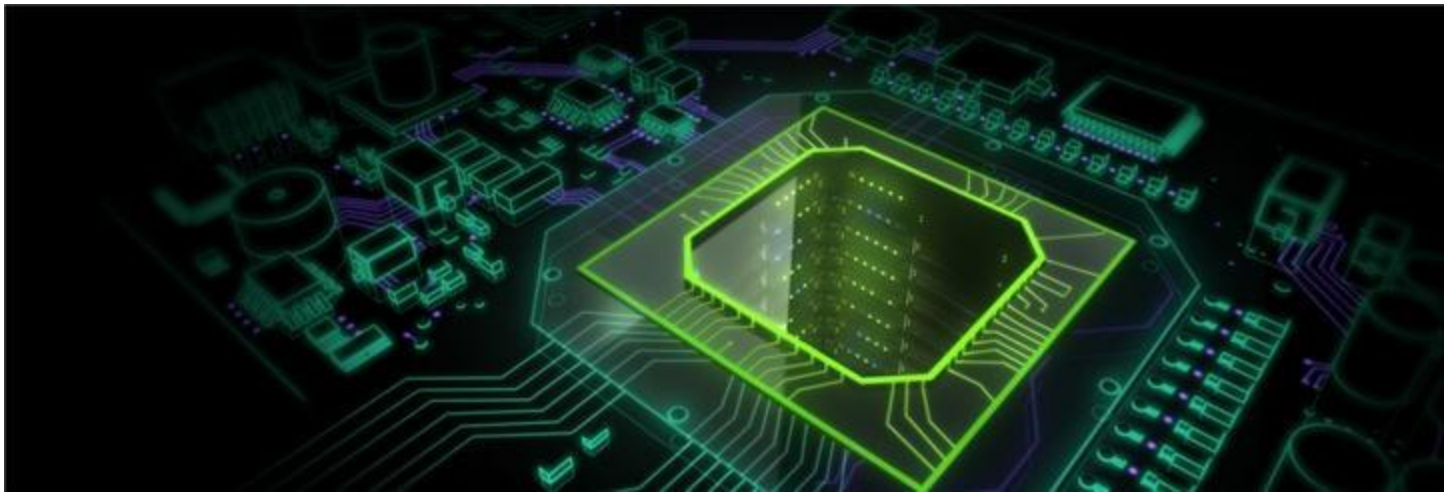
NYU
New York University

CSCI-GA.3033-004

Graphics Processing Units (GPUs): Architecture and Programming

Lecture 2: Hardware Perspective of GPUs

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com



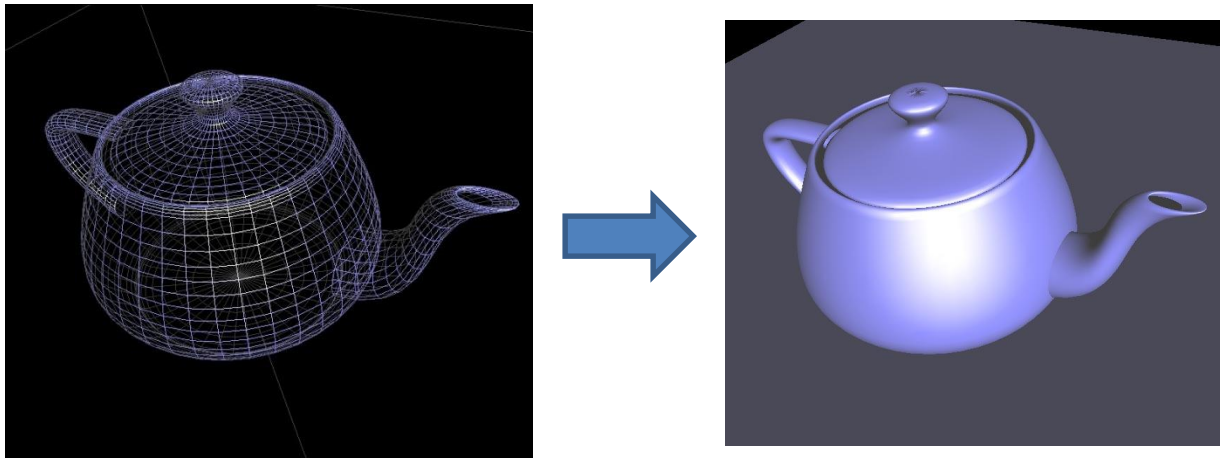
History of GPUs …
How did they evolve?

Why Looking at GPU History?

- Looking at how things evolved can highlight future directions.
- Some of the current architecture decisions won't make sense without historical perspective.

