Evolution of Graphics Pipelines

1. Understanding the Graphics Heritage
   - the era of fixed-function graphics pipelines
   - the stages to render triangles
2. Programmable Real-Time Graphics
   - programmable vertex and fragment processors
   - an example of a programmable pipeline
   - unified graphics and computing processors
   - GPU computing

MCS 572 Lecture 28
Introduction to Supercomputing
Jan Verschelde, 26 October 2016

Evolution of Graphics Pipelines

1. Understanding the Graphics Heritage
   - the era of fixed-function graphics pipelines
   - the stages to render triangles
2. Programmable Real-Time Graphics
   - programmable vertex and fragment processors
   - an example of a programmable pipeline
   - unified graphics and computing processors
   - GPU computing

evolution of graphics pipelines

Graphics processing units (GPUs) are massively parallel numeric computing processors, programmed in C with extensions.

Understanding the graphics heritage illuminates the strengths and weaknesses of GPUs with respect to major computational patterns.

The history clarifies the rationale behind major architectural design decisions of modern programmable GPUs:

- massive multithreading,
- relatively small cache memories compared to caches of CPUs,
- bandwidth-centric memory interface design.

Insights in the history provide the context for the future evolution.

advanced graphics hardware performance

Three dimensional (3D) graphics pipeline hardware evolved from large expensive systems of the early 1980s to small workstations and then PC accelerators in the mid to late 1990s.

During this period, the performance increased:

- from 50 millions pixels to 1 billion pixels per second,
- from 100,000 vertices to 10 million vertices per second.

This advancement was driven by market demand for high quality, real time graphics in computer applications.

The architecture evolved

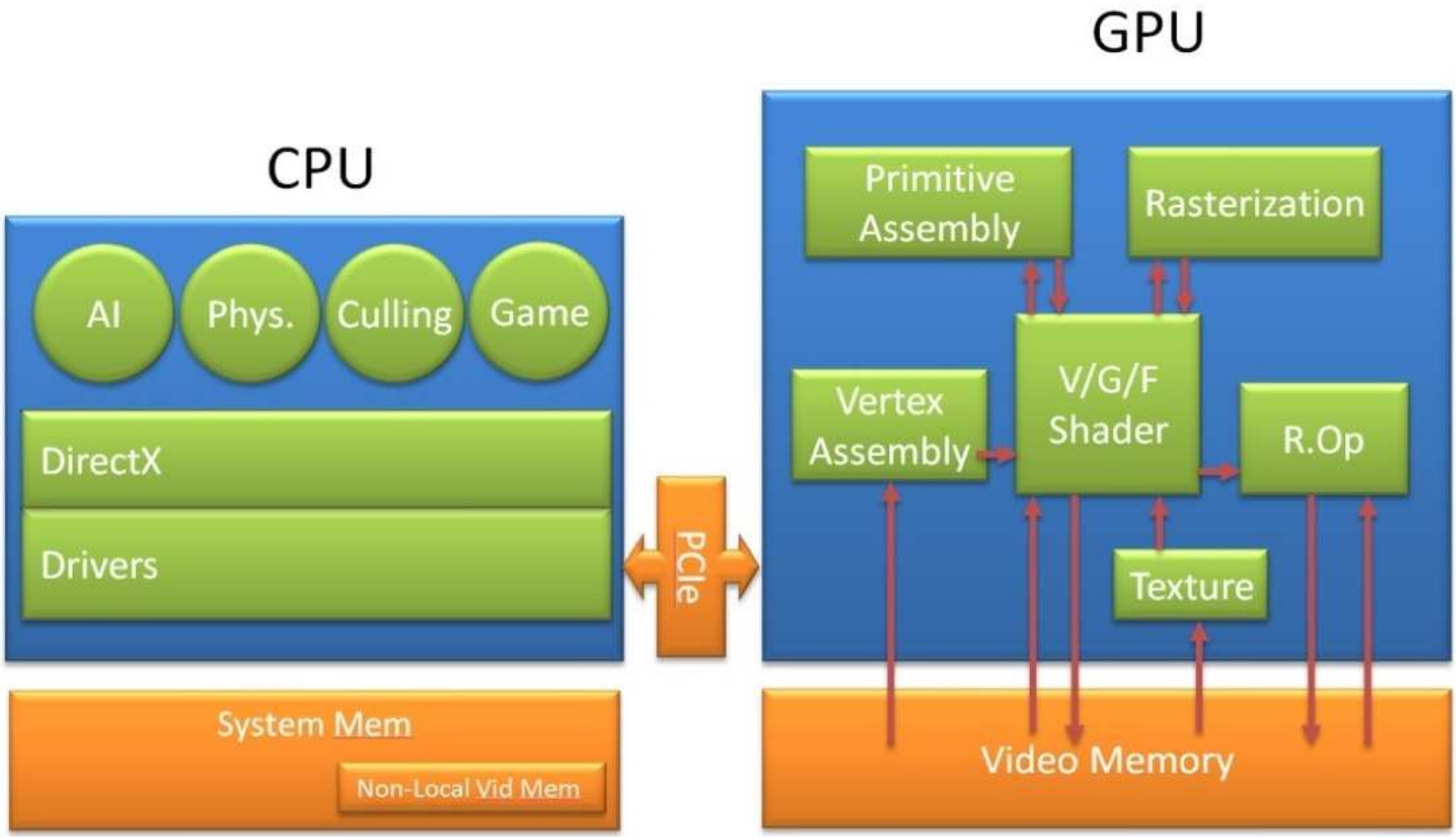
- from a simple pipeline for drawing wire frame diagrams
- to a parallel design of several deep parallel pipelines capable of rendering the complex interactive imagery of 3D scenes.

In the mean time, graphics processors became programmable.

Evolution of Graphics Pipelines

1. Understanding the Graphics Heritage
   - the era of fixed-function graphics pipelines
   - the stages to render triangles
2. Programmable Real-Time Graphics
   - programmable vertex and fragment processors
   - an example of a programmable pipeline
   - unified graphics and computing processors
   - GPU computing

displaying images



from the GeForce 8 and 9 Series GPU Programming Guide (NVIDIA)

rendering triangles

The surface of an object is drawn as a collection of triangles.

