King Abdullah University of Science and Technology

KAUST

CS 380 - GPU and GPGPU Programming
Lecture 11: GPU Architecture, Pt. 9; GPU Compute APIs, Pt. 1

Markus Hadwiger, KAUST

Reading Assignment #6 (until Oct 7)

Read (required):

- Programming Massively Parallel Processors book (4th edition), **Chapter 3** (*Multidimensional grids and data*)

Read (optional):

- Inline PTX Assembly in CUDA: `Inline_PTX_Assembly.pdf`
- Dissecting GPU Architectures through Microbenchmarking:

  Volta: `https://arxiv.org/abs/1804.06826`
  Turing: `https://arxiv.org/abs/1903.07486`

`https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9839-discovering-the-turing-t4-gpu-architecture-with-microbenchmarks.pdf`

Ampere: `https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s33322/`

Next Lectures

Lecture 12: Tue, Oct 1 (make-up lecture; 14:30 – 15:45)

Lecture 13: Thu, Oct 3

Lecture 14: Mon, Oct 7

Lecture 15: Tue, Oct 8 (make-up lecture; 14:30 – 15:45)

Lecture 16: Thu, Oct 10

no lecture on Oct 14 !

Lecture 17: Tue, Oct 15: Vulkan tutorial ? (tbd)

Lecture 18: Thu, Oct 17: Quiz #2 (only quiz)

Lecture 19: Mon, Oct 21

Lecture 20: Tue, Oct 22 (make-up lecture; 14:30 – 15:45)

Lecture 21: Thu, Oct 24

NVIDIA Architectures (since first CUDA GPU)

Tesla [CC 1.x]: 2007-2009

- G80, G9x: 2007 (Geforce 8800, ...)
  GT200: 2008/2009 (GTX 280, ...)

Fermi [CC 2.x]: 2010 (2011, 2012, 2013, …)

- GF100, ... (GTX 480, ...)
  GF104, ... (GTX 460, ...)
  GF110, ... (GTX 580, ...)

Kepler [CC 3.x]: 2012 (2013, 2014, 2016, …)

- GK104, ... (GTX 680, ...)
  GK110, ... (GTX 780, GTX Titan, ...)

Maxwell [CC 5.x]: 2015

- GM107, ... (GTX 750Ti, ...)
  GM204, ... (GTX 980, Titan X, ...)

Pascal [CC 6.x]: 2016 (2017, 2018, 2021, 2022, …)

- GP100 (Tesla P100, ...)
- GP10x: x=2,4,6,7,8, ...
  (GTX 1060, 1070, 1080, Titan X *Pascal*, Titan Xp, ...)

Volta [CC 7.0, 7.2]: 2017/2018

- GV100, ...
  (Tesla V100, Titan V, Quadro GV100, ...)

Turing [CC 7.5]: 2018/2019

- TU102, TU104, TU106, TU116, TU117, ...
  (Titan RTX, RTX 2070, 2080 (Ti), GTX 1650, 1660, ...)

Ampere [CC 8.0, 8.6, 8.7]: 2020

- GA100, GA102, GA104, GA106, ...
  (A100, RTX 3070, 3080, 3090 (Ti), RTX A6000, ...)

Hopper [CC 9.0], Ada Lovelace [CC 8.9]: 2022/23

- GH100, AD102, AD103, AD104, ...
  (H100, L40, RTX 4080 (12/16 GB), 4090, RTX 6000, ...)

Blackwell [CC 10.0]: *coming in 2024/25*

- GB200/GB202, GB20x, ...?
  (RTX 5080/5090, GB200 NVL72, HGX B100/200, ...?)

see `https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units`
and `https://en.wikipedia.org/wiki/CUDA`

NVIDIA Volta Architecture

2017/2018

(compute capability 7.0/7.2)

GV100 (cc 7.0), ... (Titan V, Tesla V100, ...)
GV10B, GV11B (cc 7.2), ... (Tegra Xavier, ...)

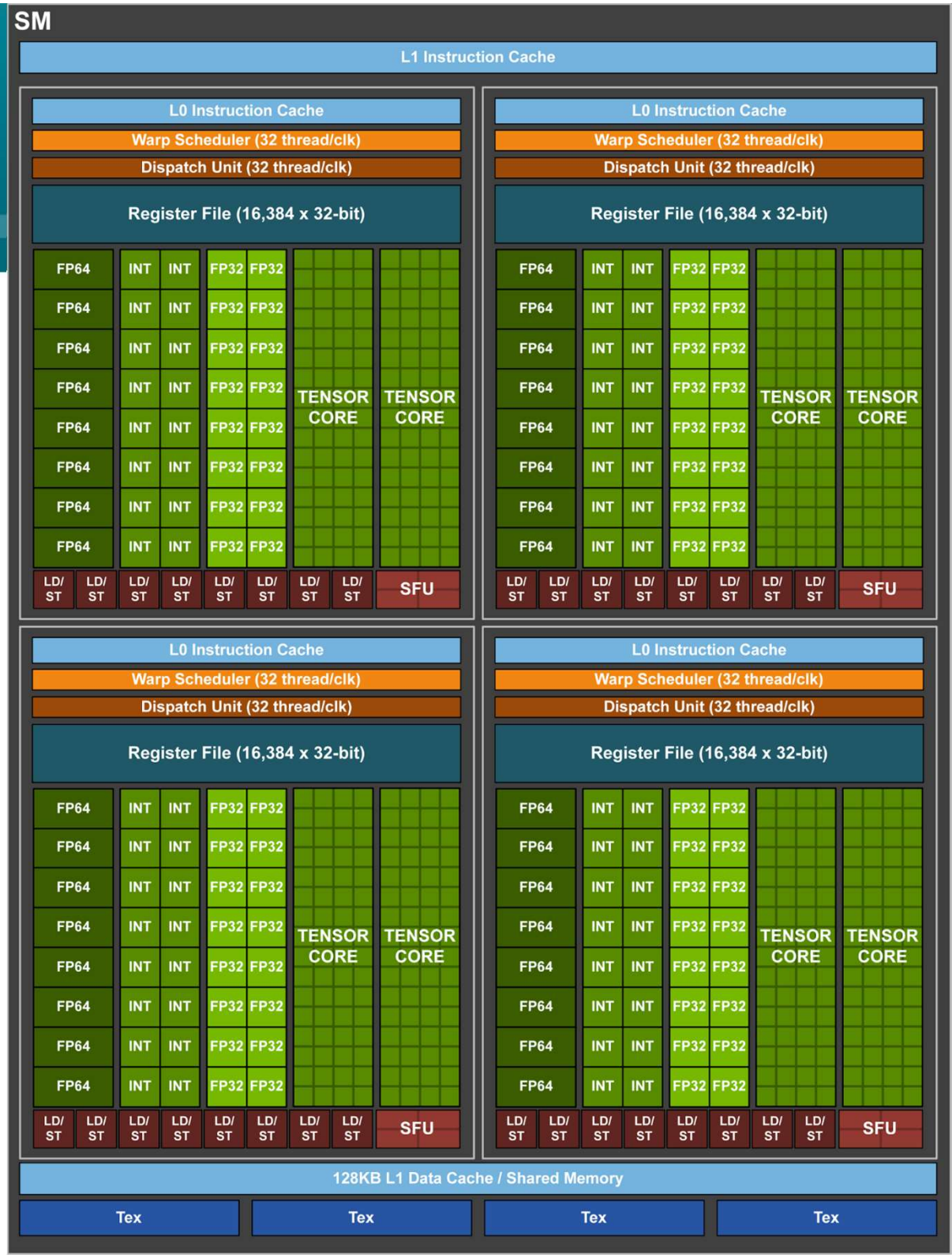
NVIDIA Volta SM

Multiprocessor: SM (CC 7.0)

- 64 FP32 + 64 INT32 cores
- 32 FP64 cores
- 32 LD/ST units; 16 SFUs
- 8 tensor cores (FP16/FP32 mixed-precision)

4 partitions inside SM

- 16 FP32 + 16 INT32 cores each
- 8 FP64 cores each
- 8 LD/ST units; 4 SFUs each
- 2 tensor cores each
- Each has: warp scheduler, dispatch unit, register file



Tensor Cores

Mixed-precision, fast matrix-matrix multiply and accumulate

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32 — FP16 — FP16 — FP16 or FP32

From this, build larger sizes, higher dimensionalities, ...

[+Tensor cores on later architectures add more data types/precisions!]

NVIDIA Volta Architecture (2017/2018)

Total chip capacity on Tesla V100 (GV100 architecture)

- 80 SMs
  - 64 FP32 cores / SM = 5,120 FP32 cores in total
  - 64 INT32 cores / SM = 5,120 INT32 cores in total
  - 32 FP64 cores / SM = 2,560 FP64 cores in total
  - 4 FP16/FP32 mixed-prec. tensor cores = 650 tensor cores in total
- 40 TPCs (2 SMs per TPC)
- 6 GPCs

Maximum capacity would be 84 SMs and 42 TPCs

NVIDIA Turing Architecture

2018/2019

(compute capability 7.5)

TU102, TU104, TU106, TU116, ... (cc 7.5)
(Titan RTX, RTX 2070, 2080, 2080Ti, Tesla T4, ...)

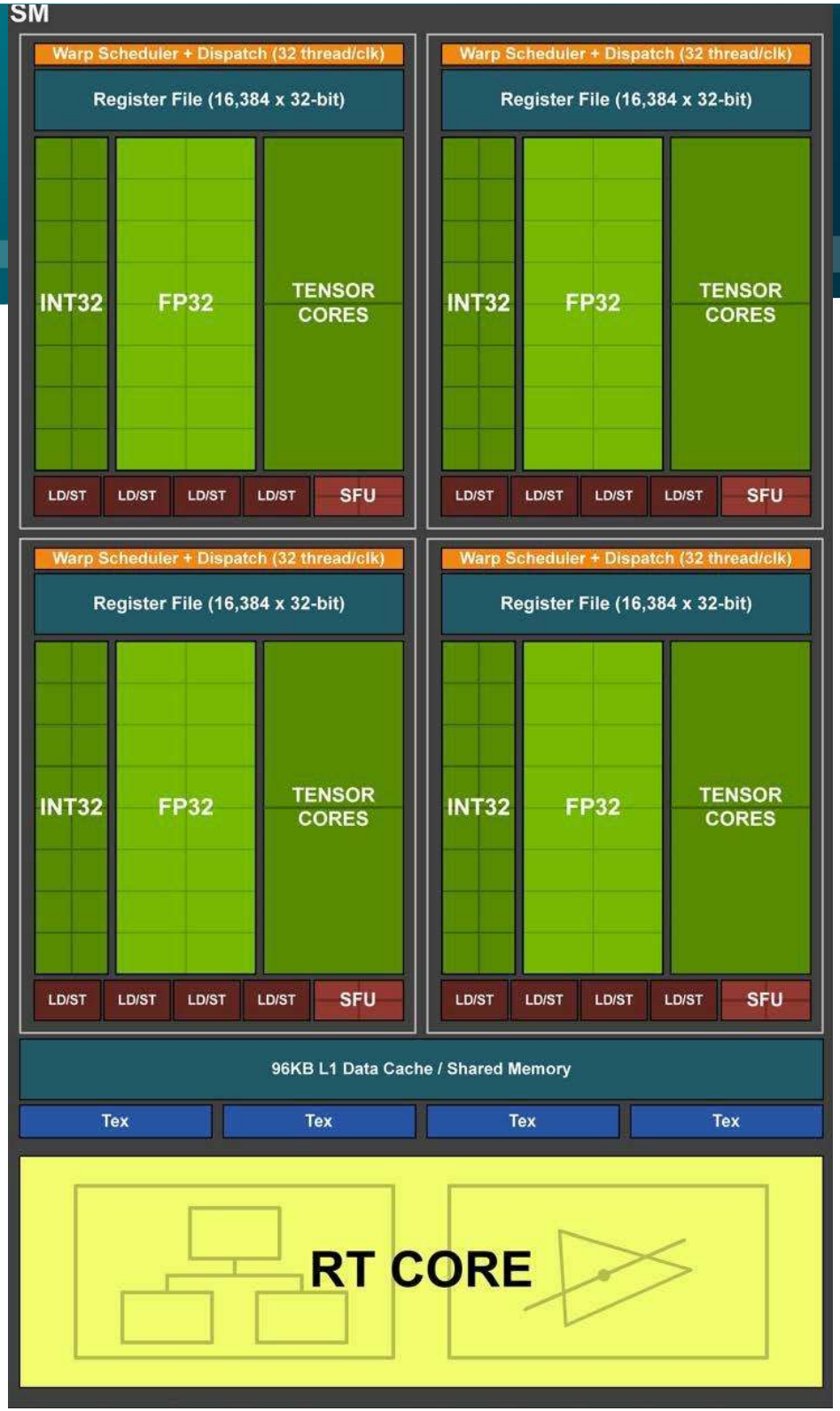
NVIDIA Turing SM

Multiprocessor: SM (CC 7.5)

- 64 FP32 + INT32 cores
- 2 (!) FP64 cores
- 8 Turing tensor cores (FP16/32, INT4/8 mixed-precision)
- 1 RT (ray tracing) core

4 partitions inside SM

- 16 FP32 + INT32 cores each
- 4 LD/ST units; 4 SFUs each
- 2 Turing tensor cores each
- Each has: warp scheduler, dispatch unit, 16K register file



NVIDIA Ampere Architecture

2020

(compute capability 8.0/8.6/8.7)

