Dissecting the Ampere GPU Architecture through Microbenchmarking

GTC 2021

Zhe Jia
Peter Van Sandt
Marco Maggioni
Jeffrey Smith
Daniele P. Scarpazza

High Performance Computing R&D Team

CITADEL

IMPORTANT DISCLAIMER

The results and conclusions in this presentation are Citadel's. Inclusion in the GTC program does not indicate any endorsement or confirmation of any of Citadel's results or conclusions by NVIDIA.

This presentation reflects the analyses and views of the authors. No recipient should interpret this presentation to represent the general views of Citadel or its personnel. Facts, analyses, and views presented in this presentation have not been reviewed by, and may not reflect information known to, other Citadel professionals.

Assumptions, opinions, views, and estimates constitute the authors judgment as of the date given and are subject to change without notice and without any duty to update.

Citadel is not responsible for any errors or omissions contained in this presentation and accepts no liability whatsoever for any direct or consequential loss arising from your use of this presentation or its contents.

ALL TRADEMARKS, SERVICE MARKS AND LOGOS USED IN THIS DOCUMENT ARE TRADEMARKS OR SERVICE MARKS OR REGISTERED TRADEMARKS OR REGISTERED SERVICE MARKS OF CITADEL.

Takeaways

- Double effective available L2 by grouping work by L2 partitions
- Explore novel opportunities for lock-free algorithms with no penalty for contention
- Design programs to maximize the utility of the memory hierarchy and operations unique to Ampere

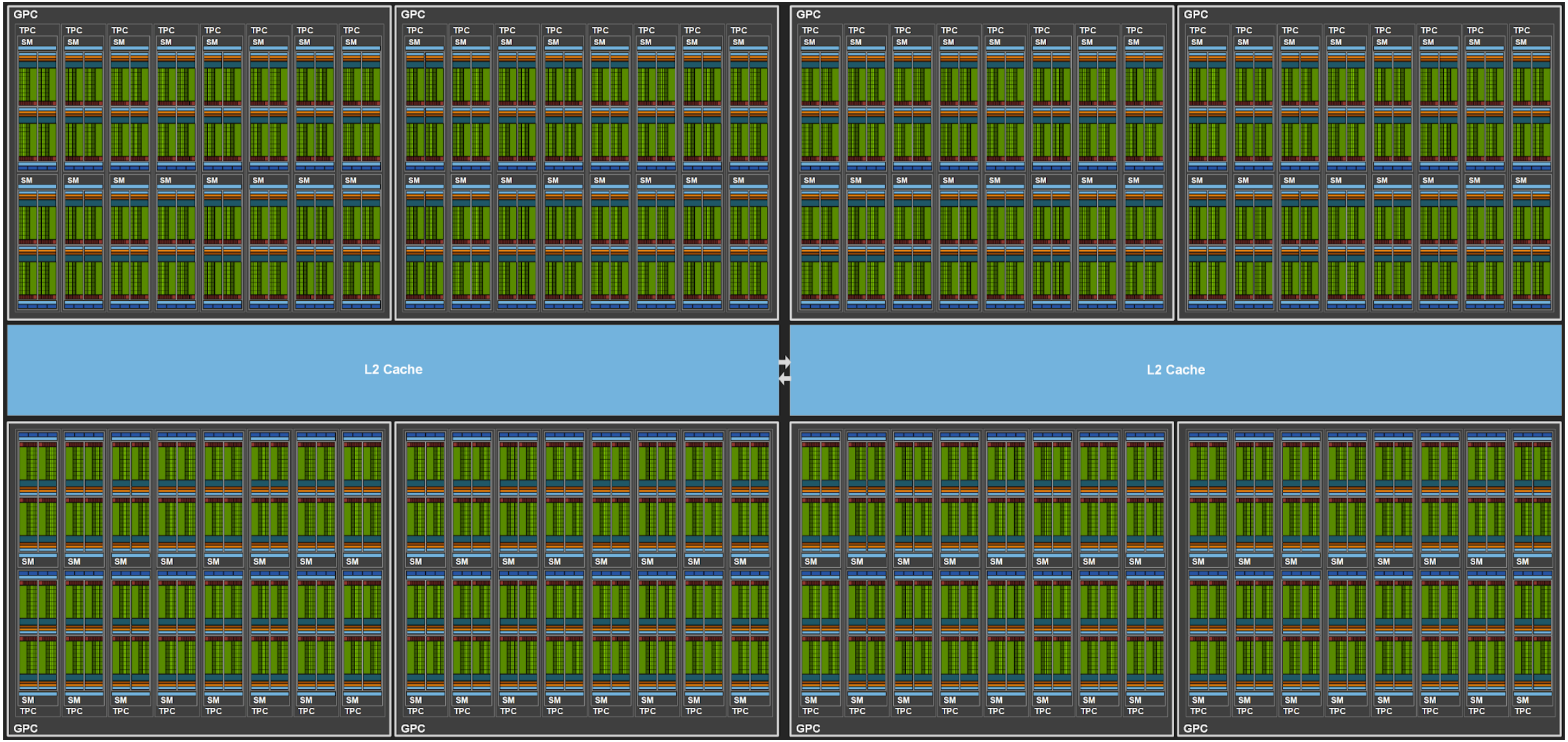
Plan: Four deep dives

1. Grouping work by L2 Partition
2. Atomics
3. Sparse matrix multiplication
4. A100 vs V100

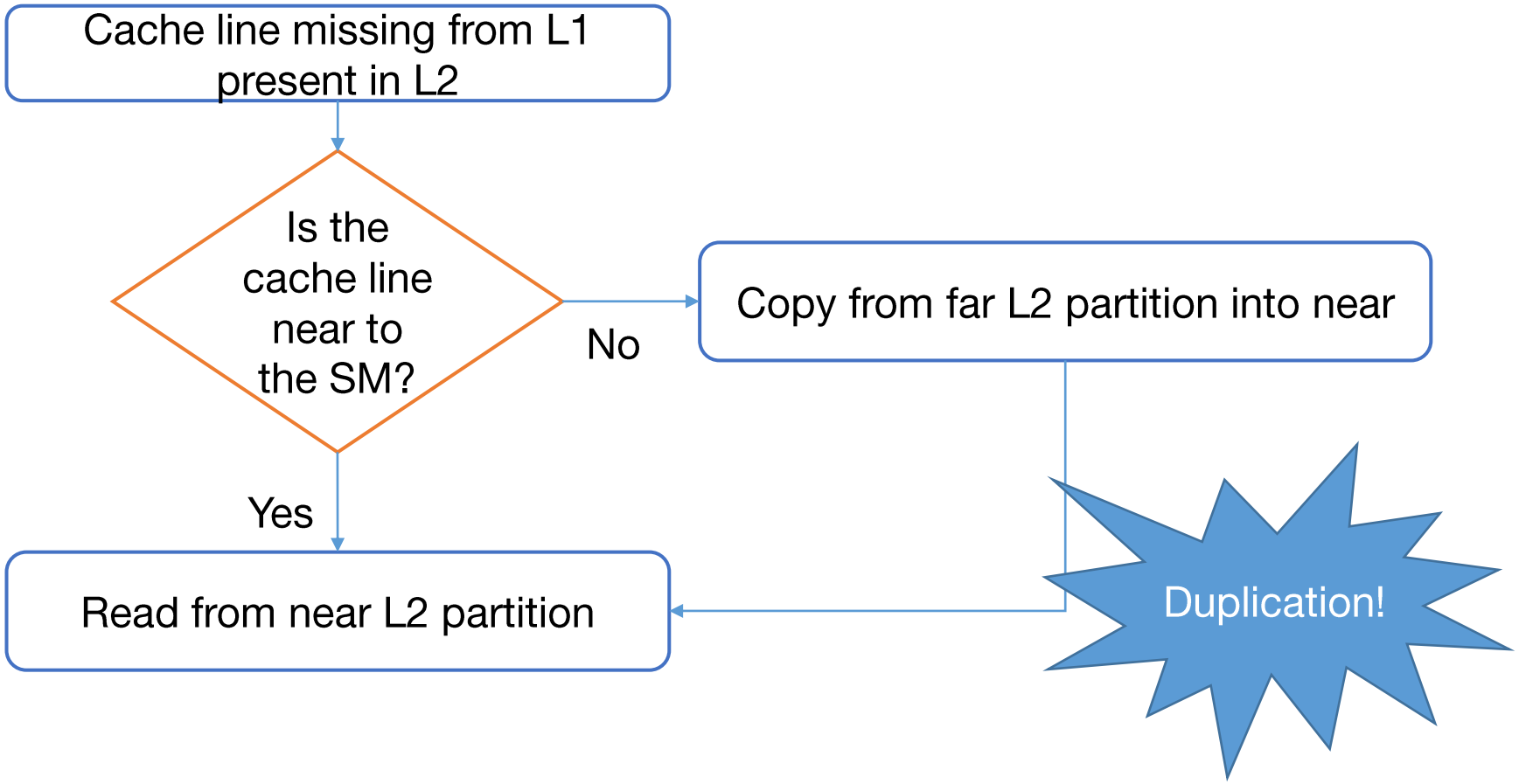
Deep dive #1

Grouping work by L2 partition

Ampere uses a split L2 design



Anatomy of a L2 hit



Double your effective available L2 space

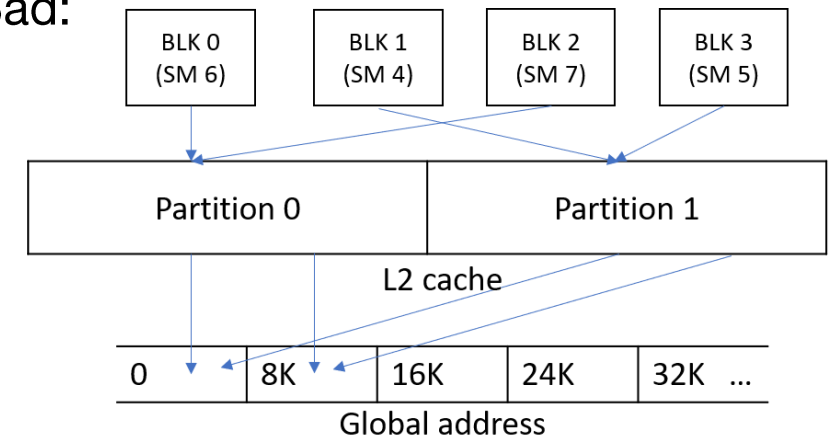
- Grouping thread blocks by L2 partition reduces duplicated cache lines
  - Each SM has a near L2 partition detectable with micro-benchmarks
  - Cache lines stay duplicated if they are used from different partitions
  - Use each cache line only from one partition
  - Thread blocks identify their near partitions using their SM ids
  - Each SM reads its work set into its near partition

- Duplicated cache lines halve available L2
  - Each partition can store 20MiB
  - To effectively use 40MiB, cache lines must not be duplicated

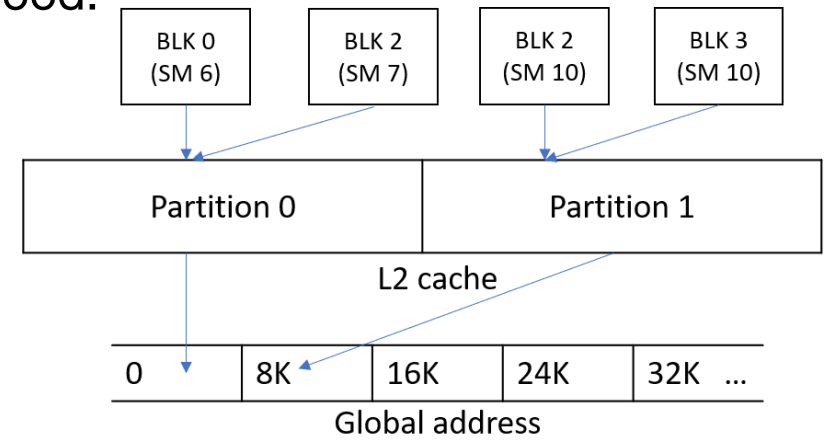
Case study: pairs of partition-aware blocks

- Result: +85% throughput when blocks are grouped by partition
- … because they use available L2 more effectively
- Experiment design
  - Each thread block loads a block size segment and does a simple computation on each element
  - Each block size segment is operated on by a pair of blocks
  - Pairs of blocks iterate over enough data to exceed L1

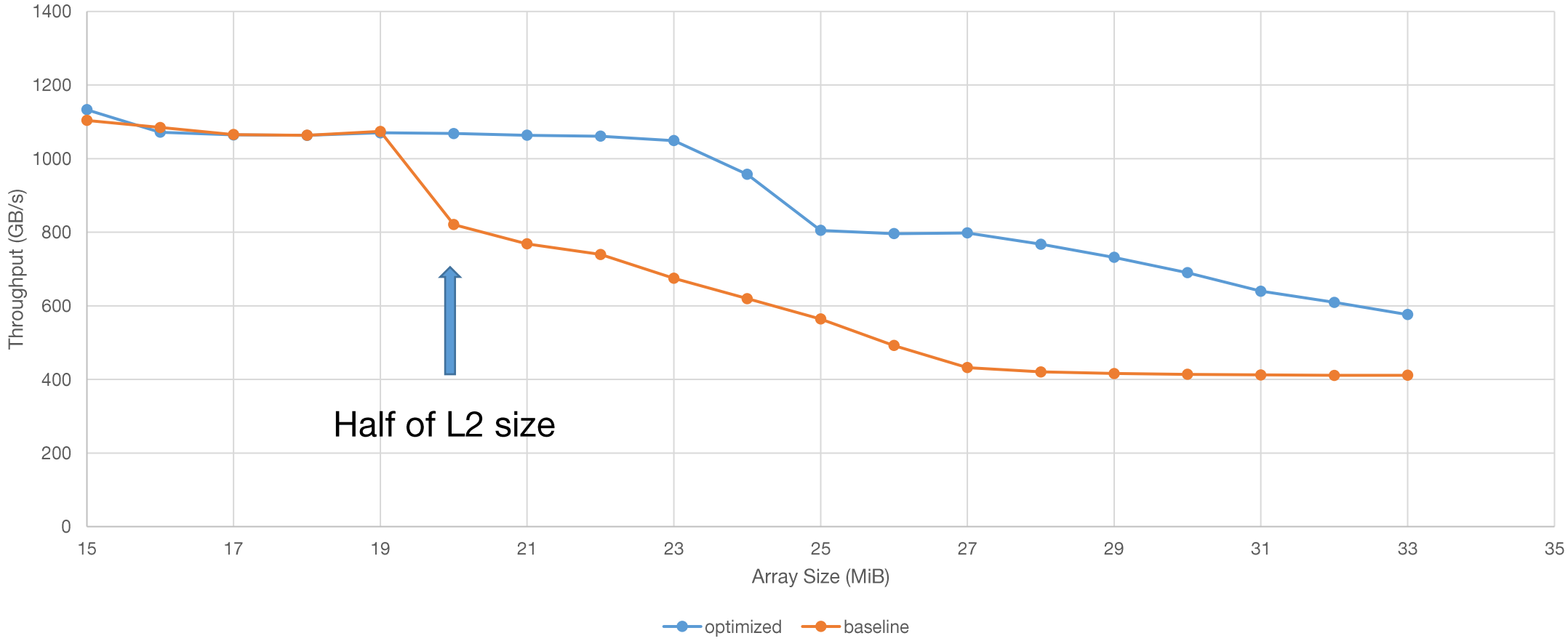
Bad:



Good:

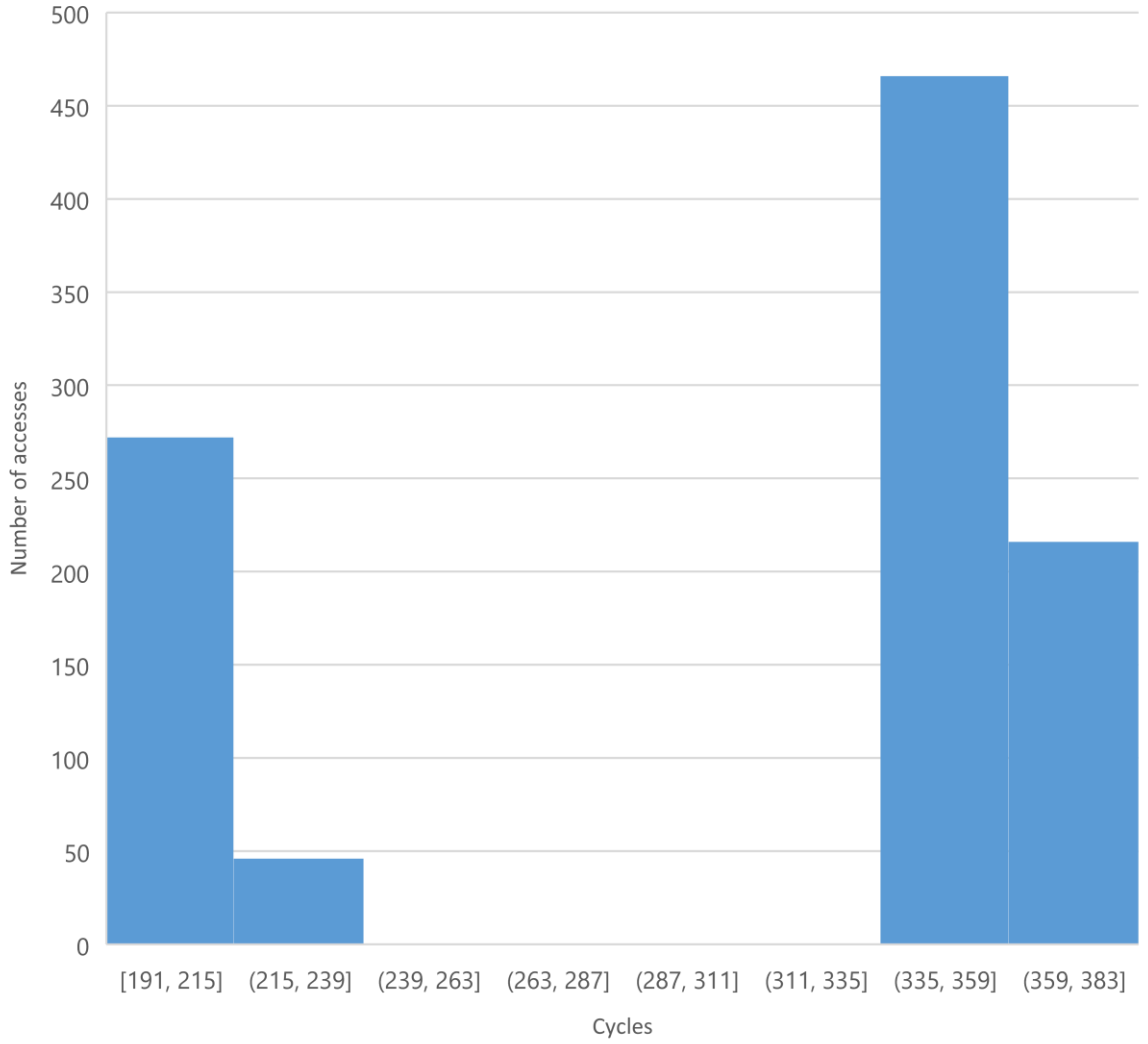


Case study: pairs of partition-aware blocks



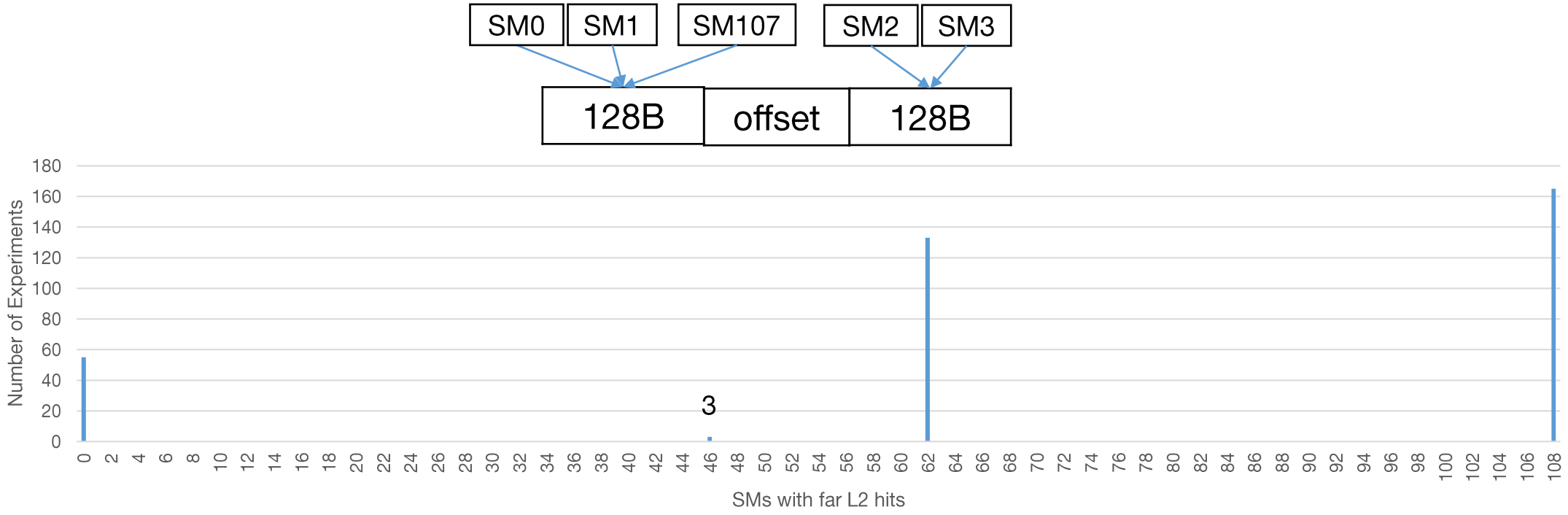
Benchmarks show a clear “near” and “far” L2 hit latency

- Studying: L2 latency seen by a single SM
- Results:
  warmed reads from one SM are distributed bimodally around 200 and 350 cycles across different addresses
- Experiment design
  - Scan a global array twice with 1 block and 1 thread bypassing higher levels of cache, including near L2 partitions
  - Measure latency for each access in the second scan
  - Each element is accessed exactly twice in total



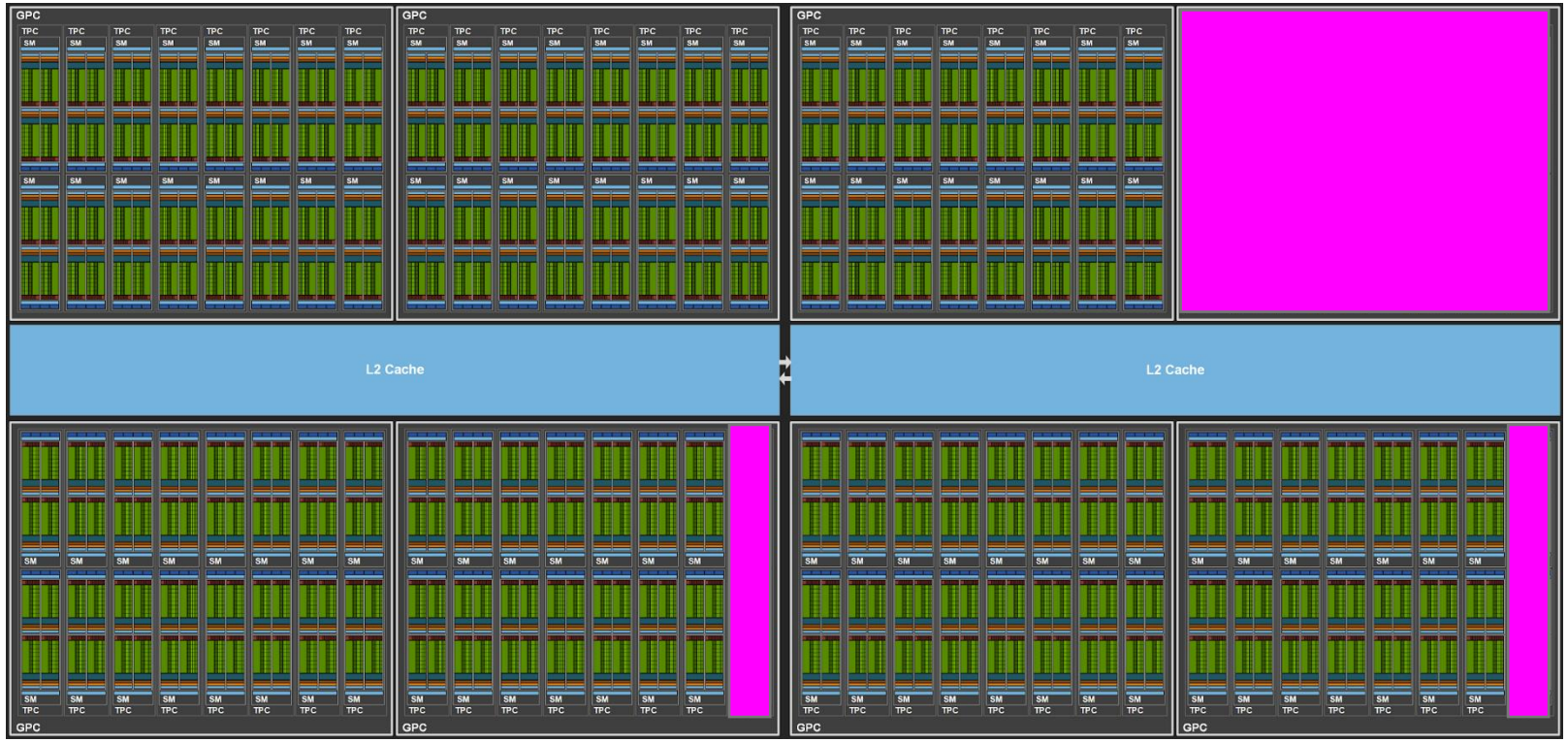
Latency distribution over all SMs

- SMs show 4 distinct frequencies of near and far latencies corresponding to SM counts in each partition
- SMs associated with partition 0 repeatedly read one 128 B segment from global memory, while let the SMs associated with partition 1 read from another 128 B segment
- We vary the byte offset between the two segments between executions, and measure access latency of each thread block in each execution



Global address space evenly distributed across L2 partitions: Discussion

- The hardware split between L2 partitions is asymmetric
- Asymmetry is not problematic because… global memory address space is **evenly** split between L2 partitions
- Guessing: NVIDIA went out of their way to route HBM traffic to partition slices evenly



Hypothetical SM layout to summarize our conjectures
for visualization purposes only - no implied faithfulness to actual topology

Global address space evenly distributed across L2 partitions: Proof

- Heatmap shows **global memory homogeneous access**
- Virtual address mapping to L2 partitions in 8-KiB granularity
- Sub-slices of L2 are associated to HBM memory controllers

Address (B)	SM0 Cycles	SM2 Cycles	SM3 Cycles	SM1 Cycles	SM6 Cycles	SM7 Cycles	SM4 Cycles
3E6000	245	392	400	244	235	245	375
3E6200	248	391	398	243	236	246	374
3E6400	240	405	394	223	231	228	388
3E6600	242	405	394	224	229	228	387
3E6800	233	394	384	235	243	219	400
3E6A00	235	394	384	234	241	218	401
3E6C00	237	381	390	253	246	238	387
3E6E00	232	381	391	251	246	237	386
3E7000	243	403	395	223	233	230	387
3E7200	241	404	393	223	233	229	386
3E7400	250	382	380	258	259	242	385
3E7600	246	383	380	257	259	243	386
3E7800	235	383	389	251	244	237	384
3E7A00	233	384	390	250	244	237	385
3E7C00	245	394	401	244	236	246	377
3E7E00	249	394	402	244	235	246	377
3E8000	385	231	239	404	394	387	234
3E8200	383	231	244	401	394	388	237
3E8400	391	236	227	392	399	376	240
3E8600	389	237	229	391	399	377	240
3E8800	396	246	241	381	388	387	232
3E8A00	400	245	237	381	387	385	228
3E8C00	395	242	249	395	388	399	225
3E8E00	399	243	251	394	387	398	228
3E9000	395	239	246	394	388	399	223
3E9200	398	241	246	394	388	397	223
3E9400	385	230	241	405	396	391	235
3E9600	387	229	238	404	396	387	234
3E9800	378	240	236	391	388	375	242
3E9A00	380	241	242	390	389	372	242
3E9C00	397	245	237	382	388	388	229
3E9E00	399	246	240	383	388	386	230
3EA000	251	404	396	230	242	237	388
3EA200	247	405	396	229	241	237	388
3EA400	238	392	399	238	229	239	376
3EA600	242	392	399	237	227	238	376

