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`

Next Lectures

Lecture 8: Tue, Sep 17 (make-up lecture; 14:30 – 15:45)

Lecture 9: Thu, Sep 19

no lecture on Sep 23 !

Lecture 10: Thu, Sep 26 (Quiz #1)

Quiz #1: Sep 26

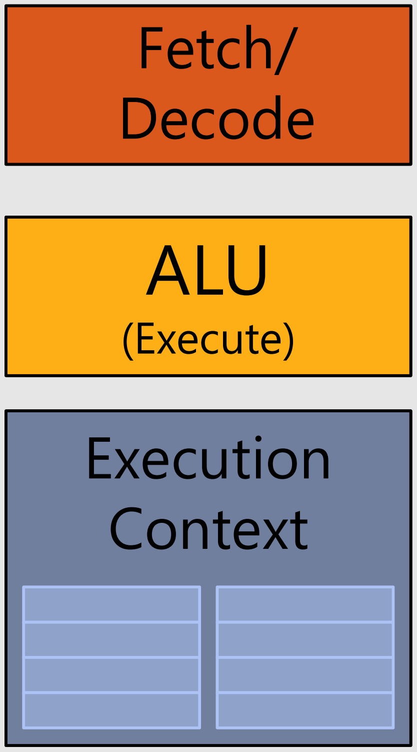
Organization

- 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

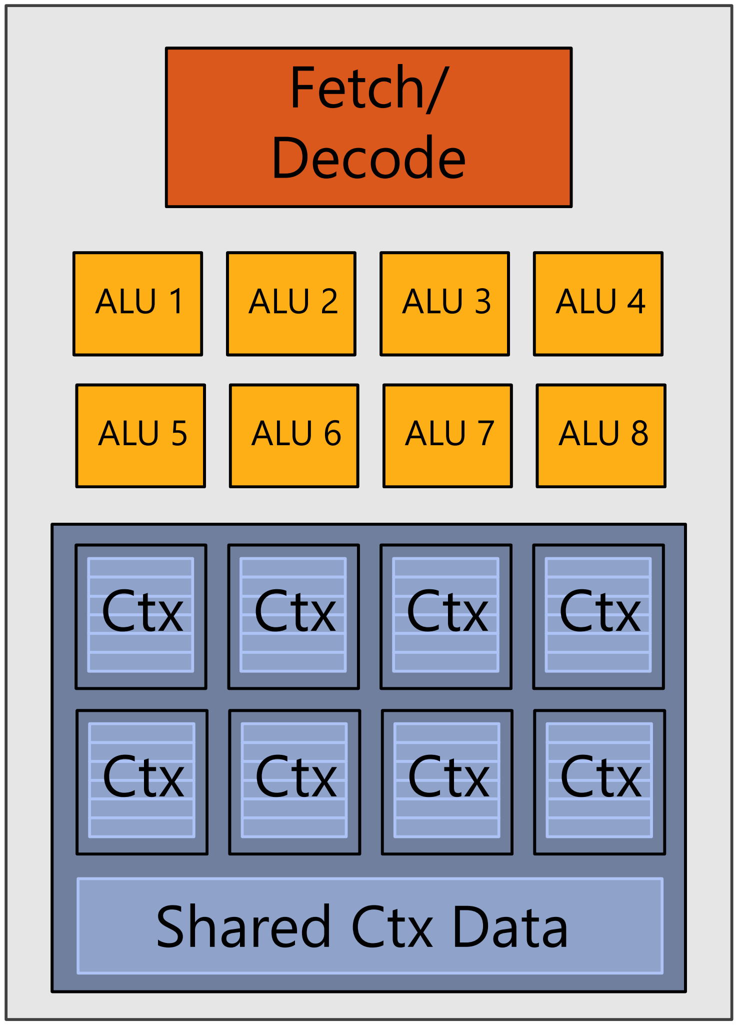
Idea #1: Slim down



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)



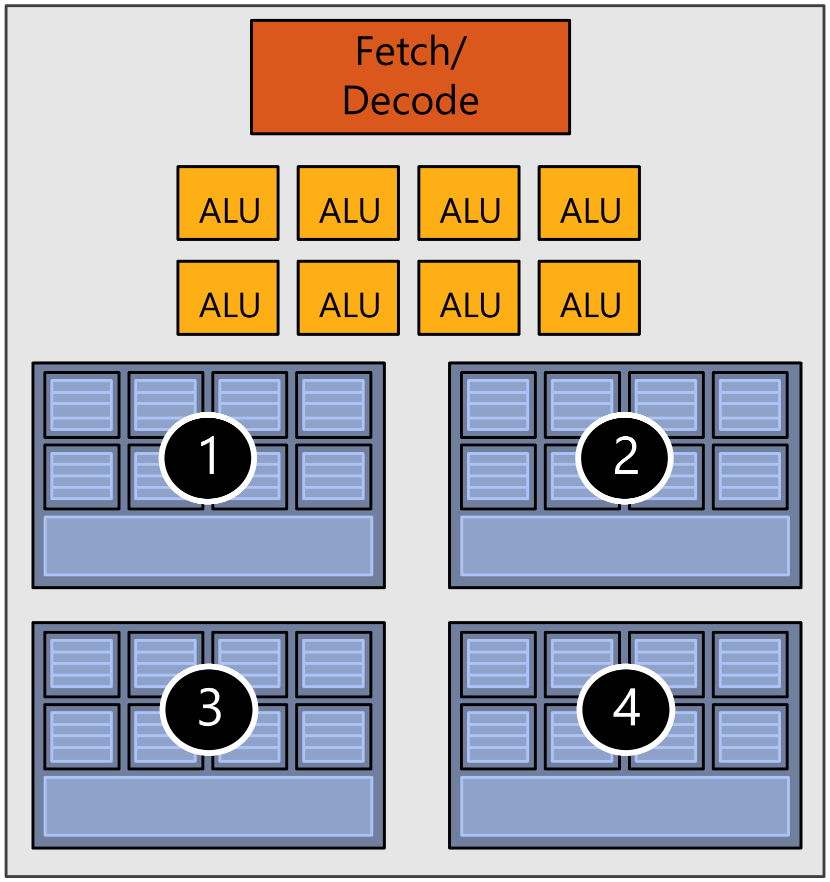
Idea #2:

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

SIMD processing

(or SIMT, SPMD)

