Dissecting the Turing GPU Architecture through Microbenchmarking

GTC 2019

Zhe Jia
Marco Maggioni
Jeffrey Smith
Daniele P. Scarpazza

High Performance Computing R&D Team

CITADEL

Summary

- GPU software performance matters
  - performance improvements save money, time and lives

- Sometimes, you can achieve peak performance
  only if you understand the architecture in depth
  - example 1: increase memory bandwidth by using wider loads
  - example 2: increase arithmetic throughput by avoiding register bank conflicts
  - … but many micro-architectural details are not disclosed by the manufacturer

- We expose the Turing T4 GPU architecture in depth
  - we discovered its details using micro-benchmarks
  - we reveal architectural details previously not published
    $\Rightarrow$ you can leverage them to improve your software performance
  - we compare them quantitatively against previous architectures
    $\Rightarrow$ get overview of the GPU evolution across generations
  - find all details in our technical report at https://goo.gl/adPpwg... that we announce today!

GPU Performance improvement reduces cost and offers opportunity

- Helps cost efficiency
  - Amazon EC2 p3.16xlarge GPU instance effective hourly costs:
    - $15.91 (data of 3/10/2019)
    - 10 instances: **~$1.4 M**/year
  - To reduce this cost:
    - Ask one HPC expert, speedup your training tasks by 2~10x
    - Save $0.7 M~$1.26 M/year.
  - **The more you optimize, the more you save!**

- Helps capture new opportunity
  - explore broader solution spaces for optimization problems
  - improve real-time inference throughput

GPU Performance improvement saves time

- AI researchers aren't cheap!


The New York Times

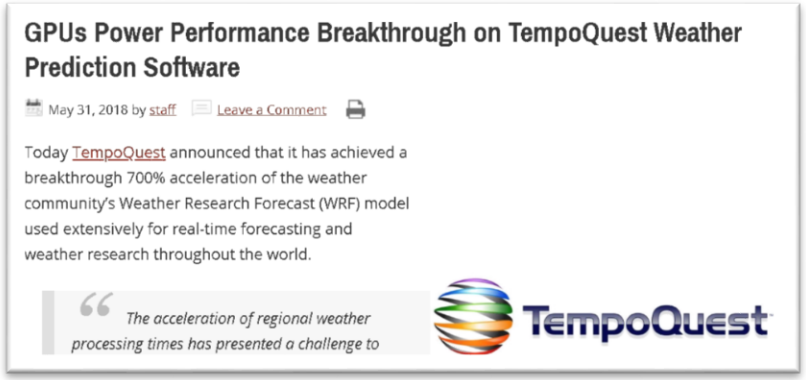
A.I. Researchers Are Making More Than $1 Million, Even at a Nonprofit

- What helps them to be more productive?
  - An infrastructure that trains their models fast
    - Choose right devices
    - Help them improve the performance of their training code

GPU Performance improvement saves lives

- Meteorologists use software to predict weather
- Increasing the compute performance of weather models helps them
  - produce warnings of extreme weather **more quickly**
  - improved model resolution provides **better accuracy**
- Improvement in either direction is crucial in saving lives and protecting property.
- Some meteorology problems are naturally **suitable for GPUs**
- Meteorologists have succeeded in using GPUs for weather/climate prediction

The Weather Company



TempoQuest

Custom optimization matters, and you can do it too!

- CUDA libraries are fast (>90% theoretical efficiency) and have a lot of hand-tuned functions, but they can’t possibly cover every single case
- NVCC is flexible, but generated code efficiency is usually closer to 80% for typical compute-bound kernels.
- Where it truly matters, can we write critical code as well as NVidia?
- YES!

Using architectural information to optimize GPU software

- Most inefficiencies in GPU software stem from failures in saturating either
  - memory bandwidth
  - instruction throughput

- Low-level architecture understanding is crucial to achieving peak GPU software performance
  - Example 1: single-precision a*X plus Y (memory-bound)
  - Example 2: simplest matrix-matrix multiplication core (compute-bound)

Example 1: single-precision a*X plus Y

- A scaled, element-wise vector-vector sum:

$$\vec{y} \coloneqq \alpha \cdot \vec{x} + \vec{y}$$

- Implementations for *cublasSaxpy* in CUDA 10.1:

  contain only 32-bit and 64-bit global-memory load/store instructions

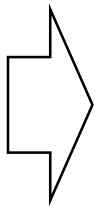
- For Turing GPUs, considering:
  - T4 has 4 LSUs per scheduler (V100: 8)
  - Turing supports 1024 threads per SM (Volta: 2,048)

- It is harder to saturate the available memory bandwidth on Turing by only increasing block/thread count (TLP).

Example 1: 128-bit vectorized memory access

- An effective strategy to increase memory access throughput: **load wider words per instruction**
- We use 128-bit vectorized memory access instructions

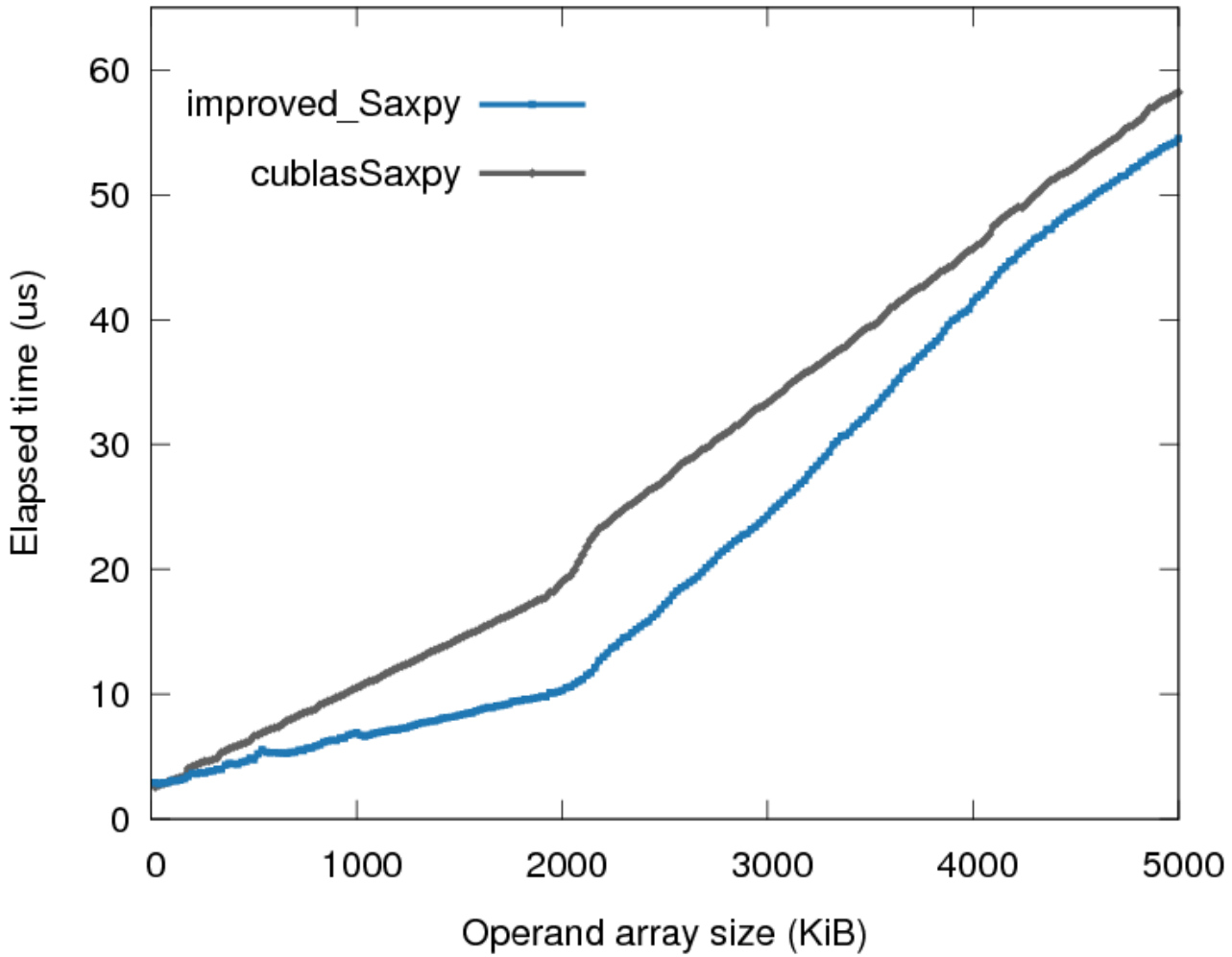
```
...
asm volatile("{\t\n"
  // registers to store input operands
  ".reg .f32 a1,b1,c1,d1;\n\t"
  ".reg .f32 a2,b2,c2,d2;\n\t"
  // loading with vectorized, 128-bit inst
  "ld.global.v4.f32 {a1,b1,c1,d1},[%0];\n\t"
  "ld.global.v4.f32 {a2,b2,c2,d2},[%1];\n\t"
  // core math operations
  ...
  // storing with vectorized, 128-bit inst
  "st.global.v4.f32 [%1],{a2,b2,c2,d2};\n\t"
"}" :: ...
```



```
.headerflags    @"EF_CUDA_SM75
EF_CUDA_PTX_SM(EF_CUDA_SM75)"
...

/*00d0*/LDG.E.128.SYS R8, [R8] ;
/*00e0*/LDG.E.128.SYS R4, [R2] ;

...

/*0150*/STG.E.128.SYS [R2], R4 ;

...
```

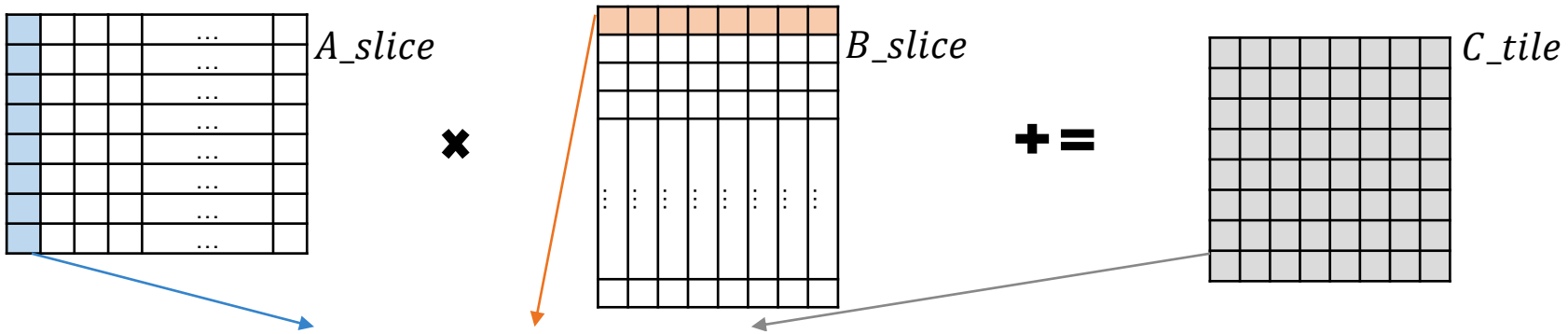
Example 1: performance improvement

- For arrays of 20 KiB - 2000 KiB
  - *improved_Saxpy* tends to be **almost 2x** as fast as cublasSaxpy



Example 2: simple matrix-matrix multiplication

- $\boldsymbol{C} \mathrel{+}= \boldsymbol{AB}$
- Sometimes, we need some variations of this kernel
- Each thread computes a C_tile (8x8) from an A_slice (8x512) and a B_slice (512x8)
- matmul is the most expensive kernel in many workloads




```
float reg_A[8], reg_B[8], reg_C[64];
for (int k=0; k<512; k++) {
  // ...
  // each thread multiplies one 8-element column vector from
  // matrix A_slice against one 8-element row vector from matrix B_slice
  for (int i = 0; i<8; i++)
    for (int j = 0; j<8; j++)
      reg_C[i*8+j] += reg_A[i]*reg_B[j];
  // ...
}
```

Don’t panic …

- Microarchitectural details are subtle and likely new to many of you
- Fortunately, the key optimization concepts aren’t that many
- Taking the time to digest them provides the *critical* insights into optimizing for the architecture
- Don’t worry if you miss any detail
  The fully fleshed out example is in our Volta report from last year
  Google “Volta Citadel” and click the first result.
  (https://arxiv.org/abs/1804.06826)

Key register bottleneck mitigation concepts

- Register files are mapped into different banks
- Instructions need source operands, and read them via ports
- An instruction reading more operands from a bank than there are ports stalls execution!
- To save port accesses, code should employ ***register reuse caches***
- Compilers should leverage reuse caches to avoid conflicts, but they don’t always succeed!
- NVidia libraries resort to these hand optimizations

  and so can you!

