

CS 380 - GPU and GPGPU Programming

Lecture 4: GPU Architecture, Pt. 2

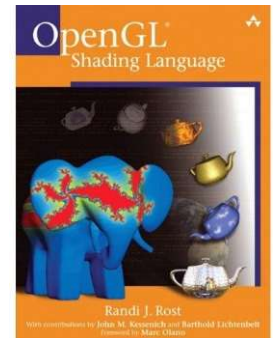
Markus Hadwiger, KAUST

Reading Assignment #2 (until Sep 15)



Read (required):

- Orange book (GLSL), Chapter 4
(*The OpenGL Programmable Pipeline*)
- Nice brief overviews of GLSL and legacy assembly shading language
https://en.wikipedia.org/wiki/OpenGL_Shading_Language
https://en.wikipedia.org/wiki/ARB_assembly_language
- Read:
https://en.wikipedia.org/wiki/Instruction_pipelining
https://en.wikipedia.org/wiki/Classic_RISC_pipeline
- Get an overview of NVIDIA Hopper and Blackwell GPU architectures:
<https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c>
<https://images.nvidia.com/aem-dam/Solutions/geforce/blackwell/nvidia-rtx-blackwell-gpu-architecture.pdf>
<https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/quadro-product-literature/NVIDIA-RTX-Blackwell-PRO-GPU-Architecture-v1.0.pdf>



Read (optional):

- GPU Gems 2 book, Chapter 30
(*The GeForce 6 Series GPU Architecture*)
http://download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch30.pdf

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, AD103, AD104, AD106, AD107, ...
(H100, L40, 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

- GB100/102, GB200, GB202/203/205/206/207, ...
(RTX 5080/5090, GB200 NVL72, HGX B100/200,
RTX 6000 PRO Blackwell, ...)

NVIDIA Blackwell SM

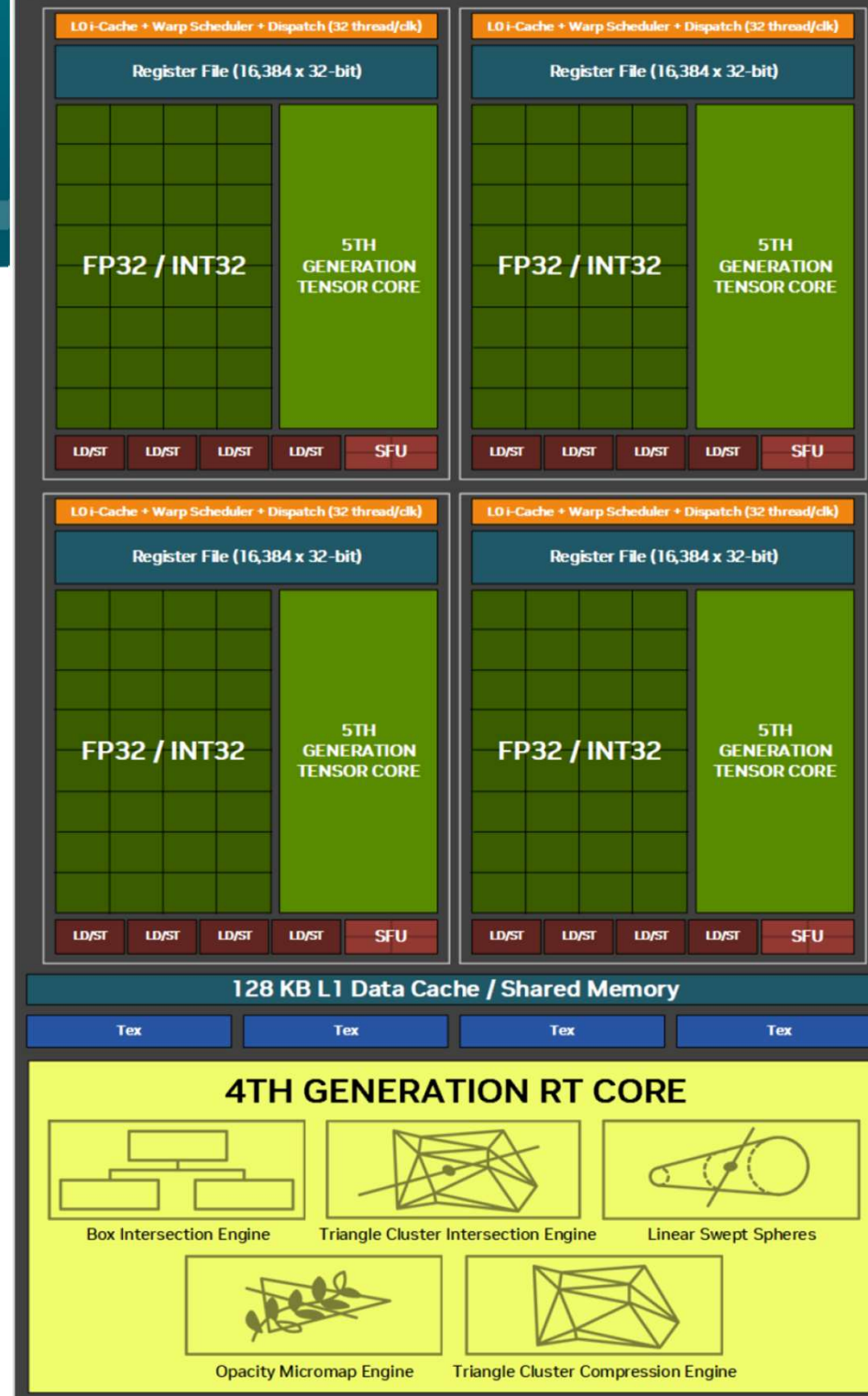
CC 12.0 SM (GB 202 Multiprocessor)

- 128 FP32/INT32 cores
- 2 FP64 cores
- 4x 5th gen tensor cores
- ++ thread block clusters, DPX insts., FP8, NVFP4, TMA

4 partitions inside SM

- 32 FP32/INT32 cores
- 4x LD/ST units each
- 1x 5th gen tensor core
- Each has: warp scheduler, dispatch unit, 16K register file

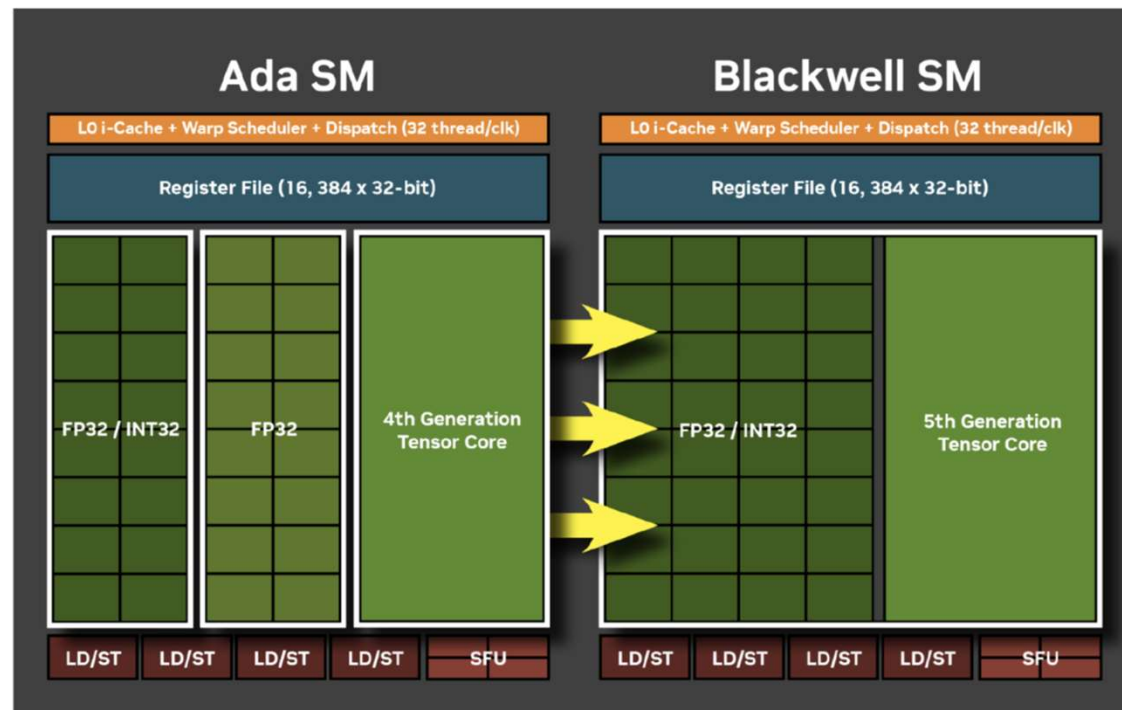
SM



NVIDIA Blackwell FP32 / INT32 Cores



Blackwell: FP32 and INT32 cores are unified, and the number of FP32 and INT32 cores is the same. (Earlier architectures: many unified, some not, e.g., until Pascal FP/INT was unified. Later (Volta or newer) depends on specific architecture)



