Mpache: Interaction Aware Multi-level Cache Bypassing on GPUs

Mengyue Xi, Tianyu Guo, Xuanteng Huang, Zejia Lin, Xianwei Zhang[#]
Sun Yat-sen University, Guangzhou, China
{ximy,guoty9,huangxt57,linzj39}@mail2.sysu.edu.cn,zhangxw79@mail.sysu.edu.cn

ABSTRACT

Graphics Processing Units (GPUs) are essential for general-purpose applications and are commonly leveraging multi-level caches to alleviate memory access pressure. However, the default cache management may lose opportunities for optimal performance in different applications. Although existing cache bypassing techniques tend to address this challenge, these methods predominantly concentrate on single-level cache, thus restricting their potential for further enhancements. To mitigate this issue, we propose `Mpache`, a novel software-based mechanism designed to bypass multi-level caches based on the characterization of load instructions. `Mpache` constructs an interaction graph and analyzes the cooperation and contention among instructions. Then, the profiling data of bypassing effectiveness guides `Mpache` to select the appropriate cache levels to bypass for each instruction. Finally, the design is integrated into the compiler to enable automatic bypassing for existing workloads. Evaluations on off-the-shelf GPUs show that `Mpache` achieves an average 1.15× speedup over the default cache policy, and effectively outperforms prior arts.

KEYWORDS

GPUs, Cache bypassing, Multi-level caches, Compiler, Load interaction

ACM Reference Format:
Mengyue Xi, Tianyu Guo, Xuanteng Huang, Zejia Lin, Xianwei Zhang[#]. 2025. Mpache: Interaction Aware Multi-level Cache Bypassing on GPUs. In *30th Asia and South Pacific Design Automation Conference (ASPDAC '25), January 20–23, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3658617.3697568

1 INTRODUCTION

Graphics Processing Units (GPUs) show remarkably high computational throughput owing to their thousands of threads, coupled with efficient thread-switching techniques to hide memory request latency. To further alleviate the bottleneck of slow global memory operations, modern GPUs consistently expand cache size to exploit data reuse at different degrees of popularity [1]. By default, requests fetch data from global memory and store retrieved data in all cache levels. Tasks with regular access patterns demonstrably benefit from this approach as contiguous elements are cached together within a single memory access. In contrast, workloads characterized by non-contiguous memory accesses exhibit minimal performance improvement [2, 3] due to the limited data locality. Caching all the data in this scenario leads to cache pollution, wasting valuable capacity on unnecessary data.

Cache bypassing has been proved as a promising strategy to mitigate cache contention caused by scattered memory accesses [4–6]. This approach strategically bypasses the cache for these accesses, preventing storing infrequently used data and thus improving overall cache utilization. However, the effectiveness of cache bypassing varies depending on two key factors. Firstly, the memory instructions exhibit varied bypassing affinity, indicating the varying efficiency and benefits of bypassing across different cache levels (on-chip L0/L1 or off-chip L2). Bypassing a specific level may yield significant speedup, but can also negate the performance, hinging on the access pattern. Furthermore, the interaction between load requests from diverse code regions and threads is crucial in determining bypass affinity.

To bypass cache in a controlled manner, a plethora of designs have been proposed to dynamically detect cache misses from memory requests [6–12] and apply thread throttling to restrict multiple threads from simultaneously loading data into cache [13–15]. While offering fine-grained control, most designs concentrate on architectural modifications and thus pose a challenge for implementing on readily available GPUs. Software-managed approaches, on the contrary, analyze data locality at the instruction level with a particular focus on code property [16–22], thereby offering strong compatibility for off-the-shelf GPUs. However, these strategies focus exclusively on the single-level cache, especially the L1 cache, and fail to coordinate across multiple cache levels. This may result in overlooking the bypass potential for different cache hierarchies.

Aiming at bypassing multi-level caches on off-the-shelf GPUs, we propose `Mpache`, a novel system to analyze the bypassing affinity for global loads based on their interactions. `Mpache` first constructs an interaction graph and divides global loads into multiple groups. This graph helps identify the potential data reuse or cache competition patterns inter- and intra-groups, which can be summarized into two types of interactions: cooperation and contention. Then via profiling, `Mpache` acquires the multi-level cache bypassing affinity for each load and its corresponding group. Next, by synergizing the two interactions, `Mpache` formulates a bypass strategy catering to the whole group for cooperation consideration and each load within a group for contention concern. Finally, combining the instruction interactions, bypassing affinities, and a calculated bypassing score, `Mpache` strategically selects the optimal bypass decision for each load. The design is further seamlessly integrated into the compiler to automate the entire process without any manual effort.

In summary, this paper makes the following contributions:

[#]Corresponding author.