Idea #3: Store multiple group contexts

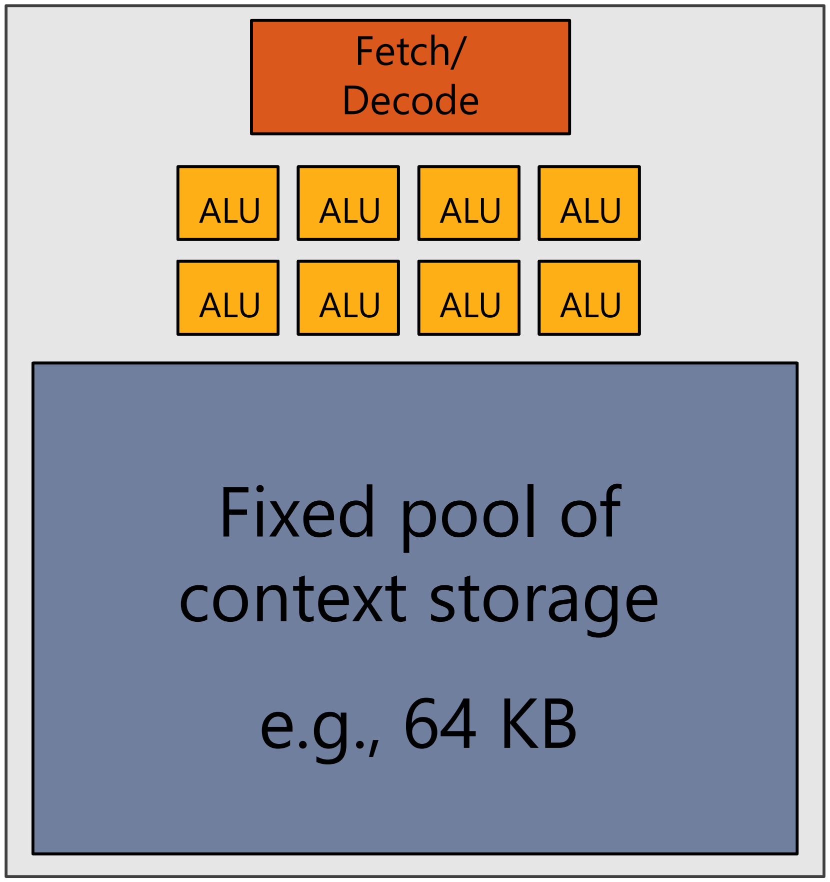


Idea #3:

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

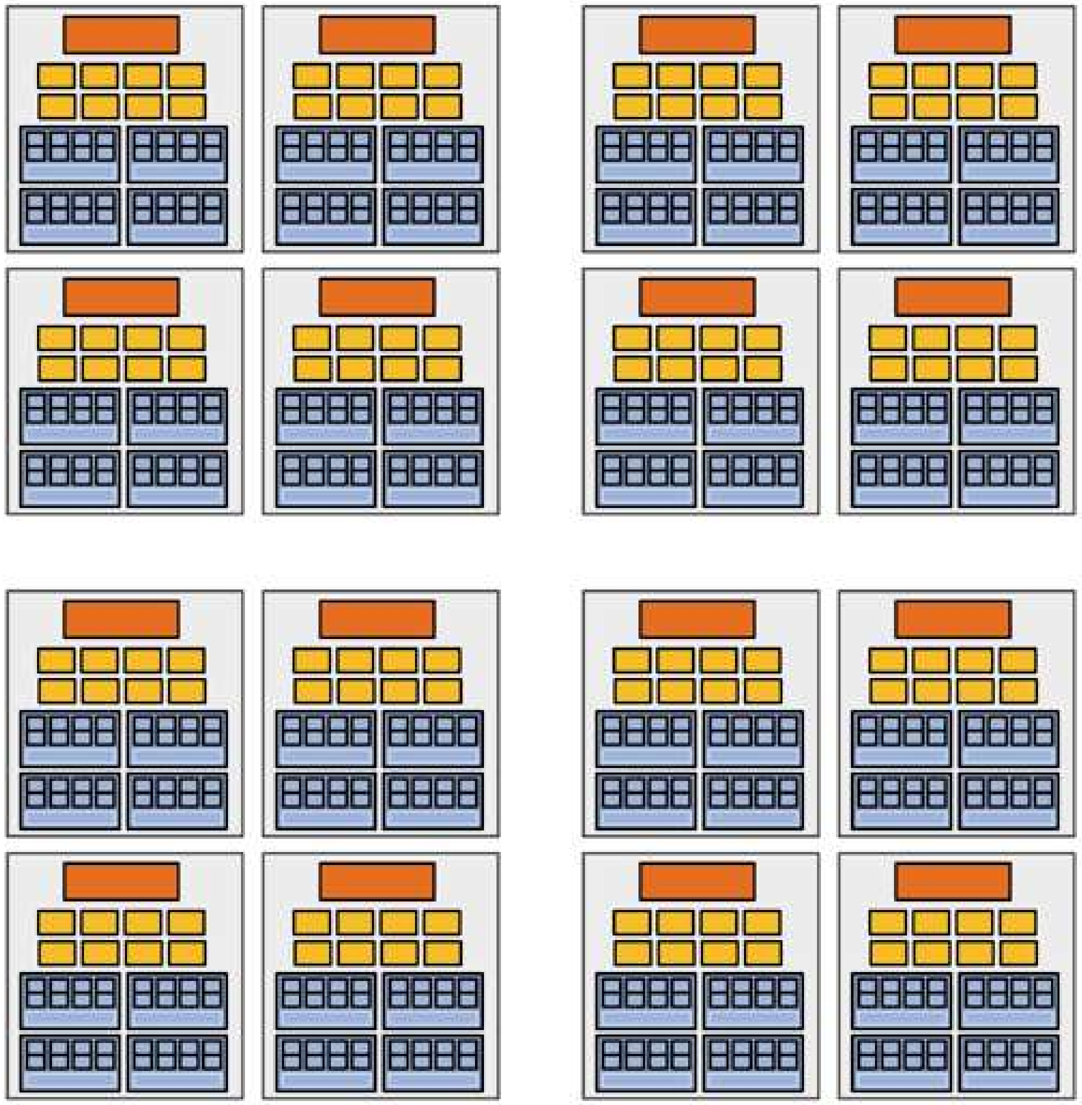
8 mul-add [mad] ALUs per core
(8*16 = **128** total)

