King Abdullah University of Science and Technology

KAUST

CS 380 - GPU and GPGPU Programming
Lecture 7: GPU Architecture, Pt. 5

Markus Hadwiger, KAUST

Reading Assignment #4 (until Sep 26)

Read (required):

- Get an overview of NVIDIA Ampere (GA102) white paper:

`https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf`

- Get an overview of NVIDIA Ampere (A100) Tensor Core GPU white paper:

`https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf`

- Get an overview of NVIDIA Hopper (H100) Tensor Core GPU white paper:

`https://resources.nvidia.com/en-us-tensor-core`

Read (optional):

- Look at the “Tuning Guides“ for different architectures in the CUDA SDK
- PTX Instruction Set Architecture (8.5): `https://docs.nvidia.com/cuda/parallel-thread-execution/`
  Read Chapters 1 – 3; get an overview of Chapter 9;
  browse through the other chapters to get a feeling for what PTX looks like
- CUDA SASS ISA (12.6), Chap. 6: `https://docs.nvidia.com/cuda/pdf/CUDA_Binary_Utilities.pdf`

Quiz #1: Sep 26

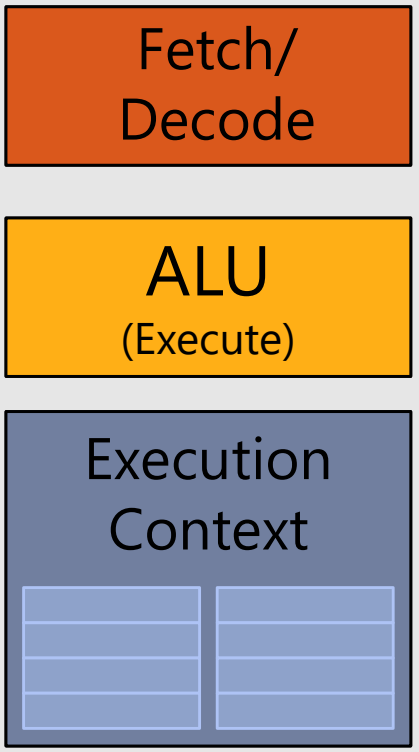
Organization

- First 30 min of lecture
- No material (book, notes, ...) allowed

Content of questions

- Lectures (both actual lectures and slides)
- Reading assignments
- Programming assignments (algorithms, methods)
- Solve short practical examples

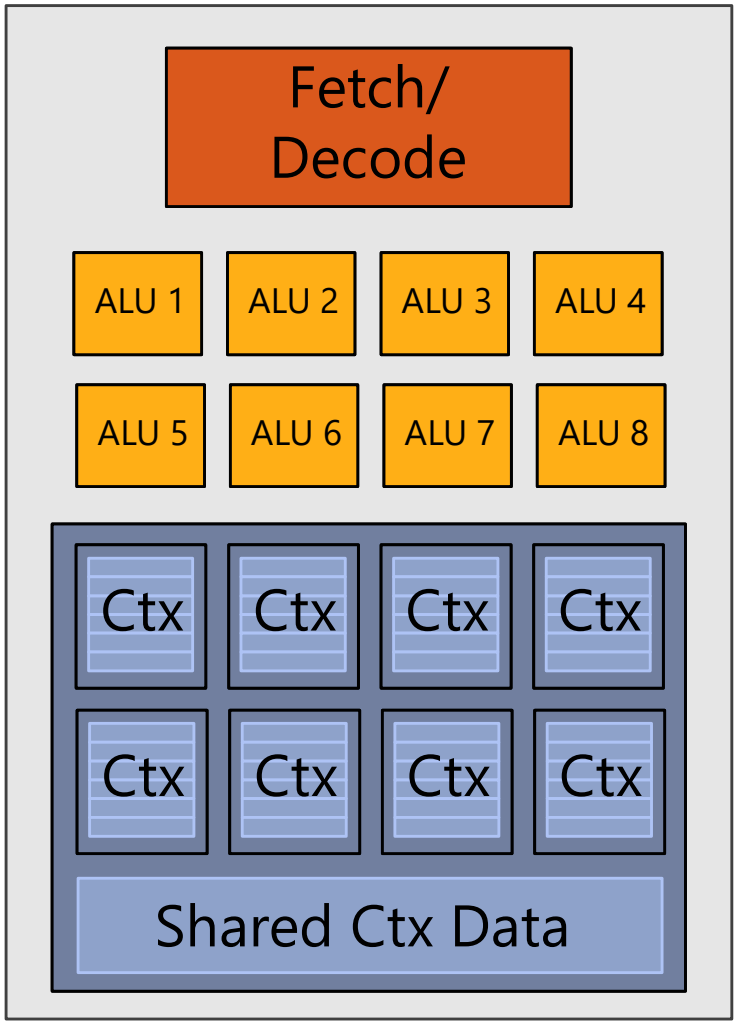
Idea #1: Slim down



Idea #1:

Remove components that help a single instruction stream run fast (i.e., that reduce latency of a single instruction stream)

Idea #2: Add ALUs (sharing inst. stream)



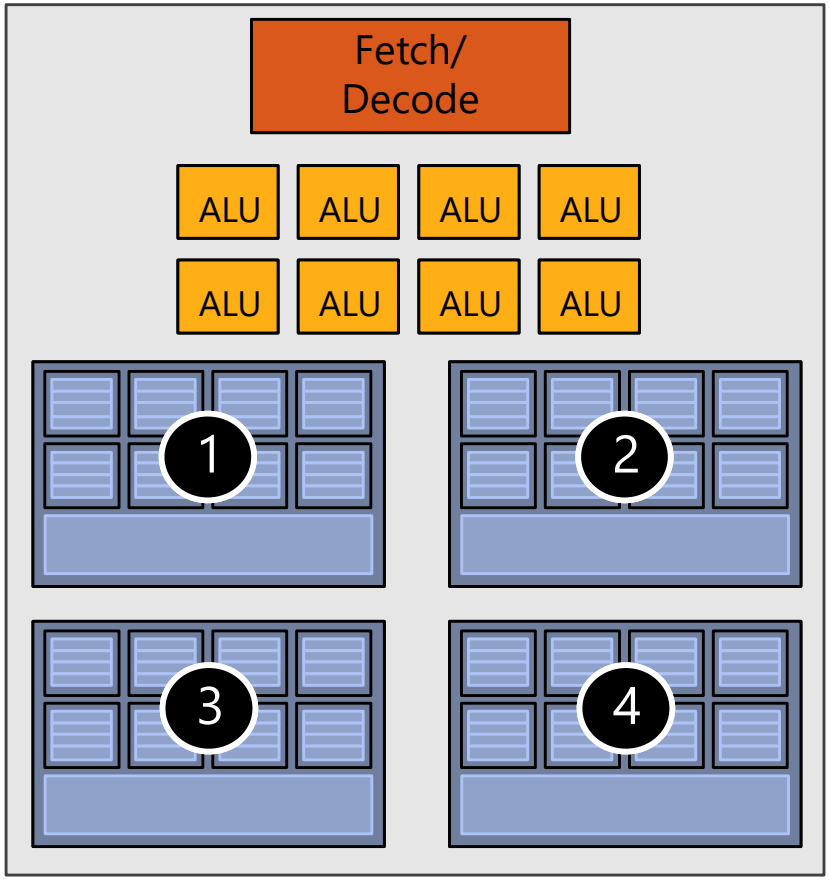
Idea #2:

Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

(or SIMT, SPMD)

Idea #3: Store multiple group contexts

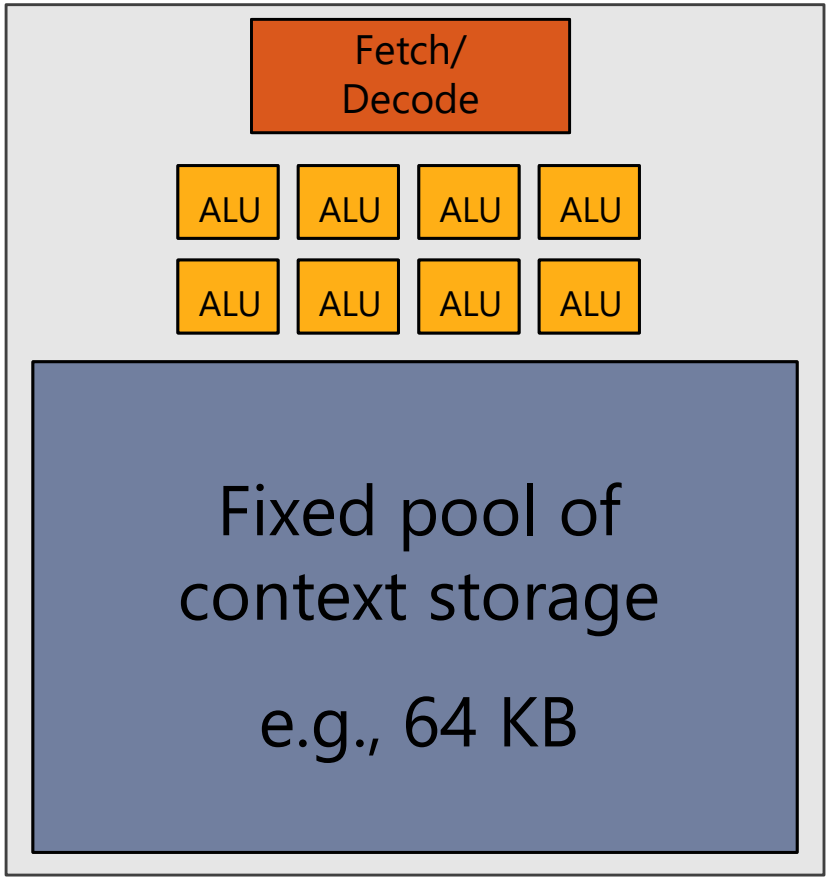


Idea #3:

Interleave execution of groups of threads

(the number of groups is *not fixed*, but depends on the context storage requirements of a given kernel!)

Idea #3: Store multiple group contexts



Idea #3:

Interleave execution of groups of threads

(the number of groups is *not fixed*, but depends on the context storage requirements of a given kernel!)

Complete GPU

16 cores

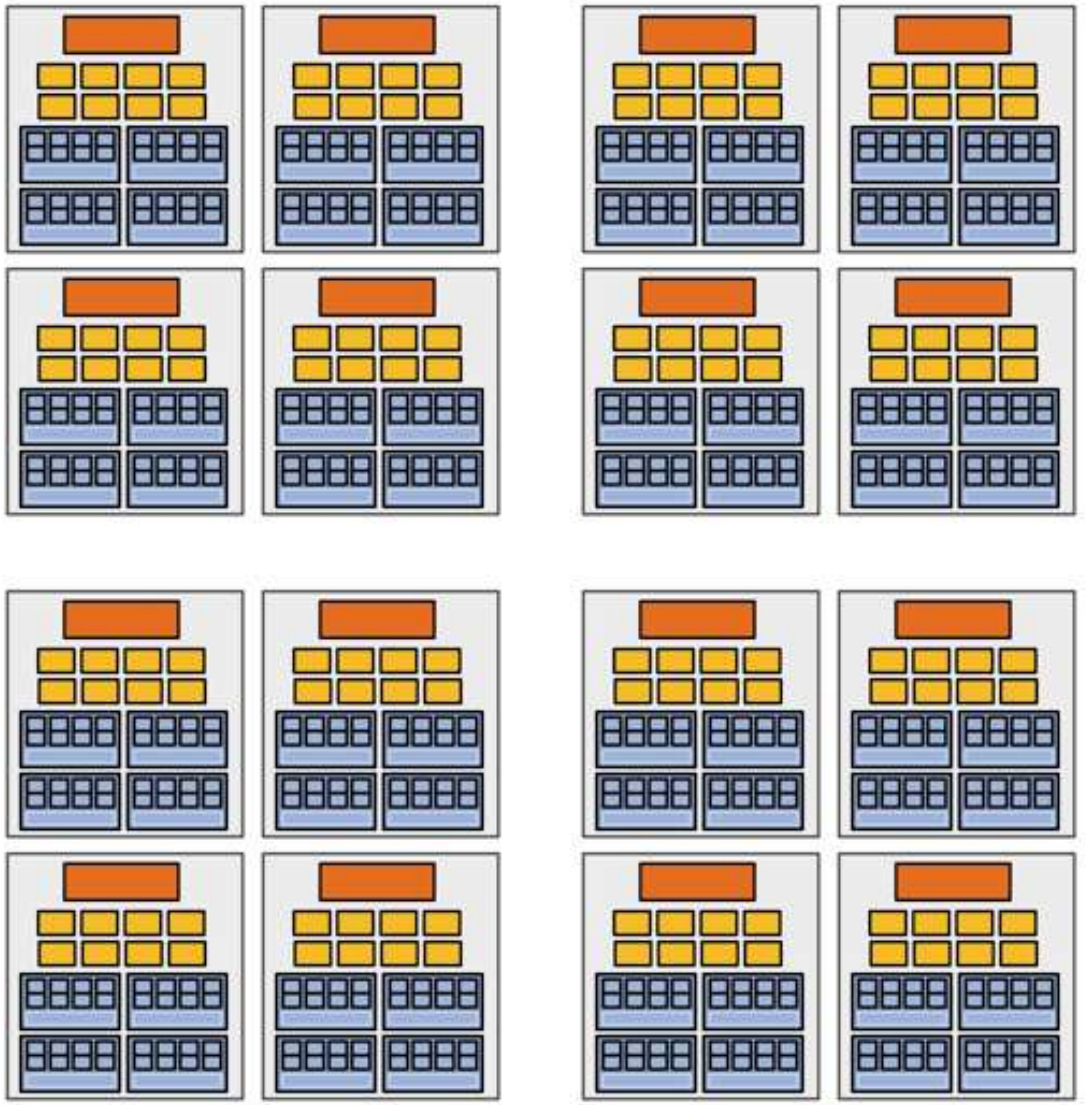
8 mul-add [mad] ALUs per core (8*16 = **128** total)

16 simultaneous instruction streams

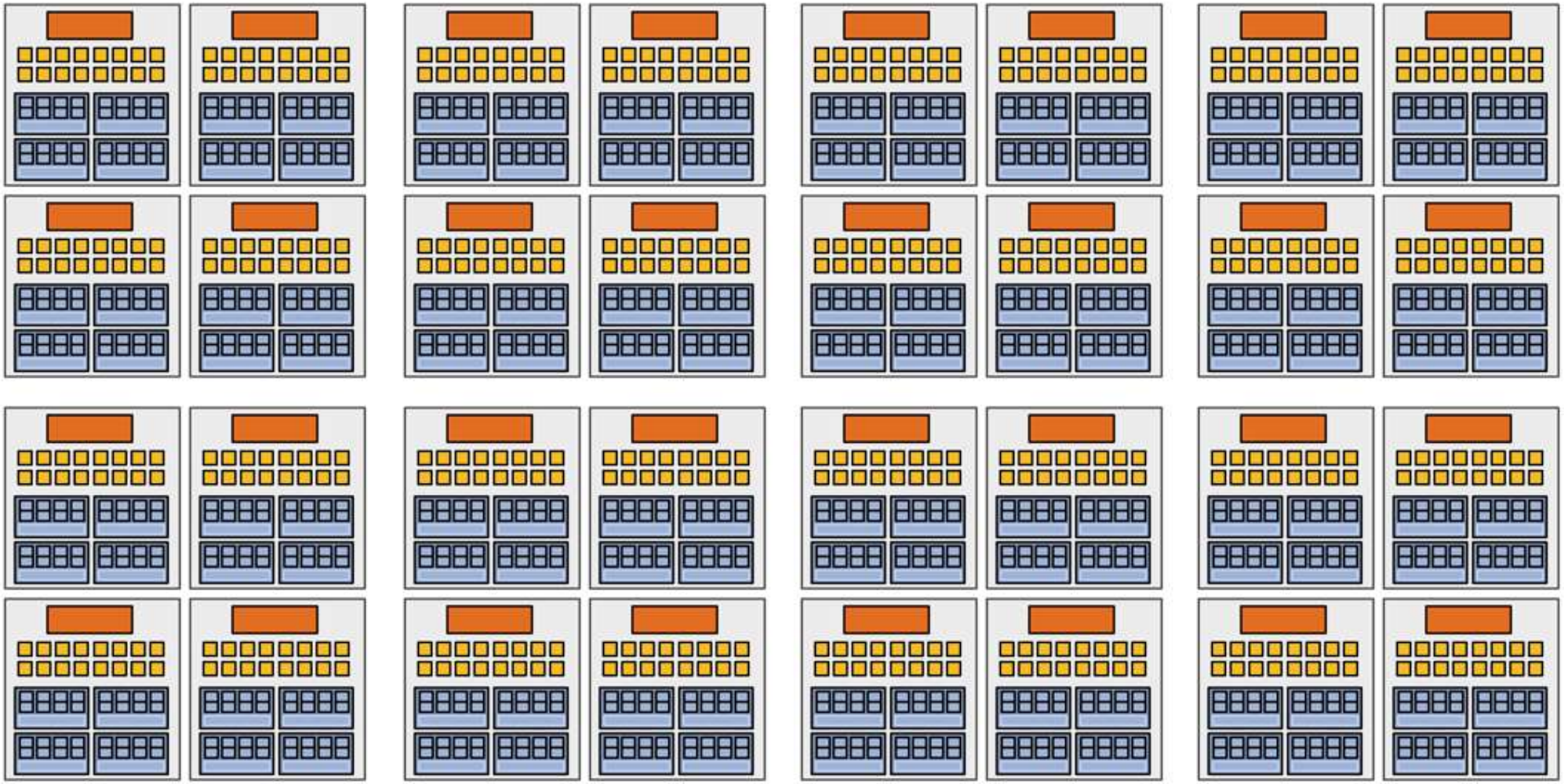
64 (4*16) concurrent (but interleaved) instruction streams

512 (8*4*16) concurrent fragments (resident threads)

= **256 GFLOPs** (@ 1GHz)
(**128** * 2 [mad] * 1G)



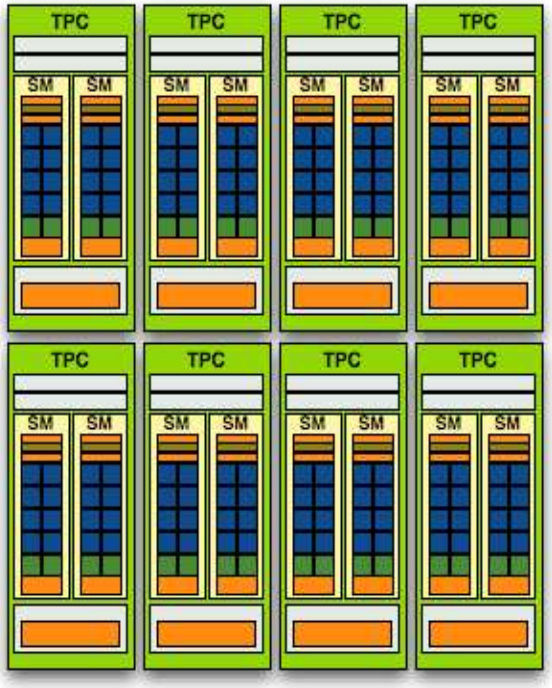
“Enthusiast” GPU (Some time ago :)