- We identify the bypassing affinity to different cache levels and highlight the lack of control in multi-level cache bypassing in existing designs.
- We propose Mpache to analyze two interactions and bypass efficiency across various cache levels.
- We implement the design on top of the compiler to automatically select the suitable strategy.
- Evaluations demonstrate that Mpache achieves a 1.15× performance speedup over default policy.

2 BACKGROUND AND MOTIVATION

2.1 GPU Memory Hierarchy

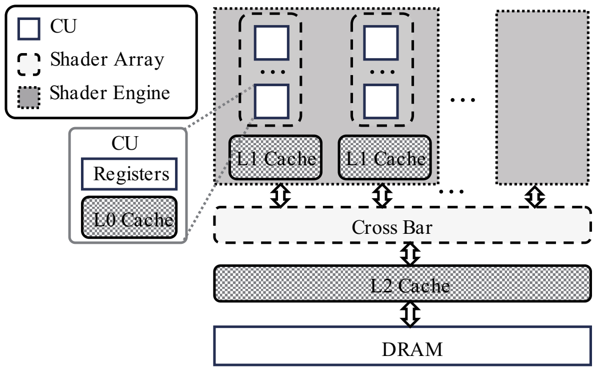


Figure 1: A general purpose GPU memory hierarchy. Arrows indicate data-flow paths.

Figure 1 depicts the GPU memory hierarchy along with possible data flow paths among these components, grounded in the recent AMD RDNA design[1] [23]. From the architectural aspects, a GPU consists of several compute units (CUs), also referred to as streaming multiprocessors (SMs). An assembly of CUs is grouped as shader arrays (SA), and a bundle of SAs is further accommodated by shader engines (SE). In terms of the memory hierarchy, each CU is equipped with private registers and L0 cache. An SA shares access to an L1 cache and all CUs of the GPU utilize a globally shared L2 cache, which is directly connected to the main DRAM memory.

2.2 Bypassing Software Support

To effectively manage the cache space for divergent workloads, software-controlled mechanisms have been proposed, with representative examples of residency control [24] and flexible policy tuning [25]. Recent AMD RDNA architectures introduce three optional annotation bits (SLC, GLC, and DLC) within memory instructions to affect data coherency and cache access behavior [23]. Correspondingly, the LLVM compiler infrastructure [25] provides Intermediate Representation (IR) features `volatile` and `nontemporal` to manipulate the annotation bits for the AMDGPU backend.

Table 1 lists the correspondence between the LLVM IR features and the annotation bits. Setting the GLC, DLC, and SLC bits adopt Miss-Evict, Miss-Evict, and Stream policies for different cache levels

[1]The paper elaborates primarily using AMD GPU architectures and terms, which are generally applicable to other vendors as well.

Table 1: The relationships between LLVM IR instructions features and RDNA2 ISA machine bits and the corresponding cache behaviors for global loads instructions.

	LLVM IR Feature	Instruction Annotation Bits	Cache Behaviors		
	volatile/ nontemporal	GLC/DLC/SLC	L0	L1	L2
	False/False	0/0/0	Cache	Cache	Cache
★	False/True	0/0/1	Cache	Cache	Stream
▲	True/False	1/1/0	Miss Evict	Miss Evict	Cache
	True/True	1/1/0	Miss Evict	Miss Evict	Cache

Bypassing Modes: ★ L2 Bypassing ▲ L0/1 Bypassing

respectively. This effectively reduces the caching of recent data and thus serves as an analogous substitution for the respective bypassing policies. Correspondingly, at the LLVM IR level, the `volatile` feature jointly controls the cache behavior of L0 and L1 on-chip caches, which is equivalent to setting GLC and DLC. The `nontemporal` feature affects the L2 off-chip cache, managed by the SLC bit, and can be overridden by `volatile`. In summary, the combinations of the `volatile` and `nontemporal` establish two bypassing modes.

- `L2 Bypassing`: set the LLVM feature `volatile` to be false and `nontemporal` to be true, corresponding to the second line of Table 1, which bypasses L2 cache only.
- `L0/1 Bypassing`: set the LLVM feature `volatile` to be true, corresponding to the third and fourth lines of Table 1, bypassing both L0 and L1 jointly.

2.3 Motivation

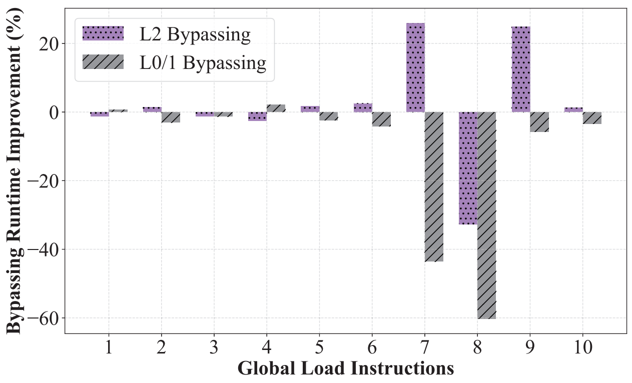


Figure 2: The percentage of runtime improvement of *spmv* for bypassing each load under `L2 Bypassing` mode and `L0/1 Bypassing` mode.