Example 2: performance improvement

- We found a **better register mapping and reuse cache selection** than NVCC generated code
- Performance improvement on T4 (128 threads): **+12%**
  The achieved efficiency matches cuBLAS

before optimization	after reuse cache optimization
FFMA R16, R12, R80, R16	FFMA R17, R12.reuse, R80.reuse, R17
FFMA R17, R80.reuse, R13, R17	FFMA R16, R12, R81.reuse, R16
FFMA R18, R80.reuse, R14, R18	FFMA R25, R13.reuse, R80.reuse, R25
FFMA R19, R80, R15, R19	FFMA R24, R13, R81.reuse, R24
FFMA R20, R80.reuse, R8, R20	FFMA R33, R14.reuse, R80.reuse, R33
FFMA R21, R80.reuse, R9, R21	FFMA R32, R14, R81.reuse, R32
FFMA R22, R80.reuse, R10, R22	FFMA R41, R15.reuse, R80.reuse, R41
FFMA R23, R80, R11, R23	FFMA R40, R15, R81.reuse, R40
FFMA R24, R12, R81.reuse, R24	FFMA R49, R8.reuse, R80.reuse, R49
FFMA R25, R13, R81, R25	FFMA R48, R8, R81.reuse, R48
FFMA R26, R14, R81.reuse, R26	FFMA R57, R9.reuse, R80.reuse, R57
FFMA R27, R15, R81.reuse, R27	FFMA R56, R9, R81.reuse, R56
FFMA R28, R8, R81.reuse, R28	FFMA R65, R10.reuse, R80.reuse, R65
FFMA R29, R9, R81.reuse, R29	FFMA R64, R10.reuse, R81.reuse, R64
FFMA R30, R10, R81.reuse, R30	FFMA R73, R11.reuse, R80, R73
...	...

GPU Manufacturers won’t tell you these architectural details

- Developers cannot exploit these opportunities
  without a **deep understanding** of GPU architecture

- In order to understand GPU architectures, we need to answer
  - what does the memory hierarchy look like?
  - how are instructions encoded?
  - what are the latency and throughput of instructions?
  - …

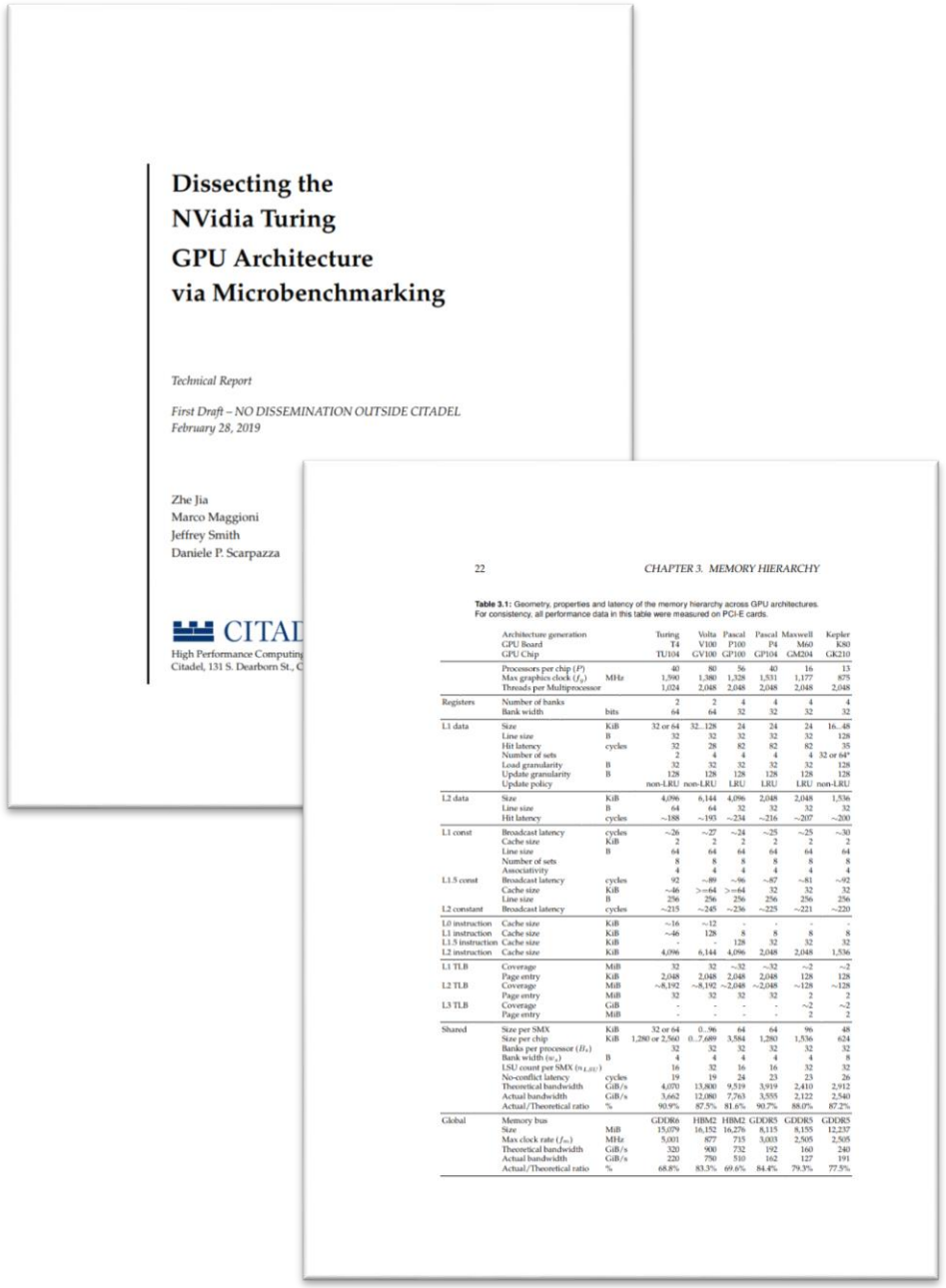
- Collecting architectural details can require **heroic** efforts, but you don’t need to.
- We have done this work for you!

Technical report

- Download it **now**
  https://goo.gl/adPpwg
  also in the process of publishing on arxiv.org

- Covers everything discussed today
  - it dissects the GPU architecture *completely*
  - plenty of details never published before that you won't find anywhere else
  - compares **every generation** from Kepler through Turing.
  - discusses how GPU architectures interact with compiled software
  - explains the experiments we performed.

- …plus much more!
  Covers everything that we can't fit into today.



Dissecting the
NVidia Turing
GPU Architecture
via Microbenchmarking

Technical Report

First Draft – NO DISSEMINATION OUTSIDE CITADEL
February 28, 2019

Zhe Jia
Marco Maggioni
Jeffrey Smith
Daniele P. Scarpazza

CITAD
High Performance Computin
Citadel, 131 S. Dearborn St., C

22 CHAPTER 3. MEMORY HIERARCHY

Table 3.1: Geometry, properties and latency of the memory hierarchy across GPU architectures. For consistency, all performance data in this table were measured on PCI-E cards.

	Architecture generation		Turing	Volta	Pascal	Pascal	Maxwell	Kepler
	GPU Board		T4	V100	P100	P4	M60	K80
	GPU Chip		TU104	GV100	GP100	GP104	GM204	GK210
	Processors per chip (P)		40	80	56	40	16	13
	Max graphics clock (f_g)	MHz	1,590	1,380	1,328	1,531	1,177	875
	Threads per Multiprocessor		1,024	2,048	2,048	2,048	2,048	2,048
Registers	Number of banks		2	2	4	4	4	4
	Bank width	bits	64	64	32	32	32	32
L1 data	Size	KiB	32 or 64	32...128	24	24	24	16...48
	Line size	B	32	32	32	32	32	128
	Hit latency	cycles	32	28	82	82	82	35
	Number of sets		2	4	4	4	4	32 or 64*
	Load granularity	B	32	32	32	32	32	128
	Update granularity	B	128	128	128	128	128	128
	Update policy		non-LRU	non-LRU	LRU	LRU	LRU	non-LRU
L2 data	Size	KiB	4,096	6,144	4,096	2,048	2,048	1,536
	Line size	B	64	64	32	32	32	32
	Hit latency	cycles	~188	~193	~234	~216	~207	~200
L1 const	Broadcast latency	cycles	~26	~27	~24	~25	~25	~30
	Cache size	KiB	2	2	2	2	2	2
	Line size	B	64	64	64	64	64	64
	Number of sets		8	8	8	8	8	8
	Associativity		4	4	4	4	4	4
L1.5 const	Broadcast latency	cycles	92	~89	~96	~87	~81	~92
	Cache size	KiB	~46	>=64	>=64	32	32	32
	Line size	B	256	256	256	256	256	256
L2 constant	Broadcast latency	cycles	~215	~245	~236	~225	~221	~220
L0 instruction	Cache size	KiB	~16	~12	-	-	-	-
L1 instruction	Cache size	KiB	~46	128	8	8	8	8
L1.5 instruction	Cache size	KiB	-	-	128	32	32	32
L2 instruction	Cache size	KiB	4,096	6,144	4,096	2,048	2,048	1,536
L1 TLB	Coverage	MiB	32	32	~32	~32	~2	~2
	Page entry	KiB	2,048	2,048	2,048	2,048	128	128
L2 TLB	Coverage	MiB	~8,192	~8,192	~2,048	~2,048	~128	~128
	Page entry	MiB	32	32	32	32	2	2
L3 TLB	Coverage	GiB	-	-	-	-	~2	~2
	Page entry	MiB	-	-	-	-	2	2
Shared	Size per SMX	KiB	32 or 64	0...96	64	64	96	48
	Size per chip	KiB	1,280 or 2,560	0...7,689	3,584	1,280	1,536	624
	Banks per processor (B_s)		32	32	32	32	32	32
	Bank width (w_s)	B	4	4	4	4	4	8
	LSU count per SMX (n_{LSU})		16	32	16	16	32	32
	No-conflict latency	cycles	19	19	24	23	23	26
	Theoretical bandwidth	GiB/s	4,070	13,800	9,519	3,919	2,410	2,912
	Actual bandwidth	GiB/s	3,662	12,080	7,763	3,555	2,122	2,540
	Actual/Theoretical ratio	%	90.9%	87.5%	81.6%	90.7%	88.0%	87.2%
Global	Memory bus		GDDR6	HBM2	HBM2	GDDR5	GDDR5	GDDR5
	Size	MiB	15,079	16,152	16,276	8,115	8,155	12,237
	Max clock rate (f_m)	MHz	5,001	877	715	3,003	2,505	2,505
	Theoretical bandwidth	GiB/s	320	900	732	192	160	240
	Actual bandwidth	GiB/s	220	750	510	162	127	191
	Actual/Theoretical ratio	%	68.8%	83.3%	69.6%	84.4%	79.3%	77.5%

