SC26 · CHICAGO · NOVEMBER 2026

LEO: Tracing GPU Stall Root Causes via Cross-Vendor Backward Slicing

Yuning Xia · John Mellor-Crummey

Rice University

NVIDIA GH200 | AMD MI300A | Intel Max 1100 (PVC)

GPU supercomputers now span three vendors

	NVIDIA	AMD	Intel
Systems	Perlmutter · Alps	Frontier · El Capitan	Aurora
ISA	SASS	AMDGCN	Xe
PC sampling	CUPTI	ROCprofiler-SDK	Level Zero
Stall reasons	13	10+	8
Waits encoded as	Scoreboard barriers	Wait counters	SWSB tokens

Same code, different GPU → different bottleneck [Davis et al., ICS'25; Kwack et al., IPDPS'25]

Performance tuning now has to work on three different ISAs

A profiler finds the hot line, not the cause

Kripke (LLNL transport proxy) · LTimes kernel · RAJA

```
1  RAJA::kernel<ExecPolicy>(
2    RAJA::make_tuple(range_Moments, range_Directions, range_Groups, range_Zones),
3    [=] (IMoment nm, IDirection d, IGroup g, IZone z) {
4      phi(nm, g, z) += ell(nm, d) * psi(d, g, z);
5    });
```

96.7% of stall cycles on GH200

- Nsight Compute points at Line 4
- The warp is waiting on a *memory dependency*

Profilers report where the warp waits, not what it waits for

A profiler finds the hot line, not the cause

Kripke (LLNL transport proxy) · LTimes kernel · RAJA

```
1  RAJA::kernel<ExecPolicy>(
```



The profiler found the hot line!

…all done?…

Line 4 is a plain multiply-add. What is it actually waiting for?

The producer hides behind synchronization

AMD MI300A · AMDGCN (illustrative)

```
1  global_load_dword v1, v[10:11], off
2  global_load_dword v2, v[12:13], off
3  global_load_dword v3, v[14:15], off
4  s_waitcnt vmcnt(1)
5  v_fma_f32 v4, v1, v2, v0
```

stall sampled here

Which loads is it waiting for?
No register says.

Vendor	How a wait names its producers
NVIDIA	Scoreboard barrier B1–B6 set and awaited in control bits
AMD	Counter: “until ≤ N vector-memory ops are in flight”
Intel	Software-scoreboard token $0–$31 on the send and the consumer

Register-only slicing dead-ends at every vendor’s waits

…but we do have tools… right?

Tool	Stall site	Direct cause	Transitive chain	Wait tracing	Blame
Nsight Compute	✓	×	×	×	×
rocprofv3	✓	×	×	×	×
VTune / unitrace	✓	✓	×	partial	×
GPA [CGO'21]	✓	✓	×	✓	✓
GPUscout · DrGPU	✓	×	×	×	×
LEO (this work)	✓	✓	✓	✓	✓