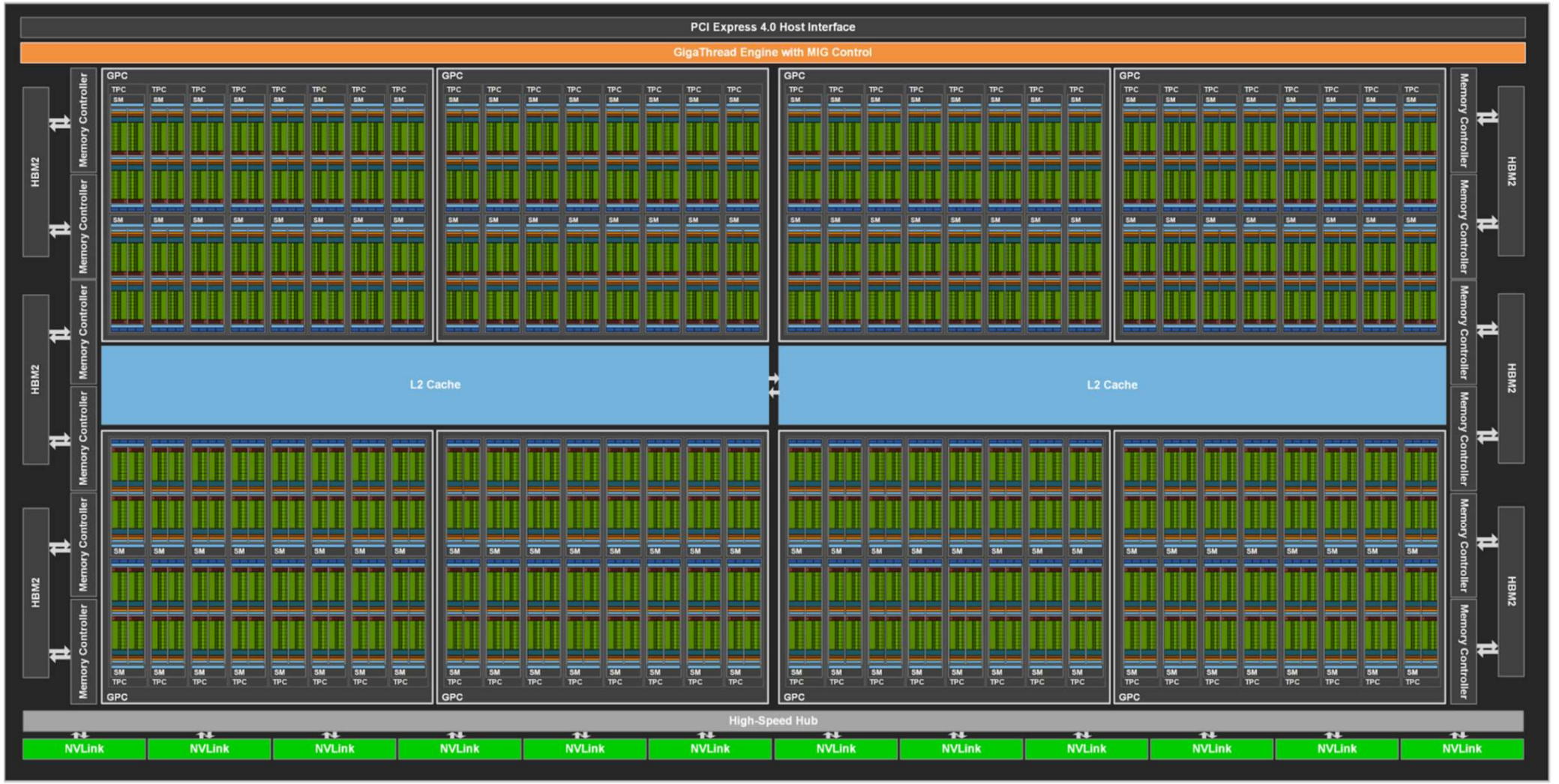
GA100 (cc 8.0), ... (A100, ...)

(x=2,3,4,6,7) GA10x (cc 8.6), ... (RTX 3070, RTX 3080, RTX 3090, ...)

GA10B (cc 8.7), ... (Jetson, DRIVE, ...)

NVIDIA Ampere GA100 Architecture (2020)

GA 100 (A100 Tensor Core GPU) Full GPU: 128 SMs (in 8 GPCs/64 TPCs)



Instruction Throughput

Instruction throughput numbers in CUDA C Programming Guide (Chapter 8.4)

	Compute Capability										
	3.5, 3.7	5.0, 5.2	5.3	6.0	6.1	6.2	7.x	8.0	8.6	8.9	9.0
16-bit floating-point add, multiply, multiply-add	N/A		256	128	2	256	128	256		128	256
32-bit floating-point add, multiply, multiply-add	192	128		64	128			64		128	
64-bit floating-point add, multiply, multiply-add	64	4		32	4		32	32	2	2	64

128 for __nv_bfloat16

128 for __nv_bfloat16

8 for GeForce GPUs, except for Titan GPUs

2 for compute capability 7.5 GPUs

ALU Instruction Latencies and Instructs. / SM

CC	2.0 (Fermi)	2.1 (Fermi)	3.x (Kepler)	5.x (Maxwell)	6.0 (Pascal)	6.1/6.2 (Pascal)	7.x (Volta, Turing)	8.0/8.6 (Ampere)	8.9/9.0 (Ada/Hopper)
# warp sched. / SM	2	2	4	4	2	4	4	4	4
# ALU dispatch / warp sched.	1 (over 2 clocks)	2 (over 2 clocks)	2	1	1	1	1	1	1
SM busy with # warps + inst	L	2L	8L	4L	2L	4L	4L	4L	4L
inst. pipe latency (L)	22	22	11	9	6	6	4	4	4
SM busy with # warps	22	22 + ILP	44 + ILP	36	12	24	16	16	16

see NVIDIA CUDA C Programming Guides (different versions) performance guidelines/multiprocessor level; compute capabilities

NVIDIA GA100 SM

Multiprocessor: SM (CC 8.0)

- 64 FP32 + 64 INT32 cores
- 32 FP64 cores
- 4 3rd gen tensor cores
- 1 2nd gen RT (ray tracing) core

4 partitions inside SM

- 16 FP32 + 16 INT32 cores
- 8 FP64 cores
- 8 LD/ST units; 4 SFUs each
- 1 3rd gen tensor core each
- Each has: warp scheduler, dispatch unit, 16K register file

NVIDIA Ampere GA10x Architecture (2020)

GA 102 (RTX 3070, 3080, 3090)

Full GPU: 84 SMs (in 7 GPCs/42 TPCs)

NVIDIA GA10x SM

Multiprocessor: SM (CC 8.6)

- 128 (64+64) FP32 + 64 INT32 cores
- 2 (!) FP64 cores
- 4 3rd gen tensor cores
- 1 2nd gen RT (ray tracing) core

4 partitions inside SM

- 32 (16+16) FP32 + 16 INT32 cores
- 4 LD/ST units; 4 SFUs each
- 1 3rd gen tensor core each
- Each has: warp scheduler, dispatch unit, 16K register file

Specs CC 3.5 – 9.0

Ampere: 8.0, 8.6, 8.7

(8.9 / Ada missing in table)

(CUDA C Programming Guide 11.8, Table 15)

	Compute Capability														
Technical Specifications	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.0**	**7.2**	**7.5**	**8.0**	**8.6**	**8.7**	**9.0**
Maximum number of resident grids per device (Concurrent Kernel Execution)	32				16	128	32	16	128	16	128				
Maximum dimensionality of grid of thread blocks	3														
Maximum x-dimension of a grid of thread blocks	$2^{31}-1$														
Maximum y- or z-dimension of a grid of thread blocks	65535														
Maximum dimensionality of a thread block	3														
Maximum x- or y-dimension of a block	1024														
Maximum z-dimension of a block	64														
Maximum number of threads per block	1024														
Warp size	32														
Maximum number of resident blocks per SM	16		32								16	32	16		32
Maximum number of resident warps per SM	64										32	64	48		64
Maximum number of resident threads per SM	2048										1024	2048	1536		2048
Number of 32-bit registers per SM	64 K	128 K	64 K												
Maximum number of 32-bit registers per thread block	64 K				32 K	64 K		32 K	64 K						
Maximum number of 32-bit registers per thread	255														

NVIDIA Ampere GA102 Architecture (2020)

GA 102 (RTX 3070, 3080, 3090, **A40**) Full GPU: 84 SMs (in 7 GPCs/42 TPCs)

- 64K 32-bit registers / SM = 256 KB register storage per SM
- 128 KB shared memory / L1 per SM

For 84 SMs on full GPU *[RTX 3090: 82 SMs]*

- 21 MB register storage, 10.5 MB shared mem / L1 storage =
  31.5 MB context+"shared context" storage !
- L2 cache size on A40, RTX 3090: 6 MB
- 10,752 FP32 cores (128 FP32 cores per SM) *[RTX 3090: 10,496]*
- 129,024 max threads in flight (max warps / SM = 48) *[RTX 3090: 125,952]*

NVIDIA Ampere GA100 Architecture (2020)

GA 100 (A100) Full GPU: 128 SMs (in 8 GPCs/64 TPCs)

- 64K 32-bit registers / SM = 256 KB register storage per SM
- 192 KB shared memory / L1 per SM

For 128 SMs on full GPU *[A100: 108 SMs]*

- 32 MB register storage, 24 MB shared mem / L1 storage = **56 MB context+"shared context" storage !**
- L2 cache size on A100: 40 MB
- 8,912 FP32 cores (64 FP32 cores per SM) *[A100: 6,912]*
- 262,144 max threads in flight (max warps / SM = 64) *[A100: 221,184]*

Turing vs. Ampere GA102

Graphics Card	GeForce RTX 2080 Founders Edition	GeForce RTX 2080 Super Founders Edition	GeForce RTX 3080 10 GB Founders Edition
GPU Codename	TU104	TU104	GA102
GPU Architecture	NVIDIA Turing	NVIDIA Turing	NVIDIA Ampere
GPCs	6	6	6
TPCs	23	24	34
SMs	46	48	68
CUDA Cores / SM	64	64	128
CUDA Cores / GPU	2944	3072	8704
Tensor Cores / SM	8 (2nd Gen)	8 (2nd Gen)	4 (3rd Gen)
Tensor Cores / GPU	368	384 (2nd Gen)	272 (3rd Gen)
RT Cores	46 (1st Gen)	48 (1st Gen)	68 (2nd Gen)
GPU Boost Clock (MHz)	1800	1815	1710
Peak FP32 TFLOPS (non-Tensor)[1]	10.6	11.2	29.8
Peak FP16 TFLOPS (non-Tensor)[1]	21.2	22.3	29.8
Peak BF16 TFLOPS (non-Tensor)[1]	NA	NA	29.8
Peak INT32 TOPS (non-Tensor)[1,3]	10.6	11.2	14.9

Turing vs. Ampere GA102

Peak FP16 Tensor TFLOPS with FP16 Accumulate[1]	84.8	89.2	119/238[2]
Peak FP16 Tensor TFLOPS with FP32 Accumulate[1]	42.4	44.6	59.5/119[2]
Peak BF16 Tensor TFLOPS with FP32 Accumulate[1]	NA	NA	59.5/119[2]
Peak TF32 Tensor TFLOPS[1]	NA	NA	29.8/59.5[2]
Peak INT8 Tensor TOPS[1]	169.6	178.4	238/476[2]
Peak INT4 Tensor TOPS[1]	339.1	356.8	476/952[2]
Frame Buffer Memory Size and Type	8192 MB GDDR6	8192 MB GDDR6	10240 MB GDDR6X
Memory Interface	256-bit	256-bit	320-bit
Memory Clock (Data Rate)	14 Gbps	15.5 Gbps	19 Gbps
Memory Bandwidth	448 GB/sec	496 GB/sec	760 GB/sec
ROPs	64	64	96
Pixel Fill-rate (Gigapixels/sec)	115.2	116.2	164.2
Texture Units	184	192	272
Texel Fill-rate (Gigatexels/sec)	331.2	348.5	465
L1 Data Cache/Shared Memory	4416 KB	4608 KB	8704 KB

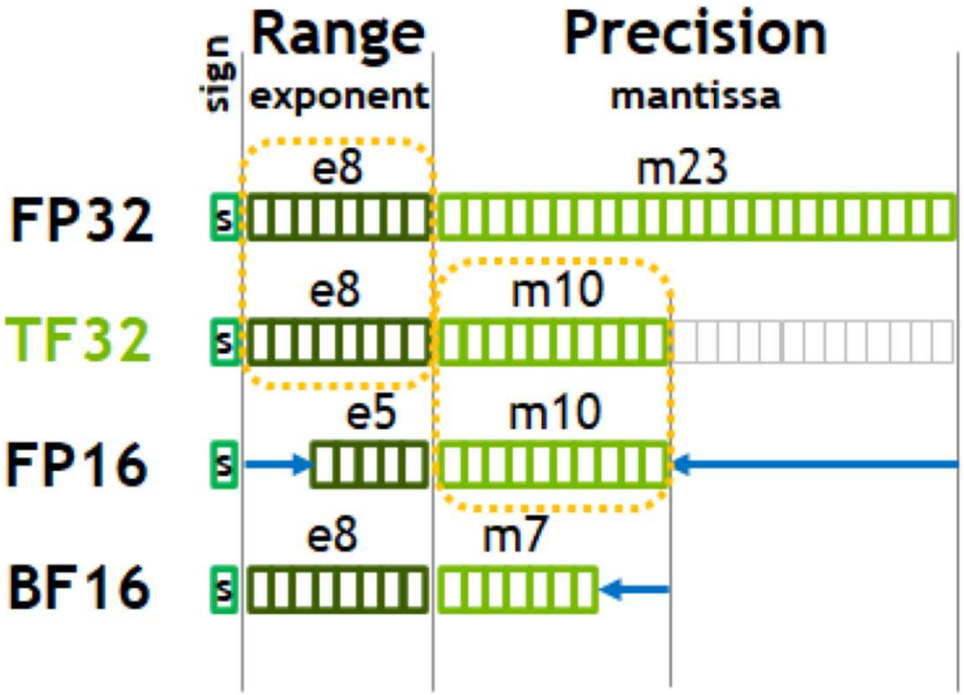
Turing vs. Ampere GA102

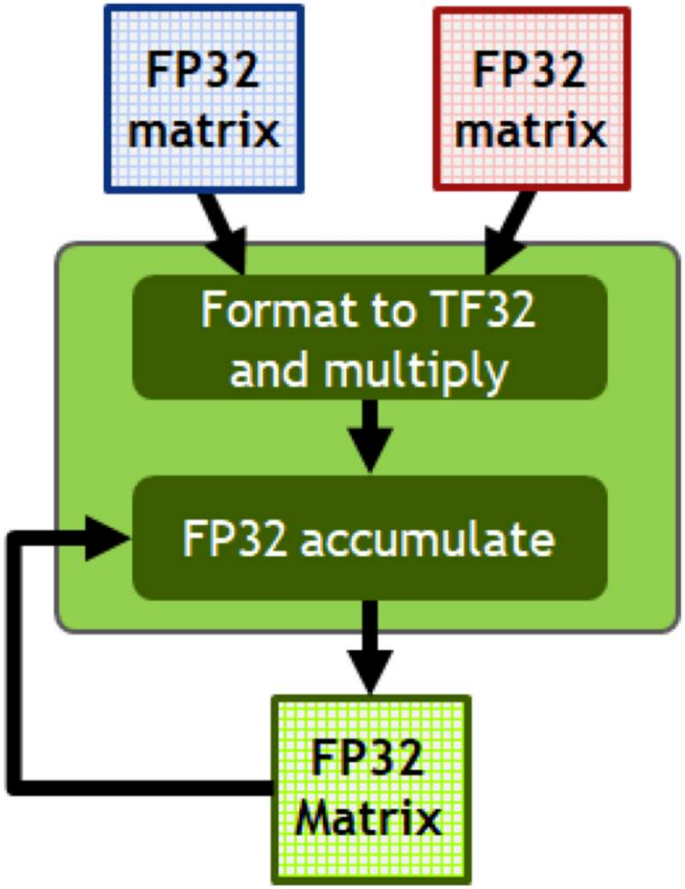
L2 Cache Size	4096 KB	4096 KB	5120 KB
Register File Size	11776 KB	12288 KB	17408 KB
TGP (Total Graphics Power)	225 W	250 W	320W
Transistor Count	13.6 Billion	13.6 Billion	28.3 Billion
Die Size	545 mm^2	545 mm^2	628.4 mm^2
Manufacturing Process	TSMC 12 nm FFN (FinFET NVIDIA)	TSMC 12 nm FFN (FinFET NVIDIA)	Samsung 8 nm 8N NVIDIA Custom Process