We illustrate the varying bypassing affinities for different cache levels of distinct load instructions using application *spmv* from

Parboil Suite [26]. Figure 2 depicts the performance improvement observed when bypassing each load instruction under two bypassing modes. The application *spmv* includes 10 global loads in total, each of which is indexed according to the IR order, denoted as $load_i$. An analysis of bypassing affinity across different cache levels reveals distinct behaviors for various load instructions. Specifically, $load_7$ and $load_9$ have a strong L2 bypassing affinity (25.94% and 24.85% respectively) but a substantial negative L0/1 bypassing affinity (-43.58% and -5.72% respectively). In contrast, $load_4$ demonstrates a moderate L0/1 bypassing affinity (1.45%), the only positive value among all loads, but a weaker affinity for L2 bypassing (-3.06%). The remaining loads exhibit negligible bypassing affinity at both L0/L1 and L2 levels and in some cases (e.g. $load_8$), bypassing significantly degrades performance.

The observations highlight the importance of careful decision when selecting instructions and cache levels for bypassing strategies. Therefore, fine-grained cache control over load instruction for multiple cache levels is urgently demanded to exploit the diverse bypassing affinities.

3 MULTI-LEVEL CACHE BYPASSING DESIGN

3.1 Overview

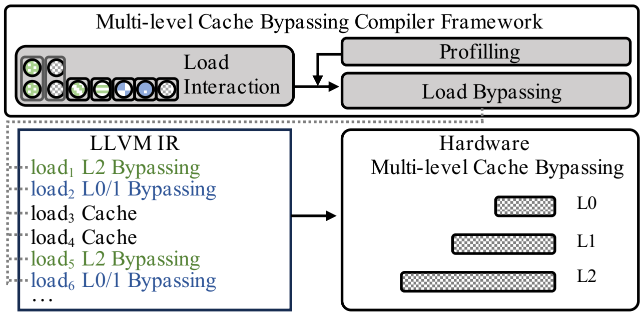


Figure 3: The proposed Mpache framework overview.

Figure 3 shows the overview of Mpache. At the compilation time, for load interaction analysis, Mpache first acquires an interaction graph that decides the loads into several groups. Then, for each group and each instruction within a group, the framework profiles the performance of the program under two bypassing modes and collects the profiling data for different bypassing affinities investigation. Finally, Mpache chooses the ideal bypassing mode for each load instruction, considering the interactions among loads and bypassing affinities of each load through the bypassing strategy.

3.2 Load Interaction

We construct an interaction graph to divide the load instructions featuring inter-instruction locality into the same group. As in Figure 4, for a GPU kernel with N global loads, we use ld_i to denote the i-th global load in the LLVM IR code order. The presented approach analyzes the IR code of GPU kernels and constructs a load graph that involves all the load instructions in this kernel. We first describe each load instruction as a node in the graph (①). Next, the approach divides the graph into multiple groups (②), where

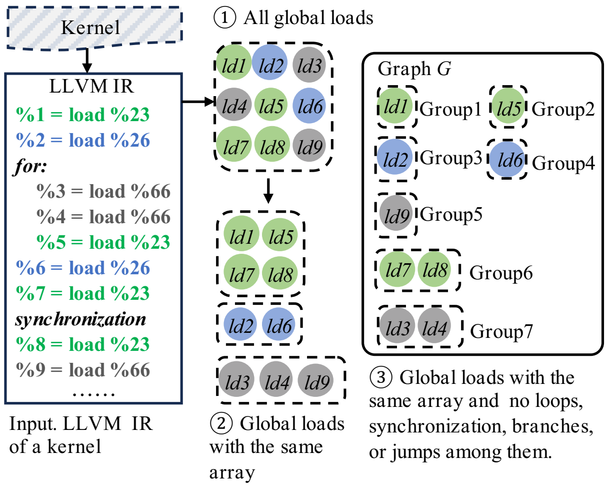


Figure 4: Load interaction graph construction.

each node in a group is referenced to the same array. Subsequently, the group is further divided into more localized sub-groups (③) devoid of control flow instructions, such as loops, branches, jumps, or synchronization primitives. In this example, the load instructions are partitioned into 7 groups. Finally, we build an interaction graph $G = (V, E)$ including multiple groups annotated as g. The node $v_i \in V$ represents ld_i. Nodes v_i and v_j are connected via an indirect edge if they are in a g.