A Little Bit of Vocabulary

- Rendering: the process of generating an image from a model
- Vertex: the corner of a polygon (usually that polygon is a triangle)
- Pixel: smallest addressable screen element



From Numbers to Screen

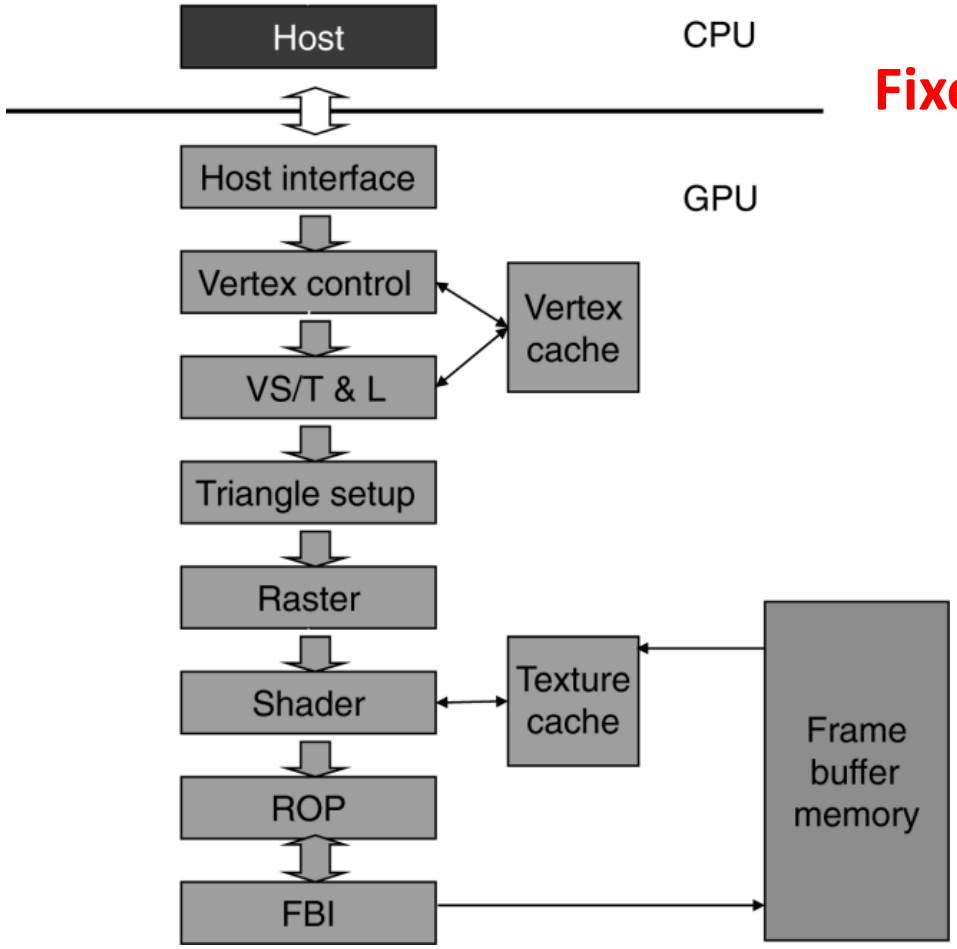
```
0.748952 -0.764952 -0.210132,
0.872245 -0.600862 -0.210132,
1.00016 -0.369596 -0.210132,
1.00939 -0.094004 -0.210132,
1.14496 0.324436 -0.210132,
1.15747 0.601712 -0.210132,
1.08016 0.793529 -0.210132,
0.98164 0.972032 -0.210132,
0.808263 0.929016 -0.210132,
0.442563 0.985585 -0.210132,
0.221794 1.00159 -0.210132,
0 1.0053 -0.210132,
-0.221794 1.00159 -0.210132,
-0.442563 0.985585 -0.210132,
-0.808263 0.929016 -0.210132,
```

Before GPUs

- Vertices to pixels:
  - Transformations done on CPU
  - Compute each pixel "by hand", in series… **slow!**

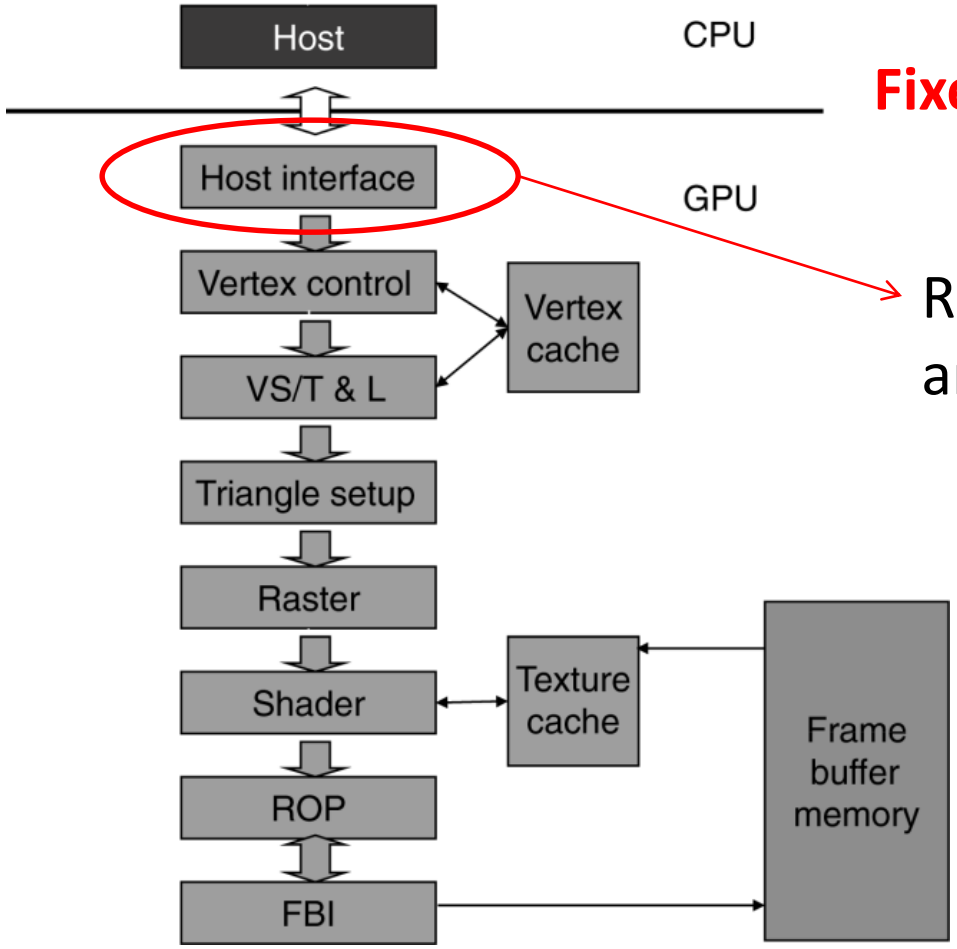
Example: 1 million triangles * 100 pixels per triangle * 10 lights * 4 cycles per light computation = **4 billion cycles**

