

Dissecting the Volta GPU Architecture through Microbenchmarking GTC 2018

Zhe Jia, Marco Maggioni, Benjamin Staiger, Daniele P. Scarpazza

High-Performance Computing Group



Everything You Ever Wanted To Know About Volta

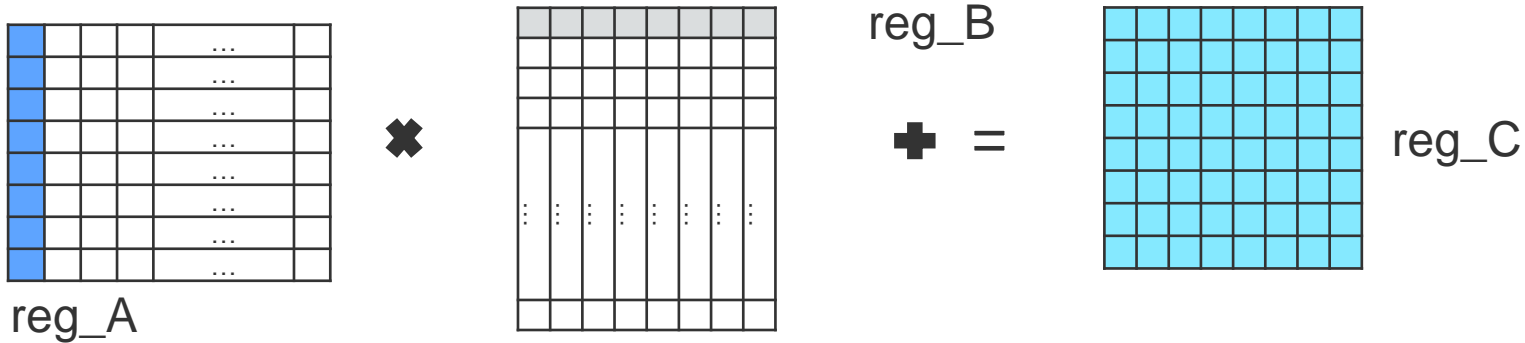
- Micro-architectural details matter – crucial to achieve peak performance
- Hard to keep up-to-date
 - new GPU generations every year
 - complexity increases at every generation
- Everything is better on Volta... but how much?
- We describe the inner workings of Volta
 - instruction encoding
 - size, properties, performance of each level in the memory hierarchy
 - latency of instructions
 - performance of atomic operations
 - performance of Tensor Cores and how their instructions operate
 - floating point throughput, at different precisions
 - host-device and peer-to-peer performance; both for PCI and NVLink devices
 - compare all findings against Pascal, Maxwell, Kepler
- ... a lot more than fits in a GTC presentation: technical report to come

Why Architectural Details Matter

- Example: simplest matrix-matrix multiplication core
 - we wrote it in CUDA C
 - compiled it with NVCC
 - we patched the binary instructions to
 - apply a better register mapping
 - increase use of register reuse caches
 - achieved a +15.4% speedup
 - this would be impossible without knowing
 - how instructions are encoded and
 - how register files are organized
 - ... and we discovered both in this very work
- Limitations of our approach
 - optimizing at such a low level requires substantial effort; it might not be worth it, except in very specific cases
 - our optimizations are device-dependent and not portable to future GPU generations
 - in a vast majority of cases, CUDA libraries and the NVCC compiler offer an excellent level of optimization and portability at the same time
 - optimizations delivered by NVCC and CUDA libraries will carry over to the next GPU generations for free

Microarchitectural Details Matter: A Case Study

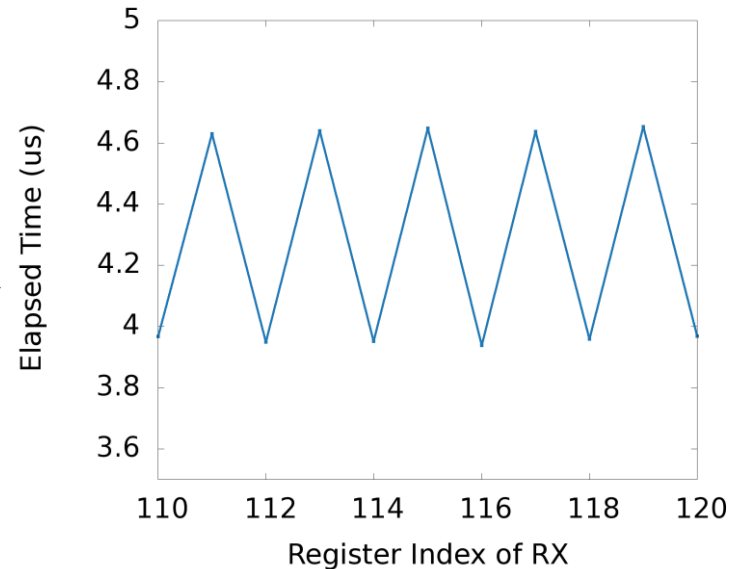
Simplest matrix multiplication kernel imaginable



```
float reg_A[8], reg_B[8], reg_C[64];
for (int k=0; k<512; k++) {
    // ...
    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];
    // ...
}
```

Case Study: Register Mapping Makes A Difference

- Volta register file has two 64-bit banks (bank 0 & bank 1)
 - Conflict: all 3 **operand** registers in the same bank
 - Bank 0: even numbered registers, e.g. R0, R2, R4, R6 ...
 - Bank 1: odd numbered registers, e.g. R1, R3, R5, R7 ...
 - Kepler, Maxwell and Pascal: 4 banks
-
- Elapsed time of identical “FFMA R6, R97, R99, RX” sequence
 - R97 and R99 are in bank 1
 - When RX is in bank 1, longer execution time