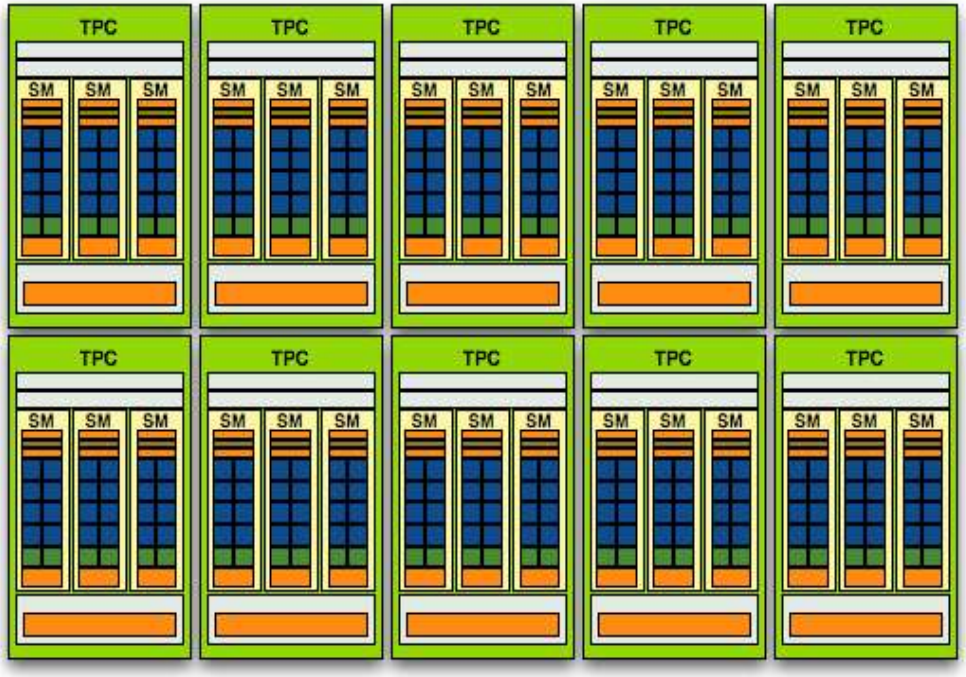
32 cores, 16 ALUs per core (512 total) = 1 TFLOP (@ 1 GHz)

NVIDIA Tesla Architecture (not the Tesla product line!), G80: 2007, GT200: 2008/2009

- G80/G92: 8 TPCs * (2 * 8 SPs) = 128 SPs [= CUDA cores]
- GT200: 10 TPCs * (3 * 8 SPs) = 240 SPs [= CUDA cores]
- **Arithmetic intensity** has increased (num. of ALUs vs. texture units)



G80 / G92

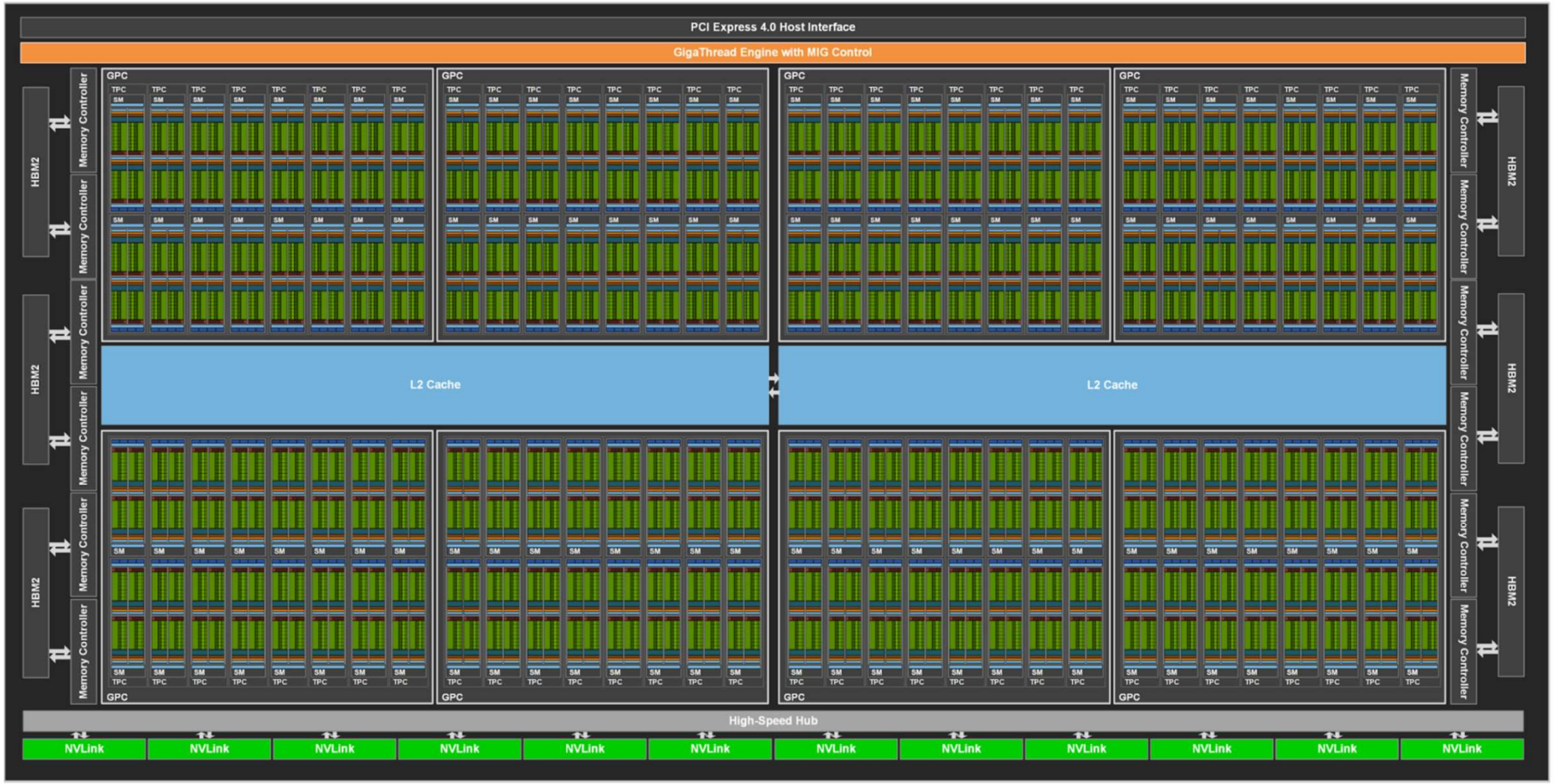


GT200

Courtesy AnandTech

NVIDIA Ampere GA100 Architecture (2020)

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



NVIDIA Hopper GH100 Architecture (2022)

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

NVIDIA Ada Lovelace AD10x Architecture (2022)

Full AD 10x

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

GPU Architecture: Real Architectures

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*

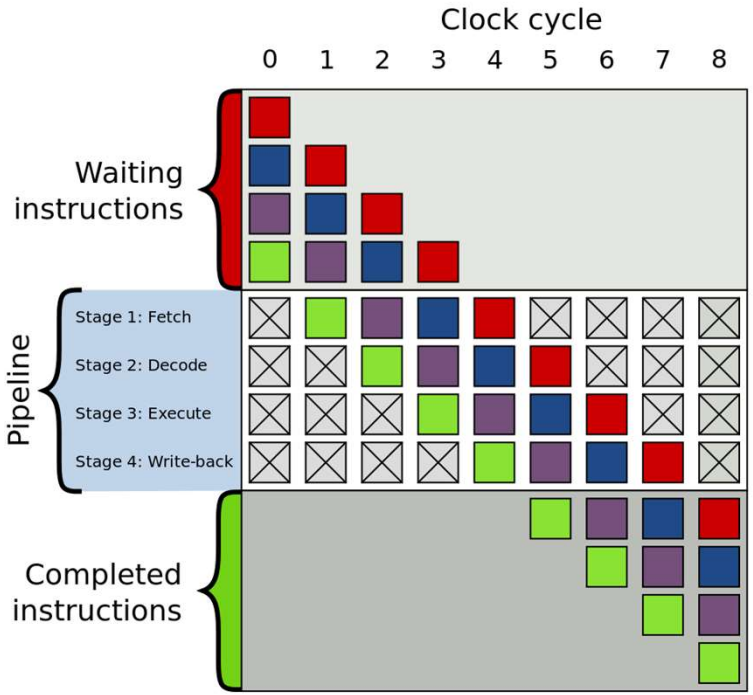
- GB200/GB202, GB20x, ...?
  (RTX 5080/5090, GB200 NVL72, HGX B100/200, ...?)

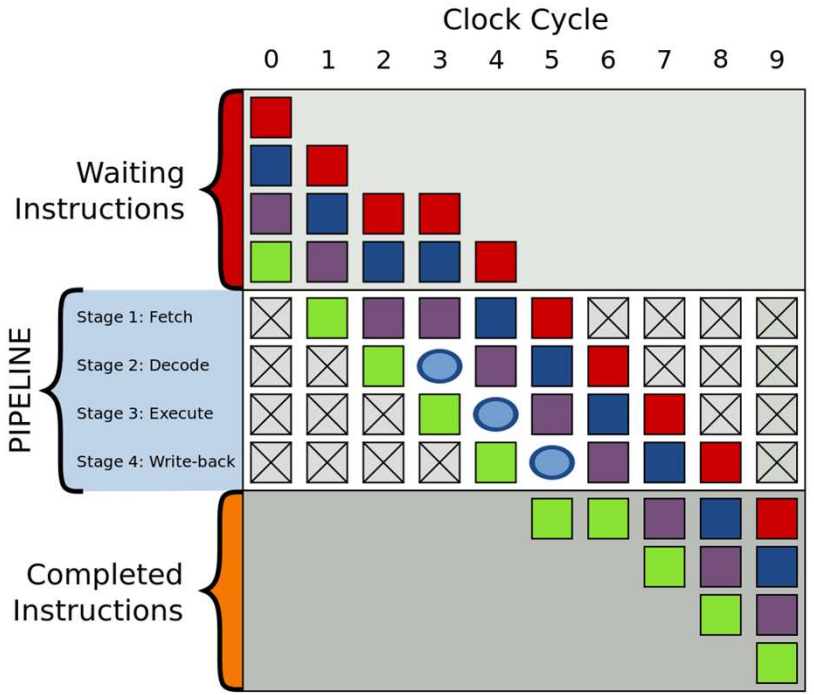
see `https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units`
and `https://en.wikipedia.org/wiki/CUDA`

Instruction Pipelining

Most basic way to exploit instruction-level parallelism (ILP)

Problem: hazards (different solutions: bubbles, …)





wikipedia

https://en.wikipedia.org/wiki/Instruction_pipelining
https://en.wikipedia.org/wiki/Classic_RISC_pipeline

Concepts: SM Occupancy in CUDA (*TLP!*)

We need to hide latencies from

- Instruction pipelining hazards (RAW – read after write, etc.)
  (also: branches; behind branch, fetch instructions from different instruction stream)
- Memory access latency

First type of latency: Definitely need to hide! (it is always there)

Second type of latency: only need to hide if it does occur (of course not unusual)

Occupancy: How close are we to *maximum latency hiding ability?*
(how many threads are resident vs. how many could be)

See run time occupancy API, or Nsight Compute: `https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#occupancy-calculator`

Concepts: Latency Hiding (Latency Tolerance)

Main goal: Avoid that instruction *throughput* goes below peak

ILP: Hide instruction pipeline latency of one instruction by pipelined execution of *independent* instruction from same thread

TLP: Hide any latency occurring for one thread (group/warp/wavefront) by *executing a different thread (group/warp/wavefront)* as soon as current thread (group/warp/wavefront) stalls:

→ *Total throughput does not go down*

GPUs

- **TLP: pull independent, not-stalling instruction from other thread group**
- ILP: pull independent instruction from same thread group (instruction stream)
- Depending on GPU: TLP often sufficient, but sometimes also need ILP
- However: If in one cycle TLP doesn't work, ILP can jump in or vice versa*

(*depending on actual microarchitecture)

ILP vs. TLP on GPUs

Main observations

- Each time unit (usually one clock cycle), a new instruction *without dependencies* should be dispatched to functional units (ALUs, SFUs, …)
- *Instruction* is a *group of threads* that is executing the same instruction: CUDA warp (32 threads), wavefront (32 or 64 threads), …
- Where can this instruction come from?
  - TLP: from another runnable warp (i.e., different instruction stream)
  - ILP: from the same warp (i.e., the same instruction stream)

How many instructions/warps per time unit (clock cycle)?

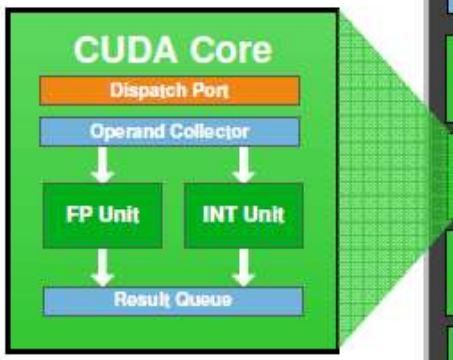
- "Scalar" pipeline (*CPI=1.0*): **TLP sufficient (if enough warps); can exploit ILP** (next instruction either from different warp, or from same warp)
- "Superscalar" (*CPI<1.0*) pipeline: dispatch more than one instruction per cycle, (#dispatchers > #warp schedulers): **need ILP!**

(CPI = clocks per instruction)

Example: “Scalar” GF100

Main concept here:

There is one instruction dispatcher
(dispatch unit / fetch/decode unit)
per warp scheduler
(warp selector)

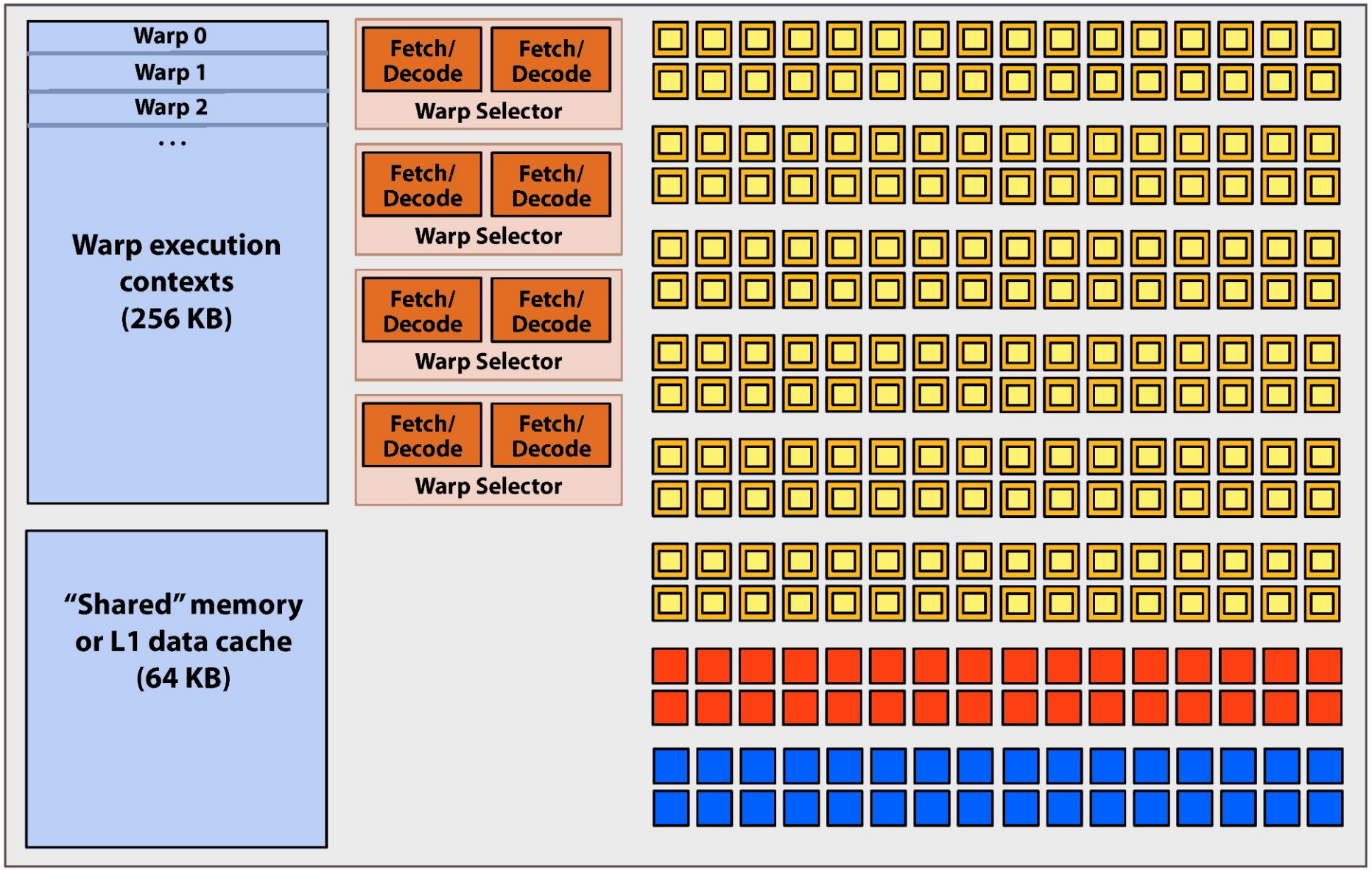


Details later...
Ignore less important subtleties...
GF100 has two warp schedulers, not one,
and each 32-thread instruction is executed
over two clock cycles, not one, etc.