16 simultaneous
instruction streams

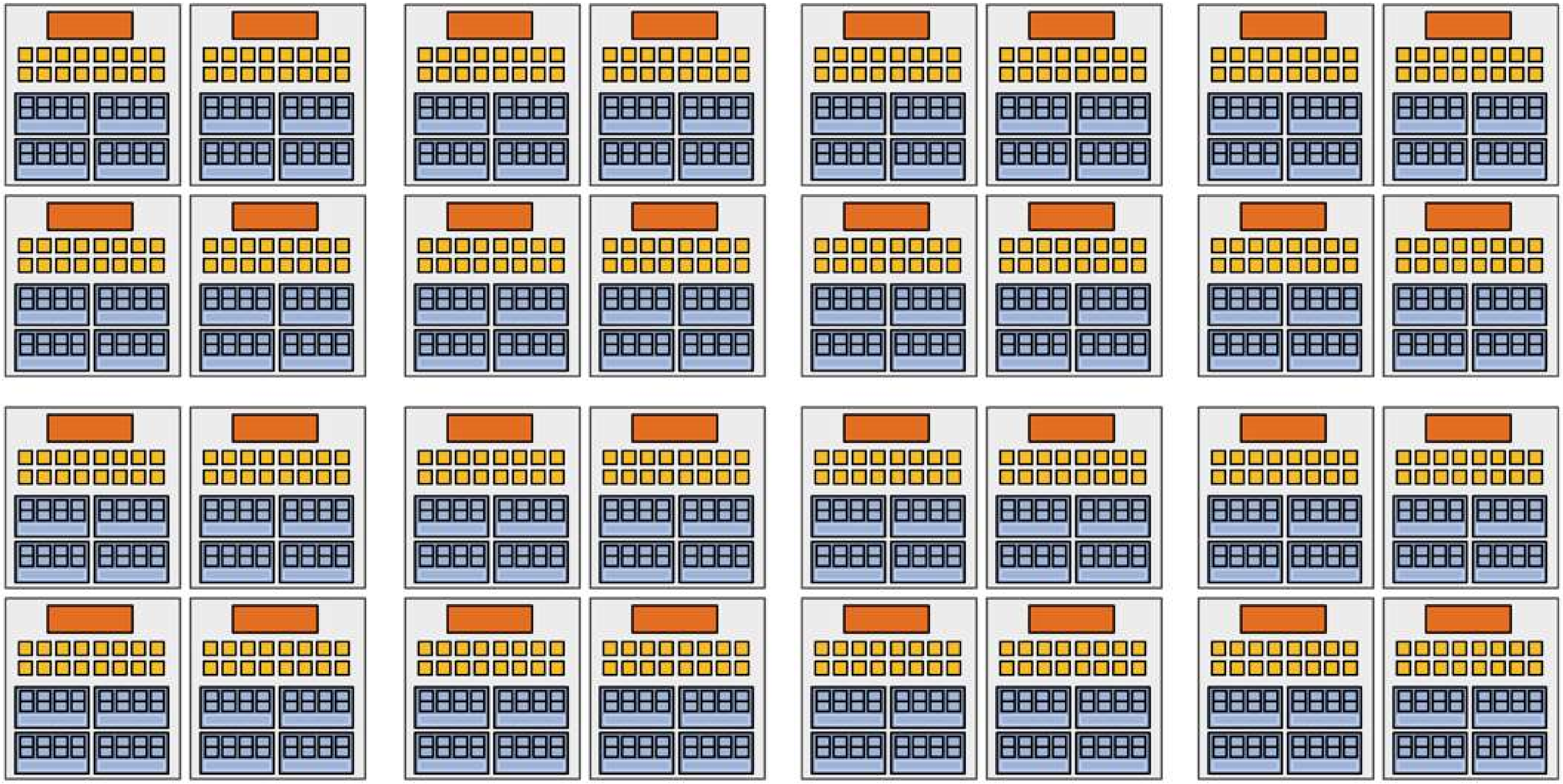
64 (4*16) concurrent (but interleaved) instruction streams

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

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



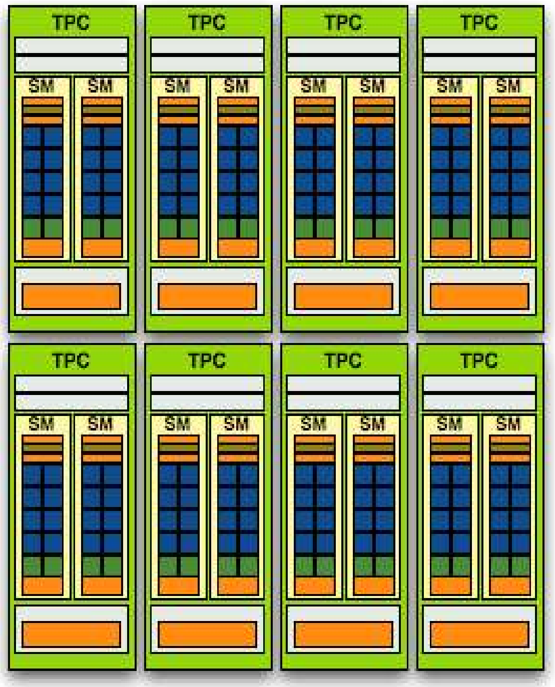
“Enthusiast” GPU (Some time ago :)



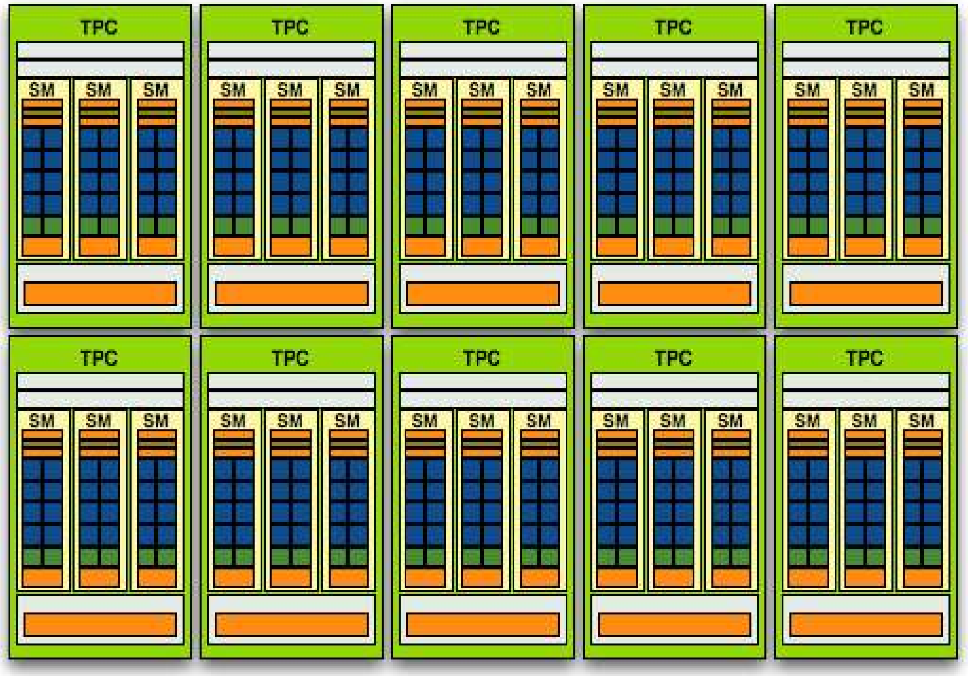
32 cores, 16 ALUs per core (512 total) = 1 TFLOP (@ 1 GHz)

