques that hide memory request latency. Each new GPU generation brings substantial improvements in computational capabilities, cache capacity, and memory bandwidth. For example, AMD's RDNA 3 [1] doubles compute performance to 61 teraflops, and enhances memory bandwidth to 960GB/s over RDNA 2 [2].

With the vast increase in GPU resources, kernel concurrency has become crucial for maximizing GPU utilization, particularly by exploiting unused resources. Many studies have explored kernel concurrency, which can be categorized into three primary domains: techniques to maximize kernel overlap [3–6]; kernel pairing selections for efficient utilization of computational and memory resources [7–9]; and scheduling algorithms for managing kernel pool dispatches [10–13]. However, these studies fail to address cache contention when concurrent kernels share the same cache levels, leading to frequent swapping and modification of cache blocks.

Conventional cache management techniques are not well-suited for handling kernel concurrency, as they are primarily designed for single-kernel execution. Approaches such as software-controlled cache bypassing [14–16] and hardware-supported memory request optimization [17–19] are insufficient for addressing the complexities of concurrent kernel execution. These methods fail to account for inter-kernel cache interactions and are prone to involve significant offline processing and hardware modifications, making them impractical for real-world applications.

To address this issue, we propose `CacheC`, a cache management strategy for kernel concurrency. Our approach leverages Large Language Models (LLMs) to analyze cache access patterns. LLMs have shown significant potential in systems and architecture [20–22], exhibiting strong analytical capabilities that allow them to identify complex cache behaviors across kernels. By incorporating prompt engineering [23], `CacheC` enhances LLM to dynamically adapt to domain-specific requirements through contextual prompts, eliminating the need for retraining. This makes our approach more advantageous than traditional model training methods.

Building on these insights, `CacheC` integrates cache pattern analysis, kernel pairing, cache bypass tuning for each pair, and concurrent kernel scheduling. For cache pattern analysis, `CacheC` employs LLM to assign a cache affinity score to each load of the pair, using three key techniques: input prompt templates, a three-stage progressive inquiry process, and iterative generation. These techniques work together to iteratively refine and optimize the results. Regarding kernel pairs, `CacheC` evaluates and ranks all potential combinations based on the complementarity of their cache patterns, determined by the cache affinity of each load. For each selected pair, `CacheC` develops a tailored cache bypass strategy to optimize load behavior, ensuring the selection of the cache level with the highest affinity. To further facilitate concurrent kernel scheduling, `CacheC` employs a greedy algorithm to iteratively select the most compatible pairs for execution, applying the corresponding bypass strategy. If no suitable pair is found, kernels are then just scheduled sequentially. In conclusion, this paper makes the following contributions:

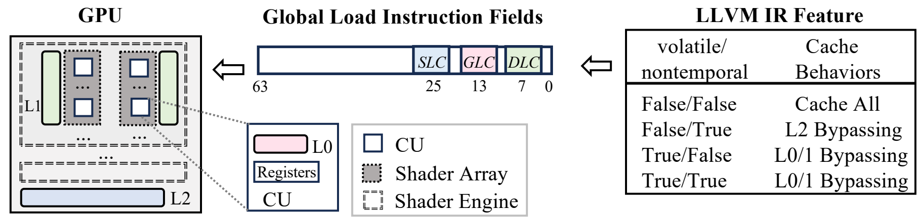
– We identify cache contention in concurrent kernels, where they compete for resources at the same affinity levels. We also highlight the limitations of traditional cache management approaches, which are agnostic to inter-kernel interactions.

– We propose leveraging LLMs with prompt engineering to overcome these limitations, as it is more time-efficient than traditional model training. This enables the analysis of complex inter- and cross-kernel cache patterns while allowing fine-grained control over cache behaviors for each load.
– We develop `CacheC`, which integrates four components: LLM-based cache pattern analysis, cache pattern-driven kernel pairing, cache bypass adjustment, and kernel scheduling. Evaluations show that `CacheC` achieves a 24.48% throughput boost and a 1.337× speedup across scheduled kernel pairs.

2 Background and Motivation

2.1 GPU Architecture

Figure 1 illustrates the GPU memory hierarchy, based on the recent AMD RDNA design [24] [1]. From an architectural perspective, a GPU consists of multiple compute units (CUs), also referred to as streaming multiprocessors (SMs). These CUs are grouped into shader arrays (SAs), and multiple SAs are further organized within shader engines (SEs). In terms of memory hierarchy, each CU is equipped with private registers and an L0 cache. An SA shares access to an L1 cache, while all CUs within the GPU utilize a globally shared L2 cache, which is directly connected to the main DRAM memory.



volatile/ nontemporal	Cache Behaviors
False/False	Cache All
False/True	L2 Bypassing
True/False	L0/1 Bypassing
True/True	L0/1 Bypassing

Fig. 1: The architecture and cache management software of a GPU, augmented by LLVM IR capabilities, facilitate precise control over memory instruction bits to regulate cache behavior.

