Objective

# Improving Intra-Warp Cache Utilization on GPUs through Compiler-Guided Exploitation of Texture Hardware

Method

Mechanism

*Note: We assume NVIDIA GPUs.*


Yi-Ting Lai

Advisor: Yi-Ping You

國立陽明交通大學
NATIONAL YANG MING CHIAO TUNG UNIVERSITY

Institute of Computer Science and Engineering

National Yang Ming Chiao Tung University

sslab 系統軟體實驗室

# OUTLINE

- What is Intra-Warp Cache Utilization?
- Why Is It Important?
- How Can Texture Hardware Help?
- How Can a Compiler Exploit It?
- How Effective Is the Exploitation?
- Limitations of the Mechanism
- Conclusion

# What is Intra-Warp Cache Utilization?

# GPU Execution Hardware: SIMT-Core Array

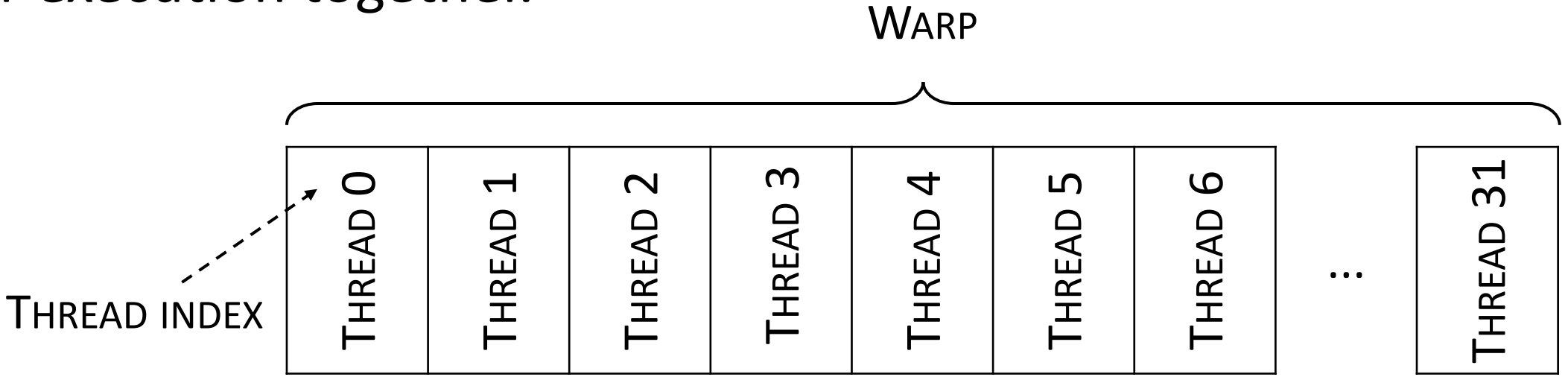
> Analogous to SIMD lanes,
> executing the same instruction in parallel.

SIMT core × 32

SIMD: single instruction, multiple data
SIMT: single instruction, multiple threads

# GPU Execution Model: Thread and Warp

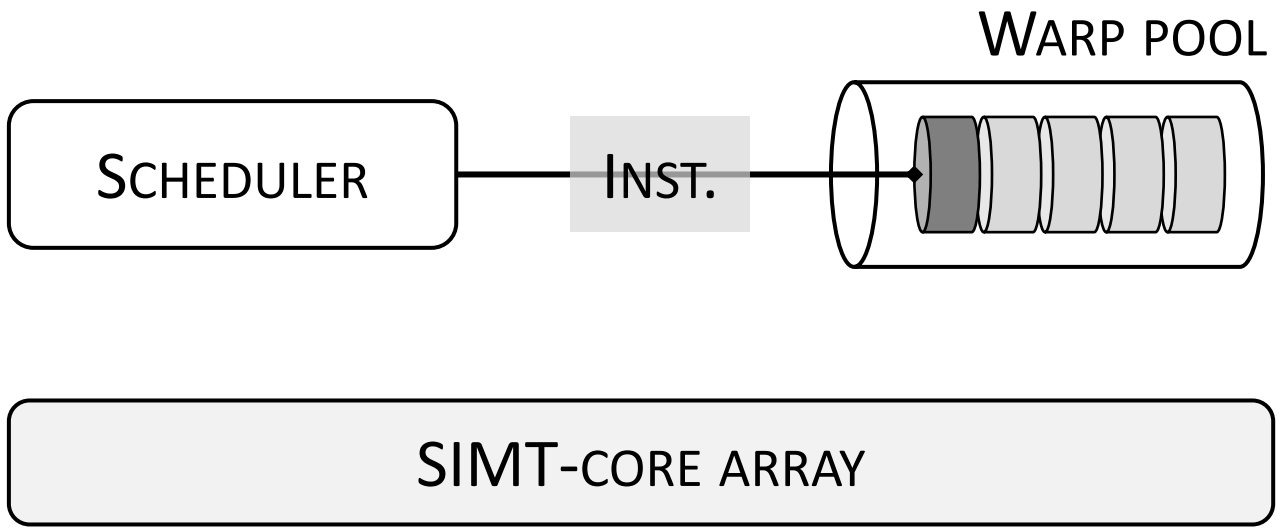
The SPMD programming model decomposes a workload into many *threads*. Every 32 threads form a *warp* and are scheduled for execution together.




SPMD: single program, multiple data

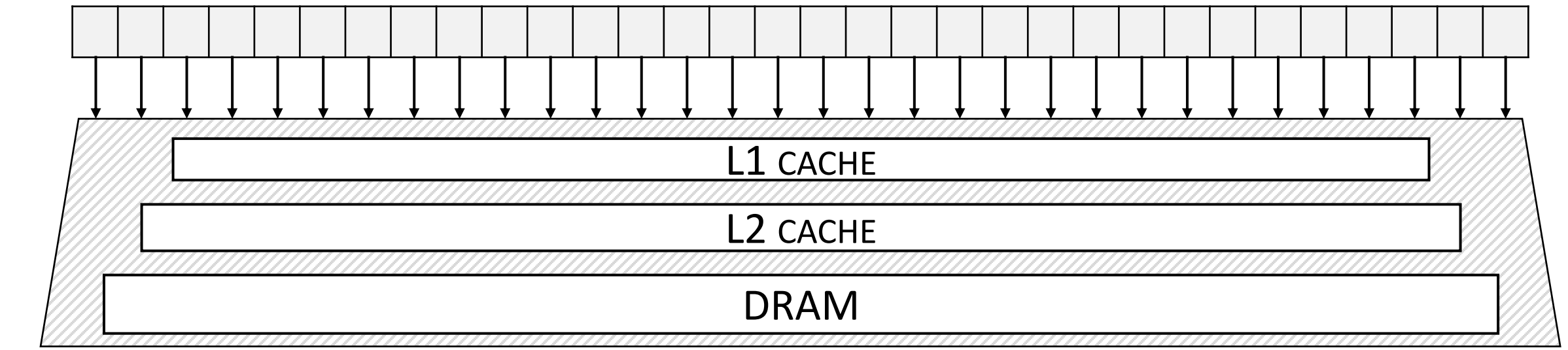
# SCHEDULING UNIT: WARP

Each cycle, the scheduler selects a warp and issues its next instruction; the selected warp may differ from the previous cycle.



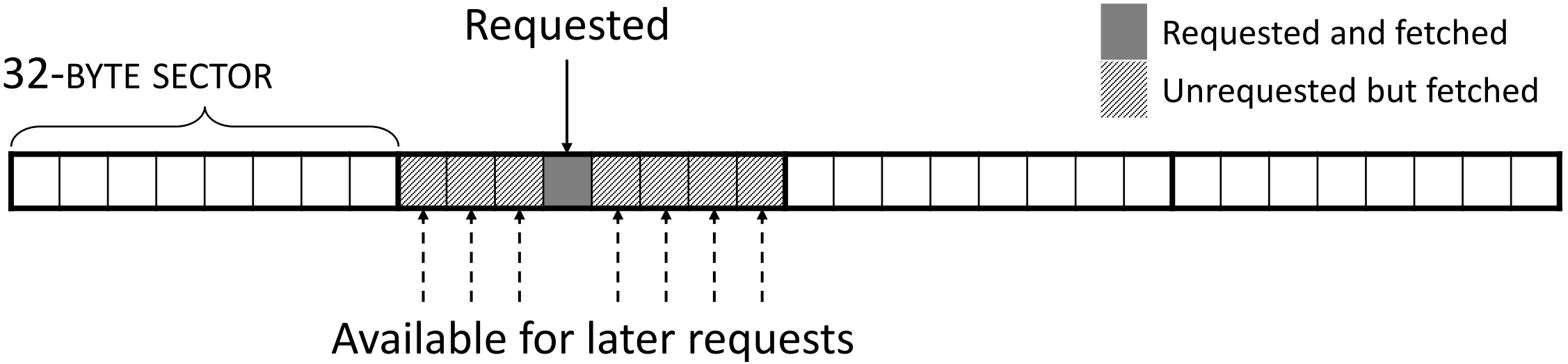
# Multi-Address Memory Instruction

> A memory instruction requests up to 32 addresses—one from each thread—which are sent to the memory hierarchy.