Turing’s GPU architecture evolution

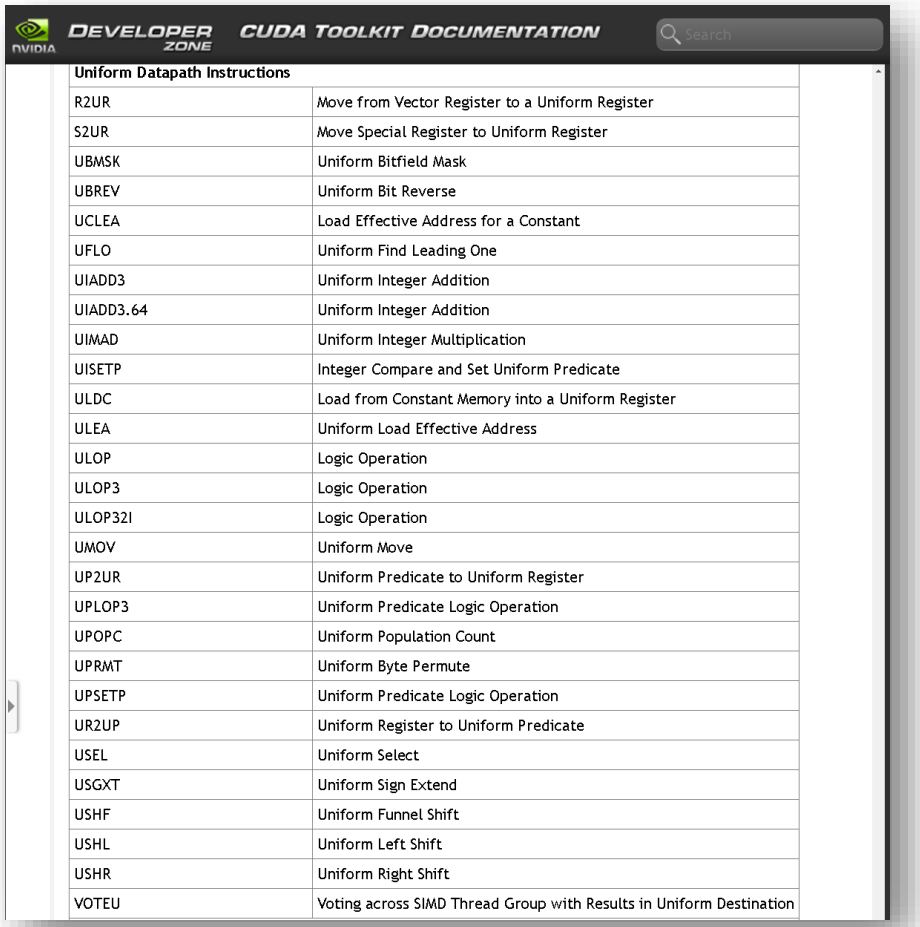
- New architectural features on Turing
  - better ILP; instruction cache friendly

- Architectural changes on recent GPUs
  - changed instruction encoding
  - improved instruction and data cache hierarchy
  - additional register ports
  - reduced native instruction dependent-issue latency
  - lower shared memory access latency
  - enlarged TLB coverage

- Compared to the P4, the T4 has
  - higher L1/L2 cache and global memory bandwidth
  - higher arithmetic throughput for matrix math

New features

Turing introduces a new datapath for integer instructions



Uniform Datapath Instructions

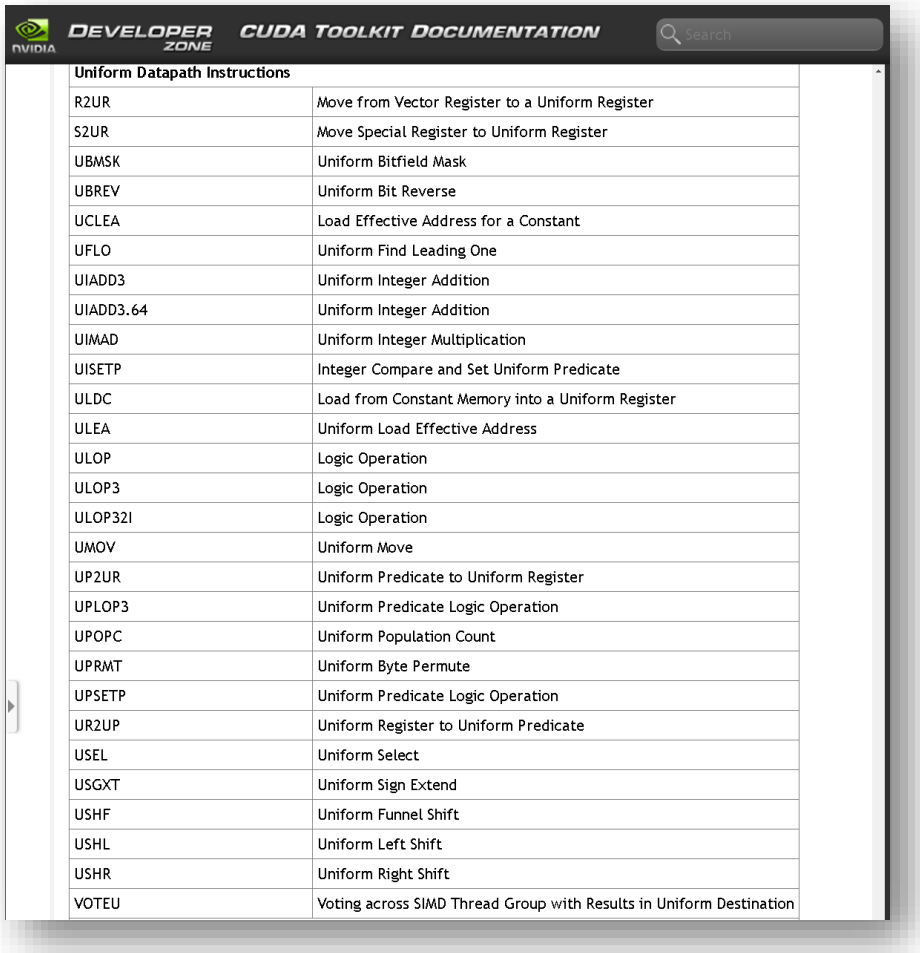
R2UR	Move from Vector Register to a Uniform Register
S2UR	Move Special Register to Uniform Register
UBMSK	Uniform Bitfield Mask
UBREV	Uniform Bit Reverse
UCLEA	Load Effective Address for a Constant
UFLO	Uniform Find Leading One
UIADD3	Uniform Integer Addition
UIADD3.64	Uniform Integer Addition
UIMAD	Uniform Integer Multiplication
UISETP	Integer Compare and Set Uniform Predicate
ULDC	Load from Constant Memory into a Uniform Register
ULEA	Uniform Load Effective Address
ULOP	Logic Operation
ULOP3	Logic Operation
ULOP32I	Logic Operation
UMOV	Uniform Move
UP2UR	Uniform Predicate to Uniform Register
UPLOP3	Uniform Predicate Logic Operation
UPOPC	Uniform Population Count
UPRMT	Uniform Byte Permute
UPSETP	Uniform Predicate Logic Operation
UR2UP	Uniform Register to Uniform Predicate
USEL	Uniform Select
USGXT	Uniform Sign Extend
USHF	Uniform Funnel Shift
USHL	Uniform Left Shift
USHR	Uniform Right Shift
VOTEU	Voting across SIMD Thread Group with Results in Uniform Destination

The uniform Datapath instructions in CUDA Binary Utilities Document

- For workloads that occupy the main datapaths with FP instructions
- Turing introduces a separate uniform datapath for integer instructions
- Index math and pointer math instructions can run in parallel with FP instructions
- FP compute-bound kernels can reach peak efficiency
- Uniform datapath instructions and uniform registers
  - e.g., UMOV UR5, 0x5000 ;

* https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth

Turing introduces a new datapath for integer instructions



Uniform Datapath Instructions	
R2UR	Move from Vector Register to a Uniform Register
S2UR	Move Special Register to Uniform Register
UBMSK	Uniform Bitfield Mask
UBREV	Uniform Bit Reverse
UCLEA	Load Effective Address for a Constant
UFLO	Uniform Find Leading One
UIADD3	Uniform Integer Addition
UIADD3.64	Uniform Integer Addition
UIMAD	Uniform Integer Multiplication
UISETP	Integer Compare and Set Uniform Predicate
ULDC	Load from Constant Memory into a Uniform Register
ULEA	Uniform Load Effective Address
ULOP	Logic Operation
ULOP3	Logic Operation
ULOP32I	Logic Operation
UMOV	Uniform Move
UP2UR	Uniform Predicate to Uniform Register
UPLOP3	Uniform Predicate Logic Operation
UPOPC	Uniform Population Count
UPRMT	Uniform Byte Permute
UPSETP	Uniform Predicate Logic Operation
UR2UP	Uniform Register to Uniform Predicate
USEL	Uniform Select
USGXT	Uniform Sign Extend
USHF	Uniform Funnel Shift
USHL	Uniform Left Shift
USHR	Uniform Right Shift
VOTEU	Voting across SIMD Thread Group with Results in Uniform Destination

The uniform Datapath instructions in CUDA Binary Utilities Document

- most regular instructions can access **both** uniform and regular registers
- most uniform datapath instructions **only** operate on uniform registers
- Turing supports **64** uniform registers
  1 URZ + UR0-UR62
- The upper limit of **total** registers is 256, including both regular and uniform registers

* https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth

NVCC expresses matrix math more succinctly for Turing

- HMMA:
  half precision matrix math instruction
- Turing offers new instruction-state options for HMMA
- NVCC uses fewer instructions to express some tensor operations
- For the same *wmma::mma_sync()* in a same kernel,
  NVCC generates 16 HMMAs for Volta,
  but only 4 HMMA for Turing

```
# Target Volta
HMMA.884.F32.F32.STEP0 R8, R26.reuse.COL, R16.reuse.COL, R8 ;
HMMA.884.F32.F32.STEP1 R10, R26.reuse.COL, R16.reuse.COL, R10 ;
HMMA.884.F32.F32.STEP2 R4, R26.reuse.COL, R16.reuse.COL, R4 ;
HMMA.884.F32.F32.STEP3 R6, R26.COL, R16.COL, R6 ;
HMMA.884.F32.F32.STEP0 R8, R20.reuse.COL, R18.reuse.COL, R8 ;
HMMA.884.F32.F32.STEP1 R10, R20.reuse.COL, R18.reuse.COL, R10 ;
HMMA.884.F32.F32.STEP2 R4, R20.reuse.COL, R18.reuse.COL, R4 ;
HMMA.884.F32.F32.STEP3 R6, R20.COL, R18.COL, R6 ;
HMMA.884.F32.F32.STEP0 R8, R22.reuse.COL, R12.reuse.COL, R8 ;
HMMA.884.F32.F32.STEP1 R10, R22.reuse.COL, R12.reuse.COL, R10 ;
HMMA.884.F32.F32.STEP2 R4, R22.reuse.COL, R12.reuse.COL, R4 ;
HMMA.884.F32.F32.STEP3 R6, R22.COL, R12.COL, R6 ;
HMMA.884.F32.F32.STEP0 R8, R2.reuse.COL, R14.reuse.COL, R8 ;
HMMA.884.F32.F32.STEP1 R10, R2.reuse.COL, R14.reuse.COL, R10 ;
HMMA.884.F32.F32.STEP2 R4, R2.reuse.COL, R14.reuse.COL, R4 ;
HMMA.884.F32.F32.STEP3 R6, R2.COL, R14.COL, R6 ;


# Target Turing
HMMA.1688.F32 R8, R12, R22, R8 ;
HMMA.1688.F32 R4, R12, R23, R4 ;
HMMA.1688.F32 R8, R2, R24, R8 ;
HMMA.1688.F32 R4, R2, R25, R4 ;
```

