King Abdullah University of Science and Technology

KAUST

CS 380 - GPU and GPGPU Programming
Lecture 9: GPU Architecture, Pt. 7

Markus Hadwiger, KAUST

Reading Assignment #5 (until Oct 6)

Read (required):

- Programming Massively Parallel Processors book (4th edition), **Chapter 2** (*Heterogeneous data parallel computing*)
- CUDA NVCC documentation [NVCC is the CUDA compiler] (currently CUDA 13.0.1, Sep 2, 2025):

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

  Read Chapters 1 – 4, and Chapter 6 (GPU Compilation); get an overview of the rest

Quiz #1: Oct 6

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

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, ...); [Nintendo Switch]
  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, 8.8]: 2020

- GA100, GA102, GA104, GA106, ...; [Nintendo Switch 2]
  (A100, RTX 3070, 3080, 3090 (Ti), RTX A6000, ...)

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

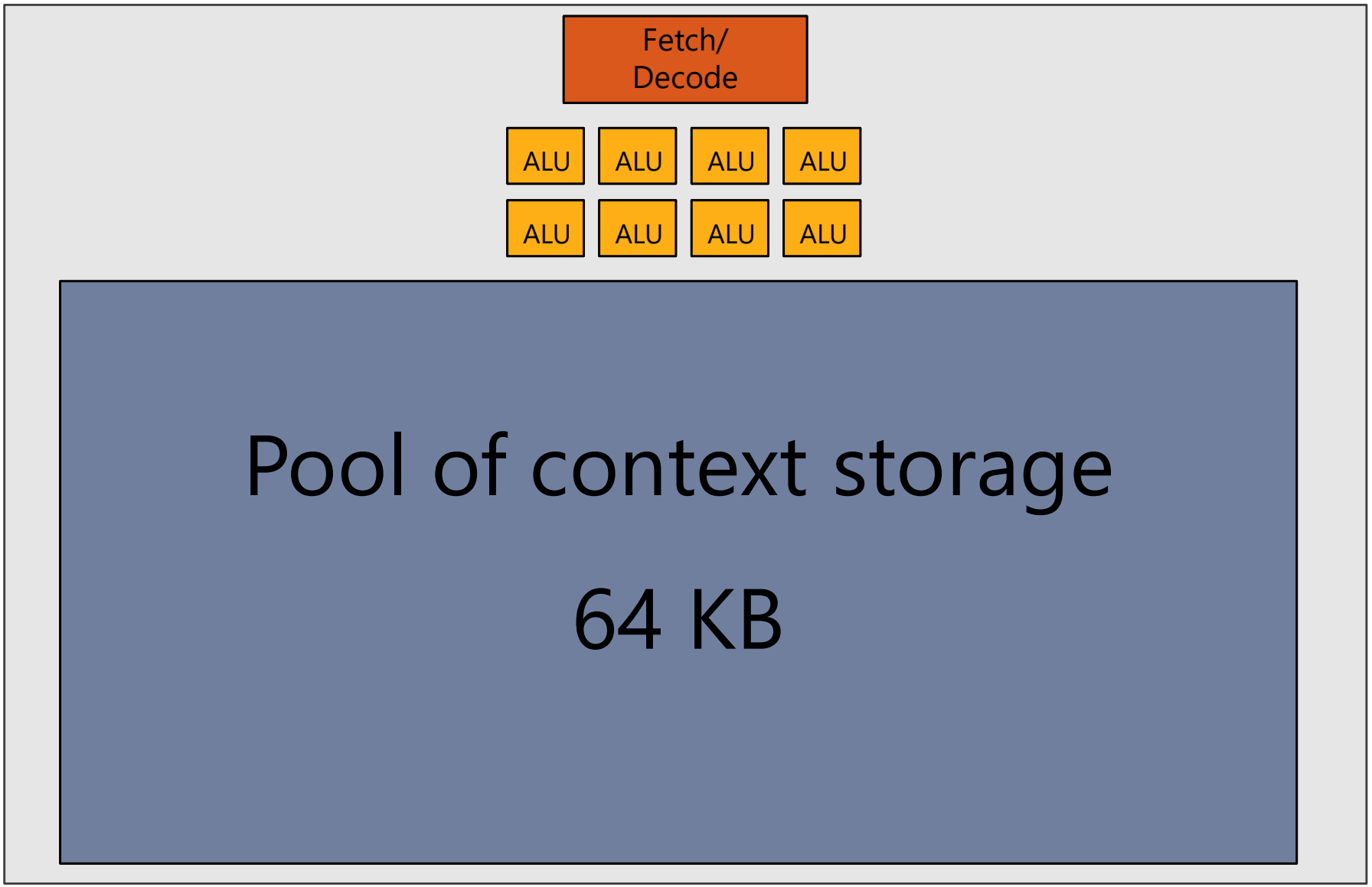
- GH100, AD102/103/104/106/107, ...
  (H100, H200, GH200, L20, L40, L40S, L2, L4,
  RTX 4080 (12/16 GB), RTX 4090, RTX 6000 (Ada), ...)

Blackwell [CC 10.0, 10.1(→11.0), 10.3, 12.0, 12.1]: 2024/2025

- GB100, GB200, GB202/203/205/206/207, G10, ...
  (RTX 5080/5090, HGX B200/B300, GB200/GB300 NVL72,
  RTX 4000/5000/6000 PRO Blackwell, B40, ...)

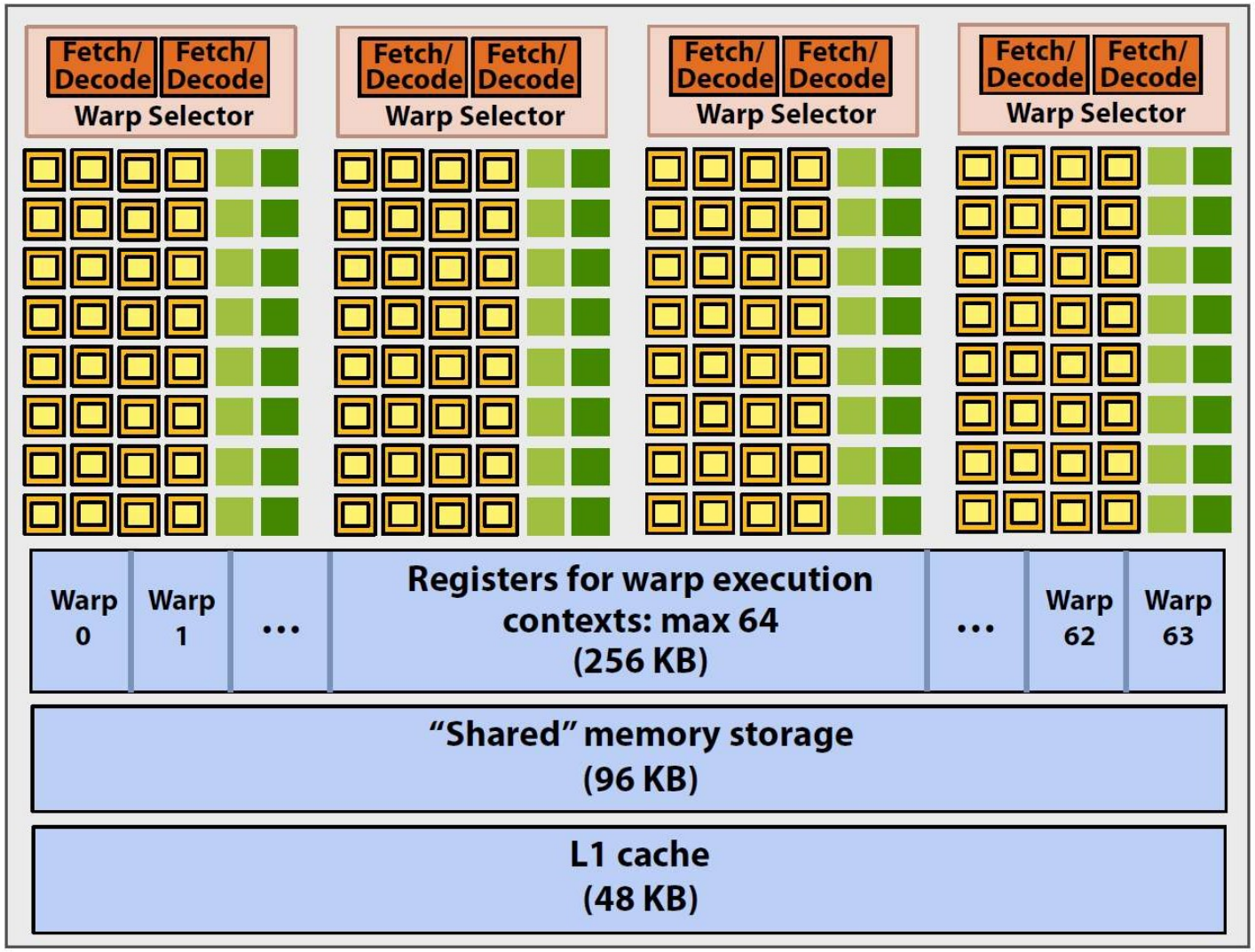
see `https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units`
and `https://en.wikipedia.org/wiki/CUDA`