Loads within the same group clearly demonstrate notable spatial and temporal locality, stemming from their shared array references and execution within uninterrupted basic blocks without control flows or synchronizations. Nonetheless, contention among these loads has been observed. When the reuse distances between two loads are excessively long or their memory footprints do not precisely align with cache line boundaries, they may share the same cache line without effectively reusing cached data. This scenario results in cache line eviction and subsequent refreshes, which adversely affect overall cache performance. Hence, we summarize the two types of instruction interactions within the same group g as follows and simultaneously incorporate both features in the multi-level cache bypassing strategy in Section 3.3.

- **cooperation**: the load instructions in the same group reuse data from each other, so we should set the bypass policy for them uniformly.
- **contention**: due to the different reuse distances of the load instructions in a group, they will also compete for the cache lines since they may extremely be allocated to the same cache lines. Hence we should also be concerned about the single bypass effect within a group.

3.3 Load Bypassing

In this section, we identify the optimal bypass modes for each load based on bypassing affinities and load interactions. As outlined in Section 3.2, loads within a group exhibit two interaction types: cooperation and contention. When cooperation interactions are dominant, loads within a group tend to have similar access patterns and high reuse potential, indicating shared bypassing preferences. Conversely, if contention prevails, bypass modes should be tailored for individual loads rather than the group. An edge case may arise where a group collectively disfavors a bypassing mode, while individual loads within it prefer the same mode; here, we balance the group's overall impact on each load's bypass choice.

To represent the bypassing affinity for a specific mode, we calculate its affinity score, noted as $afft\_score$. Equation 1 details this calculation. For each $g \in G$ and $v \in V$, we individually bypass them under a specific mode, recording the $bypassing\_time$. The affinity score is then the percentage improvement over the $baseline\_time$, where a higher score indicates a stronger affinity for that mode.

$$afft\_score = \frac{(baseline\_time - bypassing\_time)}{baseline\_time} * 100 \quad (1)$$

Algorithm 1 outlines the strategy to determine the optimal bypass modes for each global load v. Here, $modes$ is a one-dimensional boolean array where 0 denotes `L2 Bypassing` and 1 denotes `L0/1 Bypassing`, extendable for additional modes if supported by hardware. $afft\_scores$ is a two-dimensional array where $afft\_scores$ (g, j) gives the affinity score of $g \in G$ under $modes(j)$. The outputs are $nodes$, a one-dimensional array indicating the index of each bypassed $v \in V$, and $policies$, another one-dimensional array specifying the bypass mode for each corresponding $node$.

Algorithm 1 Load Bypassing Modes Selection Strategy

```
Input:
    $G(V, E)$: the load interaction graph;
    $modes$: an array consisting of [0,1];
    $afft\_scores\ (g, j)$: represents the affinity score of $g \in G$ under
    $modes(j)$.
Output:
    $nodes(i)$: represents the index of the bypassing $v \in V$ at $i$;
    $policies(i)$: represents the bypassing mode of $nodes$ at $i$.
1:  for each $g \in G$ do                              ▷ cooperation interaction
2:      initialize $combi(g)$ to $-\infty$
3:      for each $m \in modes$ do                      ▷ diverse affinities
4:          $com = afft\_scores(g, m) - \sum_{v \in g} afft\_scores(v, m)$
5:          $combi\_record(g, m) = com$
6:          if $com \geq thresh_1 * access(g)$
              ↪ and $com > combi(g)$ then
7:              $combi(g) = com$; $nodes \leftarrow$ add each $v \in g$
8:              $policies \leftarrow$ update mode of each $v \in g$ to $m$
9:          end if
10:     end for
11:     if $combi(g) \neq -\infty$ then continue end if
12:     for each $v \in g$ do                          ▷ contention interaction
13:         initialize $score(v)$ to $-\infty$
14:         for each $m \in modes$ do                  ▷ diverse affinities
15:             $sco = afft\_scores(v, m)$
16:             if $len(g) \neq 1$ and $combi(g, m) \leq 0$ then
17:                 $sco = sco + weight * combi\_record(g, m)$
18:             end if
19:             if $sco \geq thresh_2 * access(v)$
              ↪ and $sco \geq score(v)$ then
20:                 $score(v) = sco$; $nodes \leftarrow$ add $v$
21:                 $policies \leftarrow$ update mode of $v$ to $m$
22:             end if
23:         end for
24:     end for
25: end for
```

Table 2: List of evaluated GPU benchmark kernels[26–28].

Kernels	*abbr.*	Kernels	*abbr.*
spmv	SPV	lbm	LBM
hybridsort-1	HS1	paticlefilter	PAT
hybridsort-2	HS2	dct8x8-1	DT1
convolutionSeparable-1	CS1	dct8x8-2	DT2
convolutionSeparable-2	CS2		

For cooperation interaction, each group is treated as a unit. The algorithm first calculates the group's combined locality impact, com, under two modes, as shown in line 4. This impact is measured by subtracting the aggregate individual load affinities from the group's bypassing affinity, $afft\_score\ (g, m)$, for each mode m. A threshold $thresh_1$ is then applied to assess the significance of the group's locality. If the locality impact exceeds $thresh_1$ times the access count of g, the algorithm selects the higher affinity mode, adds each $v \in g$ to $nodes$, and updates $policies$ (line 6 ∼ 9).