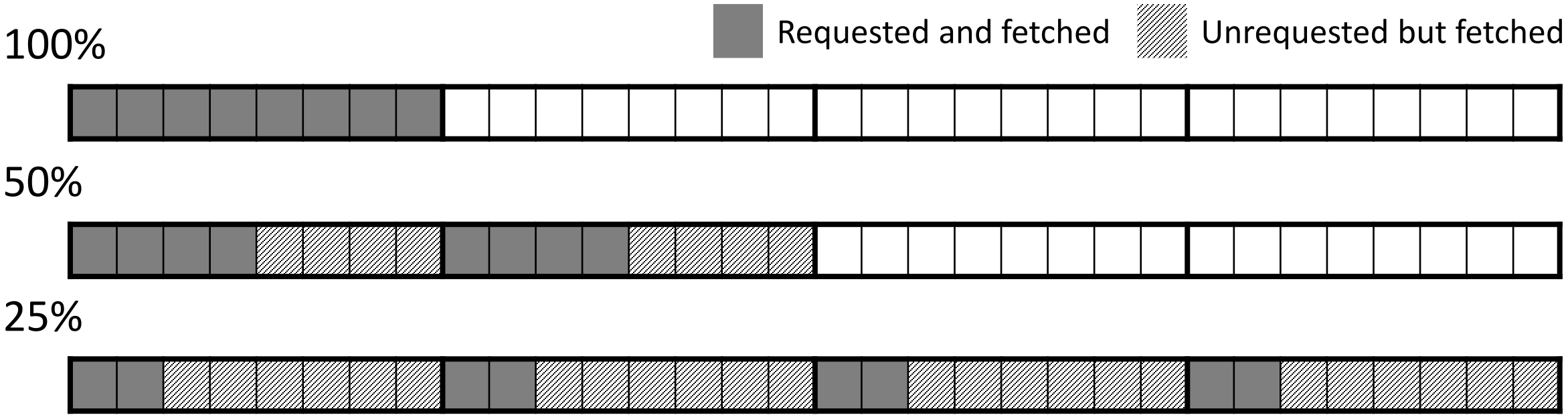
# CACHE FETCH GRANULARITY

On a cache miss, data is fetched in 32-byte cache sectors, not necessarily limited to the requested data.
Unrequested-but-fetched data may later be requested by subsequent memory instructions as cache hits.



# Definition: Intra-Warp Cache Utilization

> The percentage of fetched cache-sector data that is actually requested by a memory instruction.



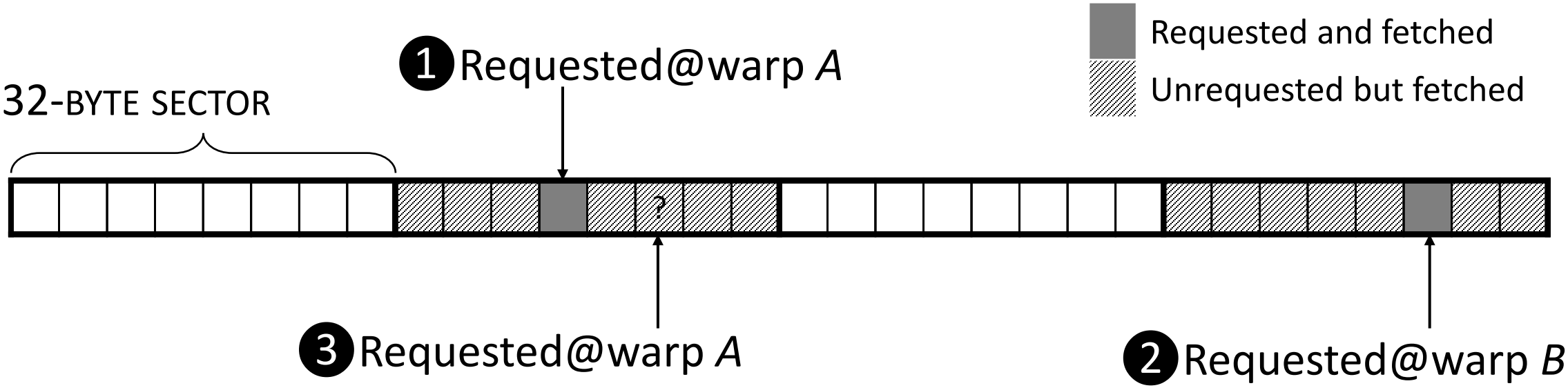

*Note: This is actually cache-sector utilization; for the cache-line counterpart, see Chapter 3.*

# Why is Intra-Warp Cache Utilization Important?

# INTER-WARP CACHE CONTENTION

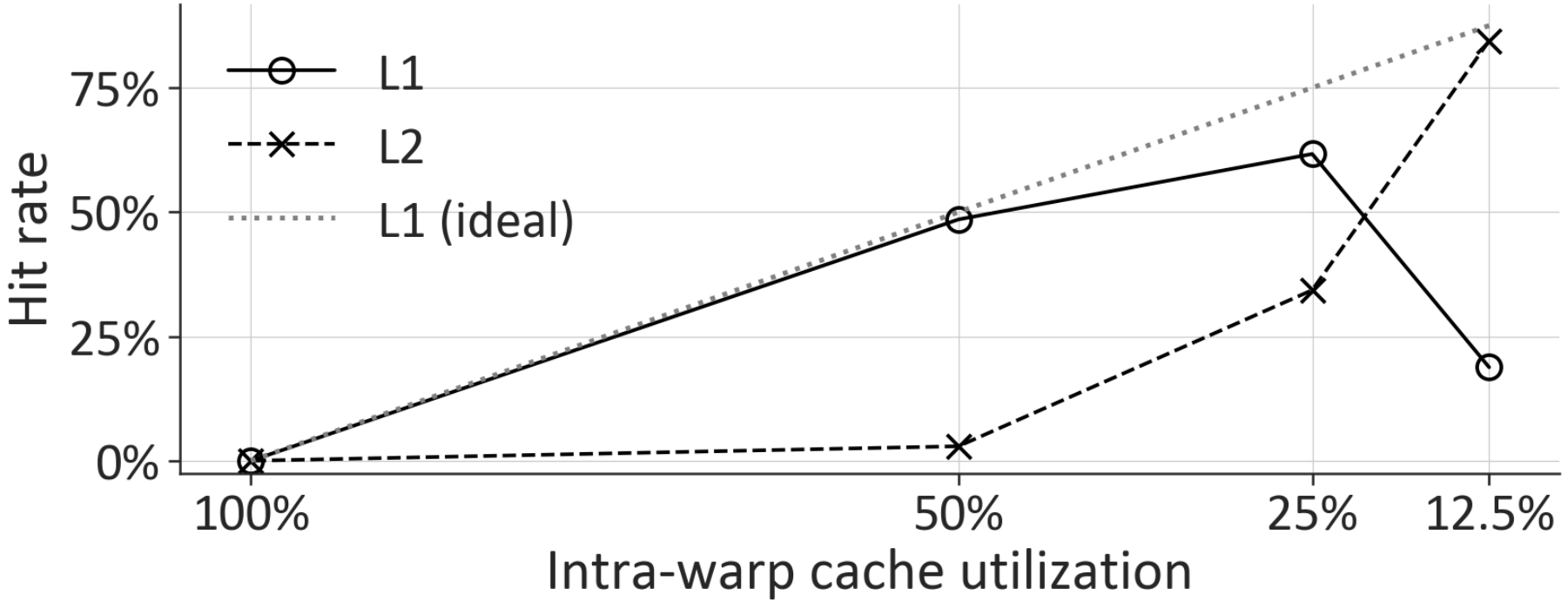
Cache is shared across all warps in the pool.
Other warps can evict data before it is requested later.

*Note: Eviction granularity is a cache line.*



# KEY OBSERVATION I

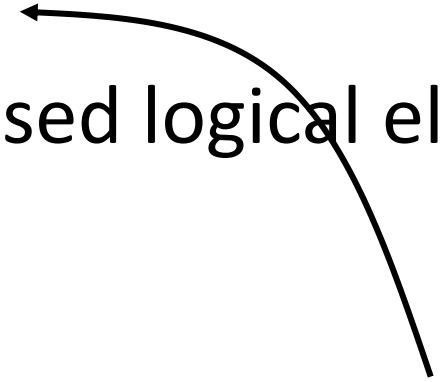
> Memory instructions should maximize utilization of fetched cache data rather than leaving it to subsequent memory instructions.




*Scenario: streaming access; each element is accessed exactly once.*

# Two Ways to Improve Intra-Warp Cache Utilization

**Access pattern:**
make threads access nearby logical elements together.
**Memory layout:**
place jointly accessed logical elements at nearby memory addresses.