General GPU Architecture (for one SM)



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)

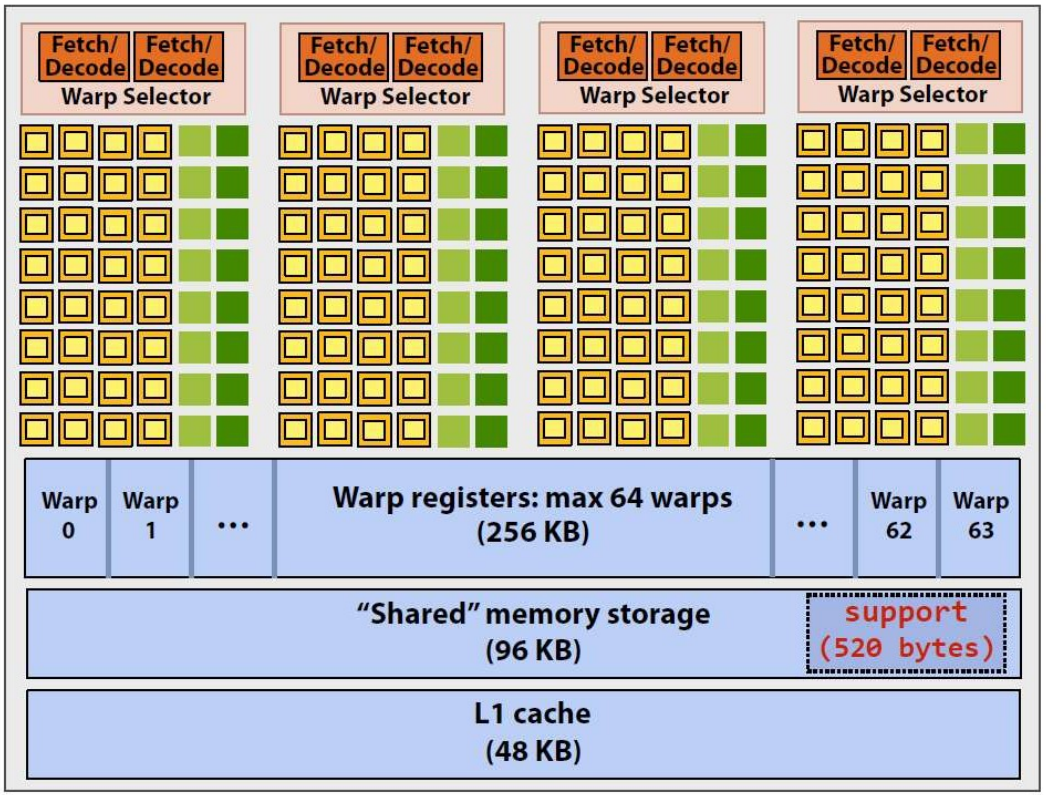
= load/store

= 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"