Architectural Changes

From Kepler to Turing

From Kepler to Turing: better hardware efficiency via software-driven scheduling

control for 7 instructions

```
Kepler:
          /*0008*/   MOV R1, c[0x0][0x44];                 /* 0x08a0bc80c0a08cc0 */
          /*0010*/   S2R R0, SR_CTAID.X;                   /* 0x64c03c00089c0006 */
          /*0018*/   S2R R3, SR_TID.X;                     /* 0x86400000129c0002 */
          /*0020*/   IMAD R0, R0, c[0x0][0x28], R3;        /* 0x86400000109c000e */
          /*0028*/   S2R R4, SR_CLOCKLO;                   /* 0x51080c00051c0002 */
          /*0030*/   MEMBAR.CTA;                           /* 0x86400000281c0012 */
          /*0038*/   LOP32I.AND R2, R3, 0xfffffffc;        /* 0x7cc00000001c0002 */
                                                           /* 0x207ffffffe1c0c08 */
```

control for 3 instructions

```
Maxwell
Pascal:   /*0008*/   MOV R1, c[0x0][0x20];                 /* 0x001c7c00e22007f6 */
          /*0010*/   S2R R0, SR_CTAID.X;                   /* 0x4c98078000870001 */
          /*0018*/   S2R R2, SR_TID.X;                     /* 0xf0c8000002570000 */
                                                           /* 0xf0c8000002170002 */
```

control for 1 instruction

```
Volta
Turing:   /*0000*/   @!PT SHFL.IDX PT, RZ, RZ, RZ, RZ;     /* 0x000000ffffffff389 */
                                                           /* 0x000fe200000e00ff */
```

From Maxwell to Turing: control information is organized as below:

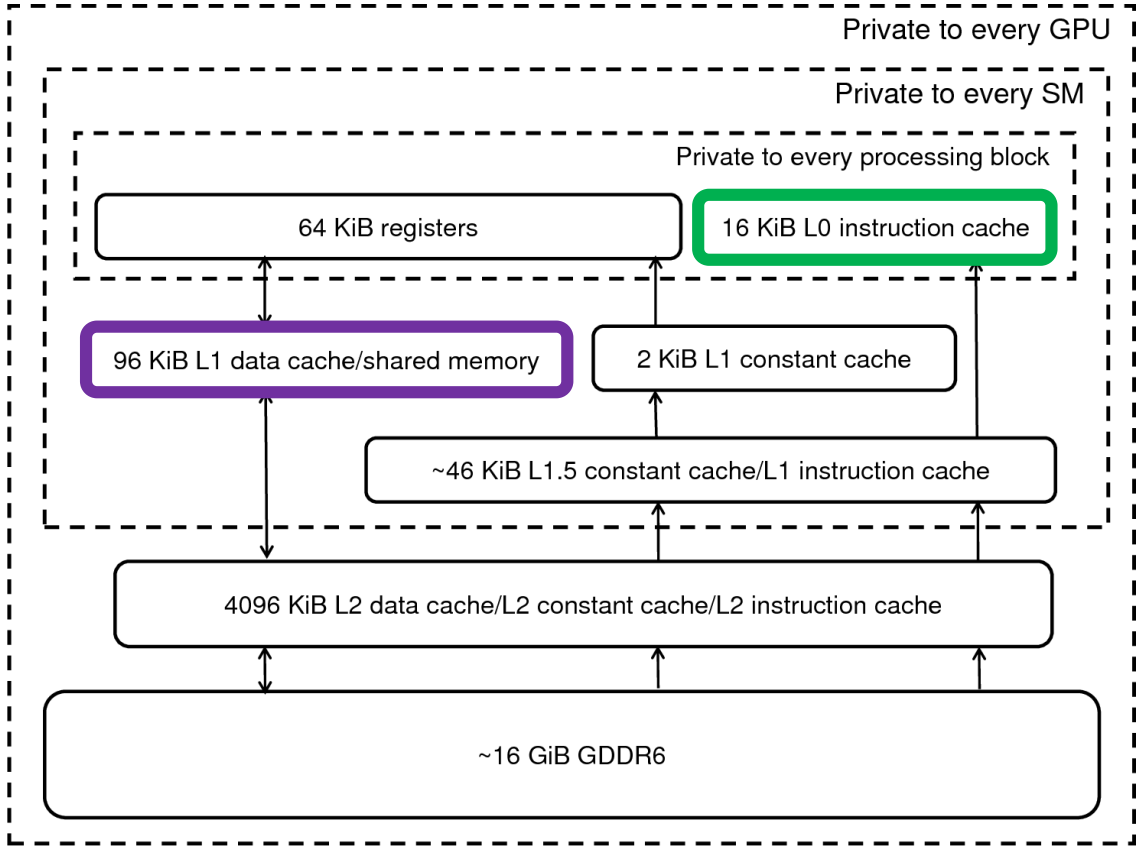
Width (bits)	4	6	3	3	1	4
Meaning	Reuse flags	Wait barrier mask	Read barrier index	Write barrier index	Yield flag	Stall cycles

Turing's memory hierarchy

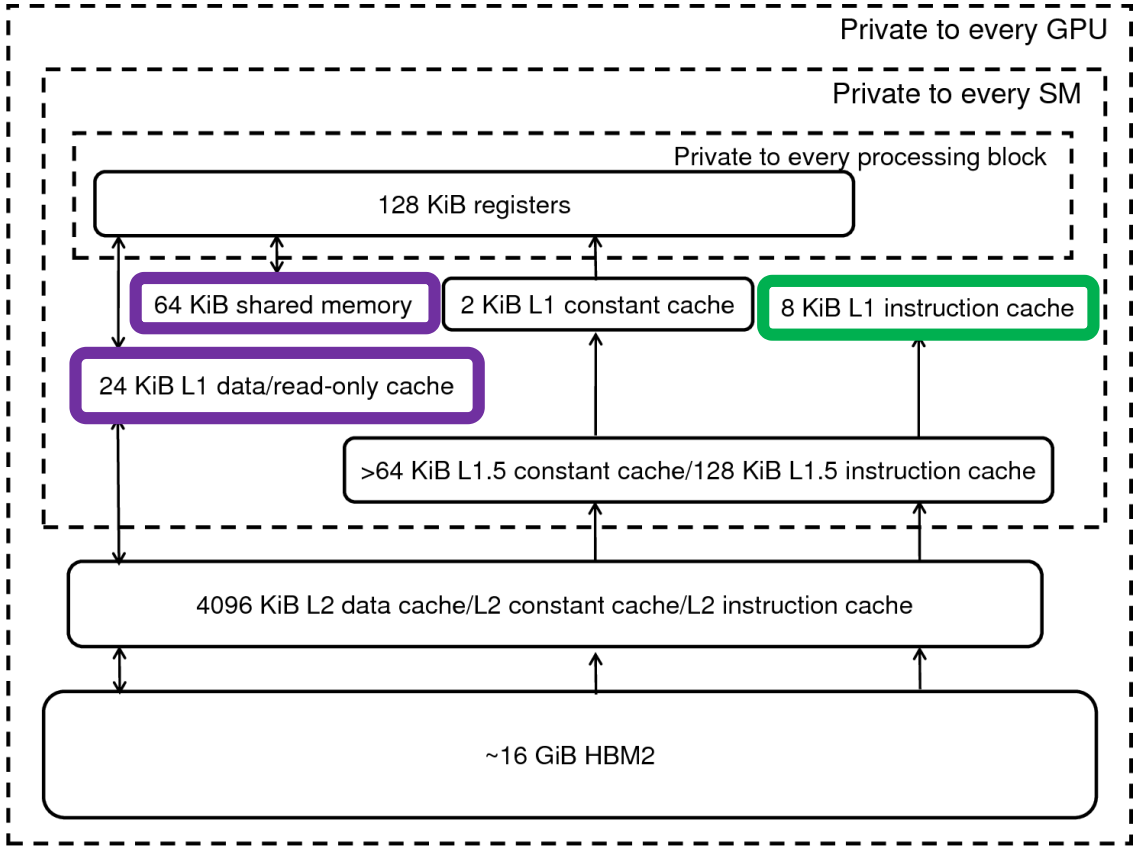
Turing has the similar memory hierarchy as Volta, and they have

- a new L0 instruction cache
- an unified shared and L1 data cache (low latency, high bandwidth; configurable)
- a new replacement policy for L1 cache to preserve large arrays

T4



P100



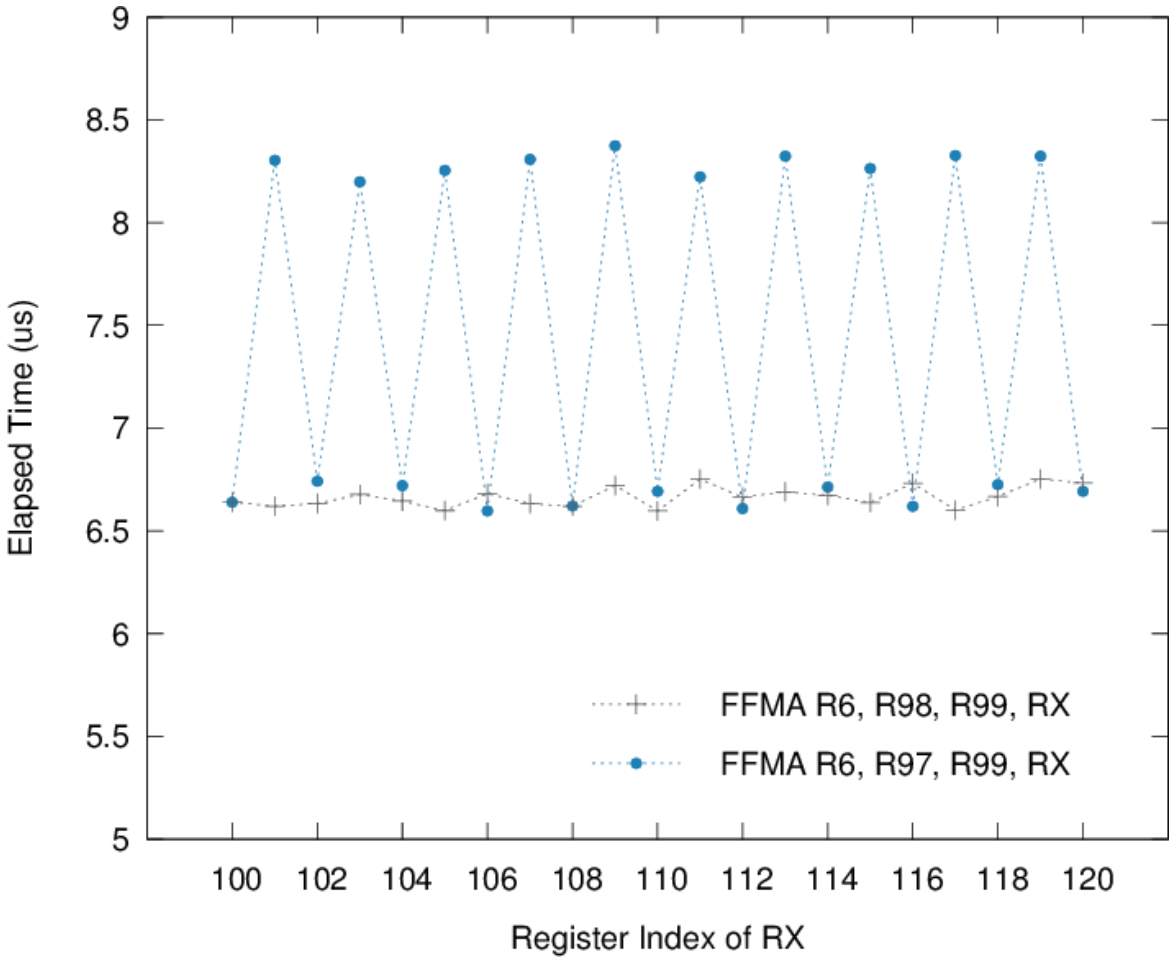
Turing and Volta have a new level of instruction cache