Caveat on NVIDIA diagrams: if two dispatchers per warp scheduler are shown, it still doesn‘t mean that the ALU pipeline is “superscalar” (often, the second dispatcher dispatches to a *non-ALU* pipeline)
... need to look at CUDA programming guide info, also given in our tables in row “*# ALU dispatch / warp sched.*”

Example: "Superscalar" ALUs in SM Architecture

NVIDIA Kepler GK104 architecture SMX unit (one "core")



= SIMD function unit, control shared across 32 units (1 MUL-ADD per clock)

= "special" SIMD function unit, control shared across 32 units (operations like sin/cos)

= SIMD load/store unit (handles warp loads/stores, gathers/scatters)

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

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
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (`__log2f`), base 2 exponential (`exp2f`), sine (`__sinf`), cosine (`__cosf`)	32			16	32		16				
32-bit integer add, extended-precision add, subtract, extended-precision subtract	160	128		64	128		64				
32-bit integer multiply, multiply-add, extended-precision multiply-add	32	Multiple instruct.					64				
							32 for extended-precision				

list continues…

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

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**

IF no other stalls occur!
(i.e., except inst. pipe hazards)

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

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**

IF no other stalls occur!
(i.e., except inst. pipe hazards)

"superscalar"

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

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**

IF no other stalls occur!
(i.e., except inst. pipe hazards)

"superscalar"

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

NVIDIA Tesla Architecture

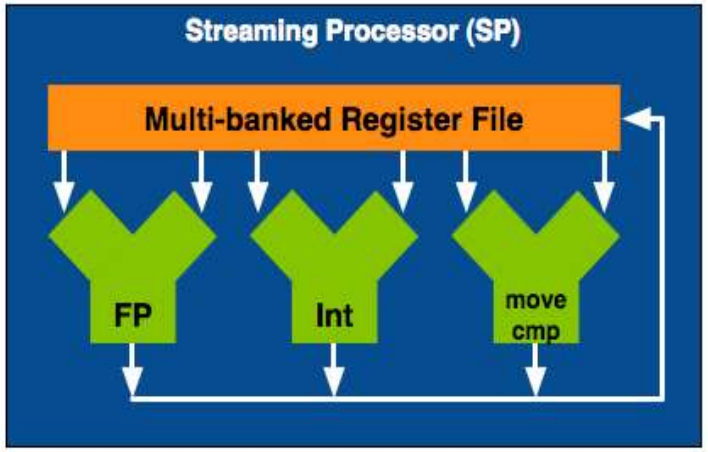
2007-2009

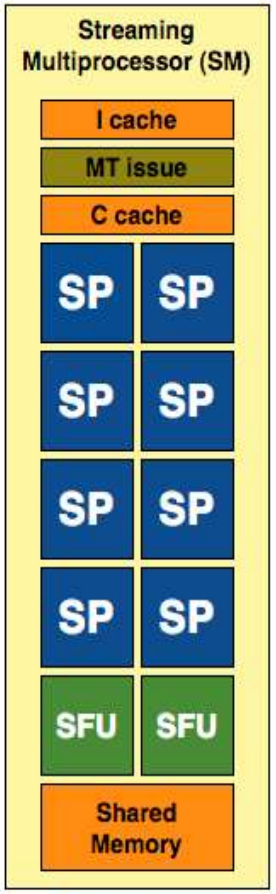
(compute capability 1.x)

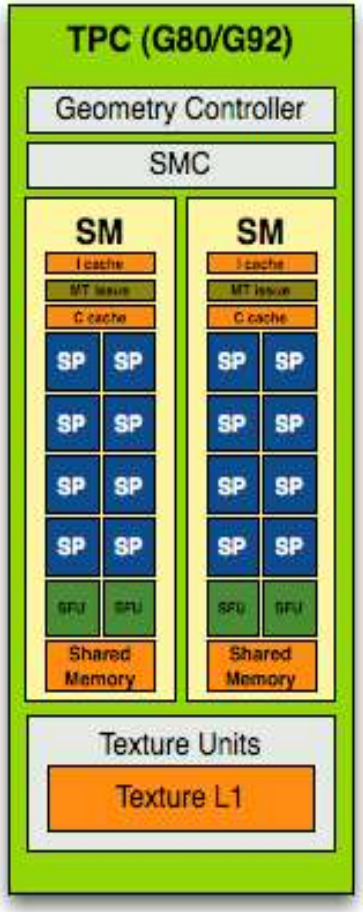
G80 (cc 1.0): 2007 (Geforce 8800, ...)
G9x (cc 1.1): 2008 (Geforce 9800, ...)
GT200 (cc 1.3): 2008/2009 (GTX 280, GTX 285, ...)

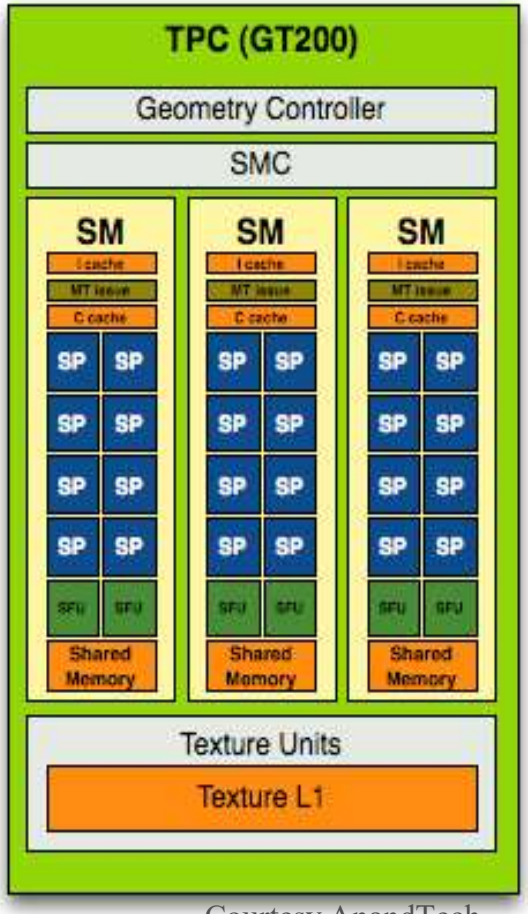
(this is not the Tesla product line!)

NVIDIA Tesla Architecture (not the Tesla product line!), G80: 2007, GT200: 2008/2009









Courtesy AnandTech

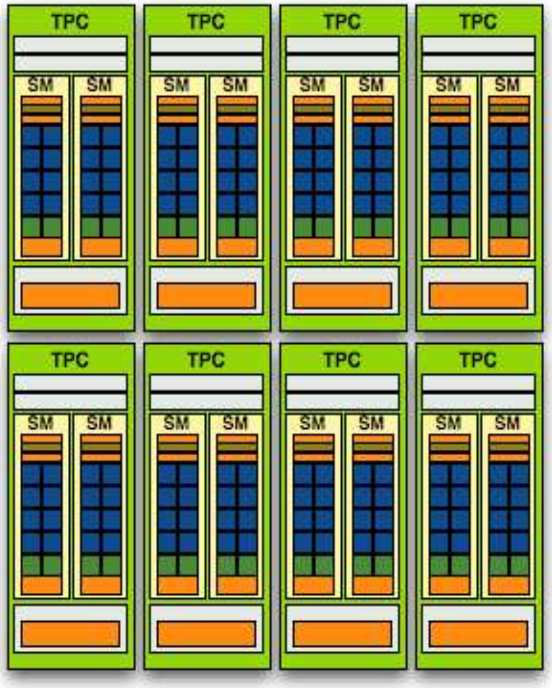
G80: first CUDA GPU!

Multiprocessor: SM (CC 1.x)

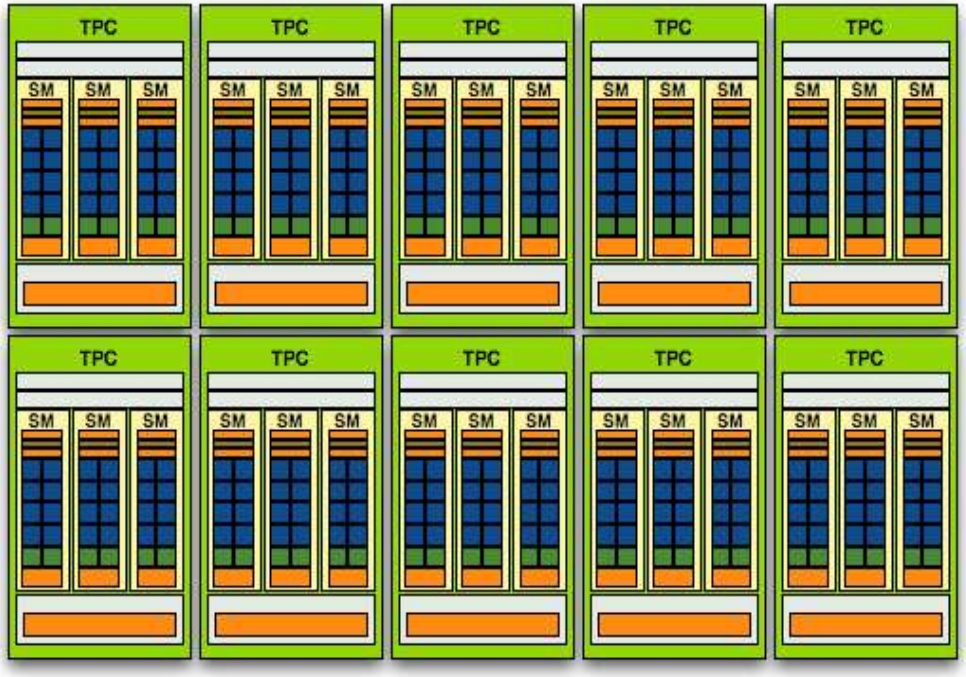
- Streaming Processor (SP) [or: CUDA core; or: FP32 / FP64 / INT32 core, ...]
- Streaming Multiprocessor (SM)
- Texture/Processing Cluster (TPC)

NVIDIA Tesla Architecture (not the Tesla product line!), G80: 2007, GT200: 2008/2009

- G80/G92: 8 TPCs * (2 * 8 SPs) = 128 SPs [= CUDA cores]
- GT200: 10 TPCs * (3 * 8 SPs) = 240 SPs [= CUDA cores]
- **Arithmetic intensity** has increased (num. of ALUs vs. texture units)



G80 / G92



GT200

Courtesy AnandTech

NVIDIA Fermi Architecture

2010

(compute capability 2.x)

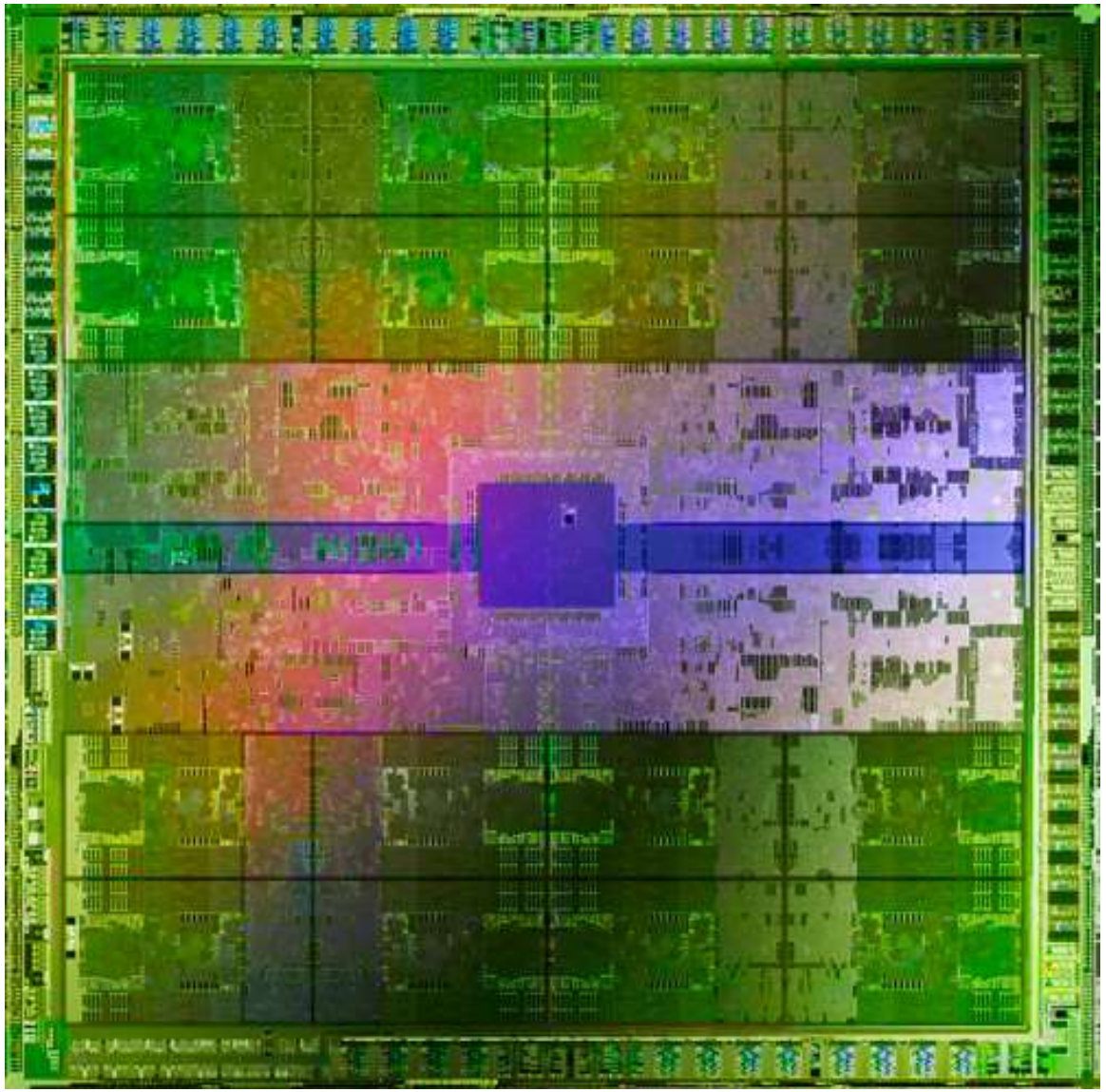
GF100 (cc 2.0), ... (GTX 480, ...)
GF104 (cc 2.1), ... (GTX 460, ...)
GF110 (cc 2.0), ... (GTX 580, ...)

NVIDIA Fermi (GF100) Architecture (2010)

Full size

- 4 GPCs
- 4 SMs each
- 6 64-bit memory controllers (= 384 bit)

NVIDIA Fermi (GF100) Die Photo



Full size

- 4 GPCs
- 4 SMs each

ALU Instruction Latencies and Instructs. / SM

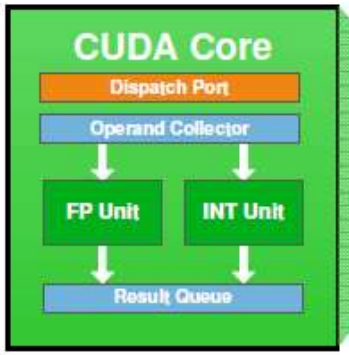
CC	2.0 (Fermi)	2.1 (Fermi)	3.x (Kepler)	5.x (Maxwell)	6.0 (Pascal)	6.1/6.2 (Pascal)	7.x (Volta, Turing)	8.0/8.6 (Ampere)	8.9/9.0 (Ada/Hopper)
# warp sched. / SM	2	2	4	4	2	4	4	4	4
# ALU dispatch / warp sched.	1 (over 2 clocks)	2 (over 2 clocks)	2	1	1	1	1	1	1
SM busy with # warps + inst	L	2L	8L	4L	2L	4L	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 GF100 SM (2010)

Multiprocessor: SM (CC 2.0)

Streaming processors now called *CUDA cores*



32 CUDA cores per Fermi GF100/GF110 streaming multiprocessor (SM)

Example GPU with 15 SMs = 480 CUDA cores (GTX 480)

Example GPU with 16 SMs = 512 CUDA cores (GTX 580)

CPU-like cache hierarchy

- L1 cache / shared memory
- L2 cache

Texture units and caches now in SM

(instead of with TPC=multiple SMs in G80/GT200)

Graphics Processor Clusters (GPC)

(instead of TPC on GT200)