this is GP104 (2016)

```
#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)** * (but here: only 1 ALU instr. !)

NVIDIA Blackwell SM

CC 10.3 SM (GB300 Blackwell Ultra)

- 128 FP32/INT32 cores
- 64(?) FP64 cores
- 4x 5th gen tensor cores
- Tensor Memory Accelerator (TMA)
- ++ thread block clusters, DPX insts., FP8, NVFP4, 256 KB Tensor Memory (TMEM), needs 4 warps = warp group for full TMEM access (1 warp/partition)

4 partitions inside SM

- 32 FP32/INT32 cores
- 8x LD/ST units each
- 1x 5th gen tensor core
- 64 KB Tensor Memory (TMEM)
- Each has: warp scheduler, dispatch unit, 16K register file

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)?

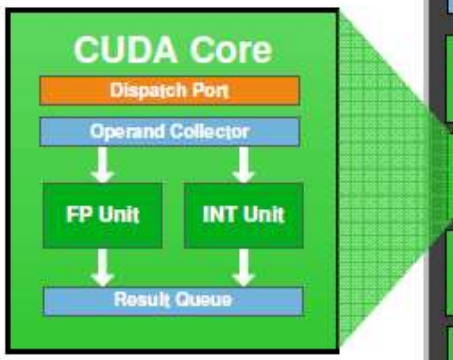
- "Scalar" pipeline (*CPI=1.0*): **TLP sufficient (if enough warps); can exploit ILP** (next instruction either from different warp, or from same warp)
- "Superscalar" (*CPI<1.0*) pipeline: dispatch more than one instruction per cycle, (#dispatchers > #warp schedulers): **need ILP!**

(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)

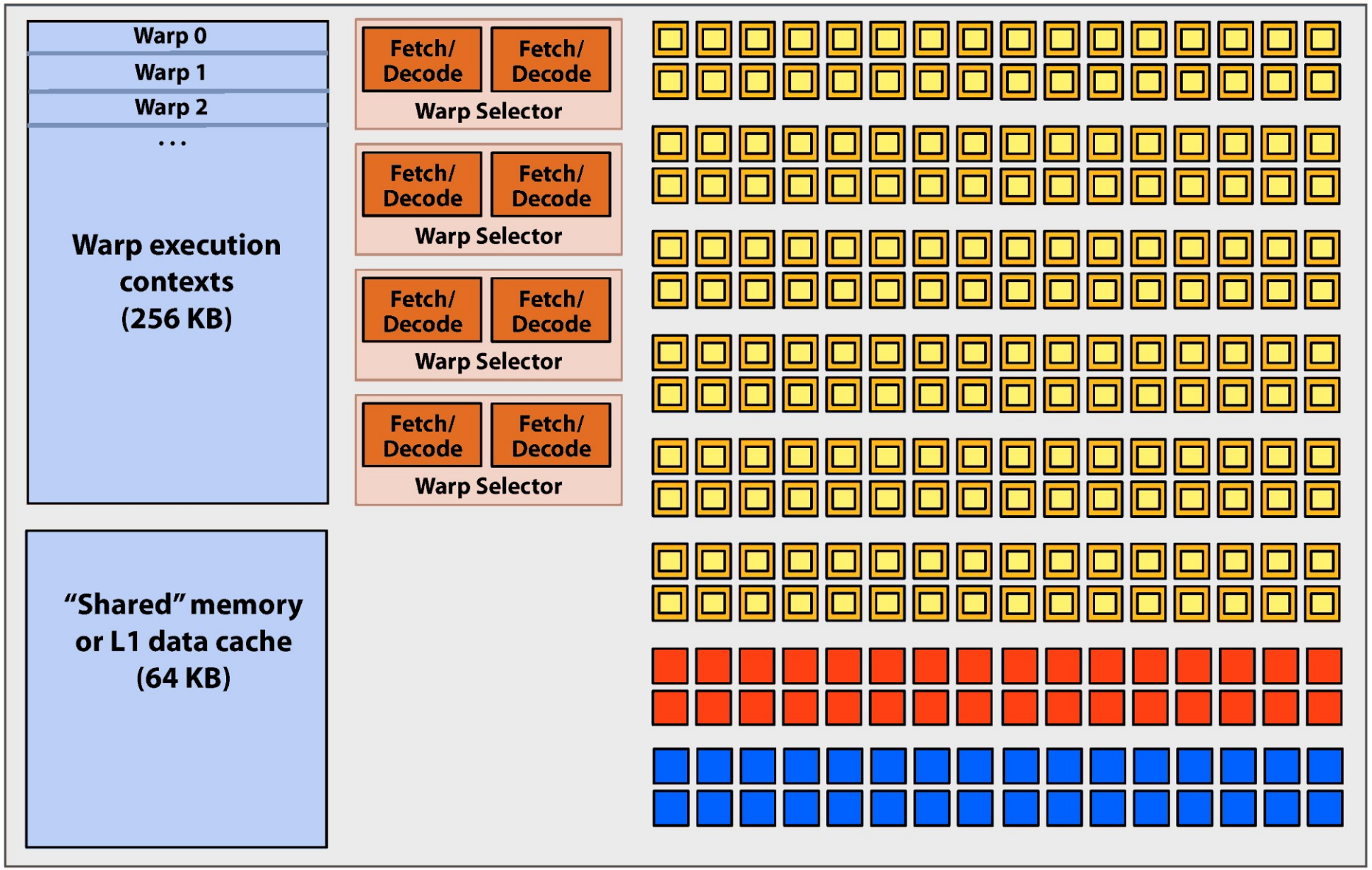


Details later...
Ignore less important subtleties...
GF100 has two warp schedulers, not one,
and each 32-thread instruction is executed
over two clock cycles, not one, etc.

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 older (<13) 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 older (<13) 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…