Early GPUs: Early 80s to Late 90s



Fixed-Function Pipeline

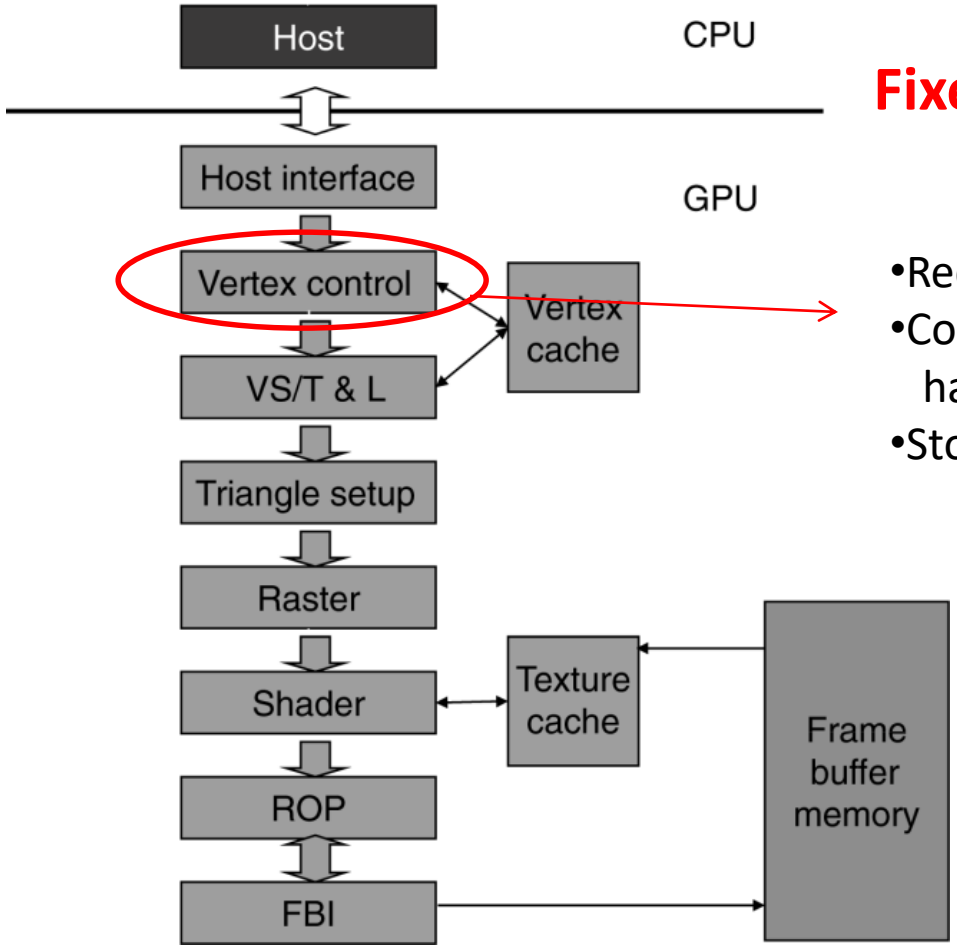
Early GPUs: Early 80s to Late 90s



Fixed-Function Pipeline

Receives graphics commands and data from CPU

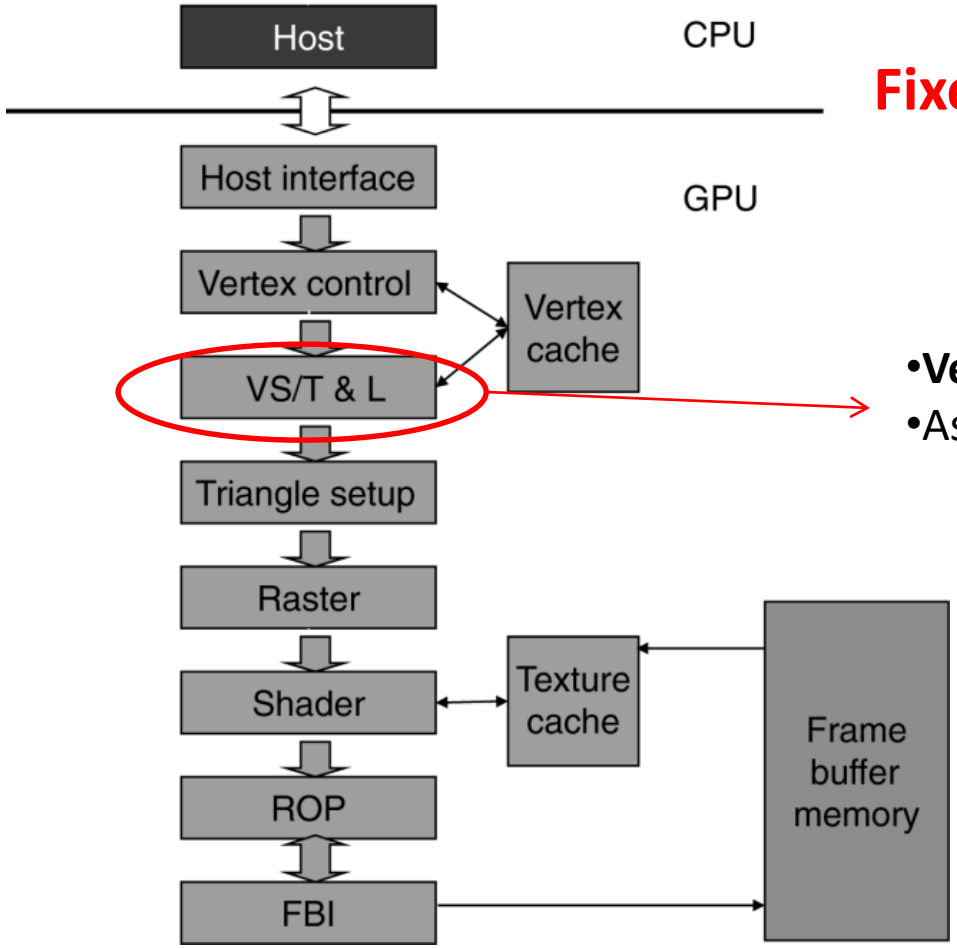
Early GPUs: Early 80s to Late 90s



Fixed-Function Pipeline