Our proposed direction

# What is Memory Layout?

# Memory Layout: $(x, y) \rightarrow i$

## Logical

| $y$ \ $x$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | (0, 0) | (1, 0) | (2, 0) | (3, 0) |
| 1 | (0, 1) | (1, 1) | (2, 1) | (3, 1) |
| 2 | (0, 2) | (1, 2) | (2, 2) | (3, 2) |
| 3 | (0, 3) | (1, 3) | (2, 3) | (3, 3) |

## Physical

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| | ???? | ???? | ???? | ???? | ???? | |

Determines the mapping between logical data and its physical placement in memory (in DRAM and within cache lines).

# Default Memory Layout: Linear

## Logical

| $y$ \ $x$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | (0, 0) | (1, 0) | (2, 0) | (3, 0) |
| 1 | (0, 1) | (1, 1) | (2, 1) | (3, 1) |
| 2 | (0, 2) | (1, 2) | (2, 2) | (3, 2) |
| 3 | (0, 3) | (1, 3) | (2, 3) | (3, 3) |

## Physical

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| | (0, 0) | (1, 0) | (2, 0) | (3, 0) | (0, 1) | |

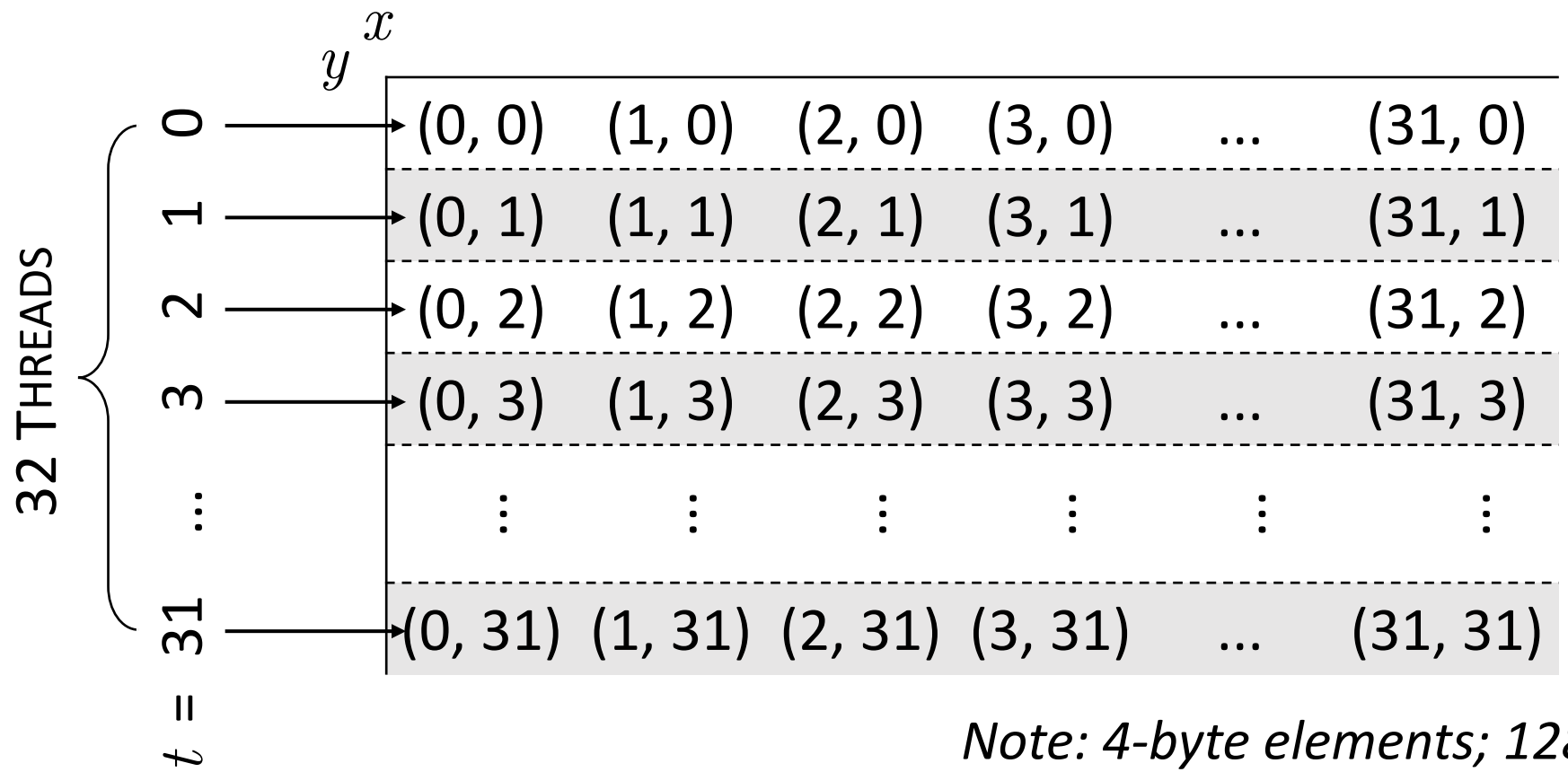
---------- Row boundary

By default, data copied to GPU is stored in a linear layout, with consecutive data of each row placed consecutively in memory.

# Why Can Linear Layout Lead to Low Intra-Warp Cache Utilization?

# WORST-CASE SCENARIO: CROSS-ROW ACCESS

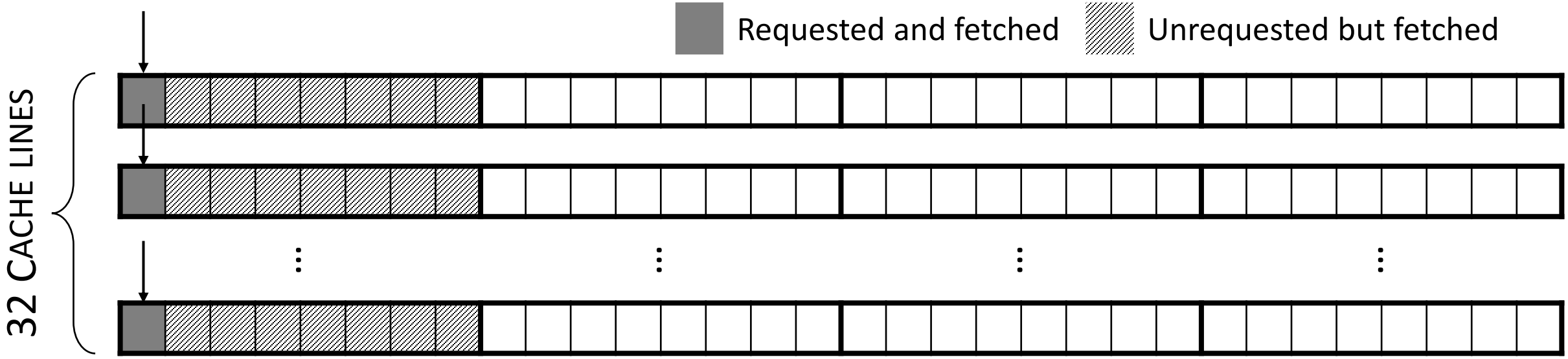
Each thread$_t$ in a warp accesses $(x,\ y + t)$.