8 KiB

8 KiB

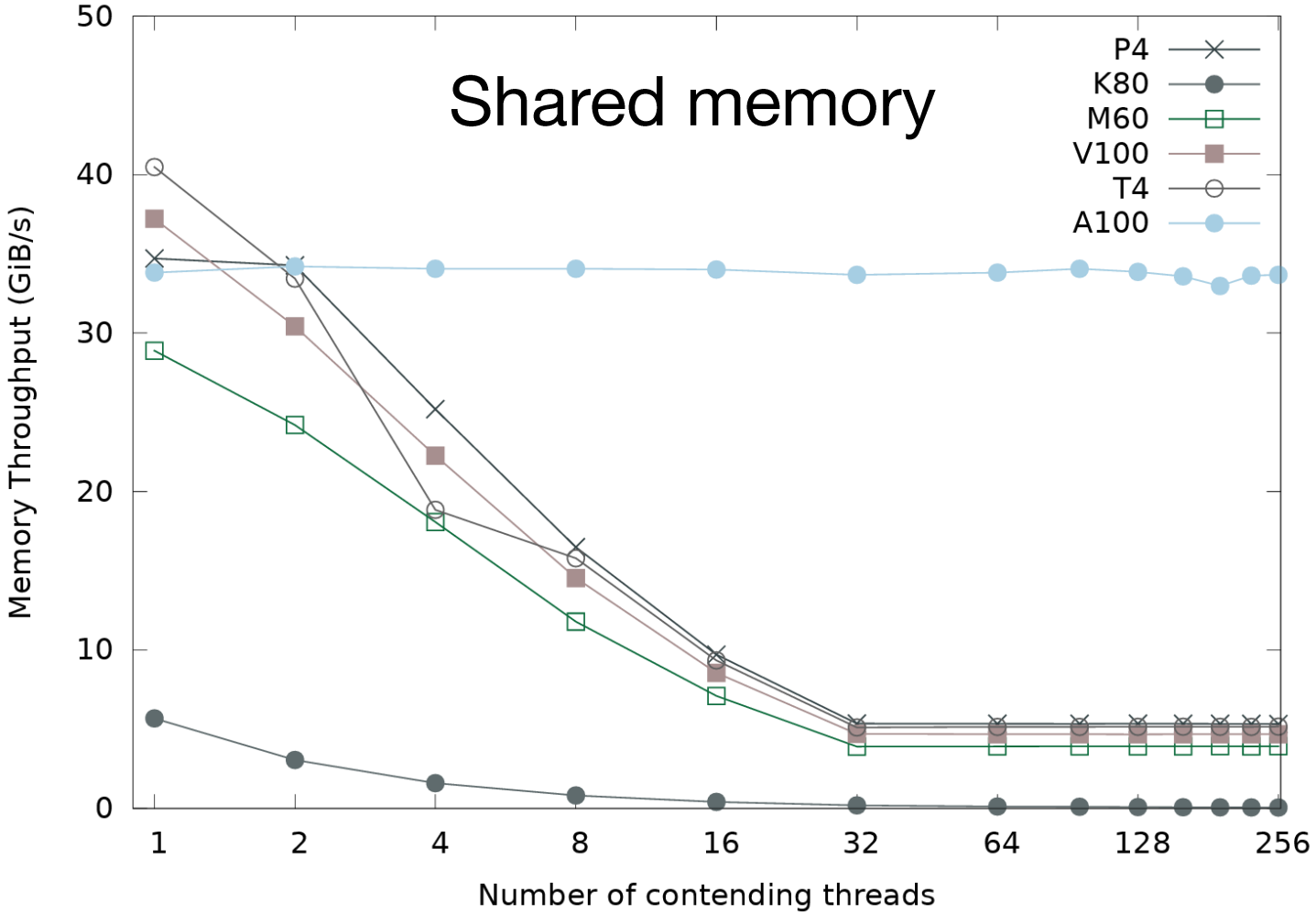
P0 P1 P0 P1

Deep dive #2

Atomics

Shared memory atomics: a new, contention-free increment

- **Ampere introduces an atomic increment instruction: ATOMS.POPC.INC**
- **Immune to contention**
  on prior generations,
  contention reduces throughput
- This feature does not extend to other operations
  – see next slides
- Additions other than 1 rely on old ATOMS.ADD instruction
  and suffer from contention

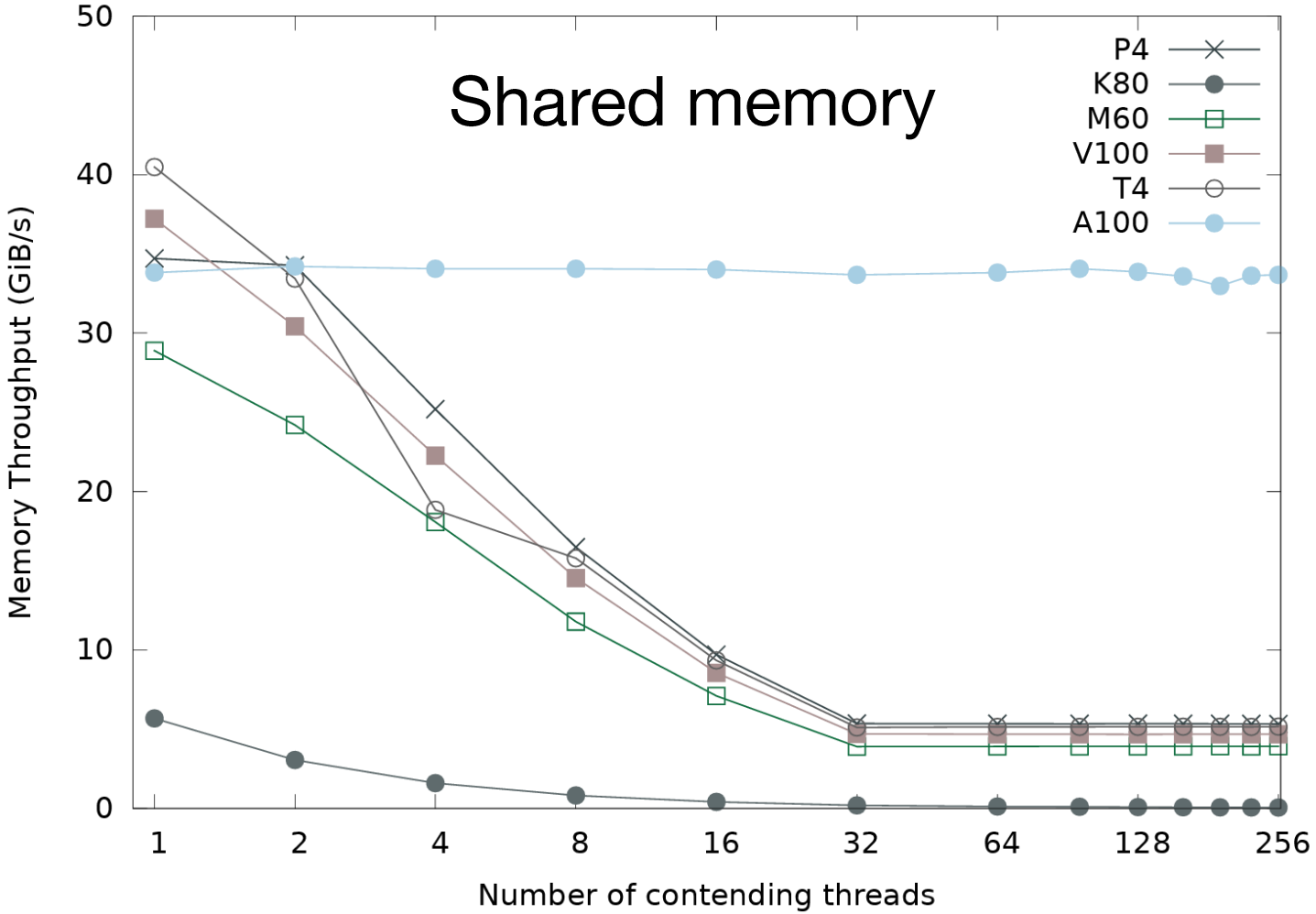


Shared memory atomics: a new, contention-free increment

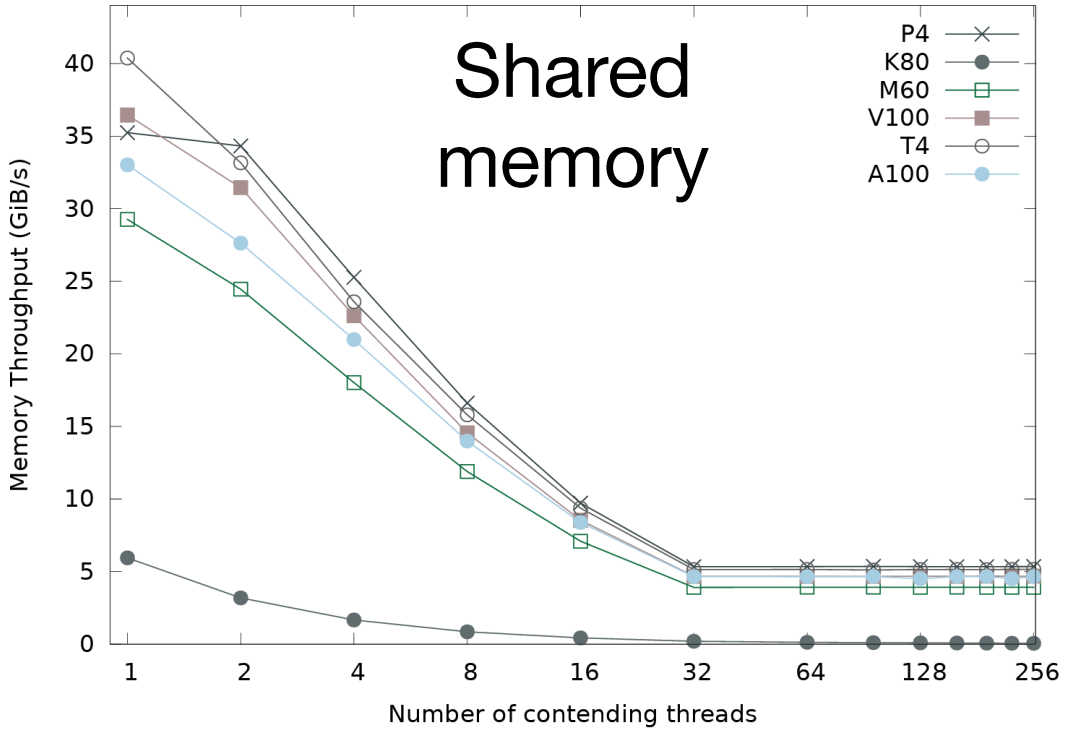
Experimental Proof

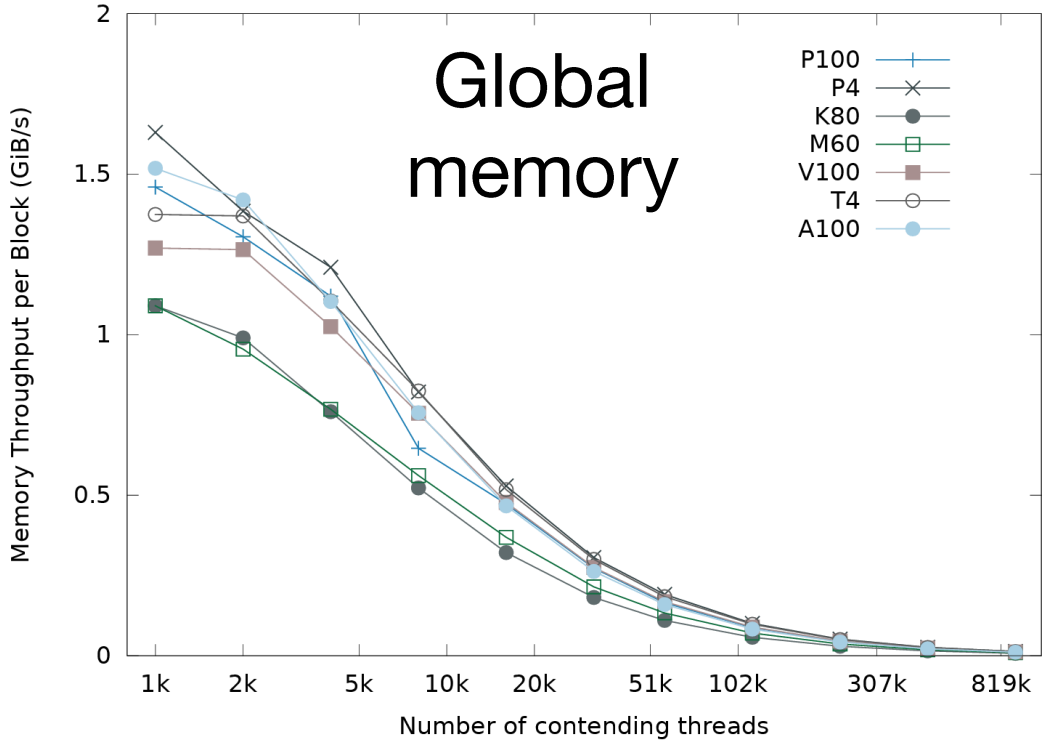
Benchmark:

- Each thread atomically increments a shared memory location by an immediate value of 1
- N threads update the same address
- Non-contending threads update sequential addresses
- Increasing thread count from 1 to 256
- One block with 256 threads