- Receives triangle data
- Converts them into a form that hardware understands
- Store the prepared data in vertex cache

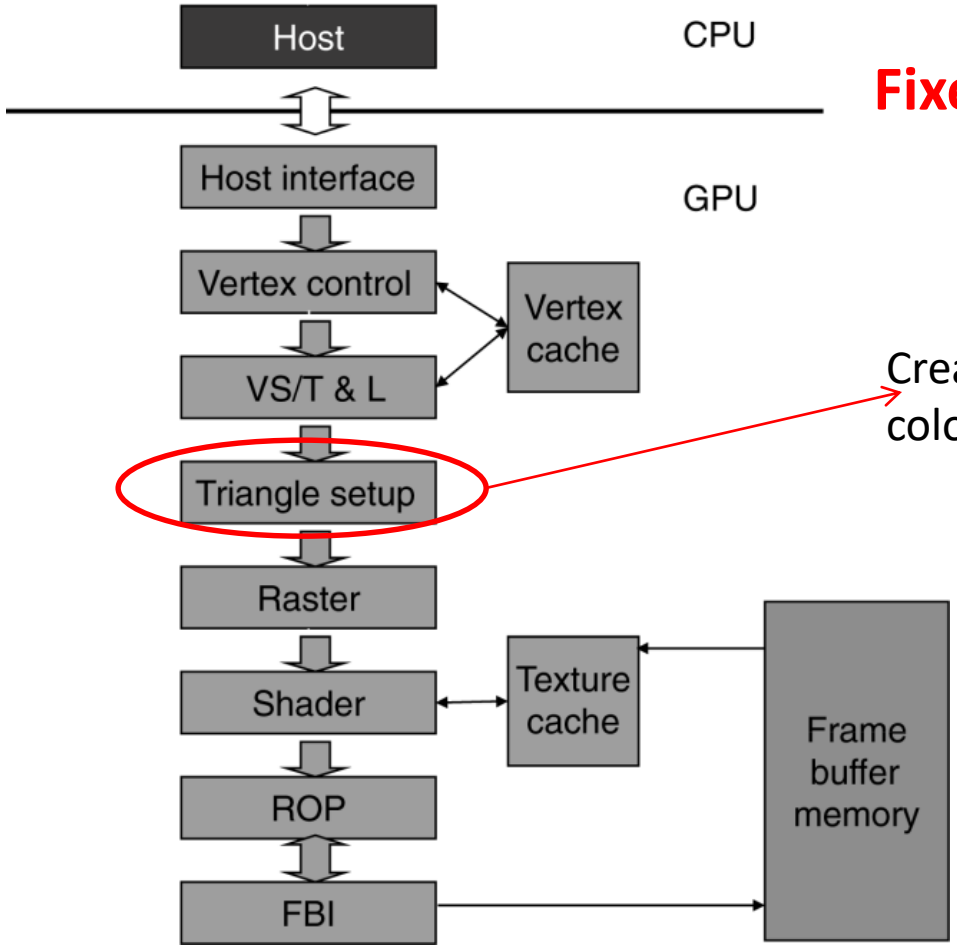
Early GPUs: Early 80s to Late 90s



Fixed-Function Pipeline

- **Vertex shading transform and lighting**
- Assigns per-vertex value (colors, …).