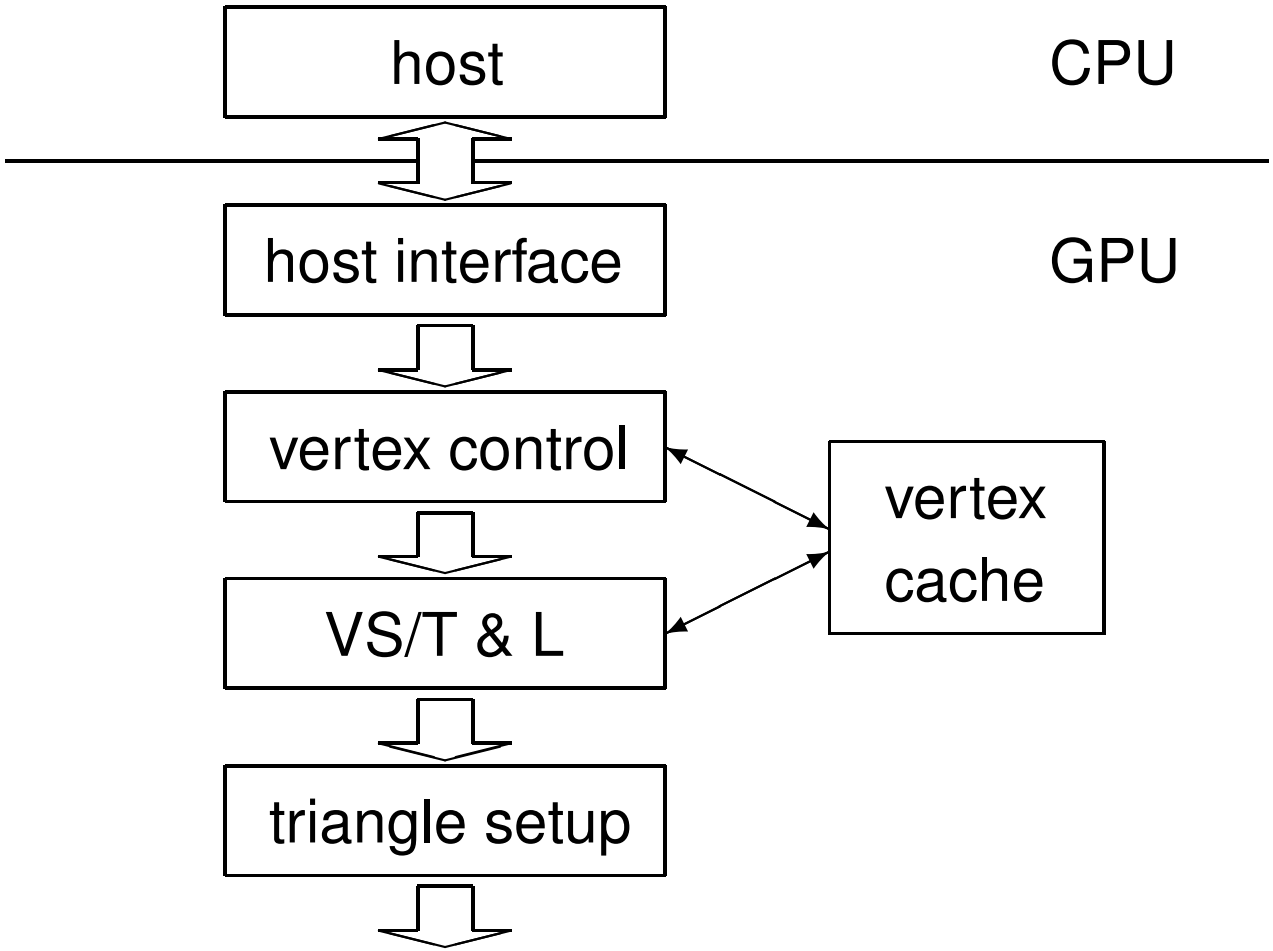
The Application Programming Interface (API) is a standardized layer of software that allows an application (e.g.: a game) to send commands to a graphics processing unit to draw objects on a display.

Examples of such APIs are DirectX and OpenGL.

The host interface (the interface to the GPU)

- receives graphics commands and data from the CPU,
- communicates back the status and result data of the execution.

a fixed-function pipeline – part one



VS/T & L = vertex shading, transform, and lighting

stages in the first part of the pipeline

1. vertex control

   This stage receives parametrized triangle data from the CPU.
   The data gets converted and placed into the vertex cache.