Instruction Throughput

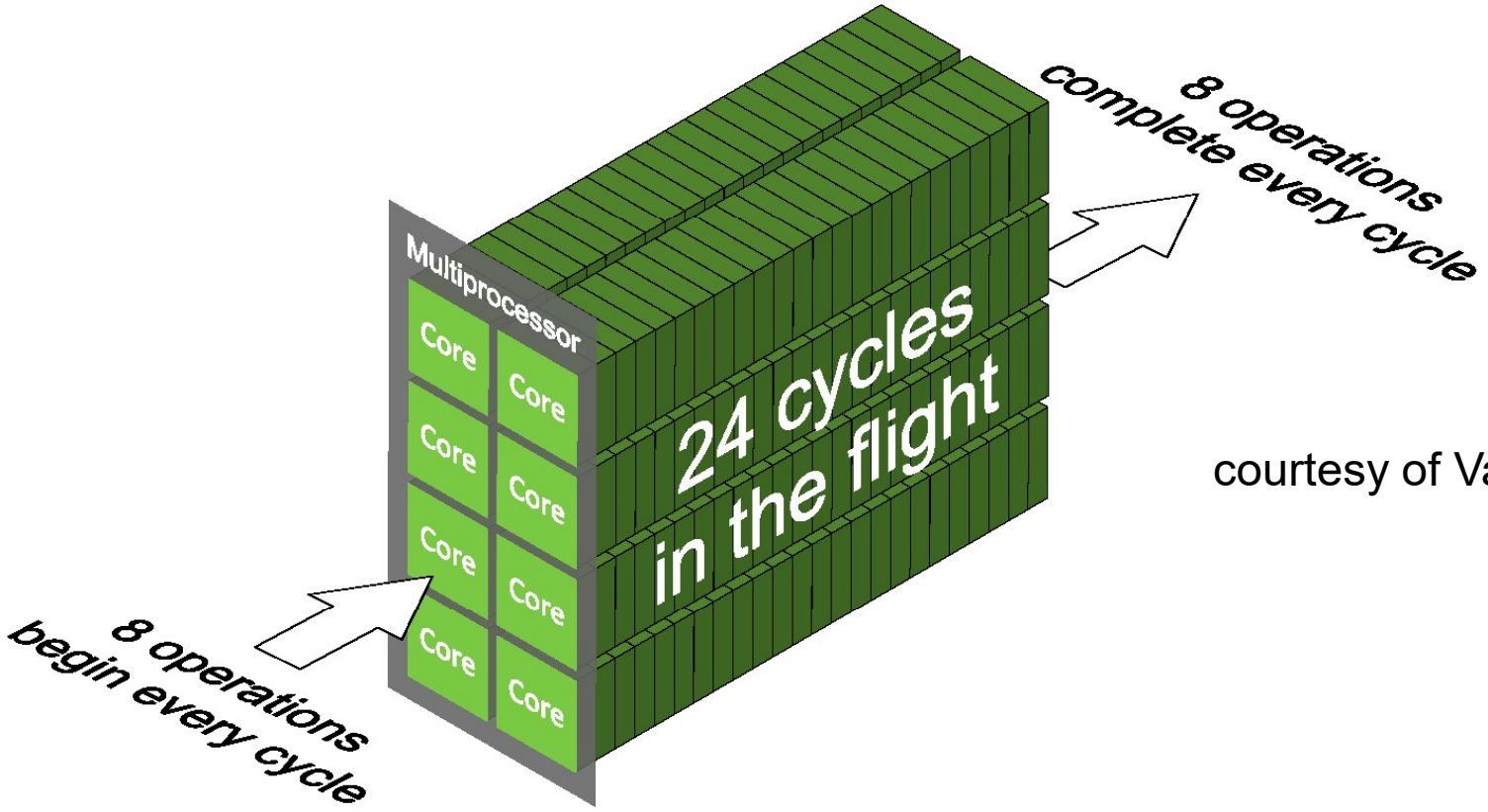
Instruction throughput numbers in CUDA 13 C Best Practices Guide (Chapter 12.1, Table 5)

Compute Capability	**7.5** Turing	**8.0** Ampere	**8.6** Ampere	**8.9** Ada	**9.0** Hopper	**10.0** Blackwell	**12.0** Blackwell
16-bit floating-point add, multiply, multiply-add (2-way SIMD): add.f16x2	64[3]	128[4]	64		128	64	
32-bit floating-point add, multiply, multiply-add: add.f32	64		128				
64-bit floating-point add, multiply, multiply-add: add.f64	2	32	2		64	64	2

[4] 64 for __nv_bfloat16

[3] multiple instructions for __nv_bfloat16

Use Little's law



courtesy of Vasily Volkov

Needed parallelism = **Latency** x **Throughput**

If we ignore ILP, this gives the minimum number of required threads(*) to keep the cores (ALUs) busy… even without any extra desired latency hiding ability!

(*) but we actually then do this with warps!

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)	10.x/12.x (Blackwell)
# warp sched. / SM	2	2	4	4	2	4	4	4	4	4
# ALU dispatch / warp sched.	1 (over 2 clocks)	2 (over 2 clocks)	2	1	1	1	1	1	1	1
SM busy with # warps + inst	L	2L	8L	4L	2L	4L	4L	4L	4L	4L
inst. pipe latency (L)	22	22	11	9	6	6	4	4	4	4
SM busy with # warps	**22**	**22 + ILP**	**44 + ILP**	**36**	**12**	**24**	**16**	**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)	10.x/12.x (Blackwell)