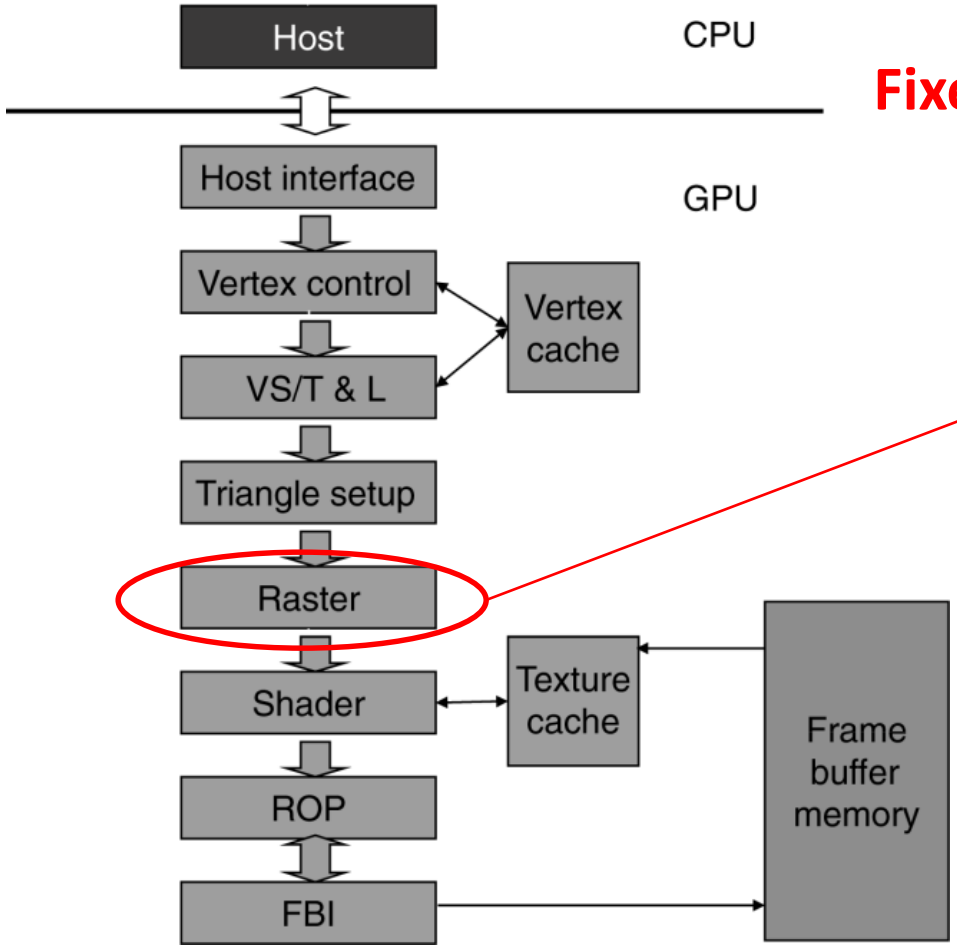
Early GPUs: Early 80s to Late 90s



Fixed-Function Pipeline

Creates edge equations to interpolate colors across pixels touched by the triangle

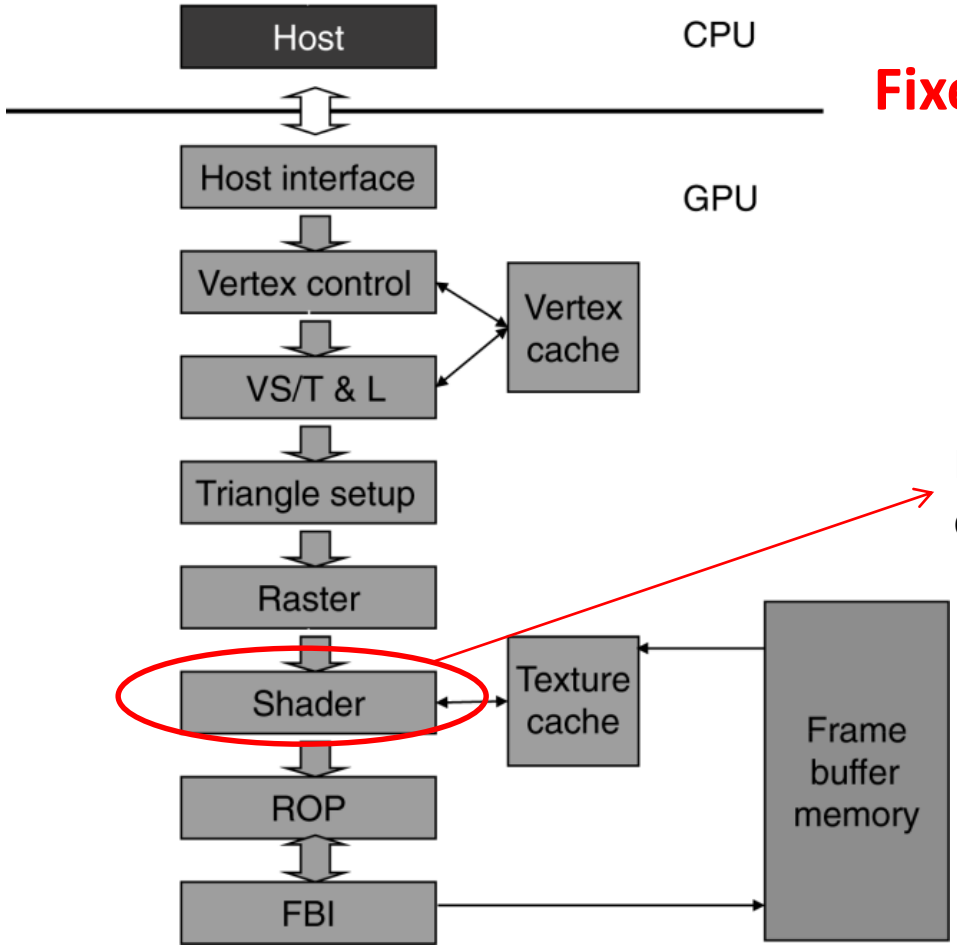
Early GPUs: Early 80s to Late 90s



Fixed-Function Pipeline

- Determines which pixel falls into which triangle
- For each pixel, interpolate per-pixel values from vertices