Case Study: Register Mapping Makes A Difference

Before

		12	13	14	15	8	9	10	11	← Reg_A
Reg_B	80	16	17	18	19	20	21	22	23	Bank 0
	81	24	25	26	27	28	29	30	31	Bank 1
	82	32	33	34	35	36	37	38	39	
	83	40	41	42	43	44	45	46	47	
	4	48	49	50	51	52	53	54	55	← Reg_C
	5	56	57	58	59	60	61	62	63	
	6	64	65	66	67	68	69	70	71	
	7	72	73	74	75	76	77	78	79	

Register mapping we adopt in our optimization:

After

		12	13	14	15	8	9	10	11	← Reg_A
Reg_B	80	17	25	33	41	49	57	65	73	Bank 0
	81	16	24	32	40	48	56	64	72	Bank 1
	82	19	27	35	43	51	59	67	75	
	83	18	26	34	42	50	58	66	74	
	4	21	29	37	45	53	61	69	77	← Reg_C
	5	20	28	36	44	52	60	68	76	
	6	23	31	39	47	55	63	71	79	
	7	22	30	38	46	54	62	70	78	

Case Study: Reuse Caches Makes A Difference

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
...	...

Performance improvement (128 threads): **+15.4%**

How Volta Encodes Instructions And Control

Kepler:

control for 7 instructions

```

/*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, 0xffffffffc; /* 0x7cc00000001c0002 */
/* 0x207ffffffe1c0c08 */

```

Maxwell

Pascal:

control for 3 instructions

```

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

```

Volta

control for 1 instruction

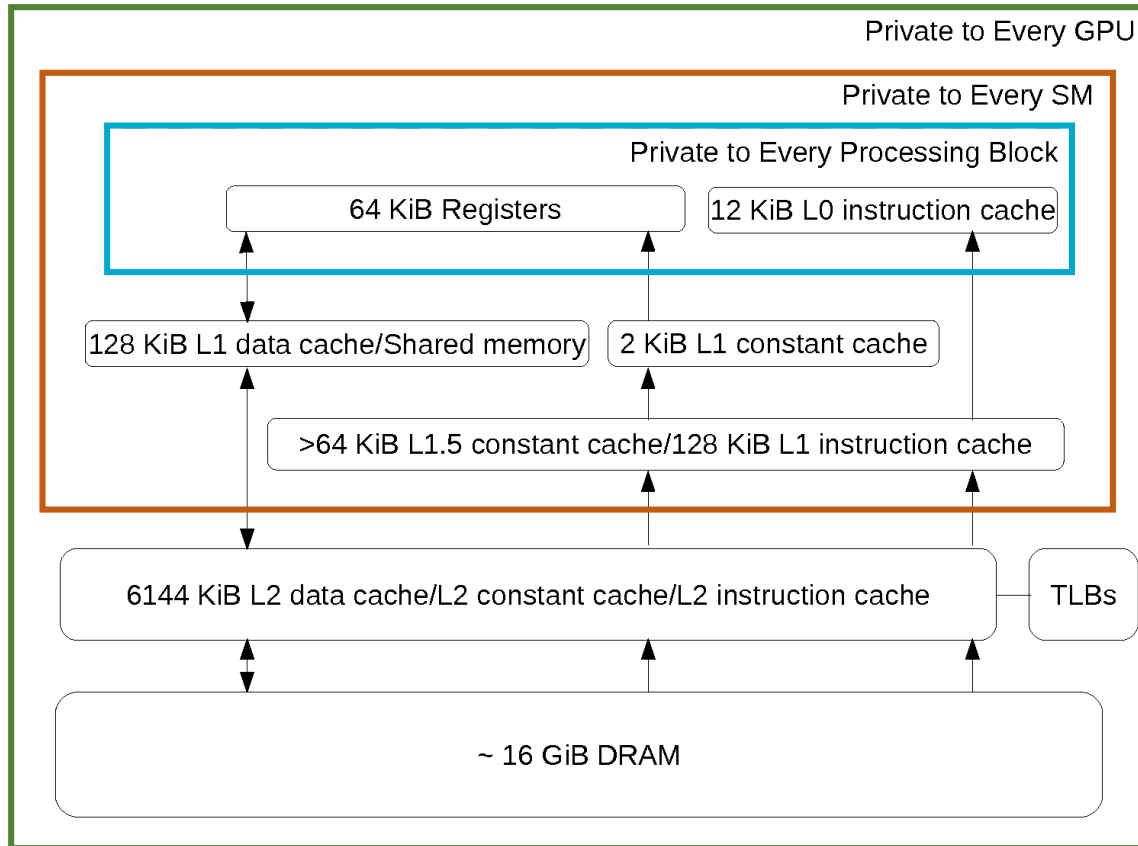
```

/*0000*/ @!PT SHFL.IDX PT, RZ, RZ, RZ, RZ; /* 0x000000ffffffff389 */
/* 0x000fe200000e00ff */