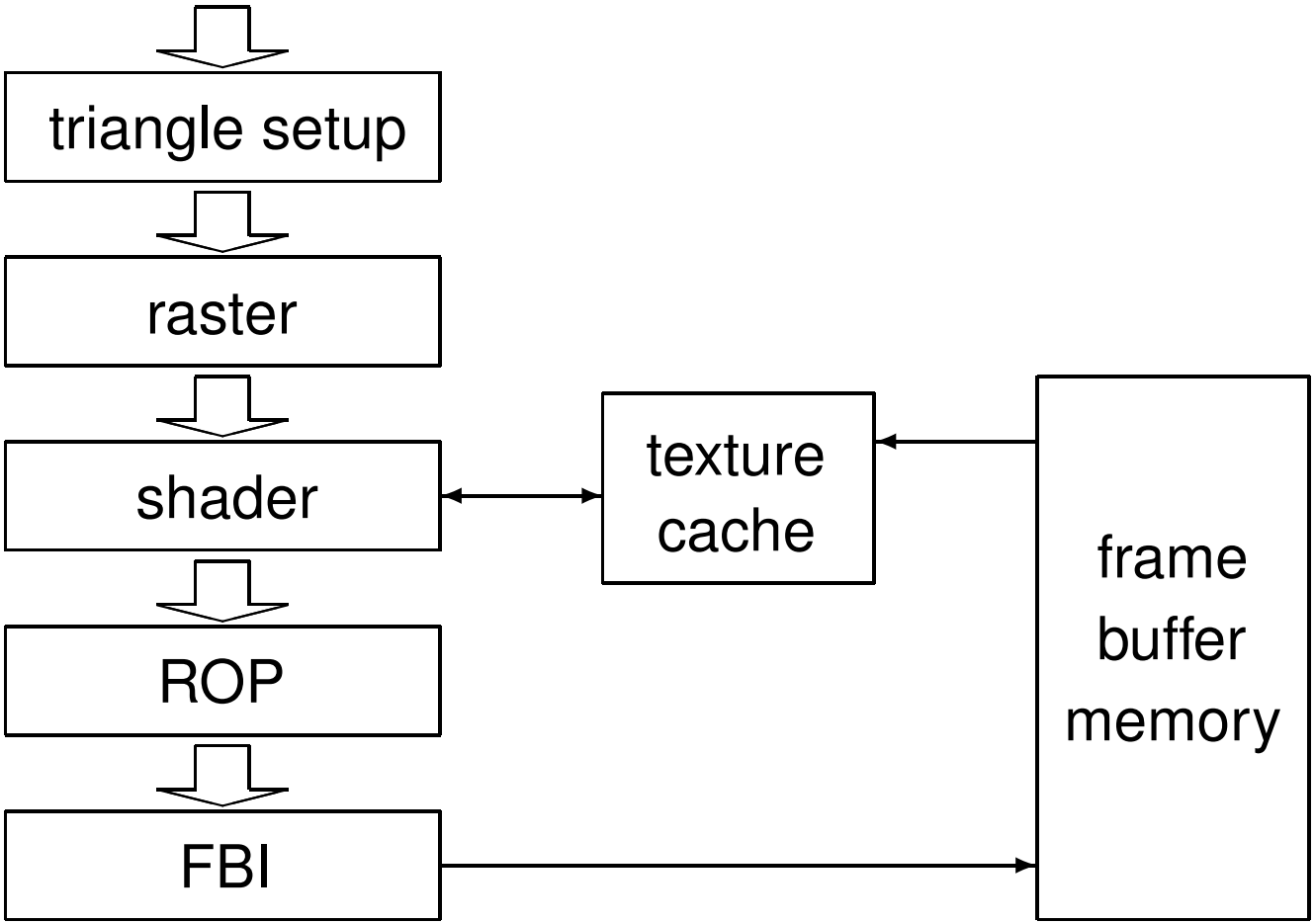
2. VS/T & L (vertex shading, transform, and lighting)

   The VS/T & L stage transforms vertices and assigns per-vertex values, e.g.: colors, normals, texture coordinates, tangents.
   The vertex shader can assign a color to each vertex,
   but color is not applied to triangle pixels until later.

3. triangle setup

   Edge equations are used to interpolate colors and other per-vertex data across the pixels touched by the triangle.

a fixed-function pipeline – part two



ROP = Raster Operation, FBI = Frame Buffer Interface

the stages in the second part of the pipeline

4. raster

   The raster determines which pixels are contained in each triangle.
   Per-vertex values necessary for shading are interpolated.

5. shader

   The shader determines the final color of each pixel as a combined effect of interpolation of vertex colors, texture mapping, per-pixel lighting, reflections, etc.

6. ROP (Raster Operation)

   The final raster operations blend the color of overlapping/adjacent objects for transparency and antialiasing effects.
   For a given viewpoint, visible objects are determined and occluded pixels (blocked from view by other objects) are discarded.

finally, the Frame Buffer Interface

7. FBI (Frame Buffer Interface)

   The FBI stages manages memory reads from and writes to the display frame buffer memory.

For high-resolution displays, there is a very high bandwidth requirement in accessing the frame buffer.

High bandwidth is achieved by two strategies:

1. using special memory designs,
2. managing simultaneously multiple memory channels that connect to multiple memory banks.

Evolution of Graphics Pipelines

1. Understanding the Graphics Heritage
   - the era of fixed-function graphics pipelines
   - the stages to render triangles

2. Programmable Real-Time Graphics
   - programmable vertex and fragment processors
   - an example of a programmable pipeline
   - unified graphics and computing processors
   - GPU computing

data independence

Stages in graphics pipelines do many floating-point operations on completely independent data, e.g.:

- transforming the positions of triangle vertices,
- generating pixel colors.

This *data independence* as the dominating characteristic is the key difference between the design assumption for GPUs and CPUs.

A single frame, rendered in $1/60^{\text{th}}$ of a second,
might have a million triangles and 6 million pixels.

the vertex shader

Vertex shader programs map the positions of triangle vertices onto the screen, altering their position, color, or orientation.

A vertex shader thread reads a vertex position (x, y, z, w) and computes its position on screen.

Geometry shader programs operate on primitives defined by multiple vertices, changing them or generating additional primitives.

Vertex shader programs and geometry shader programs execute on the vertex shader (VS/T & L) stage of the graphics pipeline.

shader programs

A shader program calculates the floating-point red, green, blue, alpha (RGBA) color contribution to the rendered image at its pixel sample image position.

The programmable vertex processor

- executes programs designated to the vertex shader stage.

The programmable fragment processor

- executes programs designated to the (pixel) shader stage.

For all graphics shader programs, instances can be run in parallel, because each works on independent data, produces independent results, and has no side effects.