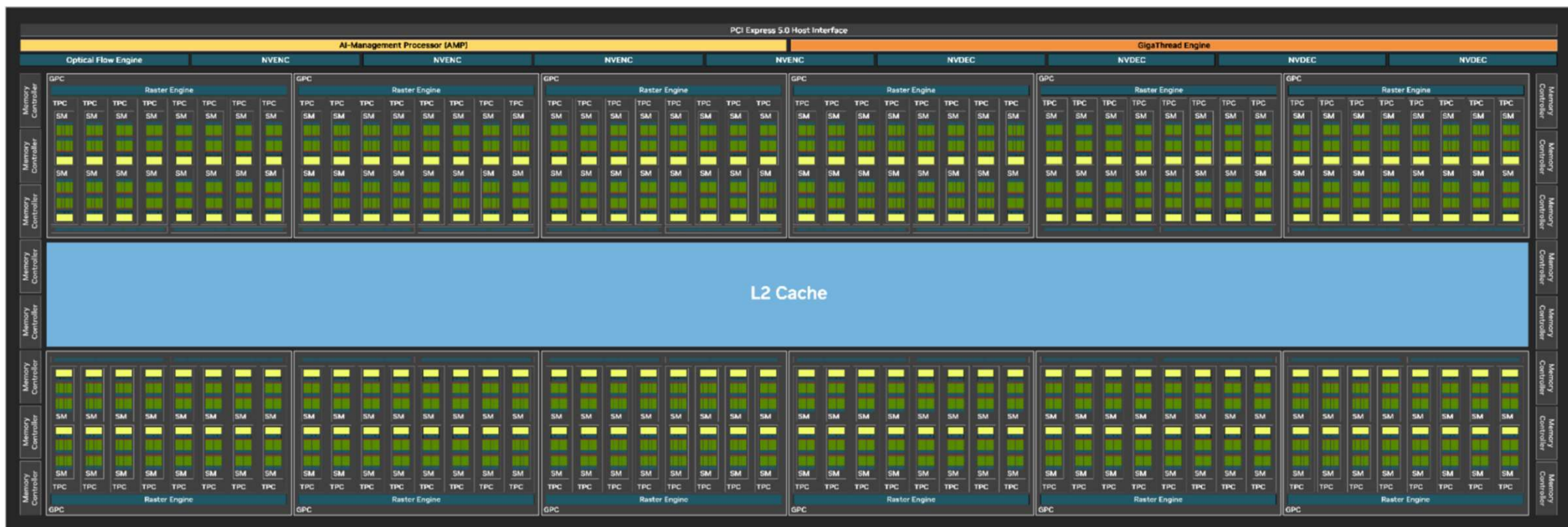
Beware: the current CUDA 13 Best Practices Guide (Sep 2, 2025) states the corresponding throughput numbers correctly in Table 5, but the current CUDA 13 C Programming Guide (Sep 2, 2025) states them incorrectly in Chapter 20! For CC 12.0, the stated amount of L1/shared memory is also wrong.

NVIDIA Blackwell GB202 Architecture (2025)



GB 202 (RTX GPU)

Full GPU: 192 SMs in 12 GPCs, 96 TPCs,
(24,576 FP32 cores)



NVIDIA Blackwell GB202 Architecture (2025)



GB 202 (RTX GPU)

Full GPU: 192 SMs in 12 GPCs, 96 TPCs
(RTX 5090: 170 SMs, 11 GPCs, 85 TPCs)

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

For 192 SMs on full GPU (RTX PRO 6000: 12 GPCs, 94 TPCs, 188 SMs = 24,064 FP32 cores)

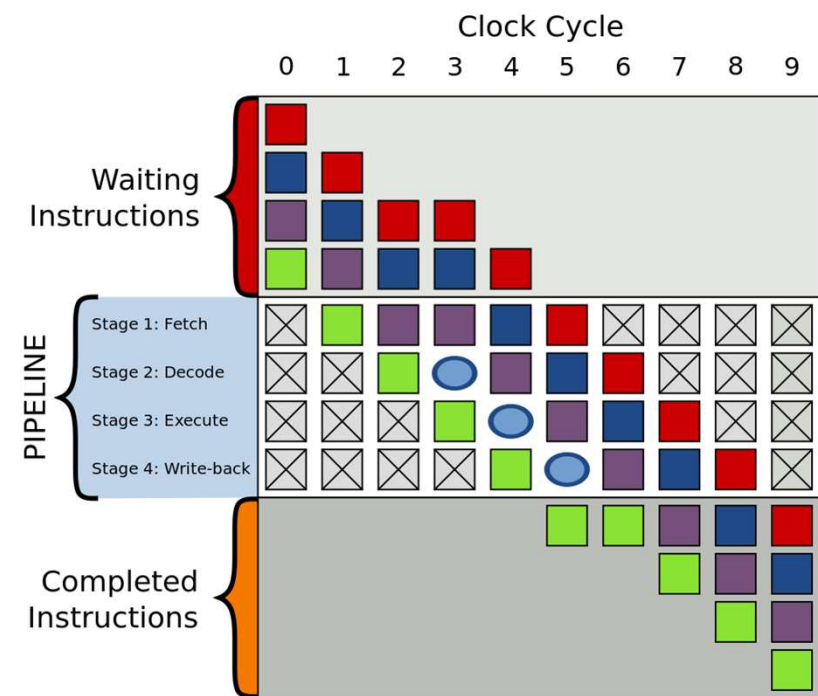
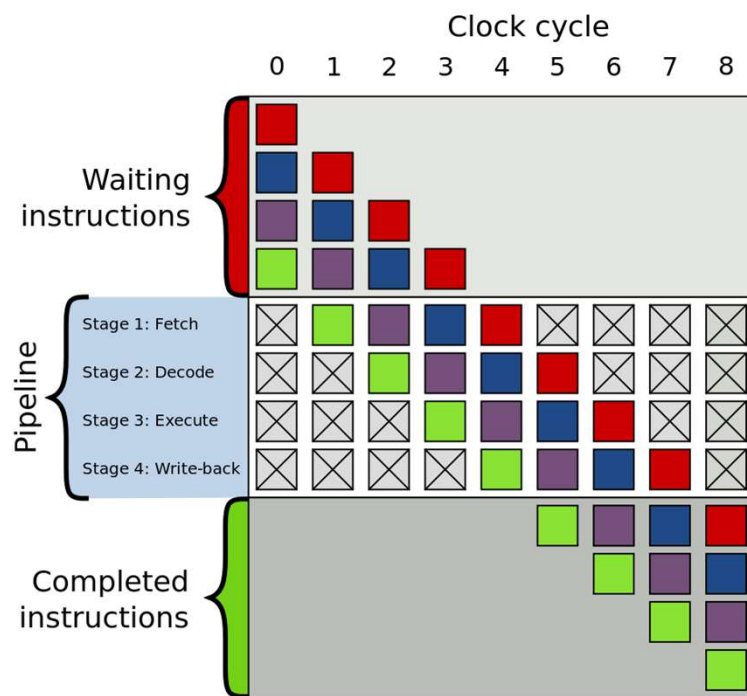
- 48 MB register storage, 24 MB shared mem / L1 storage =
72 MB context+”shared context” storage !
- L2 cache size 128 MB (e.g., RTX PRO 6000) (RTX 5090: 96 MB, RTX 5080: 64 MB)
- 24,576 FP32 cores (128 FP32 cores per SM), 768 tensor cores
(RTX PRO 6000: 752 tensor cores; RTX 5090: 170 SMs = 21,760 FP32 cores, 680 tensor cores)
- 294,912 max threads in flight (max warps / SM = 48)