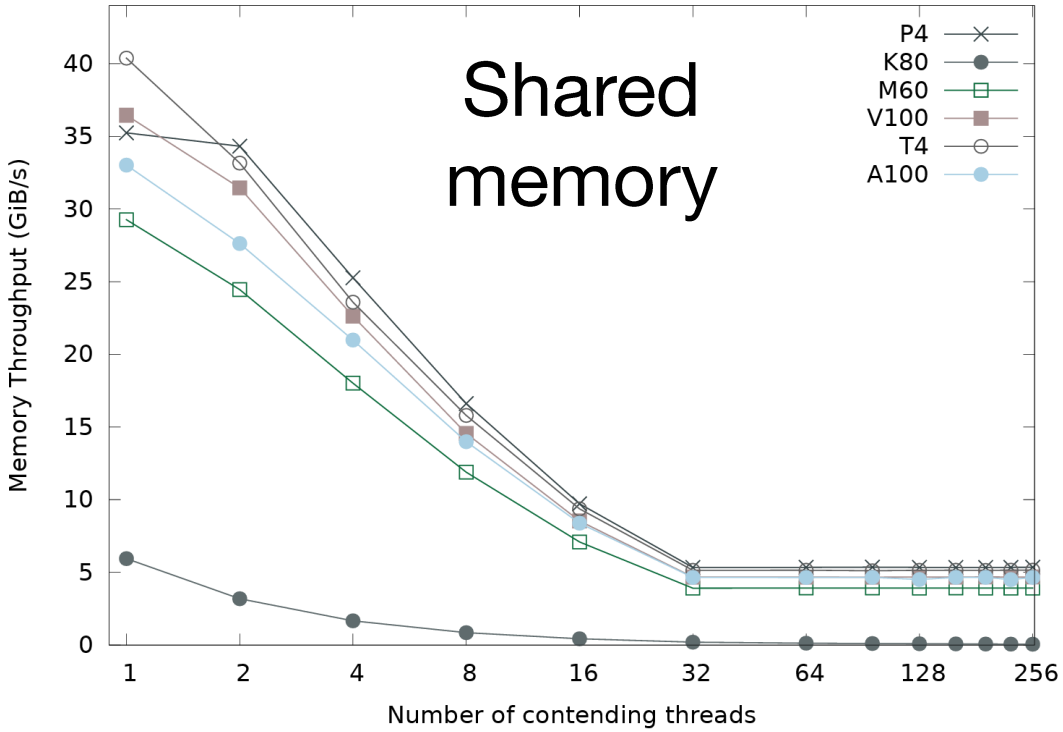
All other atomics still suffer from contention



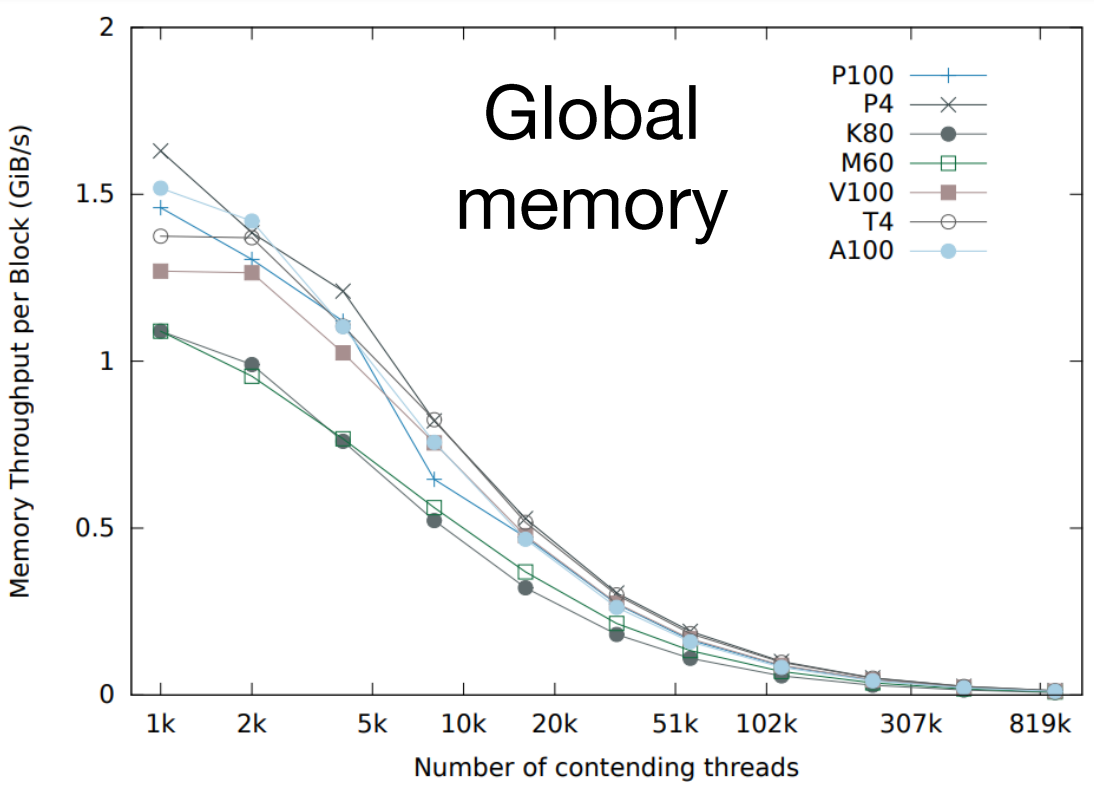


- Throughput suffers from contention on both shared and global memory
- Ampere throughput
  - on shared memory: slightly worse than V100
  - on global memory: slightly better than V100
- P4 and T4 GPUs: best throughput, due to high clock frequencies

Other atomics still suffer from contention: Proof



- atomicAdd()
- Contending threads update the same address
- Non-contending threads access sequential addresses
- Increase contending threads from 1 to 256
- One block with 256 threads



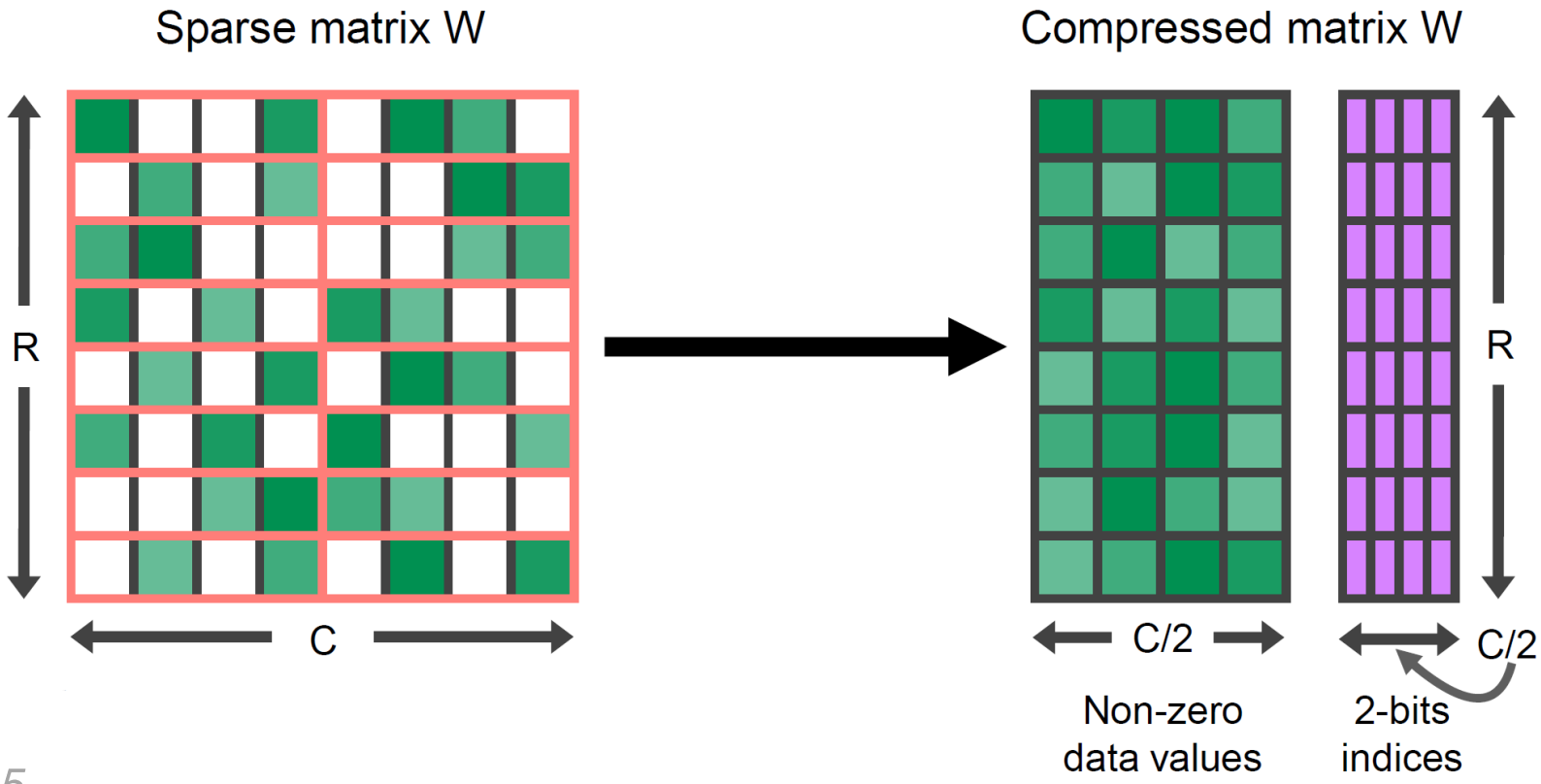
- atomicAdd( )
- Each block has 1024 threads
- All threads and blocks access same global address
- We vary block count from 1 to 896

Deep dive #3

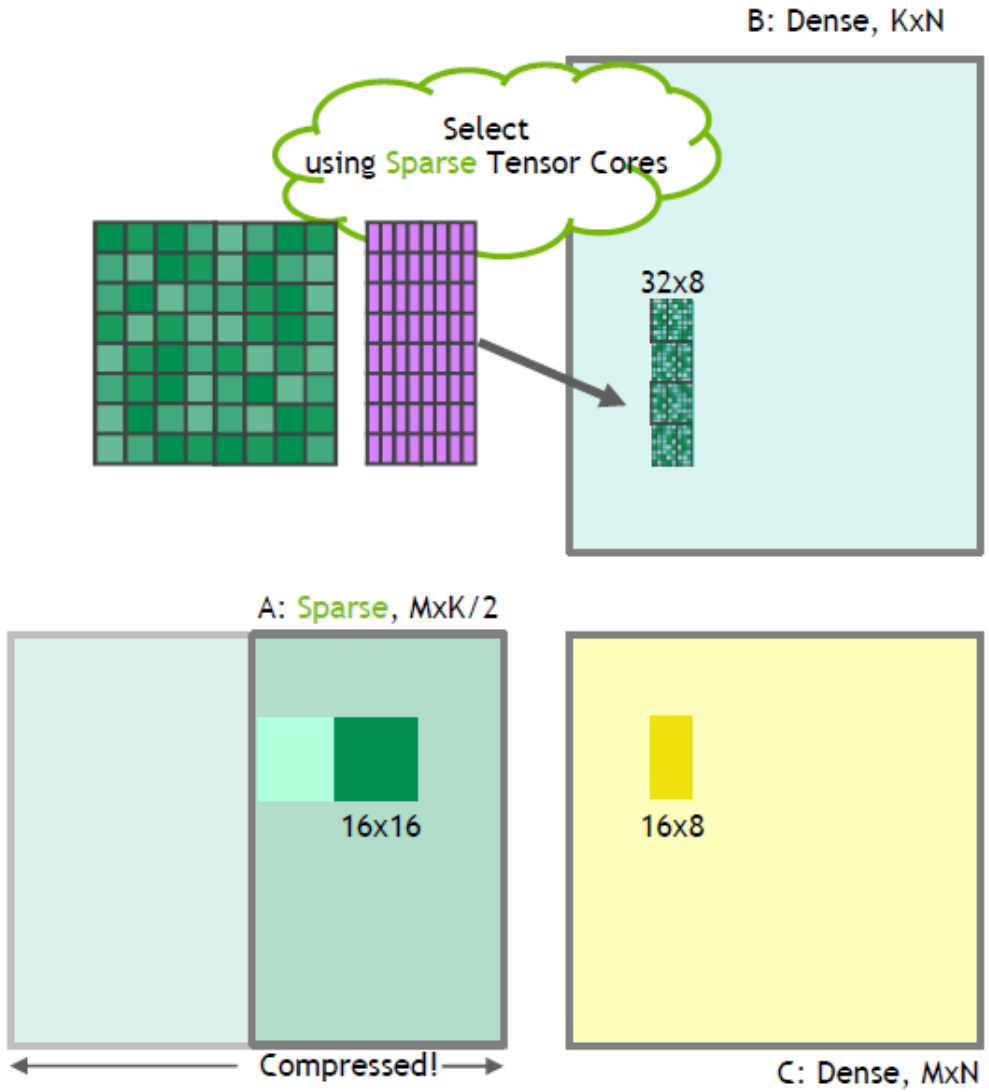
Sparse Matrix Multiplication

Fine-grained sparse matrix-matrix multiplication: Format

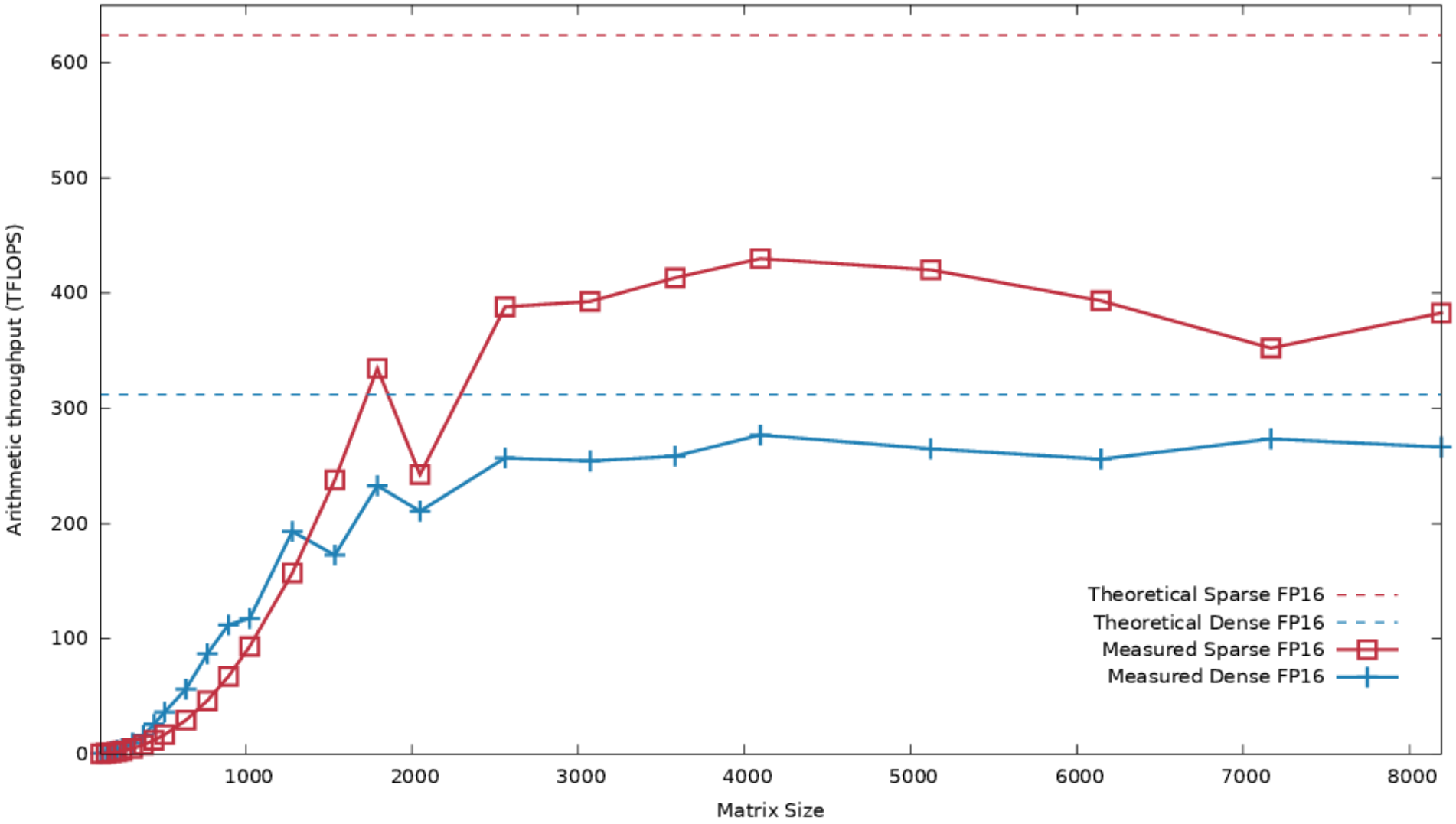
- Ampere introduces a new, hardware primitive for sparse GEMM acceleration
  - Fine-grained rather than **block-oriented**
  - Twice as fast as dense in the right conditions
- It supports a **specific** form of sparse GEMM that requires operands to be in a pre-determined format
  - At most 2 non-zeros in every contiguous group of 4 values