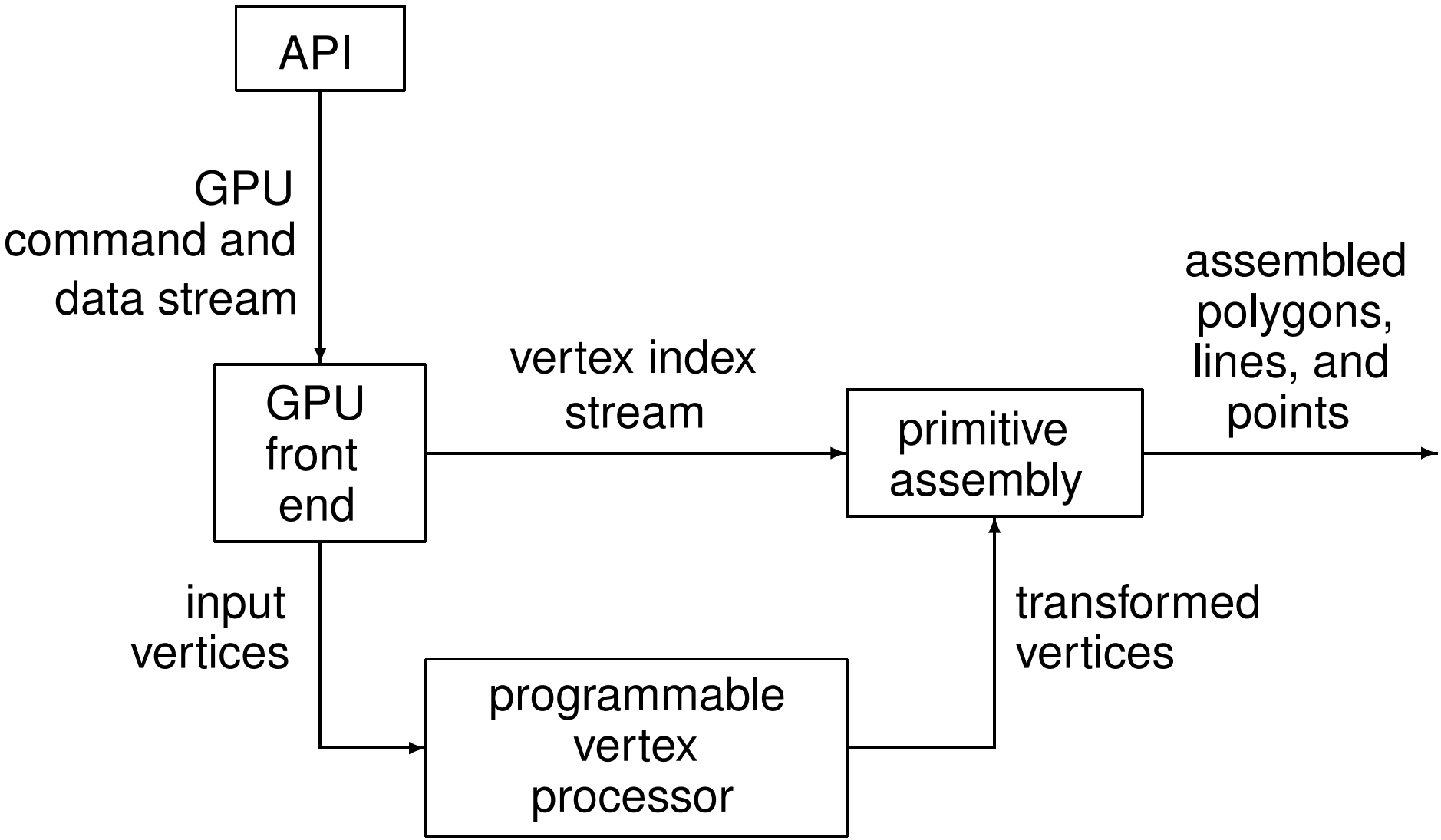
This property has motivated the design of the programmable pipeline stages into massively parallel processors.

Evolution of Graphics Pipelines

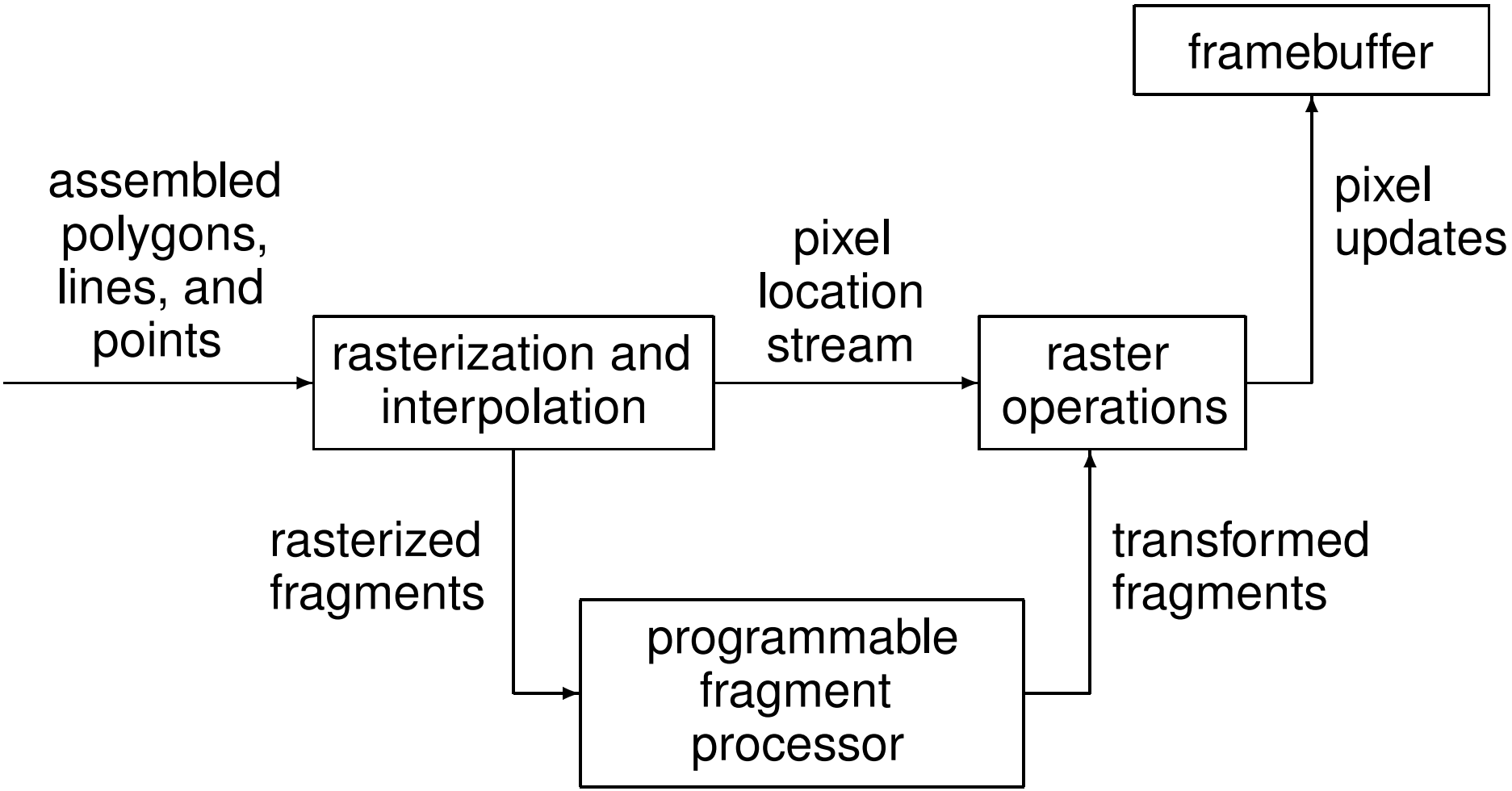
1. Understanding the Graphics Heritage
   - the era of fixed-function graphics pipelines
   - the stages to render triangles