1. Peak rates are based on GPU Boost Clock.
2. Effective TOPS / TFLOPS using the new Sparsity Feature
3. TOPS = IMAD-based integer math

Tensor Cores: Many Mixed Precision Options

New in Ampere: TF32, BF16, FP64

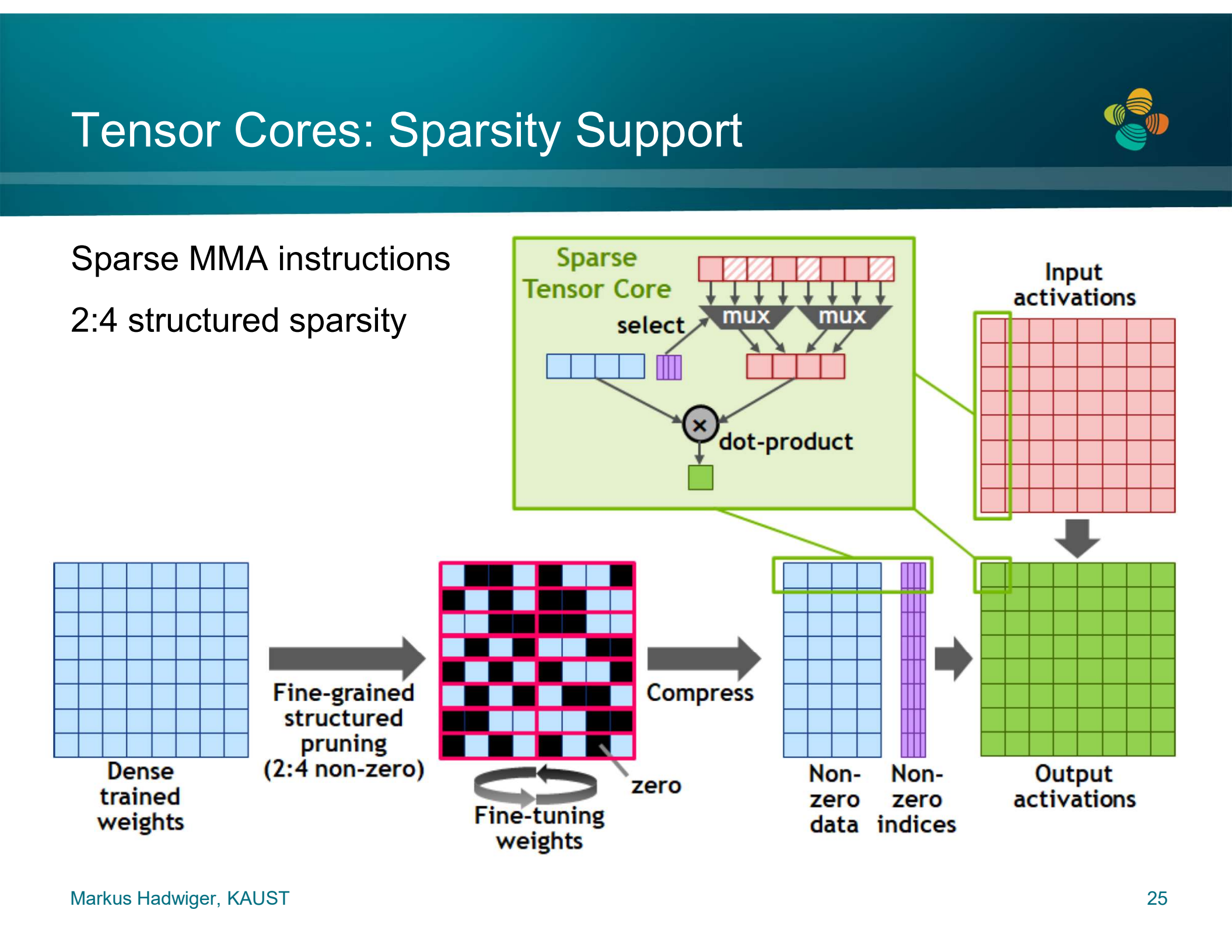




plus FP64 (new in Ampere; GA100 only)

plus INT4/INT8/binary data types (already introduced in Turing)

Tensor Cores: Sparsity Support



Sparse MMA instructions

2:4 structured sparsity

NVIDIA Hopper Architecture

2022

(compute capability 9.0)

GH100 (cc 9.0), ... (H100, ...)

NVIDIA Hopper GH100 Architecture (2022)

GH 100 (H100 Tensor Core GPU) Full GPU: 144 SMs (in 8 GPCs/72 TPCs)

Instruction Throughput

Instruction throughput numbers in CUDA C Programming Guide (Chapter 8.4)

	Compute Capability										
	3.5, 3.7	5.0, 5.2	5.3	6.0	6.1	6.2	7.x	8.0	8.6	8.9	9.0
16-bit floating-point add, multiply, multiply-add	N/A		256	128	2	256	128	256 128 for __nv_bfloat16		128	256 128 for __nv_bfloat16
32-bit floating-point add, multiply, multiply-add	192	128		64	128		64			128	
64-bit floating-point add, multiply, multiply-add	64 8 for GeForce GPUs, except for Titan GPUs	4		32	4		32 2 for compute capability 7.5 GPUs	32	2	2	64

ALU Instruction Latencies and Instructs. / SM

CC	2.0 (Fermi)	2.1 (Fermi)	3.x (Kepler)	5.x (Maxwell)	6.0 (Pascal)	6.1/6.2 (Pascal)	7.x (Volta, Turing)	8.0/8.6 (Ampere)	8.9/9.0 (Ada/Hopper)
# warp sched. / SM	2	2	4	4	2	4	4	4	4
# ALU dispatch / warp sched.	1 (over 2 clocks)	2 (over 2 clocks)	2	1	1	1	1	1	1
SM busy with # warps + inst	L	2L	8L	4L	2L	4L	4L	4L	4L
inst. pipe latency (L)	22	22	11	9	6	6	4	4	4
SM busy with # warps	22	22 + ILP	44 + ILP	36	12	24	16	16	16

see NVIDIA CUDA C Programming Guides (different versions) performance guidelines/multiprocessor level; compute capabilities

NVIDIA GH100 SM

Multiprocessor: SM (CC 9.0)

- 128 FP32 + 64 INT32 cores
- 64 FP64 cores
- 4x 4th gen tensor cores
- ++ thread block clusters, DPX insts., FP8, TMA

4 partitions inside SM

- 32 FP32 + 16 INT32 cores
- 16 FP64 cores
- 8x LD/ST units; 4 SFUs each
- 1x 4th gen tensor core each
- Each has: warp scheduler, dispatch unit, 16K register file

NVIDIA Hopper GH100 Architecture (2022)

GH 100 (H100) Full GPU: 144 SMs (in 8 GPCs/72 TPCs)

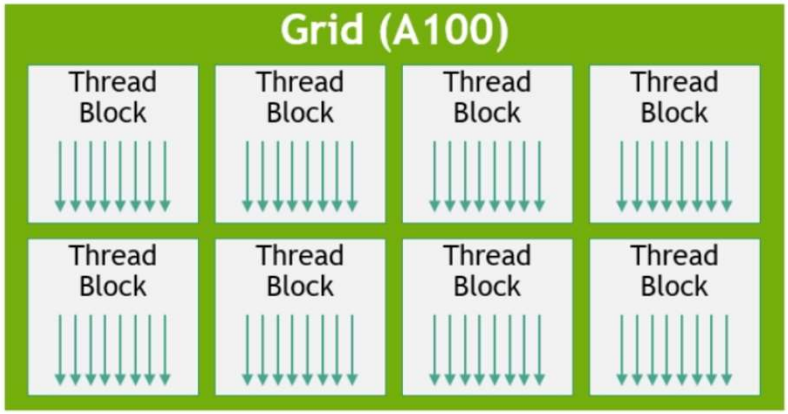
- 64K 32-bit registers / SM = 256 KB register storage per SM
- 256 KB shared memory / L1 per SM

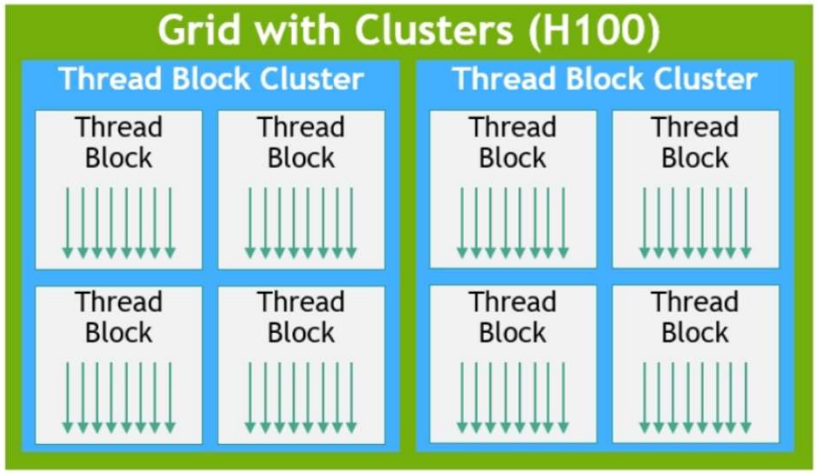
For 144 SMs on full GPU *[SXM5: 132; PCIe: 114]*

- 36 MB register storage, 36 MB shared mem / L1 storage = **72 MB context+"shared context" storage !**
- L2 cache size on H100: 50 MB
- 18,432 FP32 cores (128 FP32 cores per SM) *[SXM5: 16,896]*
- 294,912 max threads in flight (max warps / SM = 64) *[SXM5: 270,336]*

New in CC 9.0: Thread Block Clusters

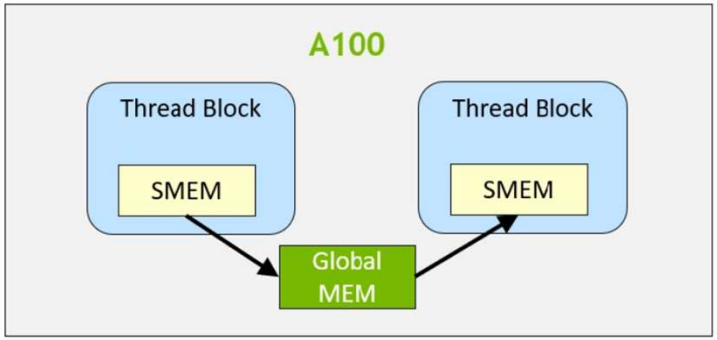
New thread hierarchy level!

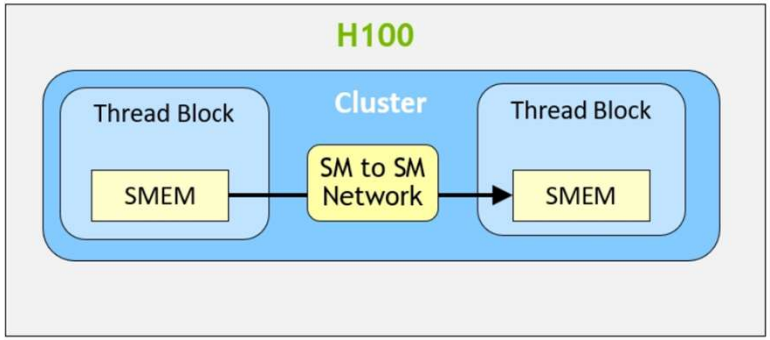




*all threads of a block are on the same **SM !***

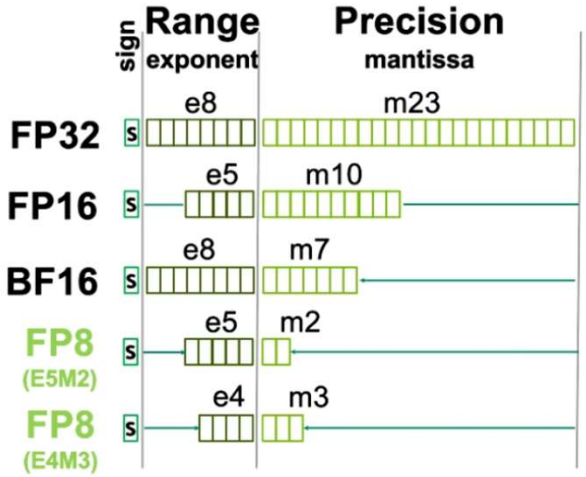
*all blocks of a cluster are on the same **GPC !***



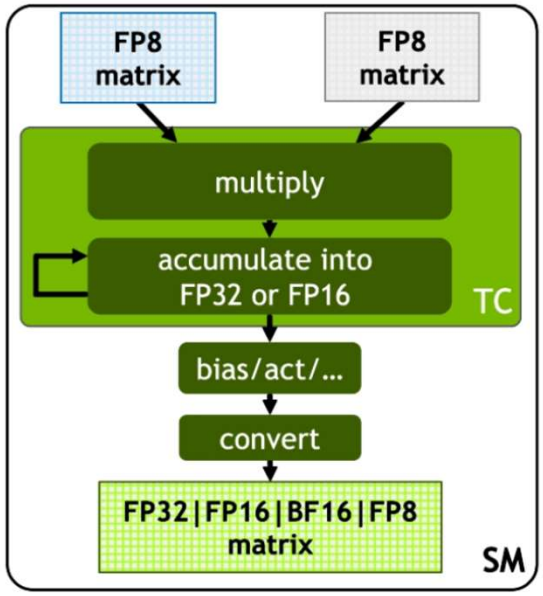


Tensor Cores: More Mixed Precision Options

New in Hopper: FP8



Allocate 1 bit to either range or precision



Support for multiple accumulator and output types

plus other data types from before (INT4/INT8/binary, …)