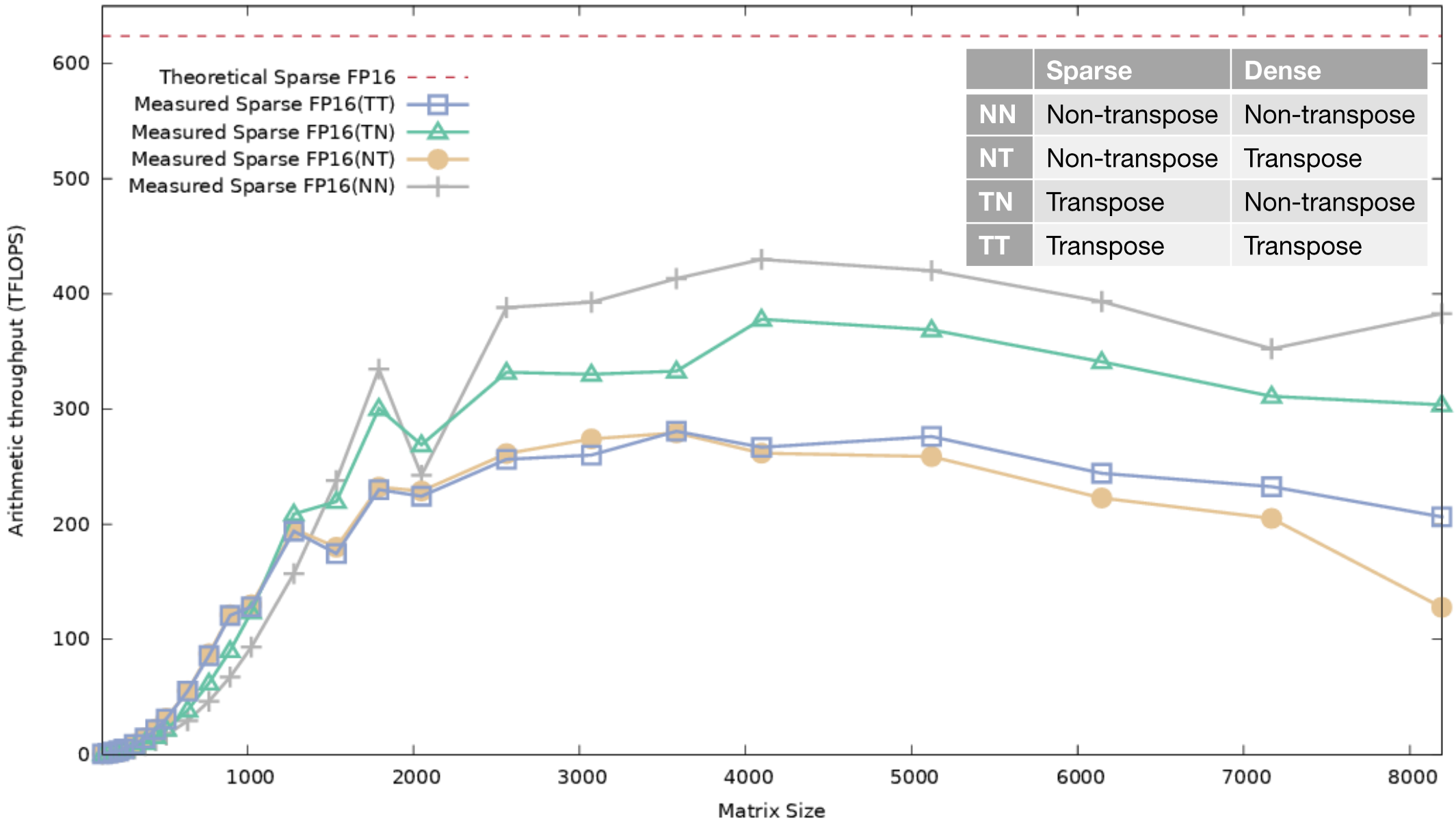
Fine-grained sparse matrix-matrix multiplication: Transpose



Sparse vs. Dense GEMM – best-case performance

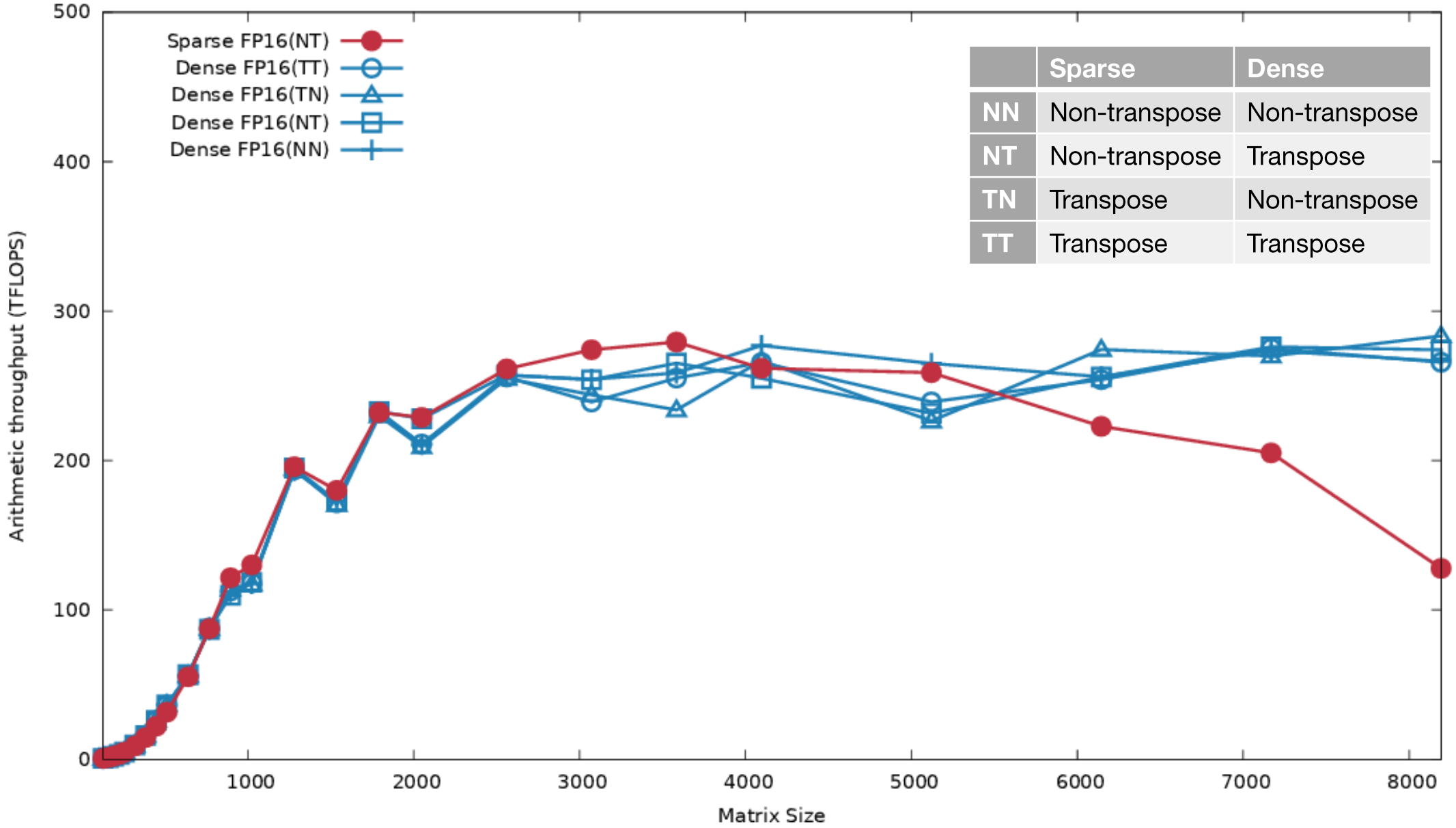


Sparse GEMM with transposed operands



	Sparse	Dense
NN	Non-transpose	Non-transpose
NT	Non-transpose	Transpose
TN	Transpose	Non-transpose
TT	Transpose	Transpose

Sparse vs. Dense GEMM “in the wrong conditions”



	Sparse	Dense
NN	Non-transpose	Non-transpose
NT	Non-transpose	Transpose
TN	Transpose	Non-transpose
TT	Transpose	Transpose

Deep dive #4

A100 vs. V100

Why compare A100 against V100?

Most recent GPU given:

- Full-length, full-height
- Similar power profile

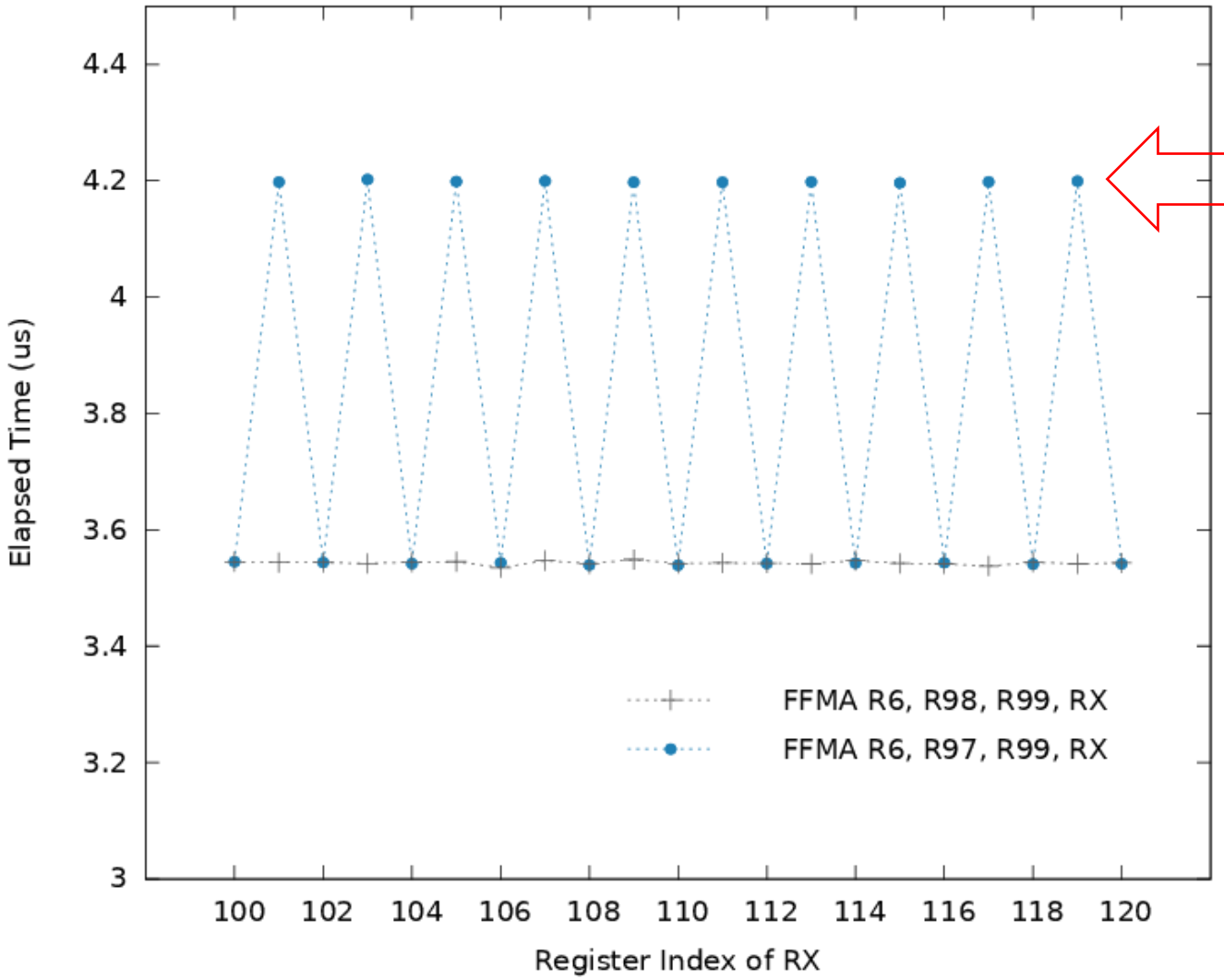
It’s the same! Instruction encoding

- 1:1 instruction to control

Width (bits)	4	6	3	3	1	4
Meaning	Reuse flags	Wait barrier mask	Read barrier index	Write barrier index	Yield flag	Stall cycles