Architecture generation			Turing	Volta	Pascal	Pascal	Maxwell	Kepler
GPU Board			T4	V100	P100	P4	M60	K80
GPU Chip			TU104	GV100	GP100	GP104	GM204	GK210
L0 instruction	Cache size	KiB	~16	~12	-	-	-	-
L1 instruction	Cache size	KiB	~46	128	8	8	8	8
L1.5 instruction	Cache size	KiB	-	-	128	32	32	32
L2 instruction	Cache size	KiB	4,096	6,144	4,096	2,048	2,048	1,536

- We found one scheduler-private L0 instruction cache on Turing and Volta, by detecting
  - the size of each cache level
  - how cache levels are distributed within the architectural blocks

- We also found
  - on all GPUs considered
    - each L1 instruction cache is private to an SM
    - the L2 cache is unified (instructions, data, and constants) and shared across all SMs
  - on Pascal, Maxwell and Kepler, each L1.5 instruction cache is private to one SM

Redesigned register ports



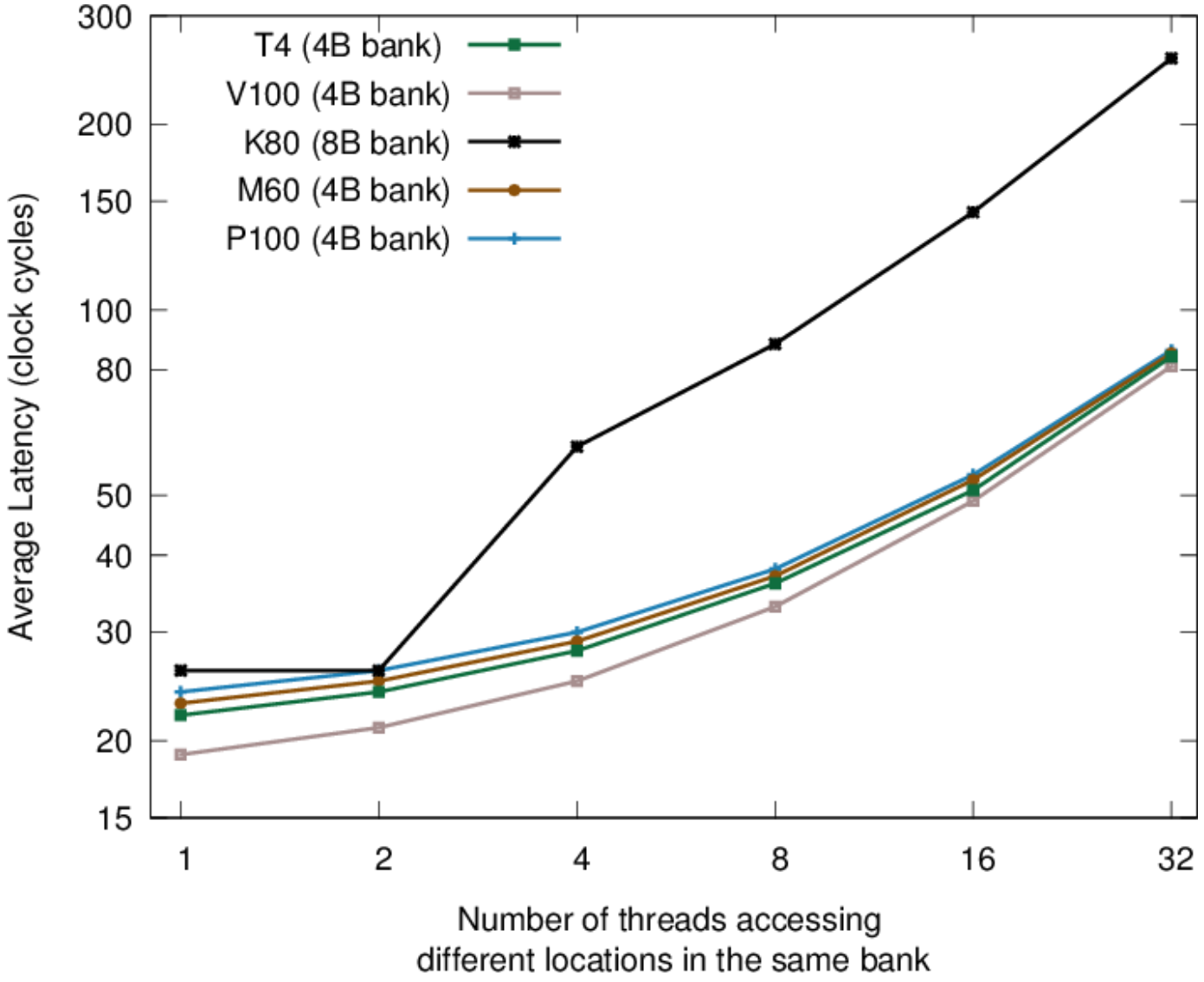
- Turing and Volta have the same register bank/port design
  2 banks with dual 32-bit ports
- On Pascal, Maxwell and Kepler
  4 single-ported banks
- Experiment
  - elapsed time of identical FFMA sequences
  - vary one source register index (RX) in two instruction sequences to cause conflict
  - In sequences of “FFMA R6, R97,R99,RX”, the choice of X can cause zero or one conflict
  - In sequence of “FFMA R6, R98,R99,RX”, the choice of X cannot cause conflicts

Turing changed native instruction latency

Architecture	Instructions	Latency (cycles)
Pascal	BFE, BFI, IADD, IADD32I, FADD, FMUL, FFMA, FMNMX, HADD2, HMUL2, HFMA2, IMNMX, ISCADD, LOP, LOP32I, LOP3, MOV, MOV32I, SEL, SHL, SHR, VADD, VABSDIFF, VMNMX, XMAD	6
	DADD, DMUL, DFMA, DMNMX	8
	FSET, DSET, DSETP, ISETP, FSETP	12
	POPC, FLO, MUFU, F2F, F2I, I2F, I2I	~14
	IMUL, IMAD	~86
Volta	IADD3, SHF, LOP3, SEL, MOV, FADD, FFMA, FMUL, ISETP, FSET, FSETP	4
	IMAD, FMNMX, DSET, DSETP	5
	HADD2, HMUL2, HFMA2	6
	DADD, DMUL, DFMA	8
	POPC	~10
	FLO, BREV, MUFU	~14
Turing	IADD3, SHF, LOP3, SEL, MOV, FADD, FFMA, FMUL, ISETP, FSET, FSETP	4
	IMAD, FMNMX, DSET, DSETP	5
	HADD2, HMUL2, HFMA2	6
	POPC, FLO, BREV, MUFU	~15
	DADD, DMUL	~48
	DFMA, DSET, DSETP	~54

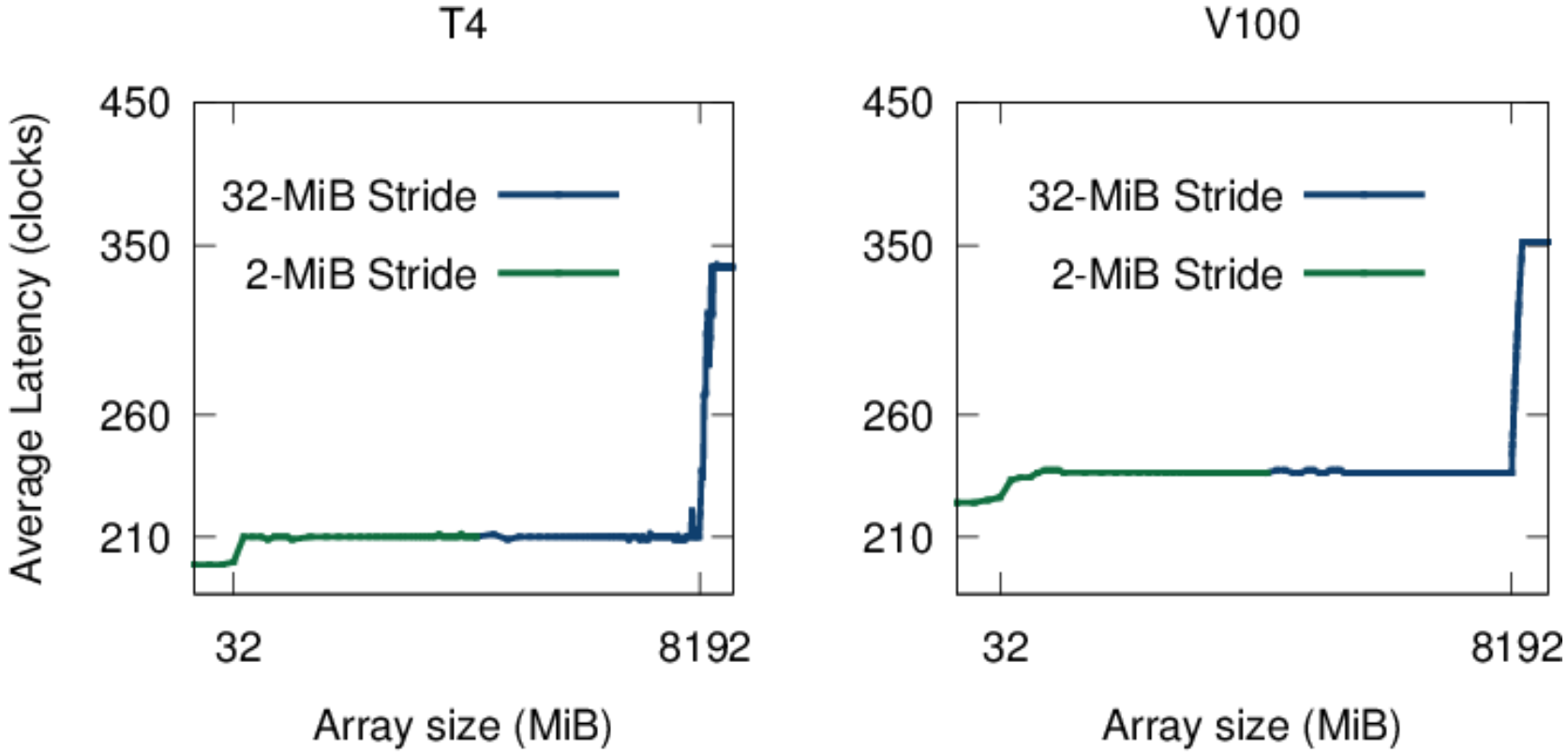
- On Turing and Volta, integer and single precision instructions have 4-cycle latency
- On Turing, double precision instructions have highest latency among three generations
- On Pascal, instructions IMAD and IMUL have long latency because they are emulated
- Turing does not seem to offer any latency improvement over Volta

Turing and Volta have lower shared memory access latency



- The T4 and V100 GPUs provide lowest latency among all the examined GPUs
- The measured average access latency increases with the number of bank conflicts (except Kepler)

