Track 3 Architectures and Accelerators EURO-PAR 2025

中山大學
SUN YAT-SEN UNIVERSITY

国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

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

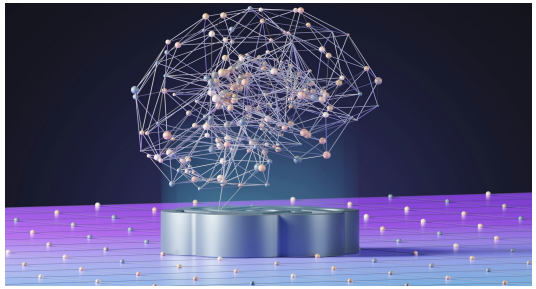
Mengyue Xi, Jingyi He, Xianwei Zhang

Email: ximy@mail2.sysu.edu.cn

Time: 16:00 - 16:20, August 28th, 2025

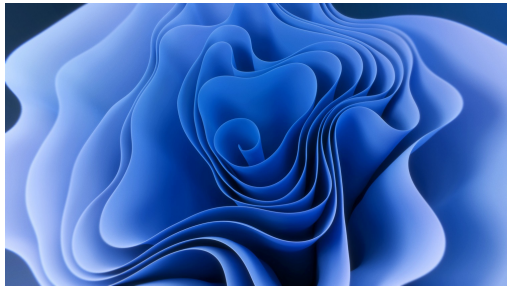
GPU Parallelism & Kernel Concurrency

- GPUs: Massively parallel architecture (SIMT)
  - Thousands of cores executing the same instruction on different data
  - How programs Run: Applications launch parallel workloads as ***kernels***
  - Ideal for AI, HPC and Graphics



AI training & inference

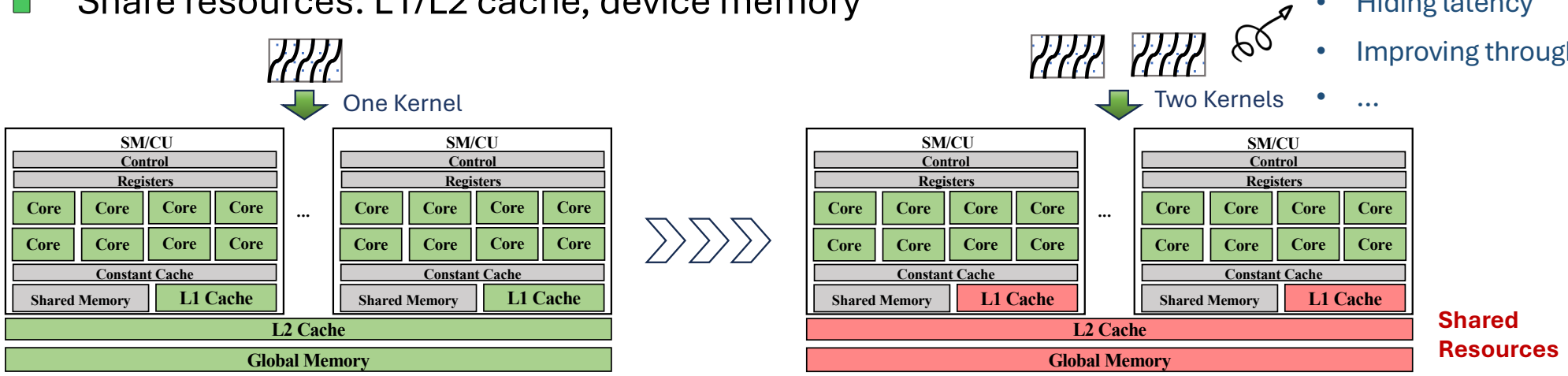
Supercomputing for Science



Real-time photorealistic rendering

GPU Parallelism & Kernel Concurrency

- GPUs: Massively parallel architecture (SIMT)
  - Thousands of cores executing the same instruction on different data
  - How programs Run: Applications launch parallel workloads as ***kernels***
- **Kernel Concurrency**: Execute multiple kernels *simultaneously*
  - Different kernels occupy the same or separate SMs
  - Share resources: L1/L2 cache, device memory