NVIDIA only

✓ supported · × not supported

No tool follows the chain through synchronization on all three vendors

Key insight

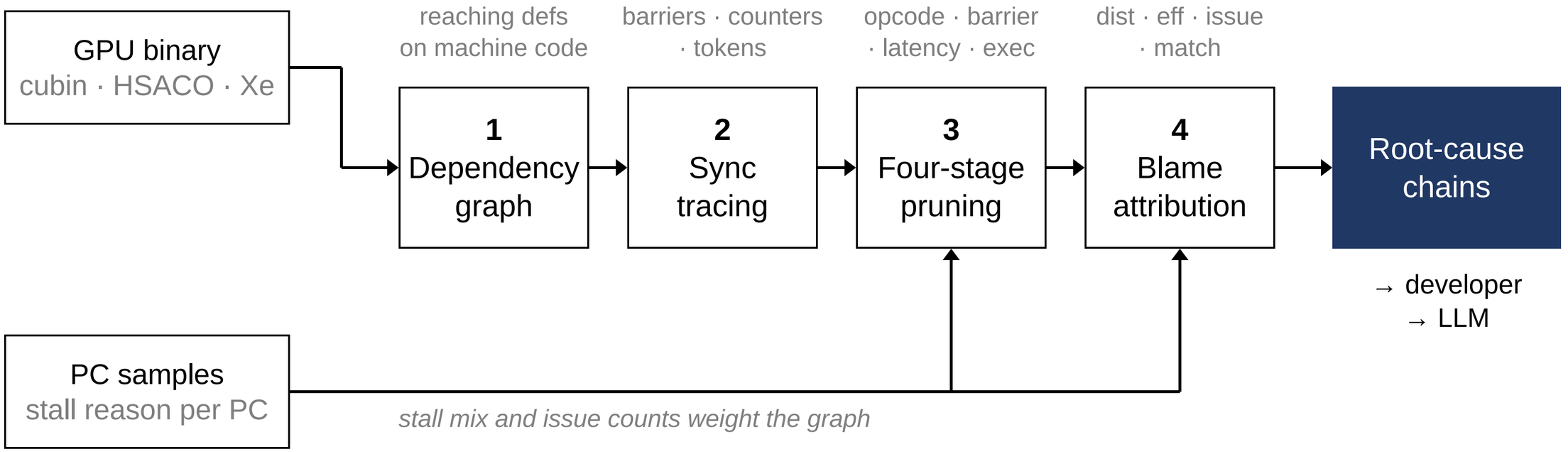
Every stall waits on dependences the compiler already wrote into the binary.

Registers + vendor waits (barriers · counters · tokens) → *one dependence graph*

PC samples decide which dependences mattered → *ranked root causes*

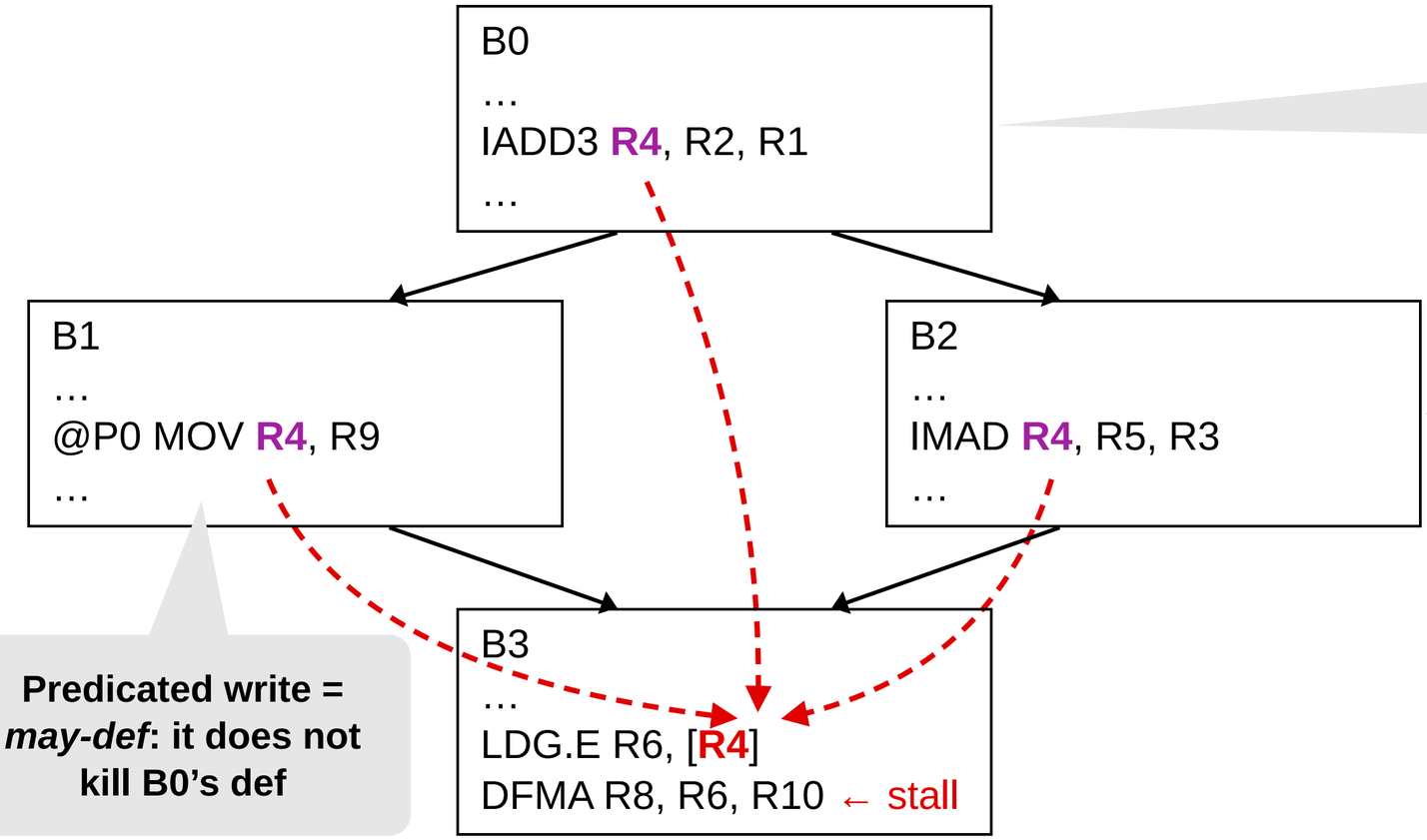
Static structure from the binary, statistical evidence from the samples