Under contention interaction, if $combi(g)$ is infinite, indicating insufficient locality in either mode (line 11), the algorithm applies individual bypass strategies for each v in the group (line 12). The equation in line 17 calculates v's bypass affinity, weighted by group affinity using $weight$ to moderate extreme aversions. If v's affinity score sco exceeds $thresh_2$ times its access count, v is added to $nodes$, and its $policies$ mode is updated (line 19 ∼ 22). This produces $nodes$ and $policies$, representing all bypassing $v \in V$ and their modes. For thresholds, we set $thresh_1$ at 1.5 for groups and $thresh_2$ at 1.0 for individual loads, assuming greater benefits from group bypassing, and apply a $weight$ of 0.1 to balance group impact.

4 EXPERIEMENTAL EVALUATION

For evaluation, we implement `Mpache` at compile time atop of the commodity AMD Radeon RX 6900 XT which uses RDNA2 architecture with GFX1030 ISA. The device contains 32KB per CU L0, 128KB per SA L1, and a 4MB global L2 cache. The proposed `Mpache` is implemented as an LLVM pass during compilation and acts as a transparent middleware between the underlying hardware and the upper-level users, thus requiring no application code refactoring. The applications used in the experiment are described in Table 2. We evaluate and compare the following cache bypassing schemes:

- **`CacheAll`**: baseline that defaults to caching all cache levels per load.
- **`PassL2`** and **`PassL0/1`**: bypassing all loads exclusively under the non-temporal mode and the volatile mode, separately.
- **`Liang`** [17]: the state-of-the-art method that implements both static and dynamic cache bypassing in a simulator. We reproduce the method's static design on readily available GPUs.
- **`SelectL2`** and **`SelectL0/1`**: the proposed bypassing strategy outlined in Section 3.3 involves selecting bypassing loads exclusively under the `L2 Bypassing` mode and the `L0/1 Bypassing` mode separately.
- **`Mpache`**: the proposed bypassing framework for selecting bypassing loads under the combination of two modes.

4.1 Performance Improvement

Figure 5 shows the runtime speedup of all cache management systems, normalized to the baseline `CacheAll`. Across benchmarks, `Mpache` achieves an average 1.152× speedup over `CacheAll`, offering 6% additional performance gains over `Liang`. This improvement stems from `Mpache`'s consideration of load interactions and multi-level bypassing affinities per load. For workloads like *SPV* and *HS1*, `Mpache` outperforms `Liang` by utilizing both `L2 Bypassing` and `L0/1 Bypassing` modes, whereas `Liang` only uses `L0/1 Bypassing`. Additionally, in workloads such as *LBM*, *DT1*, and *DT2*, `Mpache` excels by recognizing cooperation within load groups and bypassing the entire group under an optimal mode, which `Liang` lacks by only handling individual loads.

Bypassing the L0/L1 (`PassL0/1`) or L2 (`PassL2`) cache for all loads results in average performance drops of 0.713× and 0.901×, respectively, due to the elimination of potential data reuse. `PassL0/1` benefits applications like *LBM*, *DT1*, *DT2*, and *HS2*, where loads are only used once in L0/L1. However, this approach degrades performance in others, especially in *CS1* and *CS2*. By applying our instruction-aware single-level strategies, `SelectL0/1` and `SelectL2`, we achieve moderate average speedups of 1.097× and 1.040×, effectively selecting optimal bypass modes across applications.

Furthermore, the proposed `Mpache` surpasses conventional single-level bypassing techniques, `SelectL2` and `SelectL0/1`, by integrating multi-level cache bypassing decisions, achieving optimal bypassing selection. `Mpache` identifies load affinities for different cache bypassing within a kernel and across kernels. Specifically, for *SPV* that exhibit affinities for L0, L1, and L2 bypassing, `Mpache` determines the optimal strategy that combines the two modes. Conversely, for applications like *LBM* and *HS1*, which primarily exhibit either L0/L1 or L2 bypassing affinities individually, `Mpache` identifies the most suitable strategy for each application.

4.2 Cache Hits

To collect hardware utilization, we leverage the rocm-smi library to collect metrics from hardware counters. This allows us to monitor L2 cache hits and validate the effectiveness of our design. Figure 6 illustrates the normalized global L2 cache hits for all cache management strategies, normalized against `CacheAll`. Due to space constraints, the figure presents mean results across all kernels and highlights representative applications. Our proposed `Mpache` increases L2 cache hits in *SPV*, *PAT*, *CS1*, *CS2*, and *HS1*, whereas for *LBM*, *DT1*, *DT2*, and *HS2*, the sum remains the same or indicates a minor decline. This accounts for the bypassing strategy for *LBM*, *DT1*, *DT2*, and *HS2*, in which all loads are bypassed for L0 and L1 cache inferring that the cumulative L2 cache hits are not intrinsically tied to them. Conversely, for *SPV*, *PAT*, *CS1*, *CS2*, and *HS1*, ideal loads for the L0 and L1 cache or L2 cache are partially bypassed. This effectively reduces contention and frees up cache space for other loads, thus improving the sum of L2 cache hits to varying degrees based on the changing affinities for L2.