Maximize hardware utilization by:

- Balancing resource demands
- Hiding latency
- Improving throughput
- …

Concurrency Performance Barrier: Cache Contention

- Concurrent kernels compete for shared cache resources
  - Different memory access patterns collide
  - Critical data gets evicted prematurely (cache thrashing)

Concurrency Performance Barrier: Cache Contention

- Concurrent kernels compete for shared cache resources
  - Different memory access patterns collide
  - Critical data gets evicted prematurely (cache thrashing)
- ***Example***: *convolution (conv)* – single global load with regular access patterns

Independently running:

- Cache bypassing is ineffective for *conv* due to no cache contention

Cache Effects	Performance Improvement (%)
L2 Bypassing	-0.45%
L0/1 Bypassing	-1.05%

Concurrently running with other kernels:

- Cache bypassing → varying performance effects

Cache Effects	DCT_1	DCT_2	LBM	HybridSort_1	HybridSort_2	SPMV	Particlefilter
L2 Bypassing	2.35%	2.42%	5.98%	2.67%	0.26%	-0.02%	-0.01%
L0/1 Bypassing	-0.07%	-0.01%	1.83%	7.96%	0.09%	0.07%	-0.02%

Concurrency Performance Barrier: Cache Contention

Two observations:

1. Different cache contention patterns exist across kernel pairs *(reflecting varying inter-kernel interactions)*
2. Cache bypassing can *mitigate* inter-kernel cache contention

Cache Effects	Performance Improvement (%)
L2 Bypassing	-0.45%
L0/1 Bypassing	-1.05%

No cache contention

Minor L2 contention

L2 cache contention

L0/1 cache contention

Cache Effects	DCT_1	DCT_2	LBM	HybridSort_1	HybridSort_2	SPMV	Particlefilter
L2 Bypassing	2.35%	2.42%	5.98%	2.67%	0.26%	-0.02%	-0.01%
L0/1 Bypassing	-0.07%	-0.01%	1.83%	7.96%	0.09%	0.07%	-0.02%

Cache Management for Kernel Concurrency

Two observations:

1. Different cache contention patterns exist across kernel pairs *(reflecting varying inter-kernel interactions)*
2. Cache bypassing can *mitigate* inter-kernel cache contention

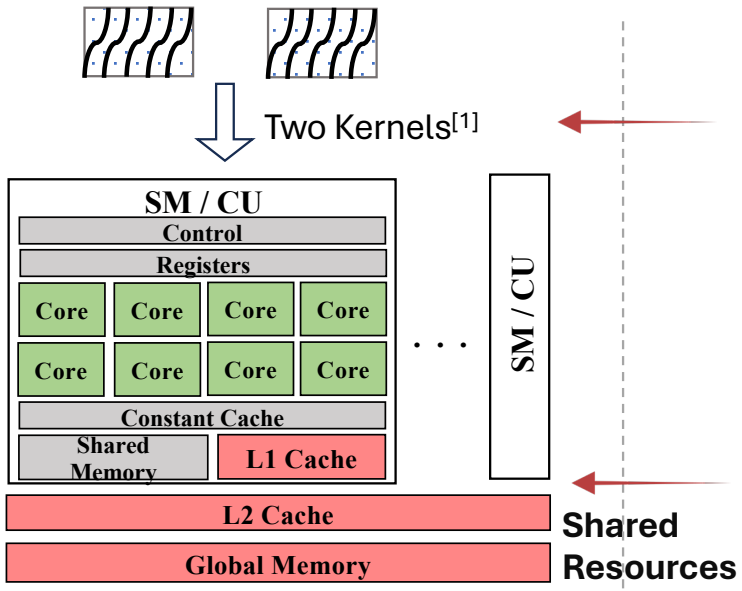
Key techniques:

Selecting kernel pair with minimal cache contention for concurrent execution

Appropriate cache bypassing strategies during kernel execution

Cache Management to Enhance Kernel Concurrency

Cache Management for Kernel Concurrency



Key techniques:

Selecting kernel pair with minimal cache contention for concurrent execution

Appropriate cache bypassing strategies during kernel execution

Kernel Concurrency + Cache Management → **Optimized Performance**

Maximize hardware utilization

Minimizes cache contention

[1] We mainly focus on the two kernels, but the approach can be easily extended to more. For details, see Section 4.5 in the paper.

