

CS 380 - GPU and GPGPU Programming

Lecture 8: GPU Architecture, Pt. 5

Markus Hadwiger, KAUST



Reading Assignment #5 (until Oct 2)

Read (required):

- Programming Massively Parallel Processors book, 4th edition,
Chapter 4 (Compute architecture and scheduling)
- NVIDIA CUDA C++ Programming Guide (v11.7, Aug 2022):
Read Chapter 2.6 (Compute Capability);
go through Appendix K (Compute Capabilities);
go through Chapter 5.2 (Maximize Utilization) and
Chapter 5.4 (Maximize Instruction Throughput)

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

Read (optional):

- NVIDIA Fermi (GF100) white paper (CC 2.x; for historical reasons and comparison to current GPUs):

https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

- NVIDIA Pascal (GP100) white paper (CC 6.x):

<https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf>

- NVIDIA Volta (V100) white paper (CC 7.0; tensor cores):

<http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>

- NVIDIA Turing (TU102, TU104, TU106) white paper (CC 7.5; ray tracing cores):

<https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf>



Quiz #1: Sep 28

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: General Architecture

Next Problem: Stalls!

Stalls occur when a core cannot run the next instruction because of a dependency on a previous operation.

Texture access latency = 100's to 1000's of cycles
(also: instruction pipelining hazards, ...)

We've removed the fancy caches and logic that helps avoid stalls.

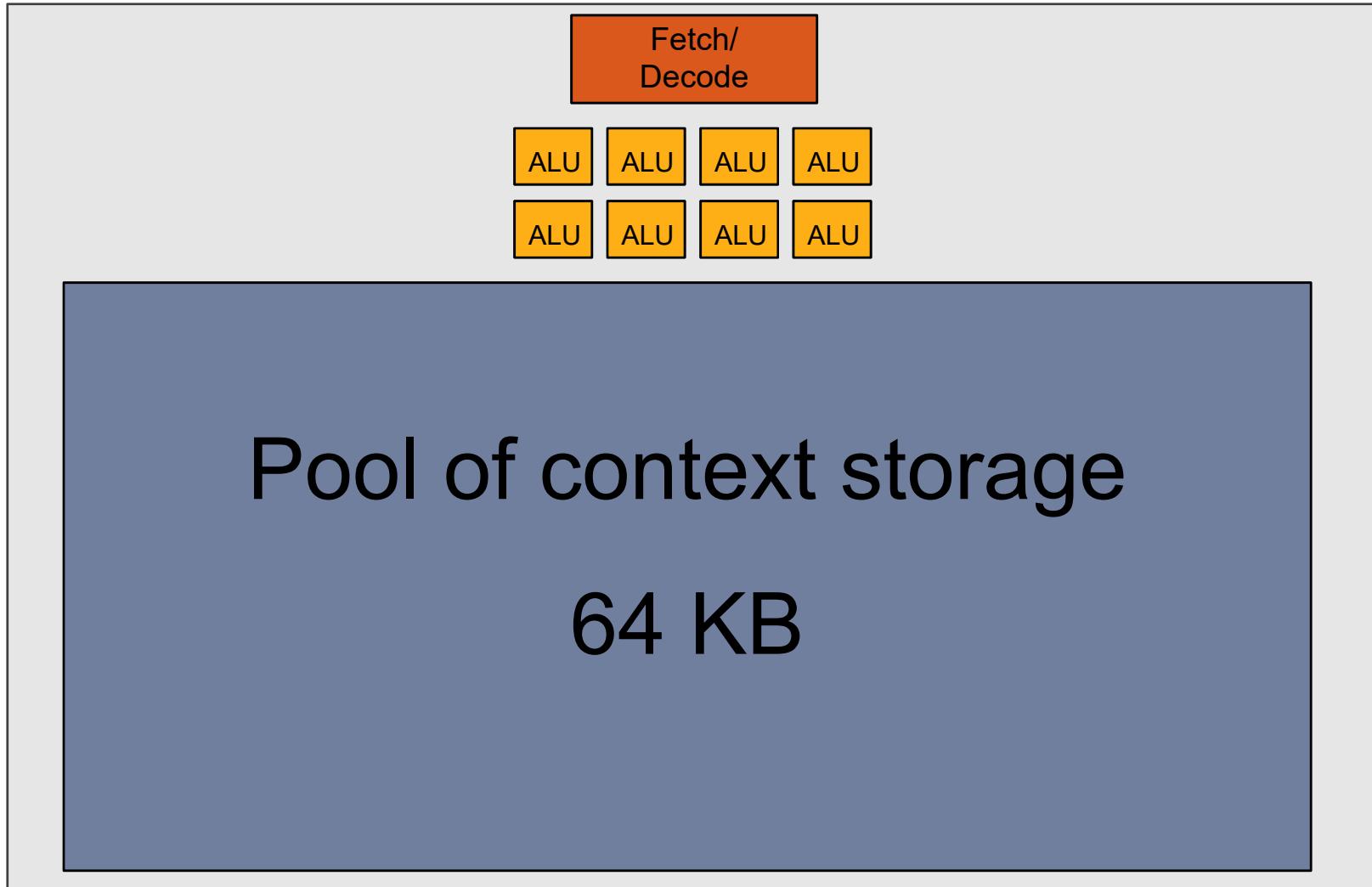
Idea #3: Interleave execution of groups

But we have **LOTS** of independent fragments.

Idea #3:

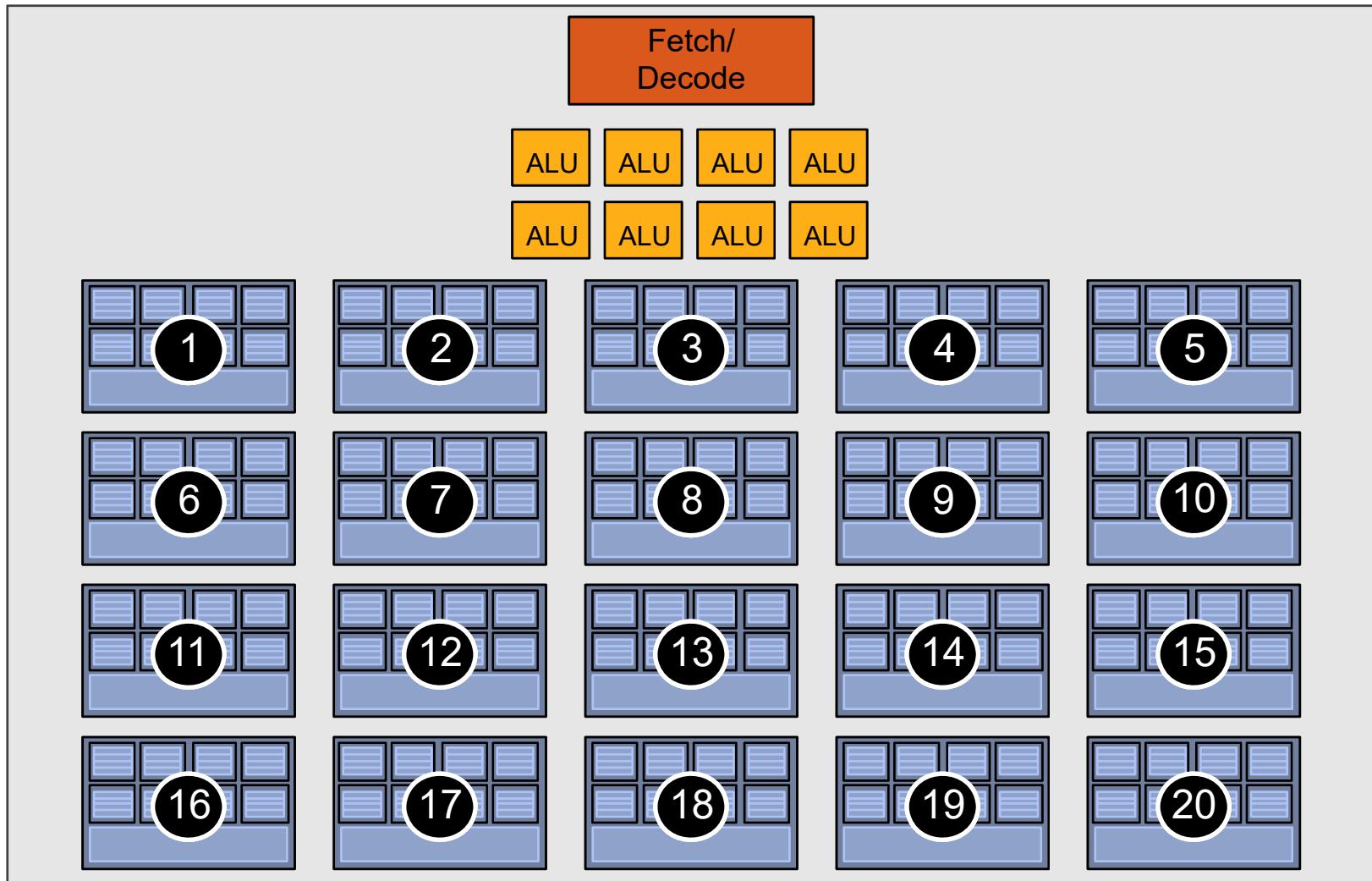
Interleave processing of many fragments on a single core
to avoid stalls caused by high latency operations.