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

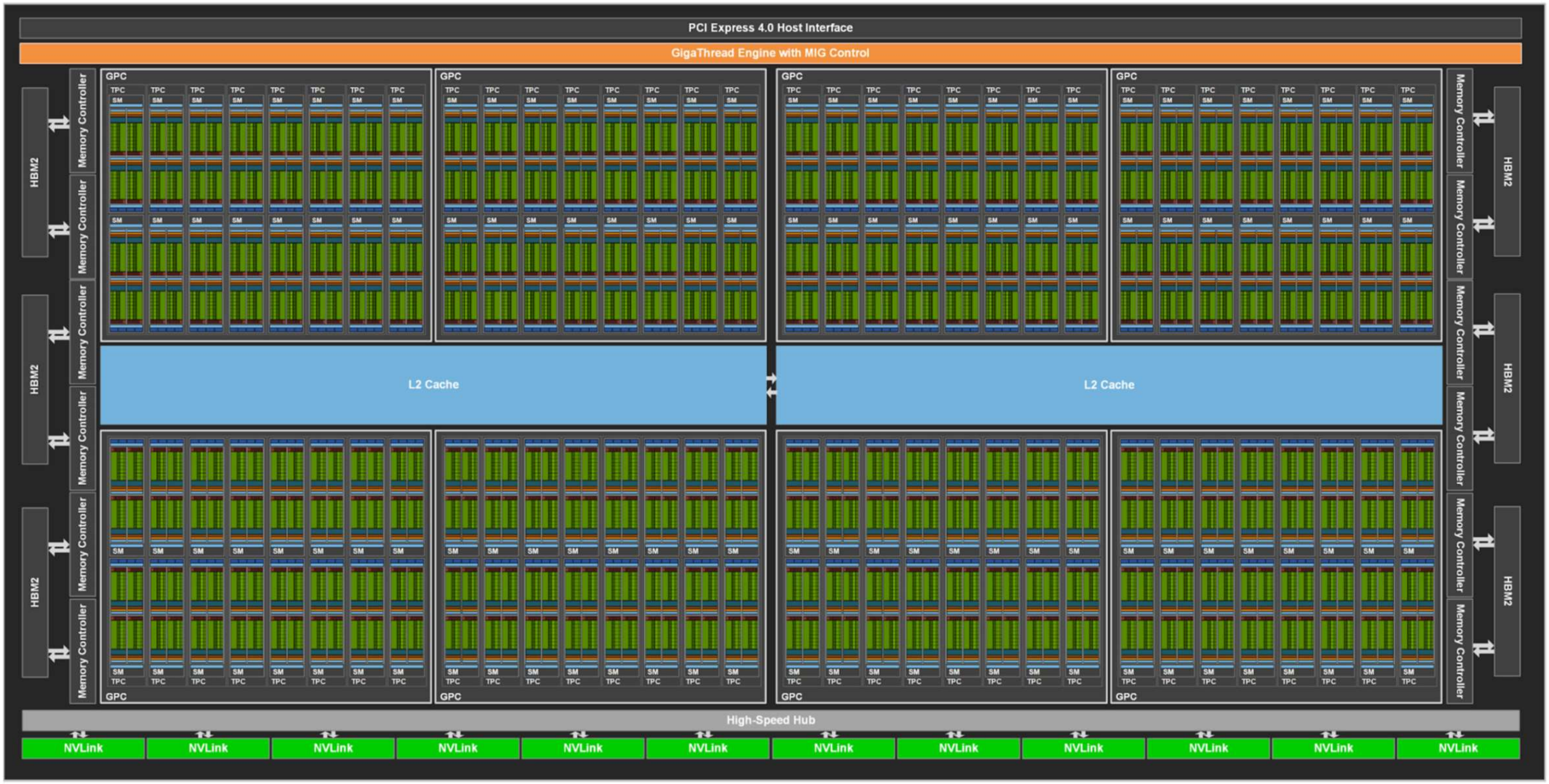


GT200

Courtesy AnandTech

NVIDIA Ampere GA100 Architecture (2020)