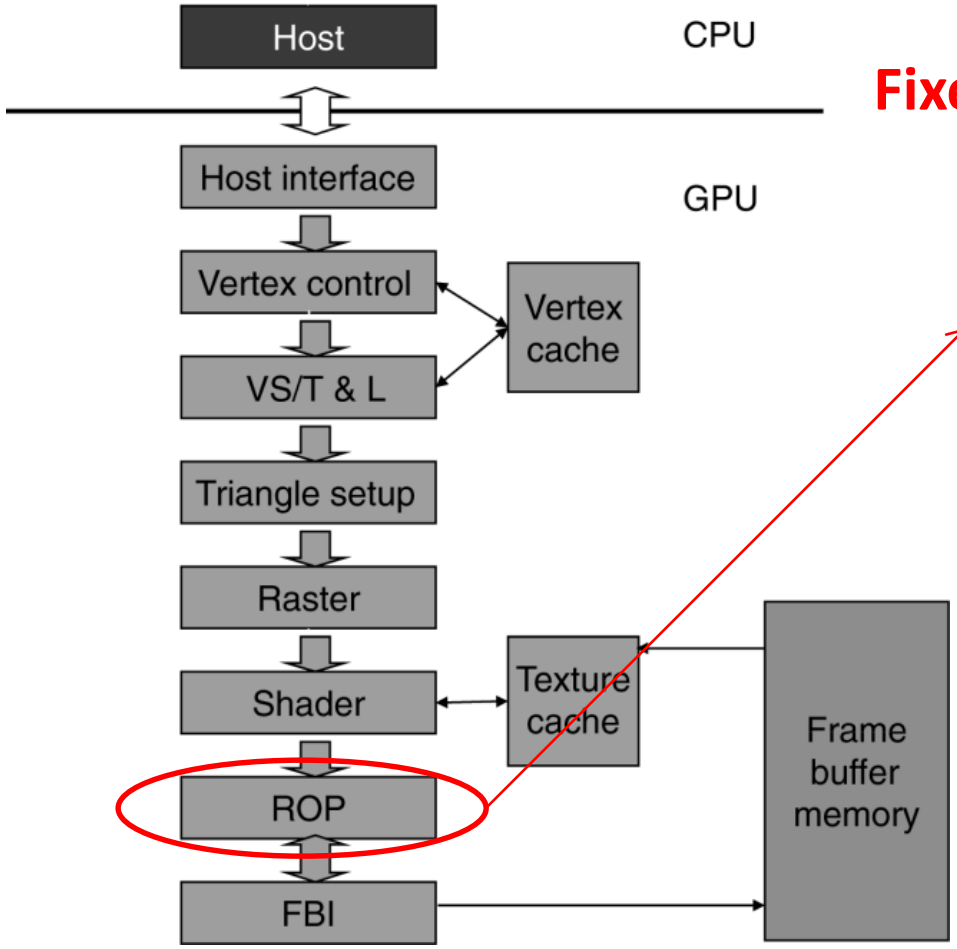
Early GPUs: Early 80s to Late 90s



Fixed-Function Pipeline

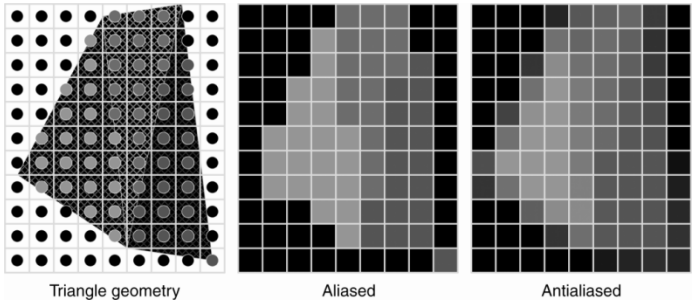
Determines the final color of each pixel