- Our prior work includes encodings from previous generations and is available on arXiv

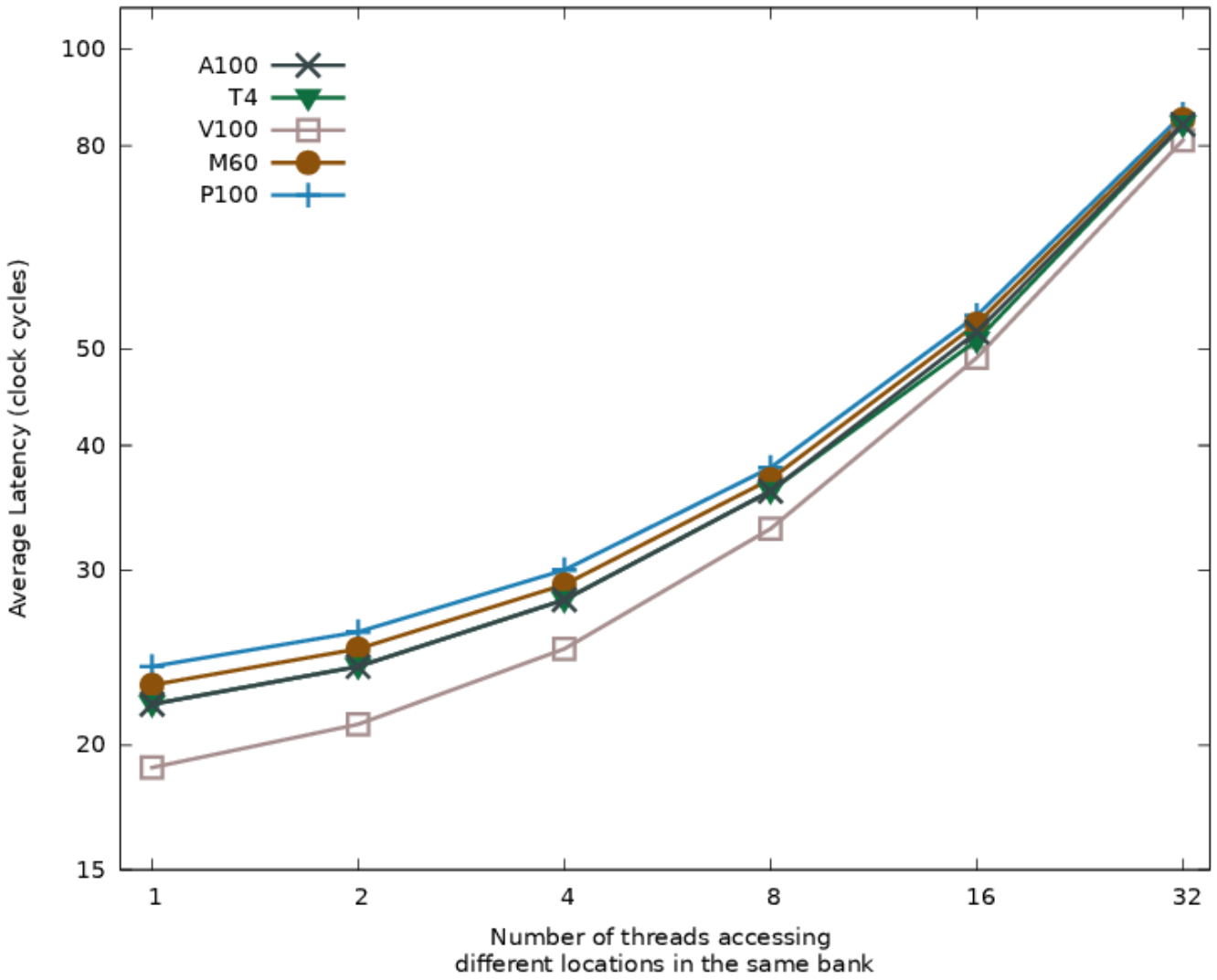
It’s the same! Dual-port registers



Register Bank Conflict

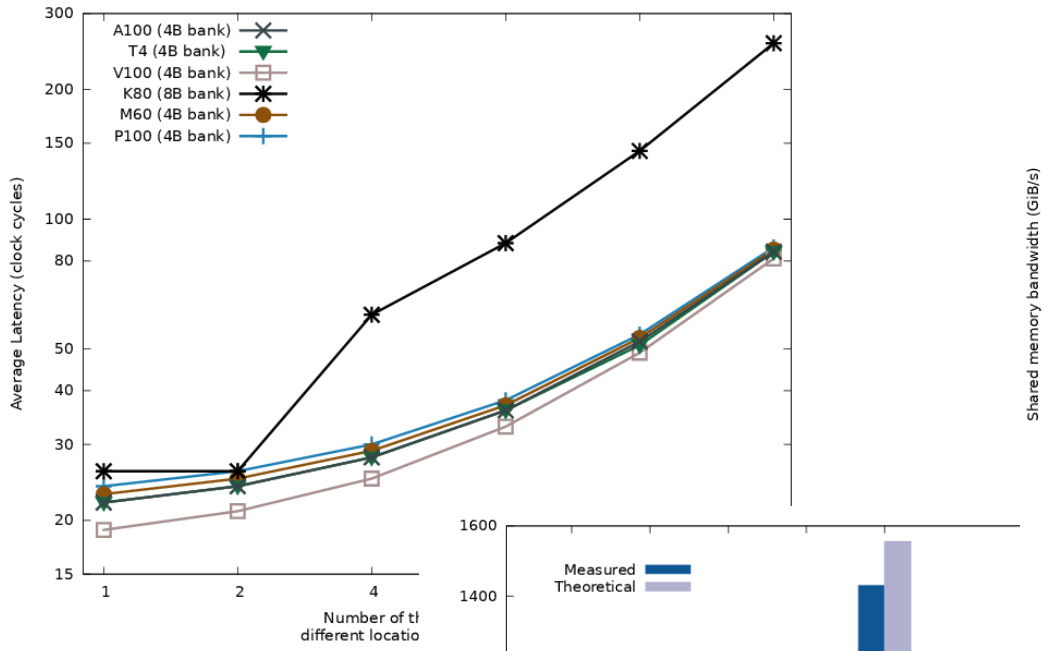
- Dual, two-port registers
- Same as in Volta
- Go read our Volta paper, it’s explained in depth there

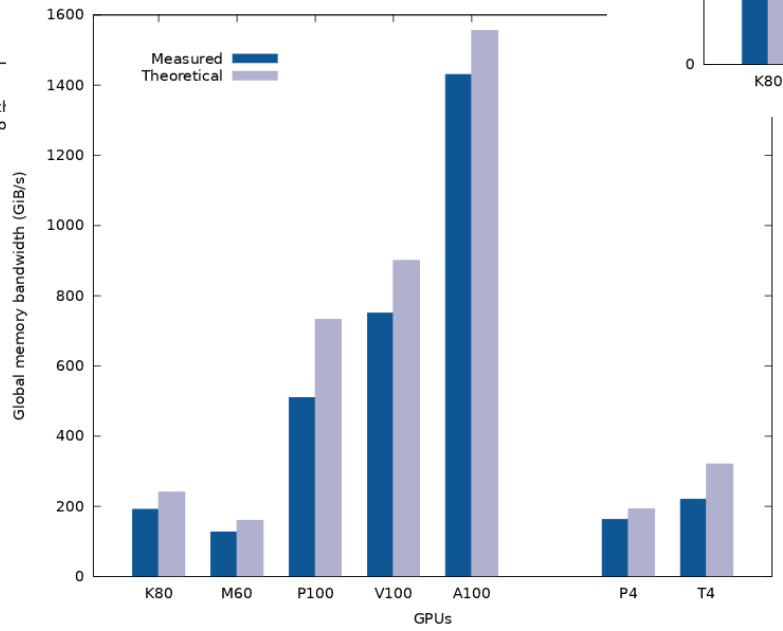
It’s the same! Shared memory latency suffers from bank conflicts

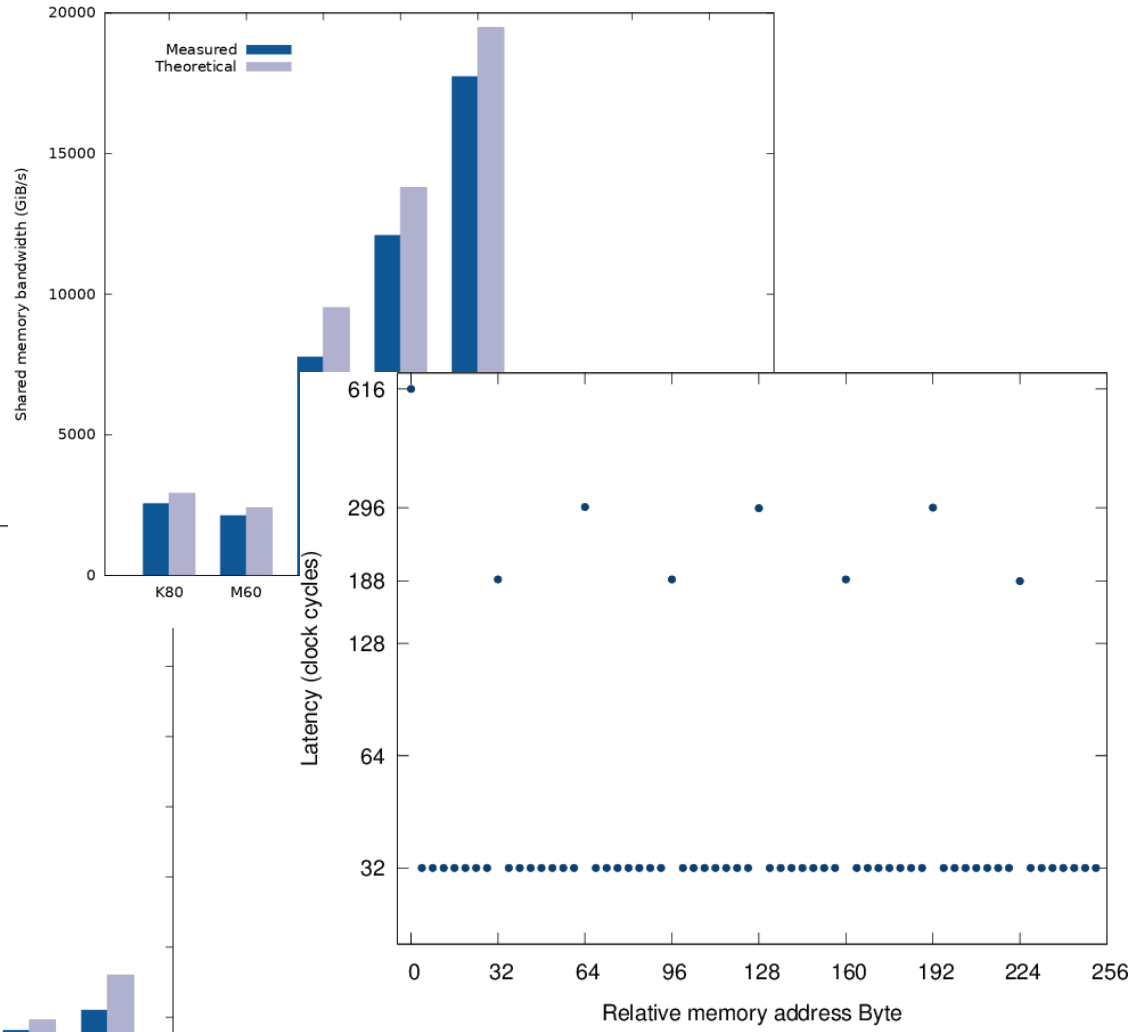


- Average access latency shows similar slow down across Ampere and previous generations with increasing bank conflicts
- V100 shows best latency among generations