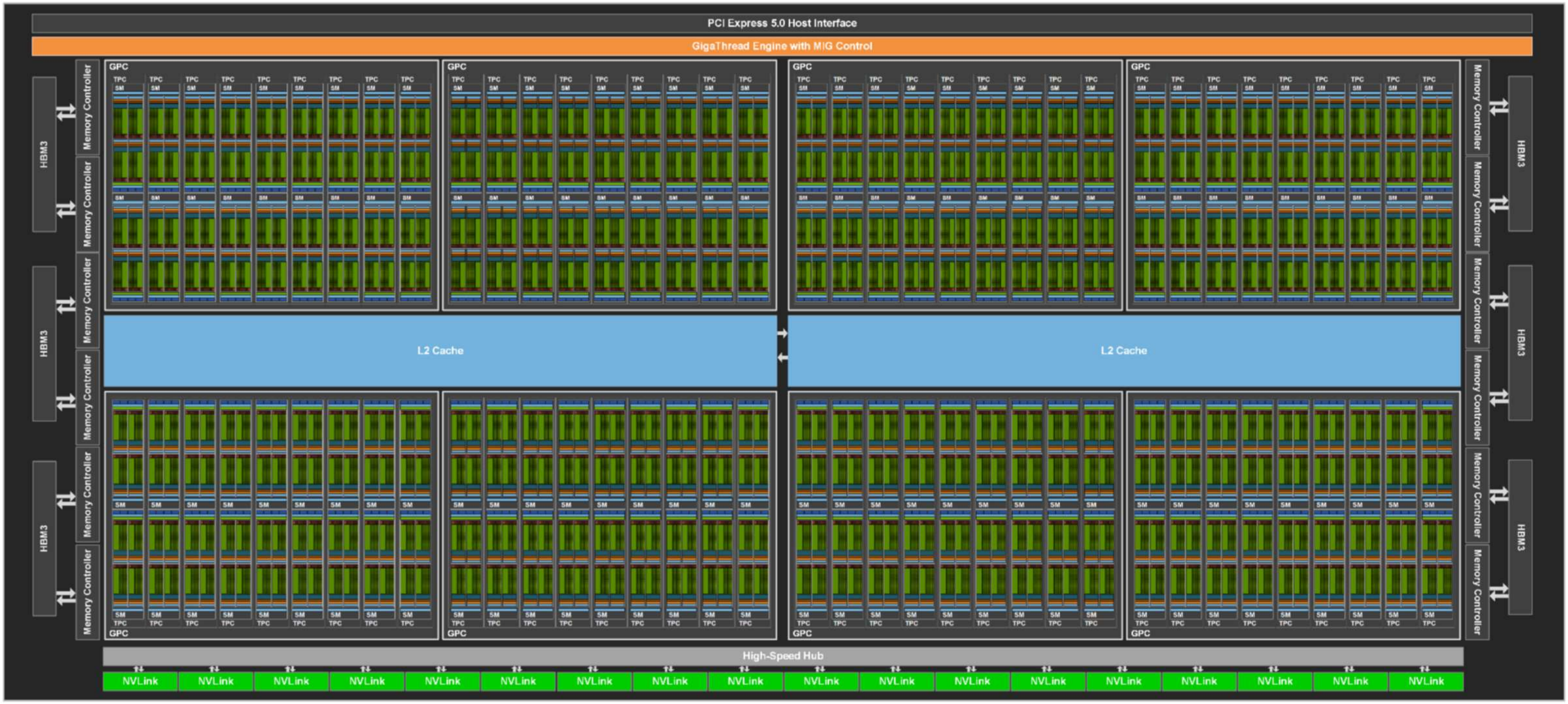
GA 100 (A100 Tensor Core GPU) Full GPU: 128 SMs (in 8 GPCs/64 TPCs)



PCI Express 4.0 Host Interface
GigaThread Engine with MIG Control
GPC
TPC
SM
L2 Cache
Memory Controller
HBM2
High-Speed Hub
NVLink

NVIDIA Hopper GH100 Architecture (2022)

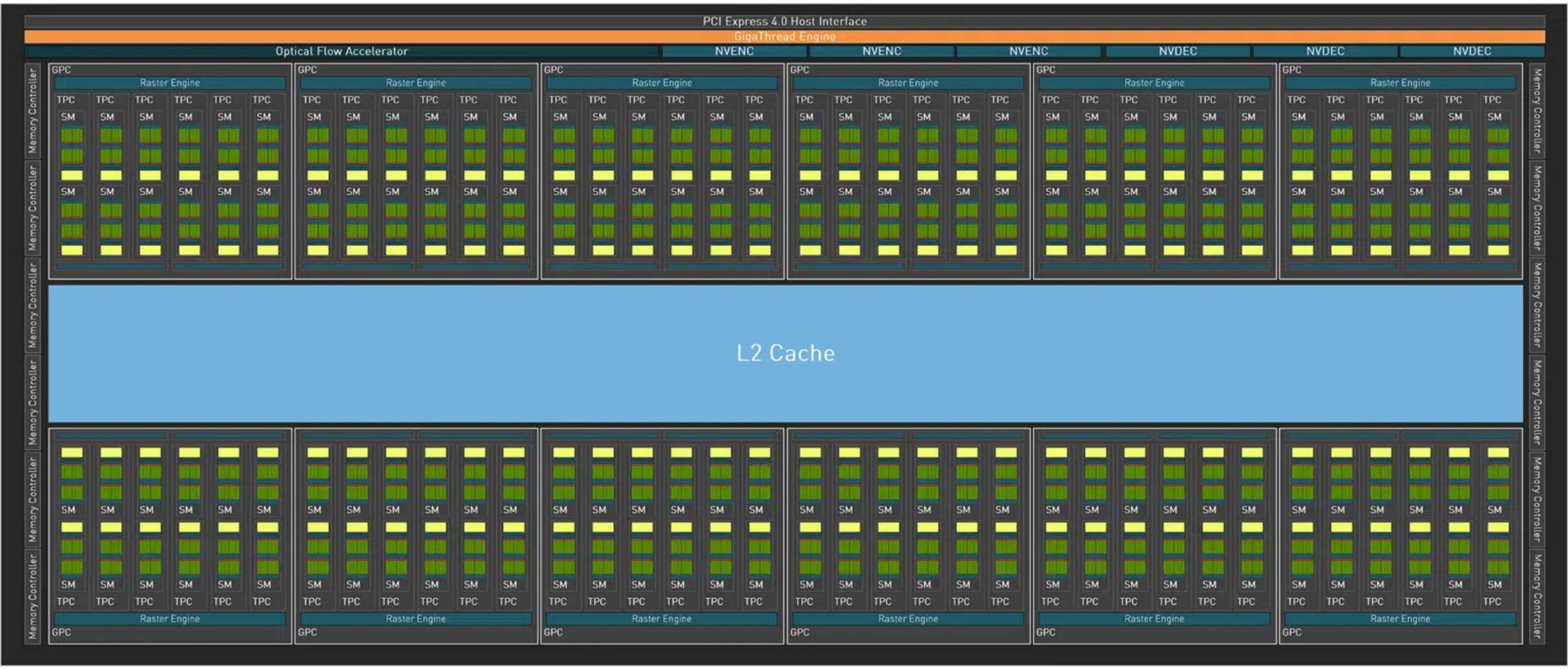
GH 100 (H100 Tensor Core GPU)   Full GPU: 144 SMs (in 8 GPCs/72 TPCs)