# warp sched. / SM	2	2	4	4	2	4	4	4	4	4
# ALU dispatch / warp sched.	1 (over 2 clocks)	2 (over 2 clocks)	2	1	1	1	1	1	1	1
SM busy with # warps + inst	L	2L	8L	4L	2L	4L	4L	4L	4L	4L
inst. pipe latency (L)	22	22	11	9	6	6	4	4	4	4
SM busy with # warps	**22**	**22 + ILP**	**44 + ILP**	**36**	**12**	**24**	**16**	**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)	10.x/12.x (Blackwell)
# warp sched. / SM	2	2	4	4	2	4	4	4	4	4
# ALU dispatch / warp sched.	1 (over 2 clocks)	2 (over 2 clocks)	2	1	1	1	1	1	1	1
SM busy with # warps + inst	L	2L	8L	4L	2L	4L	4L	4L	4L	4L
inst. pipe latency (L)	22	22	11	9	6	6	4	4	4	4
SM busy with # warps	**22**	**22 + ILP**	**44 + ILP**	**36**	**12**	**24**	**16**	**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)	10.x/12.x (Blackwell)
# warp sched. / SM	2	2	4	4	2	4	4	4	4	4
# ALU dispatch / warp sched.	1 (over 2 clocks)	2 (over 2 clocks)	2	1	1	1	1	1	1	1
SM busy with # warps + inst	L	2L	8L	4L	2L	4L	4L	4L	4L	4L
inst. pipe latency (L)	22	22	11	9	6	6	4	4	4	4
SM busy with # warps	**22**	**22 + ILP**	**44 + ILP**	**36**	**12**	**24**	**16**	**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)	10.x/12.x (Blackwell)
# warp sched. / SM	2	2	4	4	2	4	4	4	4	4
# ALU dispatch / warp sched.	1 (over 2 clocks)	2 (over 2 clocks)	2	1	1	1	1	1	1	1
SM busy with # warps + inst	L	2L	8L	4L	2L	4L	4L	4L	4L	4L
inst. pipe latency (L)	22	22	11	9	6	6	4	4	4	4
SM busy with # warps	**22**	**22 + ILP**	**44 + ILP**	**36**	**12**	**24**	**16**	**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)	10.x/12.x (Blackwell)