As computational demands continue to rise, GPUs are incorporating additional CUs and expanding memory resources. Notably, on-chip caches in GPUs have been significantly enhanced to support large-scale thread parallelism. To optimize cache management for varying workloads, software-controlled techniques such as residency control [25] and flexible policy tuning [26] have been introduced. As shown in Figure 1, AMD RDNA architectures feature annotation bits (SLC, GLC, DLC) to control data coherency and cache behavior. These can be managed through LLVM IR features `volatile` and `nontemporal` in the AMDGPU backend [26]. The annotation bits correspond to specific cache policies, where GLC, DLC, and SLC enable Miss-Evict and Stream policies that

[1] Although this paper focuses on AMD GPUs, the analysis is also applicable to other vendors.

reduce the caching of recent data. These policies are consolidated into three cache management strategies: Cache, L0/1 Bypassing, and L2 Bypassing [24]. In this paper, we analyze the L0/1 and L2 cache affinity of load instructions to utilize the aforementioned bypassing modes, thereby refining cache management policies.

2.2 Alleviate Cache Contention of Kernel Concurrency

In this section, we explore the potential cache contention during the concurrent execution of two kernels [2]. Figure 2 illustrates the global load (*ld*) behavior of the kernel *CON* (*convolution*) [27] when run independently and concurrently with other kernels [27–29], including *DT1* (*dct8*8_1*), *DT2* (*dct8*8_2*), *LBM* (*lbm*), *HS1* (*hybridsort_1*), *HS2* (*hybridsort_2*), *SPV* (*spmv*), and *PAT* (*particlefilter*). The figure highlights the performance improvements achieved by bypassing *ld* through software-controlled LLVM IR features, discussed in Section 2.1. When *CON* runs alone, L2 Bypassing (-0.45%) and L0/1 Bypassing (-1.05%) show minimal negative effects. However, when *CON* runs concurrently with other kernels, bypassing *ld* demonstrates varying degrees of improvement, suggesting that cache contention with loads from other kernels could be mitigated by bypassing. Notably, L2 contention is relatively pronounced between *CON* and *DT1* (2.35%) or *DT2* (2.42%). When *CON* is paired with *LBM*, substantial contention is observed at the L2 (5.98%), and relatively low contention is observed at the L0/1 (1.83%). In contrast, the combination of *CON* and *HS1* exhibits lower L2 contention (2.67%), but more significant L0/1 contention (7.96%). For the remaining kernel combinations, cache contention is minimal, and the bypassing effect closely mirrors that observed when running *CON* independently.

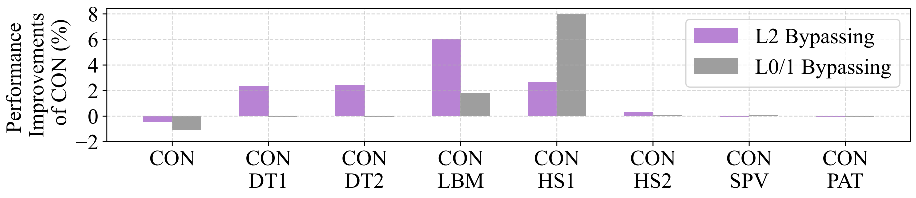


Fig. 2: Performance improvements of the sole load in *CON* when running independently and in combination with other kernels, under two bypassing modes.

These findings emphasize the importance of kernel pairing strategies based on cache behavior and the effectiveness of bypassing in mitigating contention. Kernel concurrency usually ignores such challenges, and traditional cache management fails to address them effectively. In this paper, we explore the use of LLMs to optimize cache management in kernel concurrency.

[2] This paper focuses on two kernels, but the approach can be easily extended to more, as shown in Section 4.5.

3 Design

Figure 3 illustrates the overall design of `CacheC`, consisting of four key components. First, LLM (A) analyzes the cache patterns of global load instructions across kernel pairs and generates the load score template (LST), indicating the affinity of each load. For cache management, there are two sub-components: cache pattern driven kernel pairing (B), which evaluates the fitness of cache patterns and produces the score table (ST), and cache bypassing for kernel pairs (C), which fine-tunes the cache behavior of each load, allocating the most suitable cache levels based on affinity. The final component, the kernel scheduler (D), uses the score table (ST) to allocate kernels in the pool for execution, enabling concurrent or sequential execution while applying the cache bypass adjustment.