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*

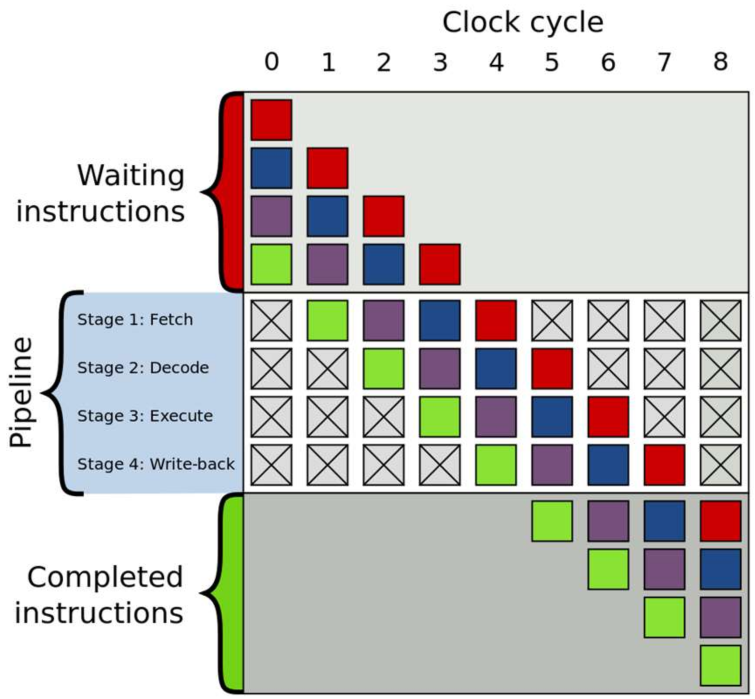
- GB200/GB202, GB20x, ...?
  (RTX 5080/5090, GB200 NVL72, HGX B100/200, ...?)

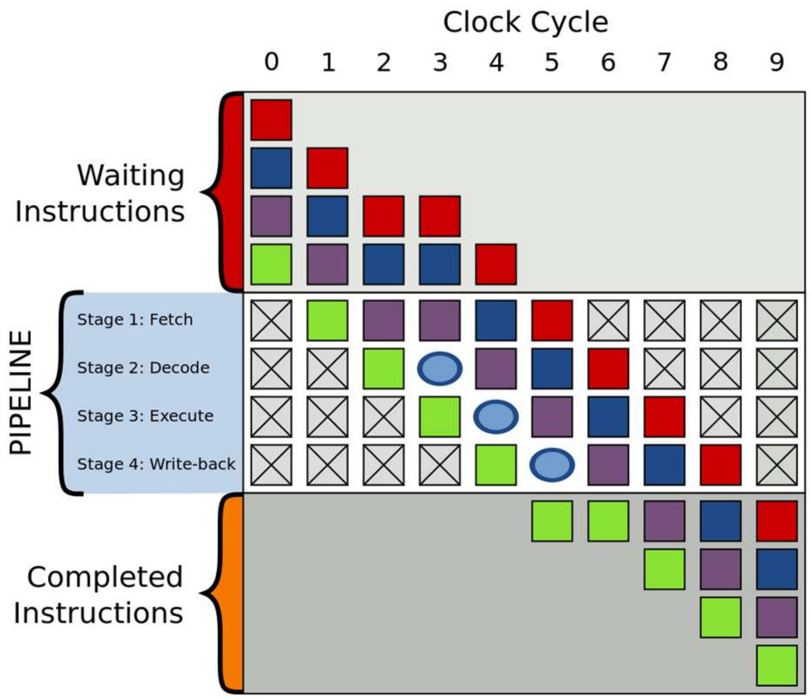
see `https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units`
and `https://en.wikipedia.org/wiki/CUDA`

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)?

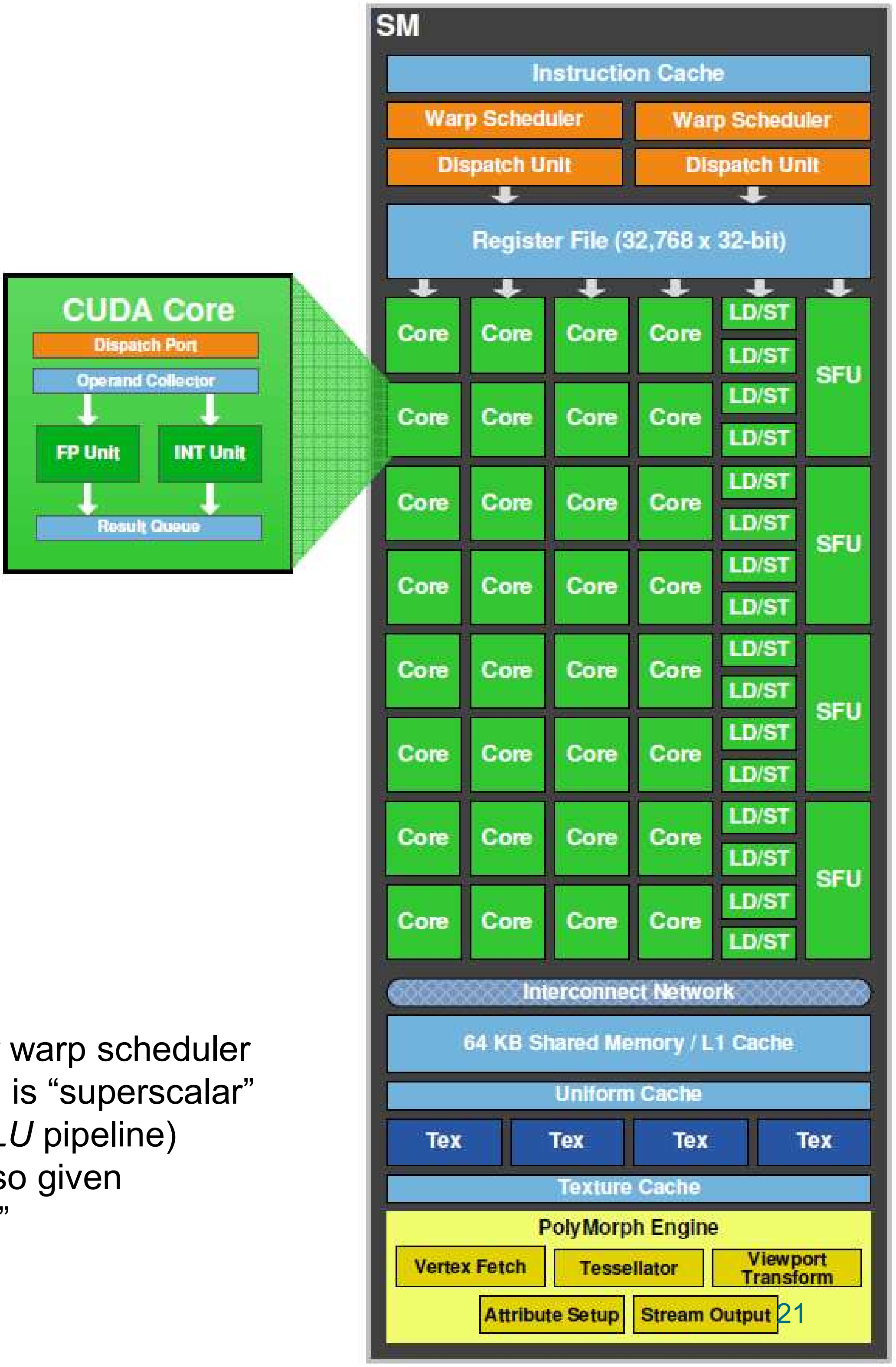
- "Scalar" pipeline (*CPI=1.0*): **TLP sufficient (if enough warps); can exploit ILP** (next instruction either from different warp, or from same warp)
- "Superscalar" (*CPI<1.0*) pipeline: dispatch more than one instruction per cycle, (#dispatchers > #warp schedulers): **need ILP!**