# warp sched. / SM	2	2	4	4	2	4	4	4	4	4
# ALU dispatch / warp sched.	1 (over 2 clocks)	2 (over 2 clocks)	2	1	1	1	1	1	1	1
SM busy with # warps + inst	L	2L	8L	4L	2L	4L	4L	4L	4L	4L
inst. pipe latency (L)	22	22	11	9	6	6	4	4	4	4
SM busy with # warps	**22**	**22 + ILP**	**44 + ILP**	**36**	**12**	**24**	**16**	**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

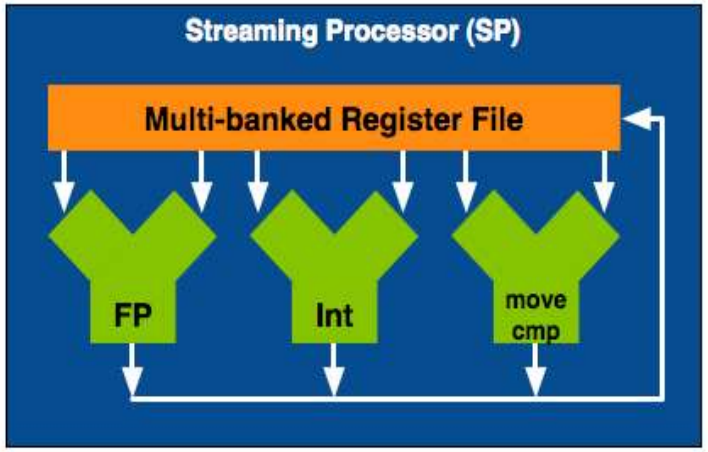
2007-2009

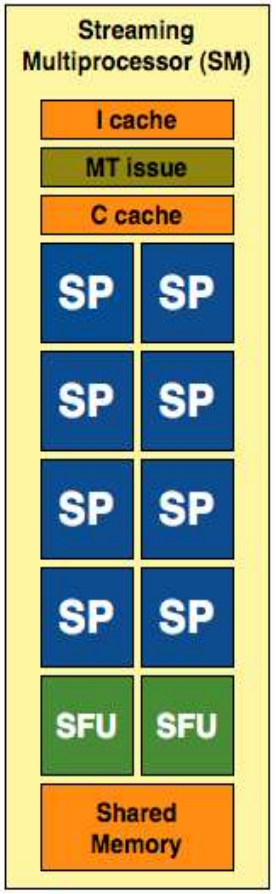
(compute capability 1.x)

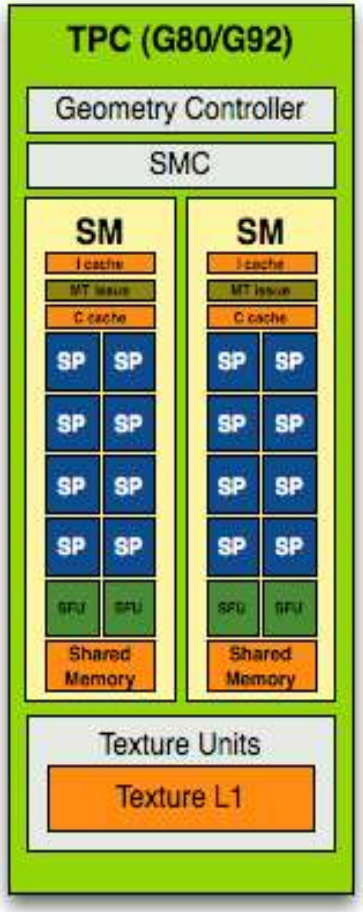
G80 (cc 1.0): 2007 (Geforce 8800, ...)
G9x (cc 1.1): 2008 (Geforce 9800, ...)
GT200 (cc 1.3): 2008/2009 (GTX 280, GTX 285, ...)

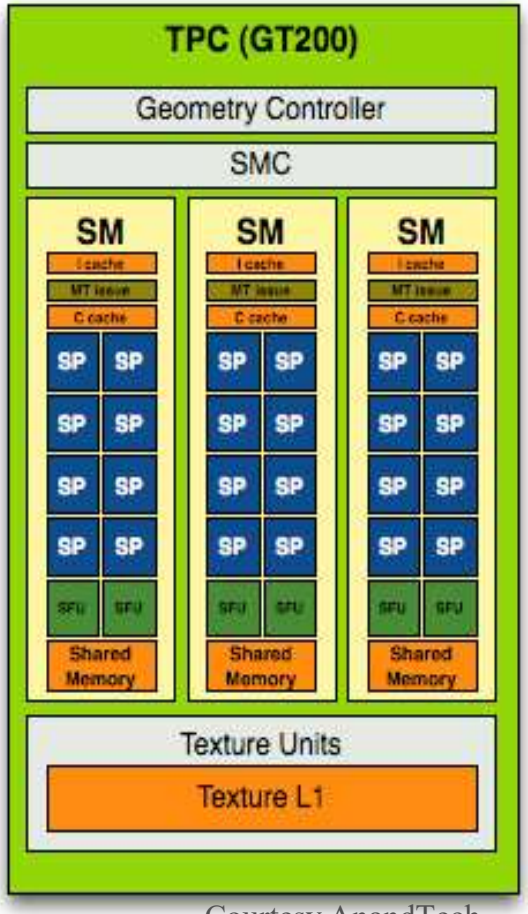
(this is not the Tesla product line!)

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









Courtesy AnandTech

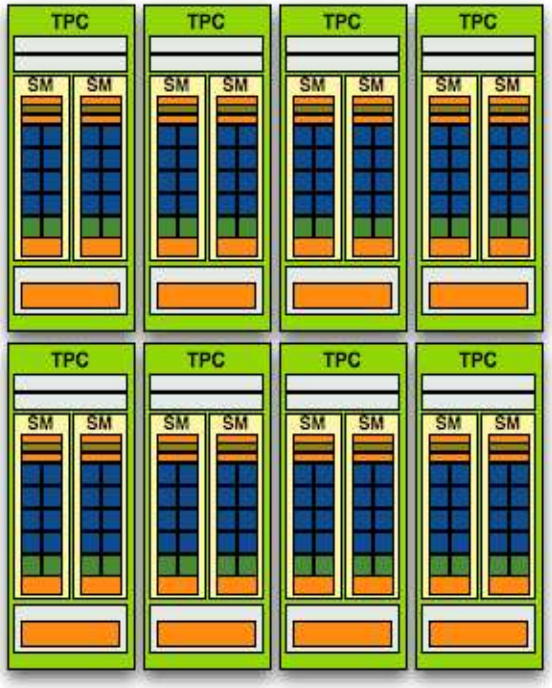
G80: first CUDA GPU!

Multiprocessor: SM (CC 1.x)

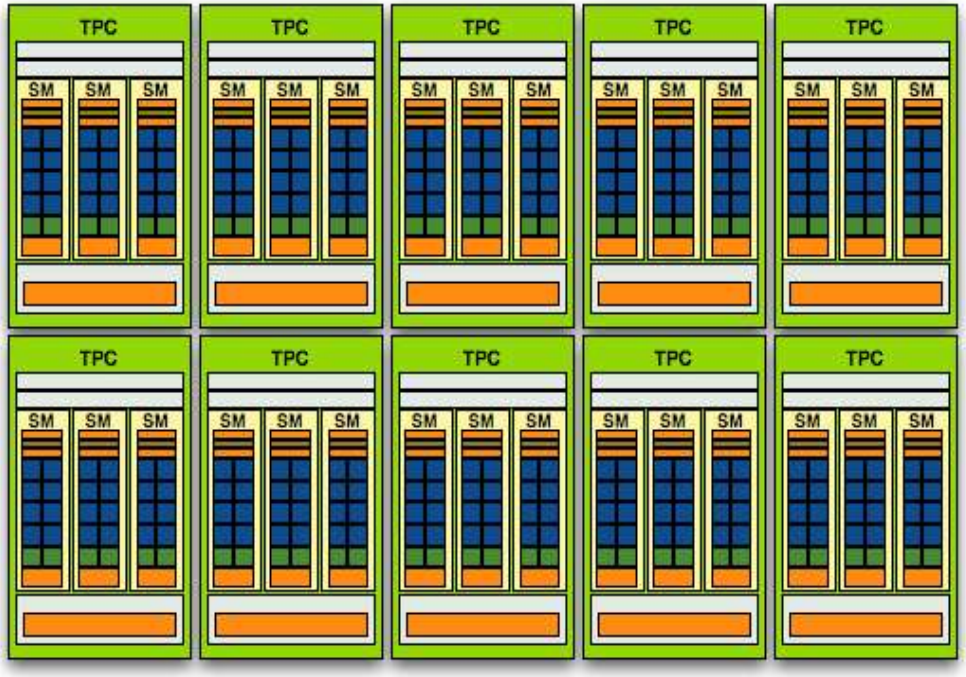
- Streaming Processor (SP) [or: CUDA core; or: FP32 / FP64 / INT32 core, ...]
- Streaming Multiprocessor (SM)
- Texture/Processing Cluster (TPC)

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)

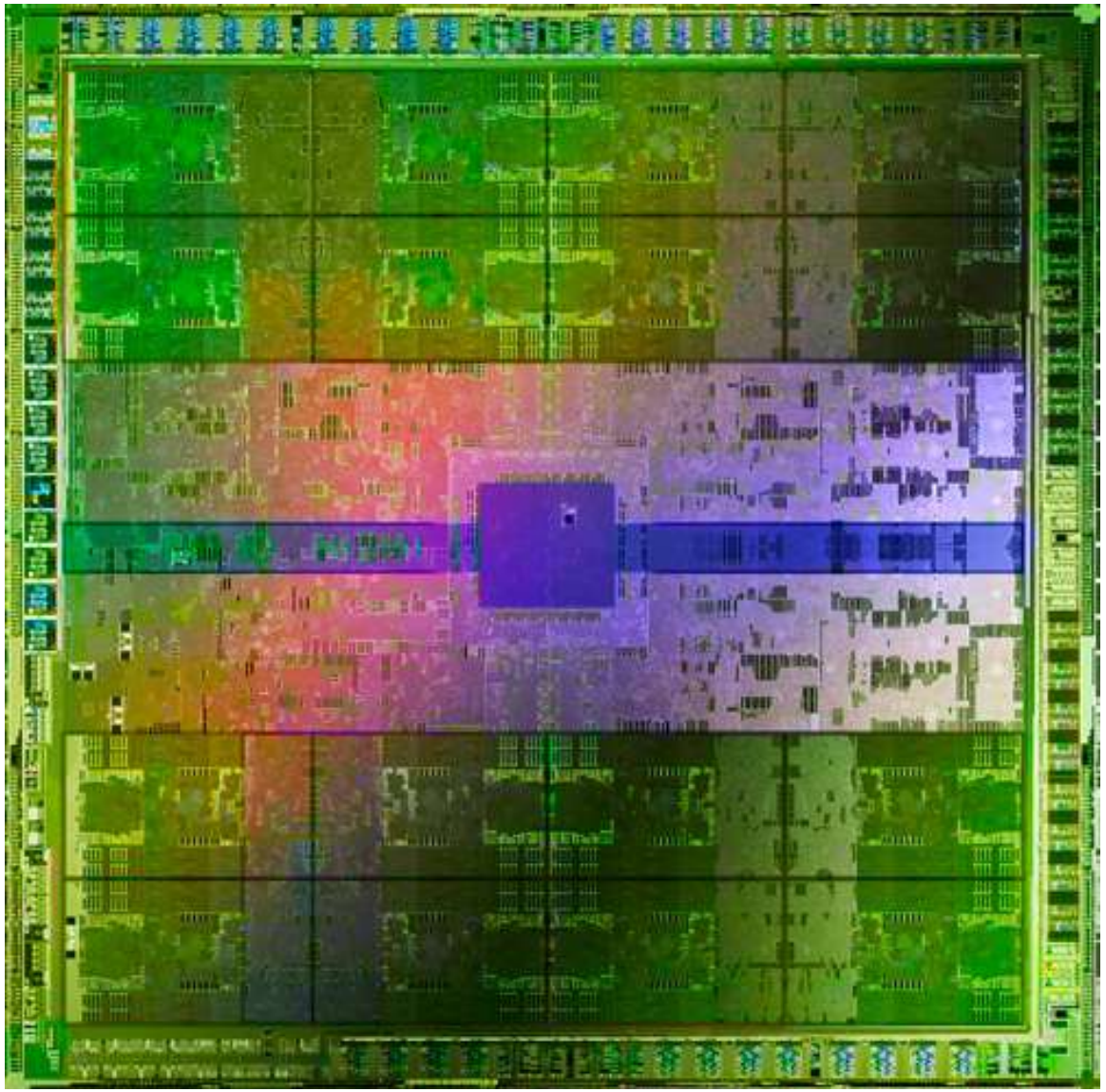
GF100 (cc 2.0), ... (GTX 480, ...)
GF104 (cc 2.1), ... (GTX 460, ...)
GF110 (cc 2.0), ... (GTX 580, ...)

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

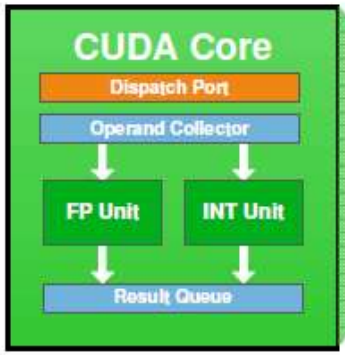