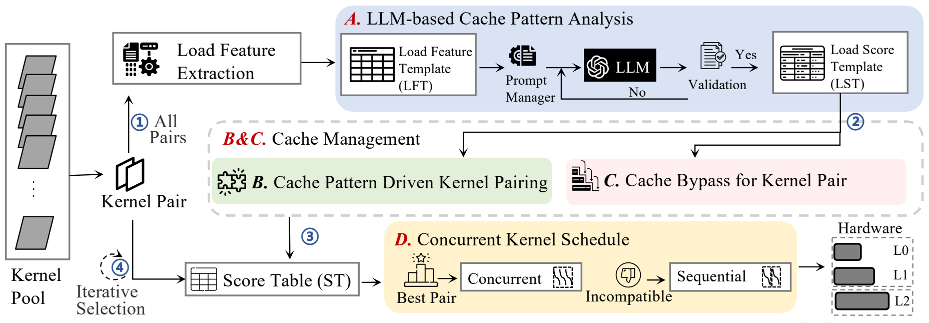


Fig. 3: Overview of `CacheC`: LLM-based Cache Pattern Analysis (A), Cache Management with Kernel Pairing (B) and Bypass (C), and Concurrent Kernel Scheduling (D).

The workflow is as follows: for all combinations of kernel pairs in the kernel pool, the load features of each pair are extracted to construct the load feature template (LFT), which is input into the LLM for cache pattern analysis (①). Once the LLM generates the LST, cache management uses it to evaluate the cache pattern fitness of each kernel pair and to generate a corresponding cache bypass strategy (②). After processing all pairs, the ST is formed (③). During scheduling (④), `CacheC` selects the best kernel pair from the kernel pool based on the ST and dispatches the pair for concurrent execution. If no pair fits the cache pattern criteria, `CacheC` schedules the kernels sequentially. During execution, the previously generated cache bypass strategy is employed to control hardware cache behaviors.

3.1 LLM-based Cache Pattern Analysis

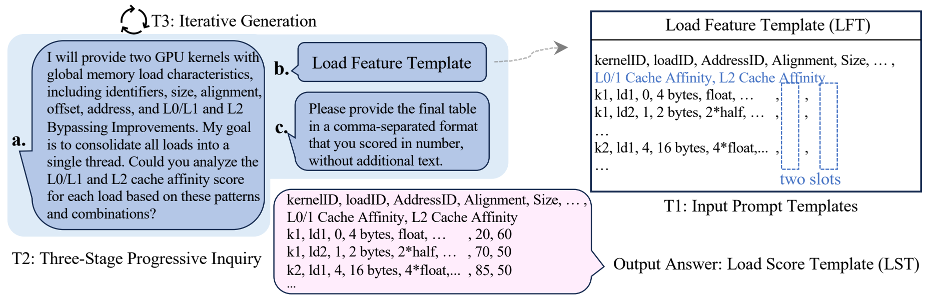
In this section, we embrace LLM to analyze the cache patterns of all global loads in kernel pairs. In `CacheC`, the load feature extraction component first captures the features of all global loads in a kernel pair, as outlined in Table 1. These load features are then input to the LLM, which generates the L0/1 and L2 cache affinity scores for all loads.

Table 1: Selected Load Features

Features	Description
Kernel ID	Kernel identification of a load
Load ID	Load identification
Address ID	Array Identification referenced by a load
Alignment	Byte alignment of a load
Size	Access size of a load
Offset	Access offset relative to referenced arrays
Access Percentage	Percentage of access numbers of a load
L0/1 Bypass	Bypassing effect in the L0/1 of a single kernel
L2 Bypass	Bypassing effect in the L2 of a single kernel

However, there are two challenges when applying LLM for cache pattern analysis in this context: *1)* How to convert load features into a text format that LLM excels in while ensuring the clarity of the content for the LLM to fully understand the task? *2)* How to narrow the LLM's outputs from a broad range of analyses to the specific and detailed cache affinity scores required? To address these challenges, `CacheC` incorporates three key techniques to ensure LLM validity and usability.

Input Prompt Templates. As T1 in Figure 4 illustrates, `CacheC` employs a structured LFT to standardize the input for the LLM. This template is formatted with well-defined attributes, ensuring consistency and enabling straightforward analysis of kernel load features. Two slots are left intentionally blank to guide the LLM in synthesizing missing load scores. The output, structured as LST, embodies a deterministic and predictable format, effectively reducing the variability commonly observed in LLM responses.


T3: Iterative Generation
a.
b.
Load Feature Template
c.
T2: Three-Stage Progressive Inquiry
Load Feature Template (LFT)
two slots
T1: Input Prompt Templates
Output Answer: Load Score Template (LST)

I will provide two GPU kernels with global memory load characteristics, including identifiers, size, alignment, offset, address, and L0/L1 and L2 Bypassing Improvements. My goal is to consolidate all loads into a single thread. Could you analyze the L0/L1 and L2 cache affinity score for each load based on these patterns and combinations?