*Note: 4-byte elements; 128-bytes logical row.*

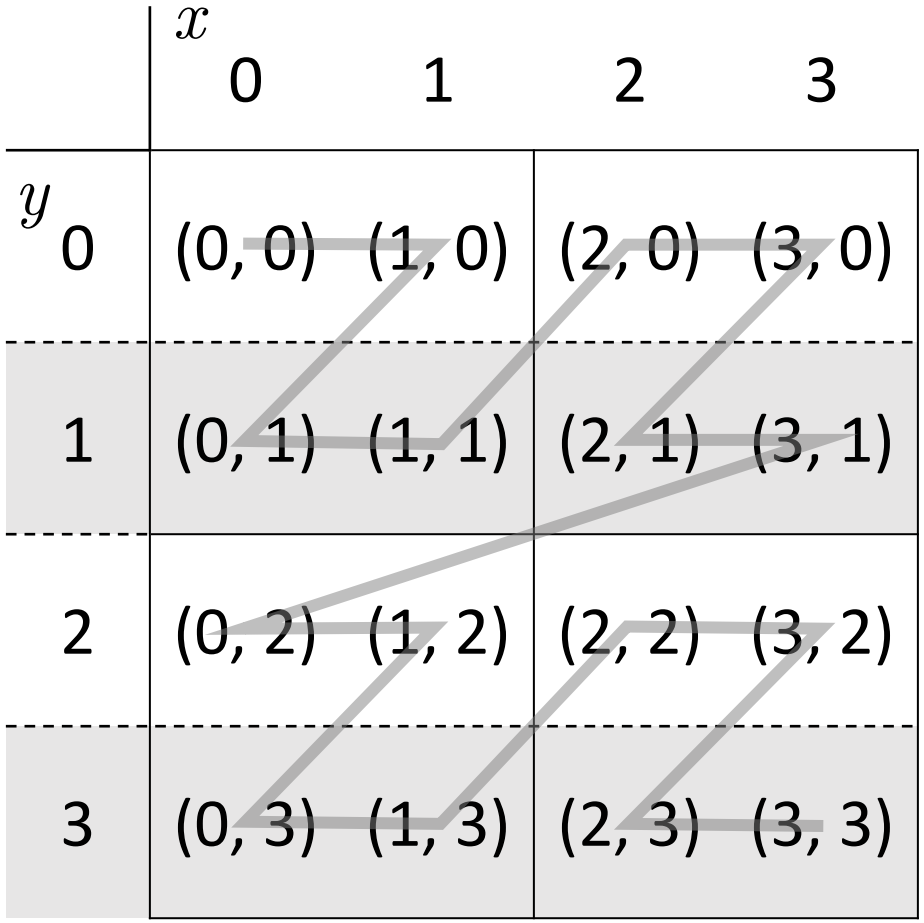
# Worst-Case Scenario: Cross-Row Access

Each thread$_t$ in a warp accesses $(x,\ y+\ t)$.
Because consecutive rows are separated by one cache line under the linear layout, the accesses map to 32 distinct cache sectors and lines, achieving 12.5% intra-warp cache utilization.

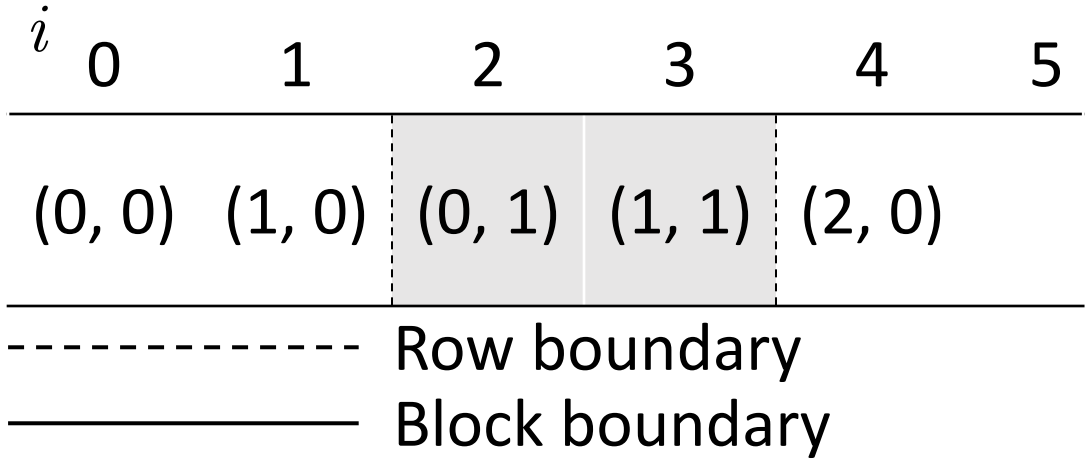


# Blocked Layout Improves Worst-Case Utilization

## Logical



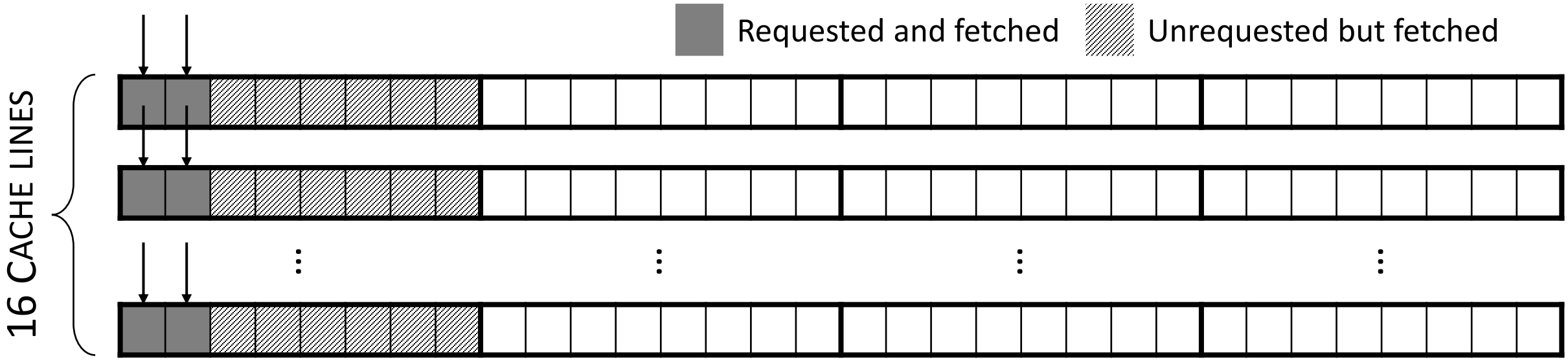

## Physical




When accesses may follow either logical dimension, a blocked layout can provide higher worst-case intra-warp cache utilization.

# Blocked Layout Improves Worst-Case Utilization

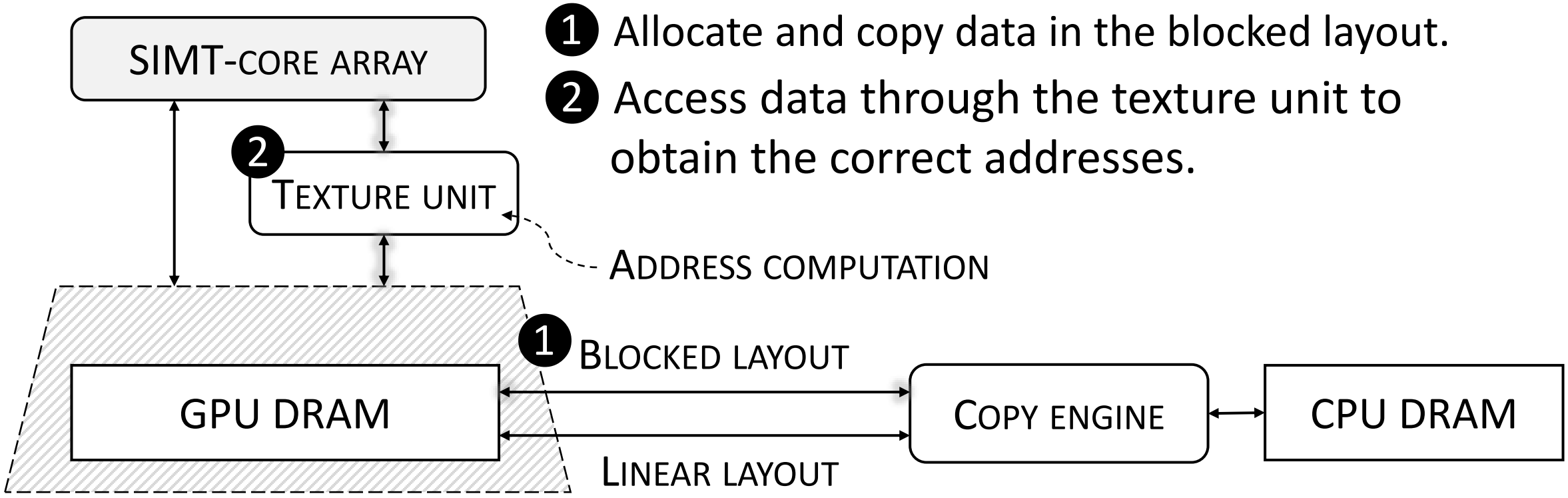
For the same cross-row access, a 2 × 2 blocked layout pairs data from adjacent rows within the same cache sector, doubling intra-warp cache utilization from 12.5% to 25%.



# How Can Texture Hardware Help?

# KEY OBSERVATION II

Existing texture hardware already employs a blocked layout.




1. Allocate and copy data in the blocked layout.
2. Access data through the texture unit to obtain the correct addresses.

# Proposed Approach

- Key observation I:
  Memory instructions should maximize utilization of fetched cache data rather than leaving it to subsequent memory instructions.
- Key observation II:
  Existing texture hardware already employs a blocked layout.

> Use compiler analysis to identify data with low intra-warp cache utilization, then transparently allocate and copy it in the blocked layout and access it through the texture unit.