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

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 CUDA 13 C Best Practices Guide (Chapter 12.1, Table 5)

Compute Capability	7.5 Turing	8.0 Ampere	8.6 Ada	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 ⁴ 64 for __nv_bfloat16	64		128	64	
	³ multiple instructions for __nv_bfloat16						
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

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 Ada	8.9 Hopper	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 ⁴ 64 for __nv_bfloat16 ³ multiple instructions for __nv_bfloat16	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

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*

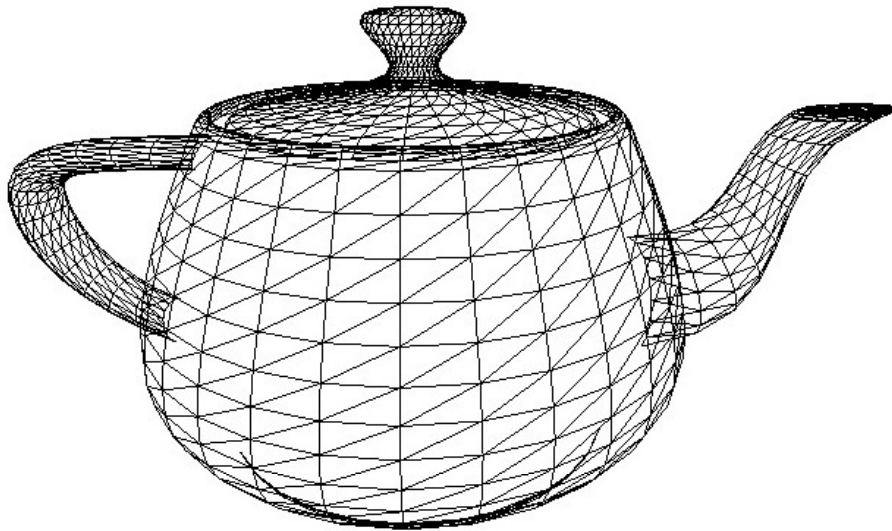
ALU Instruction Latencies and Instructs. / SM



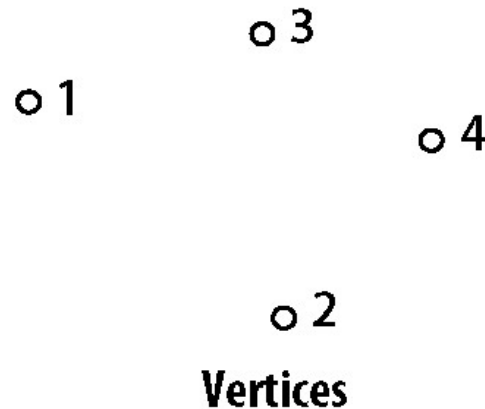
CC	2.0 (Fermi)	2.1 (Fermi)	3.x (Kepler)	5.x (Maxwell)	6.0 (Pascal)	6.1/6.2 (Pascal)	7.x (Volta, Turing)	8.0/8.6 (Ampere)	8.9/9.0 (Ada, Hopper)	10.x/12.x (Blackwell)
# warp sched. / SM	2	2	4	4	2	4	4	4	4	4
# ALU dispatch / warp sched.	1 (over 2 clocks)	2 (over 2 clocks)	2	1	1	1	1	1	1	1
SM busy with # warps + 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*

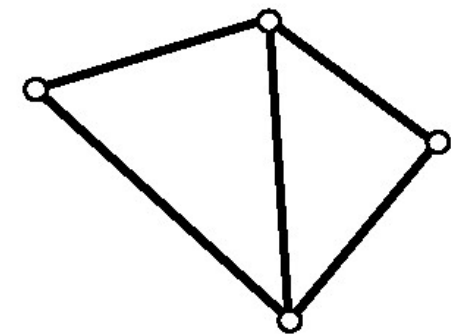
Real-time graphics primitives (entities)



Represent surface as a 3D triangle mesh



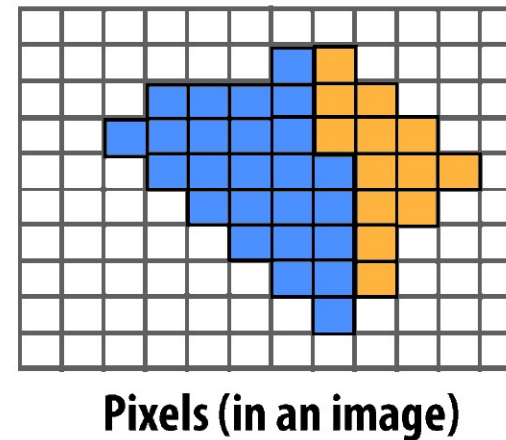
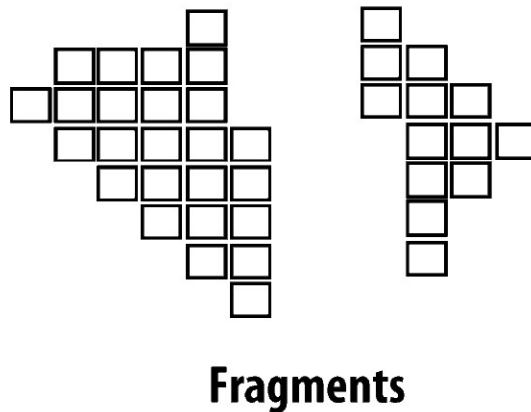
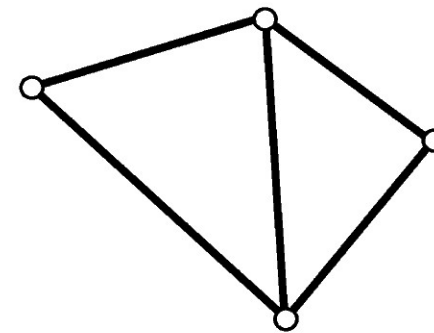
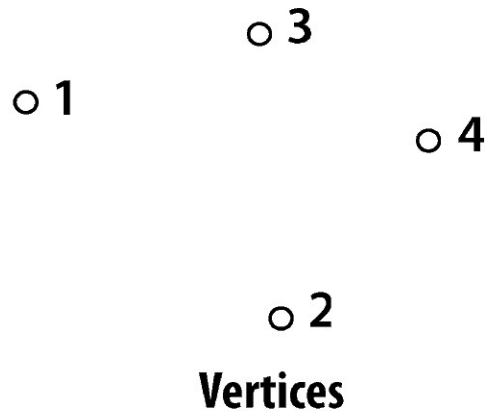
Vertices



Primitives

(e.g., triangles, points, lines)

Real-time graphics primitives (entities)



What can the hardware do?