Idea #3: Store multiple group contexts



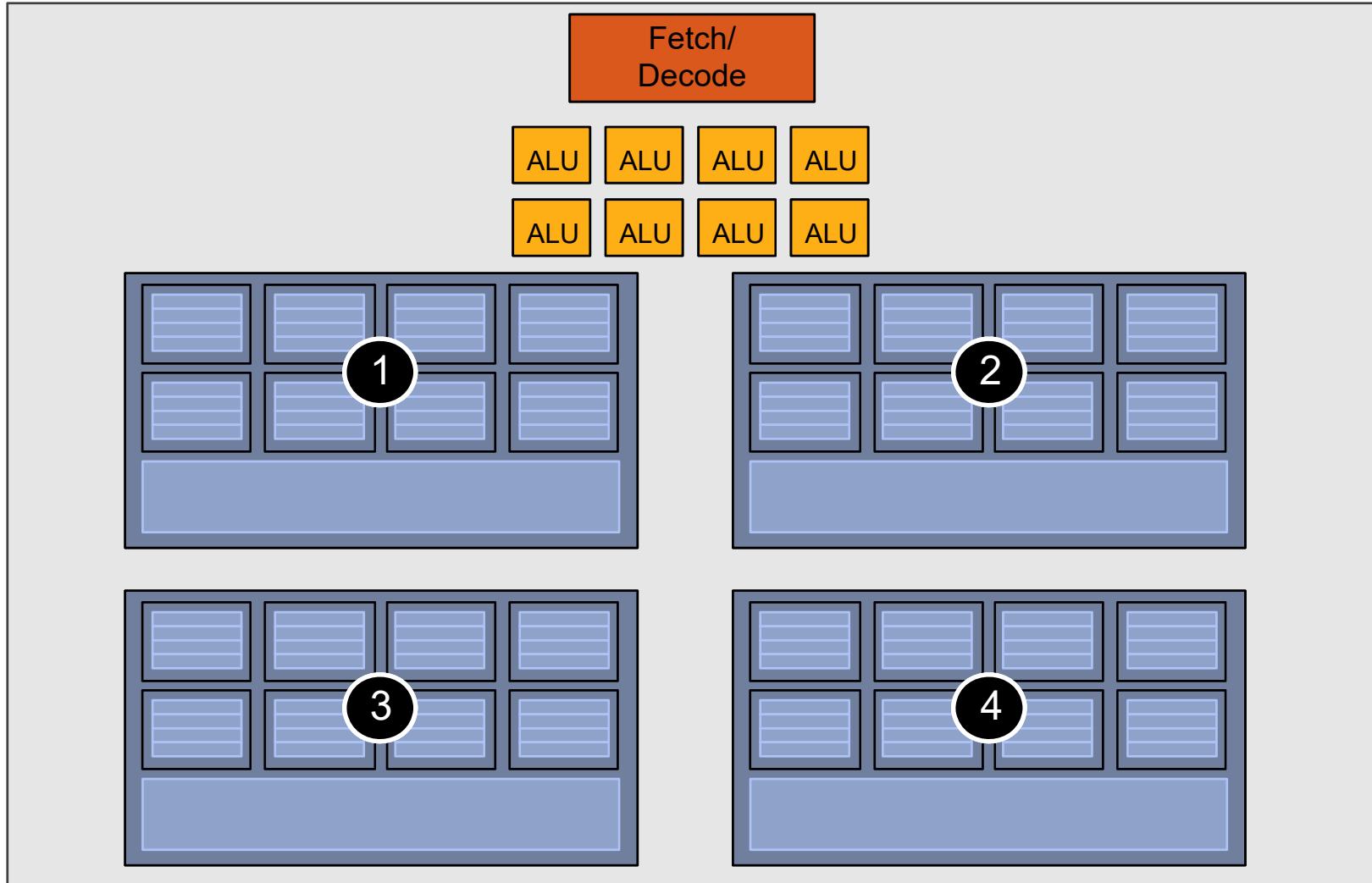
Twenty small contexts (few regs/thread)

(maximal latency hiding ability)



Four large contexts (many regs/thread)

(low latency hiding ability)



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>

Where We've Arrived...



Summary: three key ideas for high-throughput execution

1. Use many “slimmed down cores,” run them in parallel
2. Pack cores full of ALUs (by sharing instruction stream overhead across groups of fragments)
 - Option 1: Explicit SIMD vector instructions
 - Option 2: Implicit sharing managed by hardware
3. Avoid latency stalls by interleaving execution of many groups of fragments
 - When one group stalls, work on another group

GPUs are here!
(usually)

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, ...)



Interlude: PTX vs. SASS Code (1)

PTX is virtual machine ISA

SASS is actual machine ISA

For disassembly:

cuobjdump / nvdisasm

See CUDA_Binary_Utils.pdf

For debugging (and code inspection) see:

[https://developer.nvidia.com/
nsight-visual-studio-edition](https://developer.nvidia.com/nsight-visual-studio-edition)

Nsight CUDA Device Summary Disassembly matrixMul.cu matrixMul_kernel.cu

Address: _Z9matrixMulPfS_S_ii

```
0x000002ed0      MOV R1, R1;
72:
73:    // Index of the last sub-matrix of A processed by the block
74:    int aEnd    = aBegin + wA - 1;
0x000002ed8 [0083] ld.param.s32 %r14, [_cudaparm__Z9matrixMulPfS_S_ii_wA];
0x000002ed8          MVI R0, 0x1c;
0x000002ee0          R2A A1, R0;
0x000002ee8          MOV R0, g [A1+0x0];
0x000002ef0 [0084] mov.s32 %r15, %r13;
0x000002ef0          MOV32 R1, R1;
0x000002ef4 [0085] add.s32 %r16, %r14, %r15; ← PTX
0x000002ef4          IADD32 R0, R0, R1; ← SASS
0x000002ef8 [0086] sub.s32 %r17, %r16, 1;
0x000002ef8          IADD32I R8, R0, 0xffffffff;
0x000002ff0 [0087] mov.s32 %r18, %r17;
0x000002ff0          MOV R8, R8;

75:
76:    // Step size used to iterate through the sub-matrices of A
77:    int aStep   = BLOCK_SIZE;
0x000002f08 [0089] mov.s32 %r19, 16;
0x000002f08          MVI R9, 0x10;
0x000002f10 [0090] mov.s32 %r20, %r19;
0x000002f10          MOV32 R9, R9;

78:
79:    // Index of the first sub-matrix of B processed by the block
80:    int bBegin = BLOCK_SIZE * bx;
0x000002f14 [0092] mov.s32 %r21, %r2;
0x000002f14          MOV32 R4, R4;
0x000002f18 [0093] mul.lo.s32 %r22, %r21, 16;
0x000002f18          IMUL.U16.U16 R0, R4L, R31H;
0x000002f20          IMAD32I.U16 R0, R4H, 0x10, R0;
0x000002f28          SHL R2, R0, 0x10;
0x000002f30          IMAD32I.U16 R2, R4L, 0x10, R2;
0x000002f38 [0094] mov.s32 %r23, %r22;
0x000002f38          MOV R2, R2;
```



Interlude: PTX vs. SASS Code (2)

Note

- Size of instructions
(here: 16 bytes)
- MUFU.RCP computing
FP32 reciprocal on SFU
(there is no SASS division:
division is an algorithm
comprising simpler instructions)
- This is debug code:
redundant register moves
not (yet) removed by
optimizer in assembler
*(result of virtual PTX
registers being mapped
to same physical register)*
- ...

Disassembly X bicubicTexture_kernel.cuh bicubicTexture_cuda.cu binomialOptions_kernel.cu simpleCUFFT.cu oceanFFT_kernel.cu matrixMulC

Address: h0

Viewing Options

```
--- D:/development/CUDA_Samples/git_work/cuda-samples/Samples/5_Domain_Specific/bicubicTexture/bicubicTexture_kernel.cuh
__device__ float h0(float a) {
    0x0000004300bbfe00    IADD3 R1, R1, -0x18, RZ
    0x0000004300bbfe10    S2R R0, SR_LMEMHIOFF
    0x0000004300bbfe20    ISETP.GE.U32.AND P0, PT, R1, R0, PT
    0x0000004300bbfe30    BRA 0x4300bbfe50
    0x0000004300bbfe40    BPT.TRAP 0x1
    0x0000004300bbfe50    STL [R1+0x14], R21
    0x0000004300bbfe60    STL [R1+0x10], R20
    0x0000004300bbfe70    STL [R1+0xc1], R18
    0x0000004300bbfe80    STL [R1+0x8], R17
    0x0000004300bbfe90    STL [R1+0x4], R16
    0x0000004300bbfea0    STL [R1], R2
    0x0000004300bbfeb0    BMOV .32.CLEAR R18, B6
    0x0000004300bbfec0    MOV R4, R4
    0x0000004300bbfed0    MOV R4, R4
    0x0000004300bbffeo    MOV R17, R4
    0x0000004300bbffef0    return -1.0f + w1(a) / (w0(a) + w1(a)) + 0.5f;
    0x0000004300bbffef0    MOV R4, R17
    0x0000004300bbff00    MOV R20, 0x0
    0x0000004300bbff10    MOV R21, 0x0 ►
    0x0000004300bbff20    CALL.ABS.NOINC 0x0
    0x0000004300bbff30    MOV R0, R4
    0x0000004300bbff40    MOV R4, R17
    0x0000004300bbff50    MOV R16, R0
    0x0000004300bbff60    MOV R20, 0x0
    0x0000004300bbff70    MOV R21, 0x0
    0x0000004300bbff80    CALL.ABS.NOINC 0x0
    0x0000004300bbff90    MOV R0, R4
    0x0000004300bbffa0    MOV R4, R17
    0x0000004300bbffb0    MOV R2, R0
    0x0000004300bbffc0    MOV R20, 0x0
    0x0000004300bbffd0    MOV R21, 0x0
    0x0000004300bbffe0    CALL.ABS.NOINC 0x0
    0x0000004300bbfff0    MOV R4, R4
    0x0000004300bc0000    FADD R4, R2, R4
    0x0000004300bc0010    MOV R0, R16
    0x0000004300bc0020    MOV R4, R4
    0x0000004300bc0030    MOV R0, R0
    0x0000004300bc0040    MOV R4, R4
    0x0000004300bc0050    MOV R3, R4
    0x0000004300bc0060    MUFU.RCP R5, R3
    0x0000004300bc0070    FADD R3, -R2, -R3
    0x0000004300bc0080    MOV R3, R3
    0x0000004300bc0090    MOV R6, 0x3f800000
    0x0000004300bc00a0    FFMA R6, R3, R5, R6
    0x0000004300bc00b0    FCHK P0, R0, R4
    0x0000004300bc00c0    FFMA R6, R5, R6, R5
    0x0000004300bc00d0    MOV R5, RZ
    0x0000004300bc00e0    MOV R0, R0
```

(SASS on Ampere)

// h0 and h1 are the two offset functions
__device__ float h0(float a) {
 // note +0.5 offset to compensate for CUDA linear filtering convention
 return -1.0f + w1(a) / (w0(a) + w1(a)) + 0.5f;
}
__device__ float h1(float a) { return 1.0f + w3(a) / (w2(a) + w3(a)) + 0.5f; }



Interlude: PTX vs. SASS Code (2)

Note

- Size of instructions
(here: 16 bytes)
- MUFU.RCP computing
FP32 reciprocal on SFU
(there is no SASS division:
division is an algorithm
comprising simpler instructions)
- This is debug code:
redundant register moves
not (yet) removed by
optimizer in assembler
*(result of virtual PTX
registers being mapped
to same physical register)*
- ...