Cache Management for Kernel Concurrency

Cache Pattern Analysis is the backbone

Choose the Suitable Method

How to Select?

Which to Set?

Key techniques:

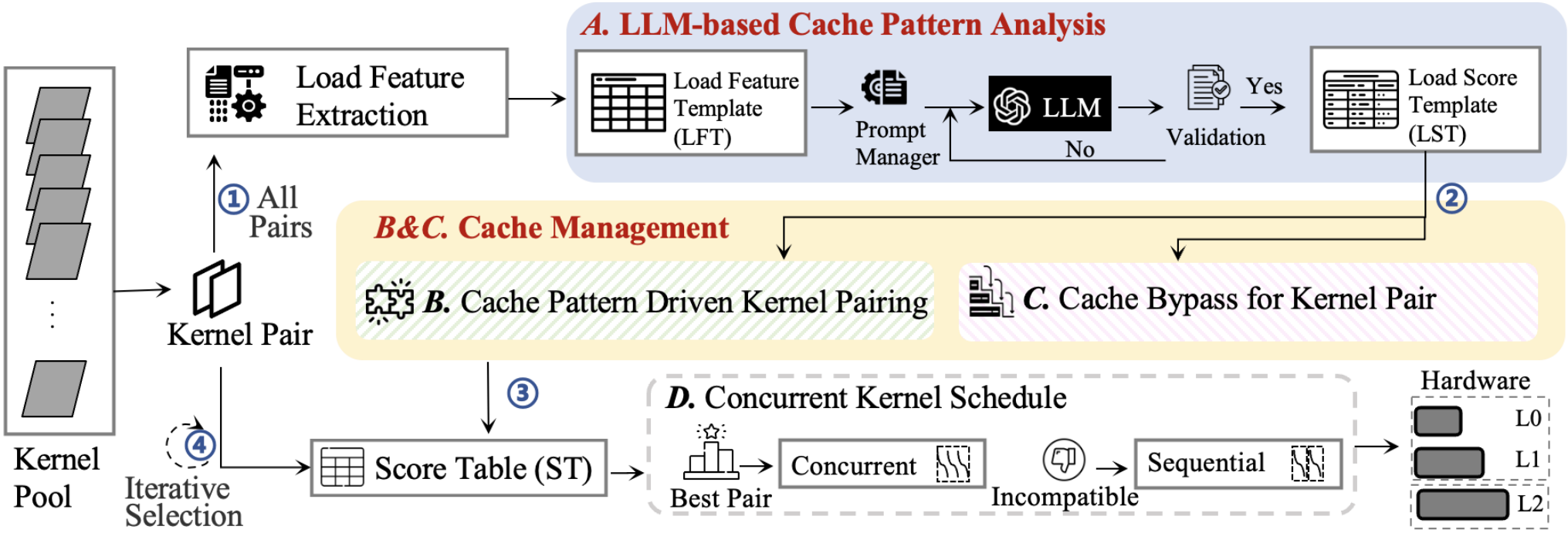
Selecting kernel pair with minimal cache contention for concurrent execution

Appropriate cache bypassing strategies during kernel execution

Method	Use Case	Setup Effort	Cost	Latency	Interpretability
Analytical Model	Static, rule-based patterns	High (expert tuning)	Medium (expert time)	Low	High
ML Training	Complex, evolving patterns	Medium (data collection + training)	High (training/resources)	Medium	Medium
LLM Query	Flexible, Natural language queries	Low (API integration)	Medium (API costs)	Medium	Low