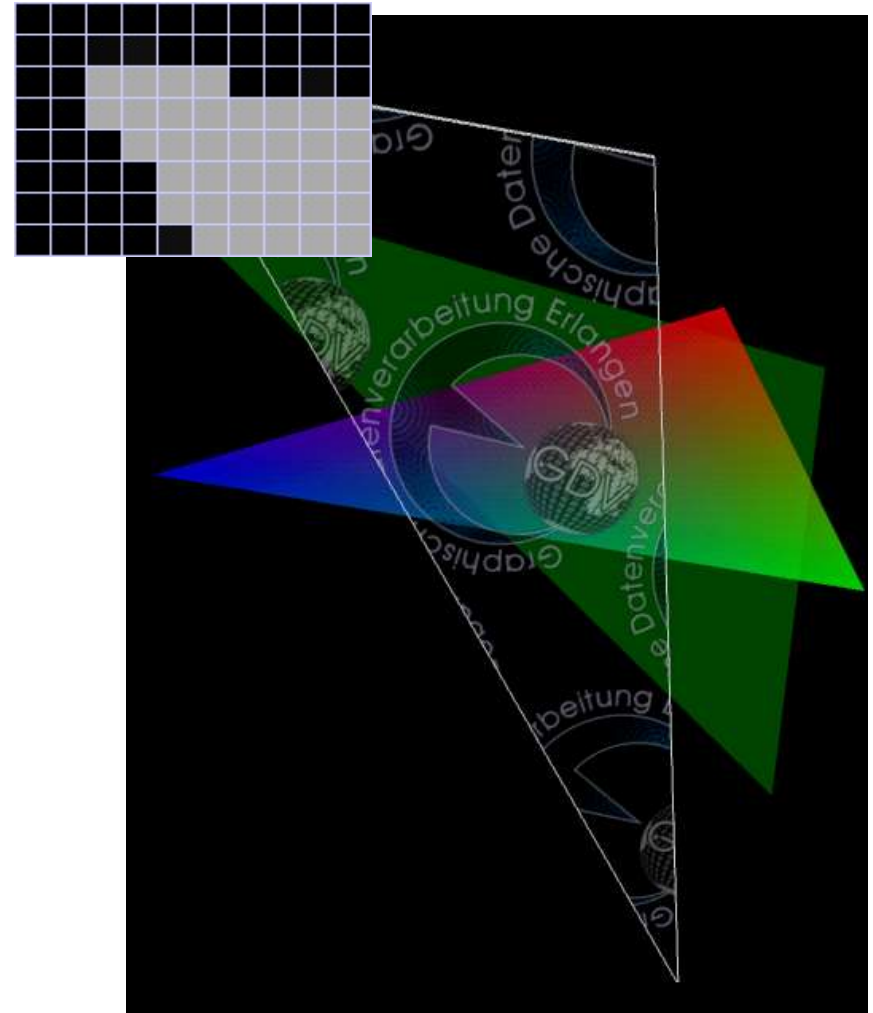


- **Rasterization**

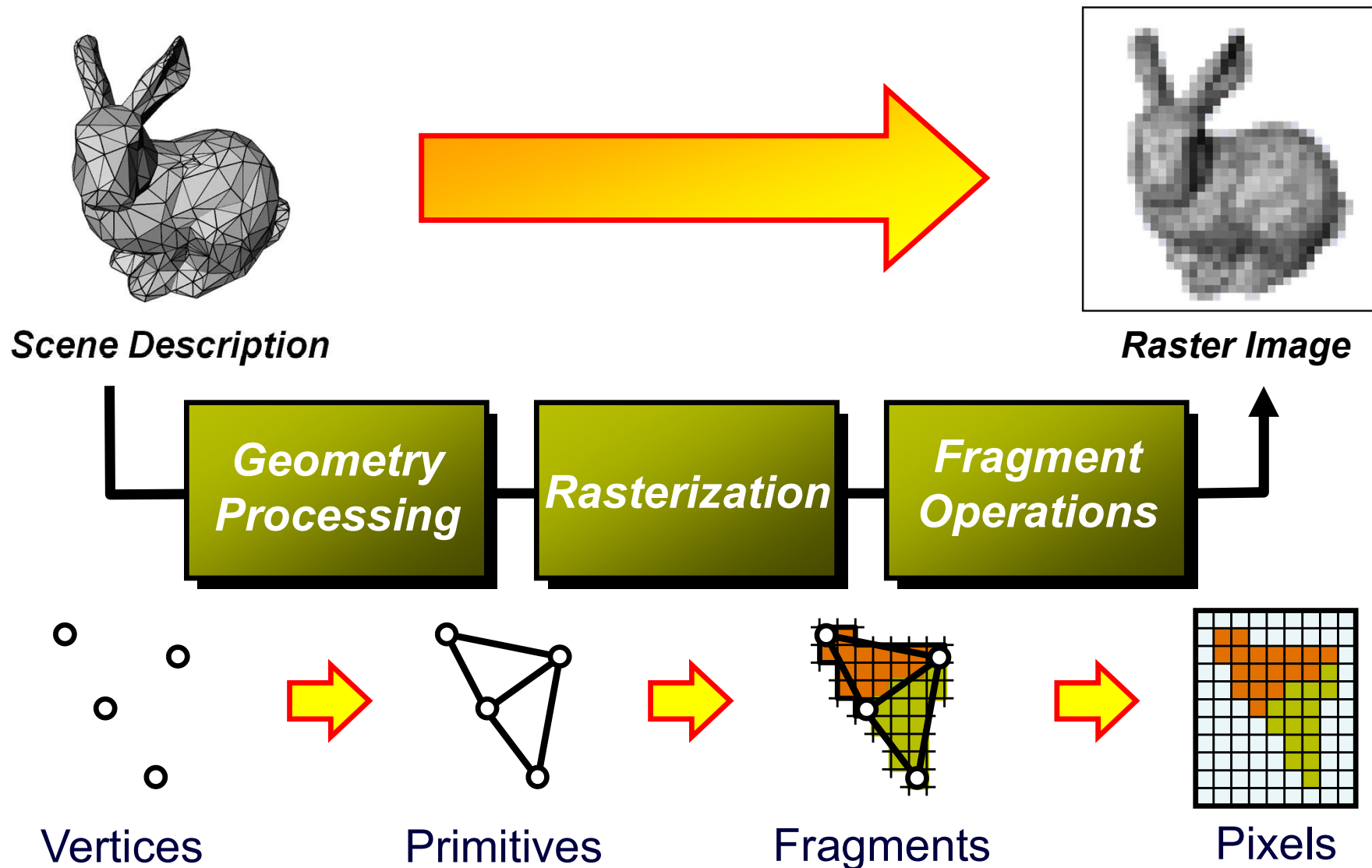
- Decomposition into fragments
- Interpolation of color
- Texturing
 - Interpolation/filtering
 - Fragment shading