CC	2.0 (Fermi)	2.1 (Fermi)	3.x (Kepler)	5.x (Maxwell)	6.0 (Pascal)	6.1/6.2 (Pascal)	7.x (Volta, Turing)	8.0/8.6 (Ampere)	8.9/9.0 (Ada, Hopper)	10.x/12.x (Blackwell)
# warp sched. / SM	2	2	4	4	2	4	4	4	4	4
# ALU dispatch / warp sched.	1 (over 2 clocks)	2 (over 2 clocks)	2	1	1	1	1	1	1	1
SM busy with # warps + inst	L	2L	8L	4L	2L	4L	4L	4L	4L	4L
inst. pipe latency (L)	22	22	11	9	6	6	4	4	4	4
SM busy with # warps	22	22 + ILP	44 + ILP	36	12	24	16	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)

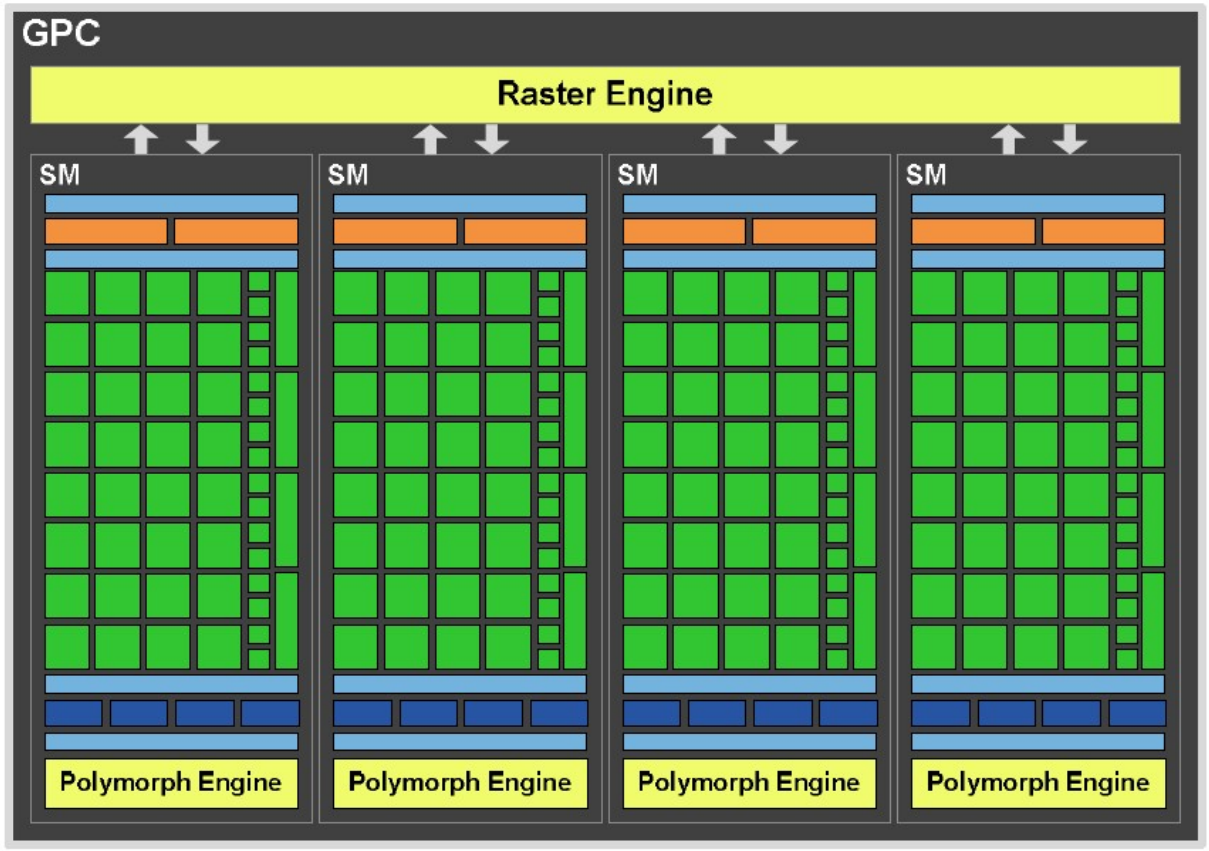
4 SMs

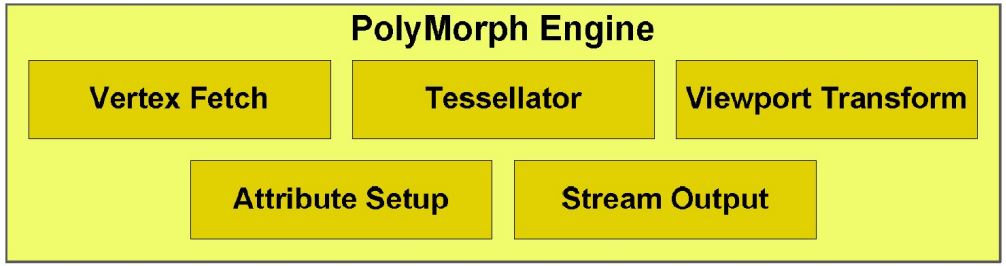
32 CUDA cores / SM

4 SMs / GPC =
128 cores / GPC

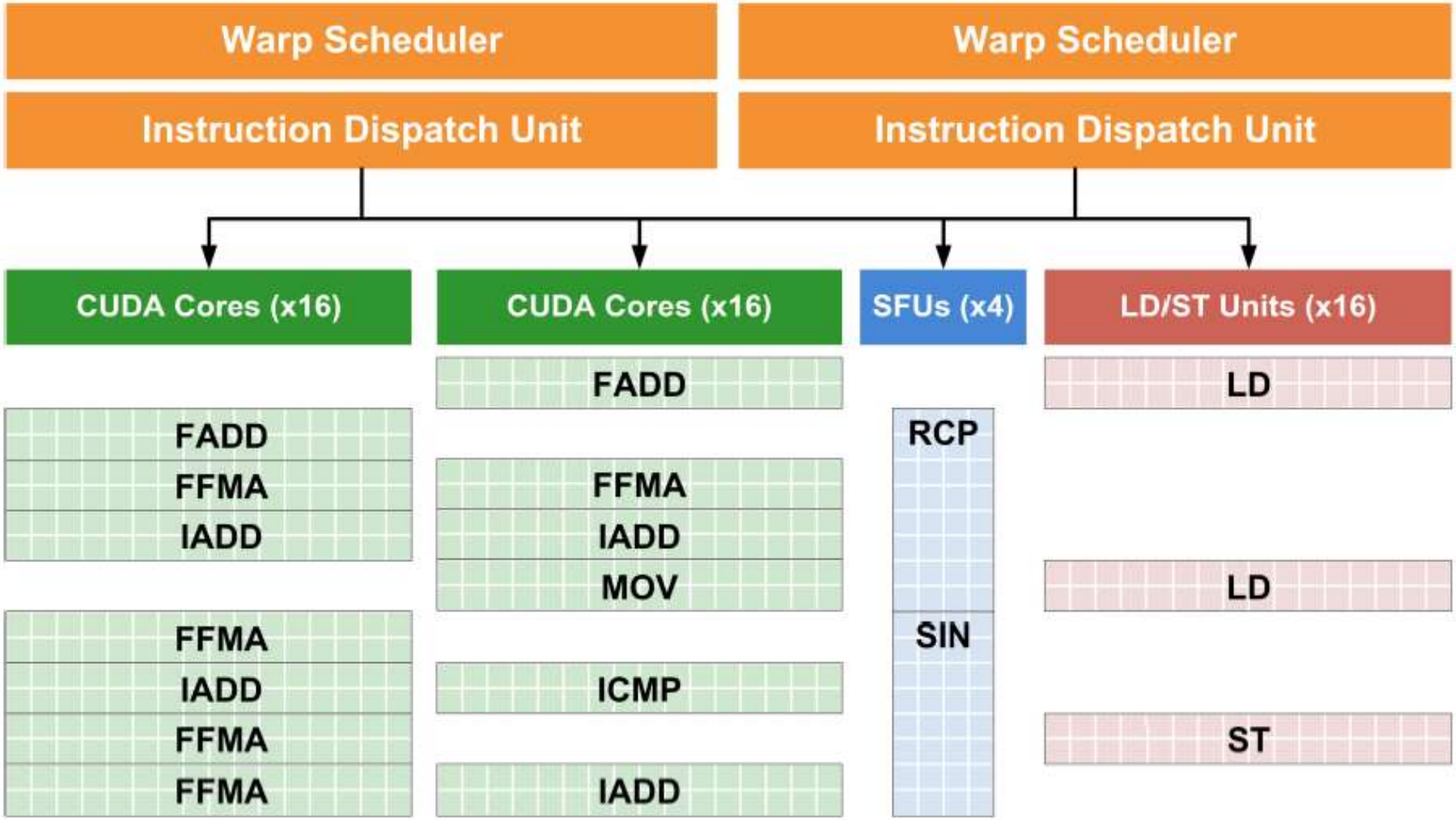
Decentralized rasterization and geometry

- 4 raster engines
- 16 ”PolyMorph” engines

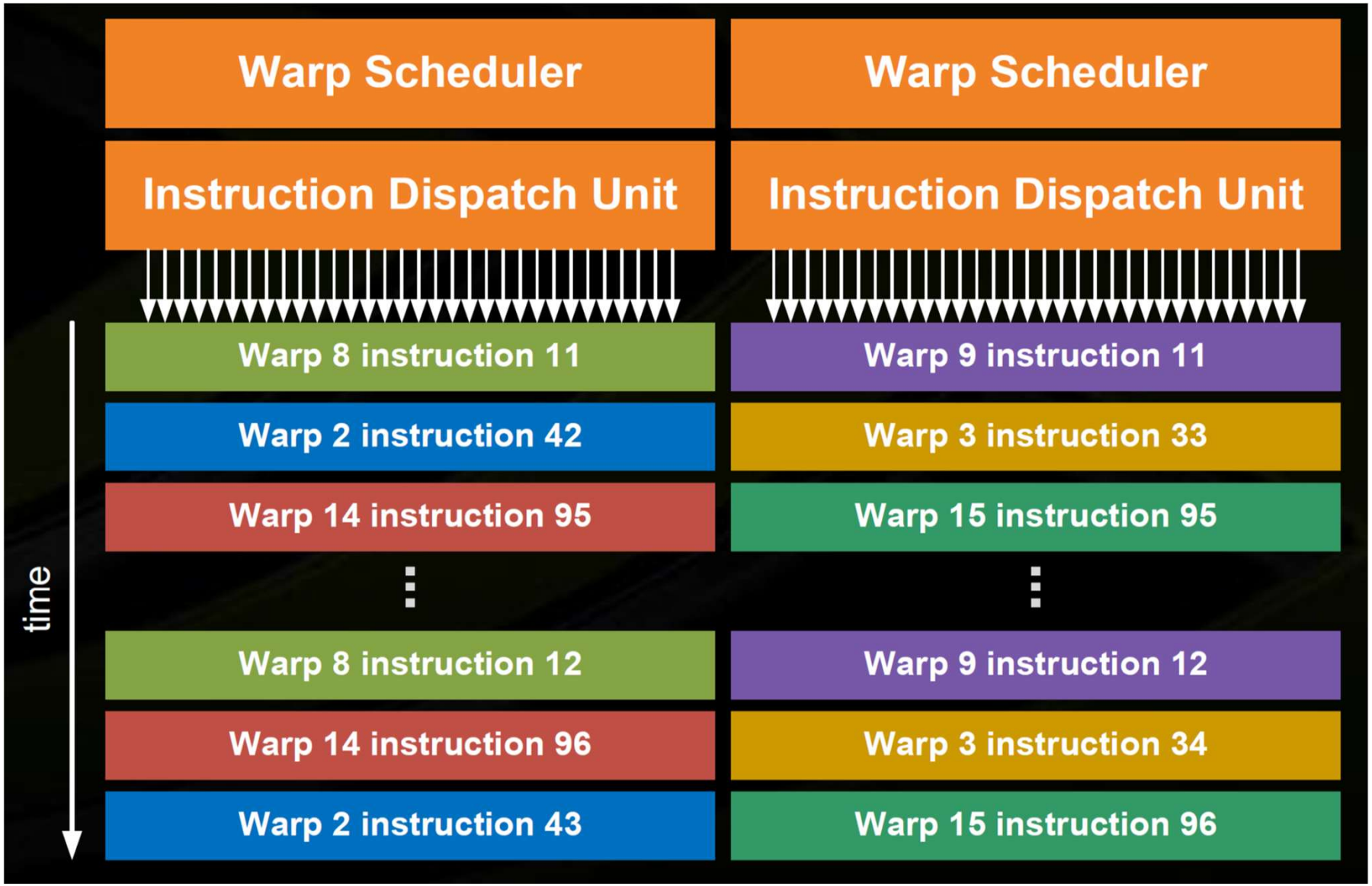




Dual Warp Schedulers



Dual Warp Schedulers

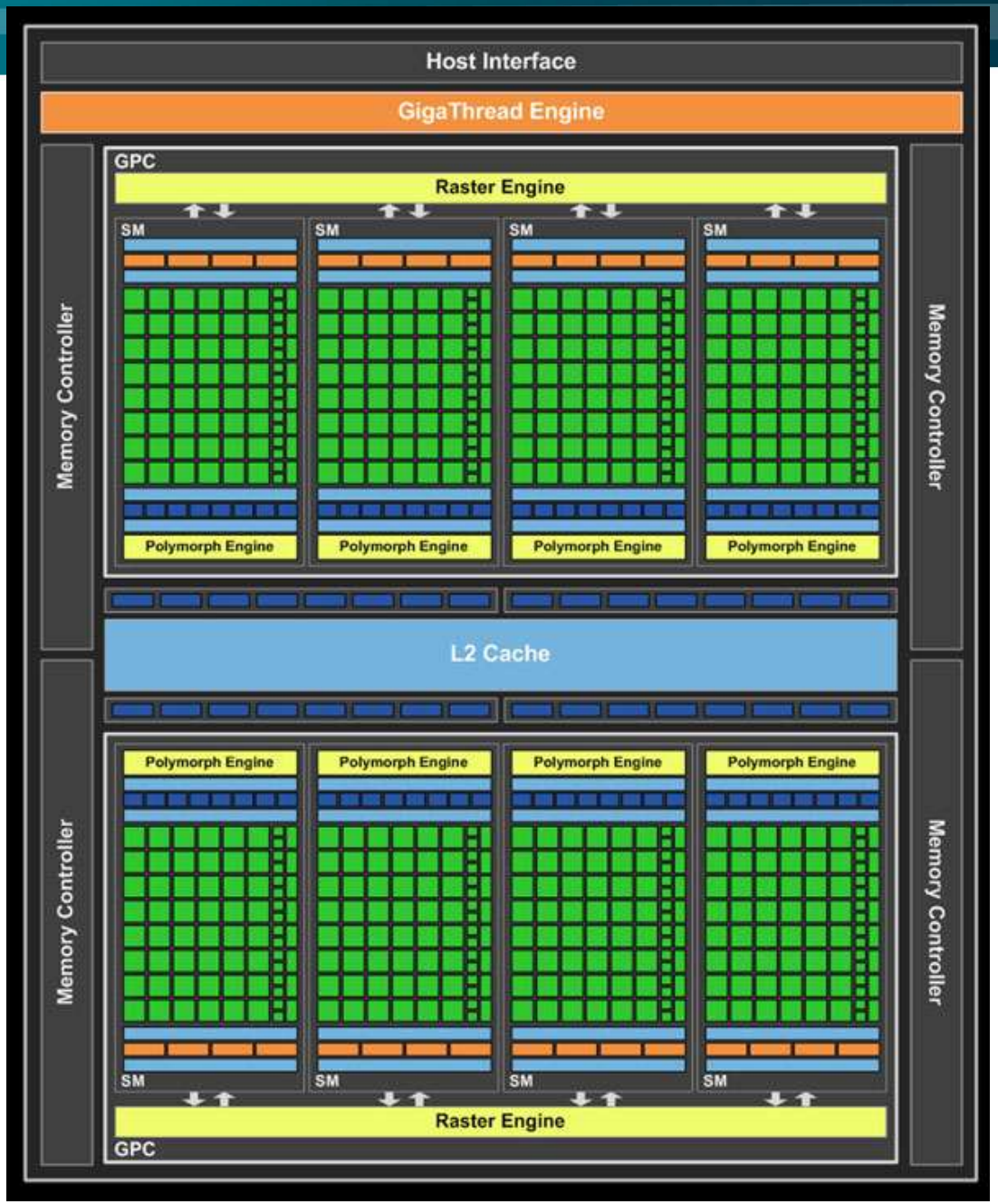


NVIDIA Fermi (GF104) Architecture (2010)

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)	10.x/12.x (Blackwell)
# warp sched. / SM	2	2	4	4	2	4	4	4	4	4
# ALU dispatch / warp sched.	1 (over 2 clocks)	2 (over 2 clocks)	2	1	1	1	1	1	1	1
SM busy with # warps + inst	L	2L	8L	4L	2L	4L	4L	4L	4L	4L
inst. pipe latency (L)	22	22	11	9	6	6	4	4	4	4
SM busy with # warps	22	22 + ILP	44 + ILP	36	12	24	16	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)

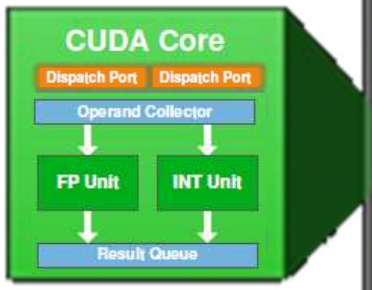
2 dispatch units / warp scheduler: "superscalar"

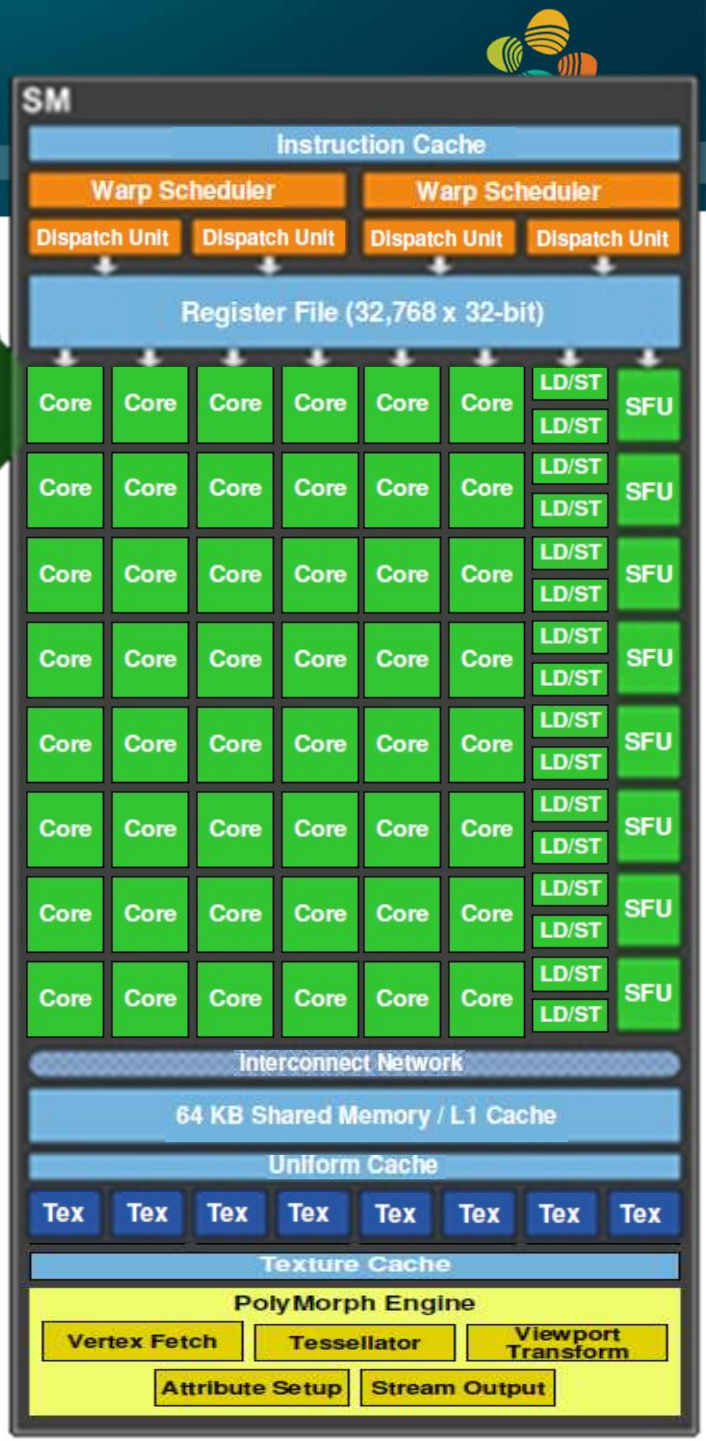