Please provide the final table in a comma-separated format that you scored in number, without additional text.

```
kernelID, loadID, AddressID, Alignment, Size, … ,
L0/1 Cache Affinity, L2 Cache Affinity
k1, ld1, 0, 4 bytes, float, …      ,    ,
k1, ld2, 1, 2 bytes, 2*half, …     ,    ,
…
k2, ld1, 4, 16 bytes, 4*float,...  ,    ,
…
```

```
kernelID, loadID, AddressID, Alignment, Size, … ,
L0/1 Cache Affinity, L2 Cache Affinity
k1, ld1, 0, 4 bytes, float, …      , 20, 60
k1, ld2, 1, 2 bytes, 2*half, …     , 70, 50
k2, ld1, 4, 16 bytes, 4*float,...  , 85, 50
...
```

Fig. 4: Three techniques: Input Prompt Templates (T1), Three-stage Progressive Inquiry (T2), and Iterative Generation (T3), along with the generated LST by LLM.

Three-stage Progressive Inquiry. Recognizing the limitations of single-round interactions with LLMs, `CacheC` employs a progressive inquiry process to incrementally refine the LLM's outputs for cache affinity analysis. As illustrated in T2 of Figure 4, this process begins with the initialization stage (a), where contextual information, including task definitions and background on cache behavior, is provided to help the LLM develop a comprehensive understanding of cache pattern analysis. It then proceeds to the feature analysis stage (b), where the LFT, detailing specific kernel load features, is supplied, directing the LLM to generate cache affinity scores based on these features. Finally, the purification stage (c) refines the LLM-generated outputs into the LST format, ensuring alignment with the predefined structure and eliminating inconsistencies. A prompt manager orchestrates this entire process, ensuring seamless progression and robust results.

Iterative Generation. As shown in T3 of Figure 4, `CacheC` uses iterative refinement to ensure high-quality outputs. If initial results are vague, the progressive inquiry process is reinitiated to achieve precise numerical cache affinity scores that align with the LST format. This iterative approach continues until the desired precision is achieved.

3.2 Cache Pattern Driven Kernel Pairing (Cpair)

After generating the LST, which includes the L0/1 and L2 cache affinity scores for all loads, `CacheC` evaluates kernel pairing based on these scores. These affinity scores assess the cache pattern fit between kernels. For all kernel pair combinations, `CacheC` generates the ST to evaluate the compatibility of each pair by considering their cache affinity scores.

The L0/L1 and L2 affinity scores ($L0/1_{ij}$ and $L2_{ij}$ for the j_{th} load of the i_{th} kernel) are extracted and normalized to calculate the bypassing score ($Bypass_{ij}$), as shown in Equation 1. The bypassing score evaluates the lack of affinity across all cache levels. Each load generates a vector, $load\_vector_{ij}$, containing the three affinity scores ($L0/1_{ij}$, $L2_{ij}$, and $Bypass_{ij}$), as shown in Equation 2. The overall cache affinity of a kernel, denoted as $kernel\_vector_i$, is obtained by calculating the weighted sum of the load affinity vectors, as illustrated in Equation 3. The weight is assigned based on the access count ratio. For M kernels (with two kernels selected in this study, as explained in Section 4.5), a more cache-compatible pair means their kernel vectors are more orthogonal, indicating distinct cache resource requirements. This relationship is captured by the $pair\_score$ in Equation 4, where a lower score indicates greater orthogonality and suitability for concurrent execution. It is evident that the maximum $pair\_score$ is 3. `CacheC` computes the $pair\_score$ for all kernel pairs and integrates the results into a ST for kernel concurrency scheduling.

$$Bypass_{ij} = \min((1 - L0/1_{ij} - L2_{ij}), 0) \tag{1}$$

$$load\_vector_{ij} = (L0/1_{ij}, L2_{ij}, Bypass_{ij}) \tag{2}$$

$$kernel\_vector_i = \sum_{j=1}^{N} load\_vector_{ij} * weight_{ij} \tag{3}$$

$$pair\_score = \prod_{i=1}^{M} kernel\_vector_i, \quad M = 2 \tag{4}$$

3.3 Cache Bypass for Kernel Pair (Cbypass)

After generating the LST, `CacheC` devises a cache bypass strategy for each kernel pair. This strategy refines cache bypassing modes for each load, ensuring effective cache allocation for loads with higher affinity. Based on the normalized L0/1 and L2 cache affinity scores, `CacheC` determines the appropriate cache management policy for each load, as outlined in Algorithm 1.

Algorithm 1 Cache Bypassing Strategy for Kernel Pair

```
Input:
    $Pair$: $(Kernel_1, Kernel_2)$, set the value of $M$ to 2;
    $L0/1_{ij}$: the L0/1 affinity score for $j_{th}$ load of $i_{th}$ Kernel;
    $L2_{ij}$: the L2 affinity score for $j_{th}$ load of $i_{th}$ Kernel;
Output:
    $policies_{ij}$: the bypassing strategy for $j_{th}$ load of $i_{th}$ Kernel.
1: for each $load_{ij} \in Pair$ do
2:     if $L0/1_{ij} < L2_{ij} \wedge L2_{ij} \geq HighThres$ then $policies_{ij} \leftarrow$ L0/1 Bypassing
3:     else if $L0/1_{ij} > L2_{ij} \wedge L0/1_{ij} \geq HighThres$ then $policies_{ij} \leftarrow$ L2 Bypassing
4:     else if $L0/1_{ij} \geq LowThres \wedge L2_{ij} \geq LowThres$ then $policies_{ij} \leftarrow$ Cache
5:     end if
6: end for
```