- **Fragment operations
(or: raster operations)**

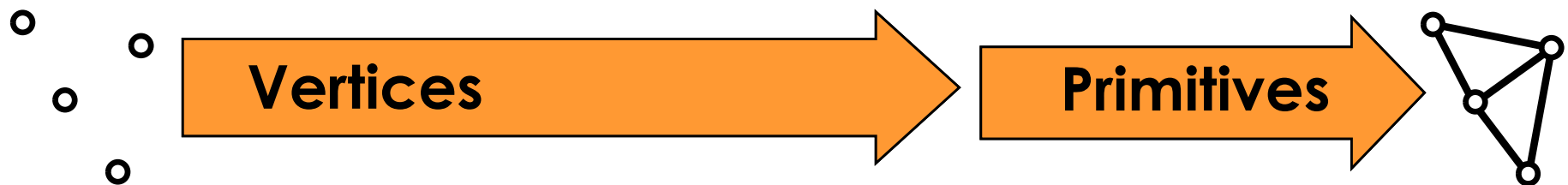
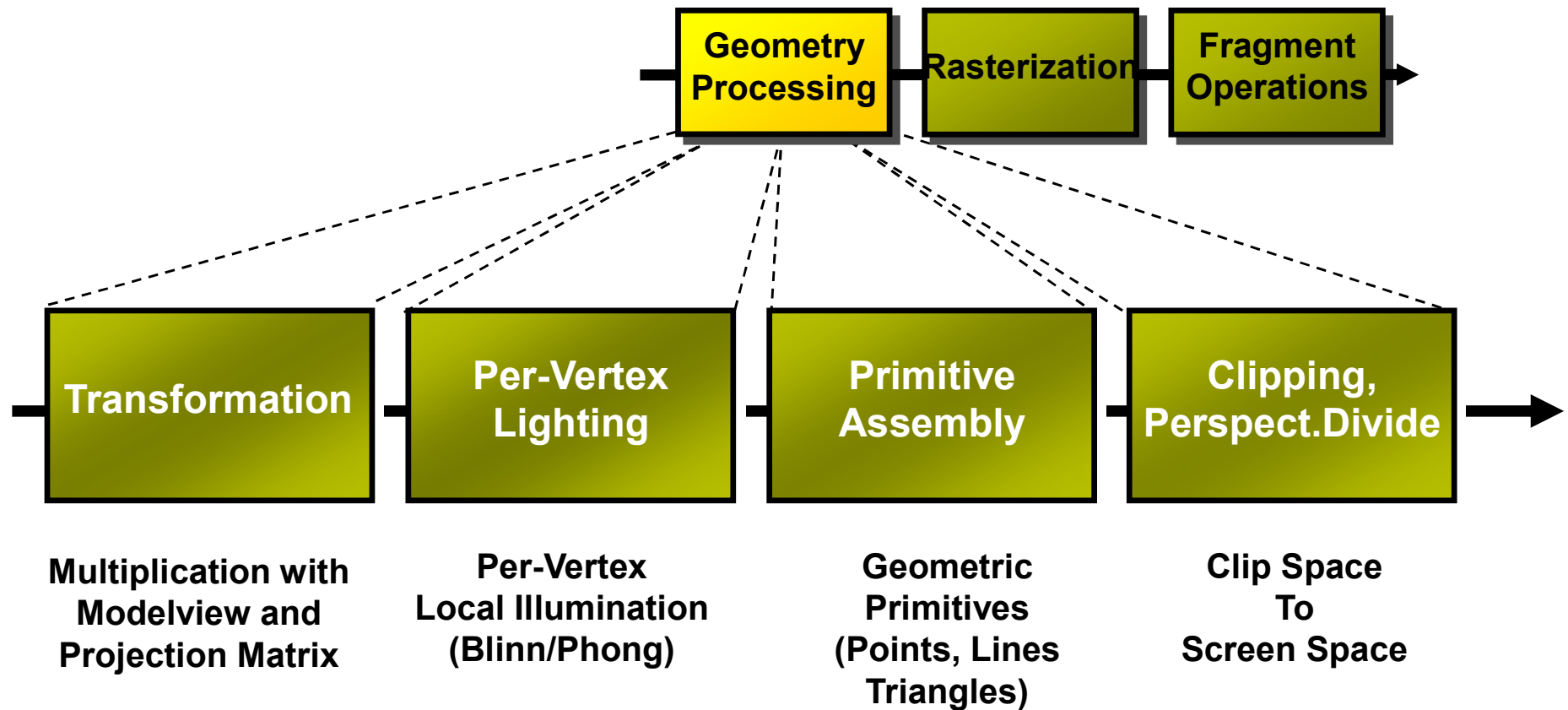
- Depth test (Z-test)
- Alpha blending (compositing)
- ...



Graphics Pipeline



Geometry Processing



Rasterization



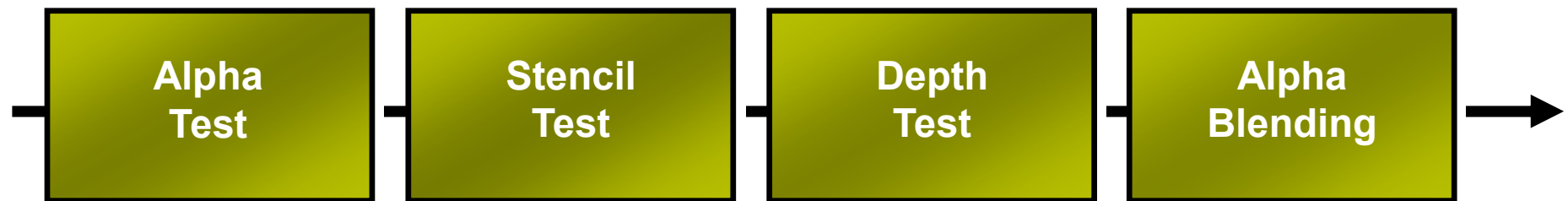
Decomposition
of primitives
into fragments

Interpolation of
texture *coordinates*
Filtering of
texture color

Combination of
primary color with
texture color



Fragment (Raster) Operations

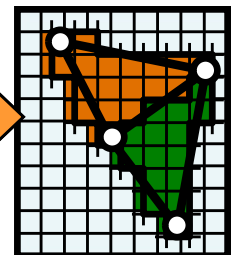
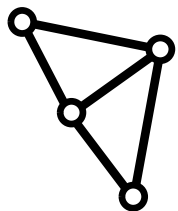


Discard all fragments within a certain alpha range

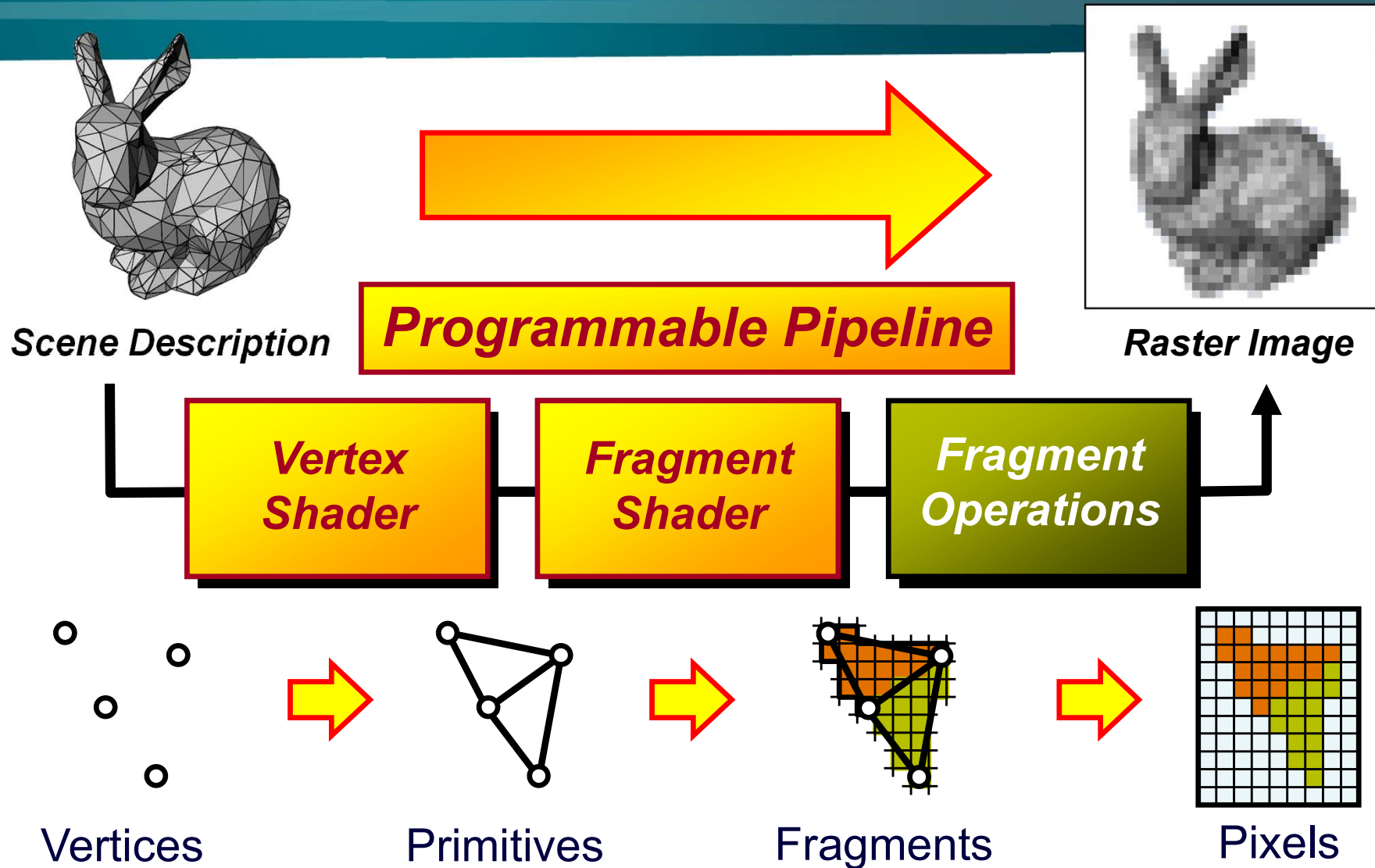
Discard a fragment if the stencil buffer is set