```

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

Volta Memory Hierarchy

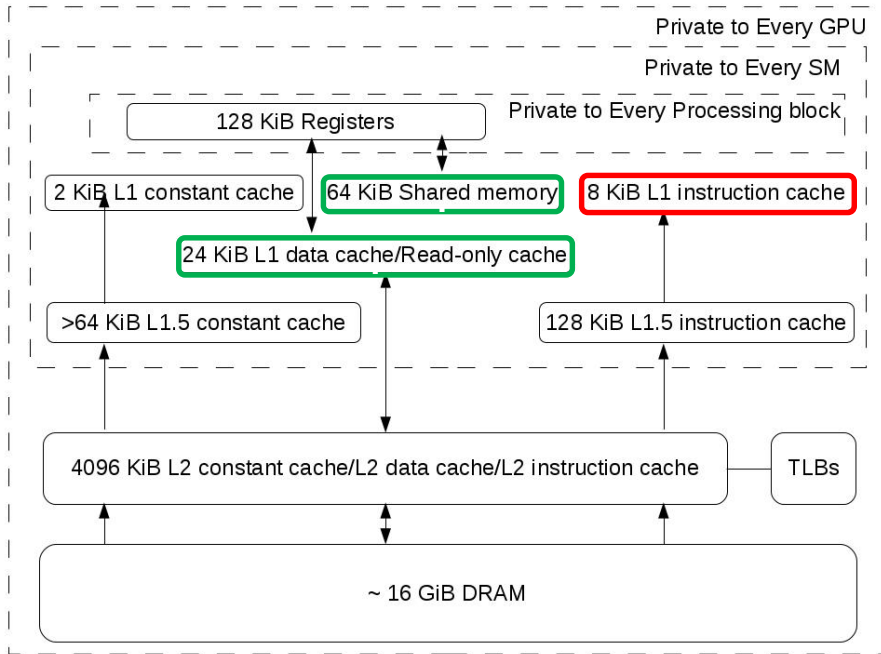


memory hierarchy for V100 GPU

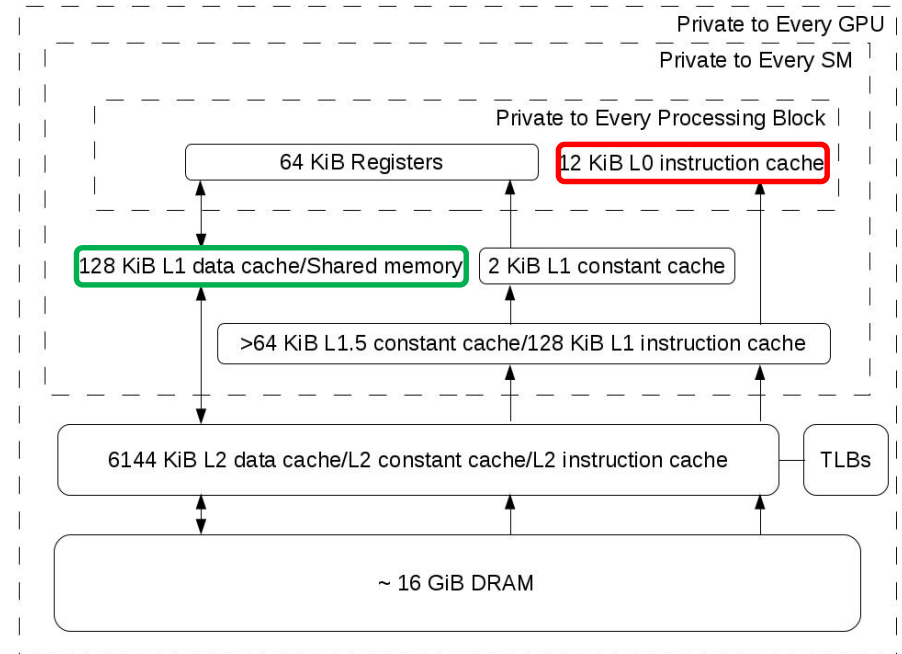
- 4 Processing Blocks (PB) on every Streaming Multiprocessor (SM)
- 80 SMs on Every GPU
- 3 levels of instruction cache: L0 is private to every PB
- 3 levels of constant cache
- 2 levels of data cache: L1 combined with shared memory