Among all Cache management, `PassL0/1` stands out due to bypassing all loads for L0/1, which leads to increased memory accesses to the L2 cache, thus boosting L2 cache hits. Similarly, `SelectL0/1` and `Liang` also show increased L2 hits as they bypass only under `L0/1 Bypassing` mode. Contrary to intuition, `PassL2` also shows an increase in L2 hits, despite theoretically bypassing all loads under `L2 Bypassing` mode. This phenomenon occurs because the hardware implementation of the L2 cache operates on a streaming model rather than direct bypassing. For the same reason, `SelectL2` gets an increase in L2 hits.

4.3 Sensitivity Studies

There are three primary tuning knobs in `Mpache`'s design (Section 3.3): the bypassing thresholds $thresh_1$ and $thresh_2$, which determine the selection of group bypassing and load bypassing based on their bypassing affinity, and *weight*, which captures the impact of group bypassing affinity on individual loads within the group. Figure 7(a) presents the average performance achieved by all applications as $thresh_1$ and $thresh_2$ vary from 1.0 to 2.0. It is evident that from 1.0 to approximately 1.5, the performance exhibits minor variations, indicating stability for both $thresh_1$ and $thresh_2$. However, performance degradation becomes noticeable for $thresh_1$ around 1.6 and diminishes near 1.8. Similarly, for $thresh_2$, performance begins to decline around 1.8 and stabilizes near 2.0. This highlights that the group bypassing threshold demonstrates greater robustness compared to the load bypassing threshold.

Figure 7(b) illustrates the average speedup variation as *weight* ranges from 0.0 to 0.5. It can be observed that from 0.1 to 0.5, the performance remains stable, indicating the robustness of our design. However, when *weight* is set to zero, there is a noticeable decline in performance. This occurs because the influence of group affinity on load instructions within a group becomes negligible. Specifically, for applications like *CS1* and *CS2*, certain groups strongly dislike `L0/1 Bypassing` while the load instructions within these groups exhibit a strong affinity. This exceptional case aligns with the discussion presented in Section 3.3. Therefore, setting *weight* greater than zero is essential to uphold the integrity of our design and facilitate performance optimization.

5 DISCUSSION

Current LLVM IR features lack comprehensive support for bypassing all cache levels (L0, L1, and L2). Consequently, due to the software-controlled mechanism, we are currently unable to observe its effects. Nevertheless, should this feature become accessible, the multi-level bypassing strategy outlined in Section 3.3 can seamlessly accommodate any number of mode selections without necessitating modifications.

We implement `Mpache` on AMD's platform since the cache annotation bits have been experimentally verified to be effective, but it is also feasible on the popular NVIDIA's products. NVIDIA exposes the cache management mechanisms to annotate PTX instructions, but these annotations are only treated as performance hints [29]. *ld.global.ca* sets the default cache policy for L1 and L2 caches, *ld.global.cg* is used specifically to bypass the L1 cache, while *ld.global.cs* is designed for bypassing both the L1 and L2 caches. Cache bypassing can be facilitated through PTX annotations, provided their effects are established. Consequently, based on these cache operators, `Mpache` can also be deployed at the PTX level on Nvidia's platform during compile time.