Disassembly X bicubicTexture_kernel.cuh bicubicTexture_cuda.cu binomialOptions_kernel.cu simpleCUFFT.cu oceanFFT_kernel.cu matrixMulC

Address: h0

Viewing Options

```
--- D:/development/CUDA_Samples/git_work/cuda-samples/Samples/5_Domain_Specific/bicubicTexture/bicubicTexture_kernel.cuh
__device__ float h0(float a) {
    0x0000004300bbfe00    IADD3 R1, R1, -0x18, RZ
    0x0000004300bbfe10    S2R R0, SR_LMEMHIOFF
    0x0000004300bbfe20    ISETP.GE.U32.AND P0, PT, R1, R0, PT
    0x0000004300bbfe30    BRA 0x4300bbfe50
    0x0000004300bbfe40    BPT.TRAP 0x1
    0x0000004300bbfe50    STL [R1+0x14], R21
    0x0000004300bbfe60    STL [R1+0x10], R20
    0x0000004300bbfe70    STL [R1+0xc1], R18
    0x0000004300bbfe80    STL [R1+0x8], R17
    0x0000004300bbfe90    STL [R1+0x4], R16
    0x0000004300bbfea0    STL [R1], R2
    0x0000004300bbfeb0    BMOV.32.CLEAR R18, B6
    0x0000004300bbfec0    MOV R4, R4
    0x0000004300bbfed0    MOV R4, R4
    0x0000004300bbffeo    MOV R17, R4
    0x0000004300bbffef0    MOV R4, R17
    0x0000004300bbfffo    MOV R20, 0x0
    0x0000004300bbfff20   MOV R21, 0x0 ►
    0x0000004300bbfff30   CALL.ABS.NOINC 0x0
    0x0000004300bbfff40   MOV R0, R4
    0x0000004300bbfff50   MOV R4, R17
    0x0000004300bbfff60   MOV R16, R0
    0x0000004300bbfff70   MOV R20, 0x0
    0x0000004300bbfff80   MOV R21, 0x0
    0x0000004300bbfff90   CALL.ABS.NOINC 0x0
    0x0000004300bbffa0    MOV R0, R4
    0x0000004300bbffb0    MOV R4, R17
    0x0000004300bbfffc0   MOV R2, R0
    0x0000004300bbfffd0   MOV R20, 0x0
    0x0000004300bbffe0    MOV R21, 0x0
    0x0000004300bbfffe0   CALL.ABS.NOINC 0x0
    0x0000004300bc0000    MOV R4, R4
    0x0000004300bc0010    FADD R4, R2, R4
    0x0000004300bc0020    MOV R0, R16
    0x0000004300bc0030    MOV R0, R0
    0x0000004300bc0040    MOV R4, R4
    0x0000004300bc0050    MOV R3, R4
    0x0000004300bc0060    MUFU.RCP R5, R3
    0x0000004300bc0070    FADD R3, -R2, -R3
    0x0000004300bc0080    MOV R3, R3
    0x0000004300bc0090    MOV R6, 0x3f800000
    0x0000004300bc00a0    FFMA R6, R3, R5, R6
    0x0000004300bc00b0    FCHK P0, R0, R4
    0x0000004300bc00c0    FFMA R6, R5, R6, R5
    0x0000004300bc00d0    MOV R5, RZ
    0x0000004300bc00e0    MOV R0, R0
```

(SASS on Ampere)

// h0 and h1 are the two offset functions
__device__ float h0(float a) {
 // note +0.5 offset to compensate for CUDA linear filtering convention
 return -1.0f + w1(a) / (w0(a) + w1(a)) + 0.5f;
}
__device__ float h1(float a) { return 1.0f + w3(a) / (w2(a) + w3(a)) + 0.5f; }



Interlude: PTX vs. SASS Code (2)

Note

- Size of instructions
(here: 16 bytes)
- MUFU.RCP computing
FP32 reciprocal on SFU
(there is no SASS division:
division is an algorithm
comprising simpler instructions)
- This is debug code:
redundant register moves
not (yet) removed by
optimizer in assembler
*(result of virtual PTX
registers being mapped
to same physical register)*
- ...

Disassembly

Address: h0

Viewing Options

```
--- D:/development/CUDA_Samples/git_work/cuda-samples/Samples/5_Domain_Specific/bicubicTexture/bicubicTexture_kernel.cuh
__device__ float h0(float a) {
    0x0000004300bbfe00    IADD3 R1, R1, -0x18, RZ
    0x0000004300bbfe10    S2R R0, SR_LMEMHIOFF
    0x0000004300bbfe20    ISETP.GE.U32.AND P0, PT, R1, R0, PT
    0x0000004300bbfe30    BRA 0x4300bbfe50
    0x0000004300bbfe40    BPT.TRAP 0x1
    0x0000004300bbfe50    STL [R1+0x14], R21
    0x0000004300bbfe60    STL [R1+0x10], R20
    0x0000004300bbfe70    STL [R1+0xc], R18
    0x0000004300bbfe80    STL [R1+0x8], R17
    0x0000004300bbfe90    STL [R1+0x4], R16
    0x0000004300bbfea0    STL [R1], R2
    0x0000004300bbfeb0    BMOV_32_CLEAR R18, B6
    0x0000004300bbfec0    MOV R4, R4
    0x0000004300bbfed0    MOV R4, R4
    0x0000004300bbfee0
    return -1.0f + w1(a) / (w0(a) + w1(a)) + 0.5f;
0x0000004300bbff00    MOV R17, R4
0x0000004300bbff10    MOV R4, R17
0x0000004300bbff20    MOV R20, 0x0
0x0000004300bbff30    MOV R21, 0x0 ►
CALL.ABS.NOINC 0x0
    MOV R0, R4
    MOV R4, R17
    MOV R16, R0
    MOV R20, 0x0
    MOV R21, 0x0
    CALL.ABS.NOINC 0x0
    MOV R0, R4
    MOV R4, R17
    MOV R2, R0
    MOV R20, 0x0
    MOV R21, 0x0
    CALL.ABS.NOINC 0x0
    MOV R4, R4
    FADD R4, R2, R4
    MOV R0, R16
    MOV R4, R4
    MOV R0, R0
    MOV R4, R4
    MOV R3, R4
    MUFU.RCP R5, R3
    FADD R3, -R2, -R3
    MOV R3, R3
    MOV R6, 0x3F800000
    FFMA R6, R3, R5, R6
    FCHK P0, R0, R4
    FFMA R6, R5, R6, R5
    MOV R5, RZ
    MOV R0, R0
}
__device__ float h1(float a) { return 1.0f + w3(a) / (w2(a) + w3(a)) + 0.5f; }
```

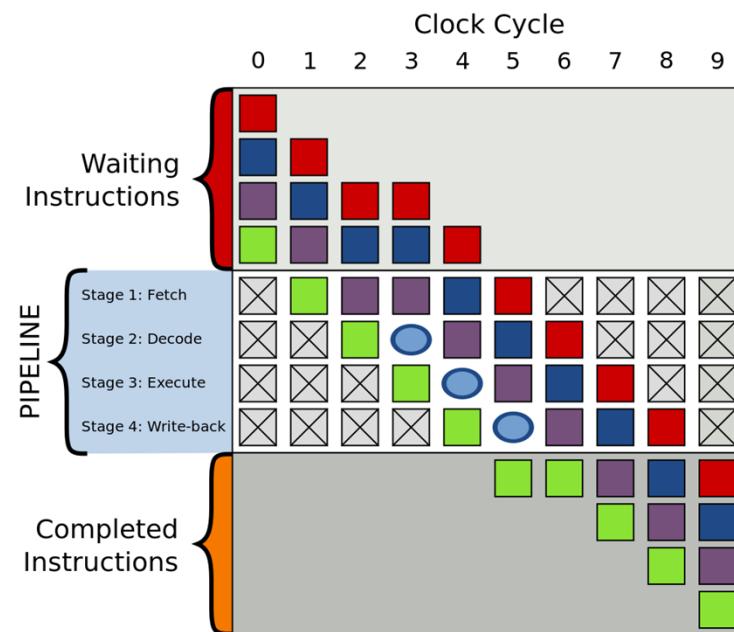
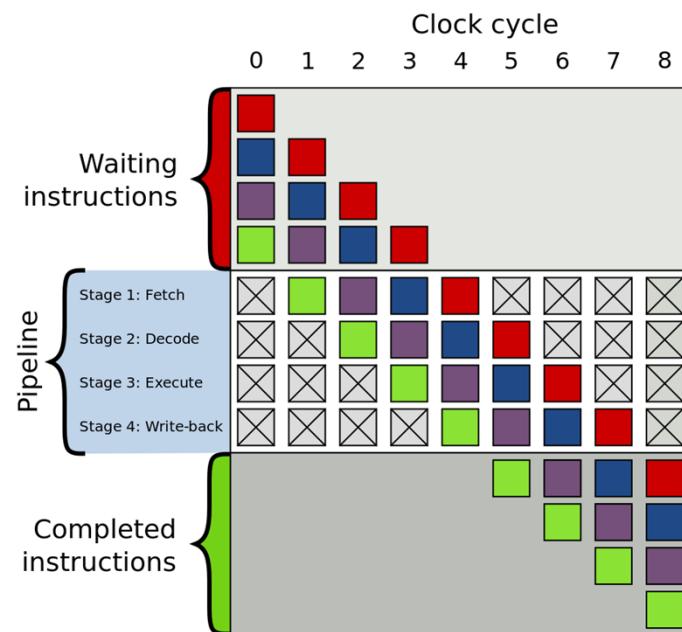
(SASS on Ampere)

Instruction Pipelining



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

Problem: hazards (different solutions: bubbles, ...)