For each load, $load_{ij}$ (where j is the load index of the i_{th} kernel in the pair), the algorithm evaluates the load's affinity using the normalized L0/1 and L2 scores, $L0/1_{ij}$ and $L2_{ij}$. If the L2 affinity exceeds the higher threshold, $HighThres$, `CacheC` applies the *L0/1 Bypassing* policy. If the L1 affinity surpasses $HighThres$, `CacheC` applies the *L2 Bypassing* policy. If both scores exceed $LowThres$, indicating sufficient affinity across all cache levels, `CacheC` assigns the *Cache* policy to the load. We set 0.5 and 0.8 for $LowThres$ and $HighThres$, respectively.

3.4 Concurrent Kernel Scheduling

After integrating the evaluated scores into the ST and applying cache bypass tuning, `CacheC` schedules kernels using a greedy algorithm, selecting the pair with the lowest ST score for concurrent execution to optimize cache utilization. Once a pair's execution is complete, `CacheC` removes it from the ST and identifies the next pair with the lowest score for dispatch. If no kernel pairs are deemed suitable for concurrent execution, `CacheC` schedules the remaining kernels sequentially, adhering to a First-In-First-Out policy.

4 Evaluation

All experiments and analyses are conducted on an AMD Radeon RX 6900 XT, leveraging the RDNA2 architecture with a GFX1030 ISA [2]. The device features a 32KB L0 cache per CU, a 128KB L1 cache per SA, and a 4MB global L2 cache. For the LLM, we employ the OpenAI ChatGPT API based on the GPT-4o model [30]. Kernel concurrency is achieved by integrating the two kernels into a single thread [4], enabling full concurrent execution and reducing interference potentially caused by the sequential execution of streams [31]. The benchmarks comprise 14 real-world kernels exhibiting diverse cache patterns, as detailed in Table 2. The experimental schemes are as follows:

Kernel execution schemes:

– `Baseline`: Kernels are all executed sequentially.
– `Rpair`: Kernels are randomly paired from the kernel pool for concurrent execution, with scheduling repeated 10 times to compute the average.
– `Cpair`: Kernels are paired for concurrent execution based on the cache pattern analysis discussed in Section 3.2.

Cache management schemes:

– `Mpache`: A conventional and representative cache management mechanism that focuses solely on intra-kernel interactions through offline profiling [16].
– `Cbypass`: Cache bypass method proposed by `CacheC` for cache management in Section 3.3.
– `Exhaustive`: An exhaustive evaluation of all possible load combinations across the three cache management modes to determine the optimal strategy.

Kernels	*abbr.*	Kernels	*abbr.*
gelu	GEL	spmv	SPV
relu	REL	paticlefilter	PAT
reverse	REV	dct8x8-1	DT1
swish-1	SW1	dct8x8-2	DT2
swish-2	SW2	hybridsort-1	HS1
convolution	CON	hybridsort-2	HS2
lbm	LBM	maxpool	MAX

Table 2: List of evaluated GPU benchmark kernels [27–29, 32].

Schedule Order	Kernels
1	CON_DT1
2	GEL_SW2
3	HS2_REV
4	HS1_SW1
5	MAX_DT2
6	PAT_REL
7	LBM
8	SPV

Table 3: Schedule orders generated by `CacheC`.

4.1 Performance Improvement

Assuming there are 14 kernels in the kernel pool, Table 3 is the kernel schedule list generated by `CacheC`. Table 4 illustrates the performance benefits achieved by different schemes, compared to `Baseline`. Specifically, it highlights the reduction in turnaround time and the improvement in throughput for the entire kernel pool under various optimization strategies.

In detail, `CacheC` outperforms all other schemes, reducing turnaround time by 19.67% and improving throughput by 24.48% with the `Cpair` and `Cbypass` methods. Additionally, `Cpair` outperforms `Rpair` by reducing turnaround time by 4.58% and improving throughput by 5.57%, demonstrating the effectiveness of `Cpair` in selecting cache-compatible kernels. `Cbypass` also surpasses `Mpache`, reducing turnaround time by 4.16% and 7.04%, respectively, while improving throughput by 5.41% and 10.02% when equipped with `Rpair` and `Cpair`. This improvement is attributed to `CacheC`'s ability to detect variations in cache patterns induced by concurrency, a capability that `Mpache` lacks.

Table 4: Reduction in turnaround time and improvement in throughput for kernel pool scheduling compared to the baseline.