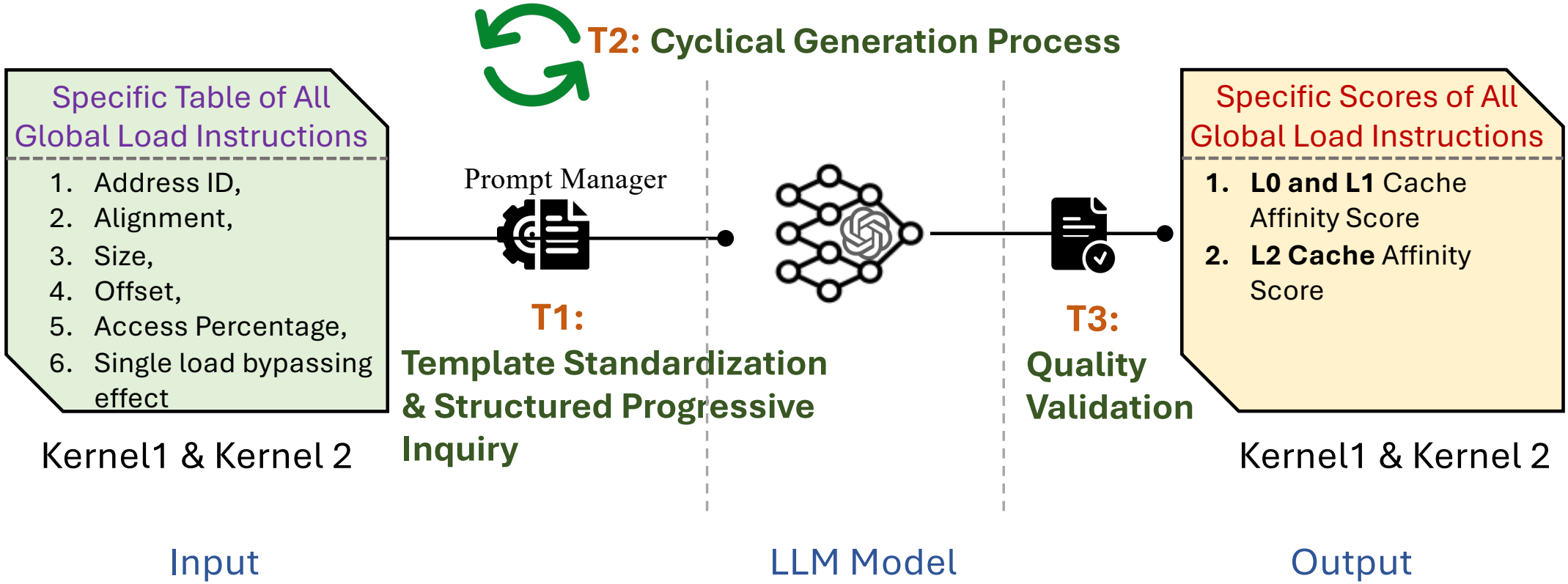
CacheC: Overall Workflow

Kernel Pairs Selection & Cache-bypass Strategies



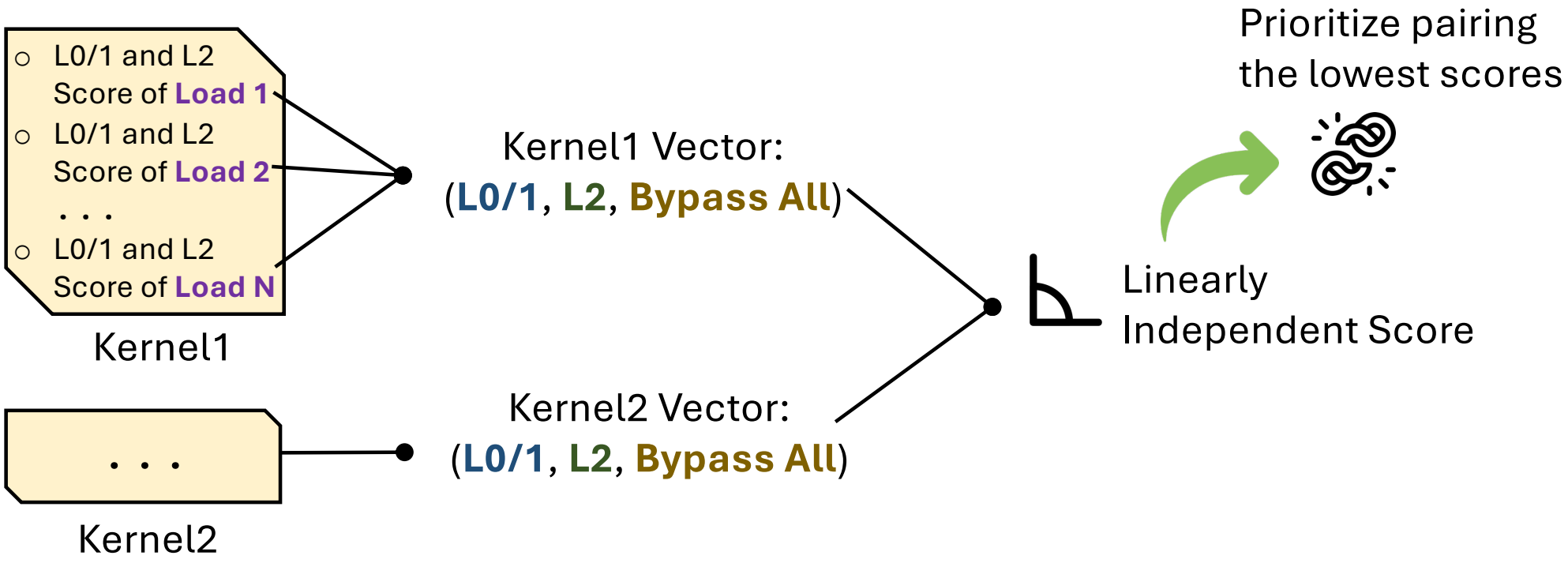