Memory Hierarchy: Volta vs. Pascal

P100



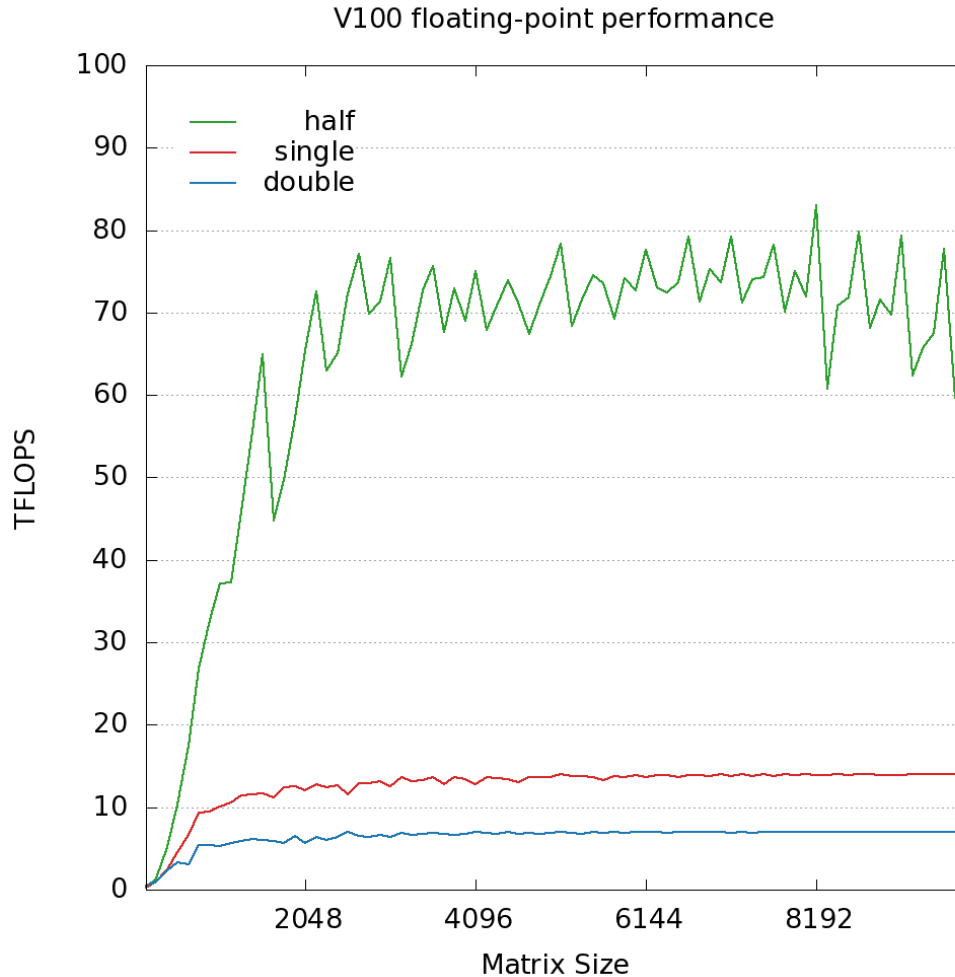
V100



	P100	V100
N of SMs	56	80
Processing block per SM	2	4

- Volta instruction cache: 12 KiB L0 in every processing block, no L1
- Pascal instruction cache: no L0, 8 KiB L1 in every SM
- Volta has combined L1 cache/shared memory

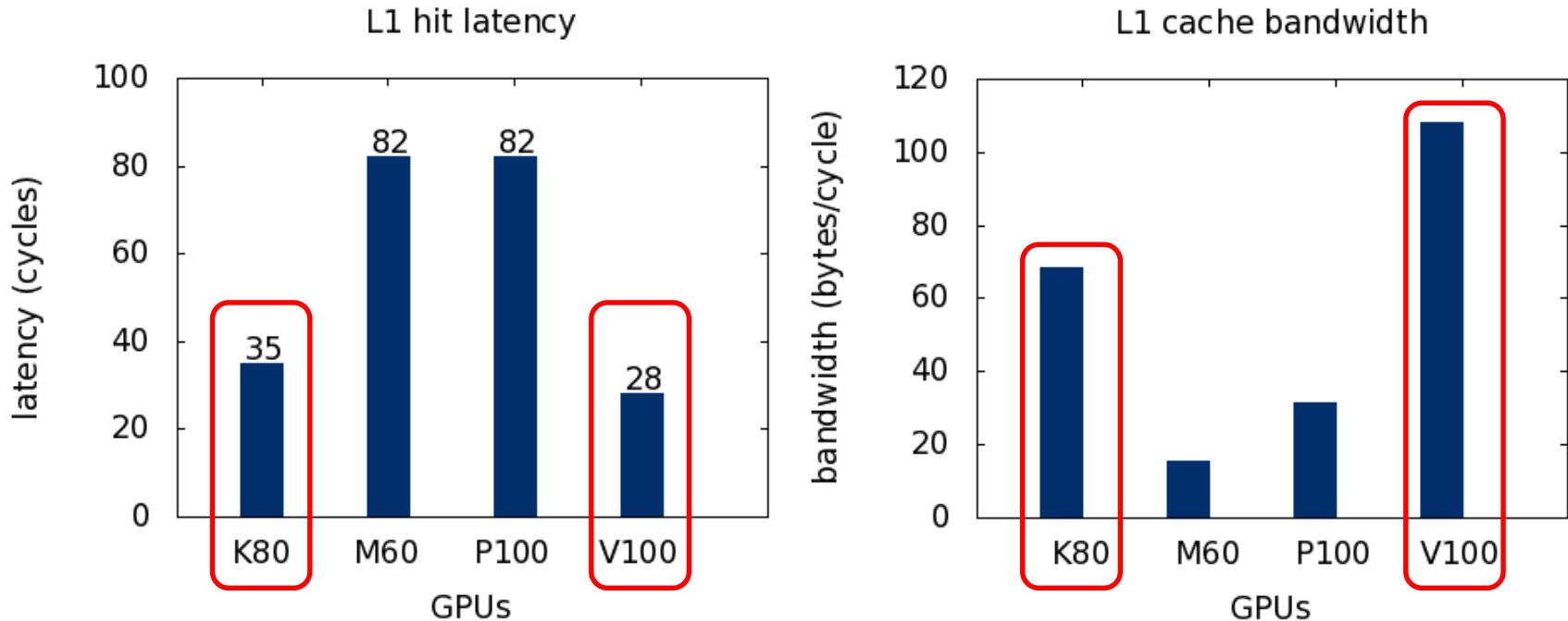
Floating Point Performance On V100



- Matrix-matrix multiplication performance with cuBLAS from CUDA 9.0
- Measured half precision performance is 5.7x of single precision performance
- cuBLAS library achieves 70% of peak performance on Tensor cores
- Theoretical performance
 - Half precision: 113 TFLOPS
 - Single precision: 14 TFLOPS
 - Double precision: 7 TFLOPS

Combined L1 Cache/Shared Memory

Volta is like Kepler: L1 and shared memory are combined Low latency, high bandwidth



- new replacement policy: Volta keeps replacing the same cache lines first when L1 is saturated.