# How Is the Approach Realized?

# GPU Submission Workflow

*Note: This is CUDA pseudocode.*

```
/* execute on GPU */
@gpu_kernel
def kernel(gpu_data):
      v = access(gpu_data, x, y + thread_idx);

/* execute on CPU */
gpu_data = allocate_on_gpu(data_shape);
copy_to_gpu(cpu_data, gpu_data, data_shape);
launch_kernel(kernel, /* kernel argument */ gpu_data);
copy_from_gpu(gpu_data, cpu_data, data_shape);
```

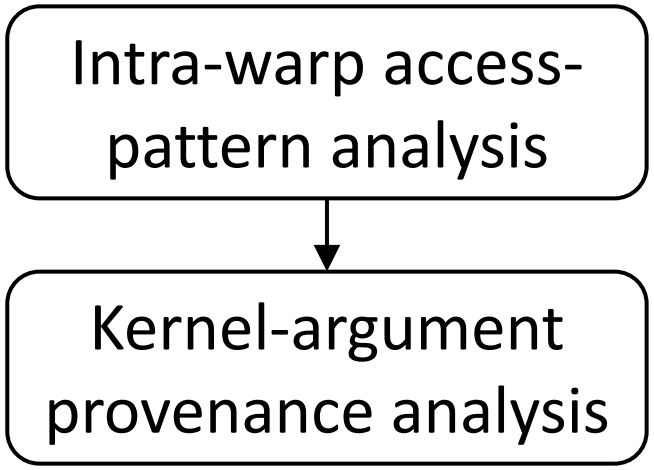
# *AutoTex*: the Compiler Pipeline

Intra-warp access-pattern analysis

Identify data with low intra-warp cache utilization from thread-index ($t$) use in its $(x,\ y)$ access pattern.

```
@gpu_kernel
def kernel(gpu_data):
      v = access(gpu_data, x, y + thread_idx);
```

# *AutoTex*: the Compiler Pipeline

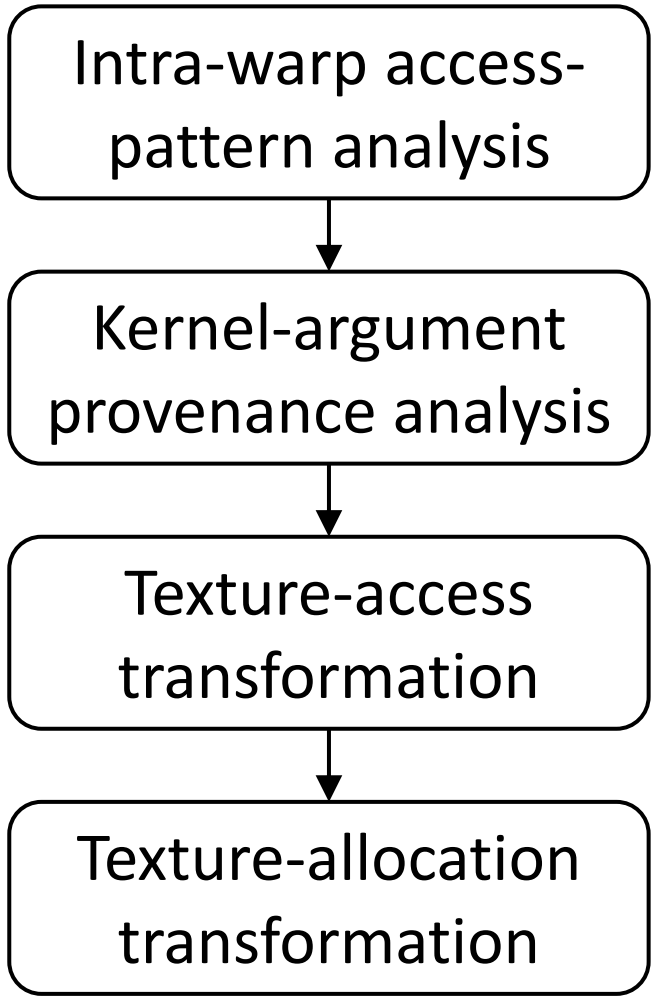



Identify data with low intra-warp cache utilization from thread-index ($t$) use in its $(x,\ y)$ access pattern.

Identify sharing kernels that must use the texture unit if their shared data is allocated in the blocked layout.

```
gpu_data = allocate_on_gpu(data_shape);
...
launch_kernel(kernel_1, gpu_data);
launch_kernel(kernel_2, gpu_data);
...
```

# *AUTOTEX*: THE COMPILER PIPELINE




Identify data with low intra-warp cache utilization from thread-index ($t$) use in its $(x,\ y)$ access pattern.

Identify sharing kernels that must use the texture unit if their shared data is allocated in the blocked layout.

Rewrite accesses to such data through the texture unit. (❷)

Allocate and copy such data using the blocked layout. (❶)

*Note: As an implementation detail,* ❷ *is rewritten before* ❶*.*

# TRANSFORMED GPU SUBMISSION

*Note: This is CUDA pseudocode.*

```
/* execute on GPU */
@gpu_kernel
def kernel(gpu_data):
      v = tex_access(gpu_data, x, y + thread_idx);

/* execute on CPU */
gpu_data = allocate_on_gpu(data_shape, BLOCKED_LAYOUT);
copy_to_gpu(cpu_data, gpu_data, data_shape, BLOCKED_LAYOUT);
launch_kernel(kernel, /* kernel argument */ gpu_data);
copy_from_gpu(gpu_data, cpu_data, data_shape, BLOCKED_LAYOUT);
```

# How Effective is the Approach?

# Experimental Environment

*Note: Also evaluated on an RTX 3050 Laptop GPU; See Chapter 7.*

**Hardware specification**

| | | |
|---|---|---|
| GPU | Model | NVIDIA GeForce RTX 4080 |
| | Arch. | Ada Lovelace |
| | # SMs | 76 |
| | L1 cache | 128 KiB |
| | L2 cache | 64 MiB |
| | DRAM | 15.6 GiB |
| CPU | Model | Intel® Core™ i7-12700KF |
| | DRAM | 31.2 GiB |

**Software version**

| | | |
|---|---|---|
| OS | Distribution | openSUSE Leap 16.0 |
| | Linux kernel | 6.12.0 |
| Compiler | LLVM | 19.1.7 |
| CUDA | Driver | 580.119.02 |
| | Toolkit | 12.9 |
| | Driver API | 13.0 |
| | Runtime API | 12.9 |

# BENCHMARK SUITES AND TRANSFORMED KERNELS

| | **NPB-GPU** | **PolyBench-ACC** | **Rodinia** | Total |
|---|---|---|---|---|
| # benchmarks | 8 | 21 | 26 | 55 |
| # transformed benchmarks | 3 | 9 | 4 | 16 |
| # kernels | 106 | 47 | 68 | 221 |
| # transformed kernels | 25 (24%) | 19 (40%) | 8 (12%) | 52 |
| # low-utilization kernels[a] | 16 | 10 | 6 | 32 |
| # sharing-only kernels[b] | 9 | 9 | 2 | 20 |

[a]Low-utilization kernels: transformed kernels containing low-utilization accesses.

[b]Sharing-only kernels: no low-utilization accesses; transformed only because they share data with another low-utilization kernel.

## Label numbers for transformed kernels

| | | | | | |
|---|---|---|---|---|---|
| 1 | IS/full_verify_gpu_kernel_1 | 19 | SP/compute_rhs_gpu_kernel_1 | 37 | gesummv/gesummv_kernel |
| 2 | IS/full_verify_gpu_kernel_2 | 20 | SP/compute_rhs_gpu_kernel_2 | 38 | gramschmidt/gramschmidt_kernel1 |
| 3 | LU/error_gpu_kernel | 21 | SP/rhs_norm_gpu_kernel_1 | 39 | gramschmidt/gramschmidt_kernel2 |
| 4 | LU/jacld_blts_gpu_kernel | 22 | SP/txinvr_gpu_kernel | 40 | gramschmidt/gramschmidt_kernel3 |
| 5 | LU/jacu_buts_gpu_kernel | 23 | SP/x_solve_gpu_kernel | 41 | mvt/mvt_kernel1 |
| 6 | LU/pintgr_gpu_kernel_1 | 24 | SP/y_solve_gpu_kernel | 42 | mvt/mvt_kernel2 |
| 7 | LU/pintgr_gpu_kernel_2 | 25 | SP/z_solve_gpu_kernel | 43 | syr2k/syr2k_kernel |
| 8 | LU/pintgr_gpu_kernel_3 | 26 | adi/adi_kernel1 | 44 | syrk/syrk_kernel |
| 9 | LU/rhs_gpu_kernel_1 | 27 | adi/adi_kernel3 | 45 | gaussian/Fan1 |
| 10 | LU/rhs_gpu_kernel_2 | 28 | adi/adi_kernel4 | 46 | gaussian/Fan2 |
| 11 | LU/rhs_gpu_kernel_3 | 29 | adi/adi_kernel6 | 47 | huffman/pack2 |
| 12 | LU/rhs_gpu_kernel_4 | 30 | atax/atax_kernel1 | 48 | huffman/vlc_encode_kernel_sm64huff |
| 13 | LU/setbv_gpu_kernel_1 | 31 | atax/atax_kernel2 | 49 | kmeans/invert_mapping |
| 14 | LU/setbv_gpu_kernel_2 | 32 | bicg/bicg_kernel1 | 50 | kmeans/kmeansPoint |
| 15 | LU/setbv_gpu_kernel_3 | 33 | bicg/bicg_kernel2 | 51 | leukocyte/GICOV_kernel |
| 16 | LU/setiv_gpu_kernel | 34 | gemver/gemver_kernel1 | 52 | leukocyte/dilate_kernel |
| 17 | LU/ssor_gpu_kernel_2 | 35 | gemver/gemver_kernel2 | | |
| 18 | SP/add_gpu_kernel | 36 | gemver/gemver_kernel3 | | |