CacheC: LLM-based Cache Pattern Analysis

- **Three techniques to ensure validation output: T1 & T2 & T3**



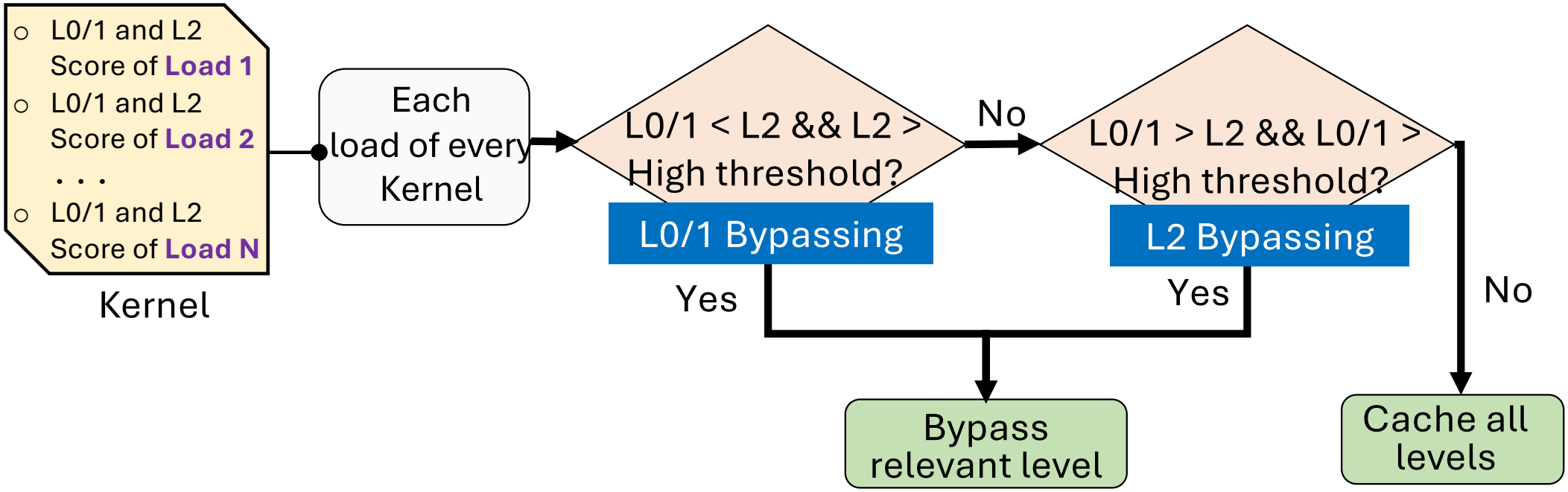
CacheC: Cache Pattern Driven Kernel Pairing