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

wikipedia

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), frontend (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)

20

Example: “Scalar” GF100

Main concept here:

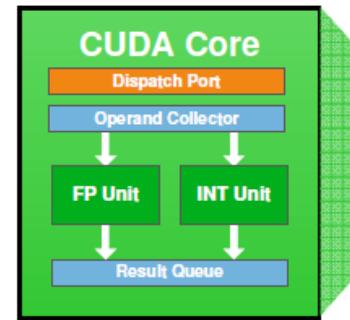
There is one instruction dispatcher
(dispatch unit / fetch/decode unit)
per warp scheduler
(warp selector)

Details later...

Ignore less important subtleties...

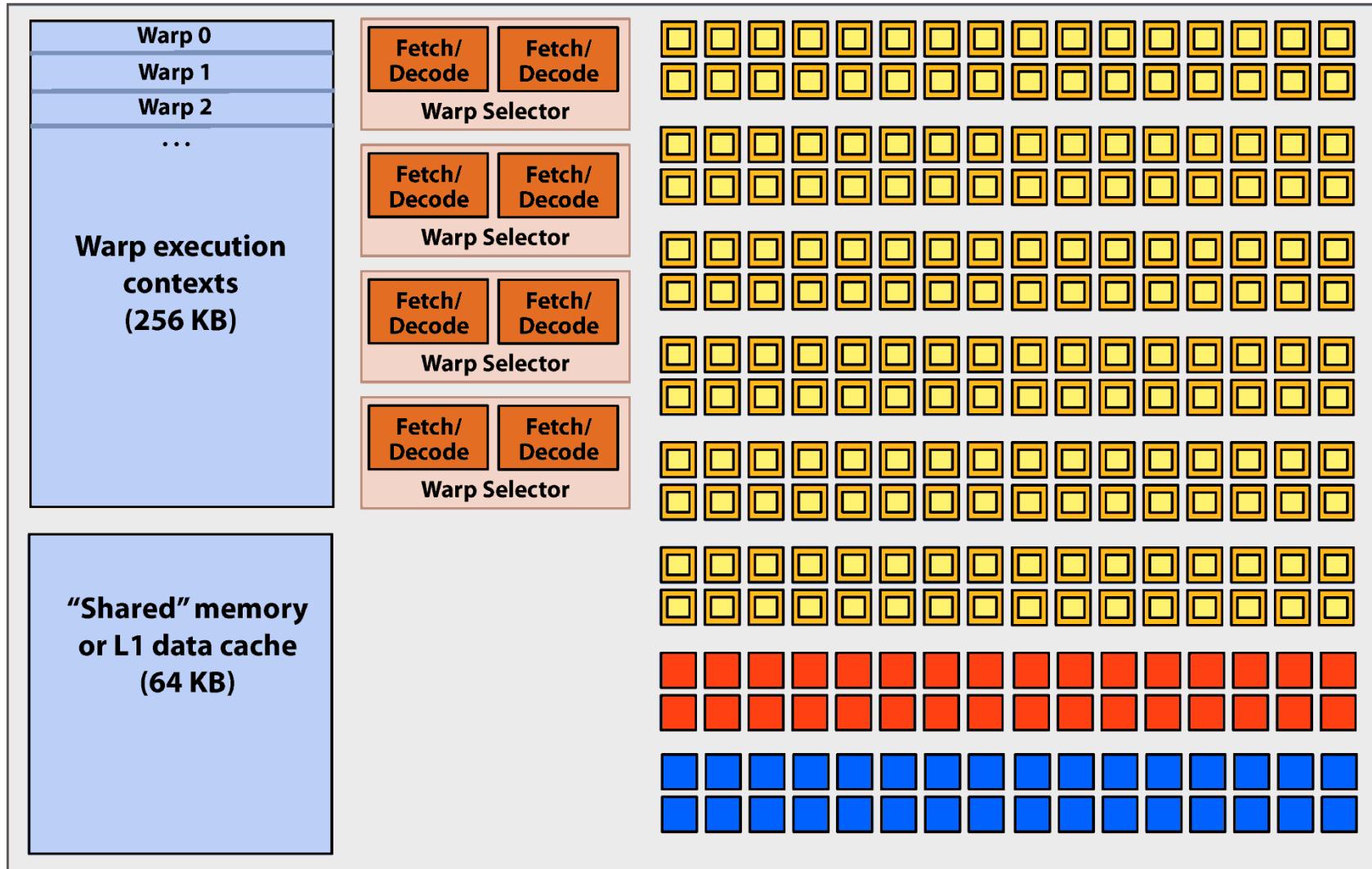
GF100 has two warp schedulers, not one,
and each 32-thread instruction is executed
over two clock cycles, not one, etc.

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”)





Instruction Throughput

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

	Compute Capability									8.9	9.0
	3.5, 3.7	5.0, 5.2	5.3	6.0	6.1	6.2	7.x	8.0	8.6		
16-bit floating-point add, multiply, multiply-add	N/A		256	128	2	256	128		256^3	?	256
32-bit floating-point add, multiply, multiply-add	192		128	64	128		64	128		?	128
64-bit floating-point add, multiply, multiply-add	64^4		4	32	4		32^5	32	2	?	64

³

⁴

⁵

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.x (Ampere)	8.9/9.x (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	?
SM busy with # warps	22	22 + ILP	44 + ILP	36	12	24	16	16	4*?

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.x (Ampere)	8.9/9.x (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	?
SM busy with # warps	22	22 + ILP	44 + ILP	36	12	24	16	16	4*?

*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



NVIDIA Tesla Architecture

2007-2009

(compute capability 1.x)

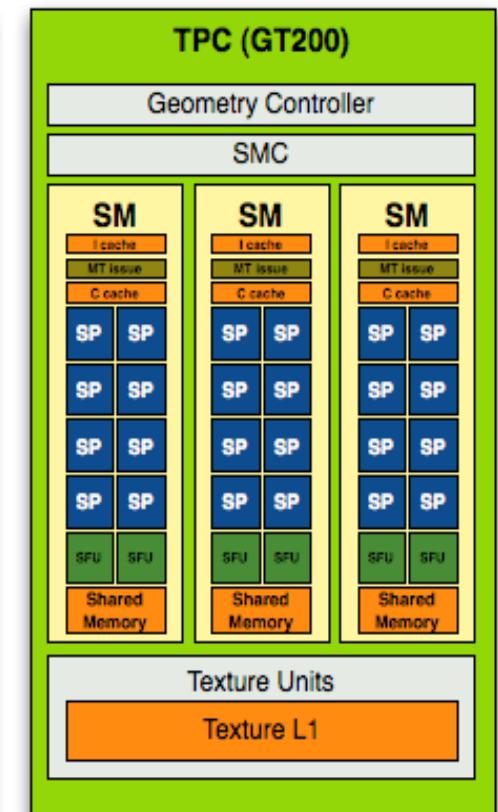
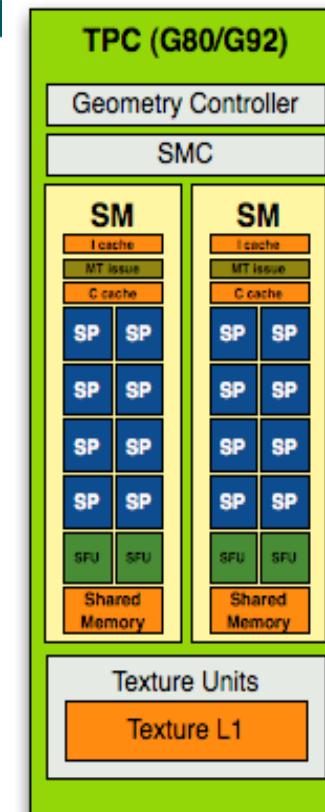
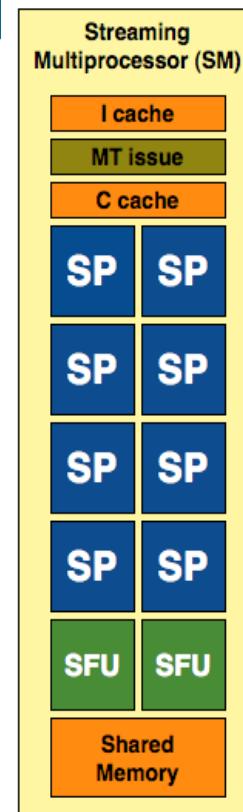
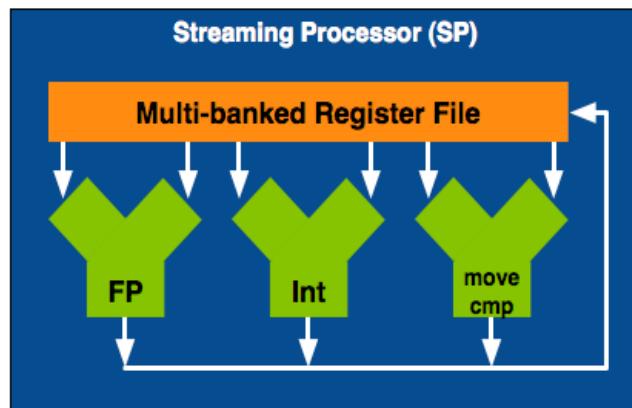
G80 (cc 1.0): 2007 (Geforce 8800, ...)