Discard all occluded fragments

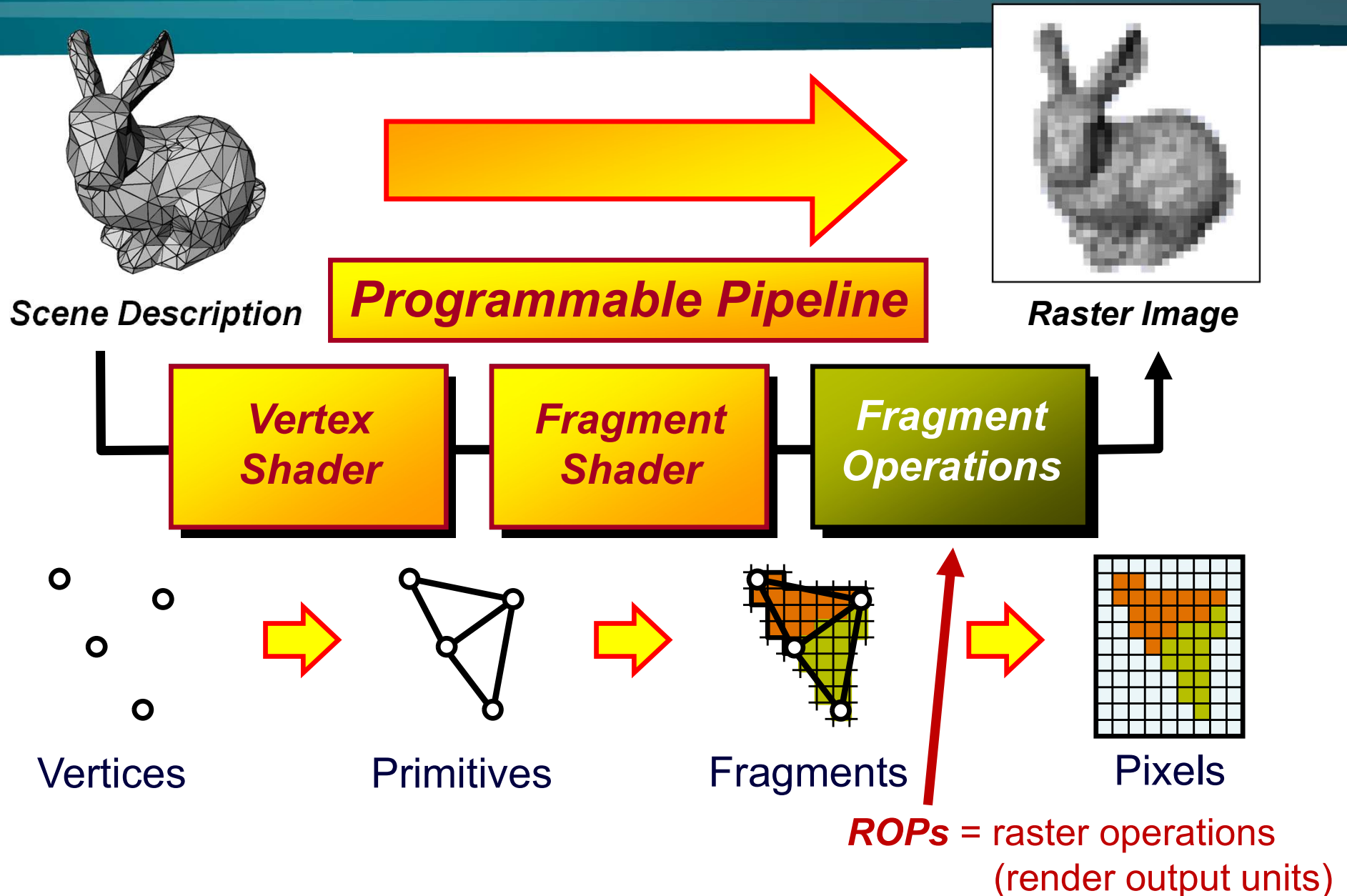
Combination of primary color with texture color



Graphics Pipeline

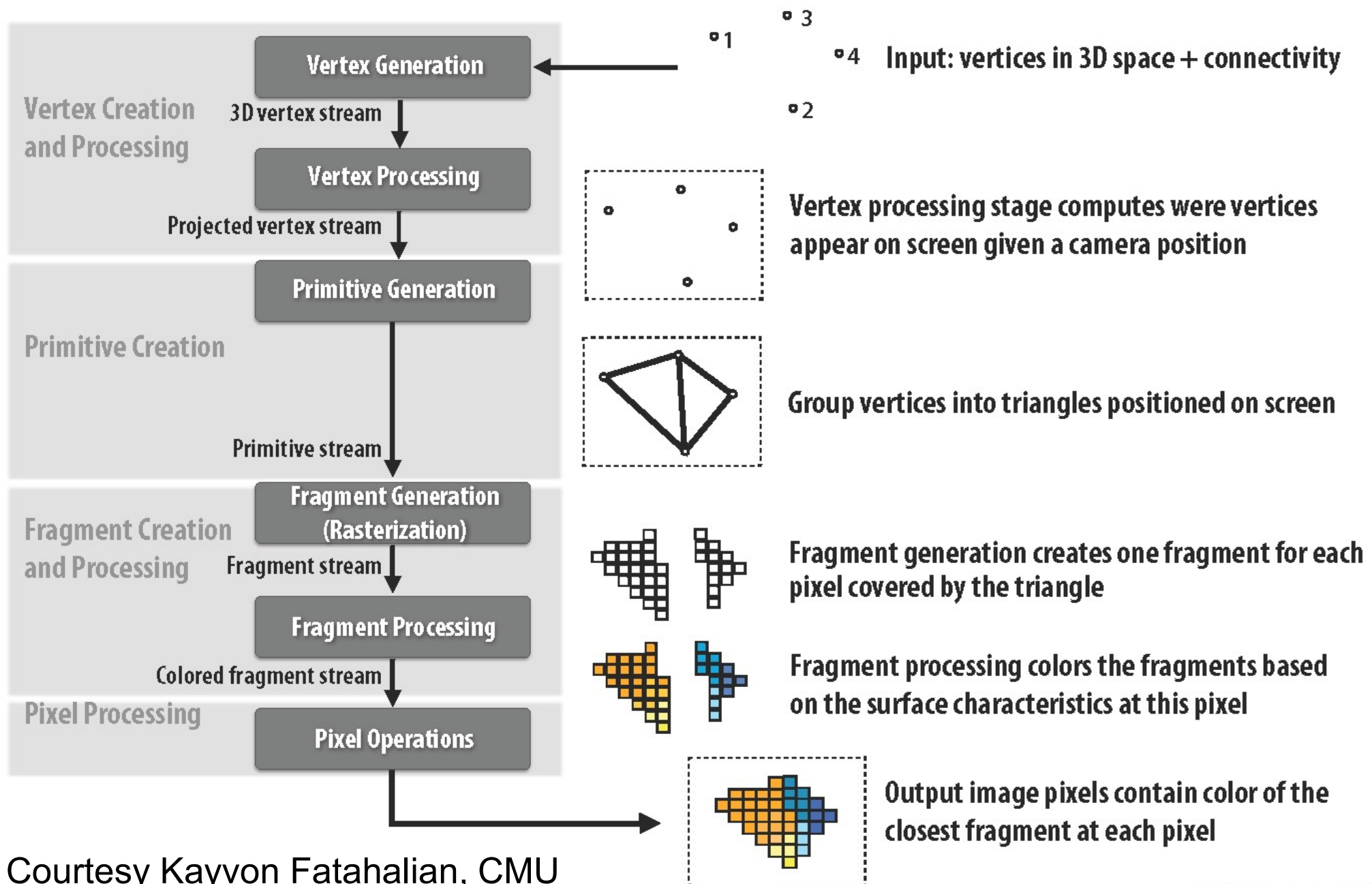


Graphics Pipeline



Graphics pipeline architecture

Performs operations on vertices, triangles, fragments, and pixels



Courtesy Kayvon Fatahalian, CMU

Direct3D 10 Pipeline (~OpenGL 3.2)