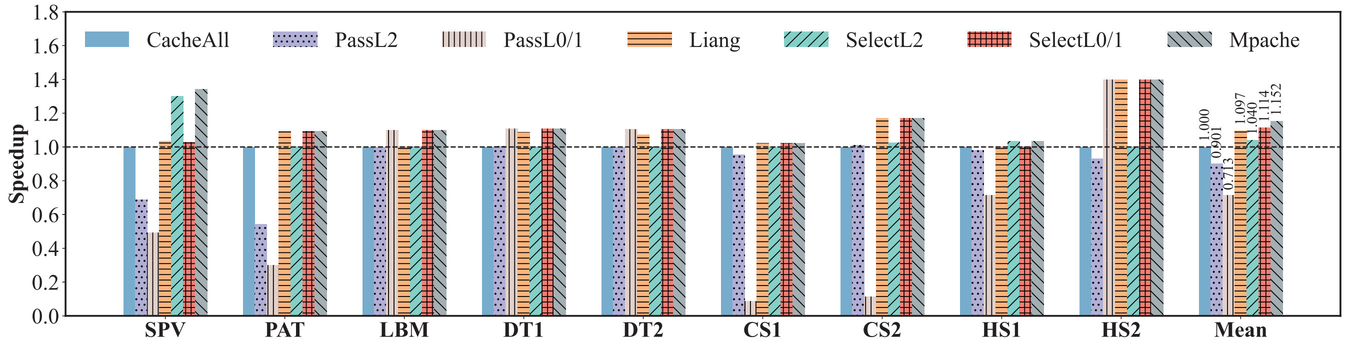


Figure 5: The speedup comparison of all the cache management systems which includes bypassing all loads only in `L2 Bypassing` (`PassL2`) or `L0/1 Bypassing` (`PassL0/1`), `Liang` [17], selective load bypassing strategies as detailed in Section 3.3 only under `L2 Bypassing` (`SelectL2`) or `L0/1 Bypassing` (`SelectL0/1`), and `Mpache`, normalized to CacheAll.

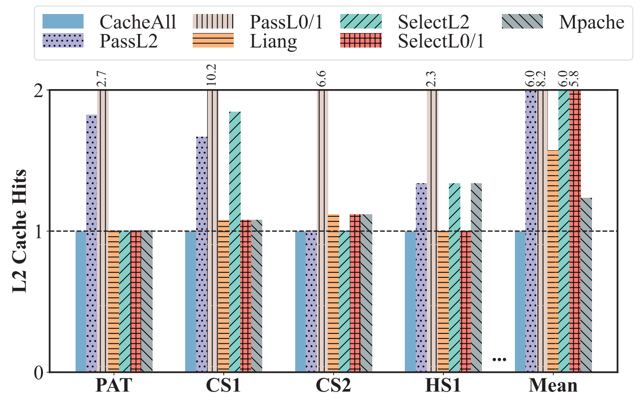
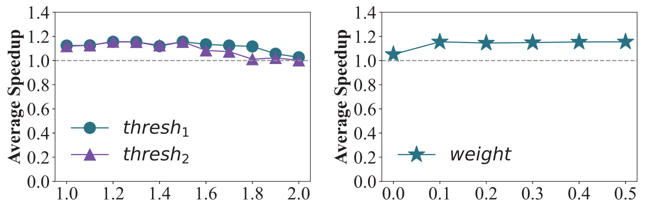


Figure 6: The global L2 cache hits comparison of all the cache management normalized to CacheAll, displaying mean results across all kernels and selected representative kernels.



(a) Bypassing thresholds *thresh*. **(b) Group impact *weight*.**

Figure 7: The average speedup of different thresholds and weights parameters.

6 RELATED WORK

Instruction-based bypassing. It performs bypassing based on analyzing the per-load instruction, which concentrates on code characteristics and is typically examined during compilation. Jia [4] determined whether the cache is on or off through the memory traffic of each load instruction. Xie [20] and Liang [17] encoded the classification into instructions at compiler time while adjusting the ratio of thread blocks. Fang [16] considered the use of different types of locality for load instructions. Those prior arts neglected the L2 cache and only focused on the L1 cache bypassing. Whereas working at the instruction level, this paper combines all cache levels and designs an effective strategy at compiler time.

Thread throttling bypassing. Enforcing the number of threads or warps that some threads access the cache but others bypass significantly mitigates the pressure on the cache. There are some studies made efforts into it. Li [15] added a threshold during compilation so that only a limited number of threads based on warp could access the cache for L1, L2, and read-only caches. Jadidi [13] presented selective caching mechanisms regulating the number of threads to both the L1 and L2 caches.

Memory request bypassing. Memory request level bypassing triggers bypassing when cache associativity stall is encountered. The orchestrated hardware modifications are often used together with cache bypassing revised design. Recent works have been devoted to memory access request-based bypassing. Kim [8] introduced a new two-level bypassing method determining cache access based on two metrics. Do [7] proposed the warp classification and the request bypassing structure to mitigate the GPU cache contention.

7 CONCLUSION

This paper highlights the diverse bypass affinity demonstrated by various memory load instructions and proposes `Mpache` to bypass multi-level caches on off-the-shelf GPUs. `Mpache` analyzes the interaction between instructions and strategically selects the cache level to bypass for each load. Moreover, `Mpache` is integrated into the compiler framework to facilitate automatic cache bypassing optimization. Experimental evaluations show that `Mpache` outperforms the default cache policy by 1.15×.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their constructive comments and suggestions. This research was supported by the National Key R&D Program of China under Grant NO. 2023YFB3002202, the National Natural Science Foundation of China-#62102465/#62472462.

REFERENCES

[1] AMD Instinct Accelerators. https://www.amd.com/en/graphics/instinct-server-accelerators.

[2] Tianao Ge, Tong Zhang, and Hongyuan Liu. ngap: Non-blocking large-scale automata processing on gpus. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 268–285, 2024.