# Evaluation Metrics

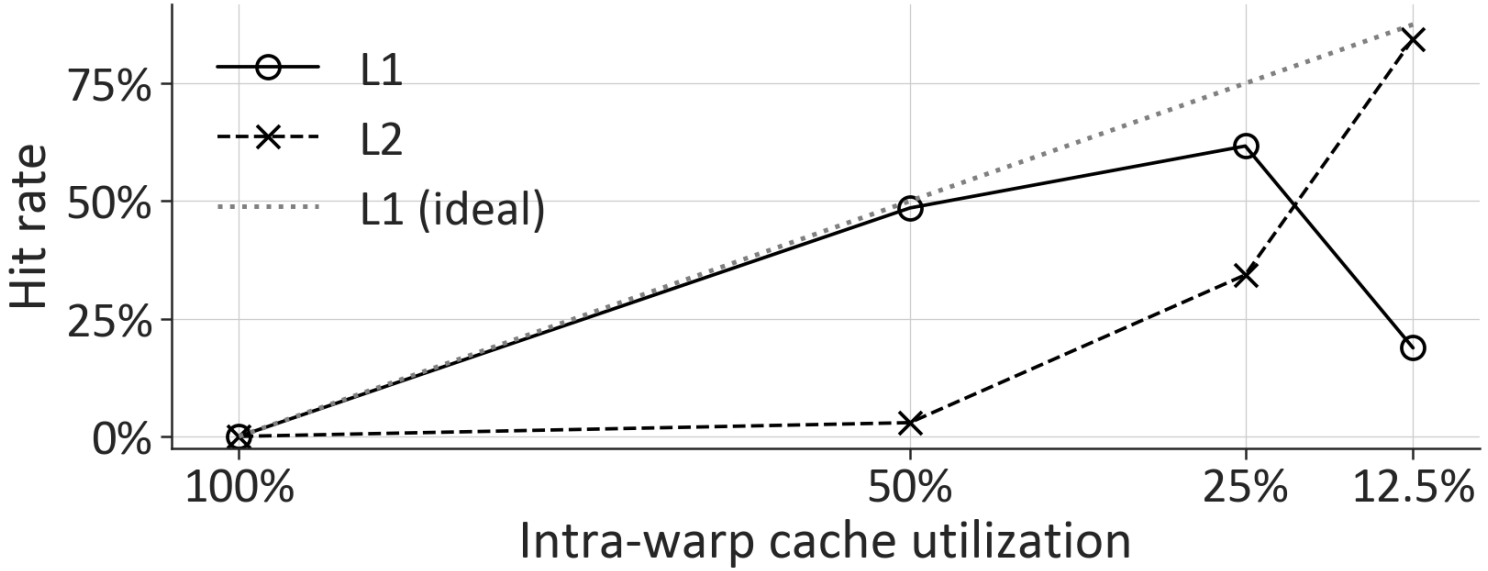
> Hit rate alone is ambiguous: higher utilization can lower it by leaving less fetched data for later hits, while effective caching raises it toward the ideal. We therefore evaluate effectiveness directly through (1) fetched-sector reduction and (2) speedup.

*Note:*

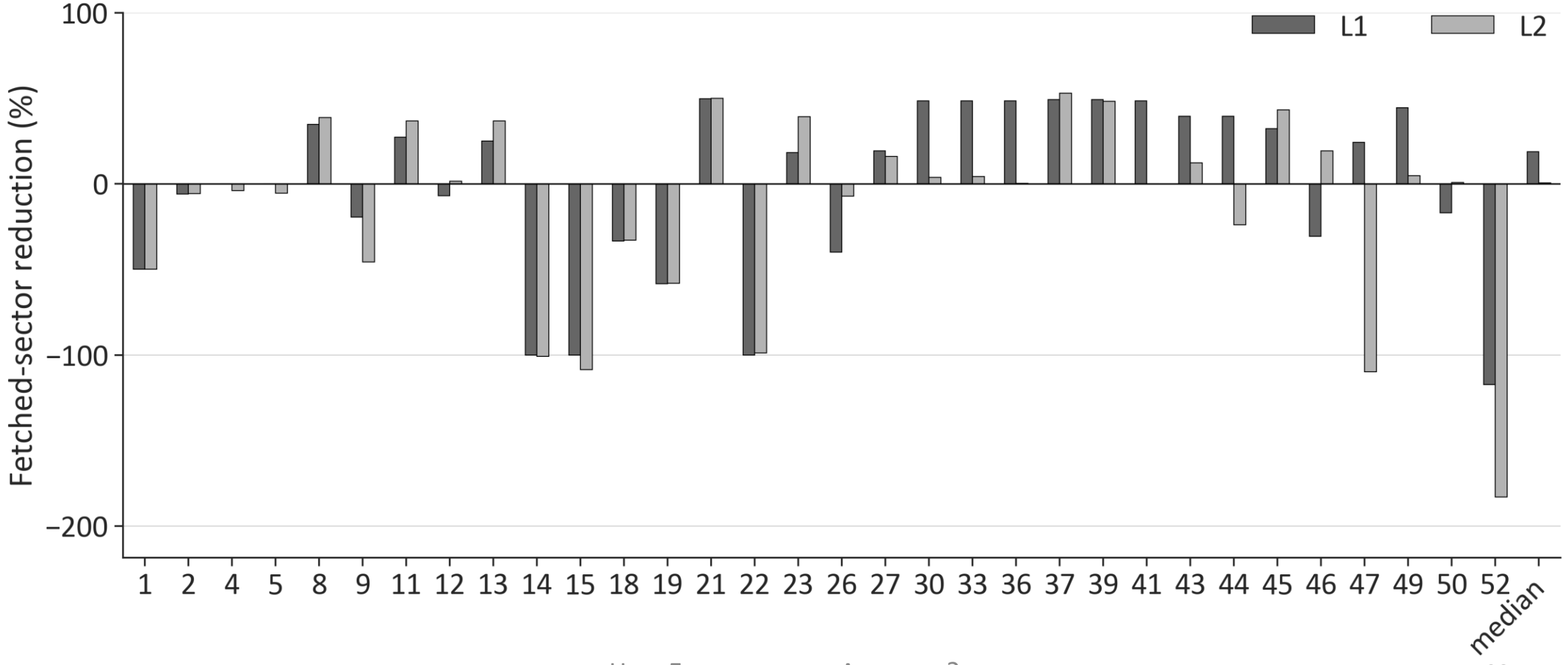
$$Reduction = \left(1 - \frac{AutoTex}{Baseline}\right) \times 100\%,$$