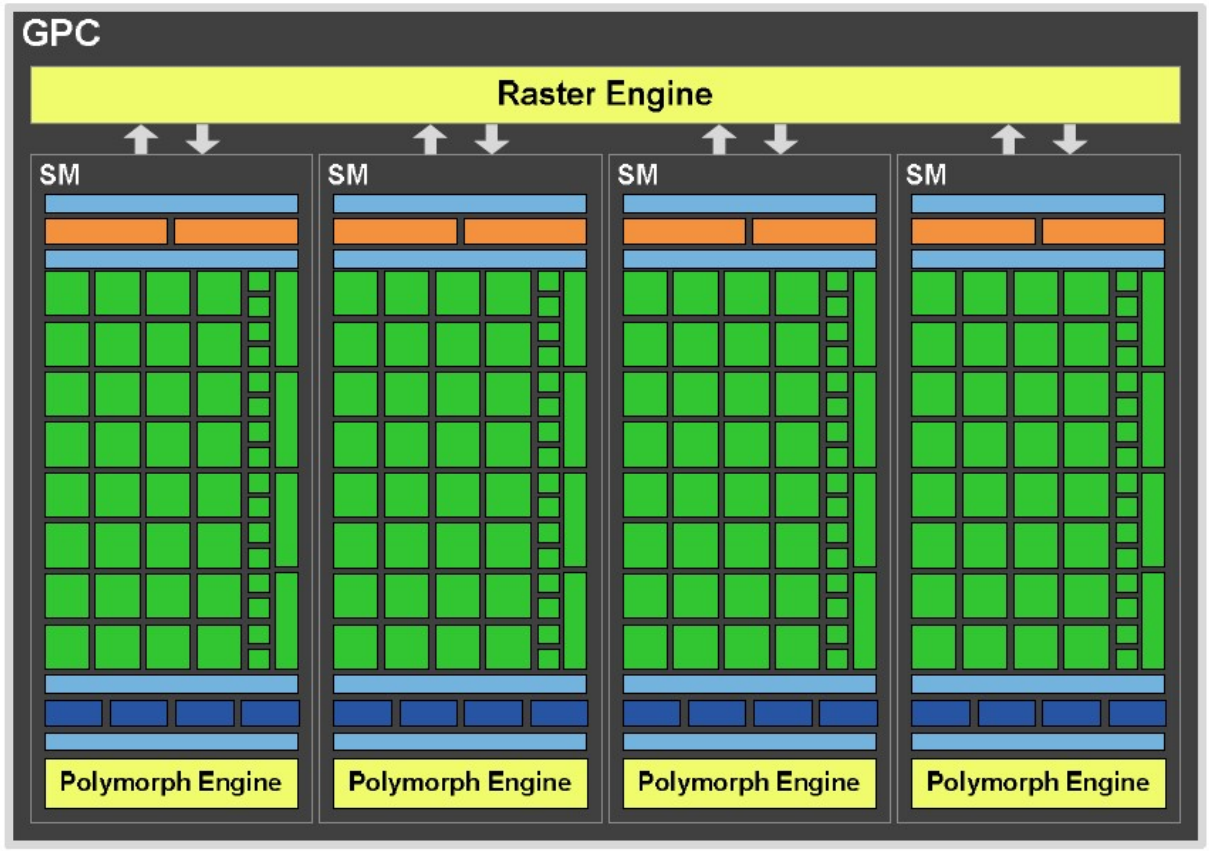
4 SMs

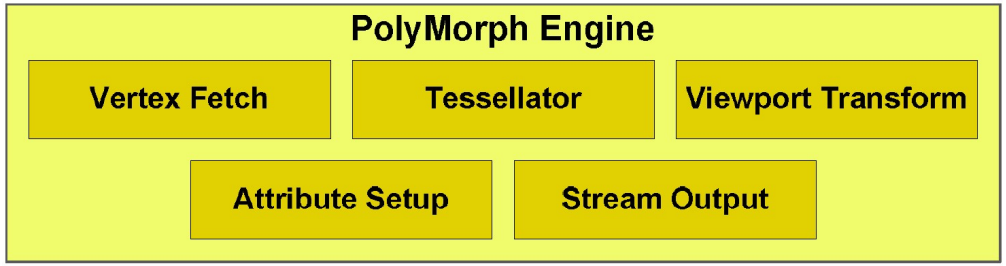
32 CUDA cores / SM

4 SMs / GPC =
128 cores / GPC

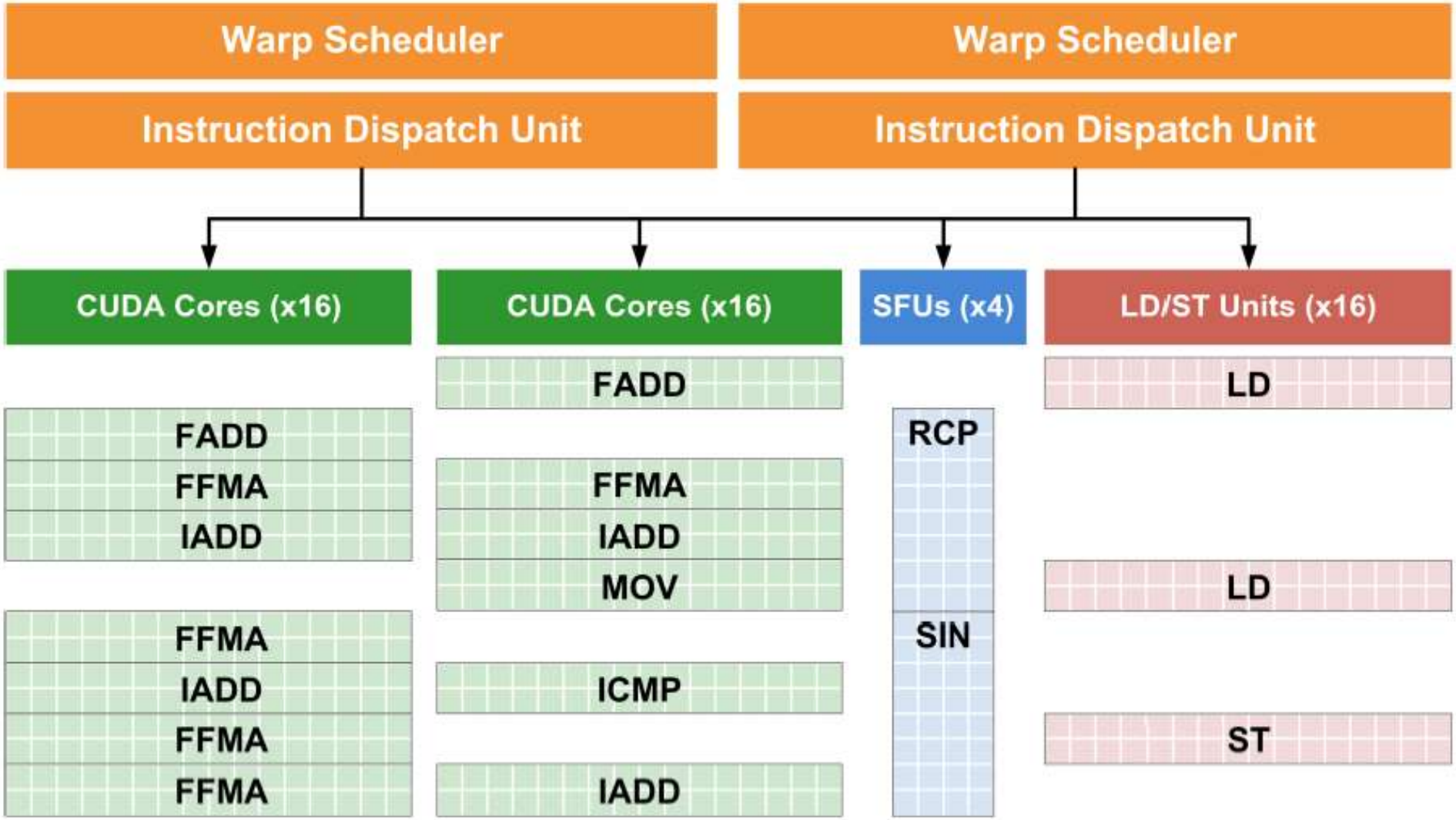
Decentralized rasterization and geometry

- 4 raster engines
- 16 ”PolyMorph” engines

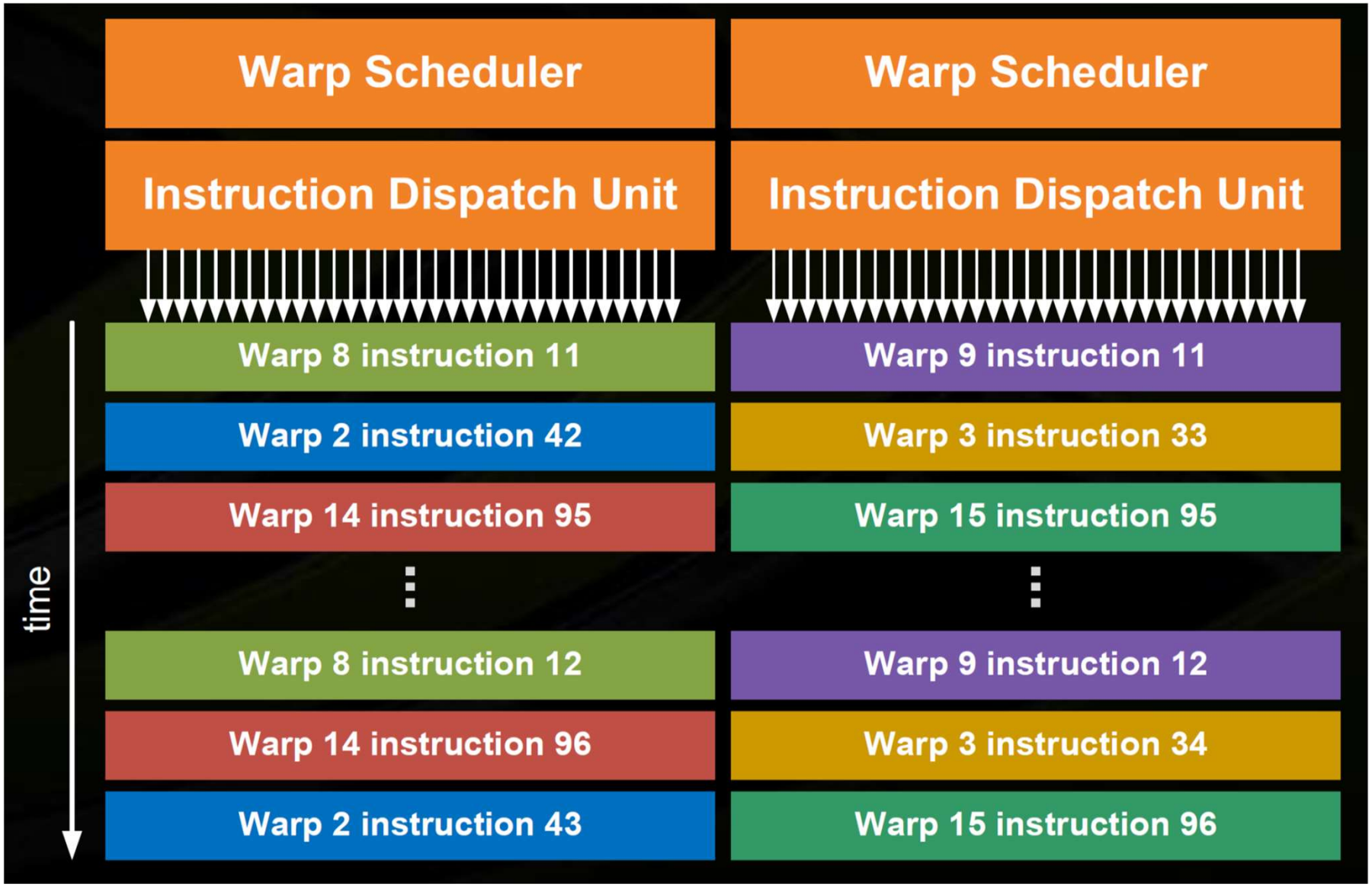




Dual Warp Schedulers



Dual Warp Schedulers

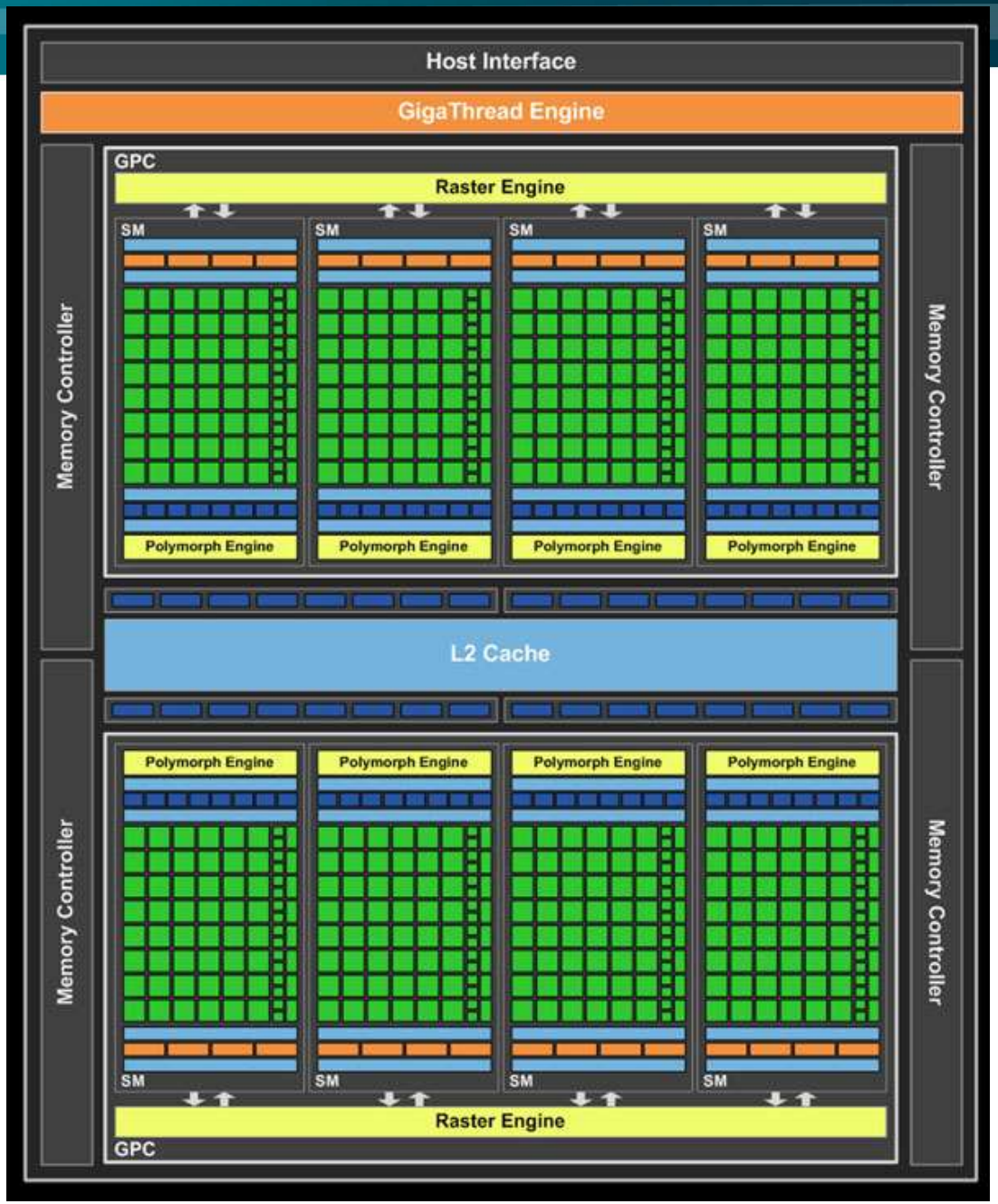


NVIDIA Fermi (GF104) Architecture (2010)

Full size GF104

- 2 GPCs
- 4 SMs each

- SM design different from GF100 / GF110 !
- Fewer total SMs, but each SM is “superscalar”



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 GF104 SM (2010)

Multiprocessor: SM (CC 2.1)

Streaming processors now called *CUDA cores*

48 CUDA cores per Fermi GF104 streaming multiprocessor (SM)

Example GPU with 7 SMs = 336 CUDA cores (GTX 460)

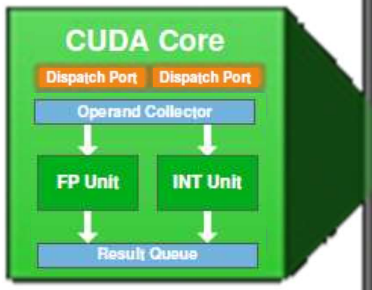
2 dispatch units / warp scheduler: "superscalar"

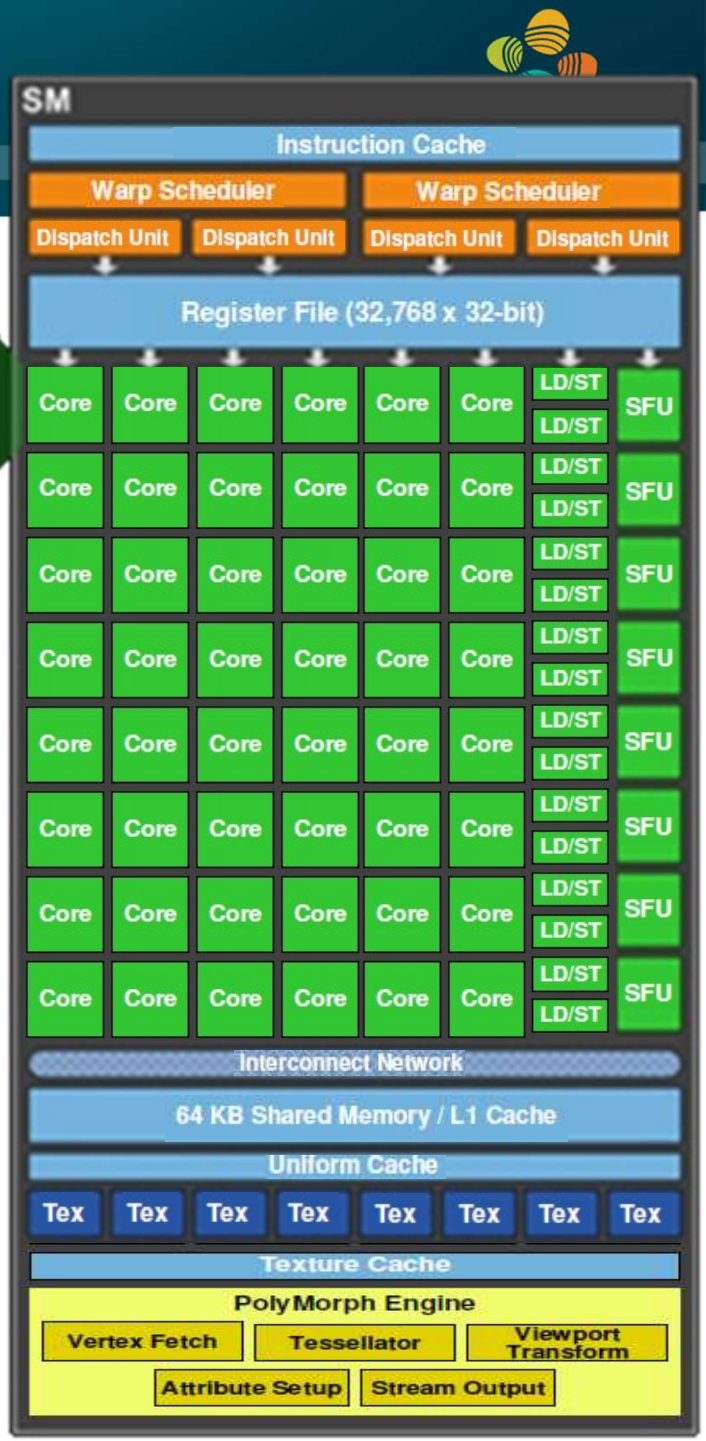
CPU-like cache hierarchy

- L1 cache / shared memory
- L2 cache

Texture units and caches now in SM

(instead of with TPC=multiple SMs in G80/GT200)

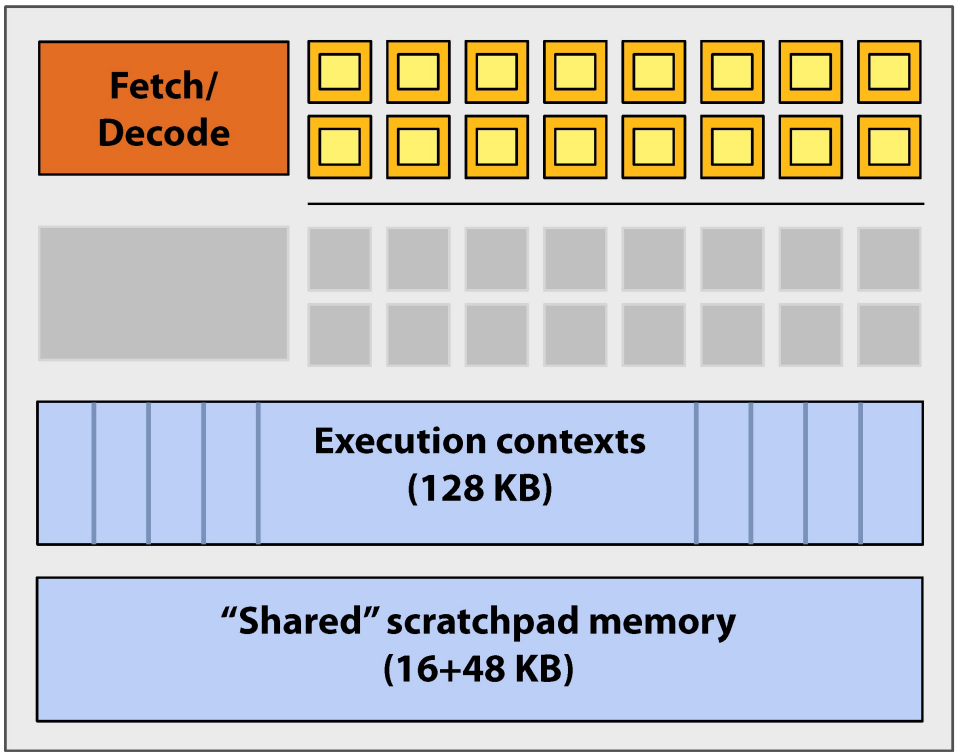


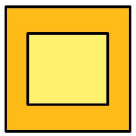


NVIDIA Fermi GF100 Architecture (2010)

NVIDIA GeForce GTX 480 "core"

CC 2.0, not 2.1 !