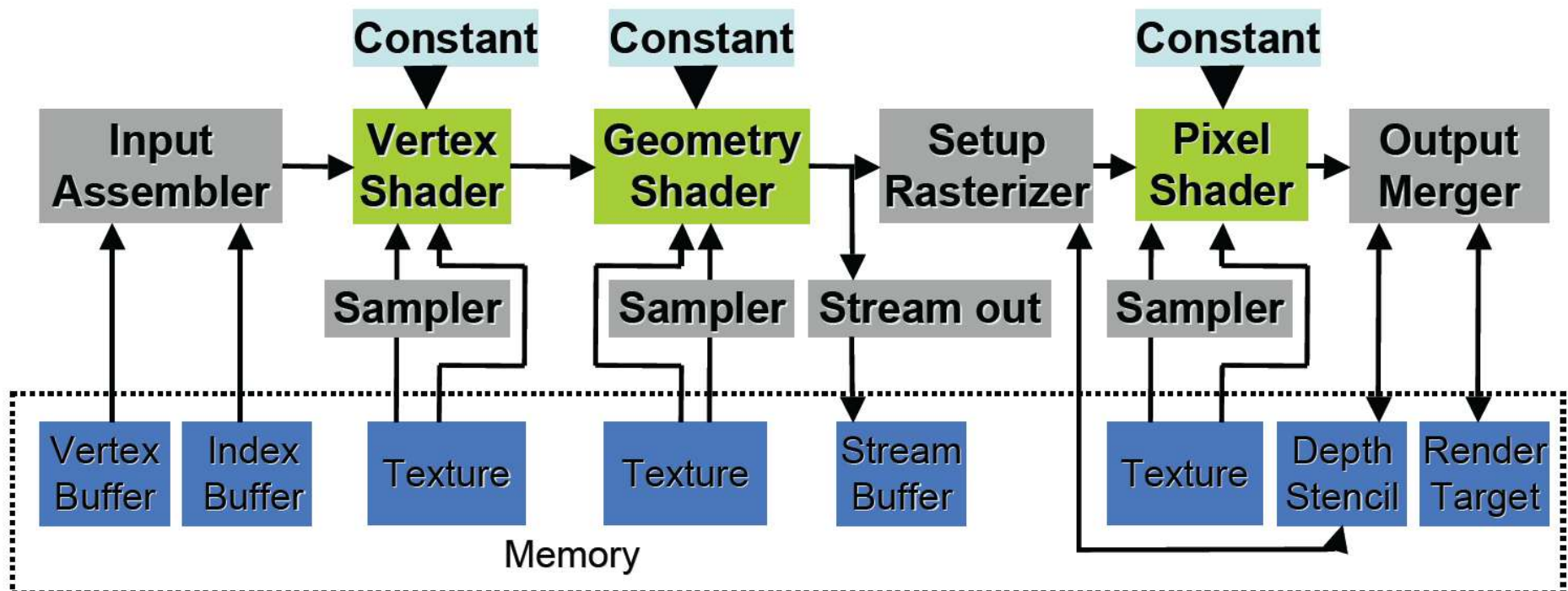
New geometry shader stage:

- Vertex -> geometry -> pixel shaders
- Stream output after geometry shader

fixed

programmable

memory



Direct3D 11 Pipeline (~OpenGL 4.x)

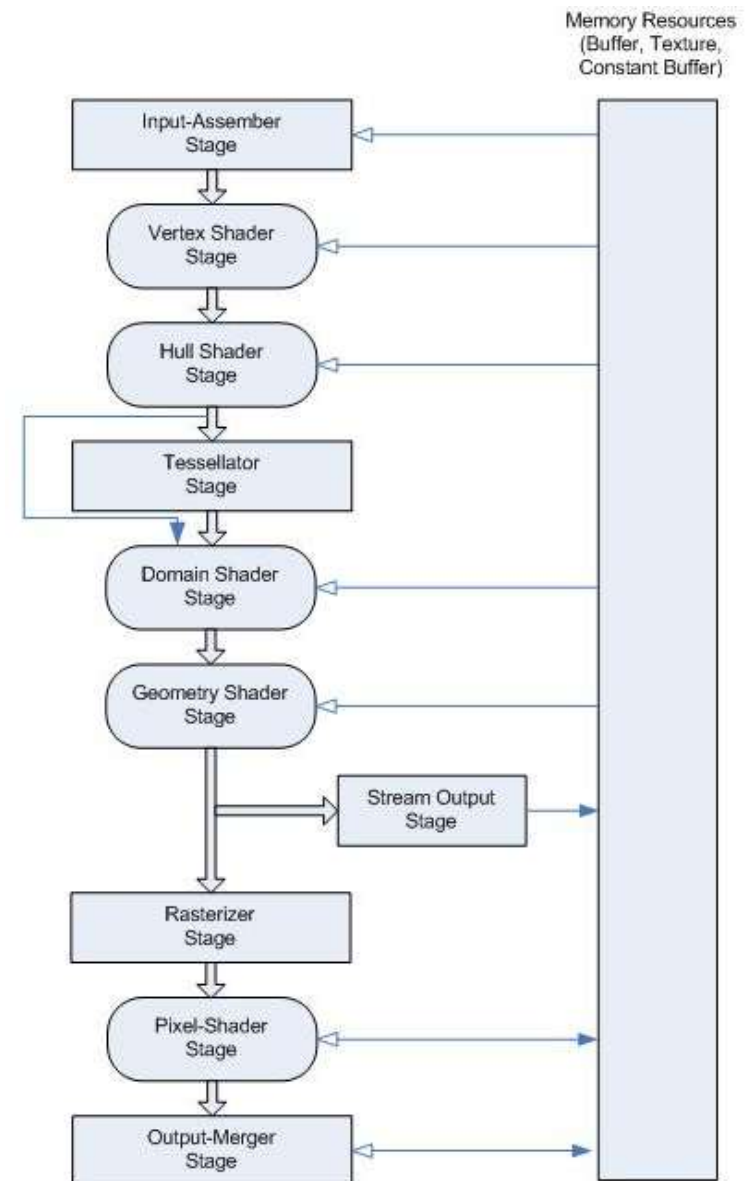


New tessellation stages

- Hull shader
(OpenGL: *tessellation control*)
- Tessellator
(OpenGL: *tessellation primitive generator*)
- Domain shader
(OpenGL: *tessellation evaluation*)