LEO slices backward from every stall



One analysis, three ISAs: only the edge leaves are vendor-specific

Step 1: Keep every producer that may reach the stall



IADD3 is address math with no samples, yet it stays in the graph

No SSA on machine code: reaching definitions per use, over the CFG

Predicated write = *may-def*: it does not kill B0's def

R4 def

R4 use

dashed: reaching definition

Over-approximate first; the samples prune later

Step 2: Vendor waits become typed dependence edges

NVIDIA · scoreboard barrier

```
1  LDG.E R2, [R10]  set B1
2  IMAD R8, R4, R5, RZ
3  FADD R9, R2, R8  wait {B1}
```

Link a wait to every setter of the awaited barrier → **mem_barrier**

AMD · wait counter

```
1  global_load_dword v1
2  global_load_dword v2
3  global_load_dword v3
4  s_waitcnt vmcnt(1)
5  v_fma_f32 v4, v1, v2, v0
```

vmcnt(N): all but the N newest pending ops → **mem_waitcnt**

Intel · SWSB token

```
1  send.ugm r20  {$3}
2  add r50, r51, r52
3  mul r60, r20, r30  {$3.dst}
```

Link to the latest setter of $3 on each path → **mem_swsb**

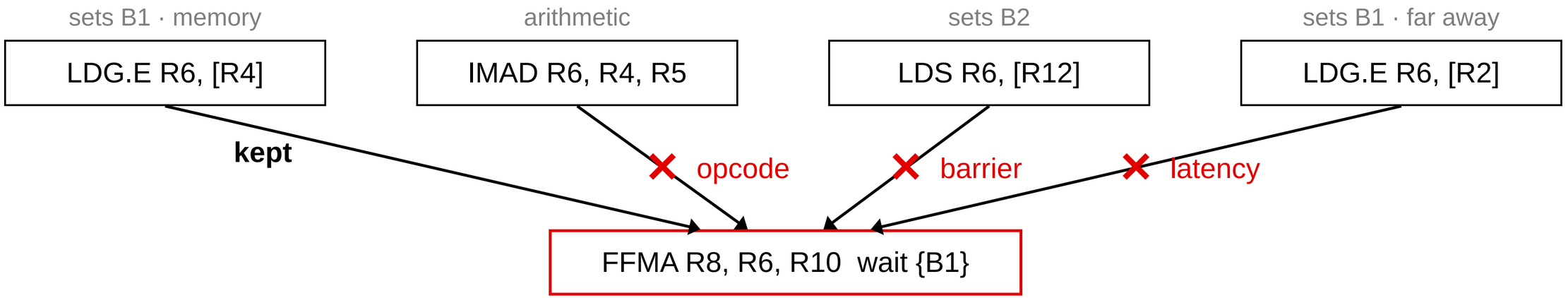
Named waits (NVIDIA, Intel) behave like pseudo-registers