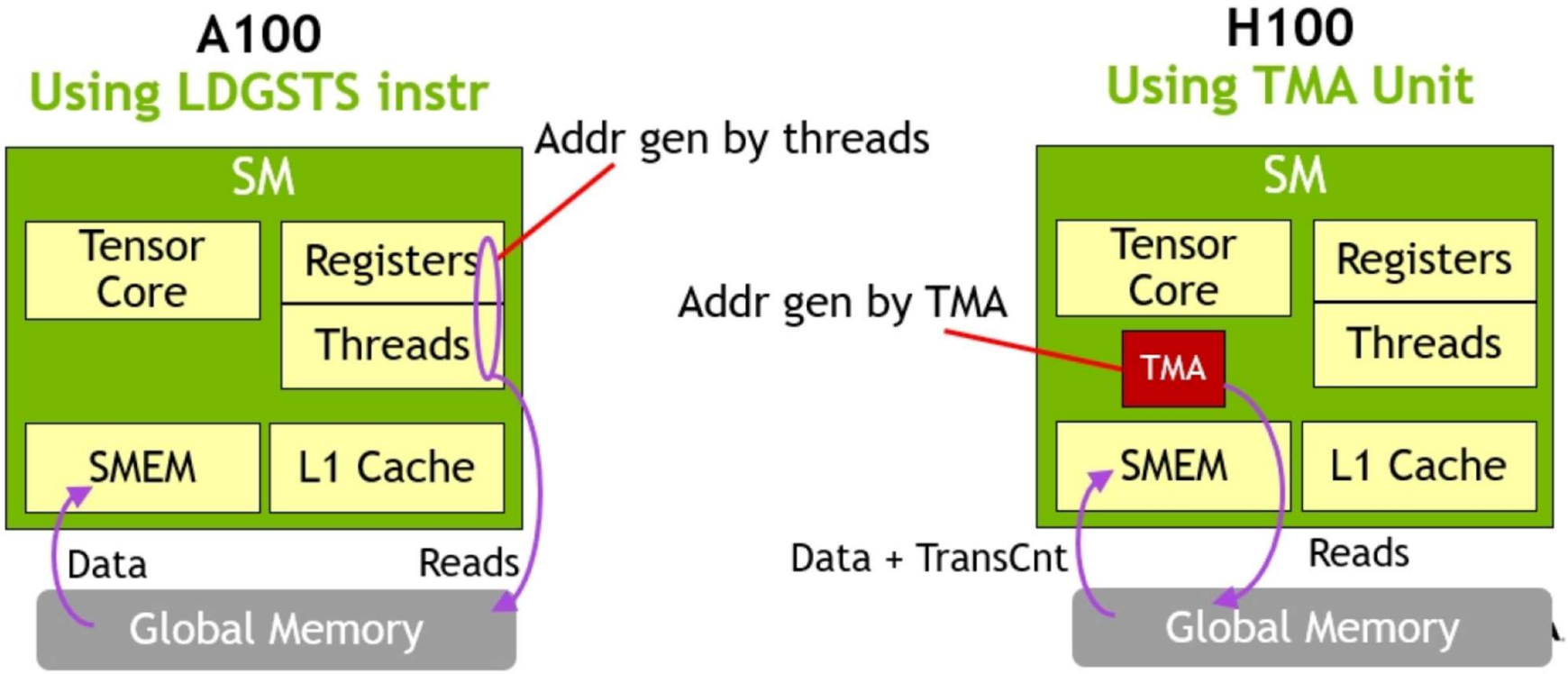
Tensor Cores: Hopper vs. Ampere

(preliminary)

	A100	A100 Sparse	H100 SXM5[1]	H100 SXM5[1] Sparse	H100 SXM5[1] Speedup vs A100
FP8 Tensor Core	NA	NA	2000 TFLOPS	4000 TFLOPS	6.4x vs A100 FP16
FP16	78 TFLOPS	NA	120 TFLOPS	NA	1.5x
FP16 Tensor Core	312 TFLOPS	624 TFLOPS	1000 TFLOPS	2000 TFLOPS	3.2x
BF16 Tensor Core	312 TFLOPS	624 TFLOPS	1000 TFLOPS	2000 TFLOPS	3.2x
FP32	19.5 TFLOPS	NA	60 TFLOPS	NA	3.1x
TF32 Tensor Core	156 TFLOPS	312 TFLOPS	500 TFLOPS	1000 TFLOPS	3.2x
FP64	9.7 TFLOPS	NA	30 TFLOPS	NA	3.1x
FP64 Tensor Core	19.5 TFLOPS	NA	60 TFLOPS	NA	3.1x
INT8 Tensor Core	624 TOPS	1248 TOPS	2000 TFLOPS	4000 TFLOPS	3.2x

Tensor Memory Accelerator (TMA)

Asynchronous transfers



Hopper vs. Ampere (1)

(preliminary)

GPU Features	NVIDIA A100	NVIDIA H100 SXM5[1]	NVIDIA H100 PCIe[1]
GPU Architecture	NVIDIA Ampere	NVIDIA Hopper	NVIDIA Hopper
GPU Board Form Factor	SXM4	SXM5	PCIe Gen 5
SMs	108	132	114
TPCs	54	66	57
FP32 Cores / SM	64	128	128
FP32 Cores / GPU	6912	16896	14592
FP64 Cores / SM (excl. Tensor)	32	64	64
FP64 Cores / GPU (excl. Tensor)	3456	8448	7296
INT32 Cores / SM	64	64	64
INT32 Cores / GPU	6912	8448	7296
Tensor Cores / SM	4	4	4
Tensor Cores / GPU	432	528	456
GPU Boost Clock (Not Finalized for H100) [3]	1410 MHz	Not Finalized	Not Finalized

Hopper vs. Ampere (2)

(preliminary)

GPU Features	NVIDIA A100	NVIDIA H100 SXM5[1]	NVIDIA H100 PCIe[1]
Texture Units	432	528	456
Memory Interface	5120-bit HBM2	5120-bit HBM3	5120-bit HBM2e
Memory Size	40 GB	80 GB	80 GB
Memory Data Rate[1]	1215 MHz DDR	Not Finalized	Not Finalized
Memory Bandwidth (Not Finalized for H100)[1]	1555 GB/sec	3000 GB/sec	2000 GB/sec
L2 Cache Size	40 MB	50 MB	50 MB
Shared Memory Size / SM	Configurable up to 164 KB	Configurable up to 228 KB	Configurable up to 228 KB
Register File Size / SM	256 KB	256 KB	256 KB
Register File Size / GPU	27648 KB	33792 KB	29184 KB
TDP[1]	400 Watts	700 Watts	350 Watts
Transistors	54.2 billion	80 billion	80 billion
GPU Die Size	826 mm2	814 mm2	814 mm2
TSMC Manufacturing Process	7 nm N7	4N customized for NVIDIA	4N customized for NVIDIA

Compute Capabilities

Data Center GPU	NVIDIA Tesla V100	NVIDIA A100	NVIDIA H100
GPU Architecture	NVIDIA Volta	NVIDIA Ampere	NVIDIA Hopper
Compute Capability	7.0	8.0	9.0
Threads / Warp	32	32	32
Max Warps / SM	64	64	64
Max Threads / SM	2048	2048	2048
Max Thread Blocks (CTAs) / SM	32	32	32
Max Thread Blocks / Thread Block Clusters	NA	NA	16
Max 32-bit Registers / SM	65536	65536	65536
Max Registers / Thread Block (CTA)	65536	65536	65536
Max Registers / Thread	255	255	255
Max Thread Block Size (# of threads)	1024	1024	1024
FP32 Cores / SM	64	64	128
Ratio of SM Registers to FP32 Cores	1024	1024	512
Shared Memory Size / SM	Configurable up to 96 KB	Configurable up to 164 KB	Configurable up to 228 KB

NVIDIA Ada Lovelace Architecture

2022/2023

(compute capability 8.9)

GA10x (cc 8.9), ... (RTX 4080 12 GB, RTX 4080 16GB,
(x=2,3,4,6,7) RTX 4090, RTX 6000, L40, ...)

NVIDIA Ada Lovelace AD10x Architecture (2022)

Full AD 10x

Full GPU: 144 SMs (in 12 GPCs/72 TPCs)

NVIDIA Ada Lovelace AD102 Architecture (2022)

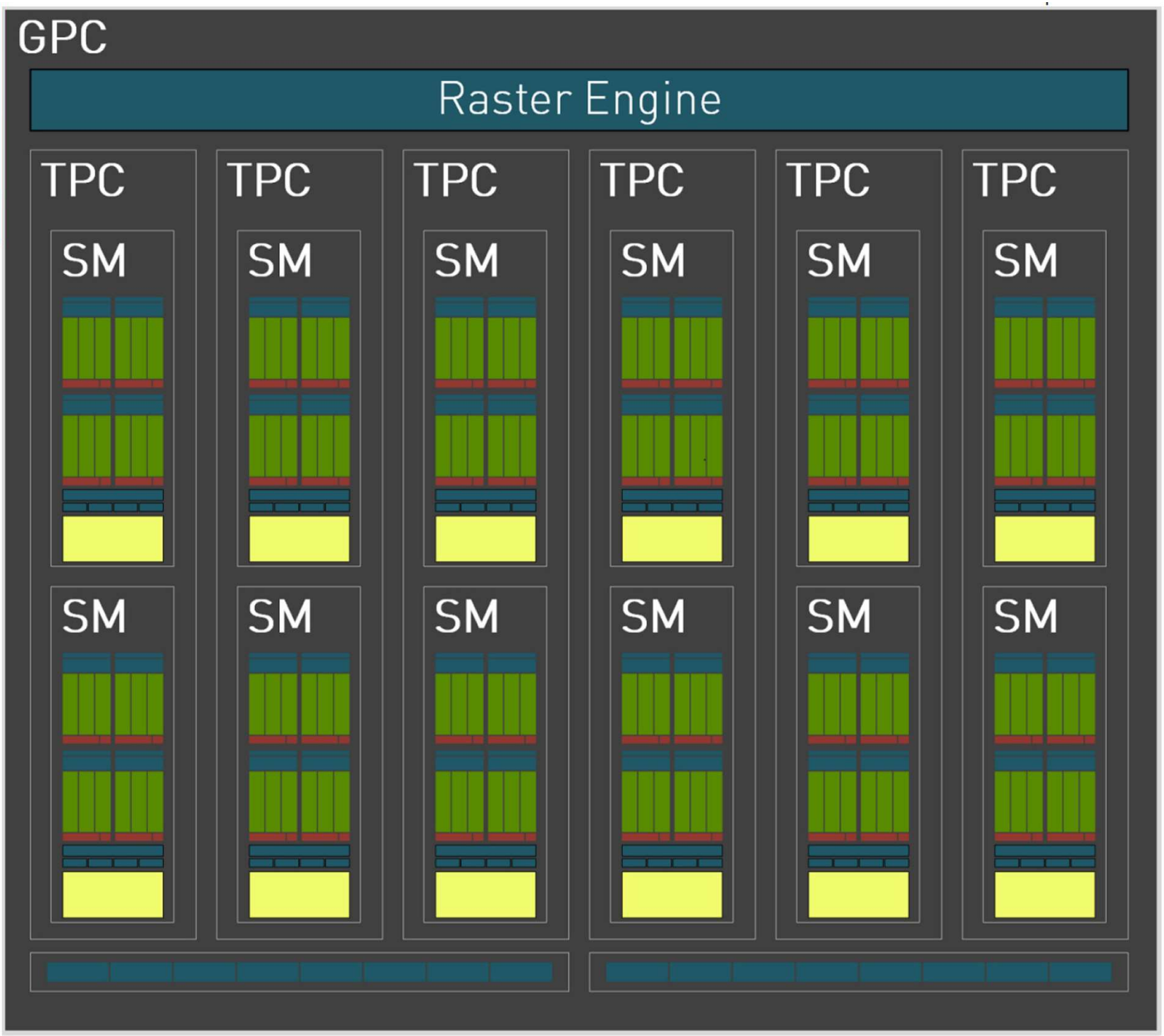
AD 102 (RTX 4090, …)

Full RTX 4090: 128 SMs (in 11 GPCs/64 TPCs)

GPC

Full GPC

- 6 TPCs
- 12 SMs
- 16 ROPs



Instruction Throughput

Instruction throughput numbers in CUDA C Programming Guide (Chapter 8.4)

	Compute Capability										
	3.5, 3.7	5.0, 5.2	5.3	6.0	6.1	6.2	7.x	8.0	8.6	8.9	9.0
16-bit floating-point add, multiply, multiply-add	N/A		256	128	2	256	128	256		128	256
32-bit floating-point add, multiply, multiply-add	192	128		64	128		64			128	
64-bit floating-point add, multiply, multiply-add	64	4		32	4		32	32	2	2	64

128 for __nv_bfloat16

128 for __nv_bfloat16

8 for GeForce GPUs, except for Titan GPUs

2 for compute capability 7.5 GPUs

ALU Instruction Latencies and Instructs. / SM

CC	2.0 (Fermi)	2.1 (Fermi)	3.x (Kepler)	5.x (Maxwell)	6.0 (Pascal)	6.1/6.2 (Pascal)	7.x (Volta, Turing)	8.0/8.6 (Ampere)	8.9/9.0 (Ada/Hopper)
# warp sched. / SM	2	2	4	4	2	4	4	4	4
# ALU dispatch / warp sched.	1 (over 2 clocks)	2 (over 2 clocks)	2	1	1	1	1	1	1
SM busy with # warps + inst	L	2L	8L	4L	2L	4L	4L	4L	4L
inst. pipe latency (L)	22	22	11	9	6	6	4	4	4
SM busy with # warps	22	22 + ILP	44 + ILP	36	12	24	16	16	16

see NVIDIA CUDA C Programming Guides (different versions) performance guidelines/multiprocessor level; compute capabilities