Schemes	Turnaround Reduction	Throughput Improvement
Rpair	7.04%	7.58%
Cpair	11.62%	13.15%
Rpair+Mpache	10.16%	11.31%
Cpair+MPache	12.63%	14.46%
Rpair+Cbypass	14.32%	16.72%
Cpair+Cbypass(`CacheC`)	**19.67%**	**24.48%**

4.2 Speedup Breakdown

It is valuable to further explore performance by examining the outcomes when two kernels, generated by `CacheC`, are paired in terms of cache pattern fitness. Table 3 presents the kernel schedule list generated by `CacheC` for the 14-kernel pool. The kernel pairs are selected and scheduled in ascending order of their scores, from low to high. In particular, `CacheC` schedules *LBM* and *SPV* for sequential execution, as this pair exhibits high scores, indicating significant cache contention and making them unsuitable for pairing.

Kernel Pair Speedup. Across the schedule lists, `Cpair` alone achieves a 1.163× speedup over the baseline (sequential execution of the two kernels), as depicted in Figure 5, demonstrating the benefits of pairing execution. In the kernel pair compatibility analysis, the speedup trend in Figure 5 is not strictly monotonic because `CacheC` prioritizes kernel pairs with complementary cache patterns, while other influencing factors are not the primary focus in this context. For example, while the *CON* and *DT1* pair has low load access demands, both kernels require significant computational resources, leading to a 0.977x performance degradation. However, `Cbypass` helps mitigate this issue by balancing resource contention and optimizing utilization. Despite various influencing factors, five out of six pairs show improvements, highlighting the importance of cache pattern compatibility in kernel selection.

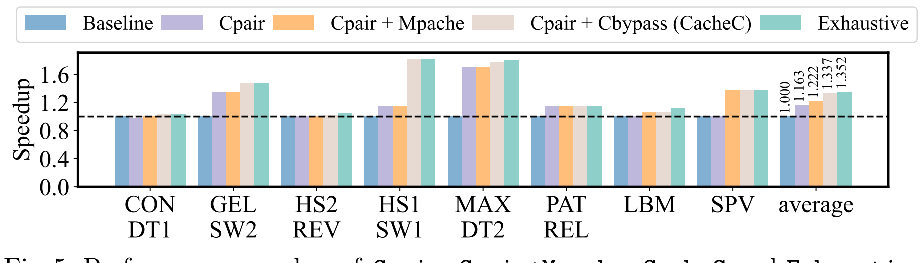


Fig. 5: Performance speedup of `Cpair`, `Cpair+Mpache`, `CacheC` and `Exhaustive` schemes, normalized to baseline.

Cache Bypass Speedup. By comparing `CacheC` and `Cpair+Mpache`, which yield average speedups of 1.337× and 1.222×, respectively, the strength of `Cbypass` is again confirmed. For four out of six kernel pairs, `Mpache` is less effective than `Cbypass`, as it fails to recognize cache pattern changes when pairing and continues to focus on individual kernels. For the single kernel *LBM*, `CacheC` achieves a 1.058× speedup, while `Mpache` results in a slightly higher 1.101× speedup, as `CacheC` omits some beneficial load patterns.

4.3 Verification of LLM outputs

The intermediate responses from the LLM are that it assigns weights to features in the LFT and evaluates each load based on cache level characteristics. It computes an overall score to assess memory behavior and identifies load interactions within and across kernels. It uses prior knowledge to refine cache behavior assessments during concurrent execution.

To strengthen the validation, `CacheC` is compared with `Exhaustive` in Figure 5, which tests all possible load combinations under different cache management modes to determine the optimal strategy, serving as the gold standard. The closer the bypassing strategies generated by `CacheC` align with those of `Exhaustive`, the more accurate the LLM outputs are. For *CON-DT1*, *GEL-SW2*, and *HS1-SW1*, the strategies are nearly identical, resulting in almost the same speedup. However, for *HS2-REV*, *MAX-DT2*, and *LBM*, `CacheC` misses some loads for bypassing, leading to slightly lower speedup compared to `Exhaustive`. In the case of *PAT-REL*, the outputs of `CacheC` show minor differences from `Exhaustive` due to small score fluctuations, which may cause the LLM to bypass cache-insensitive loads with minimal impact on performance. Overall, the close alignment between `CacheC` (average 1.337× speedup) and the exhaustive method (average 1.352× speedup) indicates that the generated cache patterns accurately reflect true affinity.

4.4 Cache Hits

We collect hardware utilization data using ROCm-SMI [33] to monitor L2 cache hits. Figure 6 shows normalized L2 cache hits for `Cpair`, `Cpair+Mpache`, `CacheC`, and `Exhaustive`. `Cpair` improves cache hits for kernel pairs with matching cache patterns, further enhanced by `Cbypass`. In contrast, `Mpache` yields fewer hits due to its L0/1 bypass strategy, which reduces measurable L2 cache activity.

4.5 Extensive Studies

In this section, we analyze the stability of `CacheC` and its extension to support multiple kernels. We also discuss `CacheC` as applied to other platforms and its associated analysis overhead.