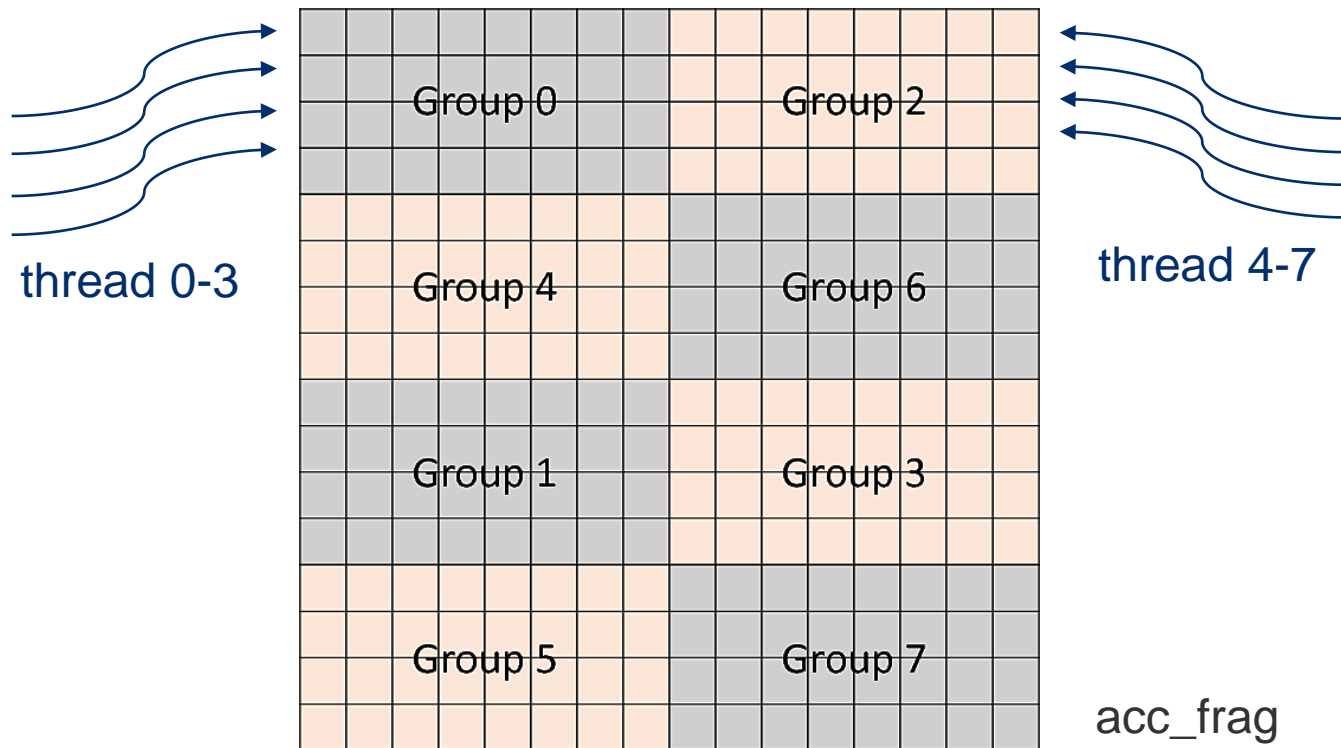
Instruction Latency: Improved

Instruction latency on Volta: widely improved

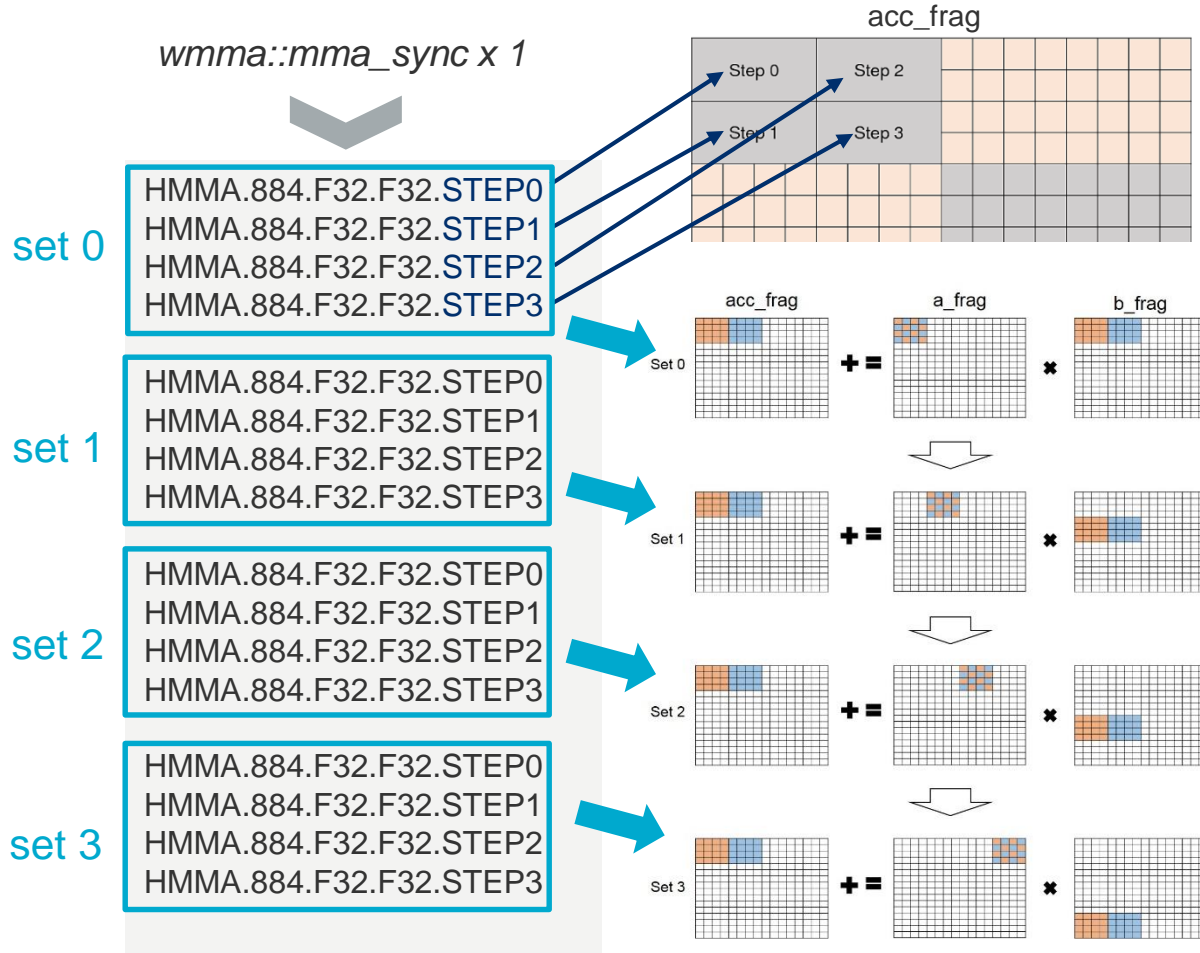
Architecture	Instructions	Latency (cycles)
Pascal	BFE, BFI, IADD, IADD32I, FADD, FMUL, FFMA, FMNMX, HADD2, HMUL2, HFMA2, IMNMX, ISCADD, LOP, LOP32I, LOP3, MOV, MOV32I, SEL, SHL, SHR, VADD, VABSDIFF, VMNMX, XMAD	6
	DADD, DMUL, DFMA, DMNMX	8
	FSET, DSET, DSETP, ISETP, FSETP	12
	POPC, FLO, MUFU, F2F, F2I, 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, FLO, BREV, MUFU	10 14