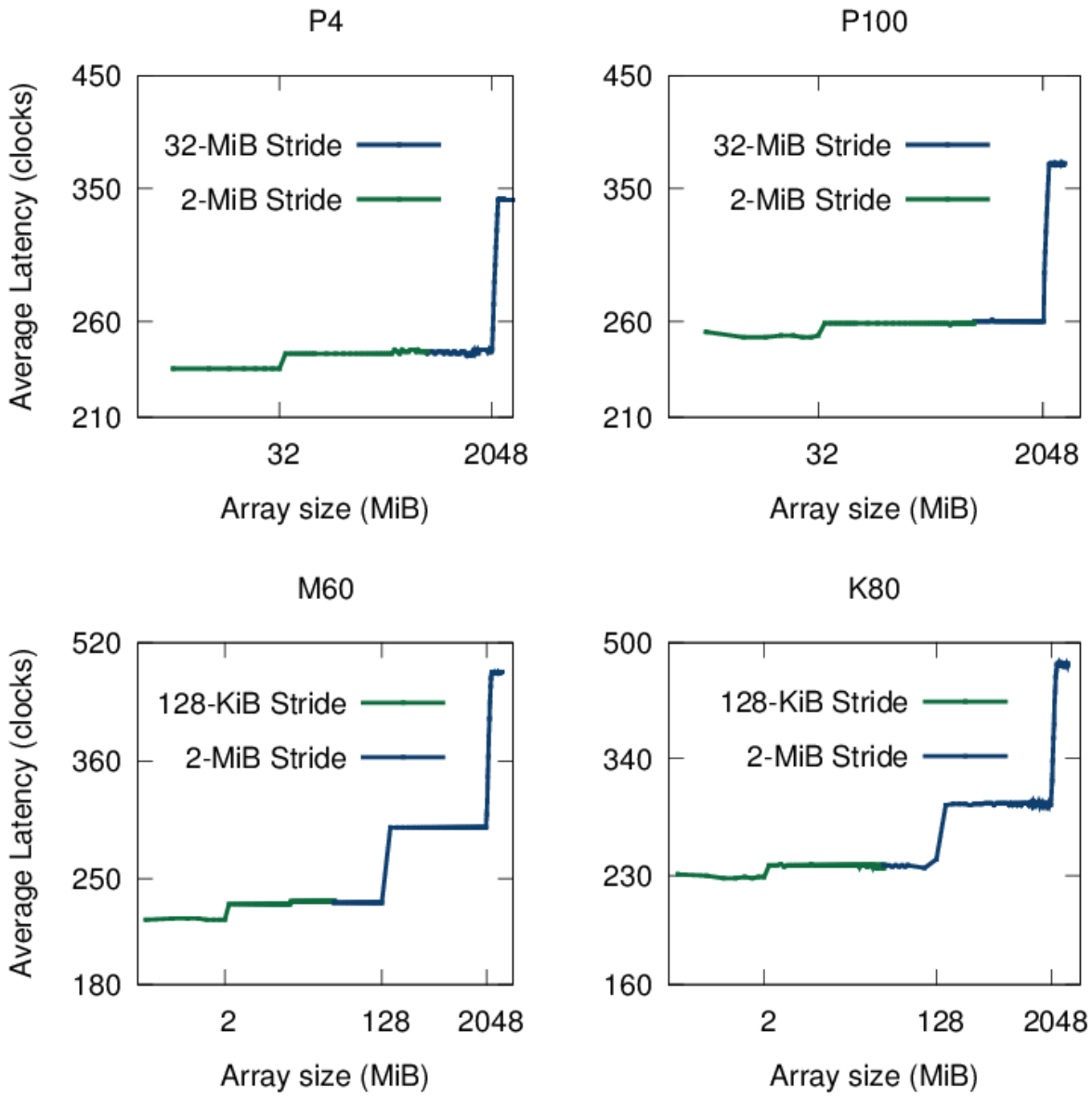
Turing and Volta enlarged TLB coverage



- On Turing and Volta, we detected
  - 2 TLB levels
  - L1 TLB: 2-MiB entries, 32-MiB coverage
  - L2 TLB: 32-MiB entries, 8192-MiB coverage

Turing and Volta enlarged TLB coverage

- On Pascal:
  - 2 TLB levels
  - L1 TLB: 2-MiB entries, 32-MiB coverage
  - L2 TLB: 32-MiB entries 2048-MiB coverage
- On Kepler and Maxwell:
  - 3 TLB levels
  - L1 TLB: 128-KiB entries, 2-MiB coverage
  - L2 TLB: 2-MiB entries, 128-MiB coverage
  - L3 TLB: 2-MiB entries, 2048-MiB coverage
- On all architectures examined:
  - L1D is virtually indexed
  - L2 is physically indexed

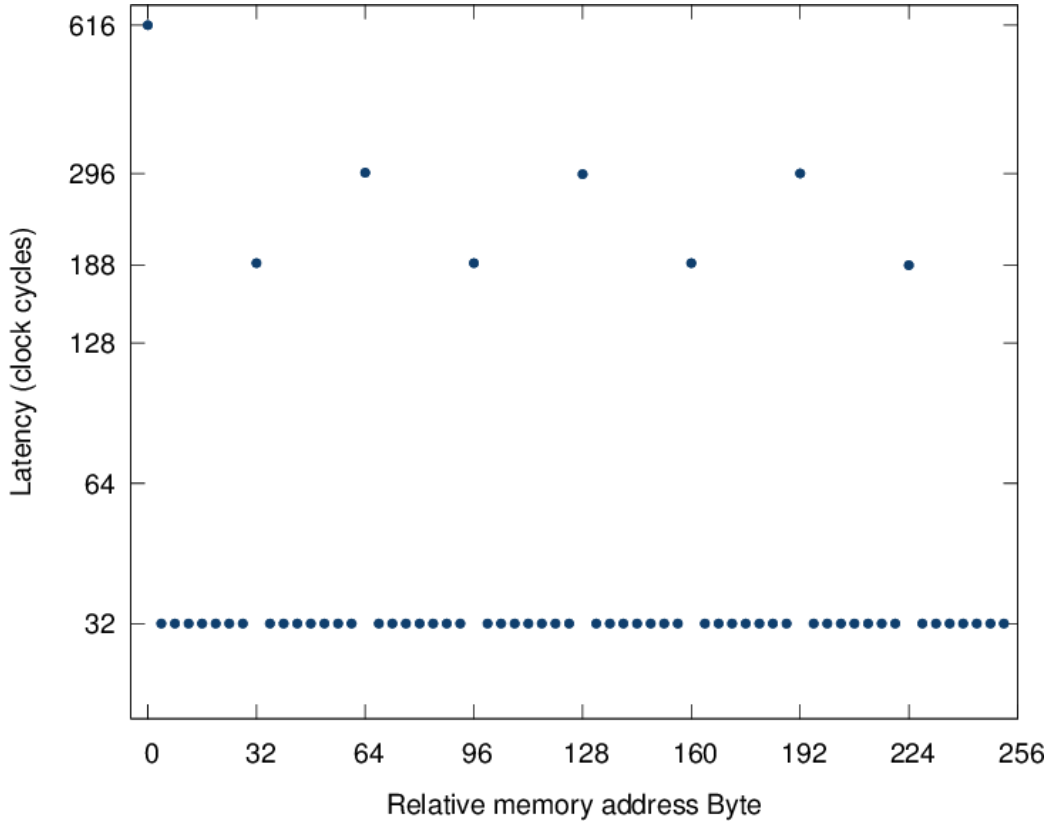


P4 vs. T4

The L1 cache on T4 enjoys lower latency than P4

	Architecture generation GPU Board GPU Chip		Turing T4 TU104	Volta V100 GV100	Pascal P100 GP100	Pascal P4 GP104
L1 data	Size	KiB	32 or 64	32...128	24	24
	Line size	B	32	32	32	32
	Hit latency	cycles	32	28	82	82
	Number of sets		4	4	4	4
	Load granularity	B	32	32	32	32
	Update granularity	B	128	128	128	128
	Update policy		non-LRU	non-LRU	LRU	LRU

- We measured L1D cache latency for all considered devices
  - fine-grained p-chase method by Mei and Chu*
- We found T4 and V100 have lower latency than P100, P4 and M60
- Experiment:
  - 32-cycle: warmed line access
  - 188-cycle: L1 miss and L2 hit
  - 296-cycle: L2 miss and TLB hit
  - 616-cycle latency: cache and TLB miss



*X. Mei and X. Chu, "Dissecting GPU memory hierarchy through microbenchmarking," IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 1, pp. 72–86, Jan 2017

The L1D cache on T4 enjoys higher bandwidth than P4

- We measured ~3x higher L1D bandwidth on T4 than on P4
- Experiment:
  - Scans an array in L1D cache, accesses as many lines as possible from every thread
- Theoretical upper bound: $n_{LSU} \times NB_{LSU}$

  n_{LSU}: LSU count per SM

  NB_{LSU}: the number of bytes that each LSU can load per cycle per instruction

Table 3.2: L1 cache load throughput per SM.

	T4	V100	P100	P4	M60	K80	
Measured throughput	58.8	108.3	31.3	15.7	15.7	68.6	bytes/cycle
Theoretical upper bound	64.0	128.0	64.0	64.0	128.0	128.0	bytes/cycle

The L2 cache on T4 enjoys higher bandwidth than P4

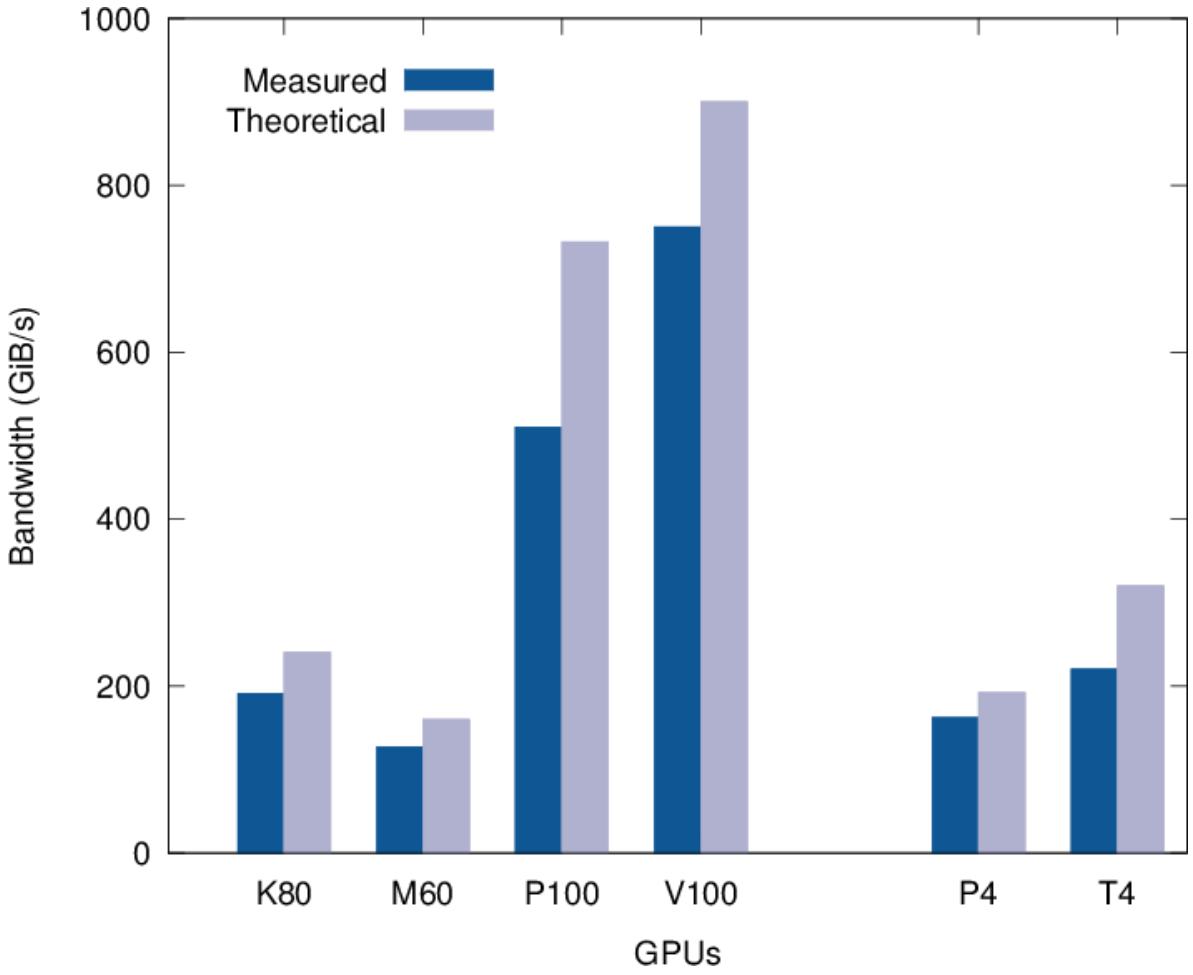
- We measured ~1.3x higher L2 bandwidth on T4 than on P4
- The L2 cache on T4:
  - unified for data, instruction and constant memory (as previous GPUs)
  - A 16-way, set-associative cache

capacity:	4,096	KiB
cache line:	64	B
average latency:	188	clock cycles
load throughput:	1,270	GB/s

Table 3.4: L2 data cache load throughput.