Stability. Figure 7 presents the standard deviation of L2-L0/1 affinity score differences for optimal kernel pairs across 10 repeated generations by the LLM. While the average deviation remains low (<0.3), maximum values are higher due to cache-insensitive loads. These fluctuations have a negligible impact on cache performance (Section 4.2).

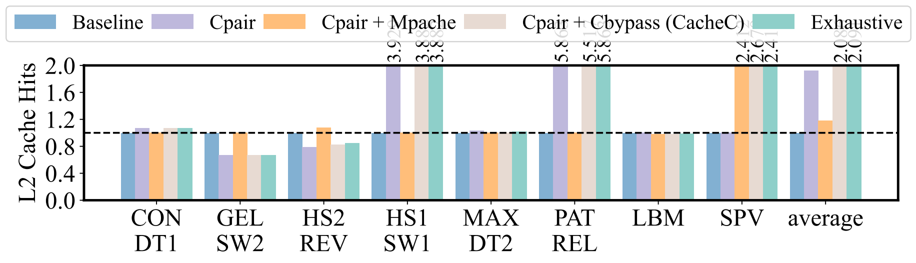


Fig. 6: L2 Cache hits of `Cpair`, `Cpair+Mpache`, `CacheC` and `Exhaustive` schemes, normalized to baseline.

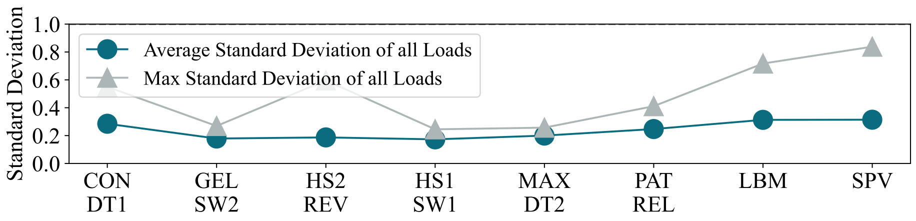


Fig. 7: Standard deviation of the difference between L2 and L0/1 affinity scores for concurrent kernel pairs.

Multiple Kernels. We extend `CacheC` to support multi-kernel concurrency, evaluating three-kernel execution scenarios. We deploy `CacheC` across four kernel groups. As demonstrated in Figure 8, concurrently executing a well-selected kernel pair consistently achieves higher performance gains than running three kernels. These results justify our focus on two-kernel optimization in this work.

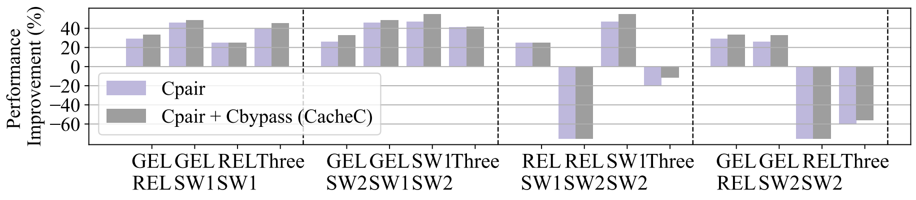


Fig. 8: Performance gains of `Cpair` and `CacheC` over `Baseline` for three-kernel execution and all two-kernel combinations from the three kernels.

Platform and Overhead. We implemented `CacheC` on AMD's platform for its open-source support and cache annotation capabilities. While NVIDIA's PTX annotations serve only as performance hints [34], `CacheC` could be adapted if their effects are reliably ensured. Additionally, compared to traditional offline cache analysis and model training, iterative dialogue operates in a more adaptive and incremental manner, which inherently reduces redundant computations. This approach mitigates the excessive overhead typically associated with large-scale precomputed models, making it a promising alternative.

5 Related Work

Kernel Concurrency and Schedule. Prior works address kernel scheduling via machine learning [4], static QoS-aware frameworks [35], and multi-objective heuristics [13]. However, these methods largely overlook cache contention, a gap addressed in this paper.

Cache Management. Efficient GPU cache management has been explored through ML-based schemes for hit prediction [36], two-level bypassing strategies [37], and multi-level load interaction analyses [16]. While prior work targets single-kernel patterns, this paper leverages LLMs to optimize inter-kernel cache behaviors.

6 Conclusion

`CacheC` is a cache management framework for concurrent kernels, achieving up to 19.67% lower turnaround, 24.48% higher throughput, and 1.337× speedup.