Tensor Cores: How Do They Work

- use warp-level primitive “*wmma::mma_sync*” to calculate $\text{acc_frag}(16 \times 16) += \text{a_frag}(16 \times 16) \times \text{b_frag}(16 \times 16)$
- 32 threads in a warp are divided in 8 groups,
- every 4 threads update an area in *acc_frag*



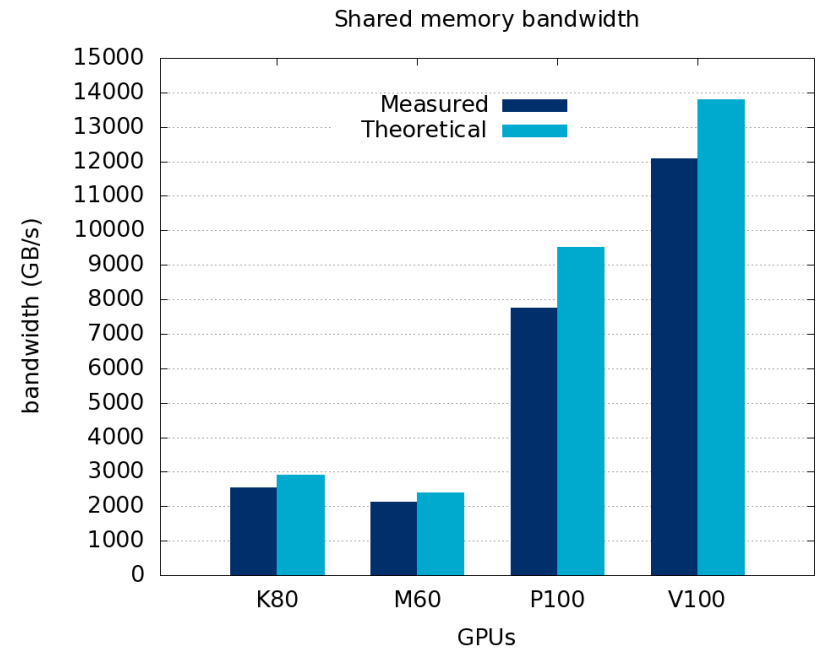
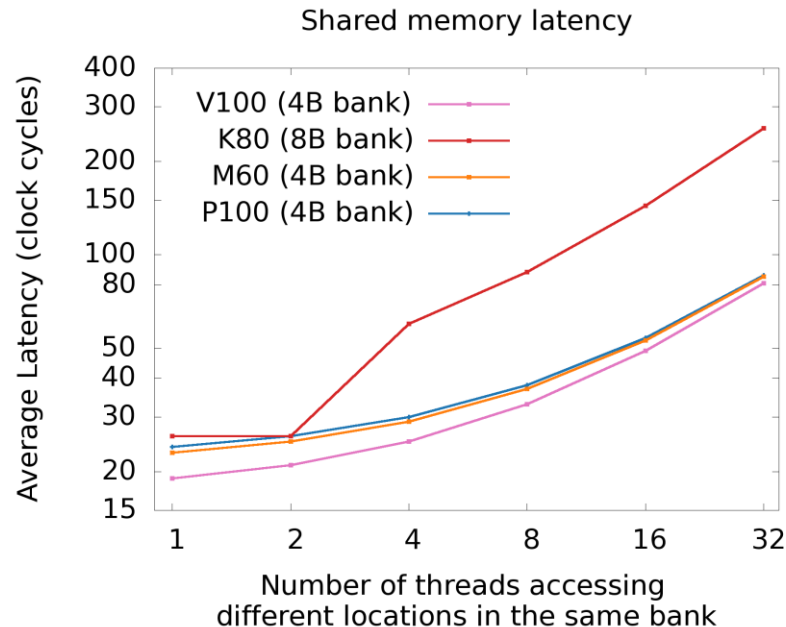
Tensor Cores: How Do They Work