(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)

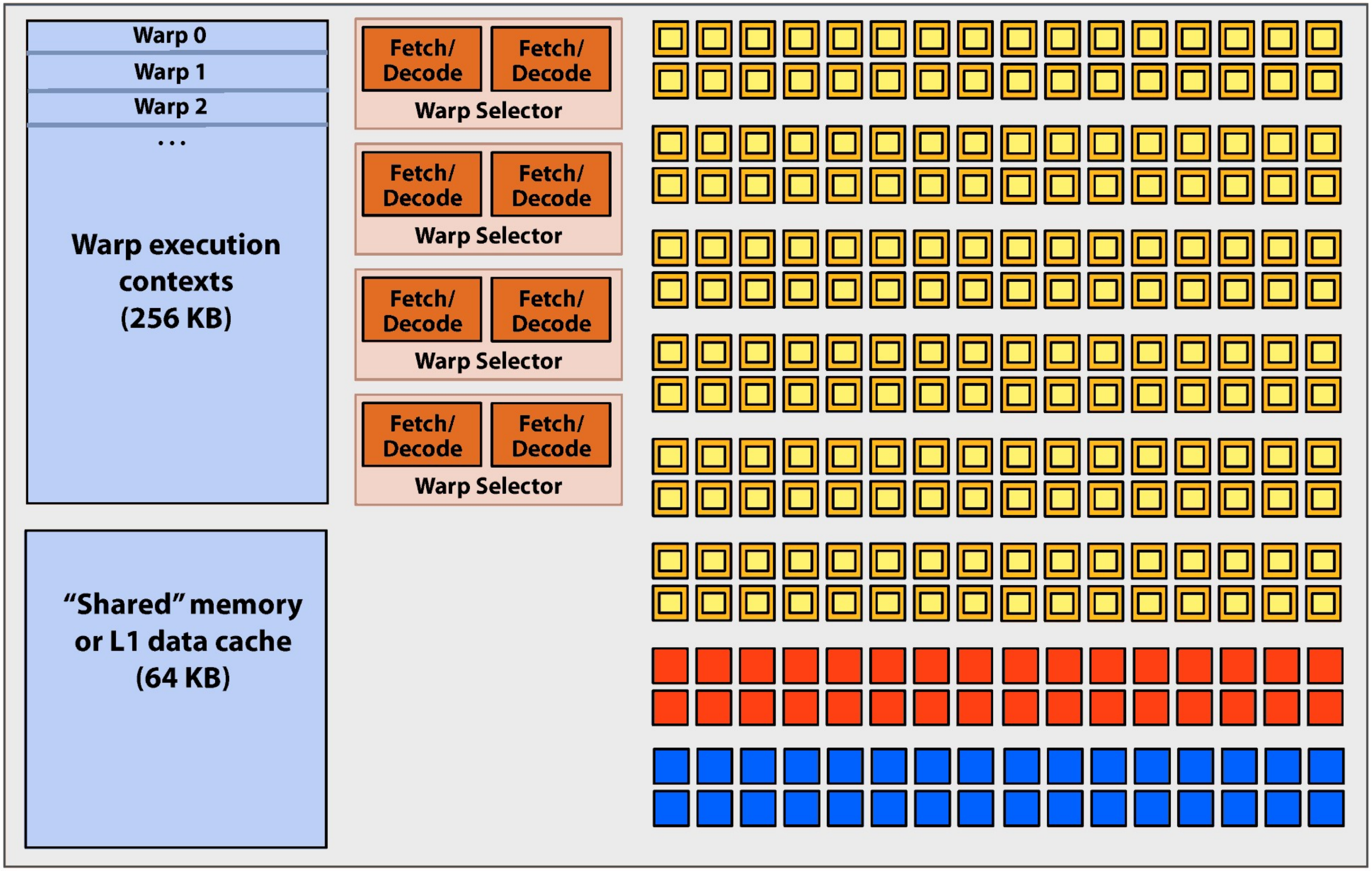


Details later...
Ignore less important subtleties...
GF100 has two warp schedulers, not one,
and each 32-thread instruction is executed
over two clock cycles, not one, etc.