CPU-like cache hierarchy

- L1 cache / shared memory
- L2 cache

Texture units and caches now in SM

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

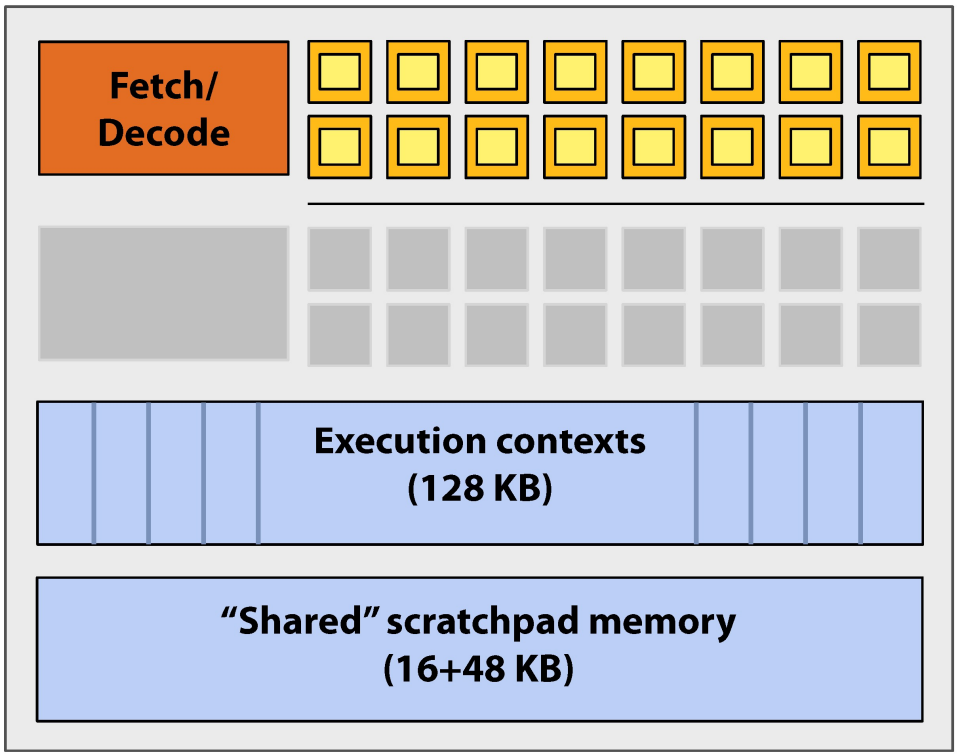


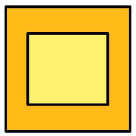


NVIDIA Fermi GF100 Architecture (2010)

NVIDIA GeForce GTX 480 "core"

CC 2.0, not 2.1 !





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

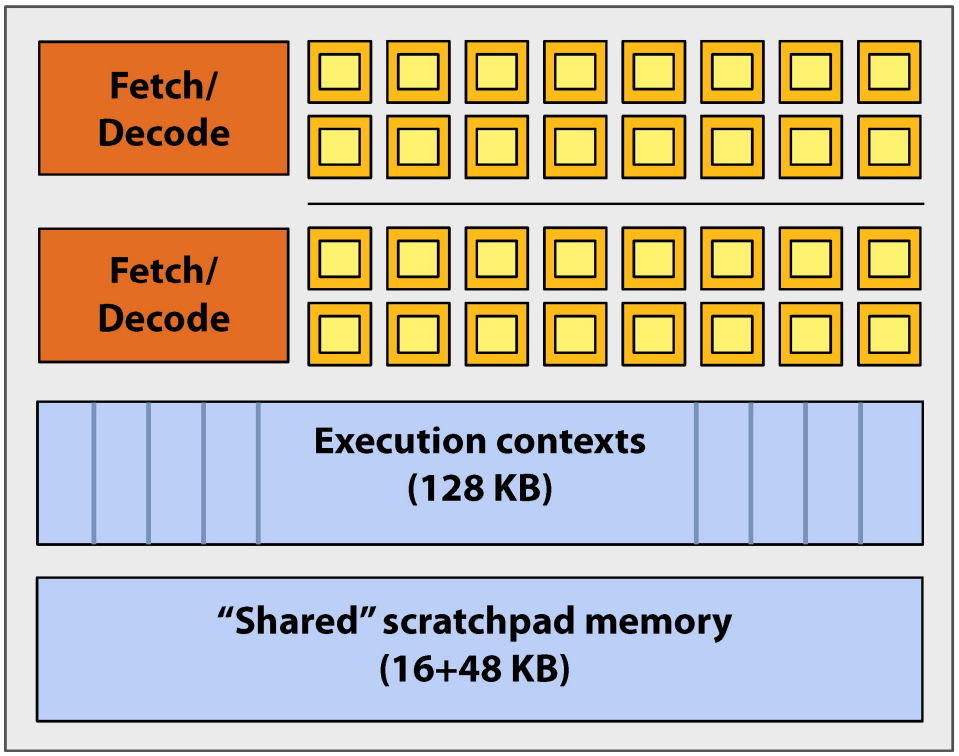
- Groups of 32 fragments share an instruction stream
- Up to 48 groups are simultaneously interleaved
- Up to 1536 individual contexts can be stored

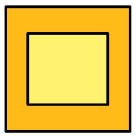
Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

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)

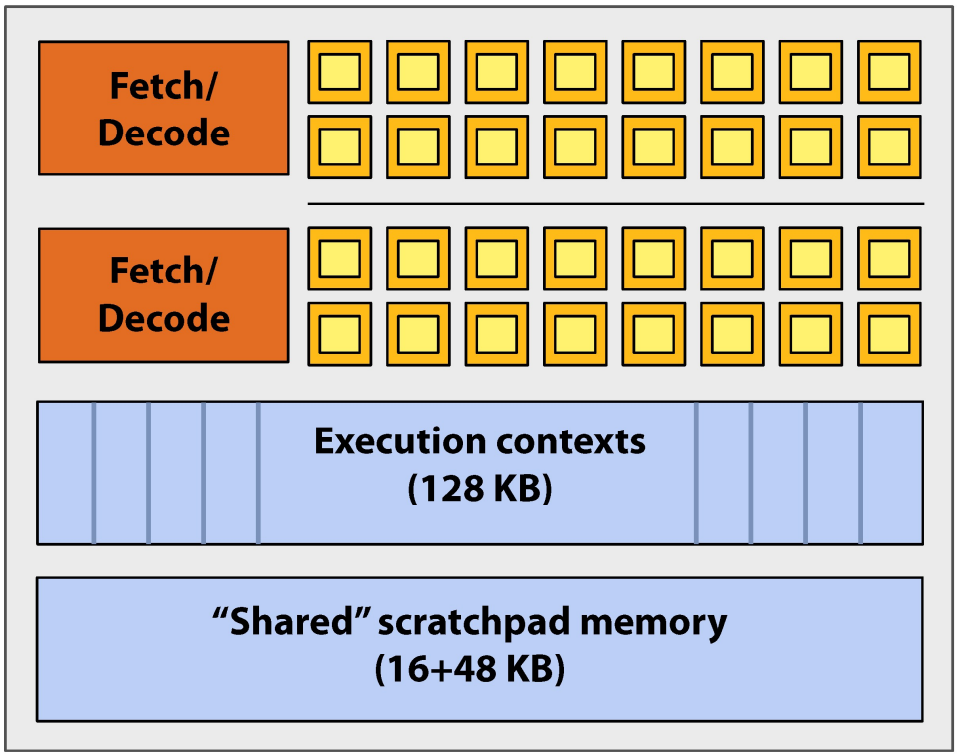
- The core contains 32 functional units
- Two groups are selected each clock (decode, fetch, and execute two instruction streams in parallel)

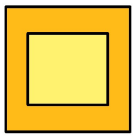
Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

NVIDIA Fermi GF100 Architecture (2010)

NVIDIA GeForce GTX 480 "SM"

CC 2.0, not 2.1 !





= CUDA core
(1 MUL-ADD per clock)

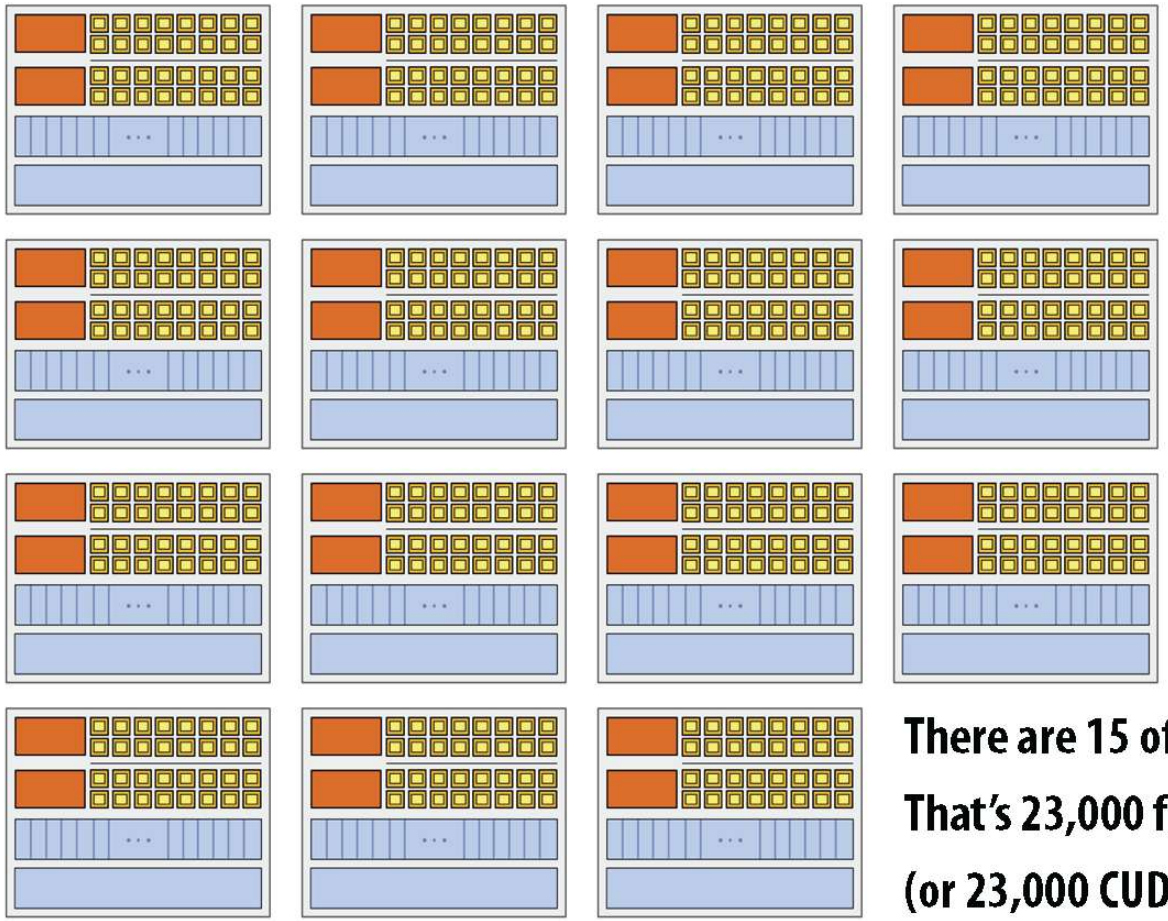
- The SM contains 32 CUDA cores
- Two warps are selected each clock (decode, fetch, and execute two warps in parallel)
- Up to 48 warps are interleaved, totaling 1536 CUDA threads

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!)

Thank you.