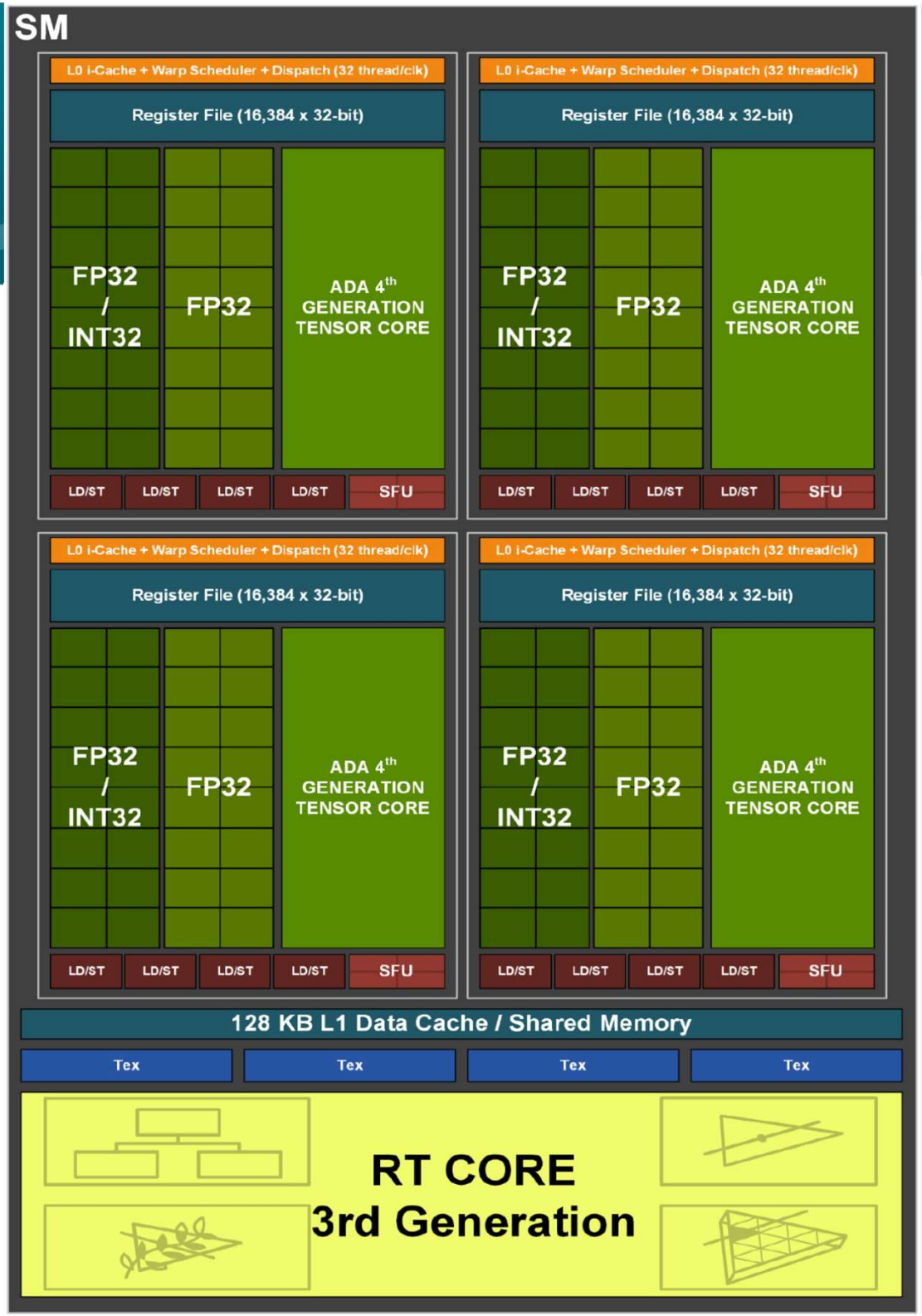
NVIDIA AD102 SM

Multiprocessor: SM (CC 8.9)

- 128 (64+64) FP32 + 64 INT32 cores
- 2 (!) FP64 cores (not in diagram)
- 4x 4th gen tensor cores
- 1x 3rd gen RT (ray tracing) core
- ++ thread block clusters, FP8, … (?)

4 partitions inside SM

- 32 (16+16) FP32 + 16 INT32 cores
- 4x LD/ST units; 4 SFUs each
- 1x 4th gen tensor core each
- Each has: warp scheduler, dispatch unit, 16K register file



NVIDIA Ada Lovelace AD10x Architecture (2022)

AD 10x / AD 102 (RTX 4090) Full GPU: 144 SMs (in 12 GPCs/72 TPCs)

- 64K 32-bit registers / SM = 256 KB register storage per SM
- 128 KB shared memory / L1 per SM

For 144 SMs on full GPU *[RTX 4090: 128; RTX 4080 16GB: 76; RTX 4080 12GB: 60]*

- 36 MB register storage, 18 MB shared mem / L1 storage =
  54 MB context+"shared context" storage !
- L2 cache size on RTX 4090: 72 MB
- 18,432 FP32 cores (128 FP32 cores per SM) *[RTX 4090: 16,384]*
- 294,912 max threads in flight (max warps / SM = 64) *[RTX 4090: 262,144]*

Comparisons

RTX GPUs

Graphics Card	GeForce RTX 2080 Ti	GeForce RTX 3090 Ti	GeForce RTX 4090
CUDA Cores	4352	10752	16384
GPCs	6	7	11
TPCs	34	42	64
SMs	68	84	128
GPU Boost Clock (MHz)	1635	1860	2520
FP32 TFLOPS	14.2	40	82.6
Tensor Cores	544 (2nd Gen)	336 (3rd Gen)	512 (4th Gen)
Tensor TFLOPS (FP8)	N/A	N/A	660.6/1321.2[1]
RT Cores	68 (1st Gen)	84 (2nd Gen)	128 (3rd Gen)
RT TFLOPS	42.9	78.1	191
Texture Units	272	336	512
Texture Fill Rate	444.7	625	1290.2
ROPS	88	112	176
Pixel Fill Rate	143.9	208.3	443.5
Memory Size and Type	11 GB GDDR6	24 GB GDDR6X	24 GB GDDR6X
Memory Clock (Data Rate)	14 Gbps	21 Gbps	21 Gbps
Memory Bandwidth	616 GB/sec	1008 GB/sec	1008 GB/sec

Comparisons

RTX GPUs

Graphics Card	GeForce RTX 2080 Ti	GeForce RTX 3090 Ti	GeForce RTX 4090
L1 Cache/Shared Memory	6528 KB	10752 KB	16384 KB
L2 Cache	5632 KB	6144 KB	73728 KB
TGP	260 W	450 W	450 W
Transistor Count	18.6 Billion	28.3 Billion	76.3 Billion
Die Size	754 mm²	628.4 mm²	608.5 mm²
Manufacturing Process	TSMC 12 nm FFN (FinFET NVIDIA)	Samsung 8 nm 8N NVIDIA Custom Process	TSMC 4N NVIDIA Custom Process

1- Using Sparsity feature

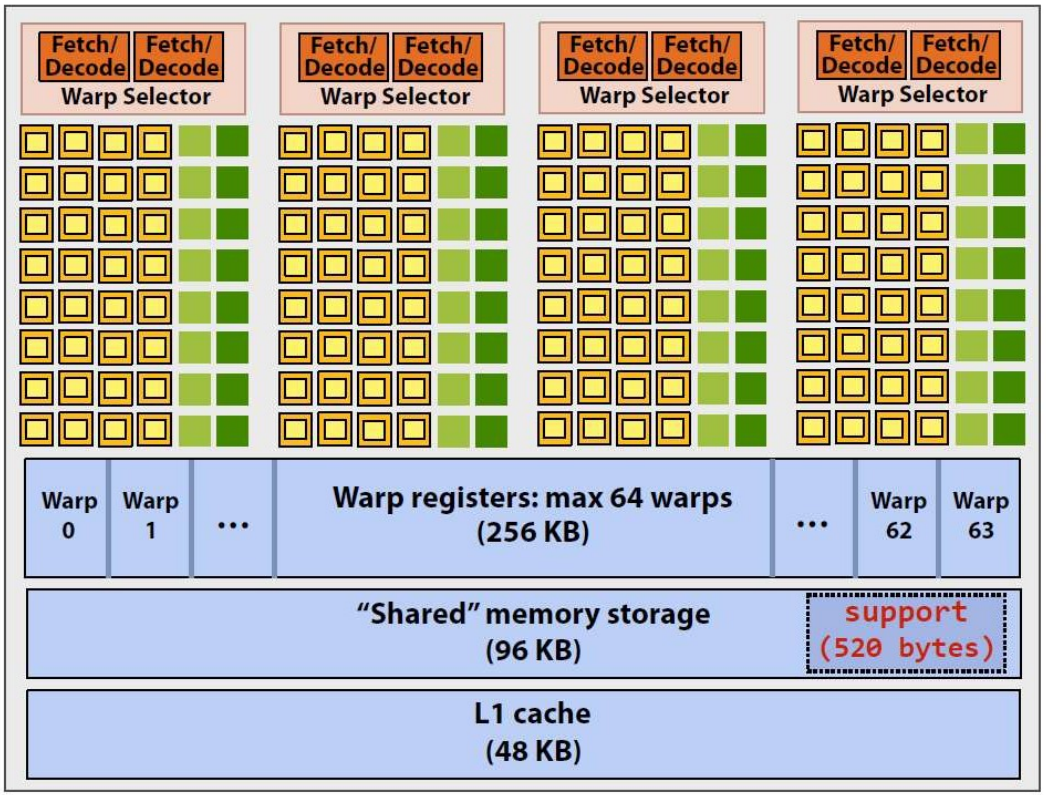
GPU Compute APIs

NVIDIA CUDA

- Old acronym: “Compute Unified Device Architecture”
- Extensions to C(++) programming language
- **`__host__`**, **`__global__`**, and **`__device__`** functions
- Heavily multi-threaded
- Synchronize threads with **`__syncthreads`**(), ...
- Atomic functions
  (before compute capability 2.0 only integer, from 2.0 on also float)

- Compile **`.cu`** files with NVCC
- Uses general C compiler (Visual C, gcc, ...)
- Link with CUDA run-time (**`cudart.lib`**) and cuda core (**`cuda.lib`**)

Teaser: Simple Typical CUDA Kernel (SM Perspective)




```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input,
                         float* output)
{
   __shared__ float support[THREADS_PER_BLK+2];
   int index = blockIdx.x * blockDim.x +
               threadIdx.x;

   support[threadIdx.x] = input[index];
   if (threadIdx.x < 2) {
      support[THREADS_PER_BLK+threadIdx.x]
        = input[index+THREADS_PER_BLK];
   }

   __syncthreads();

   float result = 0.0f;  // thread-local
   for (int i=0; i<3; i++)
     result += support[threadIdx.x + i];

   output[index] = result;
}
```

Recall, CUDA kernels execute as SPMD programs

On NVIDIA GPUs groups of 32 CUDA threads share an instruction stream. These groups called "warps".

A `convolve` thread block is executed by 4 warps (4 warps x 32 threads/warp = 128 CUDA threads per block)

(Warps are an important GPU implementation detail, but not a CUDA abstraction!)

SM core operation each clock:

- **Select up to four runnable warps from 64 resident on SM core (thread-level parallelism)**
- **Select up to two runnable instructions per warp (instruction-level parallelism) ***

CUDA Software Development

CUDA Optimized Libraries:
math.h, FFT, BLAS, …

Integrated CPU + GPU
C Source Code

NVIDIA C Compiler

NVIDIA Assembly
for Computing (PTX)