Counted waits (AMD) are positions in a FIFO queue

Compiler-inserted waits are real dependences, so they skip opcode and latency pruning

Step 3: Drop edges that cannot explain the stall

- **Opcode:** the producer must match the measured stall type
- **Barrier** (NVIDIA): the producer's barrier must be in the consumer's wait mask
- **Latency:** drop if every CFG path issues enough cycles to hide the latency
- **Execution:** optionally drop producers that never executed



Memory-dependency stall at the FFMA → only one load can explain it

Step 4: Blame follows the measured evidence

stall cycles at j, from PC samples

over the producers that survived

$$blame(i) = S(j) \cdot w(i) / \Sigma k\ w(k)$$

$$w(i) = dist \cdot eff \cdot issue \cdot match$$

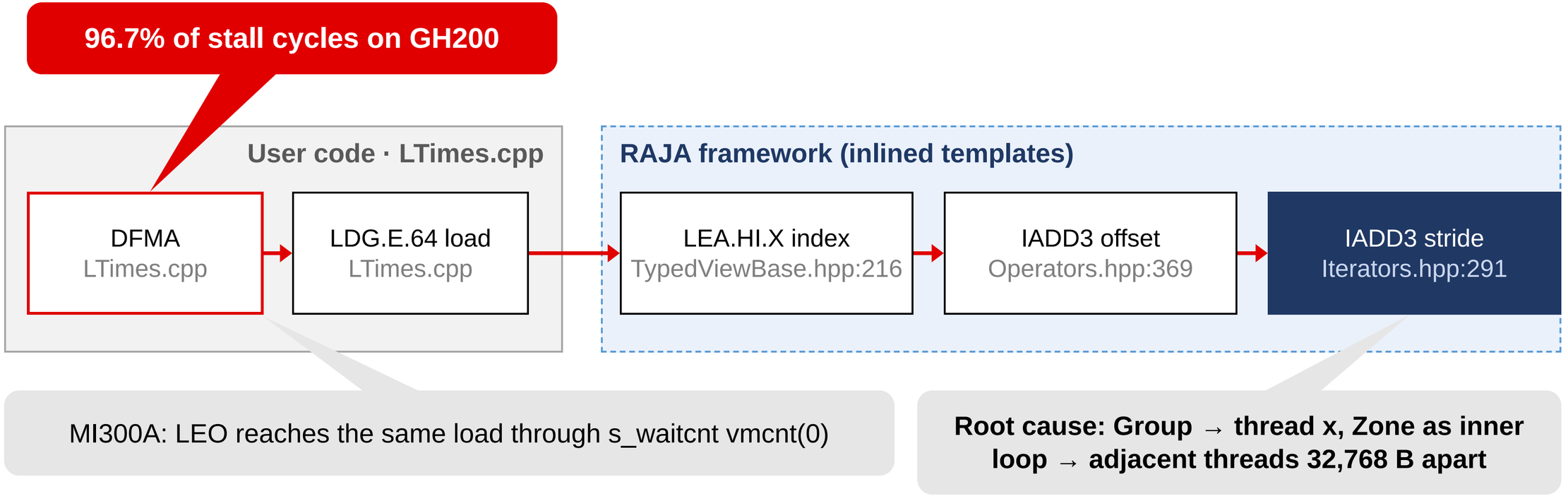
from GPA: closer, less efficient, more issued → more

new: weight by the measured stall mix

Toy example: S(j) = 1,000, 80% memory	dist	issue	match	w	blame
LDG.E R6 (memory edge, 2 away)	0.5	0.5	0.8	0.20	**667**
IADD3 R7 (arithmetic edge, 1 away)	1.0	0.5	0.2	0.10	333