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



G80: first CUDA GPU!

Multiprocessor: SM (CC 1.x)

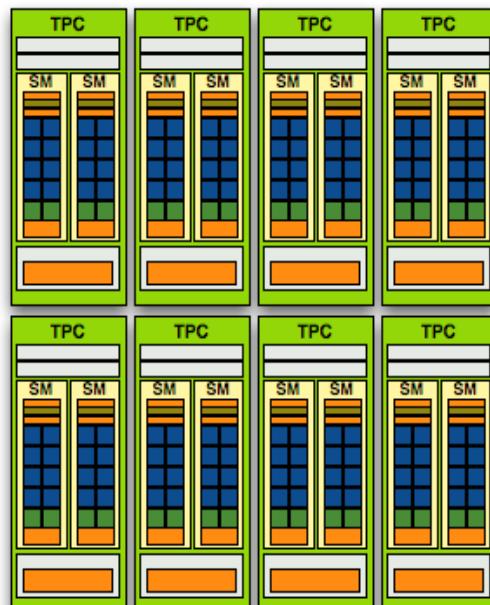
Courtesy AnandTech

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

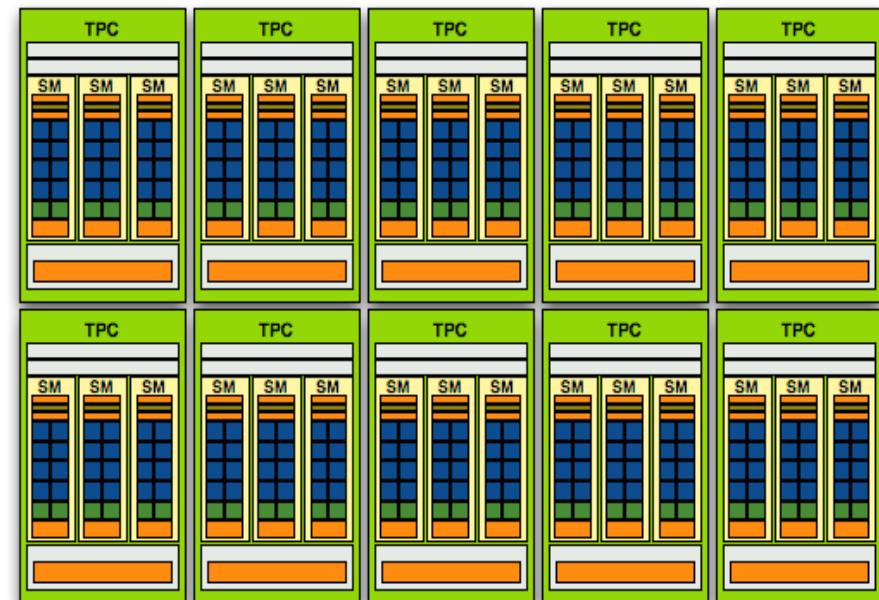
NVIDIA Tesla Architecture (not the Tesla product line!), G80: 2007, GT200: 2008/2009



- G80/G92: $8 \text{ TPCs} * (2 * 8 \text{ SPs}) = 128 \text{ SPs}$ [= CUDA cores]
- GT200: $10 \text{ TPCs} * (3 * 8 \text{ SPs}) = 240 \text{ SPs}$ [= 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)

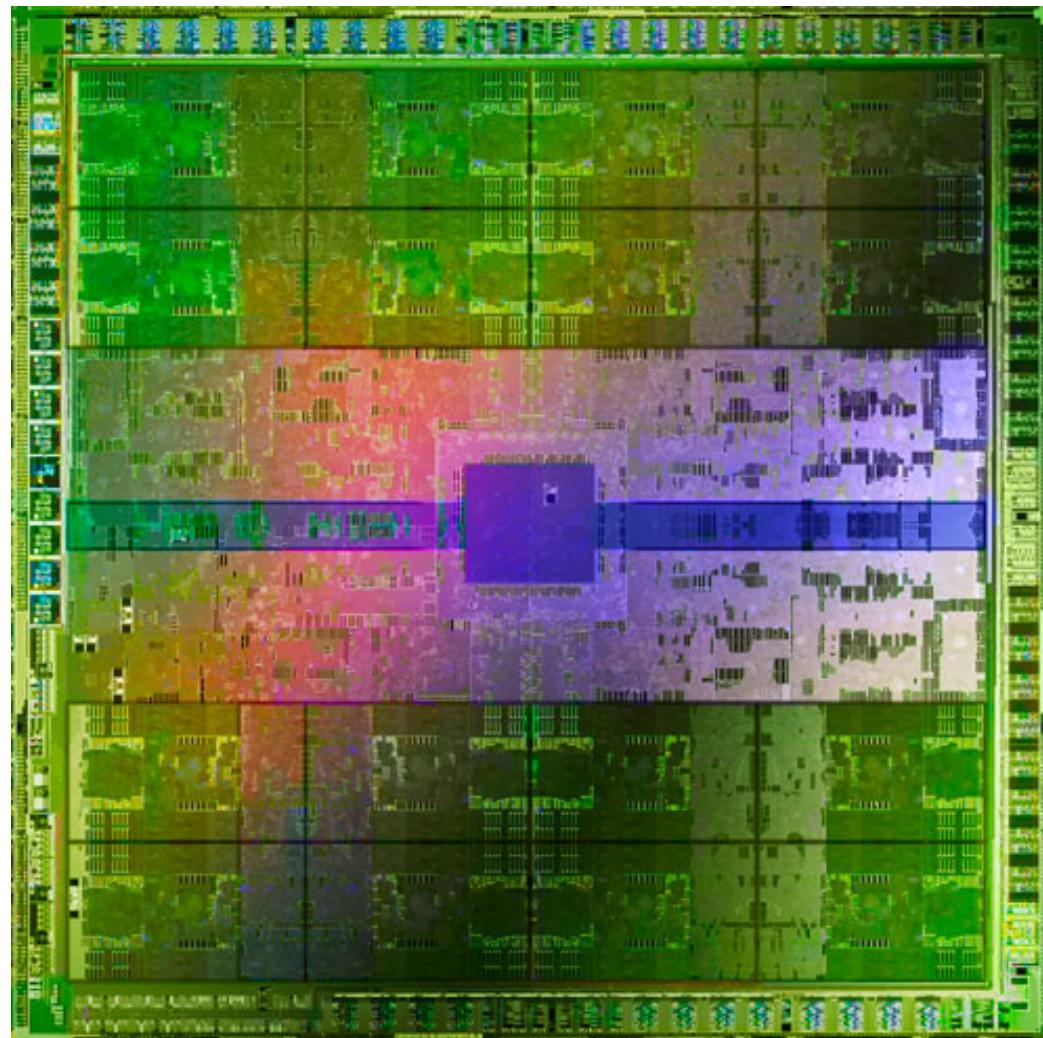


NVIDIA Fermi (GF100) Die Photo



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.x (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	?
SM busy with # warps	22	22 + ILP	44 + ILP	36	12	24	16	16	4*?

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
streaming multiprocessor (SM)

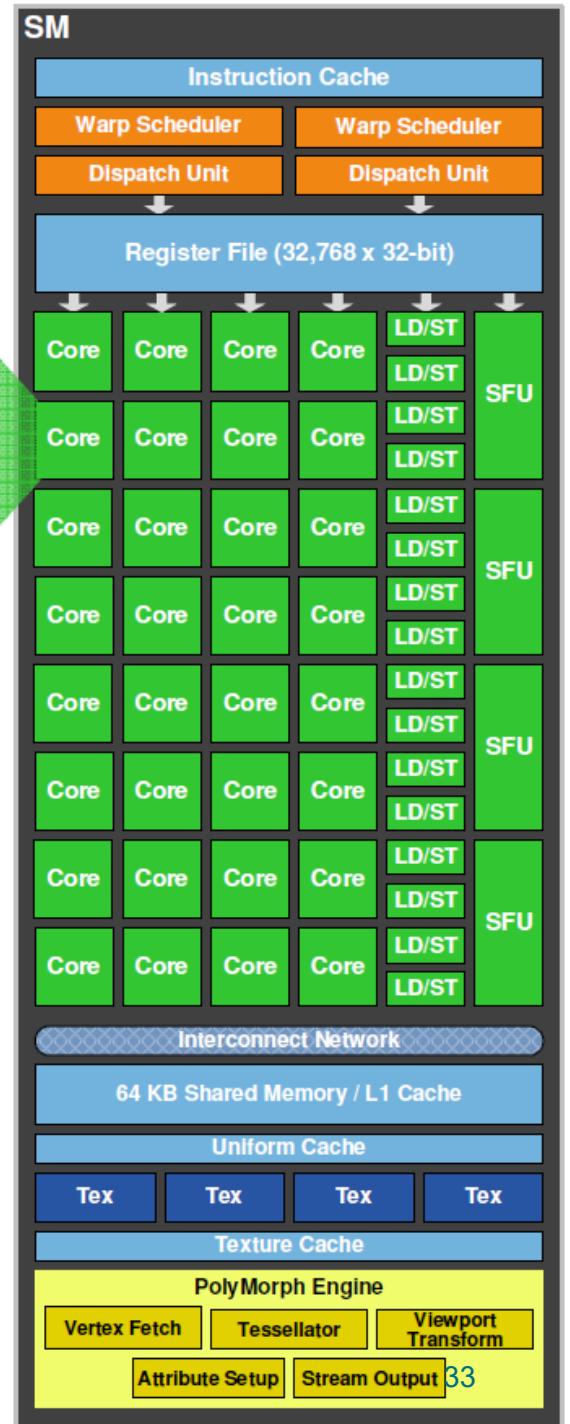
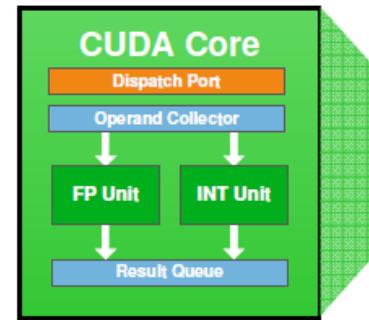
16 SMs = 512 CUDA cores