CPU Host Code

CUDA
Driver

Profiler

Standard C Compiler

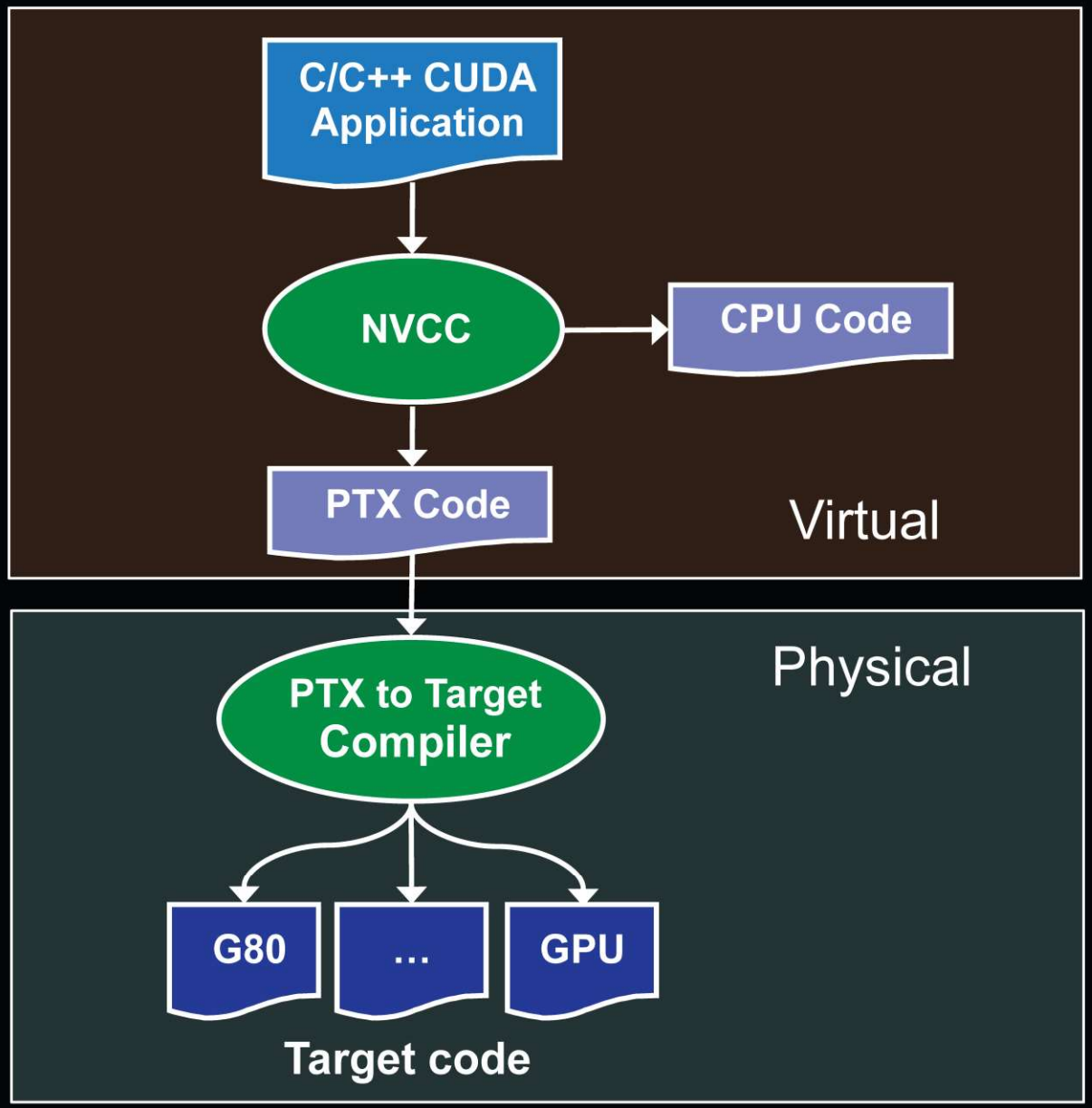
GPU

CPU

NVISION 08
THE WORLD OF VISUAL COMPUTING

NVIDIA

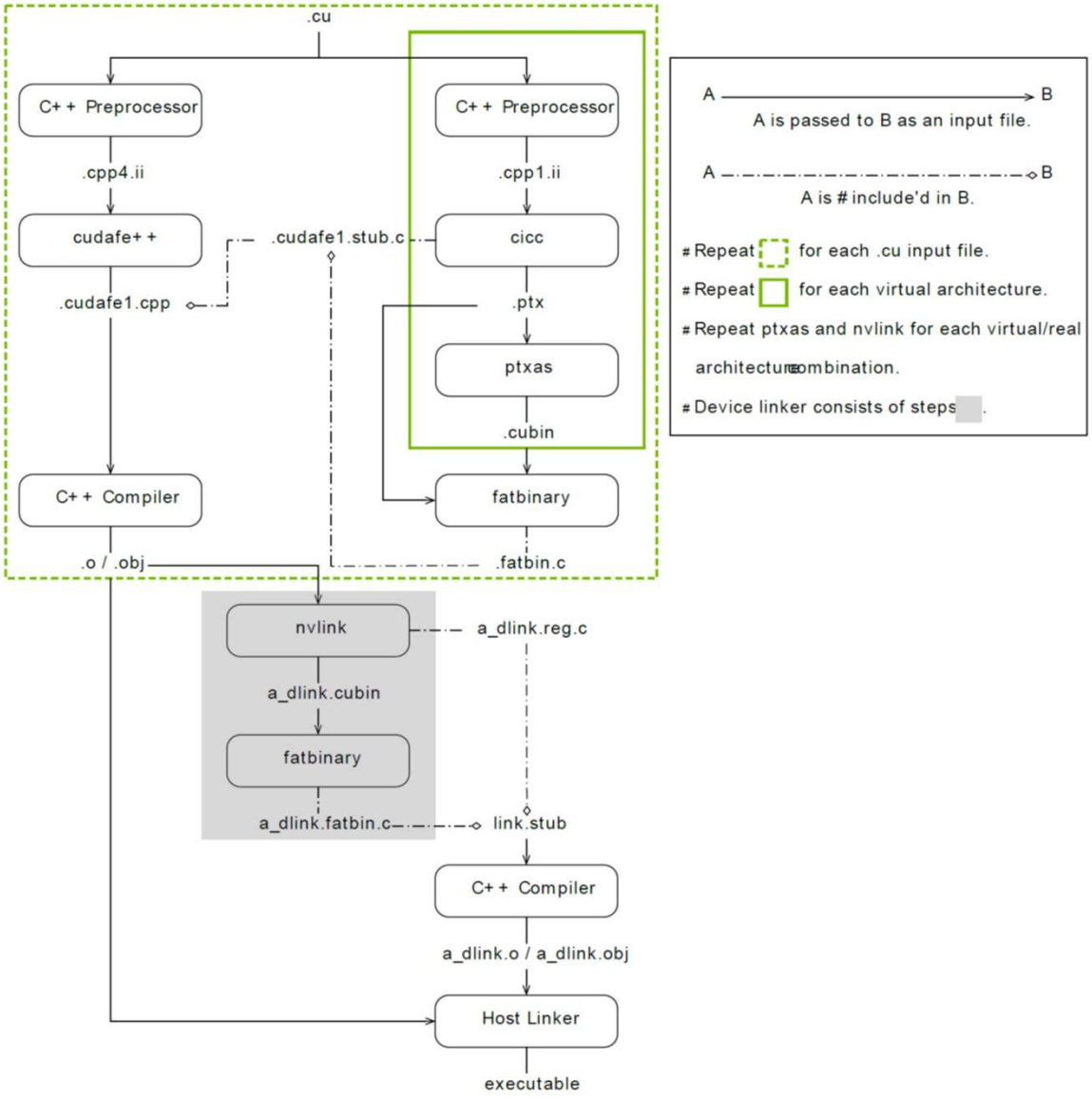
Compiling CUDA Code



CUDA Compilation Trajectory

CUDA Compiler Driver (NVCC) docs:

`CUDA_Compiler_Driver_NVCC.pdf`



CUDA Compilation Trajectory / Code Gen

4.2.7. Options for Steering GPU Code Generation

4.2.7.1. `--gpu-architecture` *arch* (`-arch`)

Specify the name of the class of NVIDIA virtual GPU architecture for which the CUDA input files must be compiled.

With the exception as described for the shorthand below, the architecture specified with this option must be a *virtual* architecture (such as compute_50). Normally, this option alone does not trigger assembly of the generated PTX for a *real* architecture (that is the role of `nvcc` option `--gpu-code`, see below); rather, its purpose is to control preprocessing and compilation of the input to PTX.

For convenience, in case of simple `nvcc` compilations, the following shorthand is supported. If no value for option `--gpu-code` is specified, then the value of this option defaults to the value of `--gpu-architecture`. In this situation, as only exception to the description above, the value specified for `--gpu-architecture` may be a *real* architecture (such as a sm_50), in which case `nvcc` uses the specified *real* architecture and its closest *virtual* architecture as effective architecture values. For example, `nvcc --gpu-architecture=sm_50` is equivalent to `nvcc --gpu-architecture=compute_50 --gpu-code=sm_50,compute_50`.

See Virtual Architecture Feature List for the list of supported *virtual* architectures and GPU Feature List for the list of supported *real* architectures.

from **`https://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf`**

CUDA Compilation Trajectory / Code Gen

4.2.7.2. `--gpu-code code,...` (`-code`)

Specify the name of the NVIDIA GPU to assemble and optimize PTX for.

`nvcc` embeds a compiled code image in the resulting executable for each specified *code* architecture, which is a true binary load image for each *real* architecture (such as sm_50), and PTX code for the *virtual* architecture (such as compute_50).

During runtime, such embedded PTX code is dynamically compiled by the CUDA runtime system if no binary load image is found for the *current* GPU.

Architectures specified for options `--gpu-architecture` and `--gpu-code` may be *virtual* as well as *real*, but the *code* architectures must be compatible with the *arch* architecture. When the `--gpu-code` option is used, the value for the `--gpu-architecture` option must be a *virtual* PTX architecture.

For instance, `--gpu-architecture=compute_60` is not compatible with `--gpu-code=sm_52`, because the earlier compilation stages will assume the availability of `compute_60` features that are not present on `sm_52`.

See Virtual Architecture Feature List for the list of supported *virtual* architectures and GPU Feature List for the list of supported *real* architectures.

Also look at compatibility guides:

https://docs.nvidia.com/cuda/pdf/NVIDIA_Ampere_GPU_Architecture_Compatibility_Guide.pdf

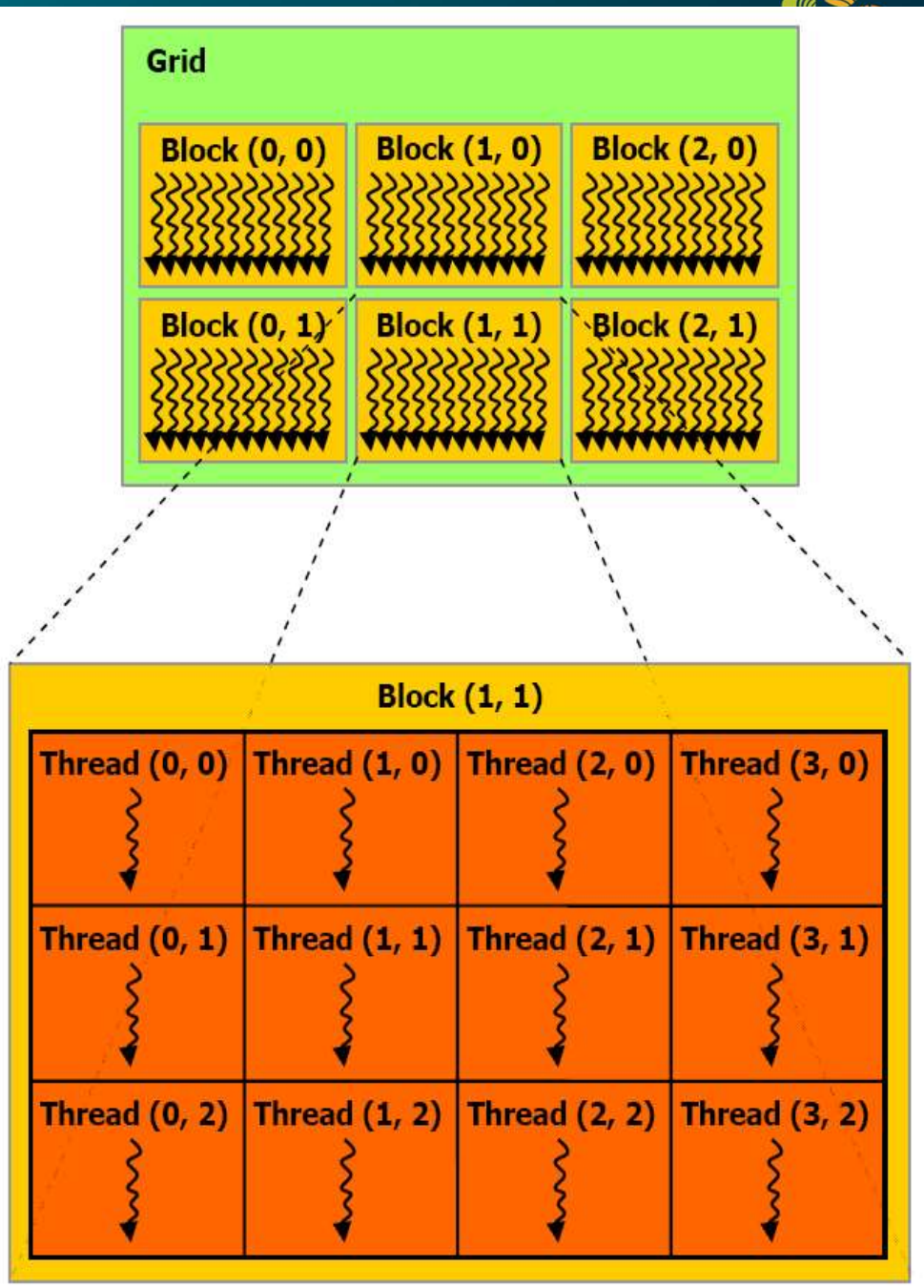
https://docs.nvidia.com/cuda/pdf/Hopper_Compatibility_Guide.pdf

CUDA Multi-Threading

CUDA model groups threads into **thread blocks**; blocks into **grid**

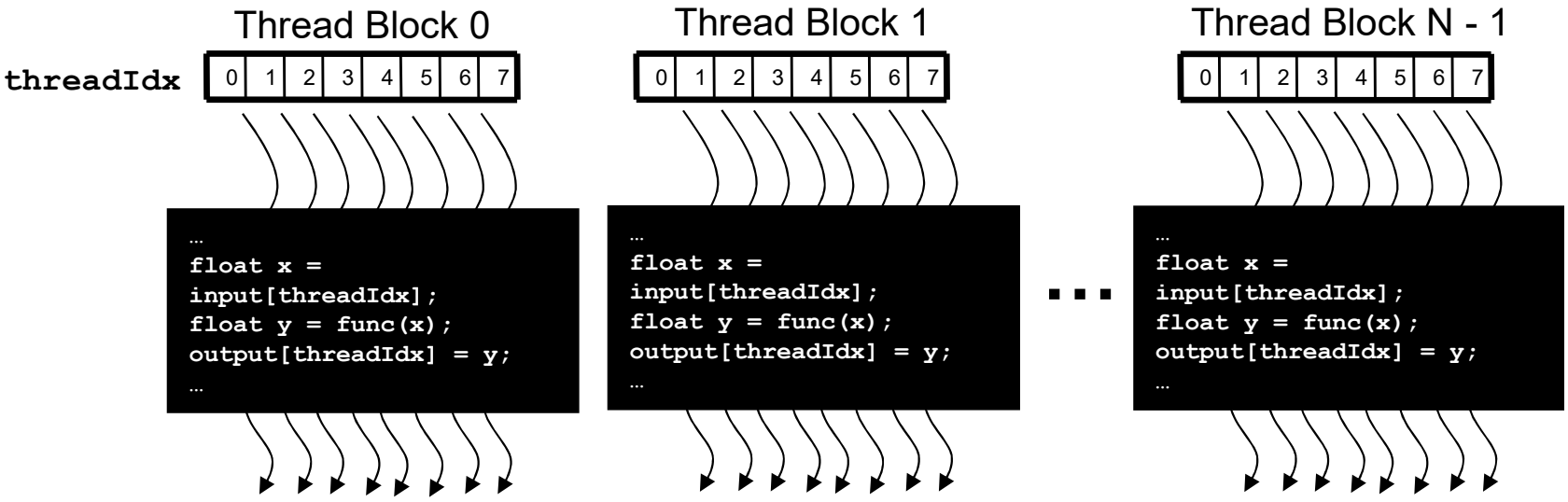
Execution on actual hardware:

- Thread blocks assigned to SM (up to 8, 16, or 32 blocks per SM; depending on compute capability)
- 32 threads grouped into a **warp** (on all compute capabilities)



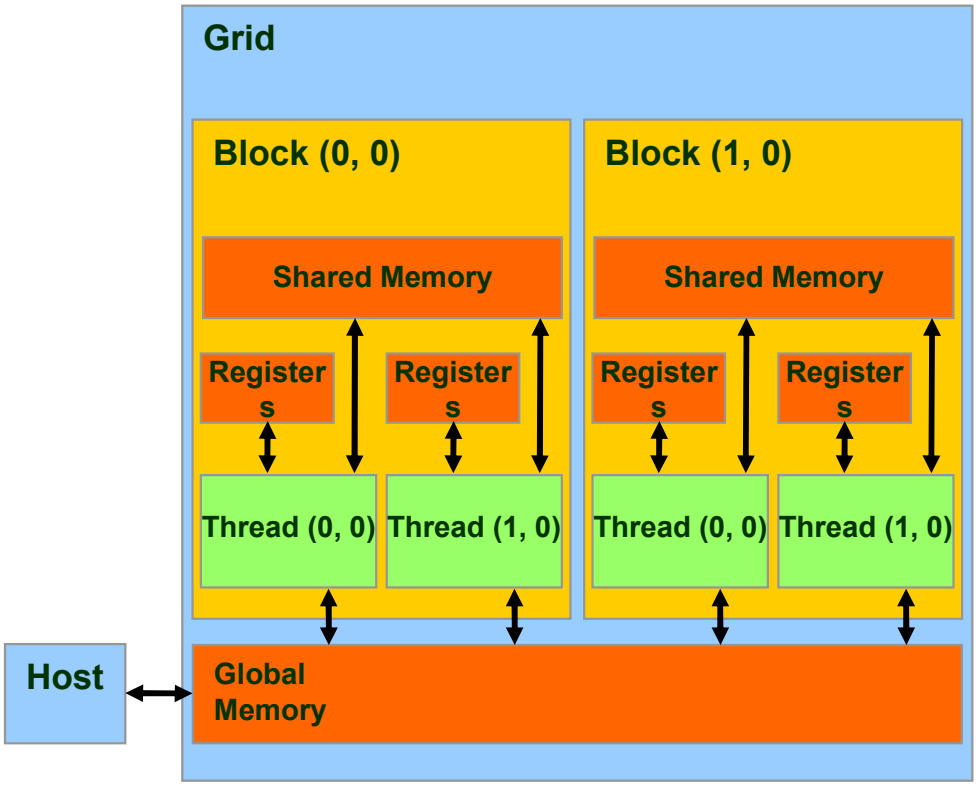
Threads in Block, Blocks in Grid

- Identify work of thread via
  - **`threadIdx`**
  - **`blockIdx`**



CUDA Memory Model and Usage

- **`cudaMalloc()`, `cudaFree()`**
- **`cudaMallocArray()`, `cudaMalloc2DArray()`, `cudaMalloc3DArray()`**
- **`cudaMemcpy()`**
- **`cudaMemcpyArray()`**
- Host ↔ host
  Host ↔ device
  Device ↔ device
- Asynchronous transfers possible (DMA)