= SIMD function unit,
control shared across 16 units
(1 MUL-ADD per clock)

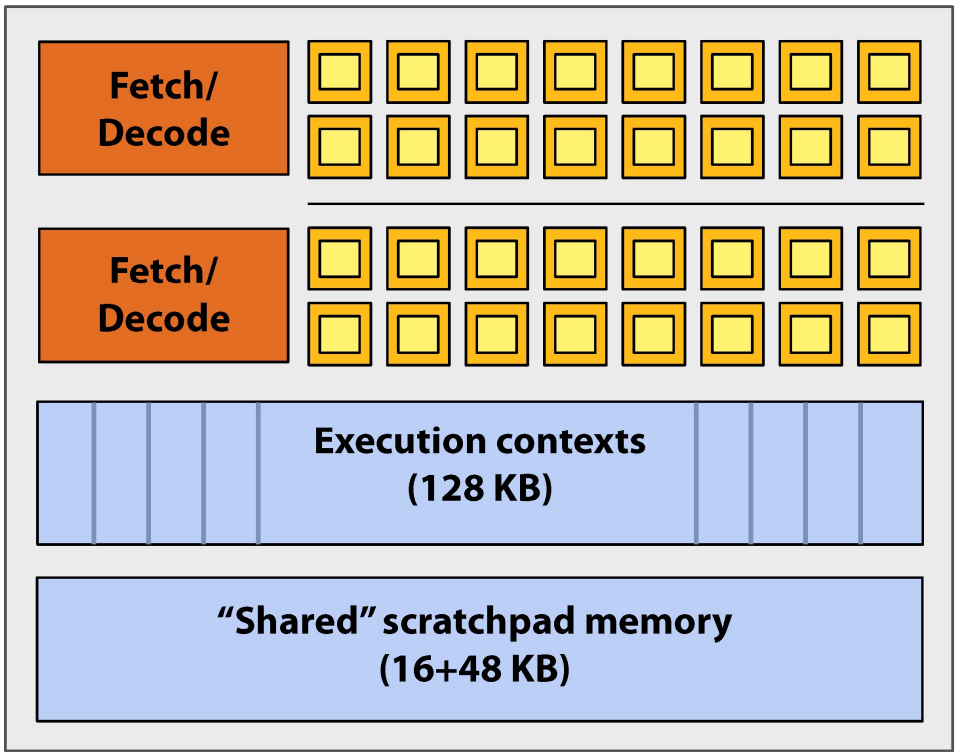
- Groups of 32 fragments share an instruction stream
- Up to 48 groups are simultaneously interleaved
- Up to 1536 individual contexts can be stored

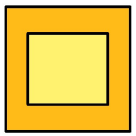
Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

NVIDIA Fermi GF100 Architecture (2010)

NVIDIA GeForce GTX 480 "core"

CC 2.0, not 2.1 !





= SIMD function unit,
control shared across 16 units
(1 MUL-ADD per clock)

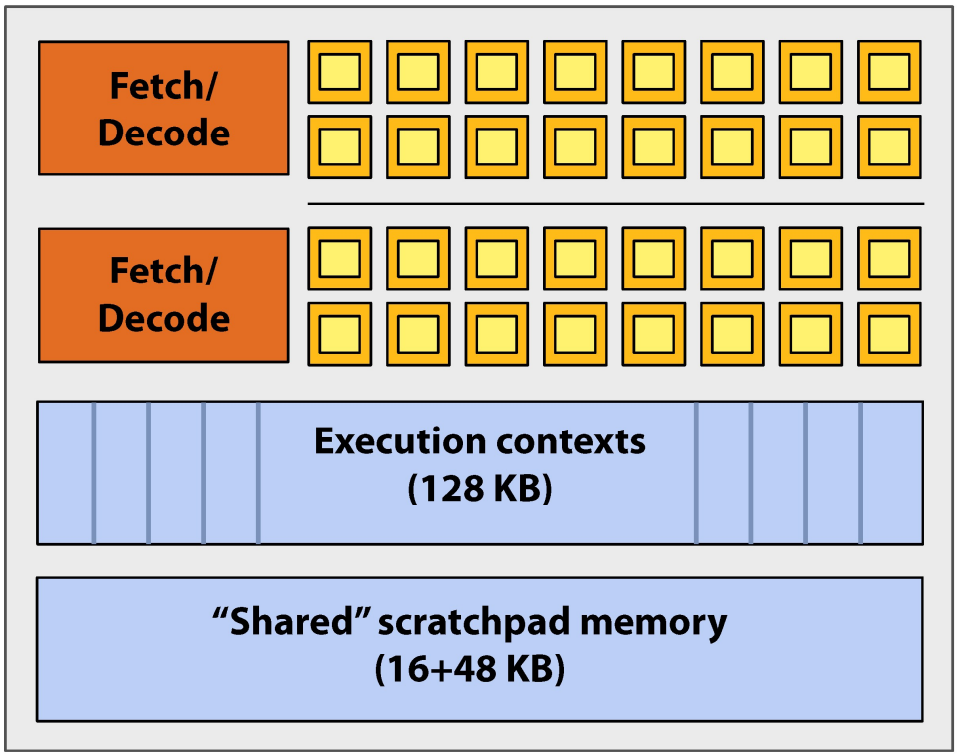
- The core contains 32 functional units
- Two groups are selected each clock (decode, fetch, and execute two instruction streams in parallel)

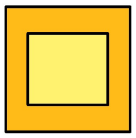
Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

NVIDIA Fermi GF100 Architecture (2010)

NVIDIA GeForce GTX 480 "SM"

CC 2.0, not 2.1 !





= CUDA core
(1 MUL-ADD per clock)

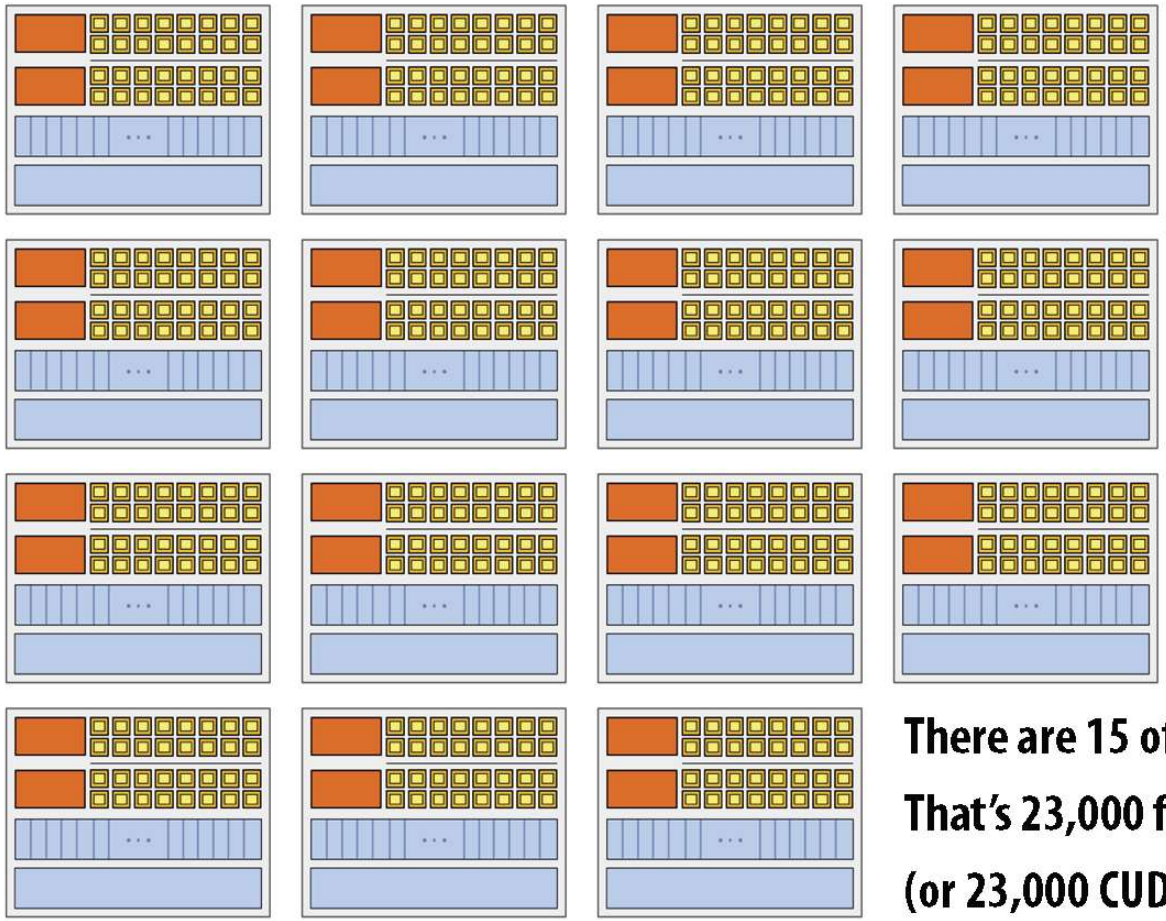
- The SM contains 32 CUDA cores
- Two warps are selected each clock (decode, fetch, and execute two warps in parallel)
- Up to 48 warps are interleaved, totaling 1536 CUDA threads

Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

NVIDIA Fermi GF100 Architecture (2010)

NVIDIA GeForce GTX 480

CC 2.0, not 2.1 !



There are 15 of these things on the GTX 480:

That's 23,000 fragments!

(or 23,000 CUDA threads!)

NVIDIA Kepler Architecture

2012

(compute capability 3.x)

GK104 (cc 3.0), ... (GTX 680, ...)
GK110 (cc 3.5), ... (GTX 780, GTX Titan (Black), ...)
GK210 (cc 3.7), ... (Tesla K80)

NVIDIA Kepler Architecture (2012)

Instruction Throughput

Instruction throughput numbers in CUDA C Programming Guide (Chapter 5.4)

	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

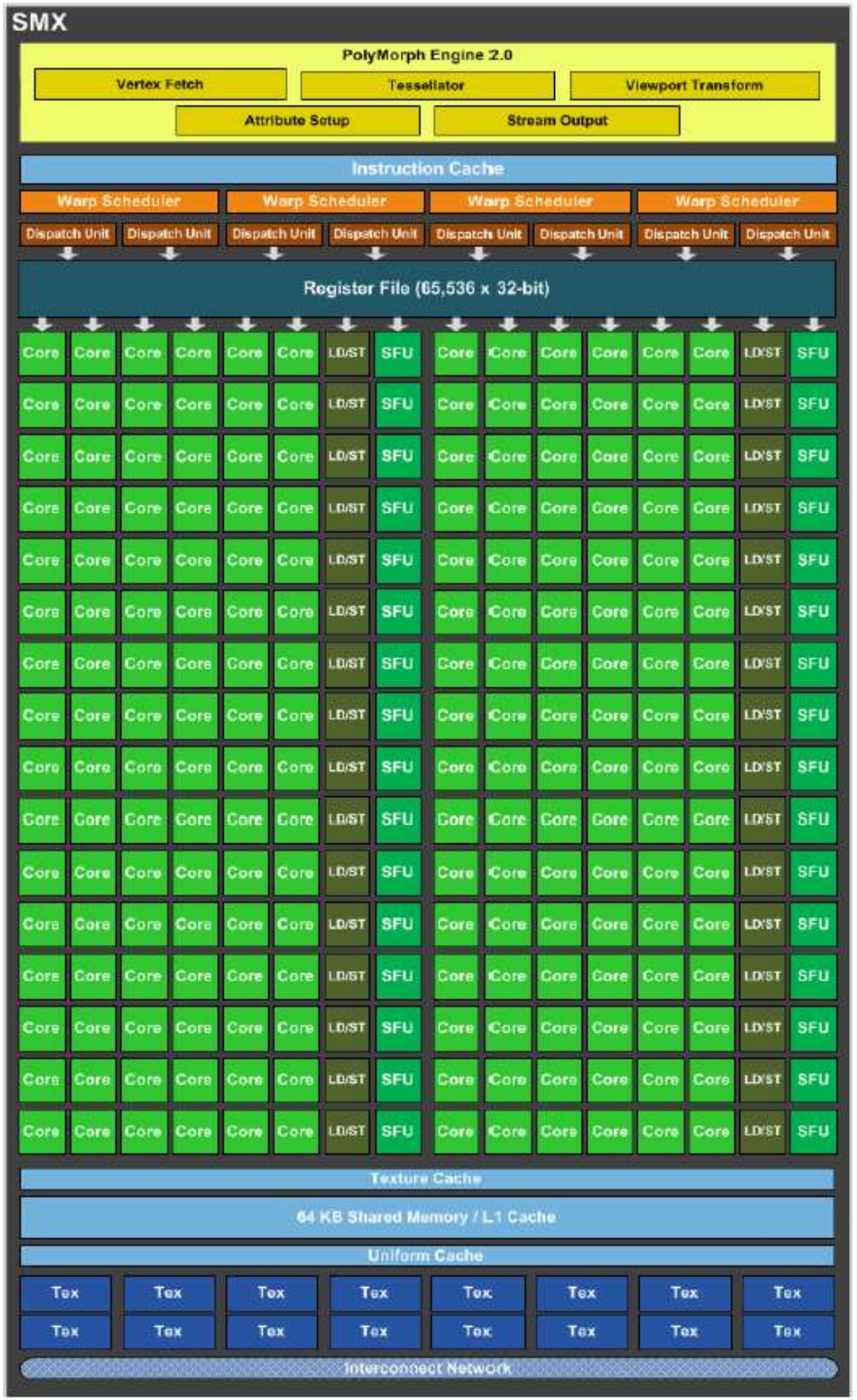
GK104 SMX

Multiprocessor: SMX (CC 3.0)

- 192 CUDA cores
  (192 = 6 * 32)
- 32 LD/ST units
- 32 SFUs
- 16 texture units

Two dispatch units per warp scheduler exploit ILP (*instruction-level parallelism*)

Can dual-issue ALU instructions! ("superscalar")



GK110 SMX

Multiprocessor: SMX (CC 3.5)

- 192 CUDA cores (192 = 6 * 32)
- 64 DP units
- 32 LD/ST units
- 32 SFUs
- 16 texture units

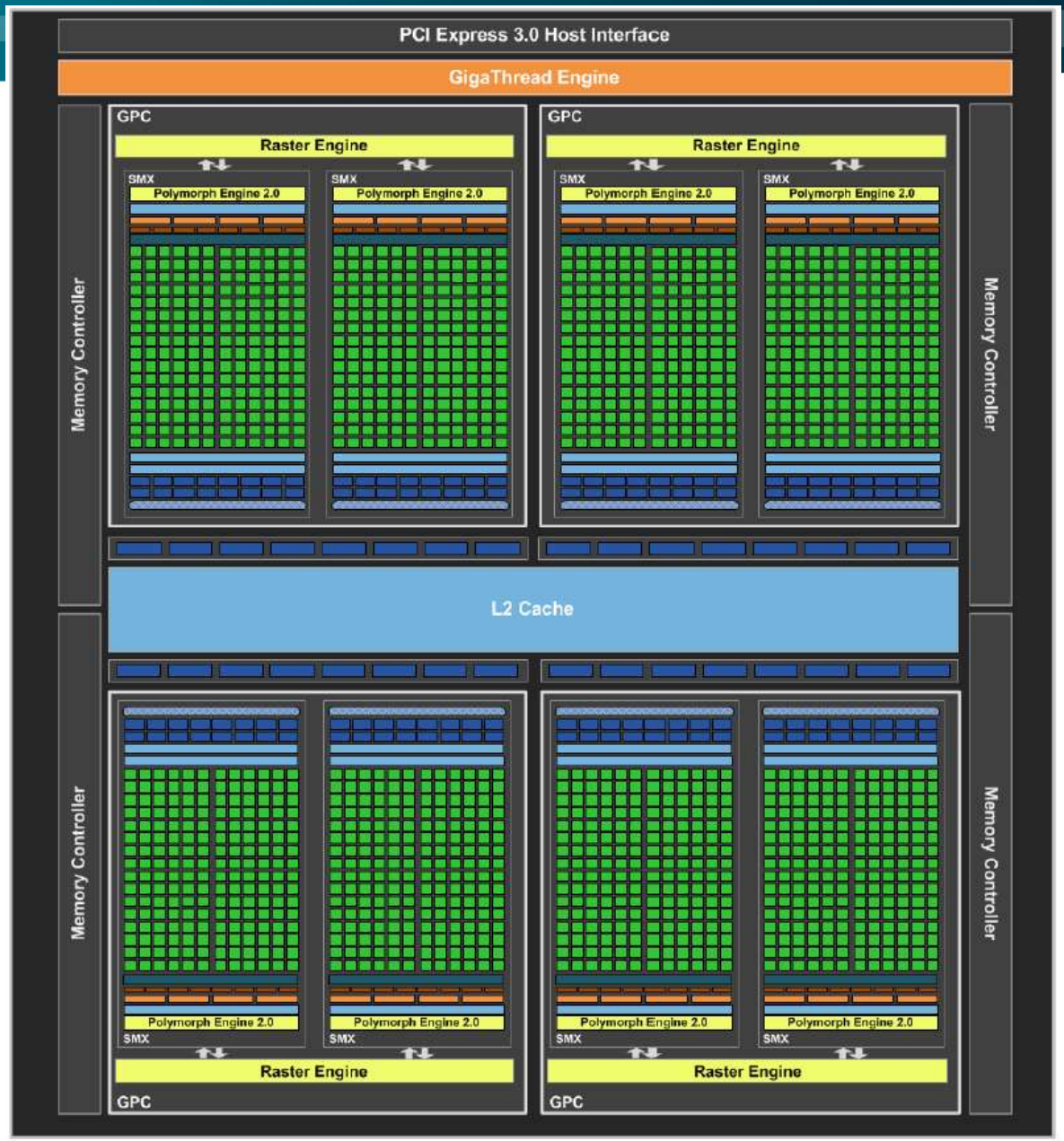
New read-only data cache (48KB)