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)

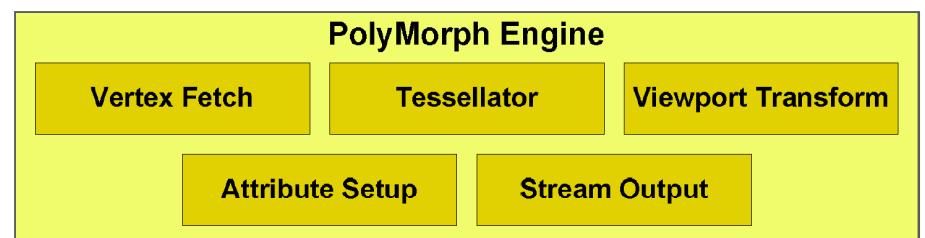
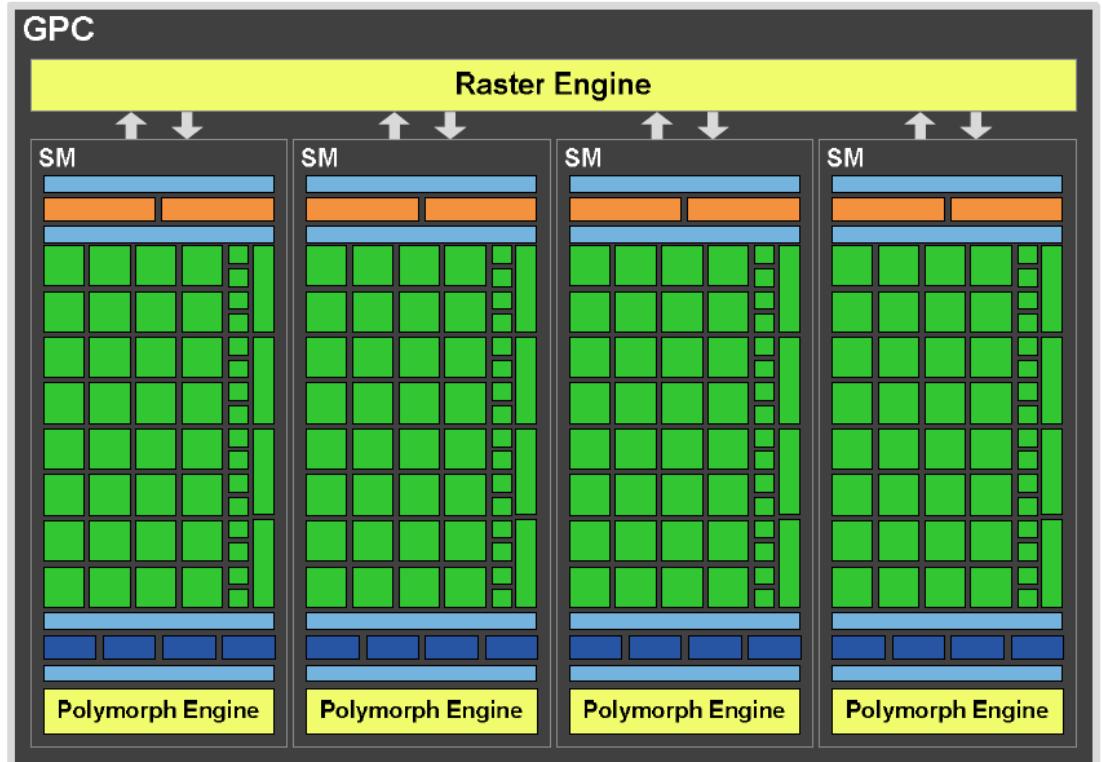
4 SMs

32 CUDA cores / SM

4 SMs / GPC =
128 cores / GPC

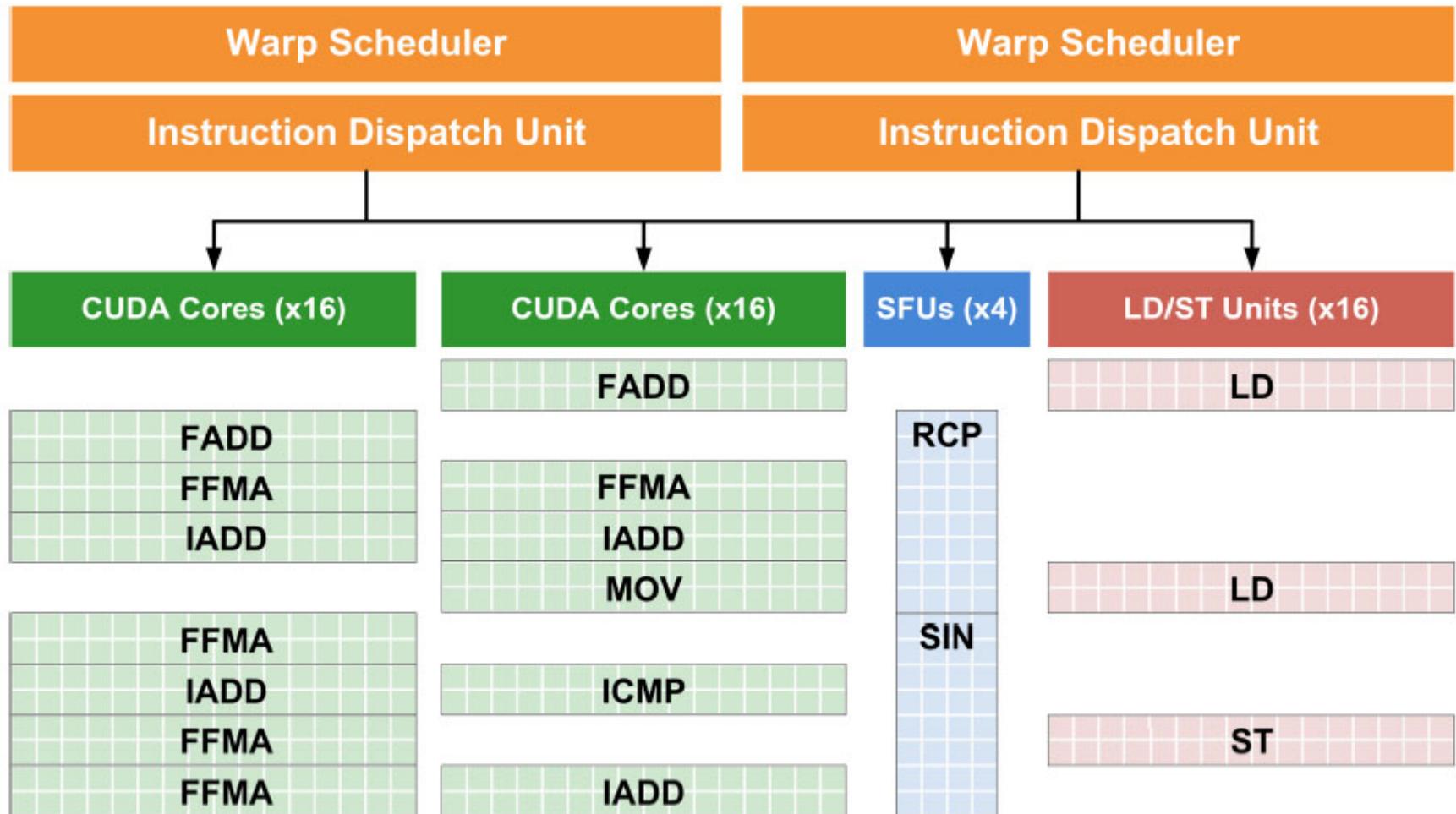
Decentralized rasterization
and geometry