Distance alone would blame IADD3 2 : 1; the measured stall mix flips it to the load

Back to Kripke: the cause is three files away



Fix: swap Zone ↔ Group in LTimes.h (2 lines) → **11.67×** on GH200 · **1.90×** on MI300A

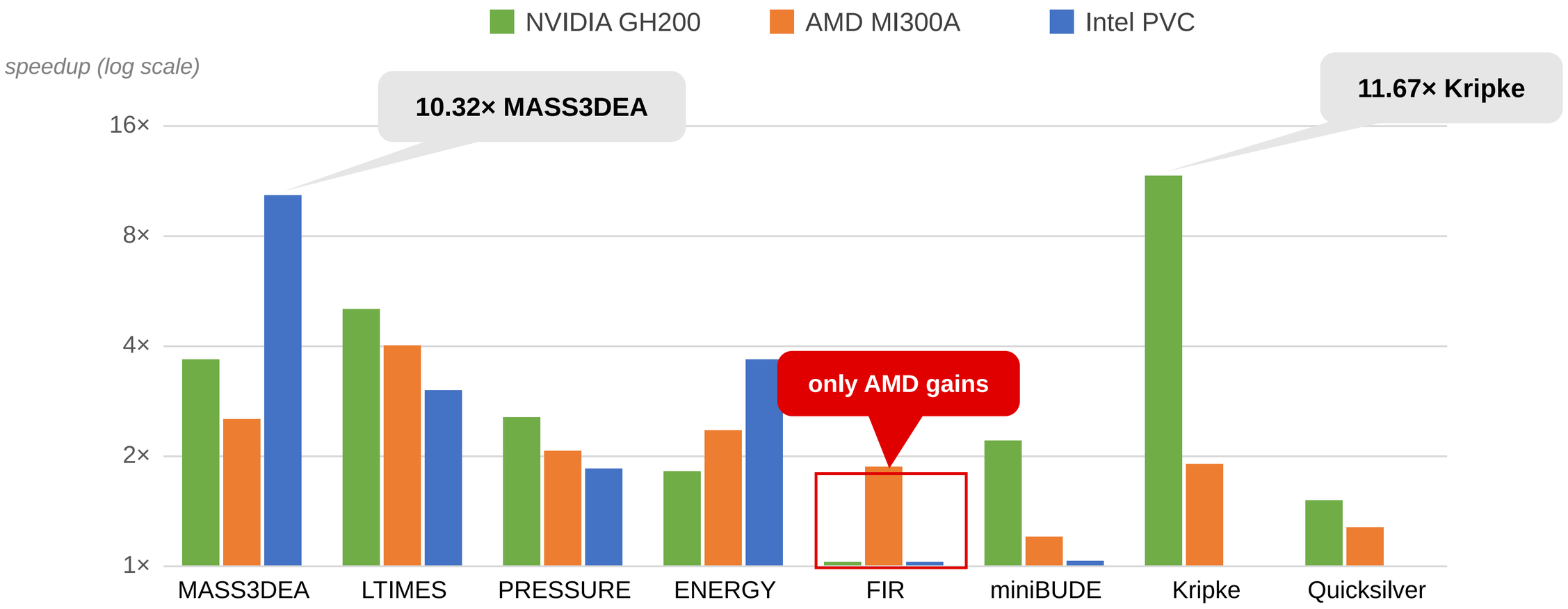
Evaluation: 21 workloads on three vendors' GPUs

	NVIDIA GH200	AMD MI300A	Intel Max 1100
Architecture	Hopper	CDNA 3	Xe-HPC (PVC)
Compute units	132 SMs	228 CUs	56 Xe-cores
Software	CUDA 12.8	ROCm 6.3	Level Zero

- **21 workloads:** 15 RAJAPerf kernels (CUDA · HIP · SYCL) + XSBench, miniBUDE, LULESH, Quicksilver, Kripke, llama.cpp
- **Plus** expert-tuned HipKittens ML kernels
- **Protocol:** one minimal fix per kernel, confined to LEO's top-ranked chain · 10 runs · every speedup > 1.05× has $p < 0.01$

The question: do fixes on LEO's top chain pay off on every vendor?