	Turing T4	Volta V100	Pascal P100	Pascal P4	Maxwell M60	Kepler K80
Throughput (GB/s)	1,270	2,155	1,624	979	446	339

T4 has higher global memory bandwidth than P4



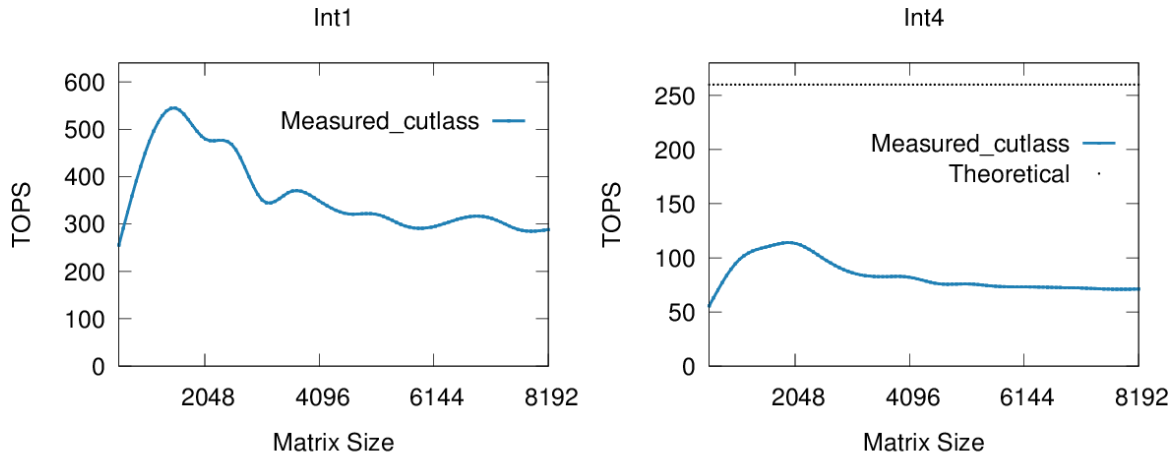
- The global mem bandwidth benchmark:
  - loads and stores global memory arrays
- We found
  - T4 enjoys a higher bandwidth than P4 due to GDDR6 memory
  - The actual-to-theoretical ratio on the T4 (68.8%) is lower than P4 (84.4%)
  - GPUs with HBM2 (V100 and P100) have higher bandwidth than those with GDDR (K80, M60, P4 and T4)

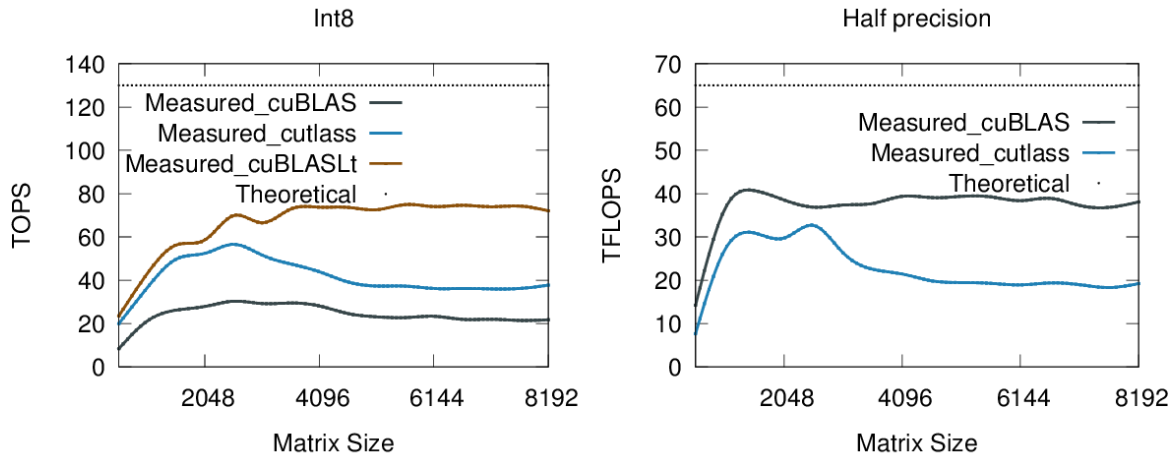
Arithmetic performance on T4

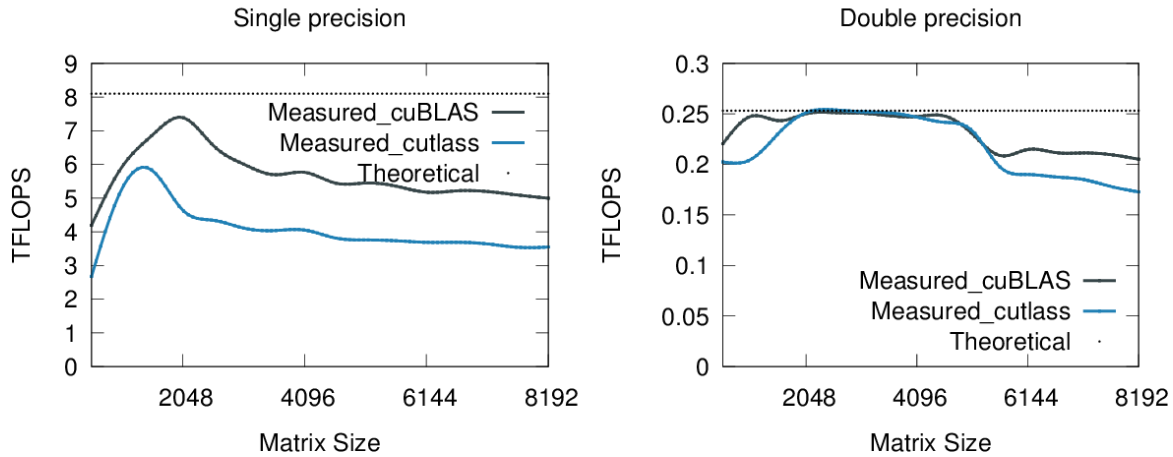
- cuBLAS 10.1 vs. CUTLASS 1.2 GEMM

- Arithmetic throughput:
  - half, single and double precision
    cuBLAS > CUTLASS
  - int8 precision
    CUTLASS > cuBLAS
    cuBLAS kernels don't use tensor cores
  - only CUTLASS support int4 and int1

- Except in double precision,
  all benchmarks don't achieve near-peak performance







Comparing arithmetic performance on T4 and P4

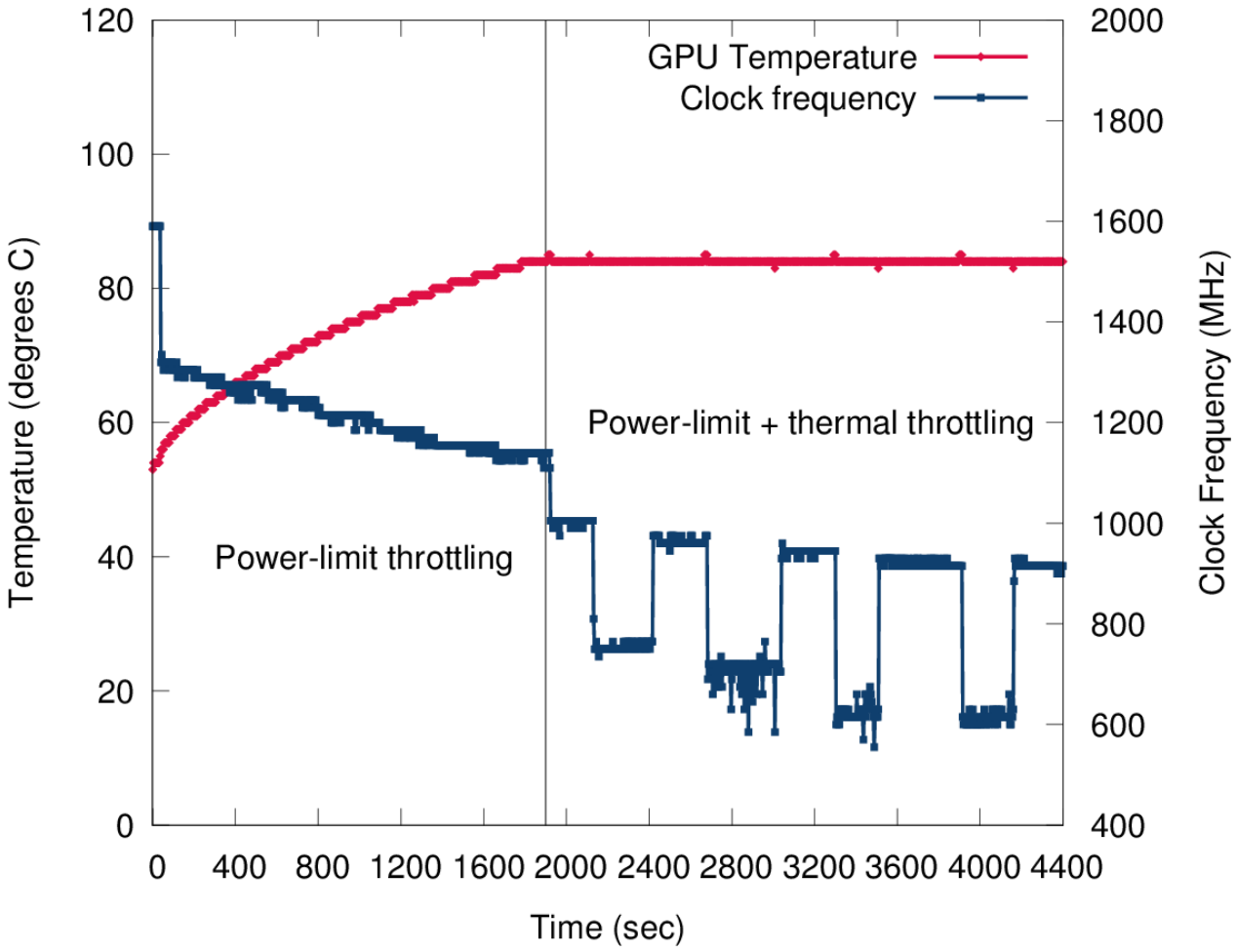
- Inference-oriented, same number of CUDA cores, similar max graphics frequencies
- Arithmetic throughput
  - single and double: very similar throughput
  - half and int8 precision:
    T4 has 6.3x more throughput than P4, thanks to tensor cores
  - int4 and int1:
    novel support on the T4!