CUDA Kernels and Threads

- Parallel portions of an application are executed on the device as kernels
  - One kernel is executed at a time
  - Many threads execute each kernel
- Differences between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Very little creation overhead
    - Instant switching
  - CUDA uses 1000s of threads to achieve efficiency
    - Multi-core CPUs can use only a few

Definitions

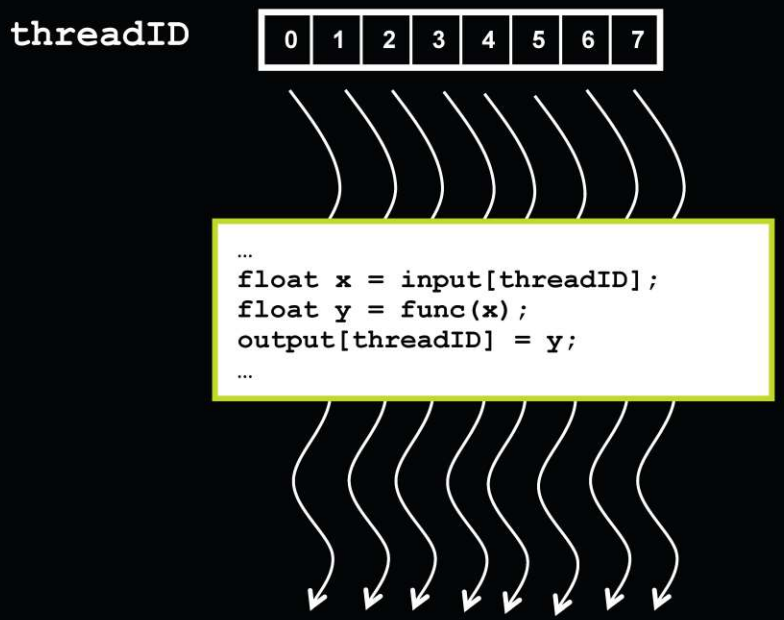
Device = GPU

Host = CPU

Kernel = function that runs on the device

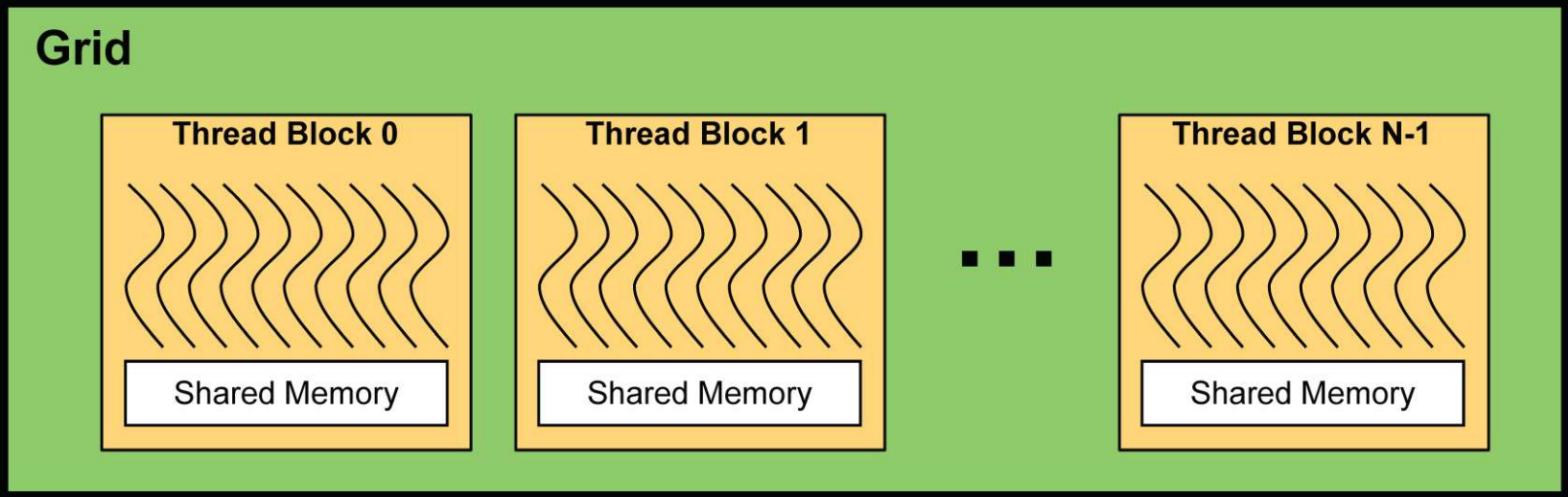
Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code
  - Each thread has an ID that it uses to compute memory addresses and make control decisions



Thread Batching

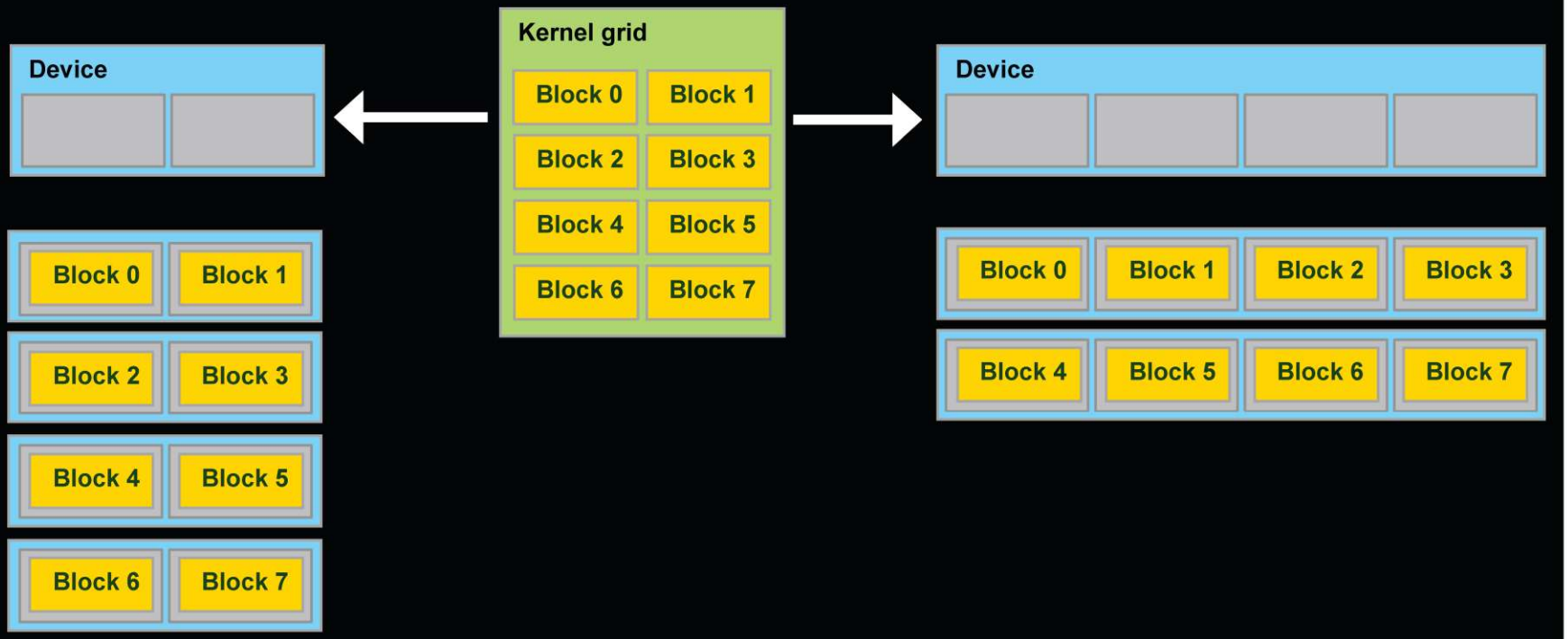
- Kernel launches a grid of thread blocks
  - Threads within a block cooperate via shared memory
  - Threads within a block can synchronize
  - Threads in different blocks cannot cooperate*
- Allows programs to *transparently scale* to different GPUs



* brand new on Hopper: thread block clusters

Transparent Scalability

- Hardware is free to schedule thread blocks on any processor
  - A kernel scales across parallel multiprocessors



Execution Model

Software

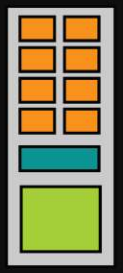
Thread

Hardware

Thread Processor

Threads are executed by thread processors

Thread Block

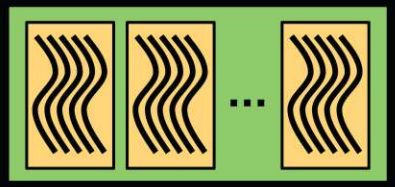


Multiprocessor

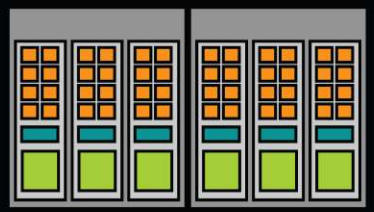
Thread blocks are executed on multiprocessors

Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)



Grid



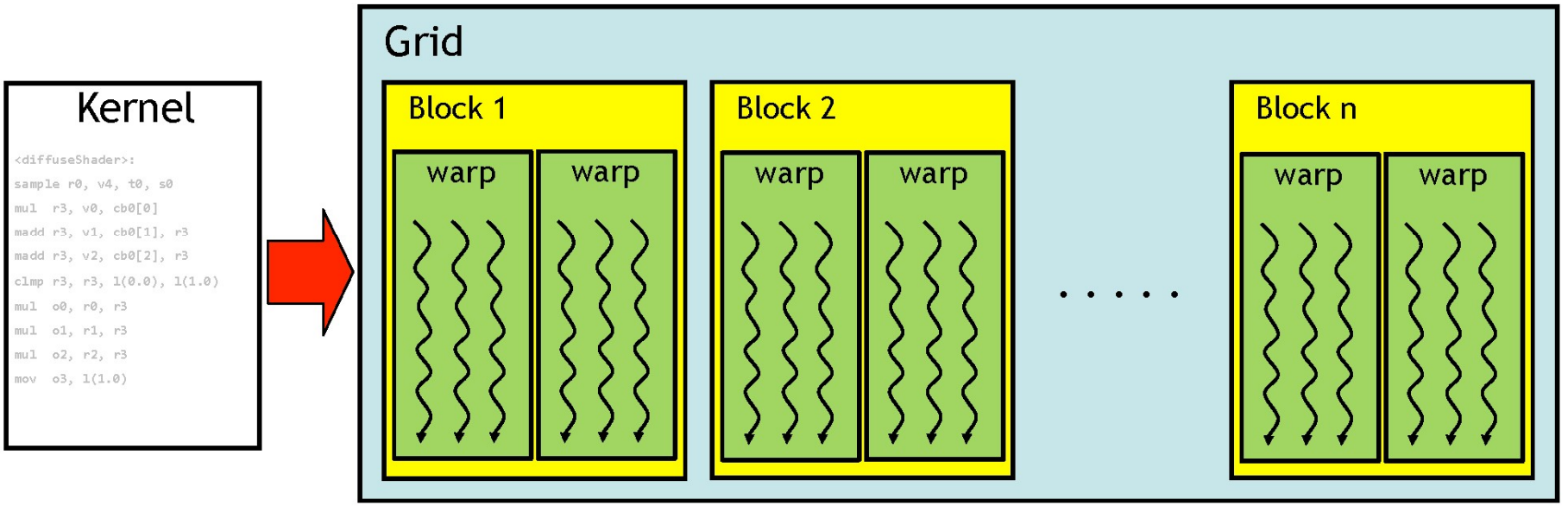
Device

A kernel is launched as a grid of thread blocks

Only one kernel can execute on a device at one time

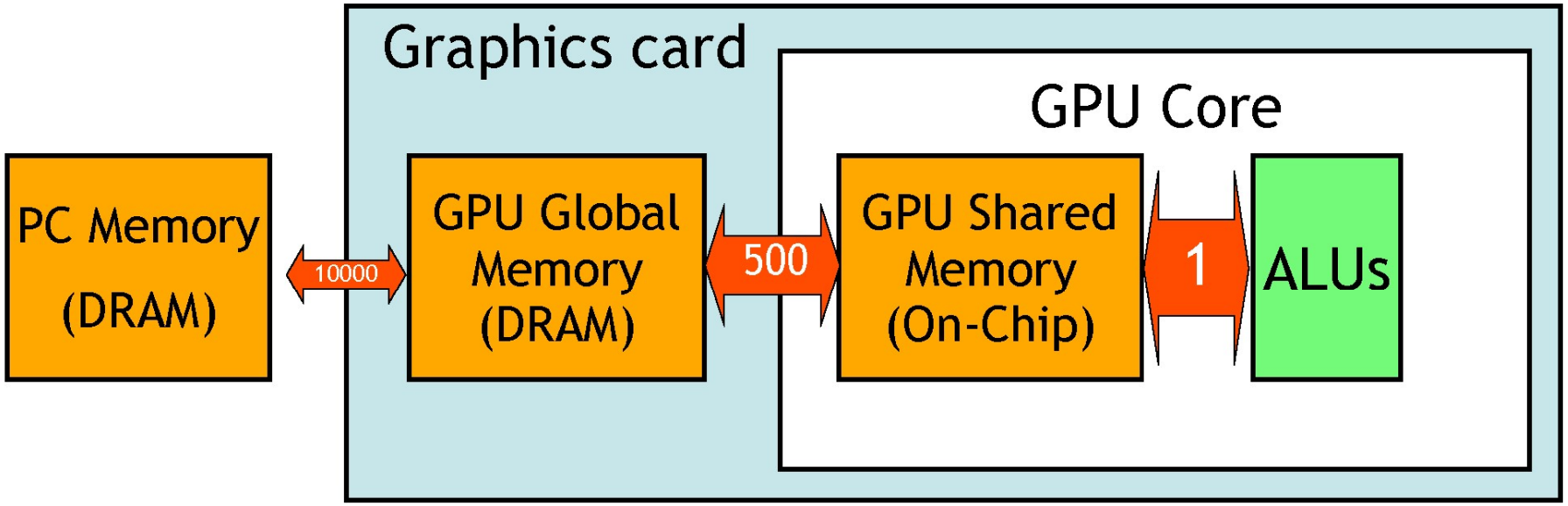
CUDA Programming Model

- Kernel
  - GPU program that runs on a thread grid
- Thread hierarchy
  - Grid : a set of blocks
  - Block : a set of warps
  - Warp : a SIMD group of 32 threads
  - Grid size * block size = total # of threads