LEO-guided fixes speed up all three vendors



Geomean over 21 workloads: **1.73×** GH200 · **1.74×** MI300A · **1.82×** PVC

Observation 1: Same kernel, three GPUs, three root causes

RAJAPerf MASS3DEA: finite-element mass-matrix assembly

NVIDIA GH200

FP64 multiply chains

DMUL → DMUL: 52.3% of stall cycles

3.66×

AMD MI300A

Register pressure

189 VGPRs, 0% occupancy: 84% of cycles at s_waitcnt

2.51×

Intel PVC

SLM-fence latency

through SYCL accessors: 28% of stalls

10.32×

One fix removes all three: precompute basis-function values in registers

Conversely, FIR is compute-bound on NVIDIA but memory-bound on AMD → only AMD gains (1.86×)

Observation 2: Fixes transfer when the bottleneck is regular



- **Same fix (11):** regular access; tiling (LTIMES, 3MM, GEMM) or kernel fusion (PRESSURE, ENERGY)
- **Vendor-specific (6):** the stall mechanism itself is architecture-specific

miniBUDE: irregular loads on all three	**Root cause → fix**	**Speedup**
NVIDIA GH200	Address math delays loads → __ldg + float4	2.20×
AMD MI300A	64 lanes load the same atoms → scalar broadcast	1.20×
Intel PVC	SYCL accessor overhead → USM pointers	1.03×

Regular bottlenecks get portable fixes; irregular ones need vendor-specific fixes

Cross-file chains also explain proxy apps and ML kernels

Quicksilver

A multiply in CollisionEvent.hh waits on a load routed through MacroscopicCrossSection.hh to NuclearData.hh

Fix: Inline the cross-file call

1.51× GH200 · 1.27× MI300A

llama.cpp · mmq.cuh

MI300A: 70.9% stall ratio, 0% occupancy. GH200: no dominant hotspot

Fix: Tile 128 → 64, direct stores

1.12× MI300A

HipKittens · RMSNorm

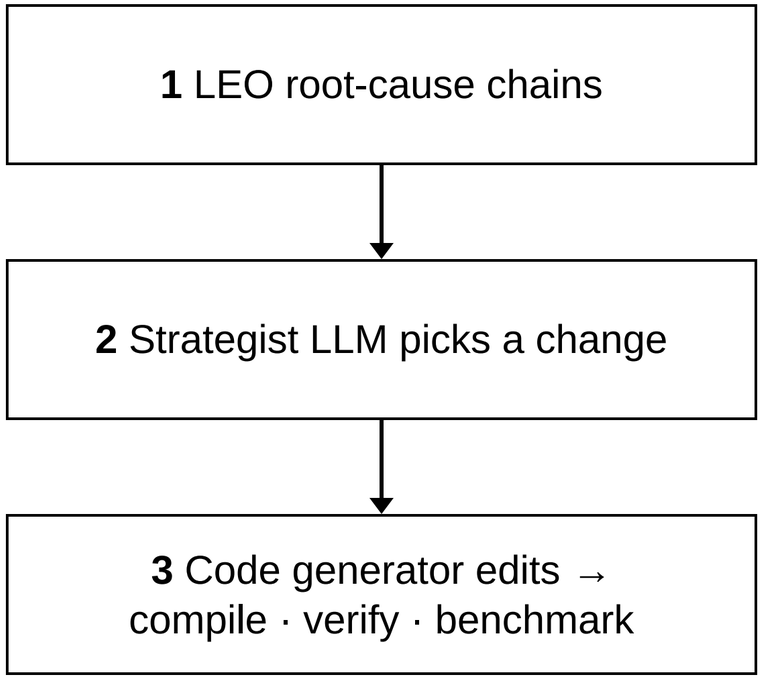
BF16 vector loads lowered to scalar global_load_ushort; 20–58% of stalls on memory