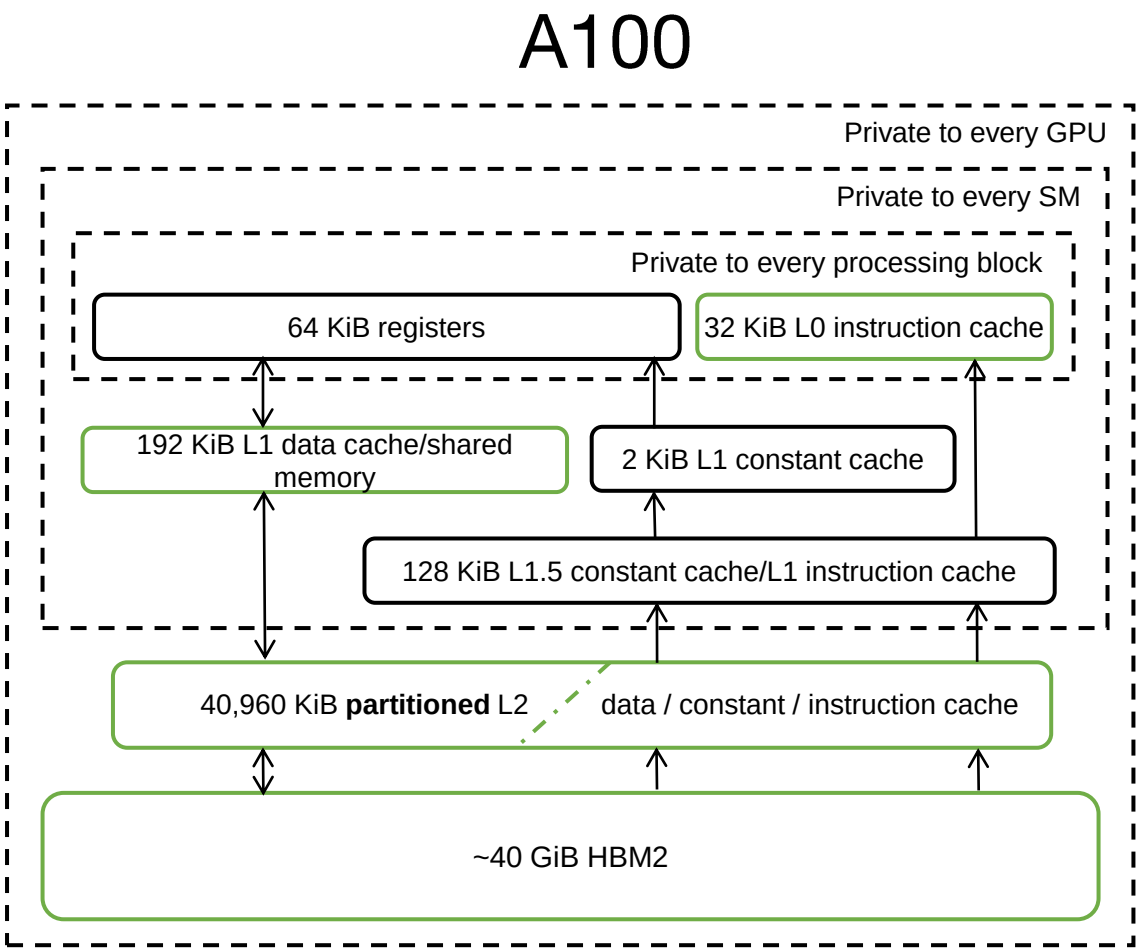
A100 benchmarks in the topology





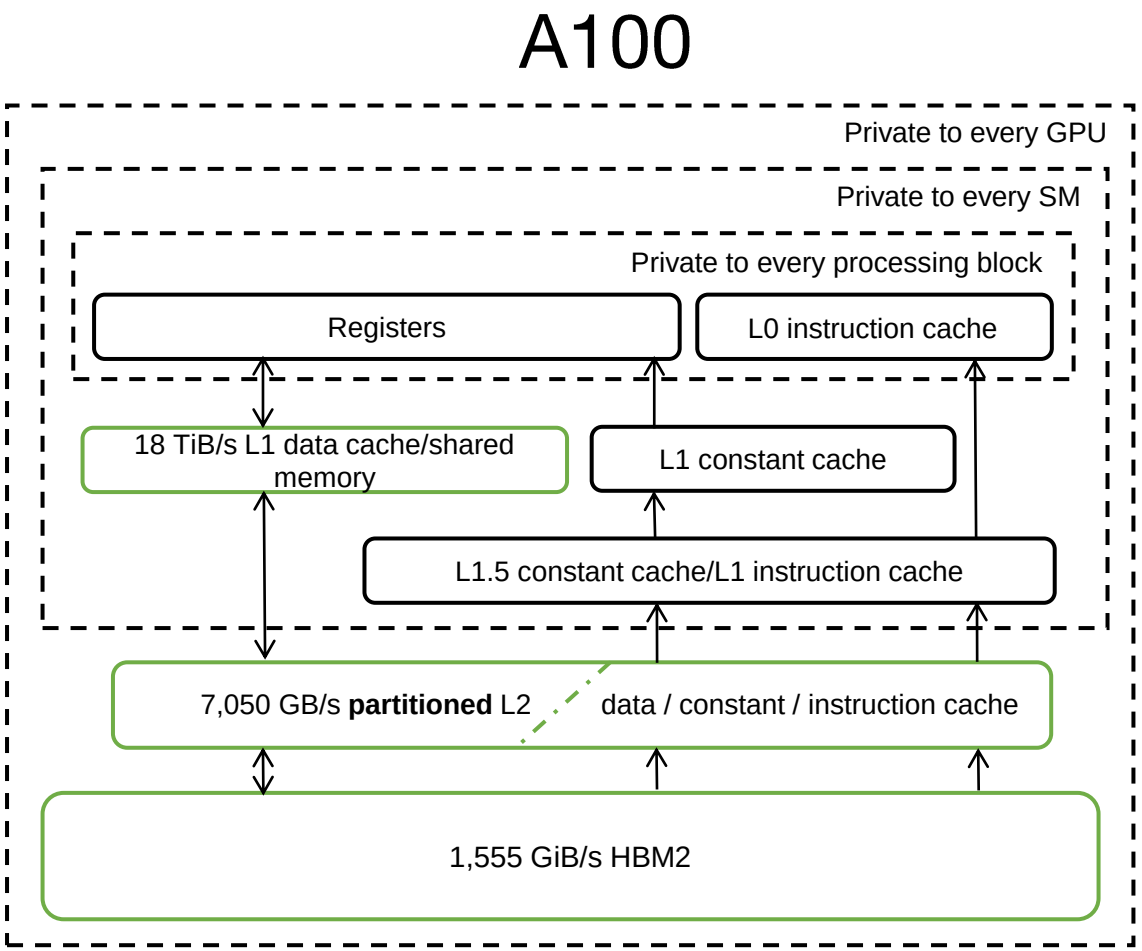


A100 (Ampere) memory capacity



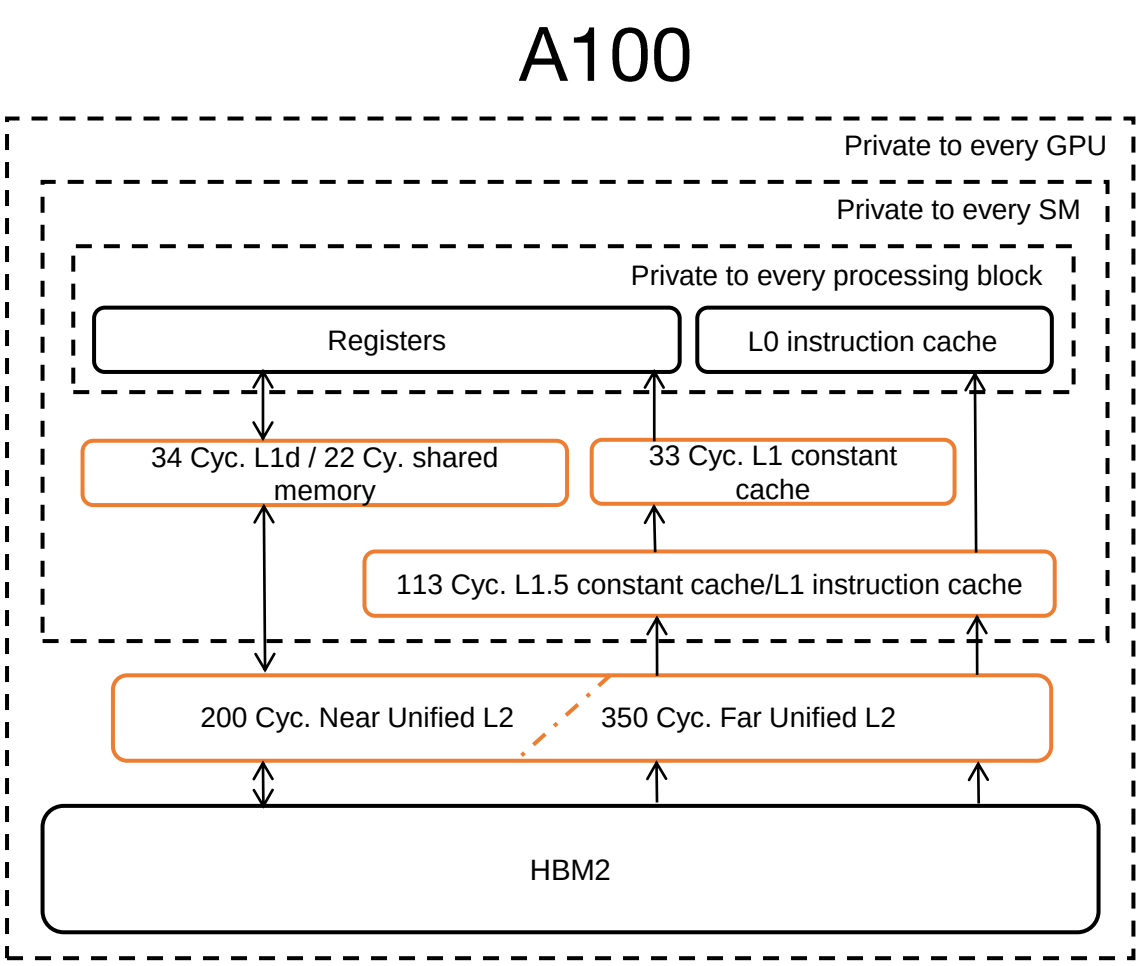
- L0 instruction cache capacity 2.7x bigger
  - Detected using micro-benchmarks
- Total L1 and shared memory capacity 1.5x bigger
  - 96% utilization of theoretical L1 max
  - 100% use of shared memory with 1 KiB reserved
- Unified L2 cache more than 6x bigger
  - Each partition more than 3x bigger
- Global memory is 2.5x bigger

A100 (Ampere) memory bandwidth



- Global memory bandwidth 1.7x faster
  - More than 1.4x increase in memory clock speed
- Theoretical L2 memory bandwidth 2.6x faster
  - More than 3% increase in graphics clock
- Shared memory bandwidth 1.4x faster
  - Same as L1 which is co-located
  - Proportional to increase in SM count from 80 to 108 and the increase in graphics clock
- Observed-theoretical ratio improved over V100
  - Global memory and L1 within 92% of theoretical maximum