NVIDIA Kepler Architecture (2012)

Three different versions

- Compute capability 3.0 (GK104)
  - Geforce GTX 680, …
  - Quadro K5000
  - Tesla K10
- Compute capability 3.5 (GK110)
  - Geforce GTX 780 / Titan / Titan Black
  - Quadro K6000
  - Tesla K20, Tesla K40
- Compute capability 3.7 (GK210)
  - Tesla K80
  - Came out much later (~end of 2014)

NVIDIA Kepler / GK104 Structure



Full size

- 4 GPCs
- 2 SMXs each

= 8 SMXs,
1536 CUDA cores

NVIDIA Kepler / GK110 Structure (1)

Full size

- 15 SMXs (Titan Black; Titan: 14)
- 2880 CUDA cores (Titan Black; Titan: 2688)
- 5 GPCs of 3 SMXs each

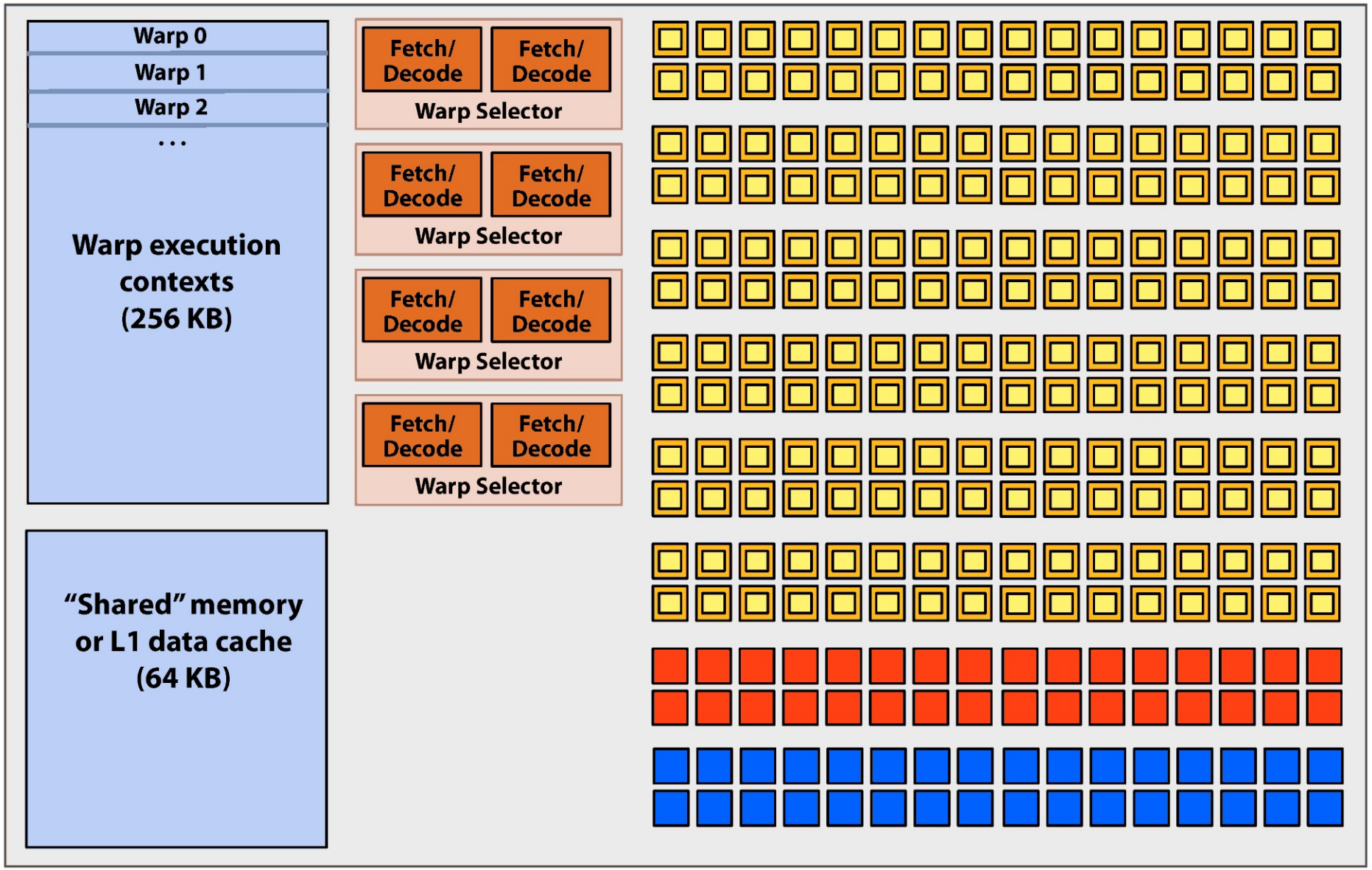
NVIDIA Kepler / GK110 Structure (2)

Titan (not Black)

- 14 SMXs
- 2688 CUDA cores
- 5 GPCs with 3 SMXs or 2 SMXs each

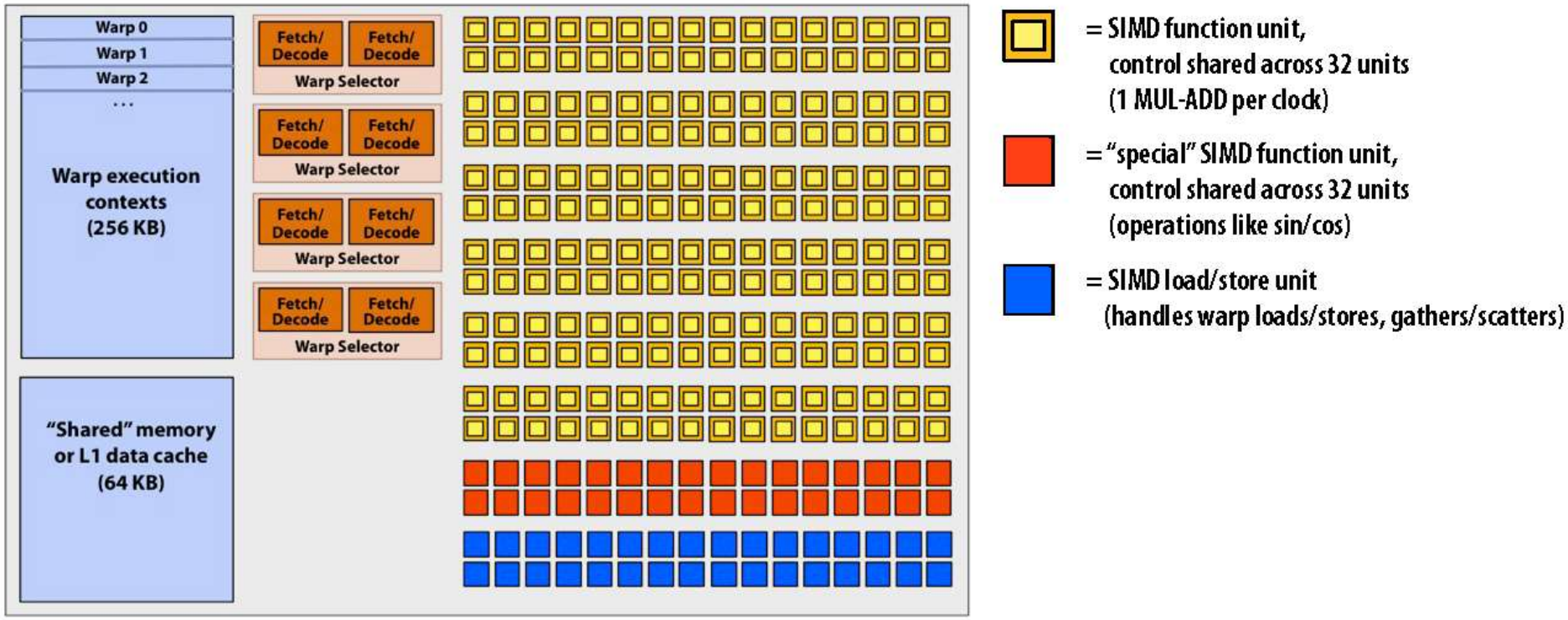
Bonus slides: NVIDIA GTX 680 (2012)

NVIDIA Kepler GK104 architecture SMX unit (one "core")



Bonus slides: NVIDIA GTX 680 (2012)

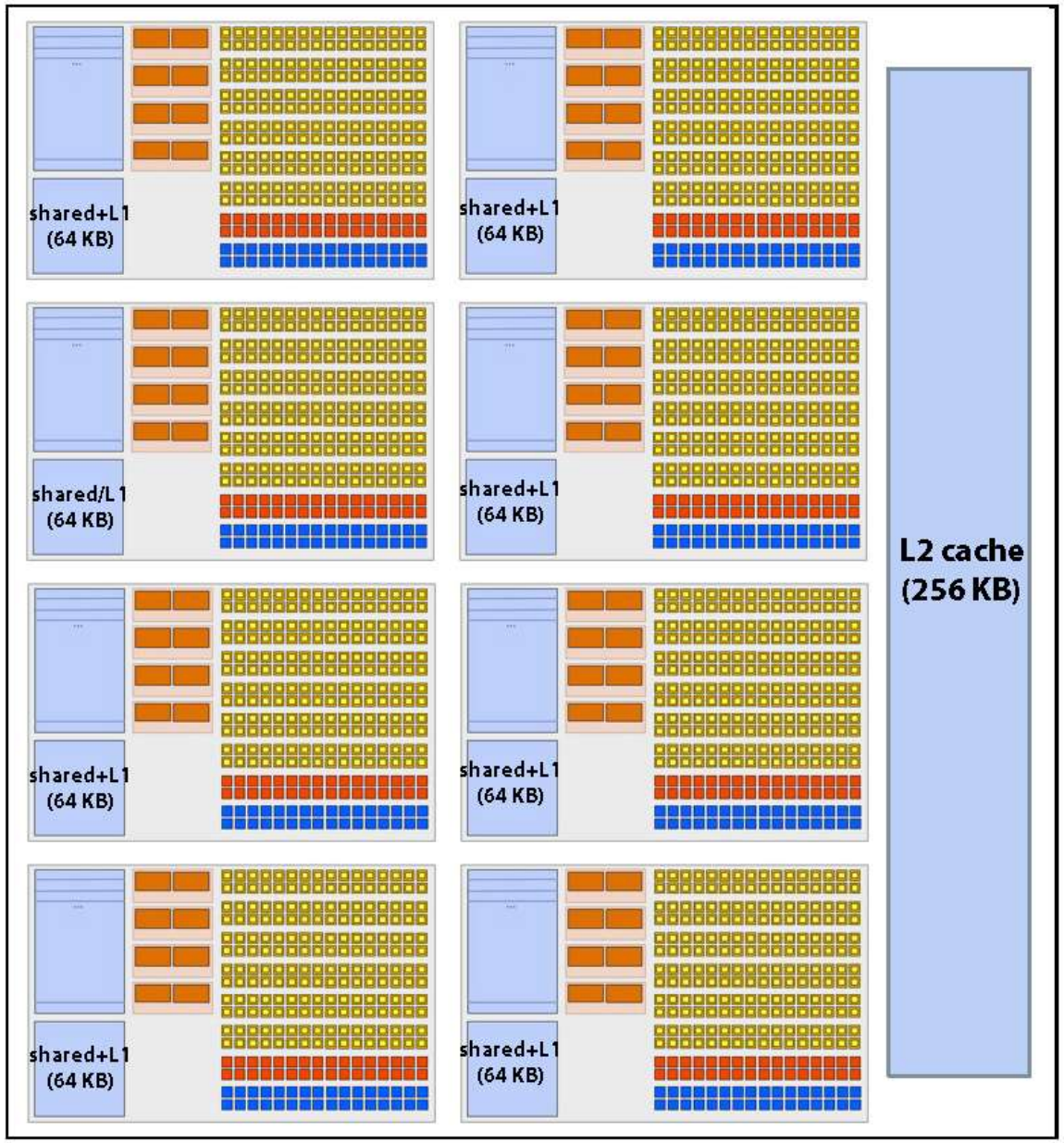
NVIDIA Kepler GK104 architecture SMX unit (one "core")



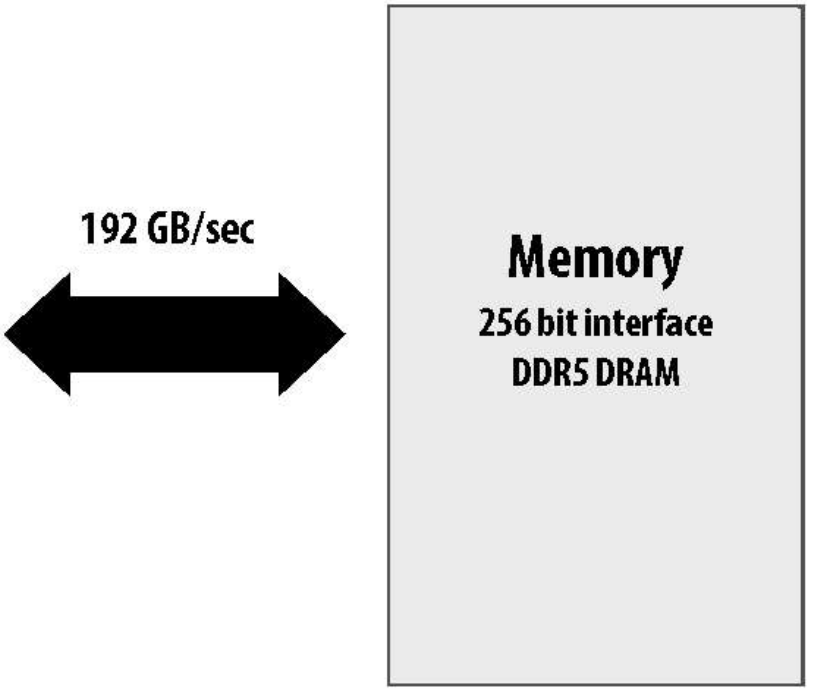
- SMX core resource limits:
  - Maximum warp execution contexts: 64 (2,048 total CUDA threads)
  - Maximum thread blocks: 16
- SMX core operation each clock:
  - Select up to four runnable warps from up to 64 resident on core (thread-level parallelism)
  - Select up to two runnable instructions per warp (instruction-level parallelism)
  - Execute instructions on available groups of SIMD ALUs, special-function ALUs, or LD/ST units

Bonus slides: NVIDIA GTX 680 (2012)

NVIDIA Kepler GK104 architecture



- 1 GHz clock
- Eight SMX cores per chip
- 8 x 192 = 1,536 SIMD mul-add ALUs
  = 3 TFLOPs
- Up to 512 interleaved warps per chip (16,384 CUDA threads/chip)
- TDP: 195 watts



NVIDIA Maxwell Architecture

2015

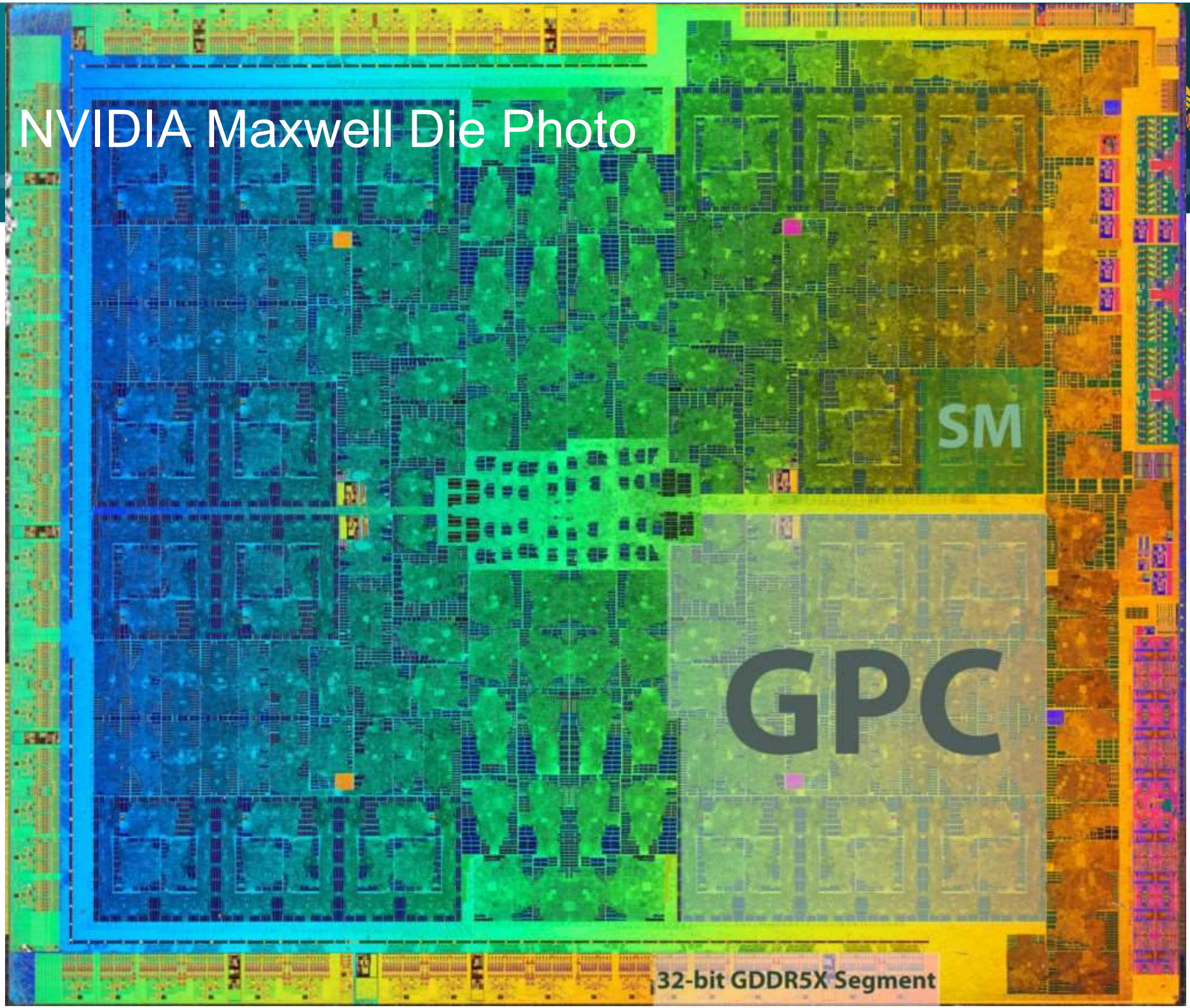
(compute capability 5.x)

GM107 (cc 5.0), ... (GTX 750Ti, ...)
GM204 (cc 5.2), ... (GTX 980, Titan X, ...)
GM20B (cc 5.3), ... (Tegra X1, ...)