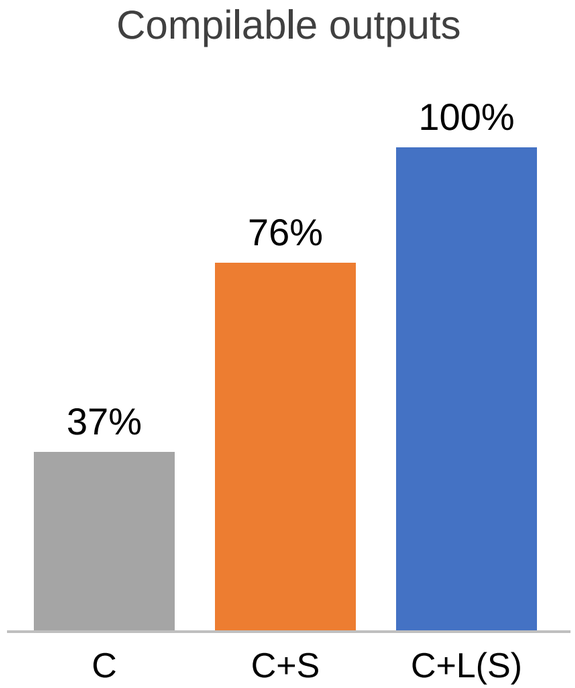
Fix: Software pipelining

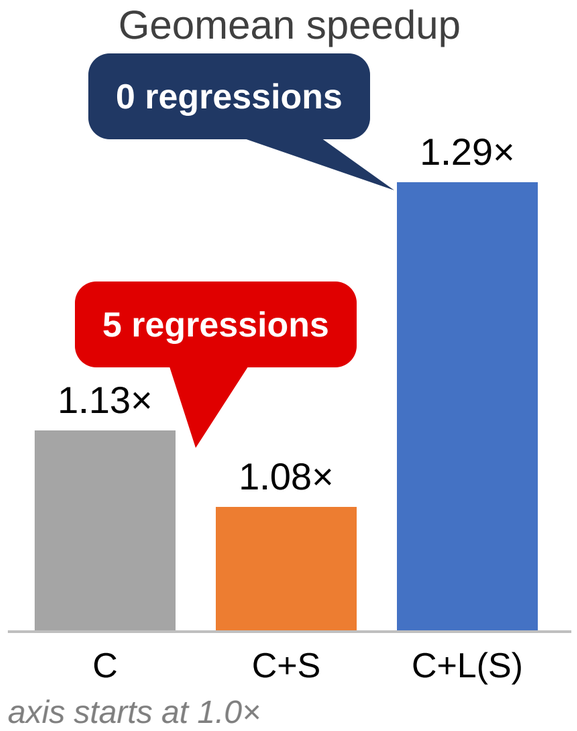
1.07–1.24× MI300A

Framework and compiler layers hide bottlenecks that only cross-file slicing exposes

LLMs tune better with LEO’s chains than with raw stall counts







C: code only · C+S: + raw stall counts · C+L(S): + LEO chains

Gemini 3.1 Pro · 15 RAJAPerf CUDA kernels · GH200 · best of 5

Raw stall counts mislead the model: VOL3D → 0.36×, PRESSURE → 0.85×

Conclusions

UNDERSTANDING stalls

Profilers show where, not why	Sync hides the producer	Three vendor encodings

TRACING root causes

Backward slicing on machine code	Barriers · counters · tokens	4-stage pruning + evidence-weighted blame

DELIVERING speedups

1.73–1.82× geomean on 3 vendors	Kripke 11.67× from 2 lines	LLM tuning: 1.29×, 0 regressions

Next: indirect-memory dependences · predicate-aware blame · multi-model LLM study

Thank you!

Questions?

Paper
arxiv.org/abs/2604.20032

Contact
yuning.xia@rice.edu