- 4 raster engines
- 16 "PolyMorph" engines



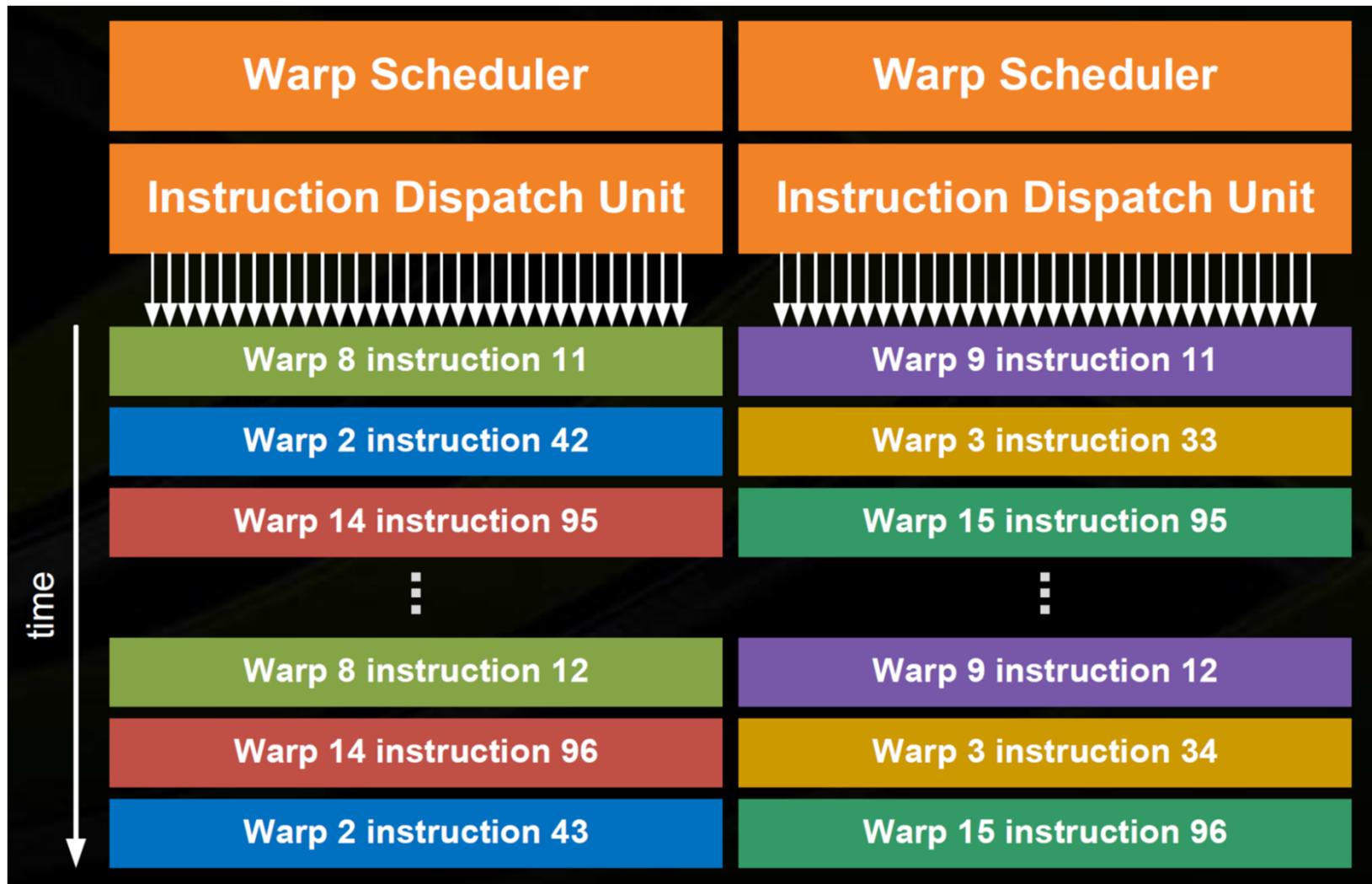


Dual Warp Schedulers





Dual Warp Schedulers





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.x (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	?
SM busy with # warps	22	22 + ILP	44 + ILP	36	12	24	16	16	4*?

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)

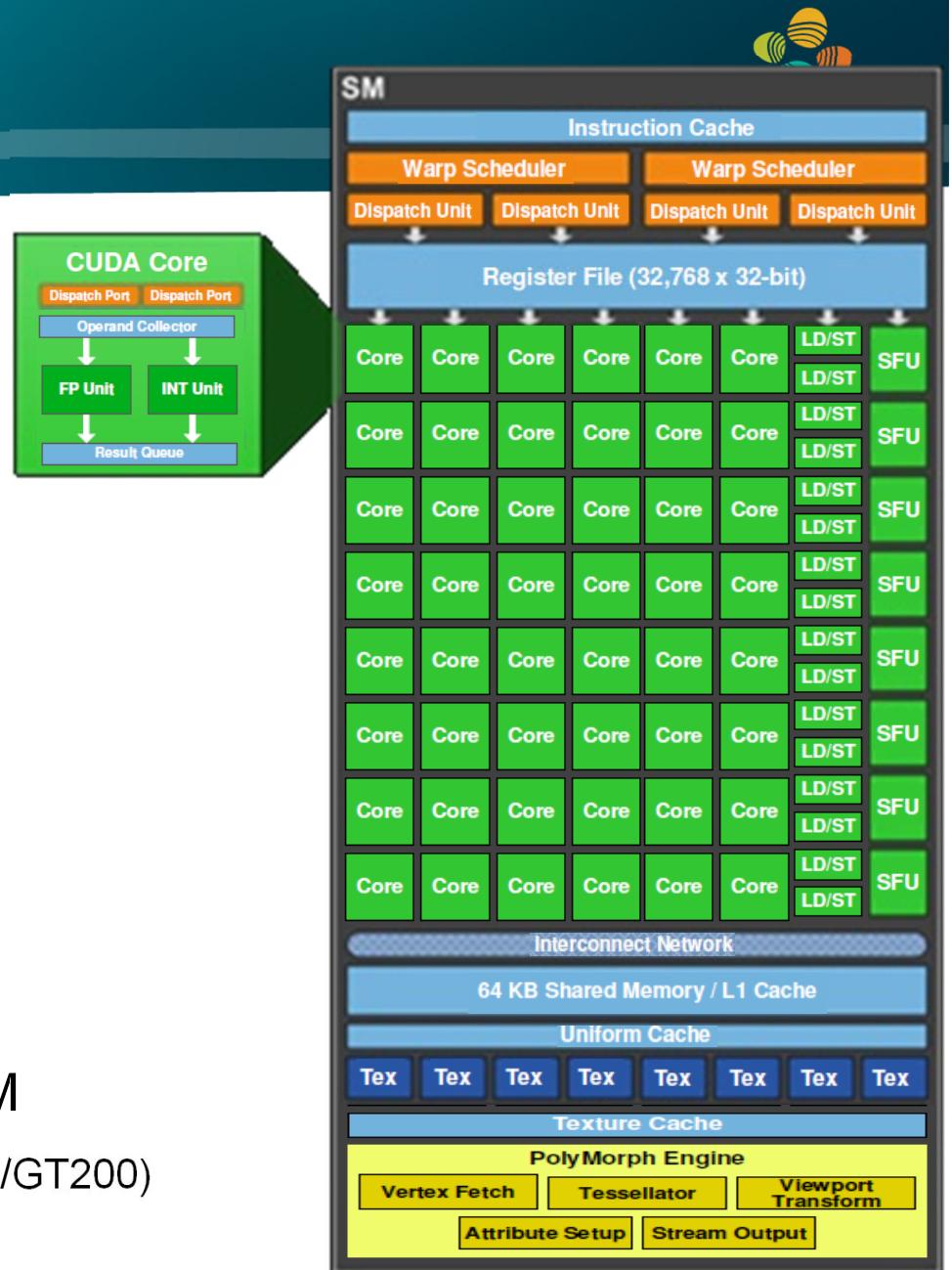
16 SMs = 512 CUDA cores

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)

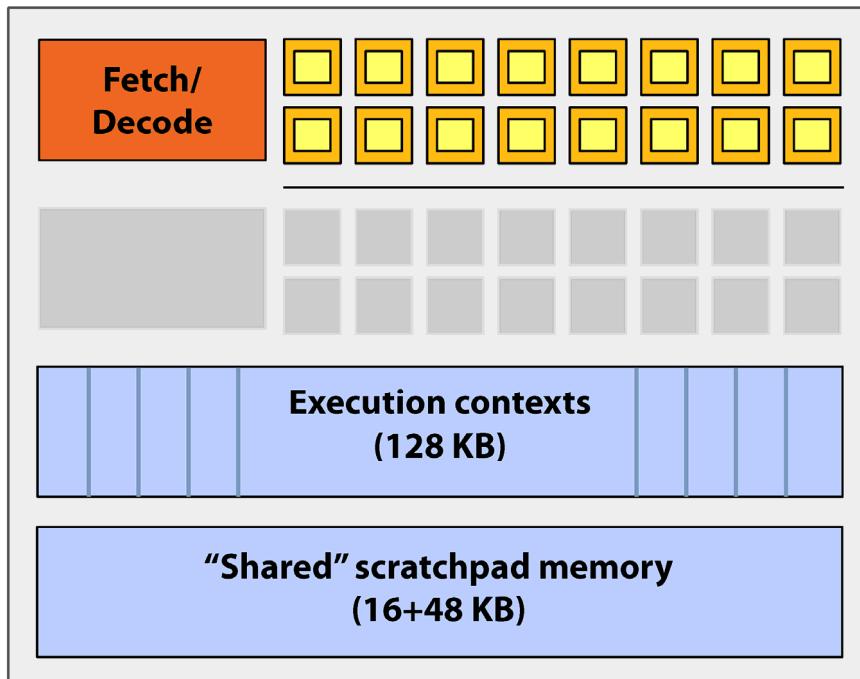


NVIDIA Fermi GF100 Architecture (2010)



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

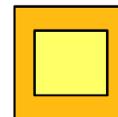
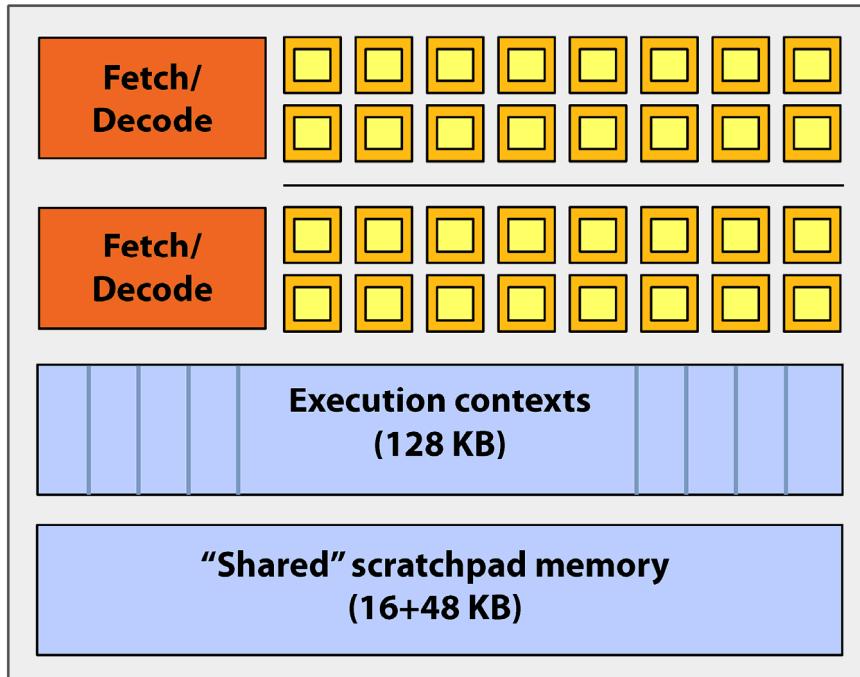
Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