- At compile time, NVCC translates one “*wmma::mma_sync*” to 16 “HMMA” instructions
- We call every 4 instructions a “set”
- At run time, different sets read from different areas in *a_frag* and *b_frag*, accumulate into same positions in *acc_frag*
- Within every set, different “STEP” flags control the updating in different areas of *acc_frag*

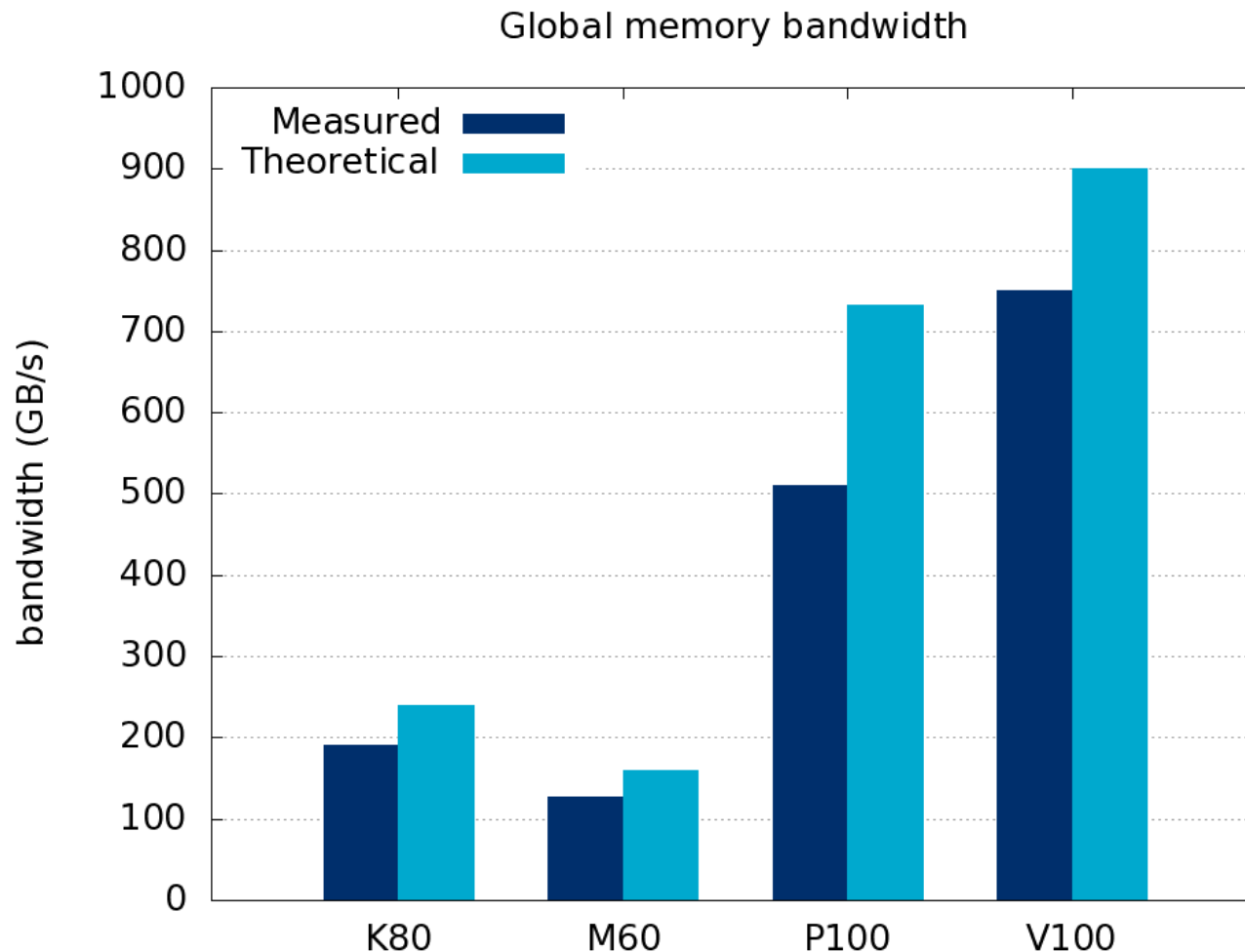
Shared Memory Performance: From Kepler To Volta

- Shared memory
 - Latency decreases **significantly** from Kepler to Volta
 - Bandwidth increase **significantly** after Maxwell



Global Memory: From Kepler To Volta

Bandwidth increases significantly thanks to HBM2 memory



Atomic Instructions: From Kepler To Volta

- Volta has the fastest atomic operations on shared memory in all contention scenarios
- On global memory, Volta doesn't win
- Kepler: shared memory atomics are very slow because they are emulated

Contention	Shared memory				Global memory			
	V100	P100	M60	K80	V100	P100	M60	K80
None	6	15	17	93	36	26	24	29
2 threads	7	17	19	214	31	31	26	69
4 threads	11	19	25	460	32	48	41	96
8 threads	18	30	31	952	41	48	41	152
16 threads	24	46	47	1936	58	50	46	264
32 threads	66	78	79	4257	76	50	46	488

What Hasn't Changed Across GPU Generations

- Unified L2 data cache
 - For all data, constant memory and instruction accesses
 - Memory copy operations populate the L2 cache
- TLB (Kepler and Maxwell: 2 levels, Pascal and Volta: 3 levels)
 - L1 cache is indexed by virtual addresses
 - L2 cache is indexed by physical addresses
- 3 levels of constant cache (L1, L1.5 and L2)
 - 4-way L1 with 64 B lines
 - L1 and L1.5 are private to every SM
 - L2 constant cache is shared by all SMs
- 3 levels of Instruction cache
 - Volta: L0 (per processing block), L1 (per SMX) and L2 (all SMX)
 - Kepler to Pascal: L1&L1.5 (per SMX), L2 (all SMX)

Stay tuned for our Technical Report

- all these findings and much more!
- in a 60+-page technical report
- we will publish it on arxiv.org
- April 9th 2018
- Stay tuned!