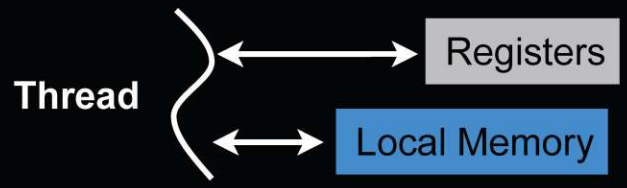
CUDA Memory Structure

- Memory hierarchy
  - PC memory : off-card
  - GPU global : off-chip / on-card
  - GPU shared/register/cache : on-chip
- The host can read/write global memory
- Each thread communicates using shared memory



Kernel Memory Access

- **Per-thread**

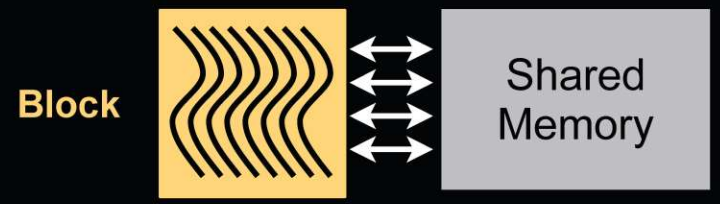


On-chip

Off-chip, ~~uncached~~

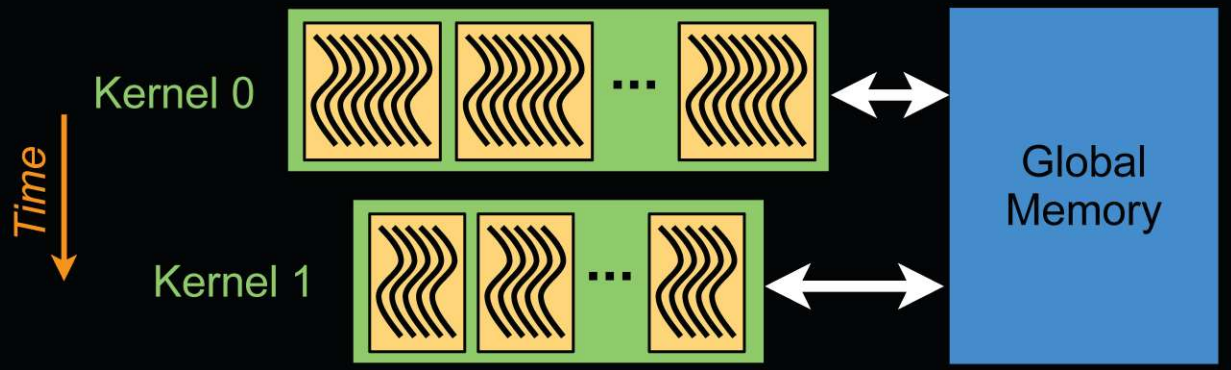
cached on Fermi or newer!

- **Per-block**



- On-chip, small
- Fast

- **Per-device**



- Off-chip, large
- ~~Uncached~~
- Persistent across kernel launches
- Kernel I/O

cached on Fermi or newer!

Memory Architecture

Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	~~No~~* YES	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	~~No~~* YES	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application
Texture	Off-chip	Yes	R	All threads + host	Application

* cached on Fermi or newer!

(Memory) State Spaces

PTX ISA 7.8 (Chapter 5)

Name	Addressable	Initializable	Access	Sharing
`.reg`	No	No	R/W	per-thread
`.sreg`	No	No	RO	per-CTA
`.const`	Yes	Yes[1]	RO	per-grid
`.global`	Yes	Yes[1]	R/W	Context
`.local`	Yes	No	R/W	per-thread
`.param` (as input to kernel)	Yes[2]	No	RO	per-grid
`.param` (used in functions)	Restricted[3]	No	R/W	per-thread
`.shared`	Yes	No	R/W	per-cluster[5]
`.tex`	No[4]	Yes, via driver	RO	Context

Notes:

[1] Variables in `.const` and `.global` state spaces are initialized to zero by default.

[2] Accessible only via the `ld.param` instruction. Address may be taken via `mov` instruction.

[3] Accessible via `ld.param` and `st.param` instructions. Device function input and return parameters may have their address taken via `mov`; the parameter is then located on the stack frame and its address is in the `.local` state space.

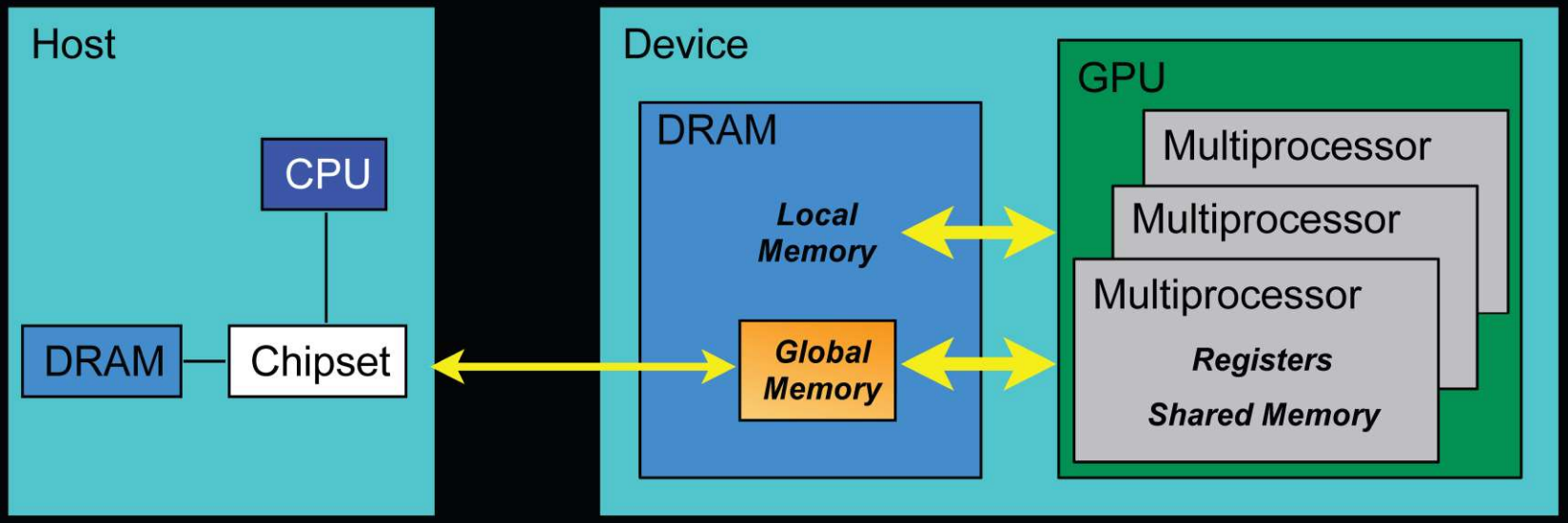
[4] Accessible only via the `tex` instruction.

[5] Visible to the owning CTA and other active CTAs in the cluster.

Managing Memory

Unified memory space can be enabled on Fermi / CUDA 4.x and newer

- CPU and GPU have separate memory spaces
- Host (CPU) code manages device (GPU) memory:
  - Allocate / free
  - Copy data to and from device
  - Applies to *global* device memory (DRAM)



GPU Memory Allocation / Release

- cudaMalloc(void ** pointer, size_t nbytes)
- cudaMemset(void * pointer, int value, size_t count)
- cudaFree(void* pointer)

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int *a_d = 0;
cudaMalloc( (void**)&a_d,  nbytes );
cudaMemset( a_d, 0, nbytes);
cudaFree(a_d);
```

Data Copies

- cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);
  - `direction` specifies locations (host or device) of `src` and `dst`
  - Blocks CPU thread: returns after the copy is complete
  - Doesn't start copying until previous CUDA calls complete
- enum cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

Executing Code on the GPU

- Kernels are C functions with some restrictions
  - Cannot access host memory — except: (*) and (**)
  - Must have void return type
  - No variable number of arguments ("varargs")
  - (Not recursive) — recursion supported on __device__ functions from cc. 2.x (i.e., basically on *all* current GPUs)
  - No static variables
- Function arguments automatically copied from host to device

(*) "unified memory programming" introduced with CUDA 6 (cc. 3.x +): allocate memory with cudaMallocManaged(); uses automatic migration

(**) also: mapped pinned (page-locked) memory ("zero-copy memory") : allocate memory with cudaMallocHost(); beware of low performance!!

Note: UVA ("unified virtual addressing"; cc. 2.x +) is something different!! just pertains to unified pointers (see cudaPointerGetAttributes(), …)

Function Qualifiers

- Kernels designated by function qualifier:
  - __global__
    - Function called from host and executed on device
    - Must return void
- Other CUDA function qualifiers
  - __device__
    - Function called from device and run on device
    - Cannot be called from host code
  - __host__
    - Function called from host and executed on host (default)
    - __host__ and __device__ qualifiers can be combined to generate both CPU and GPU code

Variable Qualifiers (GPU code)

- __device__
  - Stored in global memory (large, high latency, no cache)
  - Allocated with cudaMalloc (__device__ qualifier implied)
  - Accessible by all threads
  - Lifetime: application
- __shared__
  - Stored in on-chip shared memory (very low latency)
  - Specified by execution configuration or at compile time
  - Accessible by all threads in the same thread block
  - Lifetime: thread block
- Unqualified variables:
  - Scalars and built-in vector types are stored in registers
  - What doesn't fit in registers spills to "local" memory

CUDA 6+: __managed__ (with __device__) for managed memory (unified memory programming)

Launching Kernels

- Modified C function call syntax:

```
kernel<<<dim3 dG, dim3 dB>>>(…)
```

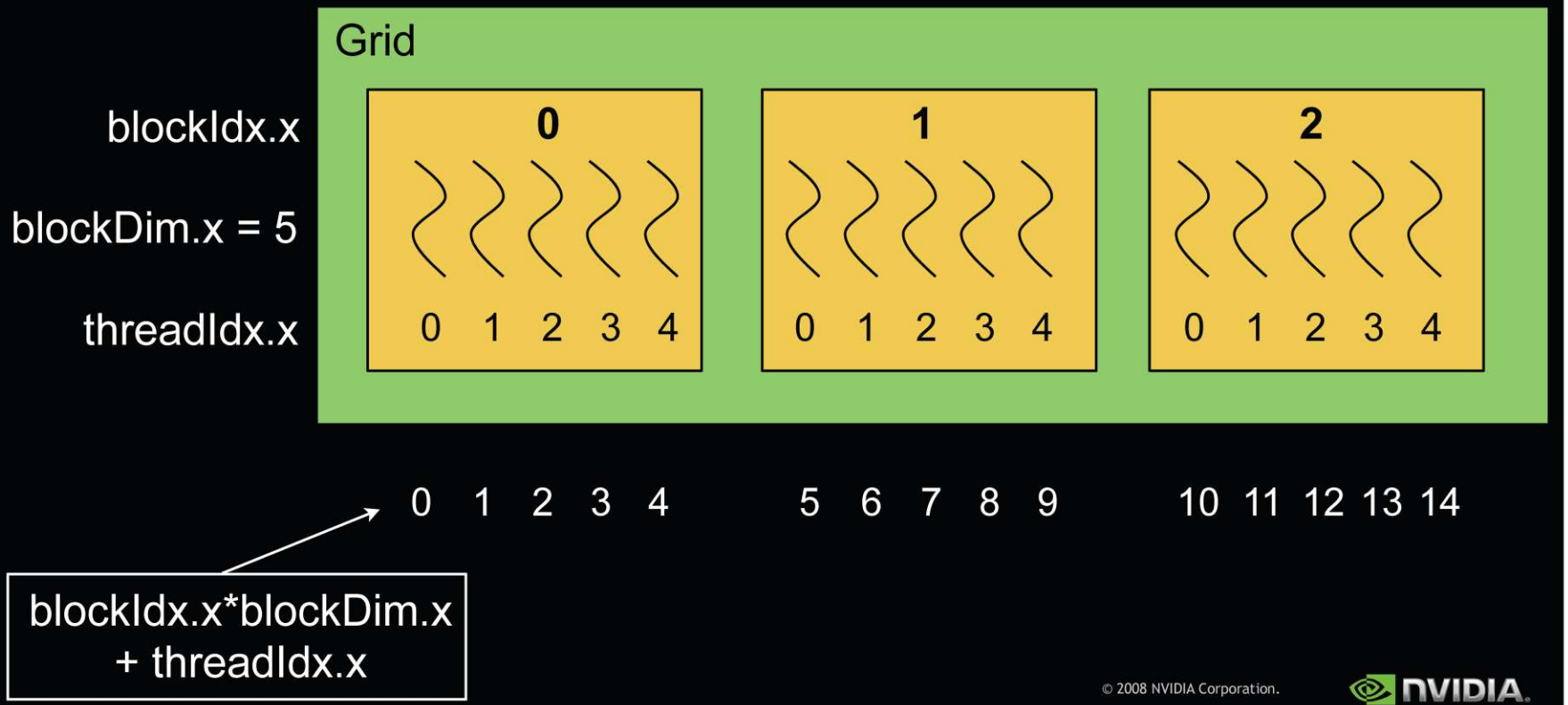
- Execution Configuration ("<<< >>>")
  - dG - dimension and size of grid in blocks
    - Two-dimensional: x and y
    - Blocks launched in the grid: dG.x * dG.y
  - dB - dimension and size of blocks in threads:
    - Three-dimensional: x, y, and z
    - Threads per block: dB.x * dB.y * dB.z
  - Unspecified dim3 fields initialize to 1

CUDA Built-in Device Variables

- All __global__ and __device__ functions have access to these automatically defined variables
  - `dim3 gridDim;`
    - Dimensions of the grid in blocks (at most 2D)
  - `dim3 blockDim;`
    - Dimensions of the block in threads
  - `dim3 blockIdx;`
    - Block index within the grid
  - `dim3 threadIdx;`
    - Thread index within the block

Unique Thread IDs

- Built-in variables are used to determine unique thread IDs
  - Map from local thread ID (threadIdx) to a global ID which can be used as array indices



Thank you.