[3] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S Emer, Mark A Horowitz, and Fredrik Kjølstad. The sparse abstract machine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 710–726, 2023.

[4] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. Characterizing and improving the use of demand-fetched caches in GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing*, pages 15–24, San Servolo Island, Venice Italy, June 2012. ACM.

[5] Xuhao Chen, Shengzhao Wu, Li-Wen Chang, Wei-Sheng Huang, Carl Pearson, Zhiying Wang, and Wen-Mei W. Hwu. Adaptive Cache Bypass and Insertion for Many-core Accelerators. In *Proceedings of International Workshop on Manycore Embedded Systems*, pages 1–8, Minneapolis MN USA, June 2014. ACM.

[6] Kyoshin Choo, David Troendle, Esraa A. Gad, and Byunghyun Jang. Contention-Aware Selective Caching to Mitigate Intra-Warp Contention on GPUs. In *2017 16th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 1–8, Innsbruck, July 2017. IEEE.

[7] Cong Thuan Do, Cheol Hong Kim, and Sung Woo Chung. Aggressive GPU cache bypassing with monolithic 3D-based NoC. *The Journal of Supercomputing*, 79(5):5421–5442, March 2023.

[8] Gwang Bok Kim and Cheol Hong Kim. New Two-Level L1 Data Cache Bypassing Technique for High Performance GPUs. *Journal of Information Processing Systems*, 17(1):51–62, February 2021.

[9] Hyunjun Kim, Sungin Hong, Hyeonsu Lee, Euiseong Seo, and Hwansoo Han. Compiler-Assisted GPU Thread Throttling for Reduced Cache Contention. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, Kyoto Japan, August 2019. ACM.

[10] Cong Thuan Do, Jong Myon Kim, and Cheol Hong Kim. Application Characteristics-Aware Sporadic Cache Bypassing for high performance GPGPUs. *Journal of Parallel and Distributed Computing*, 122:238–250, December 2018.

[11] Cong Thuan Do, Jong Myon Kim, and Cheol Hong Kim. Early miss prediction based periodic cache bypassing for high performance GPUs. *Microprocessors and Microsystems*, 55:44–54, November 2017.

[12] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. Locality-Driven Dynamic GPU Cache Bypassing. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 67–77, Newport Beach California USA, June 2015. ACM.

[13] Amin Jadidi. Selective Caching: Avoiding Performance Valleys in Massively Parallel Architectures. 2020.

[14] Yunho Oh, Gunjae Koo, Murali Annavaram, and Won Woo Ro. Linebacker: Preserving victim cache lines in idle register files of GPUs. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 183–196, Phoenix Arizona, June 2019. ACM.

[15] Ang Li, Gert-Jan Van Den Braak, Akash Kumar, and Henk Corporaal. Adaptive and transparent cache bypassing for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Austin Texas, November 2015. ACM.

[16] Juan Fang, Zelin Wei, and Huijing Yang. Locality-Based Cache Management and Warp Scheduling for Reducing Cache Contention in GPU. *Micromachines*, 12(10):1262, October 2021.

[17] Yun Liang, Xiaolong Xie, Yu Wang, Guangyu Sun, and Tao Wang. Optimizing Cache Bypassing and Warp Scheduling for GPUs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(8):1560–1573, August 2018.

[18] Gunjae Koo, Yunho Oh, Won Woo Ro, and Murali Annavaram. Access Pattern-Aware Cache Management for Improving Data Utilization in GPU. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 307–319, Toronto ON Canada, June 2017. ACM.

[19] Shin-Ying Lee and Carole-Jean Wu. Ctrl-C: Instruction-Aware Control Loop Based Adaptive Cache Bypassing for GPUs. *th International Conference on Computer Design*, 2016.

[20] Xiaolong Xie, Yun Liang, Yu Wang, Guangyu Sun, and Tao Wang. Coordinated static and dynamic cache bypassing for GPUs. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 76–88, Burlingame, CA, USA, February 2015. IEEE.

[21] Yijie Huangfu and Wei Zhang. Hardware-Based and Hybrid L1 Data Cache Bypassing to Improve GPU Performance. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 972–976, New York, NY, August 2015. IEEE.

[22] Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. An efficient compiler framework for cache bypassing on GPUs. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 516–523, San Jose, CA, November 2013. IEEE.

[23] AMD RDNA Whitepaper. https://www.amd.com/system/files/documents/rdna-whitepaper.pdf.

[24] NVIDIA Ampere GPU Architecture Tuning Guide. https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html.

[25] Syntax of AMDGPU Instruction Modifiers. https://llvm.org/docs/AMDGPUModifierSyntax.html.

[26] John A. Stratton, Christopher I. Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. 2012.

[27] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[28] NVIDIA CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit.

[29] NVIDIA PTX ISA 8.3. https://docs.nvidia.com/cuda/parallel-thread-execution//#cache-operators.