Dissecting the Volta GPU Architecture via Microbenchmarking

Technical Report
Draft; Citadel designation here.

Zhe Jia,
Marco Maggioni,
Benjamin Staiger,
Daniele P. Scarpazza

 CITADEL | Securities

19

Table 4.1: Geometry, properties and latency of the memory hierarchy on the Volta, Pascal, Maxwell and Kepler architectures. All data in this table are measured on PCI-E cards.

		Volta V100 GV100	Pascal P100 GP100	Pascal P4 GP104	Maxwell M60 GM204	Kepler K80 GK210
Registers	Number of banks	2	4	4	4	4
	bank width	64 bit	32 bit	32 bit	32 bit	32 bit
L1 data	Size	32..128 KiB	24 KiB	24 KiB	24 KiB	16..48 KiB
	Line size	32 B	32 B	32 B	32 B	128 B
	Hit latency	28	82	82	82	35
	Number of sets	4	4	4	4	32 or 64*
	Load granularity	32 B	32 B	32 B	32 B	128 B
	Update granularity	128 B	128 B	128 B	128 B	128 B
	Update policy	non-LRU	LRU	LRU	LRU	non-LRU
	Physical address indexed	no	no	no	no	no
L2 data	Size	6,144 KiB	4,096 KiB	2,048 KiB	2,048 KiB	1,536 KiB
	Line size	64 B	32 B	32 B	32 B	32 B
	Hit latency	~193	~234	~216	~207	~200
	Populated by cudaMemcpy	yes	yes	yes	yes	yes
	Physical address indexed	yes	yes	yes	yes	yes
L1 constant	Broadcast latency	~27	~24	~25	~25	~30
	Cache size	2 KiB	2 KiB	2 KiB	2 KiB	2 KiB
	Line size	64 B	64 B	64 B	64 B	64 B
	Number of sets	8	8	8	8	8
	Associativity	4	4	4	4	4
L1.5 constant	Broadcast latency	~89	~96	~87	~81	~92
	Cache size	>=64 KiB	>=64 KiB	32 KiB	32 KiB	32 KiB
	Line size	256 B	256 B	256 B	256 B	256 B
	Broadcast latency	~245	~236	~225	~221	~220
L0 instruction	Cache size	~12 KiB	-	-	-	-
L1 instruction	Cache size	128 KiB	8 KiB	8 KiB	8 KiB	8 KiB
L1.5 instruction	Cache size	128 KiB	128 KiB	32 KiB	32 KiB	32 KiB
	SMX private or shared	-	private	private	private	private
L2 instruction	Cache size	6,144 KiB	4,096 KiB	2,048 KiB	2,048 KiB	1,536 KiB
L1 TLB	Coverage	32 MiB	~32 MiB	~32 MiB	~2 MiB	~2 MiB
	Page entry	2 MiB	2 MiB	2 MiB	128 KiB	128 KiB
L2 TLB	Coverage	~8,192 MiB	~2,048 MiB	~2,048 MiB	~128 MiB	~128 MiB
	Page entry	32 MiB	32 MiB	32 MiB	2 MiB	2 MiB
L3 TLB	Coverage	-	-	-	~2,048 MiB	~2,048 MiB
	Page entry	-	-	-	2 MiB	2 MiB
Specifications	Processors per chip (P)	80	56	20	16	13
	Max graphics clock (f_g)	1,380 MHz	1,328 MHz	1,531 MHz	1,177 MHz	875 MHz
Shared memory	Size per SMX	up to 96 KiB	64 KiB	64 KiB	96 KiB	48 KiB
	Size per chip	up to 7,689 KiB	3,584 KiB	1,280 KiB	1,536 KiB	624 KiB
	Banks per processor (B_s)	32	32	32	32	32
	Bank width (w_s)	4 B	4 B	4 B	4 B	8 B
	No-conflict latency	19	24	23	23	26
	Theoretical bandwidth	13,800 GiB/s	9,519 GiB/s	3,919 GiB/s	2,410 GiB/s	2,912 GiB/s
	Measured bandwidth	12,080 GiB/s	7,763 GiB/s	3,559 GiB/s	2,122 GiB/s	2,540 GiB/s
Global memory	Memory bus	HBM2	HBM2	GDDR5	GDDR5	GDDR5
	Size	16,152 MiB	16,276 MiB	8,115 MiB	8,115 MiB	12,237 MiB
	Max clock rate (f_m)	877 MHz	715 MHz	3,003 MHz	2,505 MHz	2,505 MHz
	Theoretical bandwidth	900 GiB/s	732 GiB/s	192 GiB/s	160 GiB/s	240 GiB/s
	Measured bandwidth	750 GiB/s	510 GiB/s	162 GiB/s	127 GiB/s	191 GiB/s
	Measured/Theoretical Ratio	83.3%	69.6%	84.4%	79.3%	77.5%

Thank you!
Questions welcome