	T4	P4	
Double precision	253	231	GFLOPS
Single precision	7,174	6,944	GFLOPS
Half precision	41,616	6,571	GFLOPS
Int8 precision	74,934	24,172	GOPS
Int4 precision	114,384	-	GOPS
Int1 precision	552,230	-	GOPS

	T4	P4
Max graphics frequency (MHz)	1,590	1,531
N of CUDA cores	2560	2560

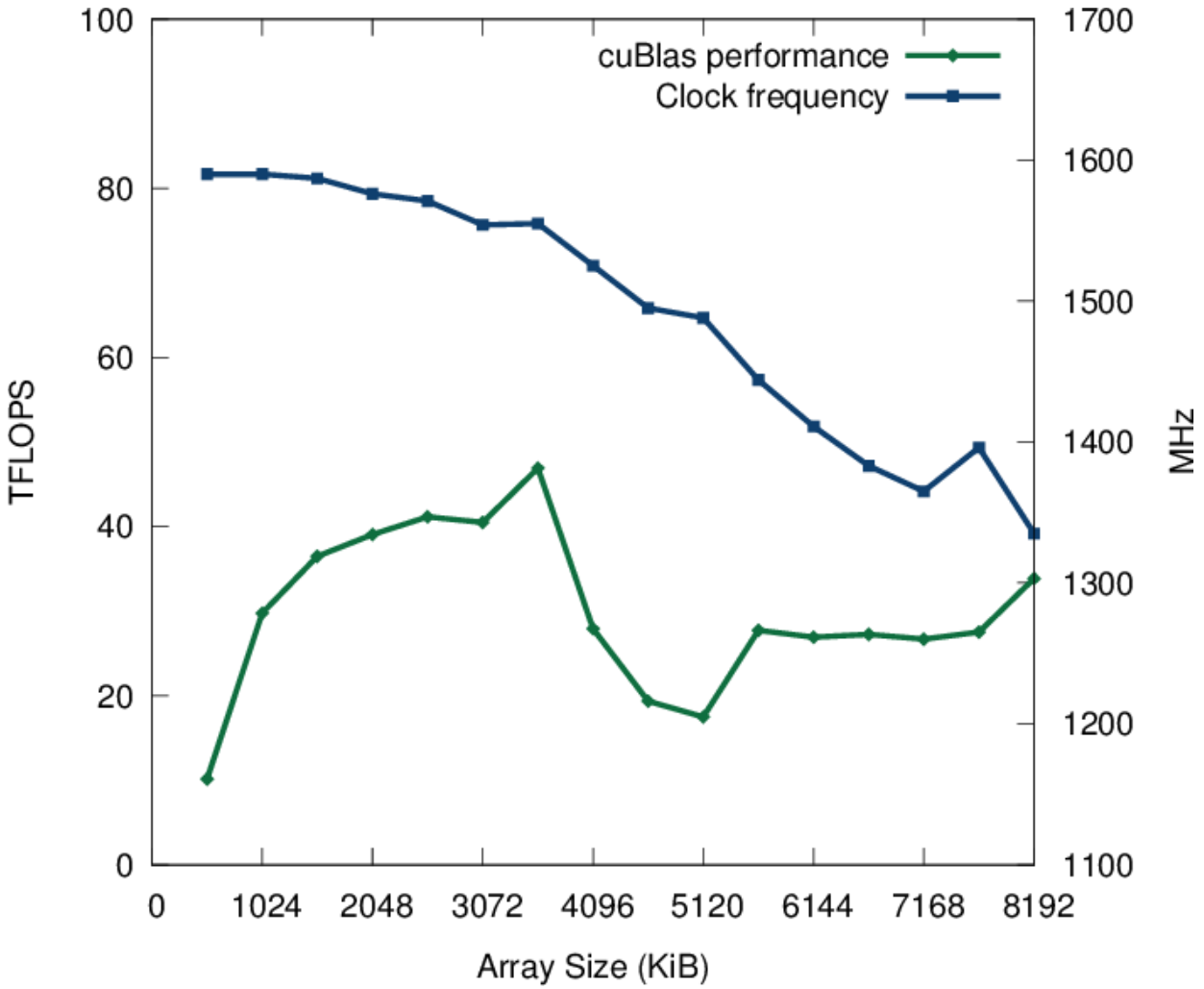
Power and thermal limits

Clock throttling causes T4 cannot achieve peak performance



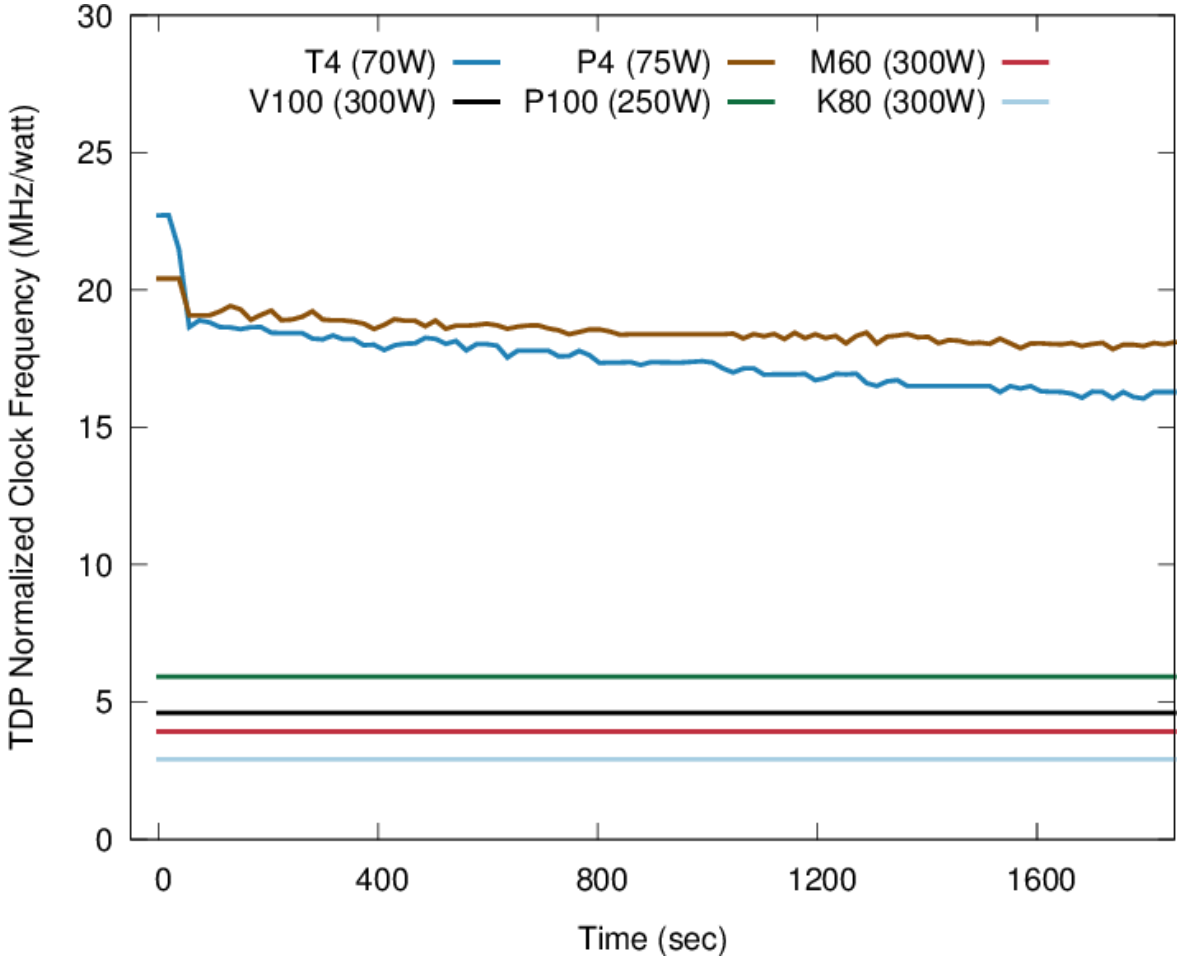
- Clock throttling reported from both:
  - Reduced silicon efficiency under rising temps
  - Max operating temperature limit
- Experiment:
  - Set initial clock frequency to 1,590 MHz
  - Repeated 4096x4096 cublasSgemm
  - Record both temperature and frequency
- Warmer transistors leak, so less power (and clocks) is available for computation
- Thermal limits are enforced by sharp intermittent clock frequency reductions.

Power-limit throttling relates to matrix size



- On T4, the power-limit throttling hurts overall arithmetic throughput
- Clock throttling with variable matrix sizes:
  - input matrices of progressively increased size
  - similar temperatures
  - bigger matrices → lower clocks/throughput

T4 and P4 boards are more prone to power throttling



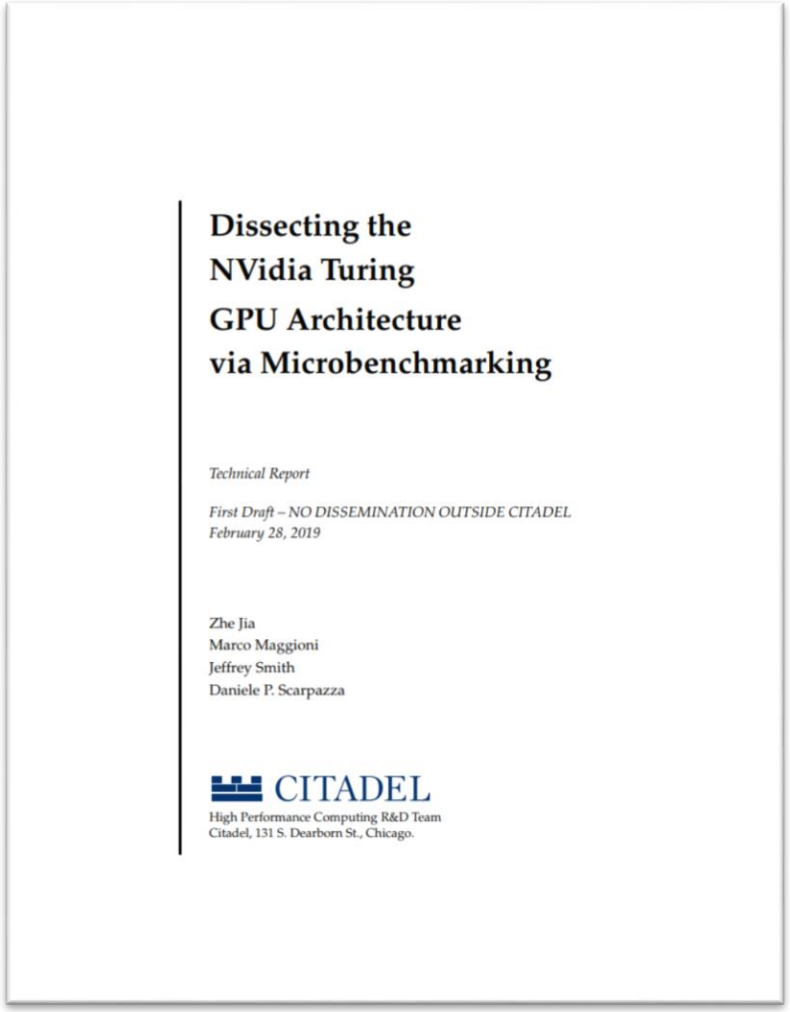
- Reason:
  - smaller die and cooler sizes
  - lower power limit
- Clock throttling on different devices:
  - input matrices of the same size
  - GPU limited by Max Operating Temps
  - on T4 and P4, power-limit throttling triggers immediately
  - on other GPUs, we observed barely any power limit throttling

Thank you!

- Download Now at: https://goo.gl/adPpwg

  Also in the process of publishing on arxiv.org

- Questions?



This presentation solely 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 herein have not been reviewed by, and may not reflect information known to other Citadel professionals