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

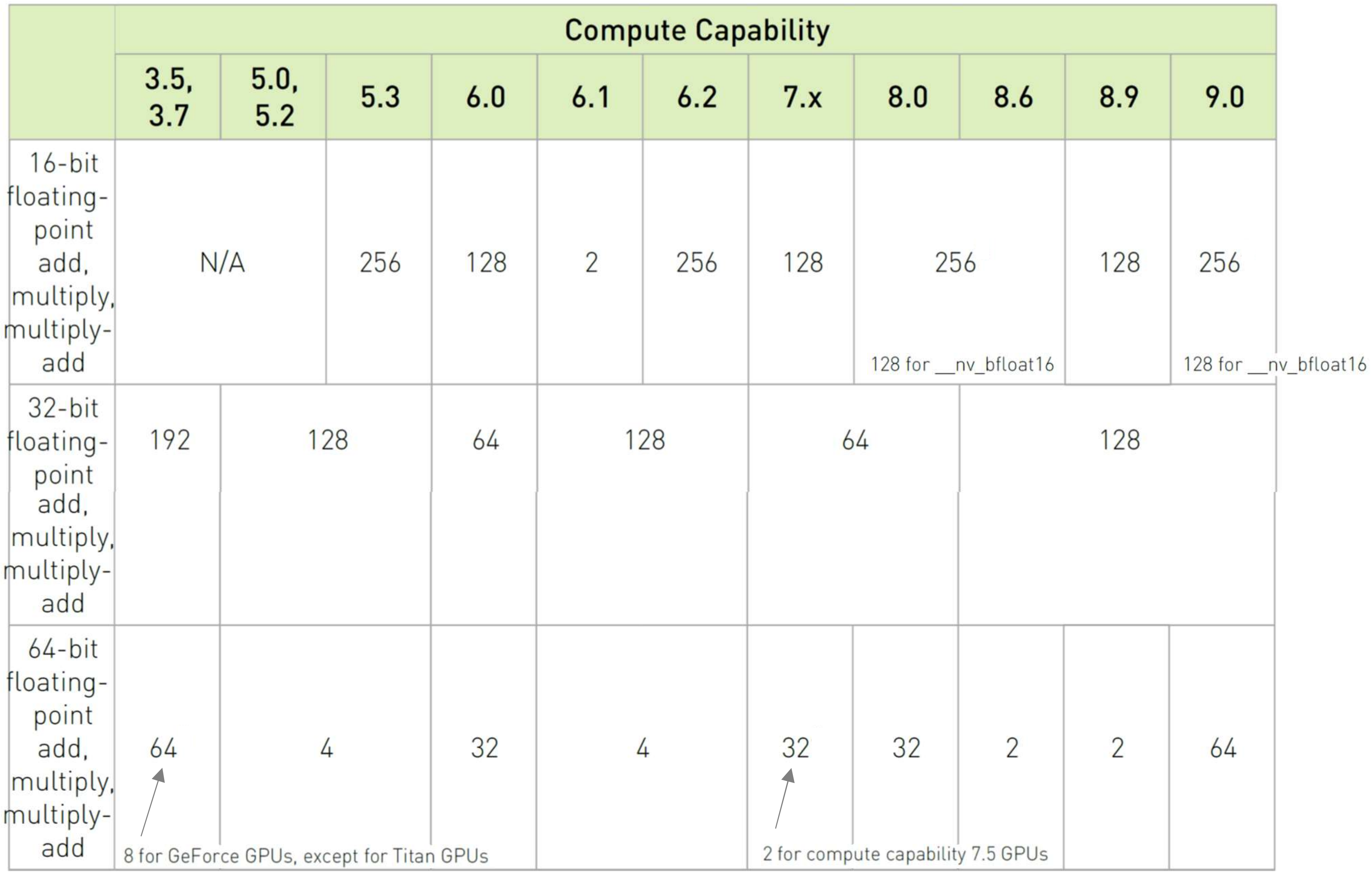
NVIDIA Kepler GK104 architecture SMX unit (one "core")



🟨	= 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

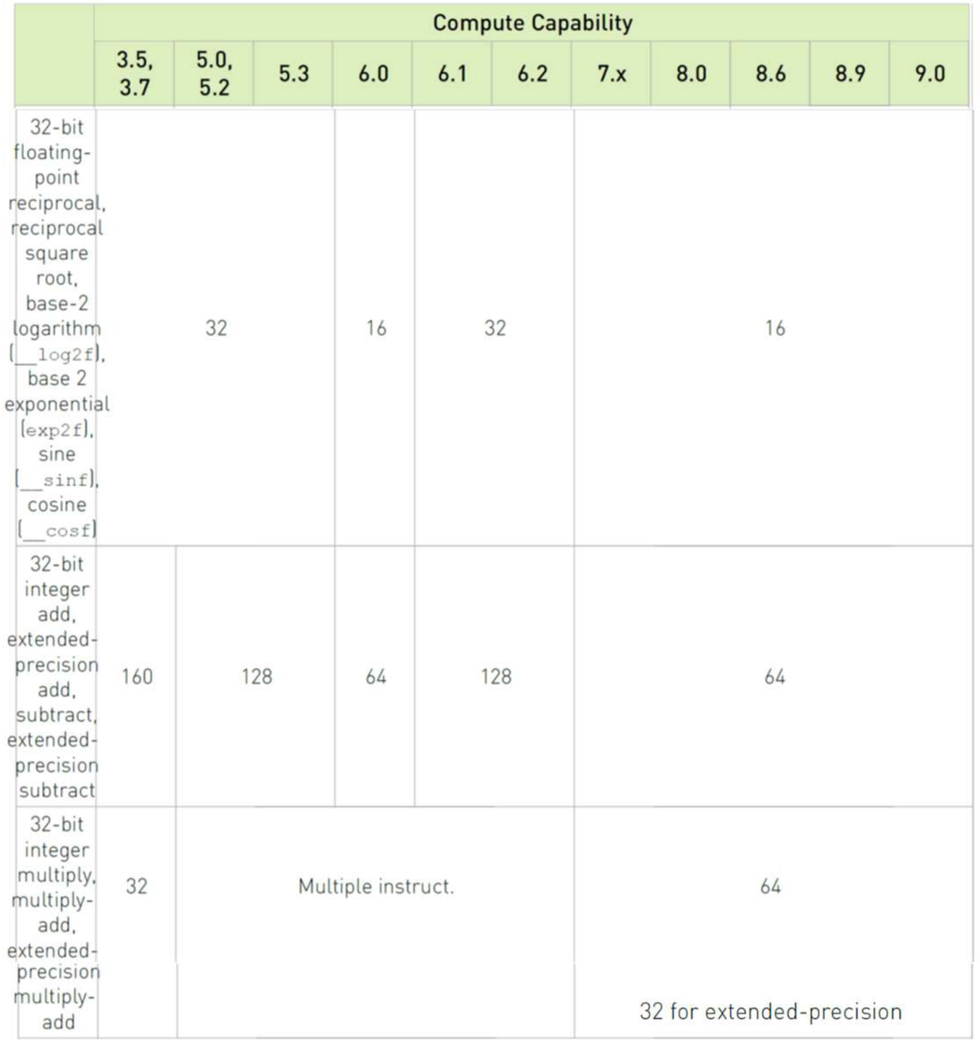
Instruction throughput numbers in CUDA C Programming Guide (Chapter 8.4)



	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	32	32	16	32	32	16	16	16	16	16
32-bit integer add, extended-precision add, subtract, extended-precision subtract	160	128	128	64	128	128	64	64	64	64	64
32-bit integer multiply, multiply-add, extended-precision multiply-add	32	Multiple instruct.	Multiple instruct.	Multiple instruct.	Multiple instruct.	Multiple instruct.	64 32 for extended-precision	64 32 for extended-precision	64 32 for extended-precision	64 32 for extended-precision	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

Thank you.