$$Speedup = \frac{Baseline}{AutoTex},$$

*where Baseline uses the LLVM O3 pipeline.*

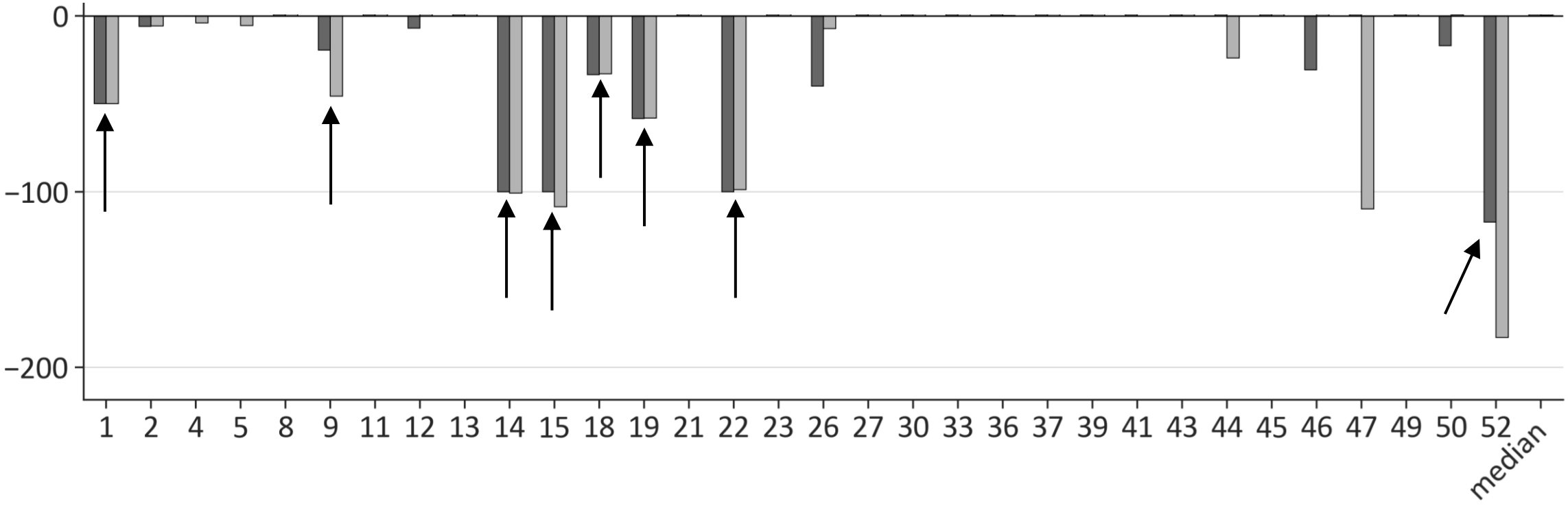


# Fetched Sector Reduction on Low-Utilization Kernels

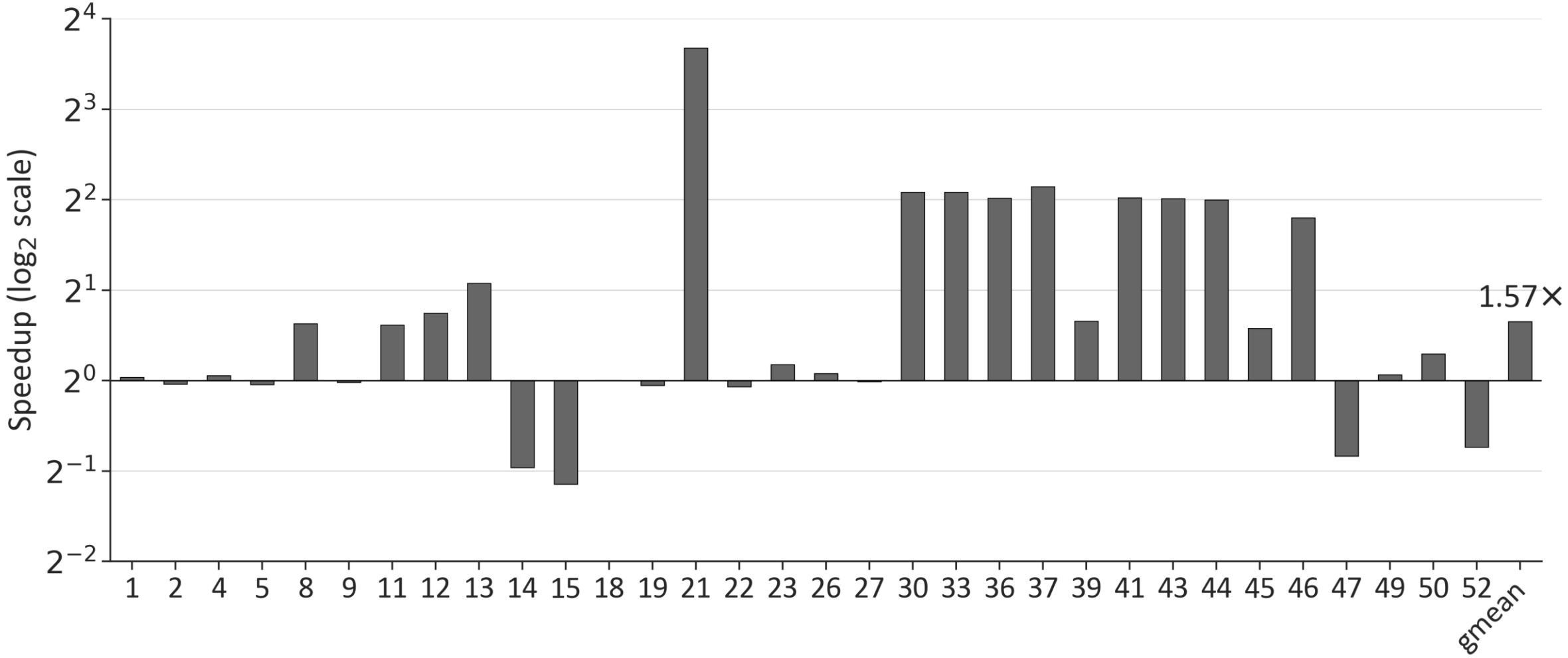


# False Positives in Access-Pattern Analysis

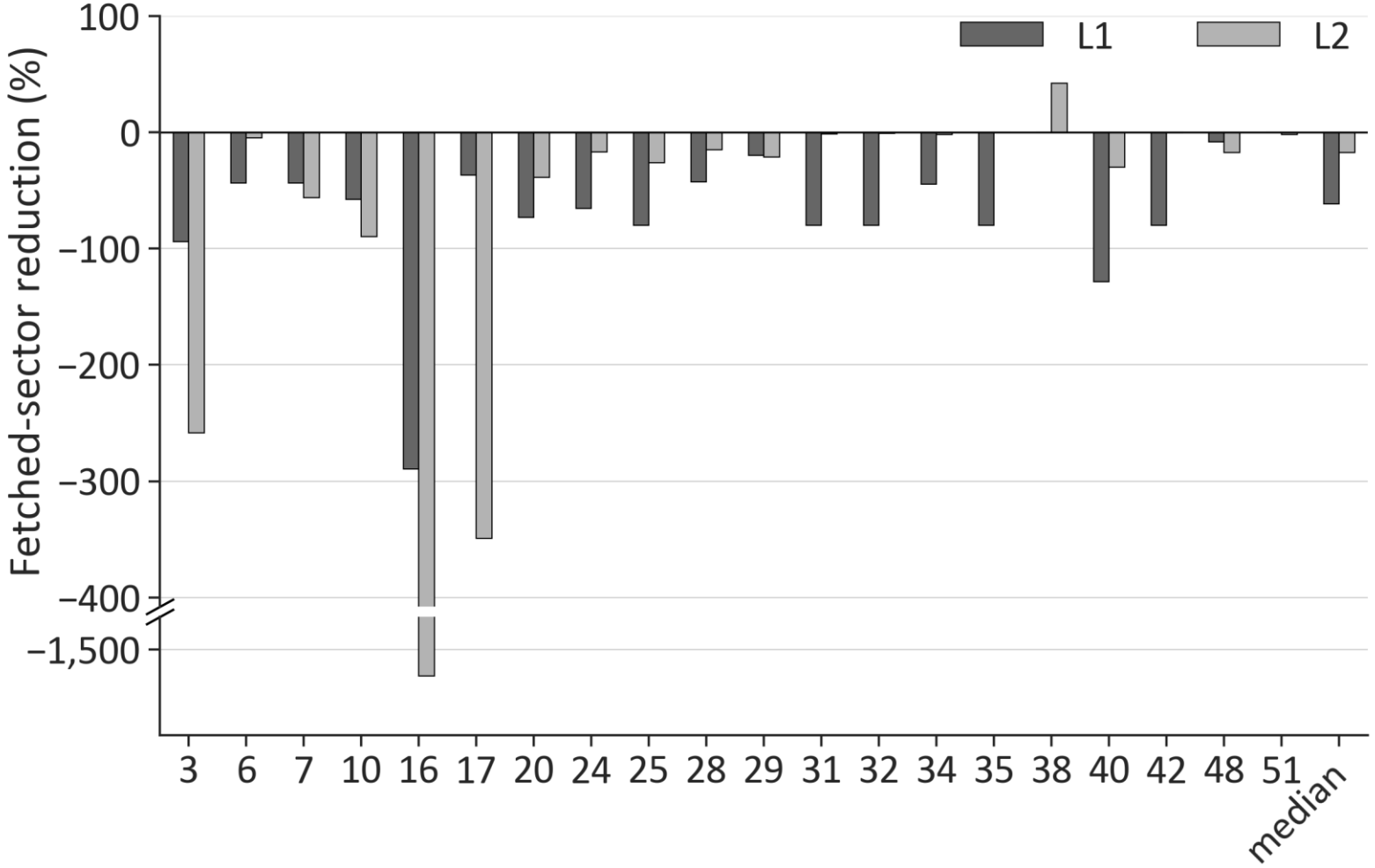
AutoTex currently treats arithmetic on thread indices as indicative of low utilization, e.g., $(x,\ y + t\ /\ r)$.
Such false positives can increase fetched sectors.