- **Principle**: The *more orthogonal* the two kernels are, the *less cache contention* they have



CacheC: Cache Bypass for Kernel Pair

- Bypass L0/1 when affinity for L2 is sufficient, and bypass L2 when affinity for L1 is sufficient



Experiments: Setup

- Platform: AMD Radeon RX 6900 XT, ROCm 5.5.0, LLVM 14.0.0
- Workloads: 14 real-world kernels (*abbr.*)

GEL	REL	REV	SW1	DT1	HS1	SPV
PAT	CON	LBM	SW1	DT2	HS2	MAX

- Schemes:

Kernel execution schems	Desciption	Cache management schemes	Desciption
Baseline	Sequential execution	**Mpache**	*ASPDAC 2025*
Rpair	Randomly paired for concurrent execution	**Cbypass**	Cache Bypass for Kernel Pair
Cpair	Cache Pattern Driven Kernel Pairing	**Exhaustive**	The best strategy after exhaustive exploration

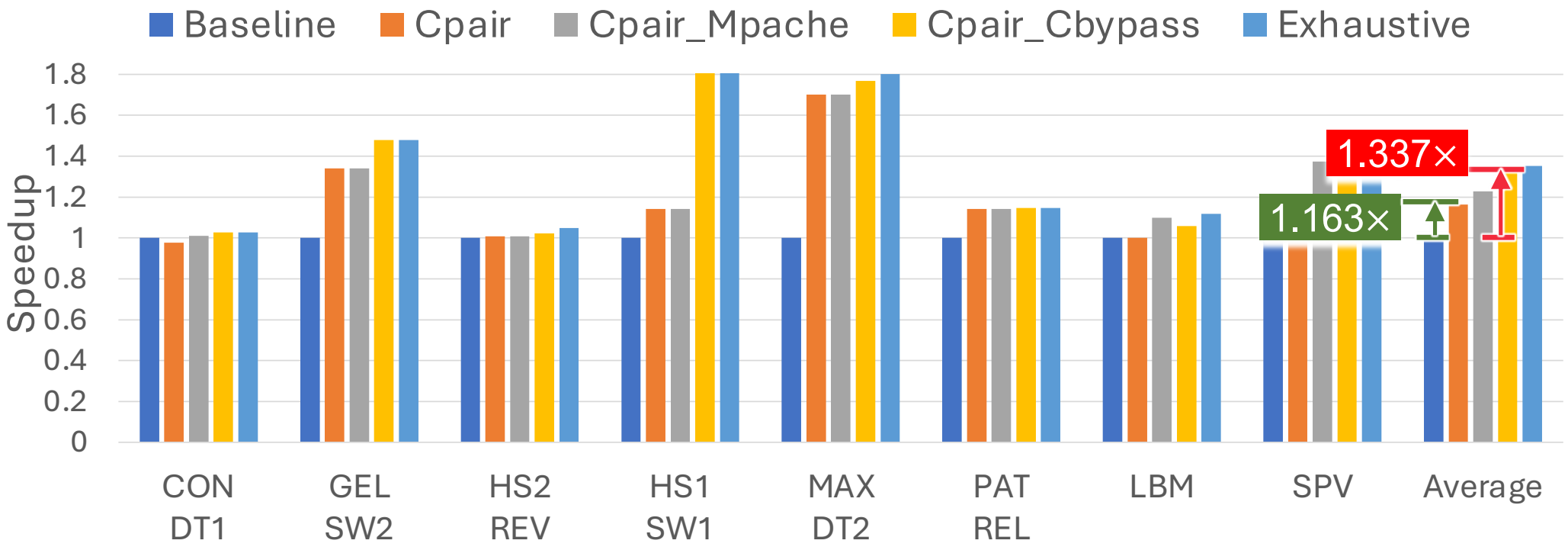
Experiments: Turnaround and Throughput

- When normalized to the baseline, *Cpair* outperforms *Rpair*, *Cbypass* surpasses *Mpache*, and *CacheC* achieves the best performance.