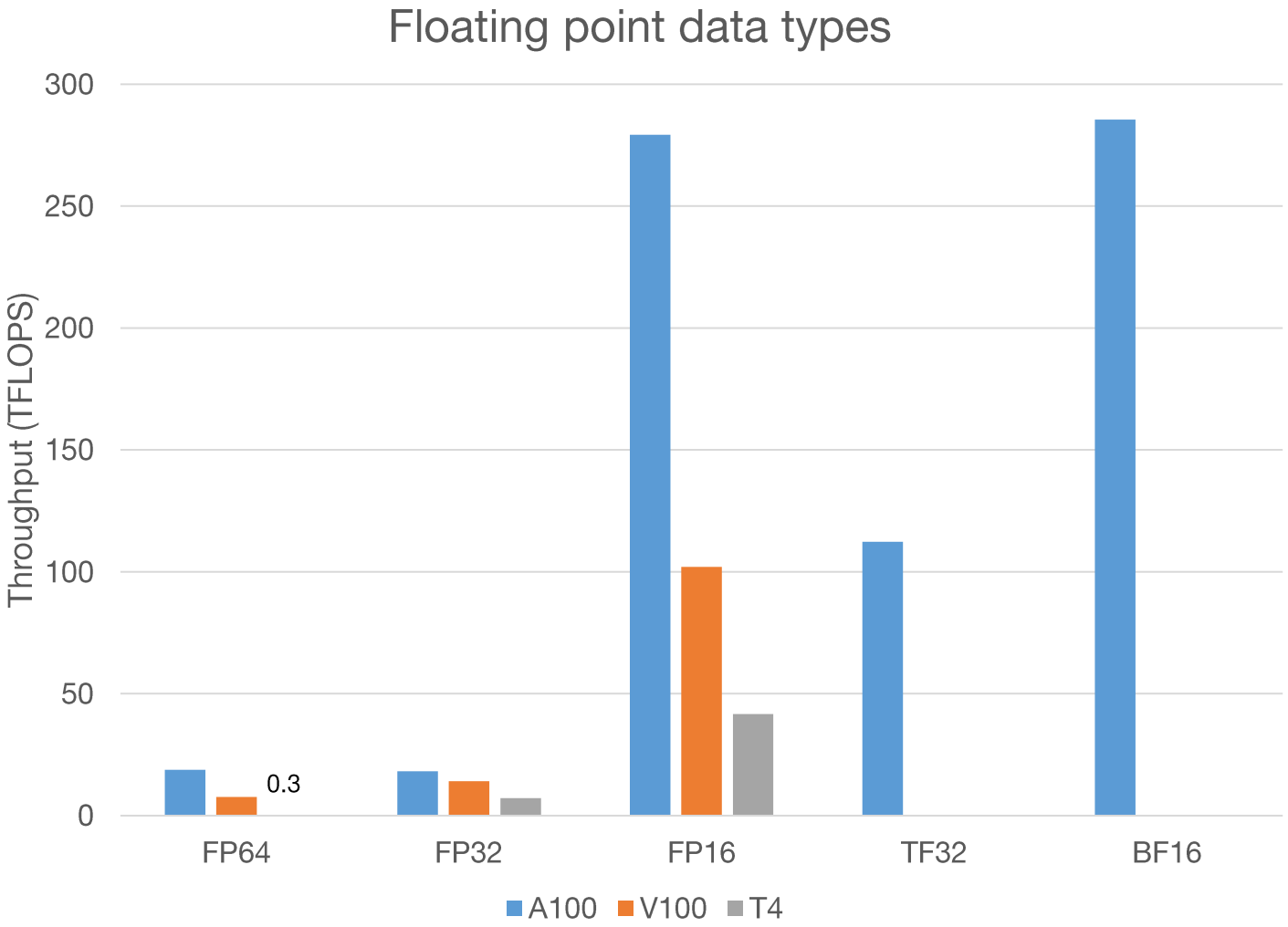
A100 (Ampere) memory latency

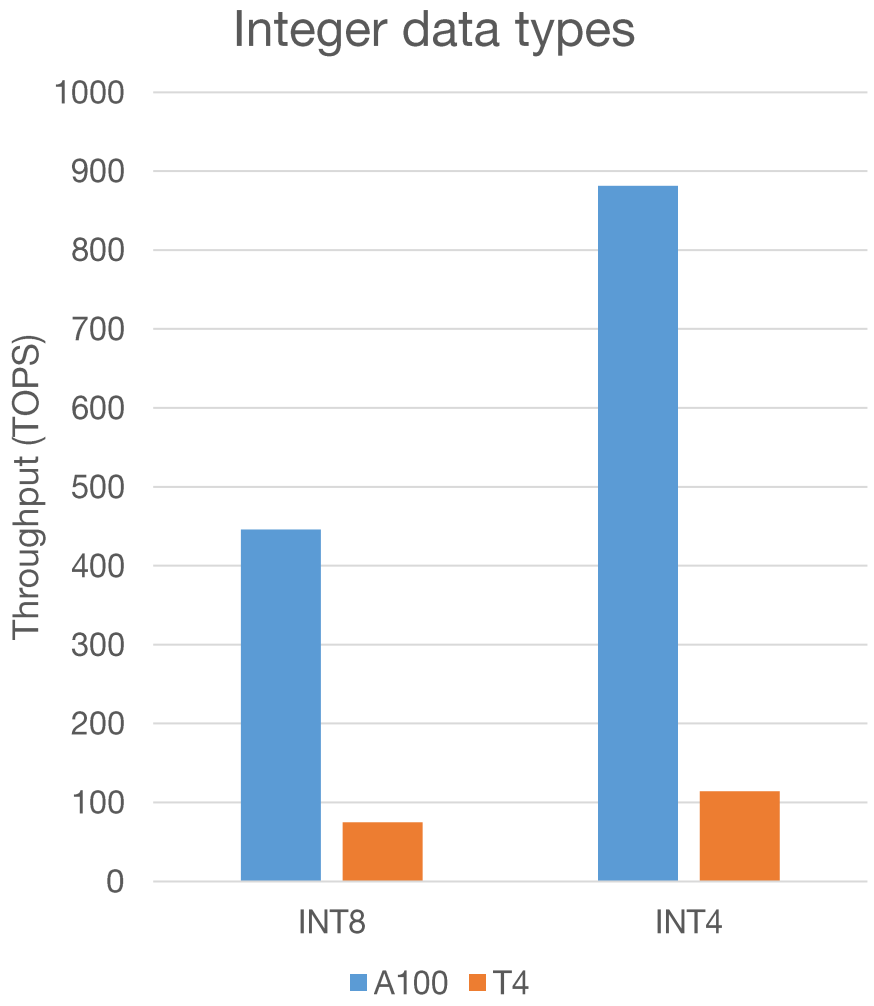


Cyc. = graphics cycles

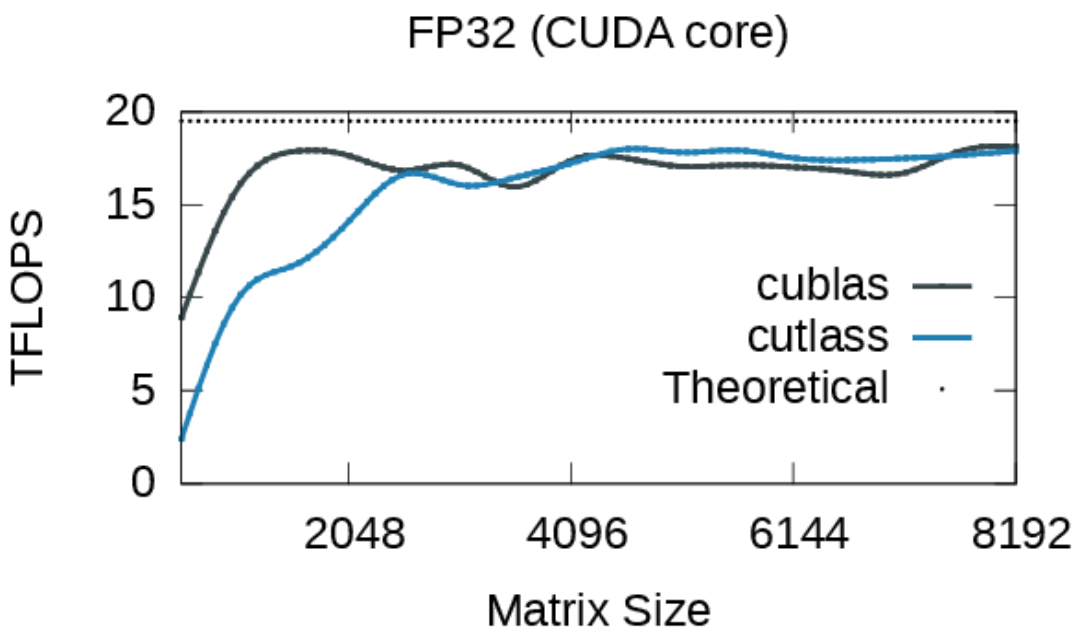
- Slightly longer latencies on:
  - L1 data cache
  - L1 constant cache
  - L2 cache
- Partitioned L2 split between ~200 and ~350 cycles
- Observed with micro-benchmarks

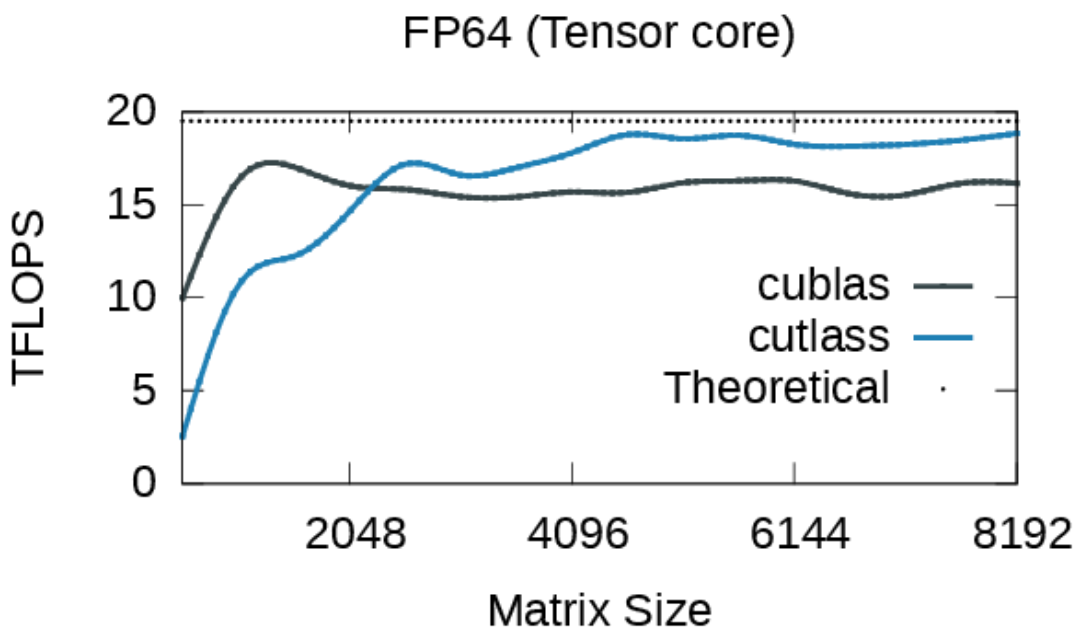
GEMM throughput across precisions





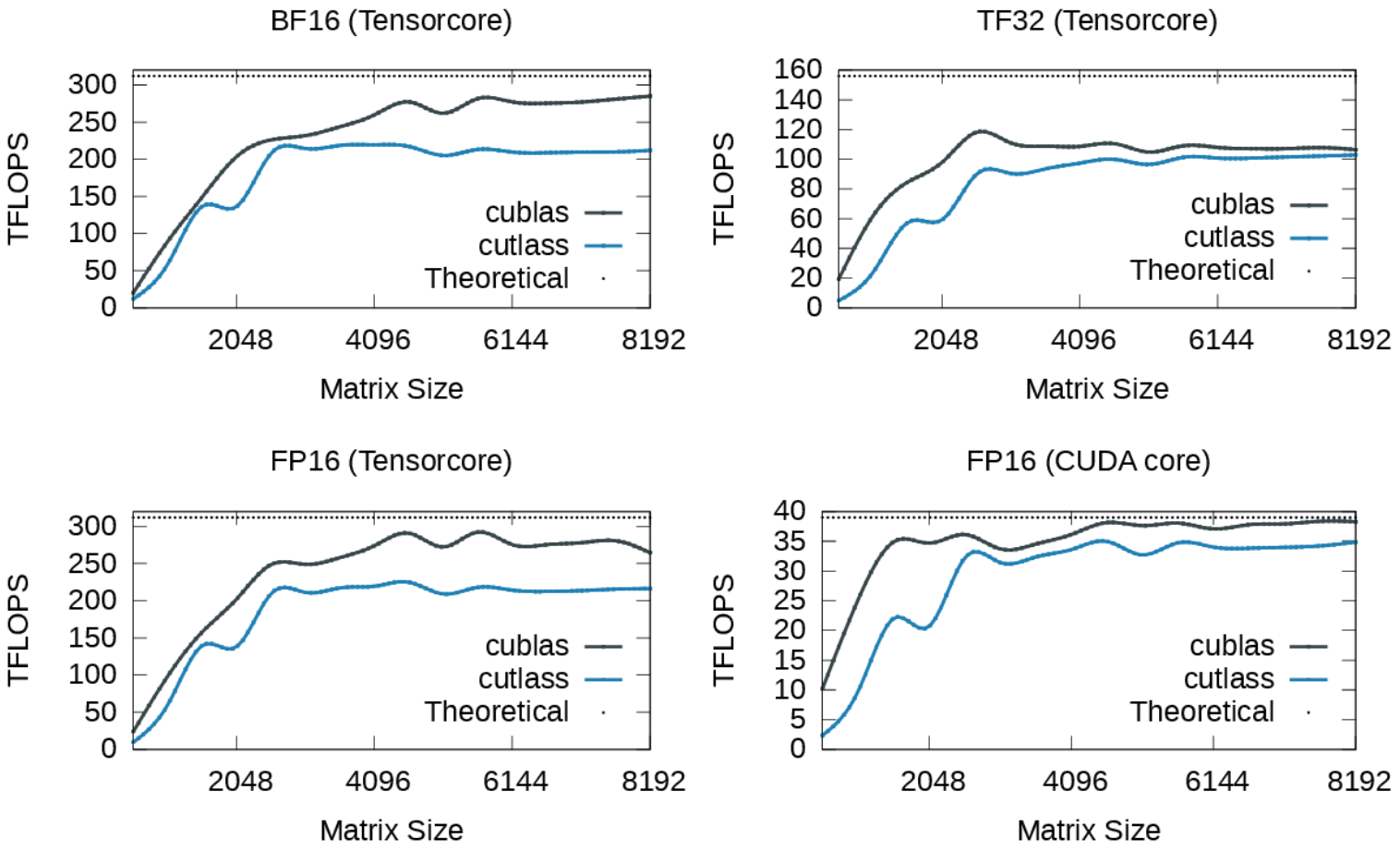
cuBLAS vs CUTLASS GEMM: Float





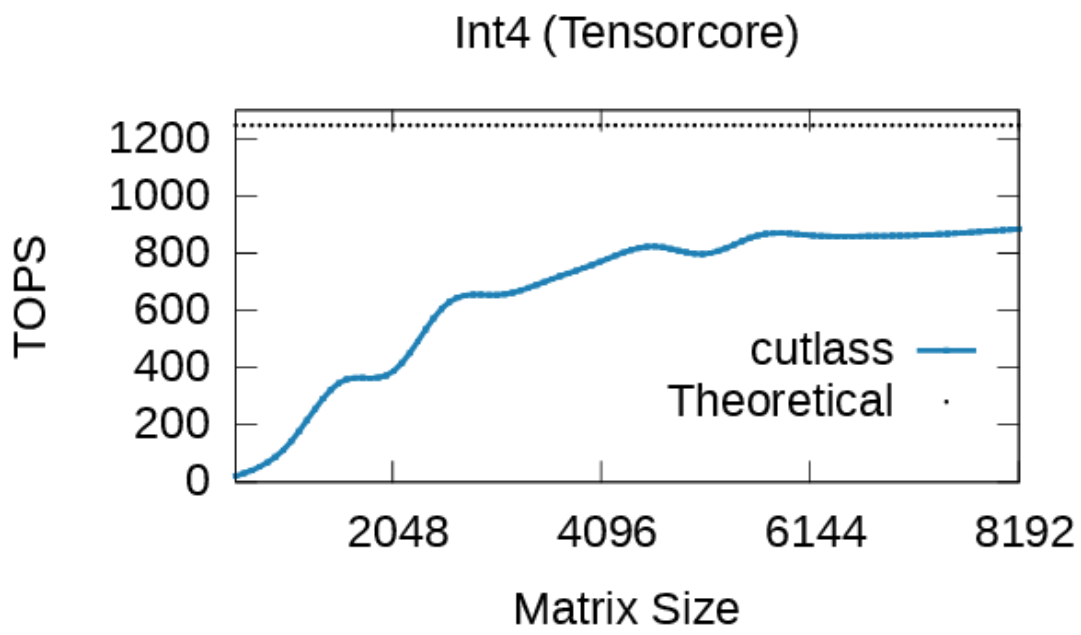
cuBLAS 11.2 vs. CUTLASS 2.4 GEMM

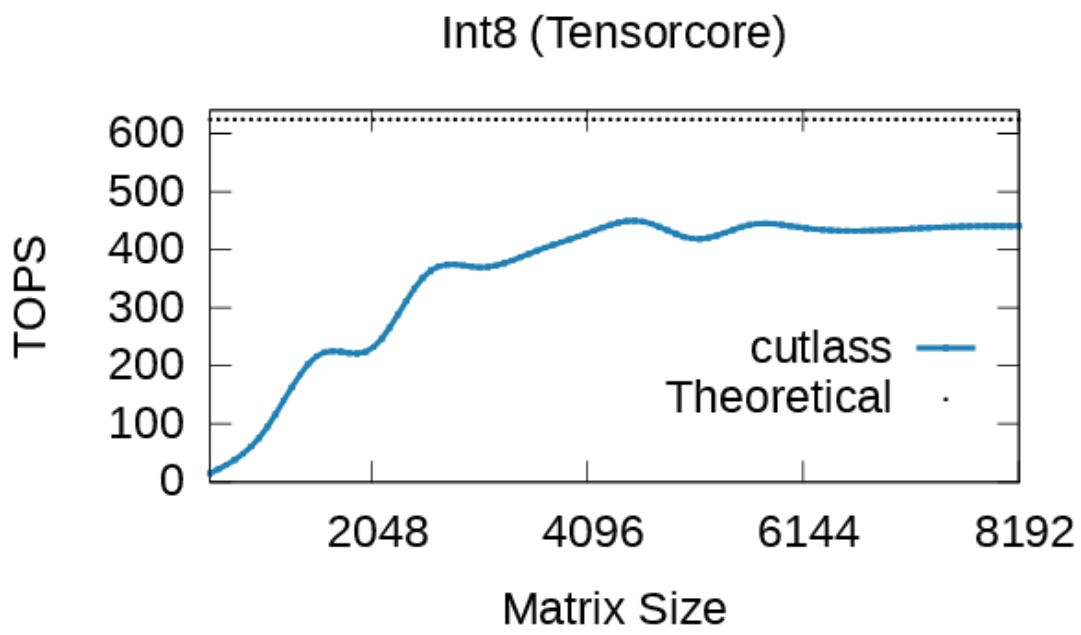
cuBLAS vs CUTLASS GEMM: Reduced



cuBLAS 11.2 vs. CUTLASS 2.4 GEMM

cuBLAS vs CUTLASS GEMM: Integer





cuBLAS 11.2 vs. CUTLASS 2.4 GEMM

Conclusion

- Group thread blocks by L2 partition to maximize effective L2 capacity
- Shared memory atomic increment is fast under contention
- Sparse matrix multiply performance depends on transpose
- Along the way, we also gave a comprehensive overview of what's new in Ampere architecturally

- Stay tuned for our tech report on arXiv around June