2. Programmable Real-Time Graphics
   - programmable vertex and fragment processors
   - an example of a programmable pipeline
   - unified graphics and computing processors
   - GPU computing

vertex processor in a pipeline



fragment processor in a pipeline



fixed-function tasks and programmable processors

Between the programmable graphics pipeline stages are dozens of fixed-function stages that perform well-defined tasks far more efficiently than a programmable processor could and which would benefit far less from programmability.

For example, between the vertex processing stage and the pixel (fragment) processing stage is a *rasterizer*.
The rasterizer — it does rasterization and interpolation — is a complex state machine that determines exactly which pixels (and portions thereof) lie within each geometric primitive's boundaries.

The mix of programmable and fixed-function stages is engineered to balance performance with user control over the rendering algorithm.

Evolution of Graphics Pipelines

1. Understanding the Graphics Heritage
   - the era of fixed-function graphics pipelines
   - the stages to render triangles
2. Programmable Real-Time Graphics
   - programmable vertex and fragment processors
   - an example of a programmable pipeline
   - unified graphics and computing processors
   - GPU computing

unified graphics and computing processors

Introduced in 2006, the GeForce 8800 GPU mapped the separate programmable graphics stages to an array of unified processors.

The graphics pipeline is physically a recirculating path that visits the processors three times, with much fixed-function tasks in between.

More sophisticated shading algorithms motivated a sharp increase in the available shader operation rate, in floating-point operations.

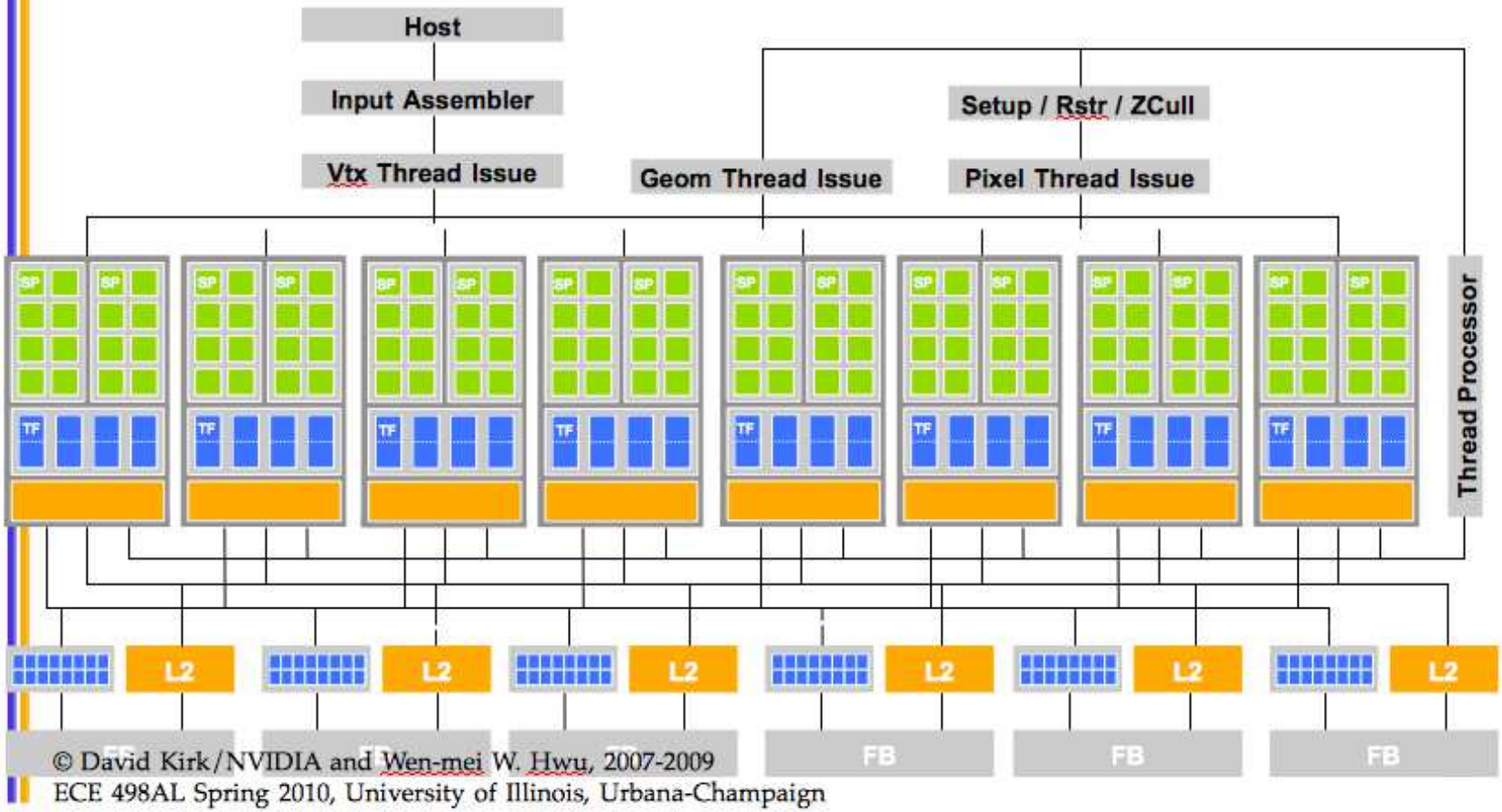
High-clock-speed design made programmable GPU processor array ready for general numeric computing.