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

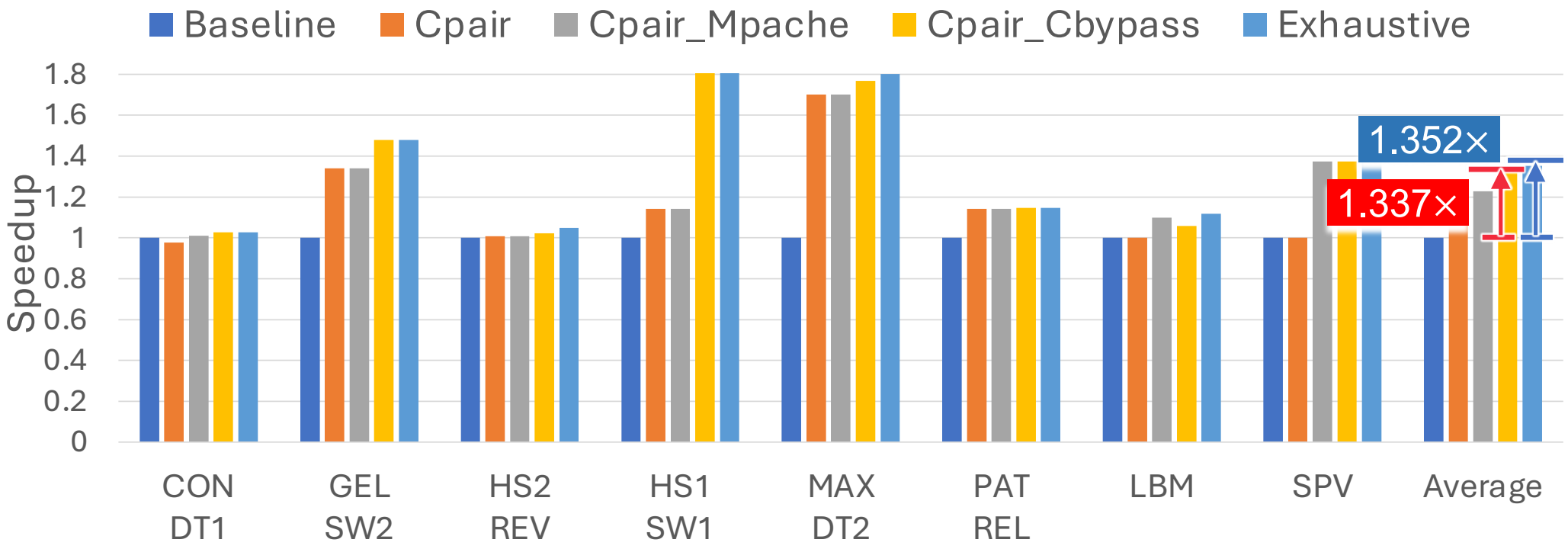
Experiments: Speedup Breakdown

- For the best pairs, ***Cpair*** achieves an average speedup of **1.163x**, while ***Cbypass*** further improves it to **1.337x** on average.



Experiments: LLM Validation

- ***Exhaustive*** achieves a ***1.352x*** speedup with a 1.011x discrepancy compared to ***CacheC***, and the bypass strategy is nearly the same.



More in the Paper

- GPU Bypassing Technical Details
- Detailed process of Cache Pattern Analysis
- Concurrent Kernel Scheduling Mechanism
- Experiments
  - Cache Hit Rate Analysis
  - Multiple Kernels Extension
  - Stability Evaluation
  - Platform and Overhead Discussion

• • •

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].

Conclusion

- Cache contention emerges as a limiting factor that impedes further optimization of kernel concurrency performance
- *CacheC*: Cache Management for GPU Kernel Concurrency
  - LLM-Based Cache Pattern Analysis
  - Cache-Pattern-Driven Kernel Pairing (*CPair*)
  - Cache Bypass for Kernel Pairs (*Cbypass*)
  - Concurrent Kernel Scheduling
- *CacheC* outperforms traditional cache management and random kernel pairing

中山大學
SUN YAT-SEN UNIVERSITY

Thank You!

Email: ximy@mail2.sysu.edu.cn

Mengyue Xi, Jingyi He, Xianwei Zhang