Early GPUs: Early 80s to Late 90s

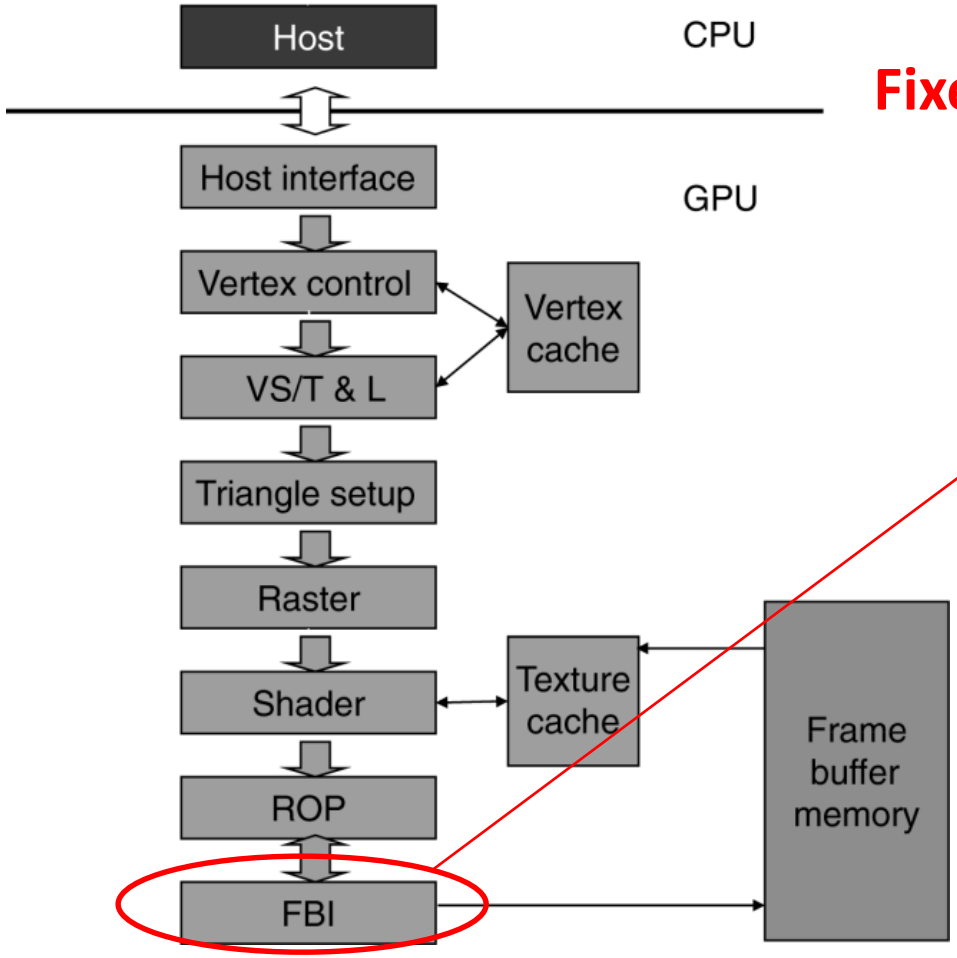


Fixed-Function Pipeline

The **raster operation**: performs color raster operations that blend the color of overlapping objects for transparency and antialiasing