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)

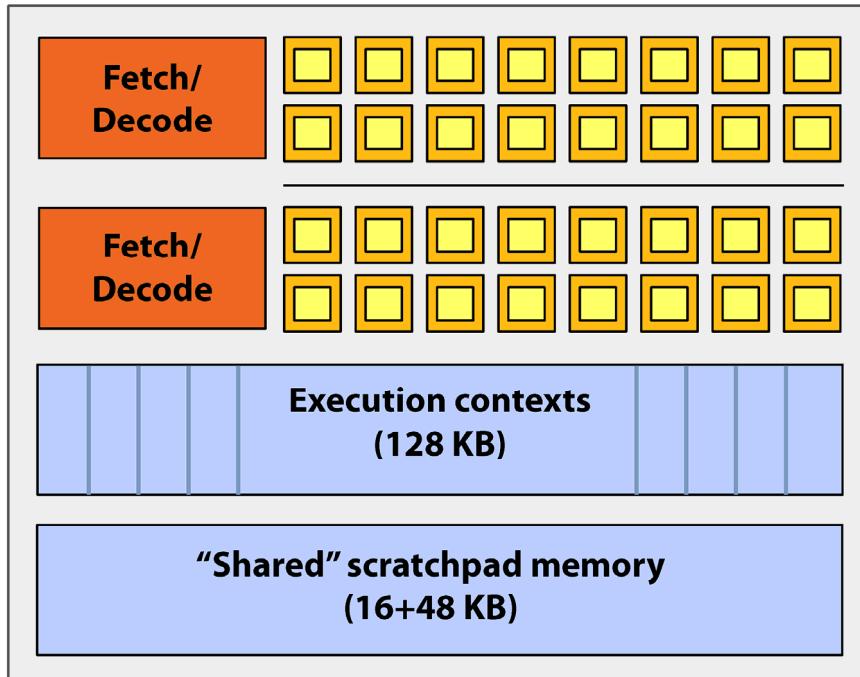
Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

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**

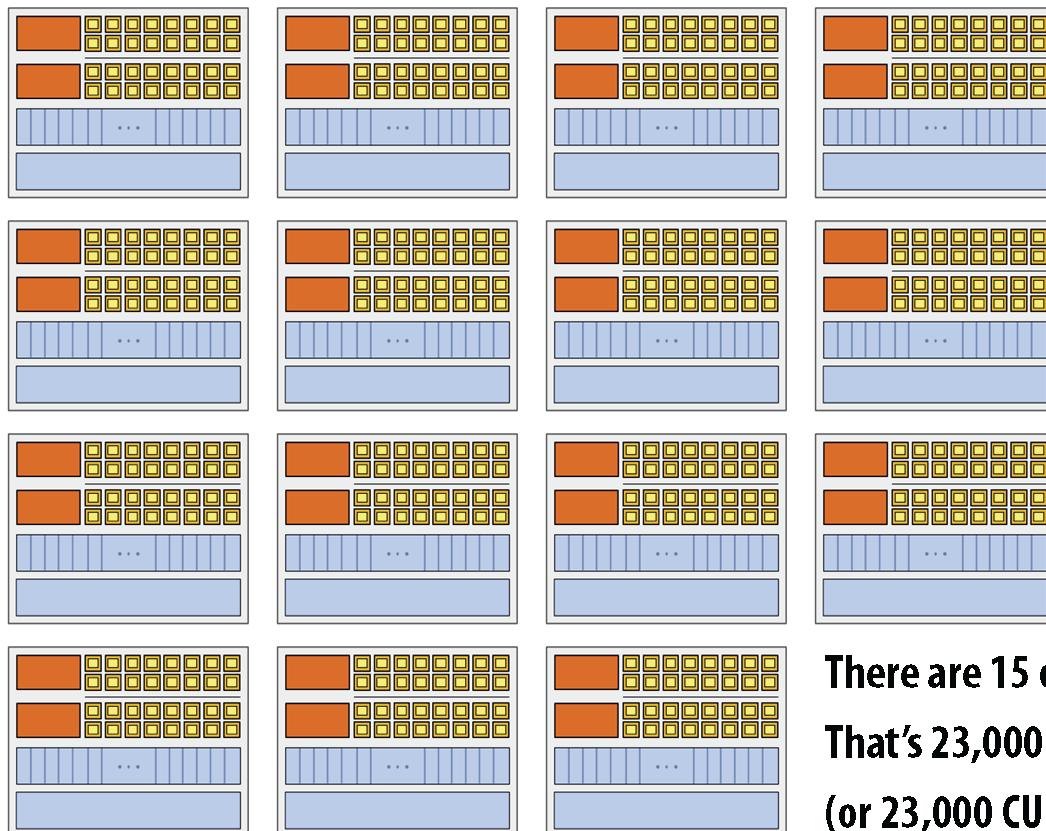
Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

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 Ampere Architecture

2020

(compute capability 8.0/8.6/8.7)

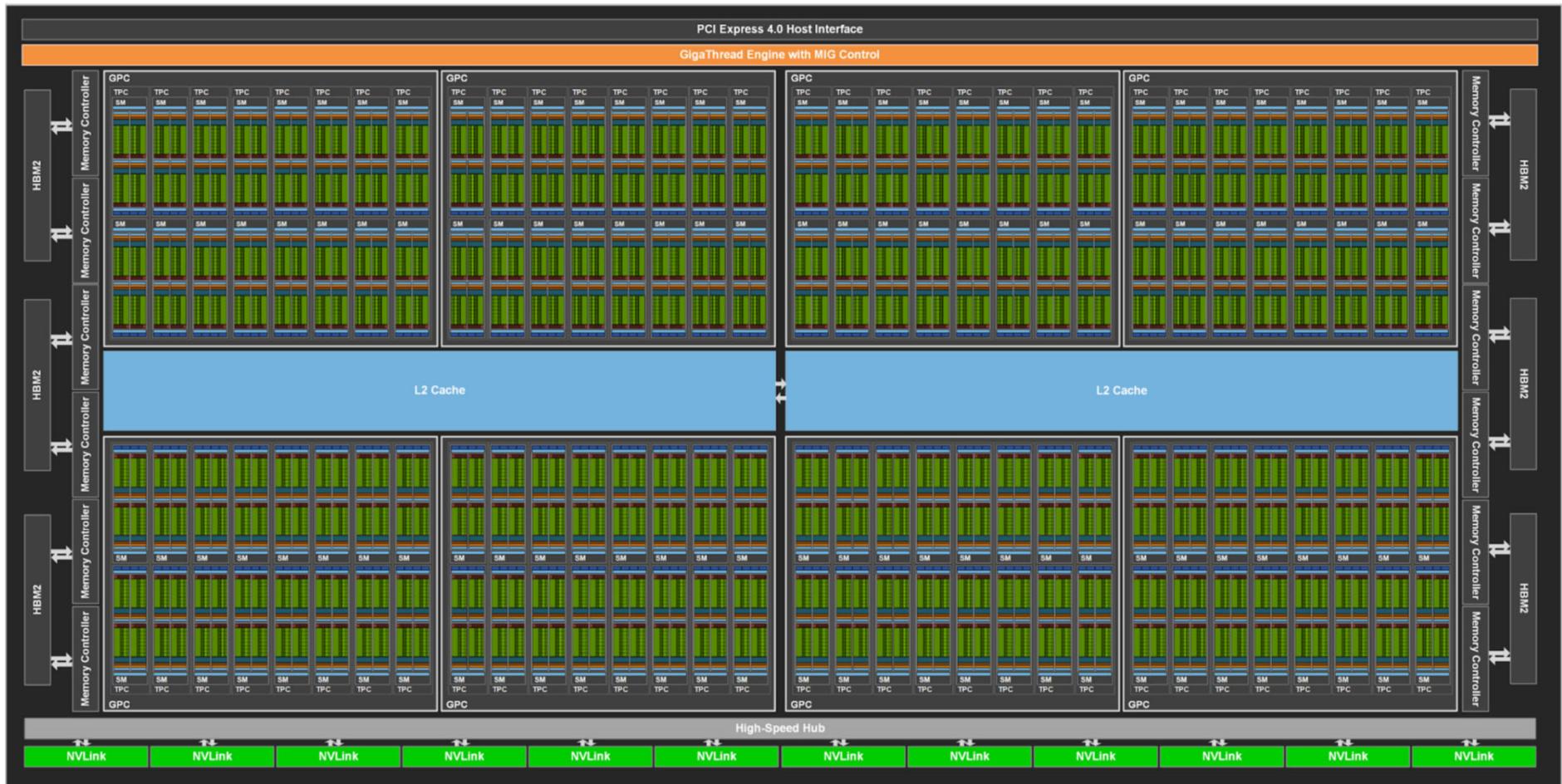
- GA100 (cc 8.0), ... (A100, ...)
- (x=2,3,4,6,7) GA10x (cc 8.6), ... (RTX 3070, RTX 3080, RTX 3090, ...)
- GA10B (cc 8.7), ... (Jetson, DRIVE, ...)

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 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
16-bit floating-point add, multiply, multiply-add	N/A		256	128	2	256	128	256 ³	
32-bit floating-point add, multiply, multiply-add	192		128	64		128	64	128	
64-bit floating-point add, multiply, multiply-add	64 ⁴		4	32	4		32 ⁵	32	2

³

⁴

⁵

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.x (Ampere)	8.9/9.x (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	?
SM busy with # warps	22	22 + ILP	44 + ILP	36	12	24	16	16	4*?

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 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

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



Thank you.