Nintendo Switch/OLED (2017/2021)!

NVIDIA Maxwell Architecture (2015)

NVIDIA Maxwell Die Photo



Instruction Throughput

Instruction throughput numbers in CUDA C Programming Guide (Chapter 5.4)

	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

Maxwell (GM) Architecture

Multiprocessor: SMM (CC 5.x)

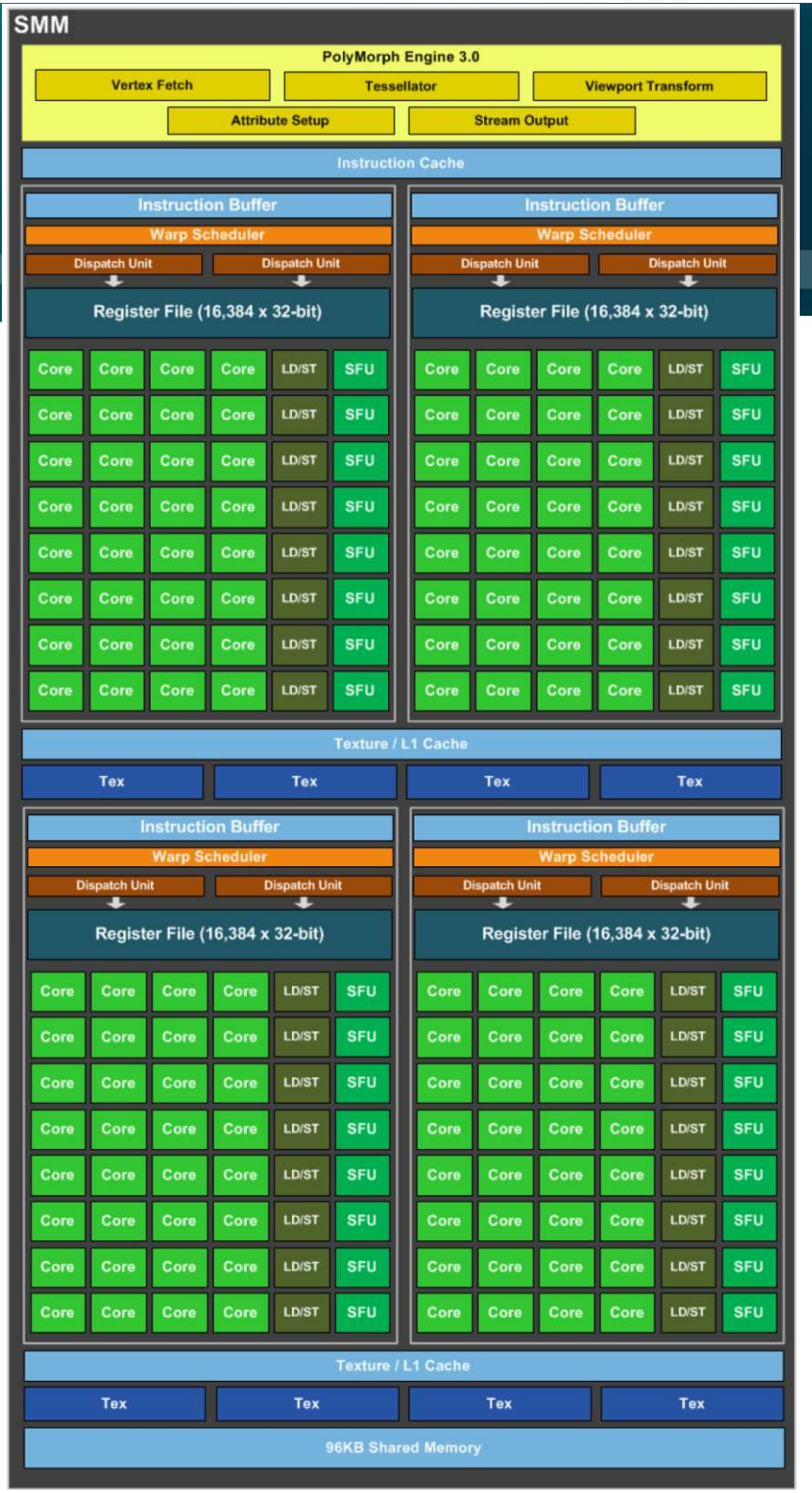
- 128 CUDA cores
- 4 DP units; 32 LD/ST units; 32 SFUs
- 8 texture units

4 partitions inside SMM

- 32 CUDA cores each
- 8 LD/ST units; 8 SFUs each
- Each has its own register file, warp scheduler, two dispatch units *(but cannot dual-issue ALU insts.!)*

Shared memory and L1 cache now separate!

- L1 cache shares with texture cache
- Shared memory is its own space



Maxwell (GM) Architecture

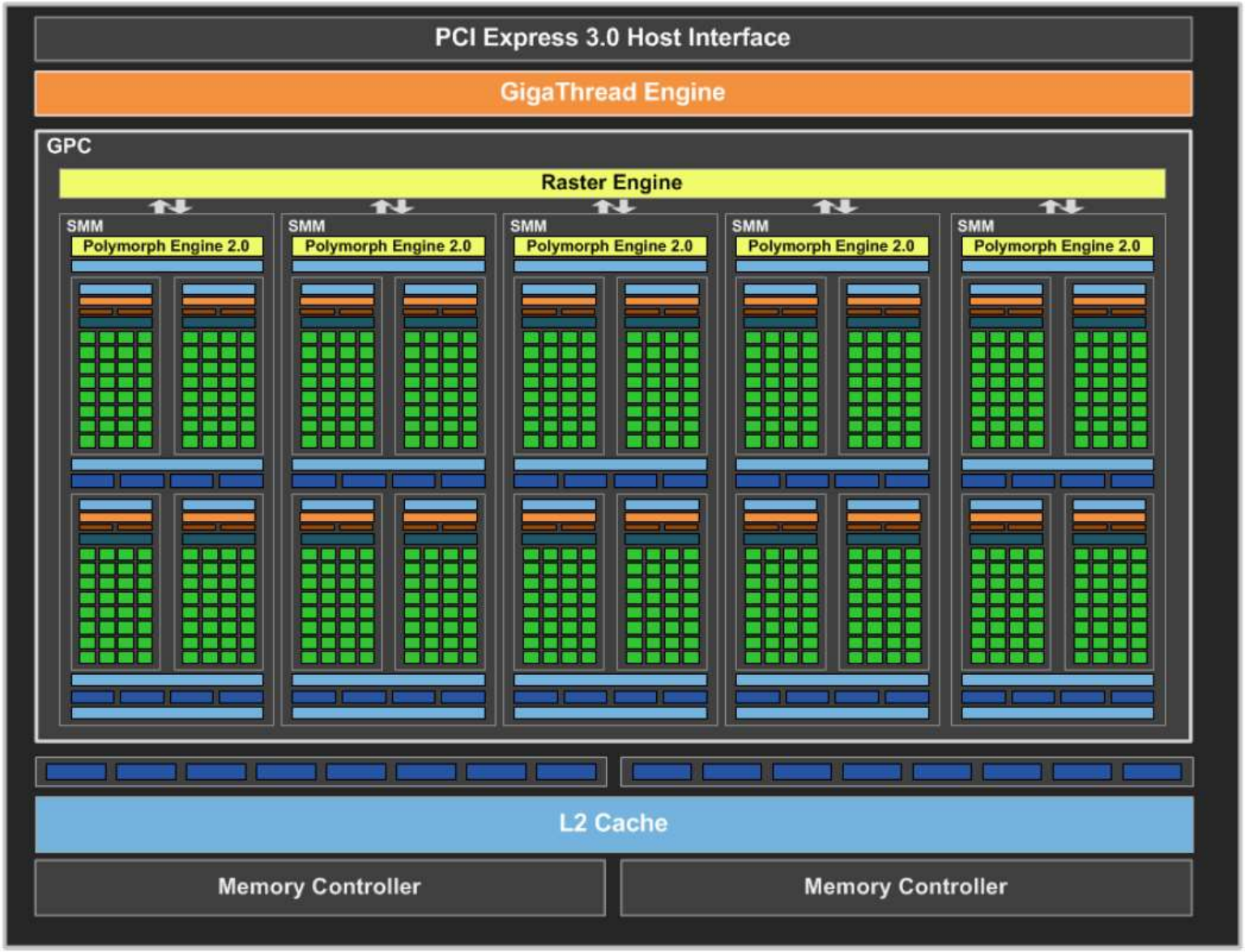
First gen.

GM107

(GTX 750Ti)

5 SMMs

(640 CUDA cores in total)



Maxwell (GM) Architecture

Second gen.

GM204

(GTX 980)

16 SMMs

(2048 CUDA cores in total)

4 GPCs of 4 SMMs

Maxwell (GM) vs. Kepler (GK) Architecture

GK107 vs. GM107

GPU	GK107 (Kepler)	GM107 (Maxwell)
CUDA Cores	384	640
Base Clock	1058 MHz	1020 MHz
GPU Boost Clock	N/A	1085 MHz
GFLOPs	812.5	1305.6
Texture Units	32	40
Texel fill-rate	33.9 Gigatexels/sec	40.8 Gigatexels/sec
Memory Clock	5000 MHz	5400 MHz
Memory Bandwidth	80 GB/sec	86.4 GB/sec
ROPs	16	16
L2 Cache Size	256KB	2048KB
TDP	64W	60W
Transistors	1.3 Billion	1.87 Billion
Die Size	118 mm^2	148 mm^2
Manufacturing Process	28-nm	28-nm

Maxwell (GM) vs. Kepler (GK) Architecture

GK107 vs. GM204

GPU	GeForce GTX 680 (Kepler)	GeForce GTX 980 (Maxwell)
SMs	8	16
CUDA Cores	1536	2048
Base Clock	1006 MHz	1126 MHz
GPU Boost Clock	1058 MHz	1216 MHz
GFLOPs	3090	4612[1]
Texture Units	128	128
Texel fill-rate	128.8 Gigatexels/sec	144.1 Gigatexels/sec
Memory Clock	6000 MHz	7000 MHz
Memory Bandwidth	192 GB/sec	224 GB/sec
ROPs	32	64
L2 Cache Size	512KB	2048KB
TDP	195 Watts	165 Watts
Transistors	3.54 billion	5.2 billion
Die Size	294 mm^2	398 mm^2
Manufacturing Process	28-nm	28-nm

NVIDIA Pascal Architecture

2016

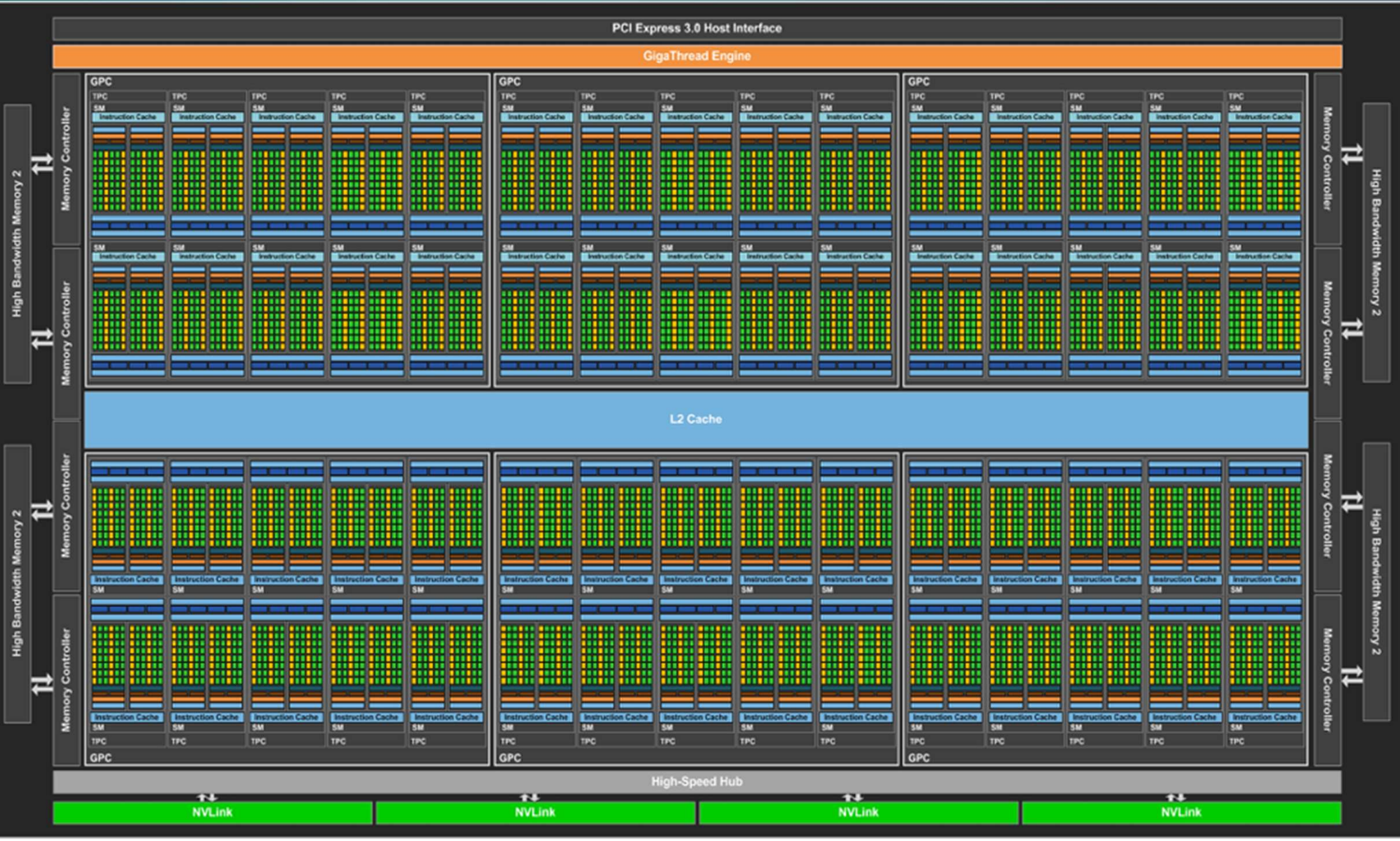
(compute capability 6.x)

GP100 (cc 6.0), ... (Tesla P100, ...)

(x=2,4,6,7,8) GP10x (cc 6.1), ... (GTX 1080, Titan X *Pascal*/Xp, Tesla P4/40, ...)

GM10B (cc 6.2), ... (Tegra X2, ...)

NVIDIA Pascal Architecture (2016)



Instruction Throughput

Instruction throughput numbers in CUDA C Programming Guide (Chapter 5.4)

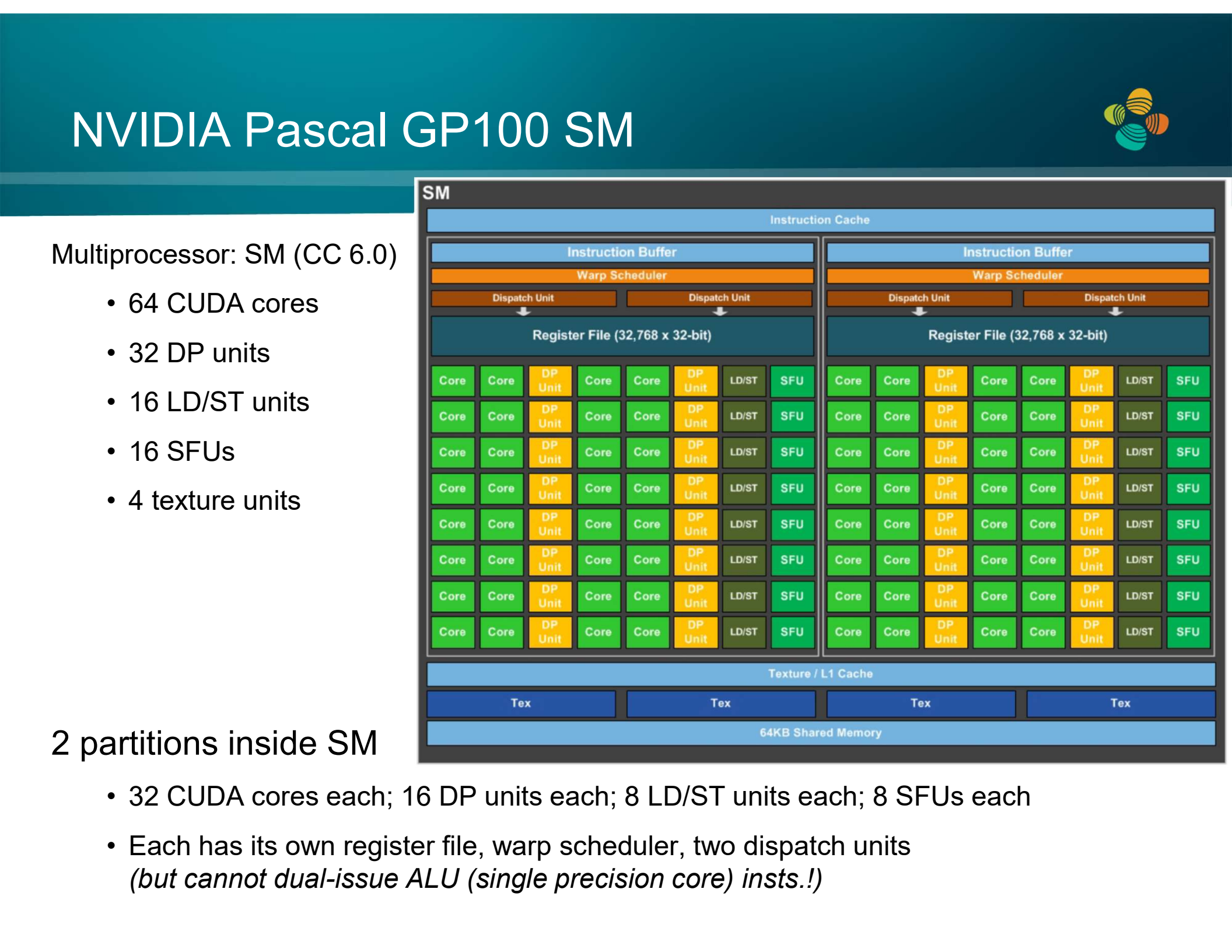
	Compute Capability										
	3.5, 3.7	5.0, 5.2	5.3	6.0	6.1	6.2	7.x	8.0	8.6	8.9	9.0
16-bit floating-point add, multiply, multiply-add	N/A		256	128	2	256	128	256 128 for __nv_bfloat16		128	256 128 for __nv_bfloat16
32-bit floating-point add, multiply, multiply-add	192	128		64	128		64		128		
64-bit floating-point 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 Pascal GP100 SM



Multiprocessor: SM (CC 6.0)

- 64 CUDA cores
- 32 DP units
- 16 LD/ST units
- 16 SFUs
- 4 texture units

2 partitions inside SM

- 32 CUDA cores each; 16 DP units each; 8 LD/ST units each; 8 SFUs each
- Each has its own register file, warp scheduler, two dispatch units
  (but cannot dual-issue ALU (single precision core) insts.!)