Outside this pipeline

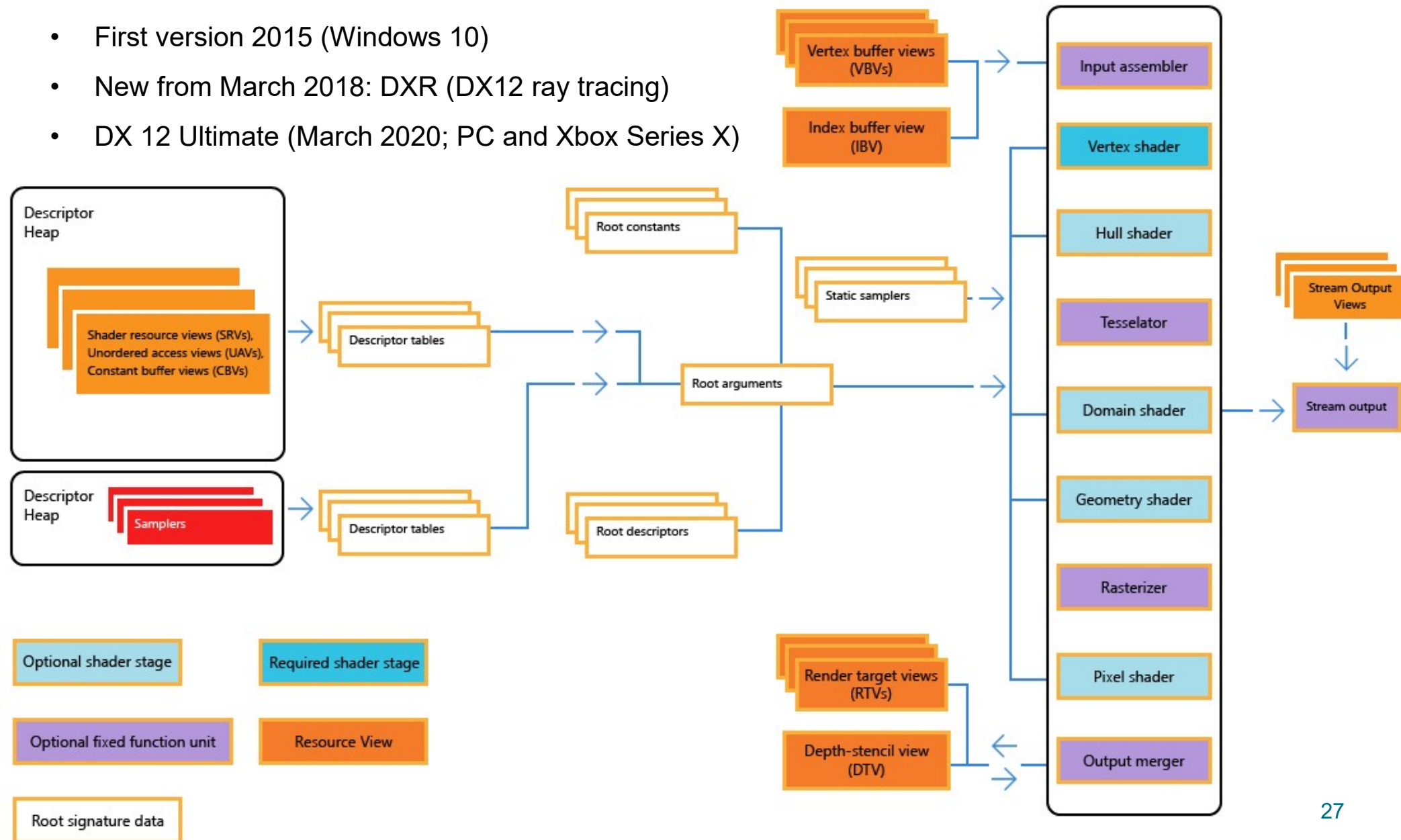
- Compute shader
- (Ray tracing cores, D3D 12)
- (Mesh shader pipeline, D3D 12.2)



Direct3D 12 Traditional Geometry Pipeline



- First version 2015 (Windows 10)
- New from March 2018: DXR (DX12 ray tracing)
- DX 12 Ultimate (March 2020; PC and Xbox Series X)



Direct3D 12 Mesh Shader Pipeline



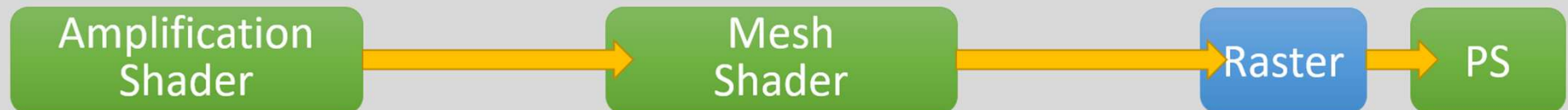
Reinventing the Geometry Pipeline

- Mesh and amplification shaders: new high-performance geometry pipeline based on compute shaders (DX 12 Ultimate / feature level 12.2)
- Compute shader-style replacement of IA/VS/HS/Tess/DS/GS

Legacy D3D12 graphics pipeline

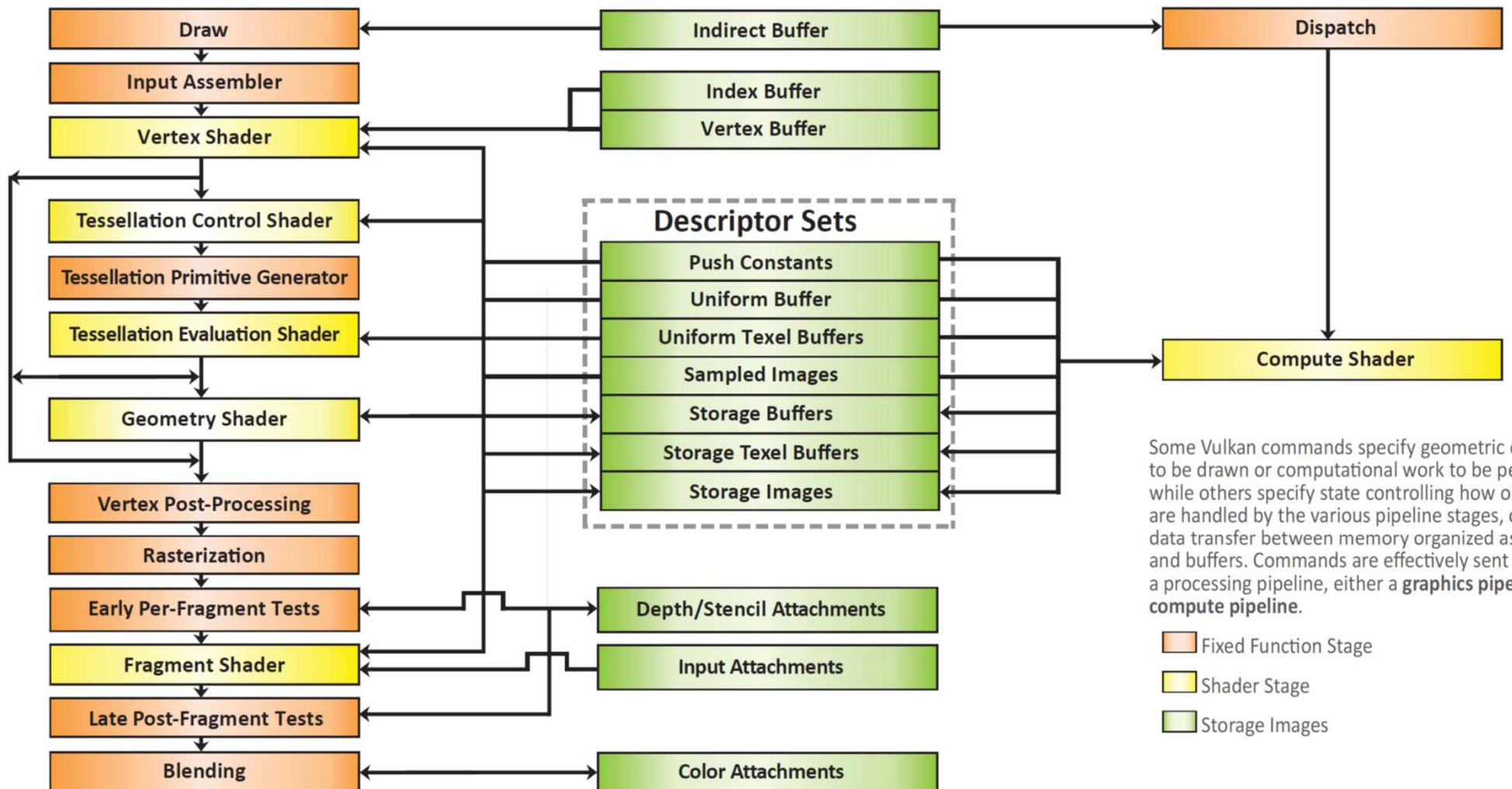


Mesh shader pipeline



See talk by Shawn Hargreaves: <https://www.youtube.com/watch?v=CFXKTXTi134>

Vulkan (1.3)



Some Vulkan commands specify geometric objects to be drawn or computational work to be performed, while others specify state controlling how objects are handled by the various pipeline stages, or control data transfer between memory organized as images and buffers. Commands are effectively sent through a processing pipeline, either a **graphics pipeline** or a **compute pipeline**.

- Fixed Function Stage
- Shader Stage
- Storage Images

Vulkan (1.3)



- Mesh and task shaders: new high-performance geometry pipeline based on compute shaders (Mesh and task shaders also available as OpenGL 4.5/4.6 extension: GL_NV_mesh_shader)

TRADITIONAL PIPELINE



TASK/MESH PIPELINE



vulkan.org

github.com/KhronosGroup/Vulkan-Guide

<https://www.khronos.org/blog/mesh-shading-for-vulkan>

Thank you.