Acknowledgements. We are grateful to the anonymous reviewers for their helpful suggestions. This work is supported by Guangdong S&T Program under Grant No. 2024B0101040005, National Natural Science Foundation of China-#62472462/#62402534/#62461146204.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. https://www.amd.com/zh-cn/products/graphics/desktops/radeon/7000-series/amd-radeon-rx-7900xtx.html
2. https://www.amd.com/zh-cn/products/graphics/desktops/radeon/6000-series/amd-radeon-rx-6900-xt.html
3. Pai, S., et al.: Improving GPGPU concurrency with elastic kernels. ACM SIGARCH Computer Architecture News (2013)
4. Wen, Y., et al.: Merge or Separate?: Multi-job Scheduling for OpenCL Kernels on CPU/GPU Platforms. In: Proceedings of the General Purpose GPUs (2017)
5. Dai, H., et al.: Accelerate GPU Concurrent Kernel Execution by Mitigating Memory Pipeline Stalls. In: 2018 HPCA (2018)
6. Wu, W., et al.: TurboMGNN: Improving Concurrent GNN Training Tasks on GPU With Fine-Grained Kernel Fusion (2023)
7. Jiao, Q., et al.: Improving GPGPU energy-efficiency through concurrent kernel execution and DVFS. In: 2015 CGO (2015)
8. Yu, L., et al.: Moka: Model-based concurrent kernel analysis. In: 2017 IEEE International Symposium on Workload Characterization (IISWC) (2017)
9. Wen, Y., et al.: MaxPair: Enhance OpenCL Concurrent Kernel Execution by Weighted Maximum Matching (2018)
10. Shekofteh, S.K., et al.: ccuda: Effective co-scheduling of concurrent kernels on gpus. IEEE Transactions on Parallel and Distributed Systems (2020)

11. López-Albelda, B., et al.: Flexsched: Efficient scheduling techniques for concurrent kernel execution on gpus. J. Supercomput. (2022)
12. Shobaki, G., et al.: Optimizing occupancy and ILP on the GPU using a combinatorial approach. In: CGO (2020)
13. Alizadeh, N.B., et al.: Multi-Objective Concurrent Kernel Scheduling for Multi-GPU Systems. In: 2024 ICEE (2024)
14. Adufu, T., et al.: L2 Cache Access Pattern Analysis using Static Profiling of an Application. In: 2023 COMPSAC (2023)
15. Adufu, T., et al.: Optimizing performance using gpu cache data residency based on application's access patterns. In: 2023 APNOMS
16. Xi, M., et al.: Mpache: Interaction aware multi-level cache bypassing on gpus. In: ASPDAC 2025 (2025)
17. Jadidi, A., et al.: Selective Caching: Avoiding Performance Valleys in Massively Parallel Architectures (2020)
18. Do, C.T., et al.: Aggressive GPU cache bypassing with monolithic 3D-based NoC. The Journal of Supercomputing (2023)
19. Xu, X., et al.: ATA-Cache: Contention Mitigation for GPU Shared L1 Cache With Aggregated Tag Array (2024)
20. Hadi, M.U., et al.: A survey on large language models: Applications, challenges, limitations, and practical usage. Authorea Preprints (2023)
21. Zhai, Q., Zhang, Z., Xiao, R.: Llm based end-to-end branch predictor optimization generator. In: ASAP (2024)
22. Long, Y., et al.: Discuss Before Moving: Visual Language Navigation via Multi-expert Discussions (Sep 2023)
23. Sahoo, P., et al.: A systematic survey of prompt engineering in large language models: Techniques and applications. arXiv preprint arXiv:2402.07927 (2024)
24. AMD RDNA Whitepaper. `https://www.amd.com/system/files/documents/rdna-whitepaper.pdf`
25. NVIDIA Ampere GPU Architecture Tuning Guide. `https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html`
26. Syntax of AMDGPU Instruction Modifiers. `https://llvm.org/docs/AMDGPUModifierSyntax.html`
27. Jin, Z., et al.: A benchmark suite for improving performance portability of the sycl programming model. In: 2023 ISPASS (2023)
28. Che, S., et al.: Rodinia: A benchmark suite for heterogeneous computing. In: 2009 IEEE International Symposium on Workload Characterization (IISWC) (2009)
29. Stratton, J.A., Rodrigues, C.I., Sung, I.J., Obeid, N., Chang, L.W., Anssari, N., Liu, G.D., mei W. Hwu, W.: Parboil: A revised benchmark suite for scientific and commercial throughput computing (2012)
30. Hello GPT-4o. https://openai.com/index/hello-gpt-4o/
31. HIP Runtime API Reference: Stream Management
32. NVIDIA CUDA Toolkit. `https://developer.nvidia.com/cuda-toolkit`
33. rocm_smi_lib. https://github.com/ROCm/rocm_smi_lib
34. NVIDIA PTX ISA 8.3. https://docs.nvidia.com/cuda/parallel-thread-execution//#cache-operators
35. Zhao, H., et al.: Tacker: Tensor-CUDA Core Kernel Fusion for Improving the GPU Utilization while Ensuring QoS. In: 2022 HPCA (2022)
36. Sun, H., et al.: Ncache: A machine-learning cache management scheme for computational ssds. TCAD (2023)
37. Kim, et al.: New Two-Level L1 Data Cache Bypassing Technique for High Performance GPUs. Journal of Information Processing Systems (2021)