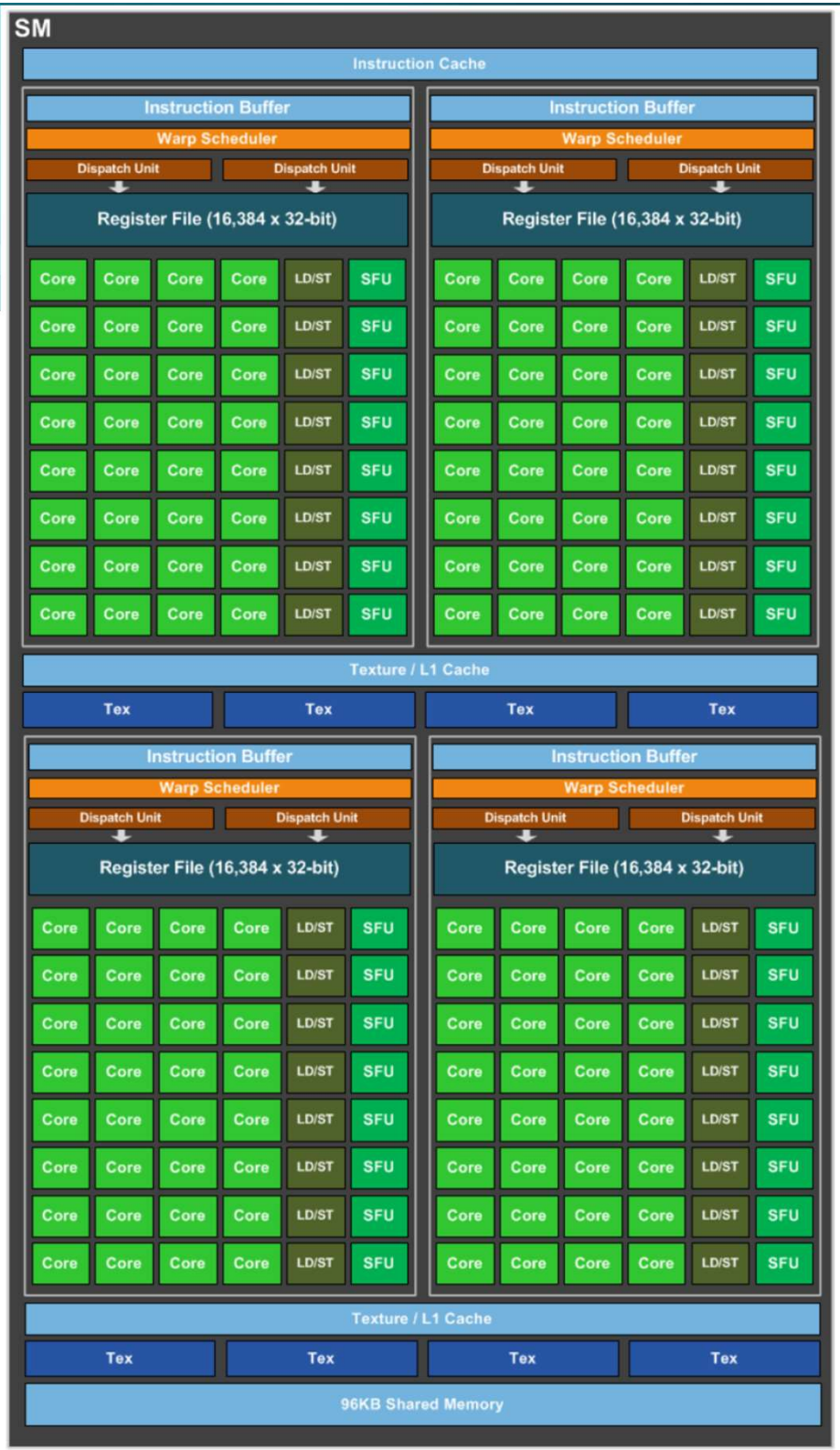
NVIDIA Pascal GP104 SM

Multiprocessor: SM (CC 6.1/6.2)

- 128 CUDA cores
- 32 LD/ST units
- 32 SFUs
- 8 texture units

4 partitions inside SM

- 32 CUDA cores; 8 LD/ST units; 8 SFUs
- Each has its own register file, warp scheduler, two dispatch units *(but cannot dual-issue ALU insts.!)*



NVIDIA Pascal Architecture (2016)

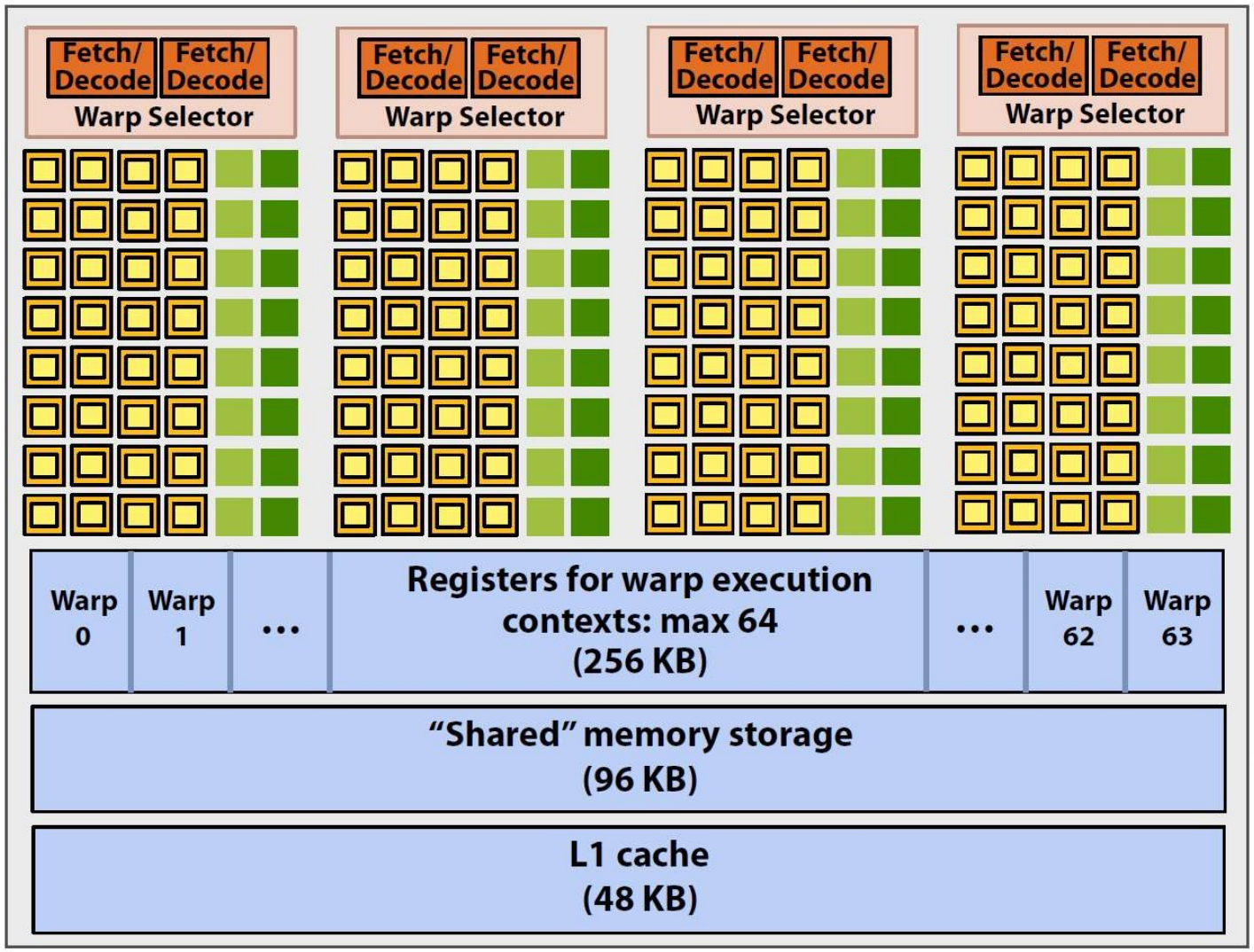
Total chip capacity on Tesla P100 (GP100)

- 56 SMs
  - 64 CUDA cores / SM = 3,584 CUDA cores in total
  - 32 DP units / SM = 1,792 DP units in total
- 28 TPCs (2 SMs per TPC)
- 6 GPCs

Maximum capacity would be 60 SMs and 30 TPCs

NVIDIA GTX 1080 (2016)

This is one NVIDIA Pascal GP104 streaming multi-processor (SM) unit



= SIMD functional unit, control shared across 32 units (1 MUL-ADD per clock)

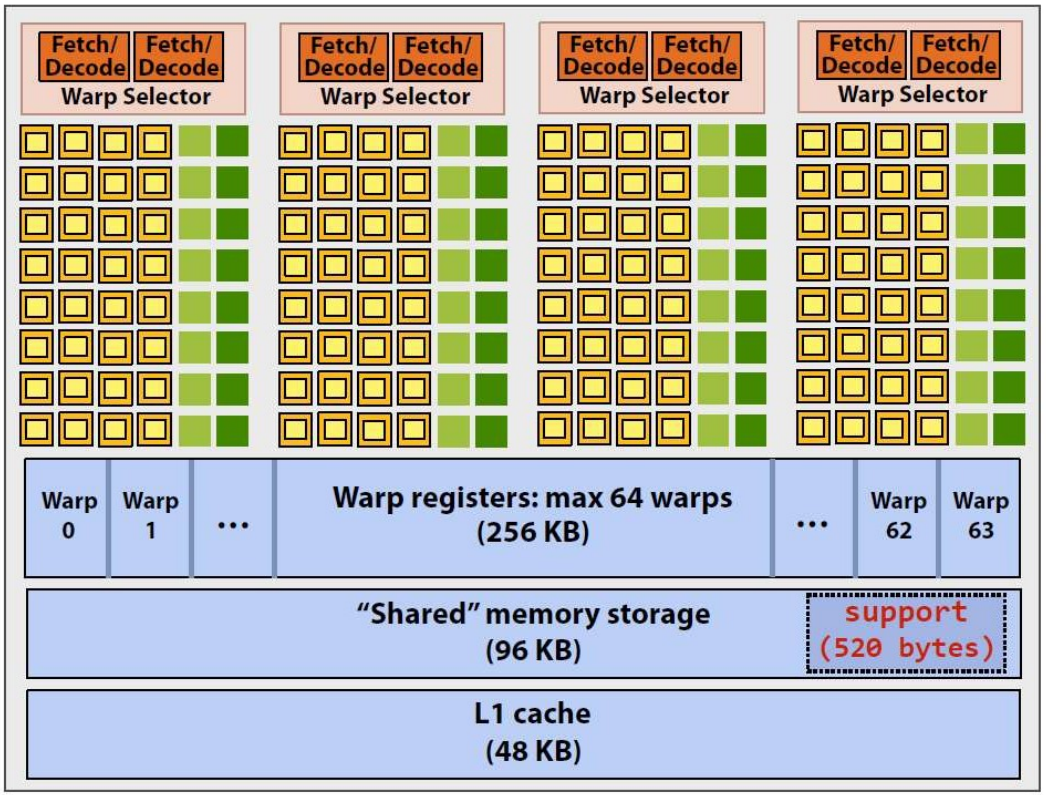
= load/store

= SIMD special function unit (sin, cos, etc.)

SM resource limits:

- Max warp execution contexts: 64 (2,048 total CUDA threads)
- 96 KB of shared memory

Running a single thread block on a SM "core"




```
#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) ***

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 Architecture (2017/2018)

Instruction Throughput

Instruction throughput numbers in CUDA C Programming Guide (Chapter 5.4)

	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

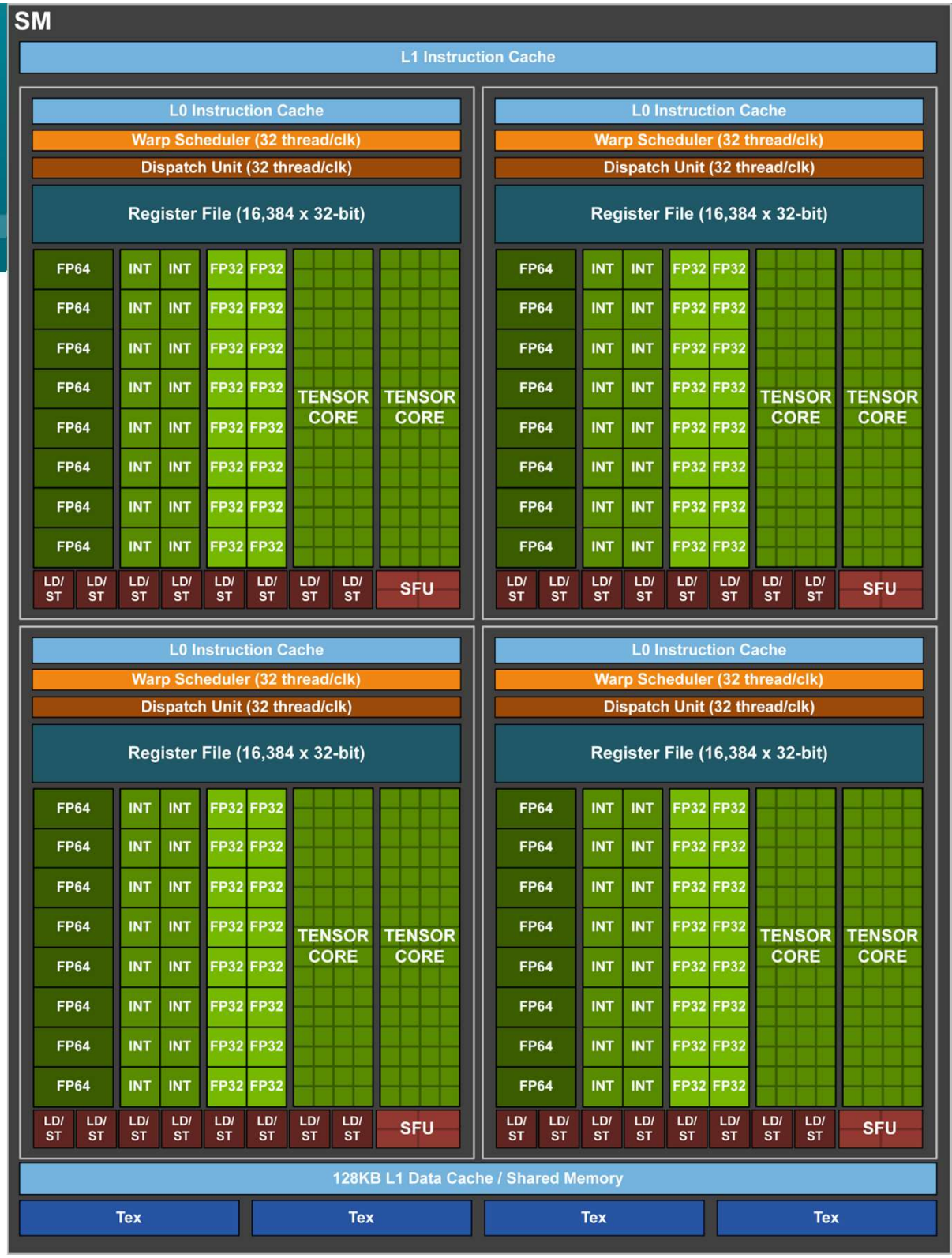
NVIDIA Volta SM

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)

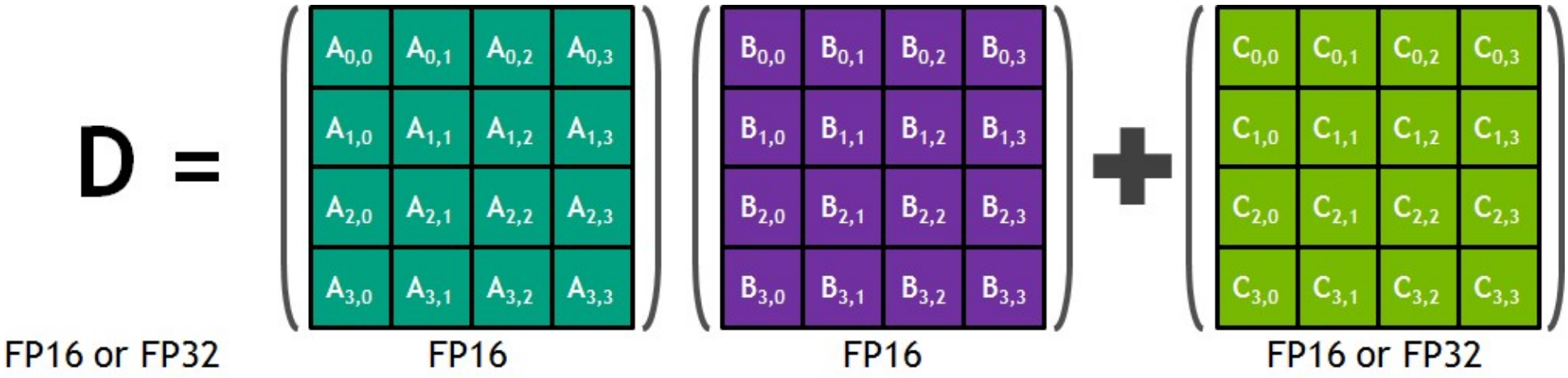
4 partitions inside SM

- 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



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

Kepler – Volta Specs

Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1455 MHz
Peak FP32 TFLOP/s*	5.04	6.8	10.6	15
Peak FP64 TFLOP/s*	1.68	.21	5.3	7.5
Peak Tensor Core TFLOP/s*	NA	NA	NA	120
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm^2	601 mm^2	610 mm^2	815 mm^2
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN

Turing (vs. Pascal)

Apart from RT cores, Volta and Turing are very similar
(and both have compute capability 7.x: Volta: 7.0, Turing: 7.5)

GPU Features	GeForce GTX 1080	GeForce RTX 2080	Quadro P5000	Quadro RTX 5000
Architecture	Pascal	Turing	Pascal	Turing
GPCs	4	6	4	6
TPCs	20	23	20	24
SMs	20	46	20	48
CUDA Cores / SM	128	64	128	64
CUDA Cores / GPU	2560	2944	2560	3072
Tensor Cores / SM	NA	8	NA	8
Tensor Cores / GPU	NA	368	NA	384
RT Cores	NA	46	NA	48
		TU104		TU104

Thank you.