# SPEEDUP ON LOW-UTILIZATION KERNELS

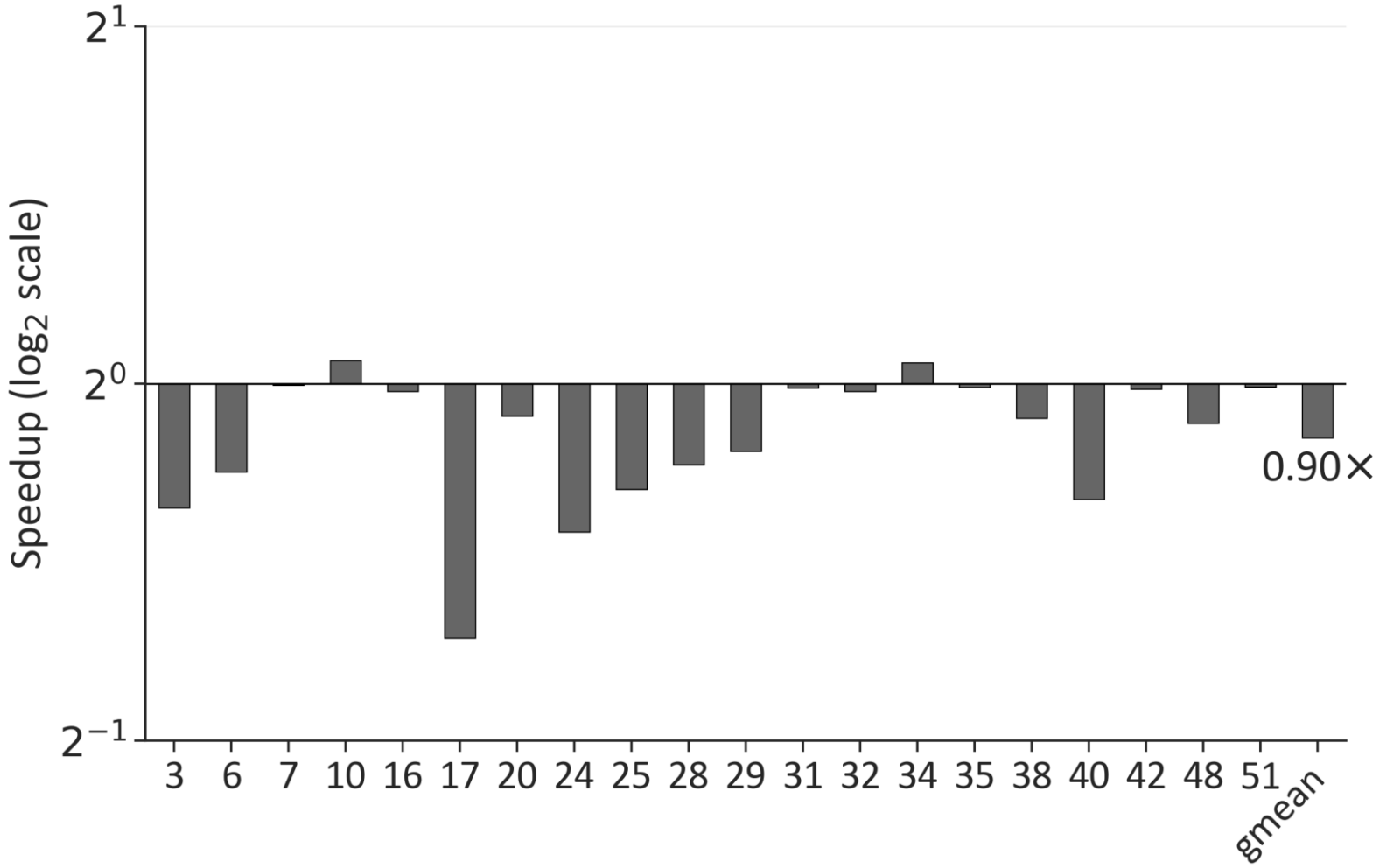


# Fetched-Sector Reduction on Sharing-Only Kernels




The shared data is already accessed with high intra-warp cache utilization,
so the blocked layout can increase fetched sectors.

# SPEEDUP ON SHARING-ONLY KERNELS



# Per-Kernel Evaluation Takeaways

- Blocked layout reduces fetched sectors and improves performance, while false positives can cause mis-optimization.
- Sharing-only transformations can instead increase fetched sectors and degrade performance.

How do these opposing effects translate to overall benchmark speedup?

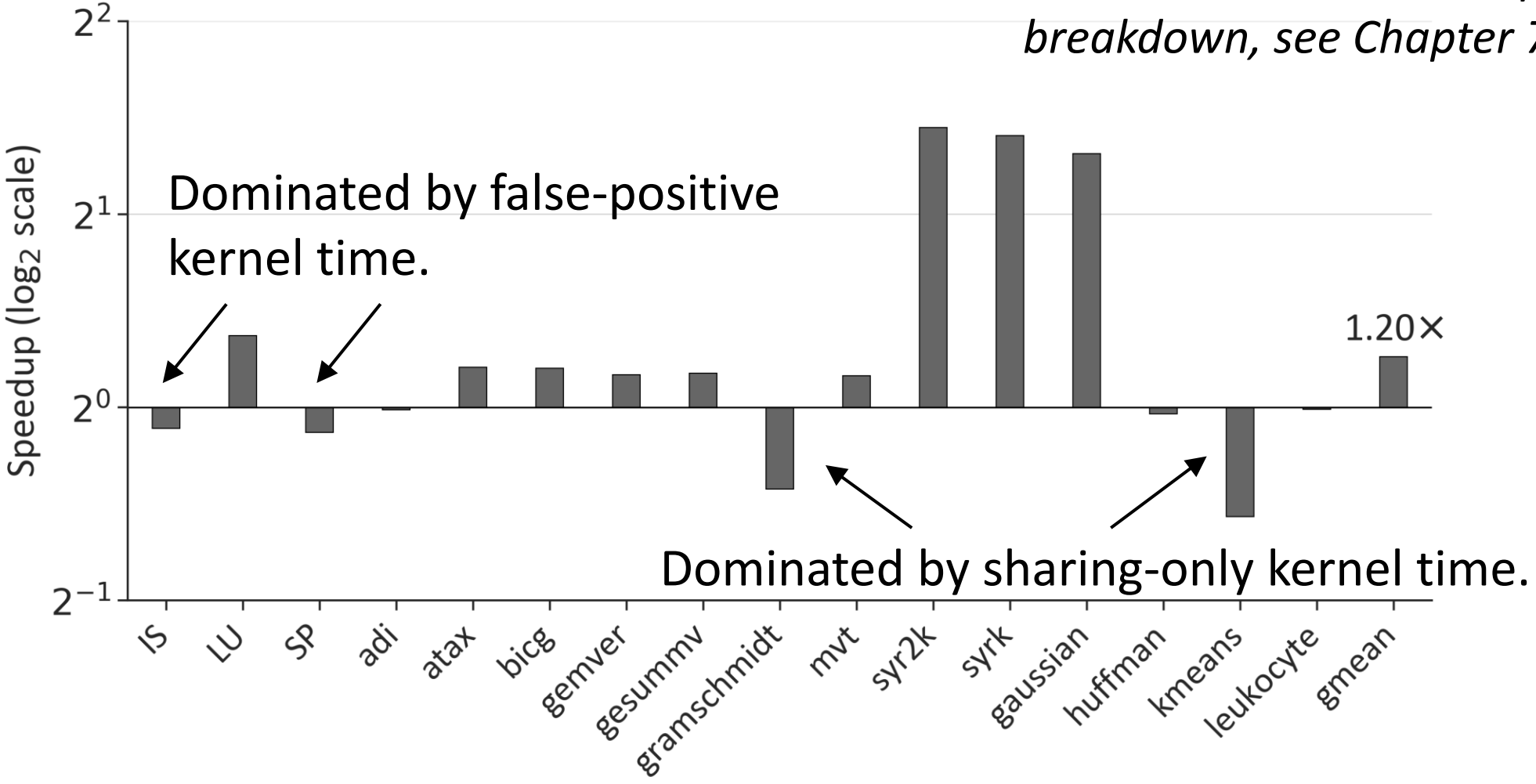
# Benchmark Runtime Components

In addition to kernel execution, benchmark time includes

- data allocation (`allocate_on_gpu`),
- data copy (`copy_to_gpu`, `copy_from_gpu`), and
- kernel-launch (`launch_kernel`) time.

# Overall Benchmark Speedup

*Note: For a detailed component breakdown, see Chapter 7.*



# Limitations of the Layout Mechanism

- 2D blocked layouts: up to 131,072 × 65,536 4-byte elements; 3D: up to 16,384 4-byte elements per dimension.
- Other specialized GPU hardware, particularly tensor cores and the Tensor Memory Accelerator, relies on linear layout.

# CONCLUSION

- **Objective:** Improve intra-warp cache utilization on GPUs.
- **Mechanism:** Exploit the hardware-managed blocked layout provided by texture hardware.
- **Method:** Use compiler analysis and transformation to transparently apply the blocked layout to data with low intra-warp cache utilization and access it through the texture unit.
- **Result:** 1.57× geometric-mean speedup on low-utilization kernels and 1.20× on transformed benchmarks.
- **Contribution:** Texture hardware can be repurposed as a compiler-exploitable memory-layout optimization.