Original GPGPU programming used APIs (DirectX or OpenGL): to a GPU everything is a pixel.

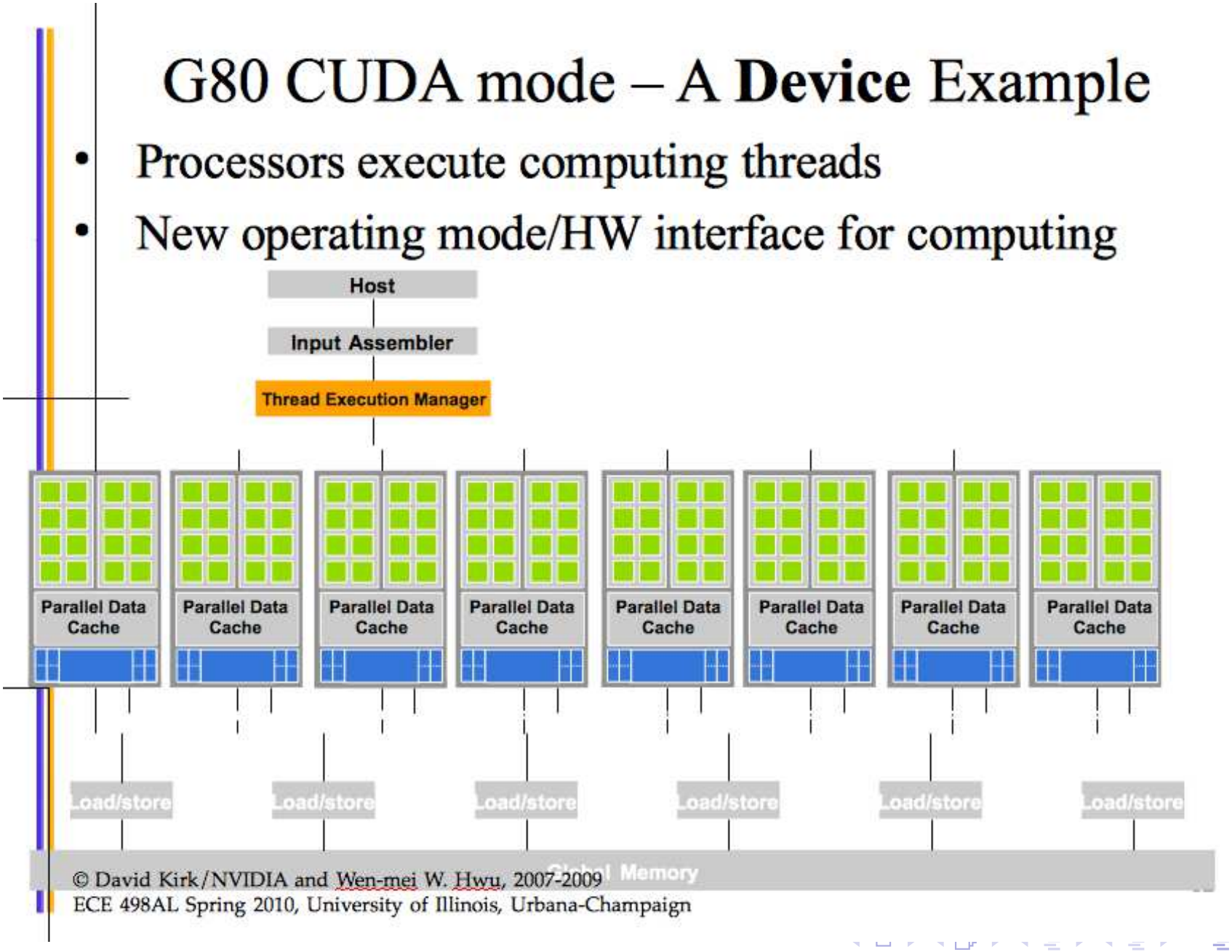
GeForce 8800 GPU for GPGPU programming

G80 – Graphics Mode

- The future of GPUs is programmable processing
- So – build the architecture around the processor



GeForce 8800 GPU with new interface



Evolution of Graphics Pipelines

1. Understanding the Graphics Heritage
   - the era of fixed-function graphics pipelines
   - the stages to render triangles
2. Programmable Real-Time Graphics
   - programmable vertex and fragment processors
   - an example of a programmable pipeline
   - unified graphics and computing processors
   - GPU computing

GPU computing

Drawbacks of the GPGPU model:

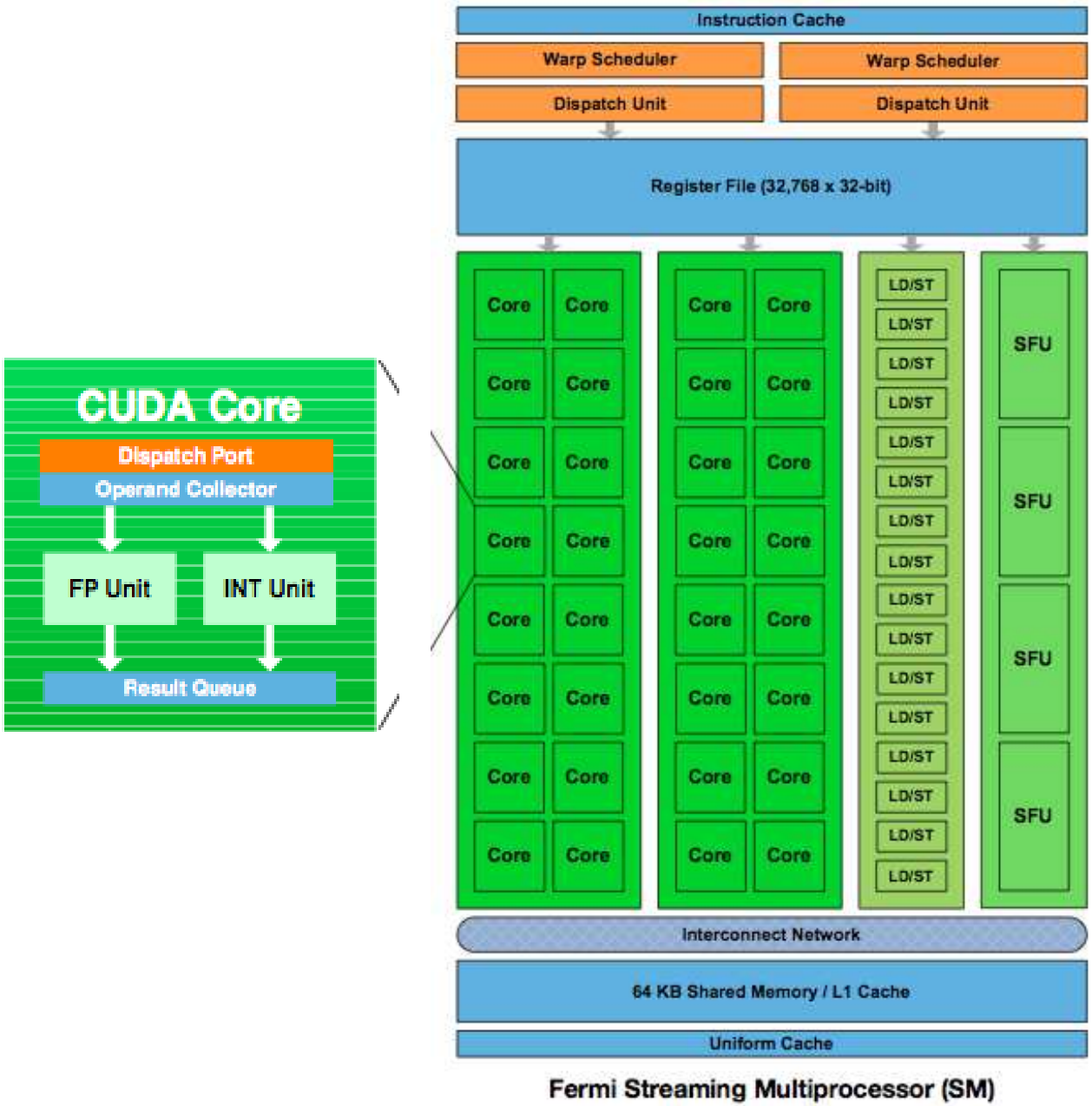
1. The programmer must know APIs and GPU architecture well.
2. Programs expressed in terms of vertex coordinates, textures, shader programs, add to the complexity.
3. Random reads and writes to memory are not supported.
4. No double precision is limiting for scientific applications.

Programming GPUs with CUDA (C extension): GPU computing.

Chapter 2 in the textbook ends mentioning the GT200.
The next generation is code-named Fermi:

- 32 CUDA cores per streaming multiprocessor,
- 8× peak double precision floating point performance over GT200,
- true cache hierarchy, more shared memory,
- faster context switching, faster atomic operations.

the 3rd generation Fermi Architecture

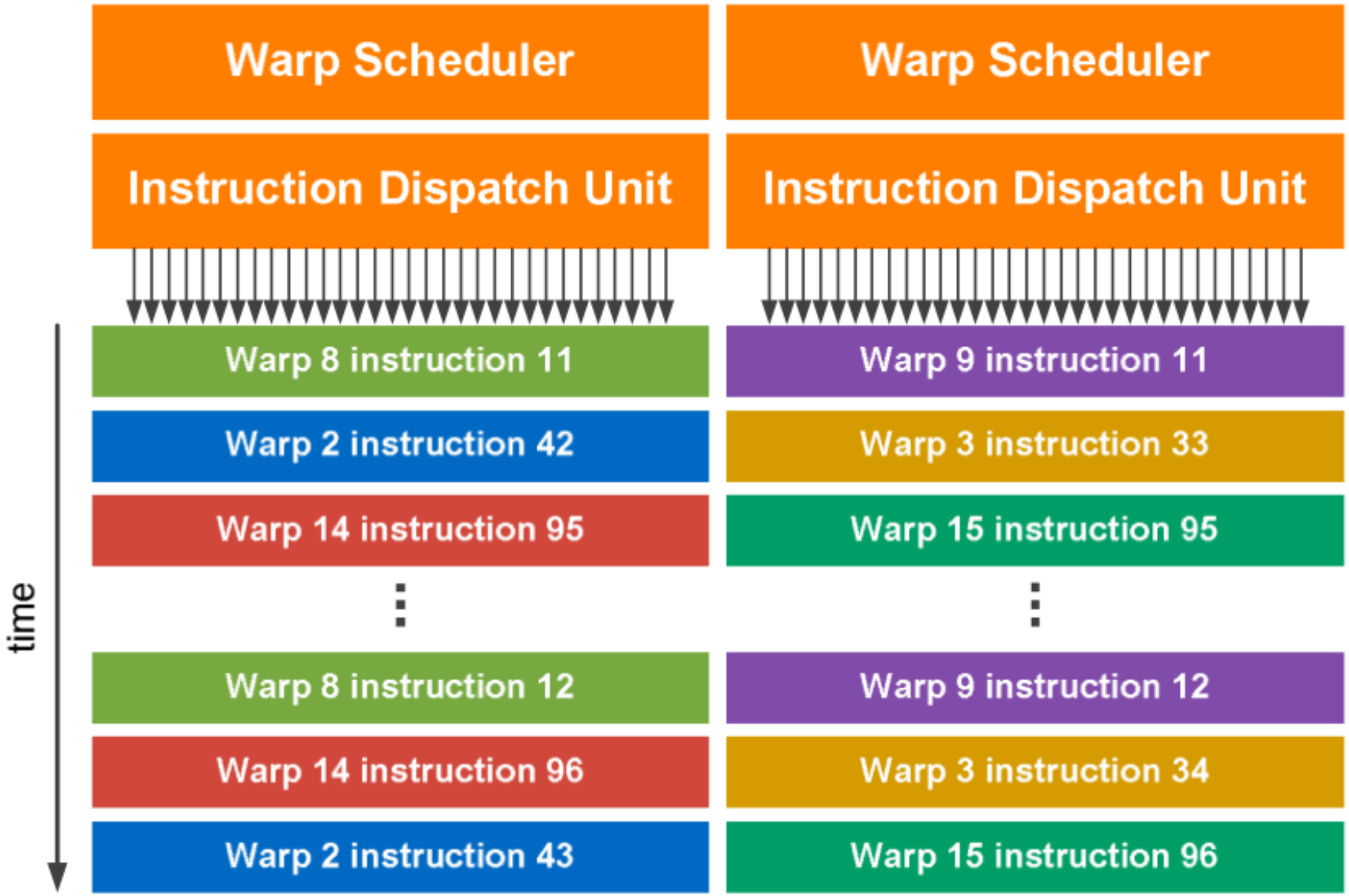


Fermi Streaming Multiprocessor (SM)

Pascal's Streaming Multiprocessor

from the NVIDIA Tesla P100 Whitepaper

dual warp scheduler



summary table

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Special Function Units (SFUs) / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

the Kepler architecture

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

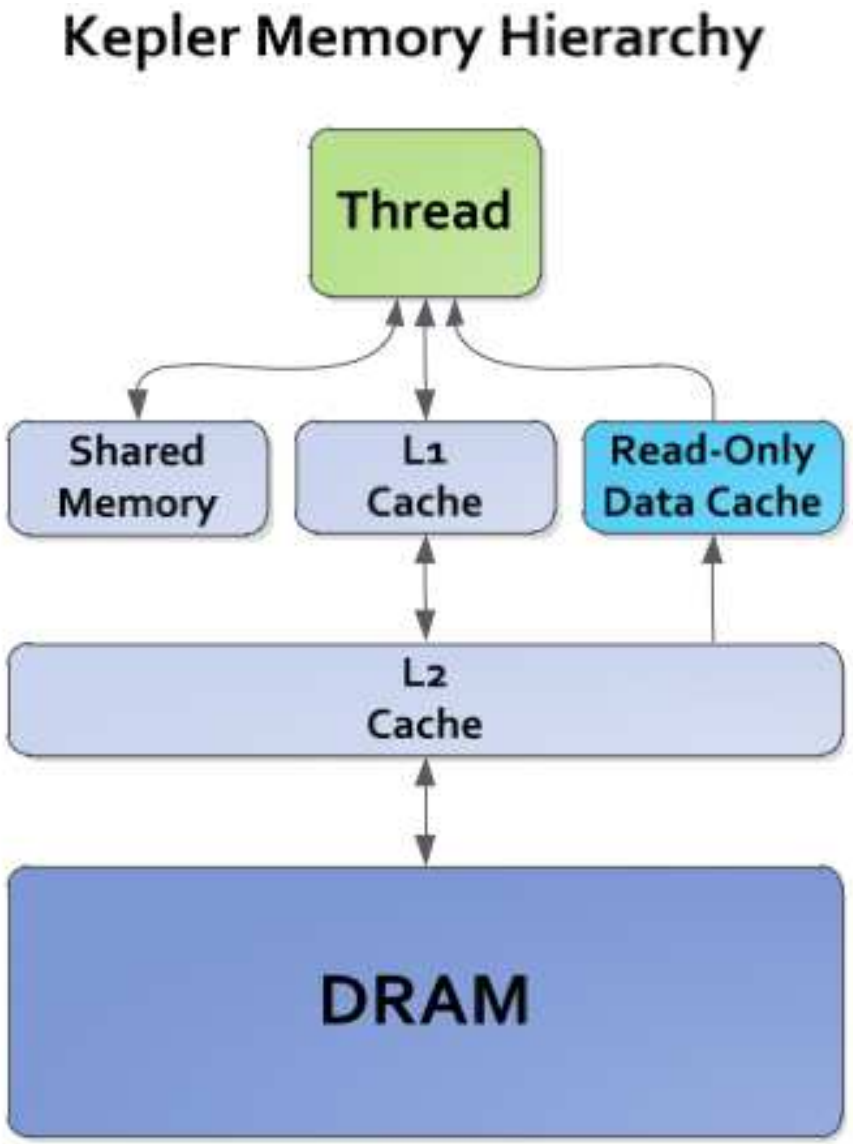
Compute Capability of Fermi and Kepler GPUs

Kepler and Pascal

GPU	Kepler GK110	Maxwell GM200	Pascal GP100
Compute Capability	3.5	5.2	6.0
Threads / Warp	32	32	32
Max Warps / Multiprocessor	64	64	64
Max Threads / Multiprocessor	2048	2048	2048
Max Thread Blocks / Multiprocessor	16	32	32
Max 32-bit Registers / SM	65536	65536	65536
Max Registers / Block	65536	32768	65536
Max Registers / Thread	255	255	255
Max Thread Block Size	1024	1024	1024
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB

from the NVIDIA Tesla P100 Whitepaper

memory hierarchies



summary and references

To fully utilize the GPU, one must use thousands of threads, whereas running thousands of threads on multicore CPU will swamp it.

Data independence is the key difference between GPU and CPU.

We covered most of chapter 2 of the textbook by Kirk & Hwu.

NVIDIA. **Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi.**
NVIDIA. **Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110.**
NVIDIA. **Whitepaper NVIDIA Tesla P100.**
Available at `http://www.nvidia.com`

Chapter 12 of the book of Wilkinson & Allen is on Image Processing.