Early GPUs: Early 80s to Late 90s

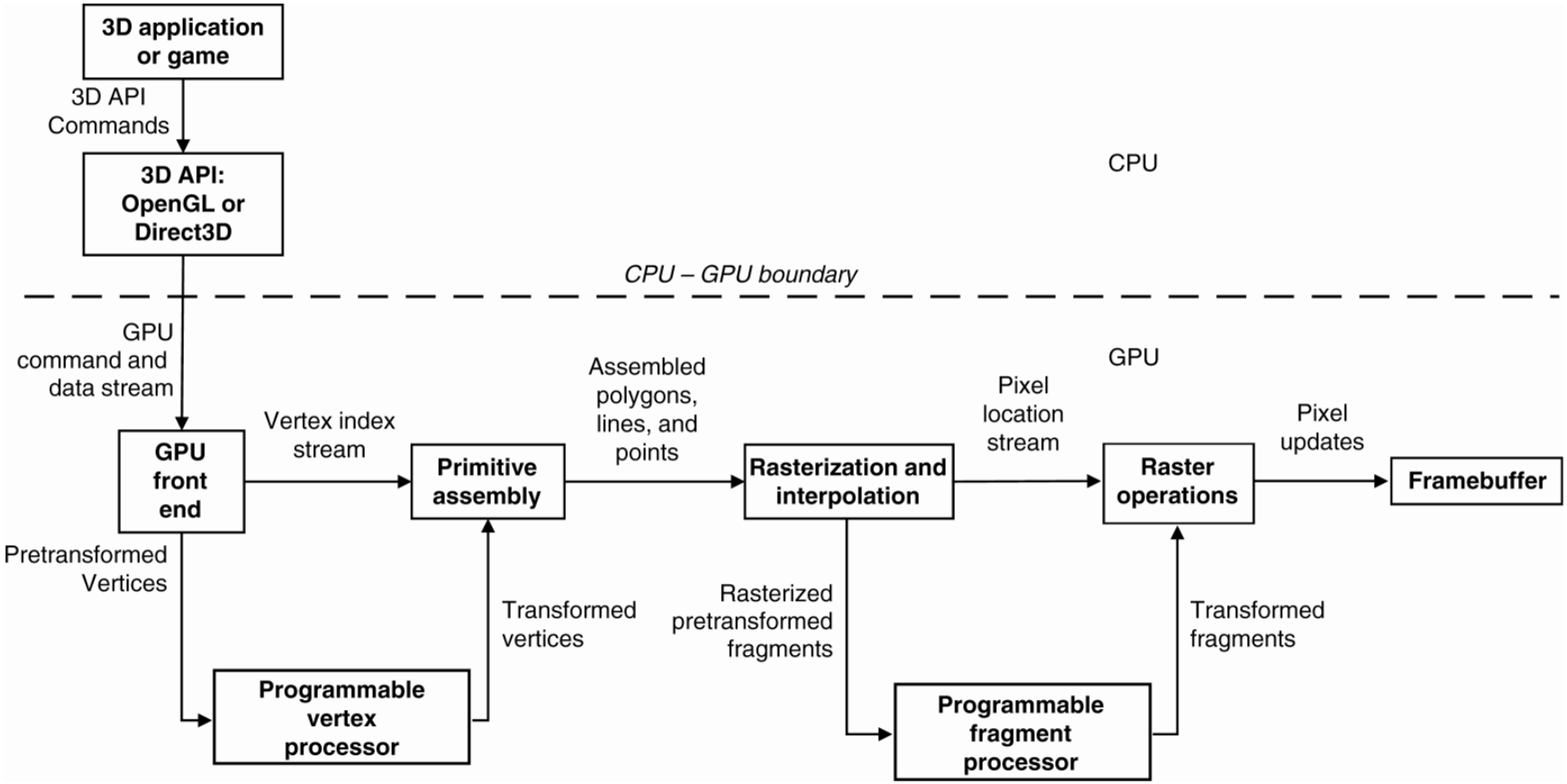


Fixed-Function Pipeline

The **frame buffer interface** manages memory reads/writes.

Next Steps

- In 2001:
  - NVIDIA exposed the application developer to the instruction set of VS/T&L stage
- Later:
  - General programmability extended to to shader stage → trend toward unifying the functionality of the different stages as seen by the application programmer.
  - In graphics pipelines, certain stages do a great deal of floating-points arithmetic on a completely independent data.
    - Data independence is exploited → key assumption in GPUs



Fragment = a technical term usually meaning a single pixel

In 2006

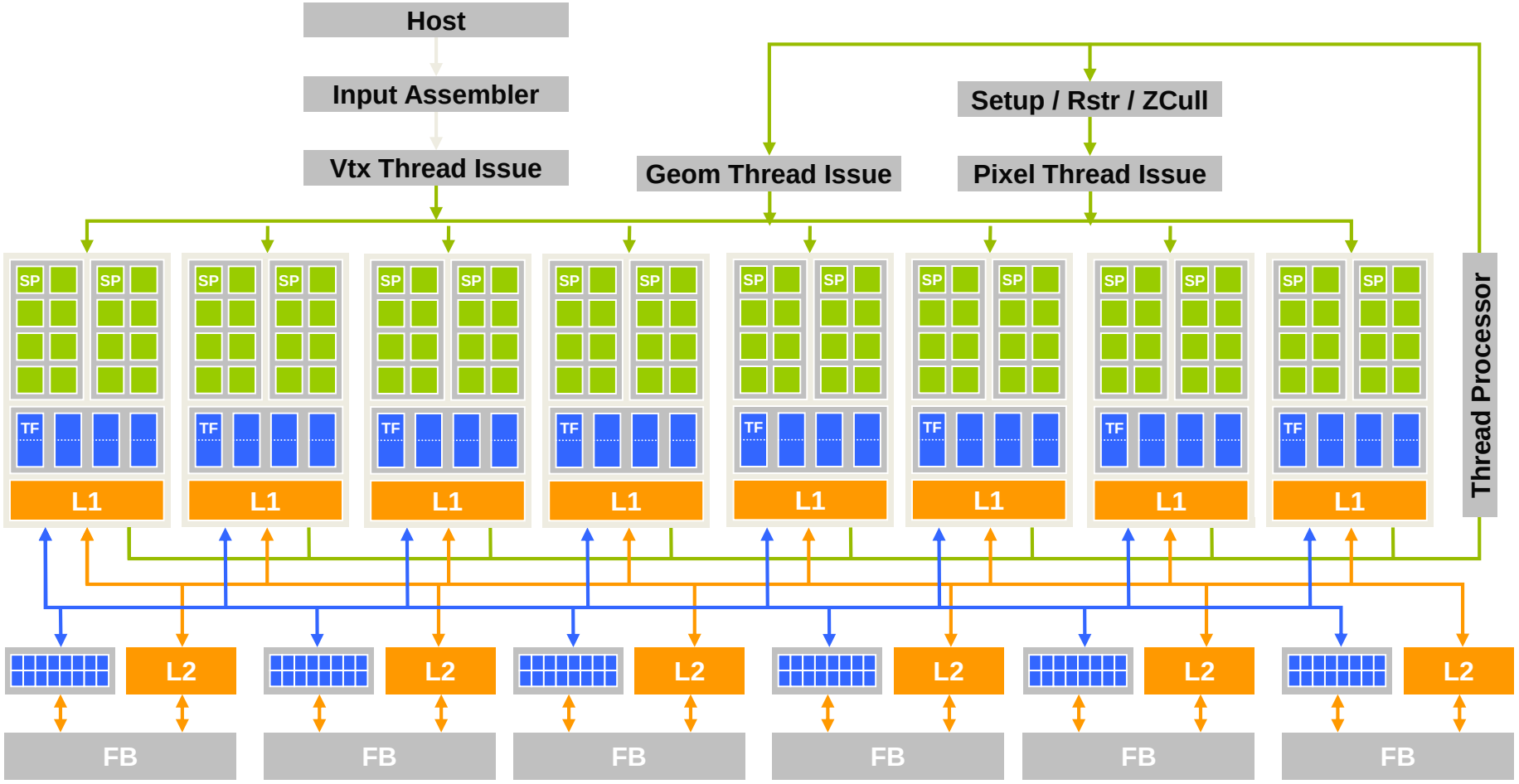
- NVIDIA GeForce 8800 mapped separate graphics stage to a unified array of processors
  - For vertex shading, geometry processing, and pixel processing
  - Allows dynamic partition

Regularity + Massive Parallelism







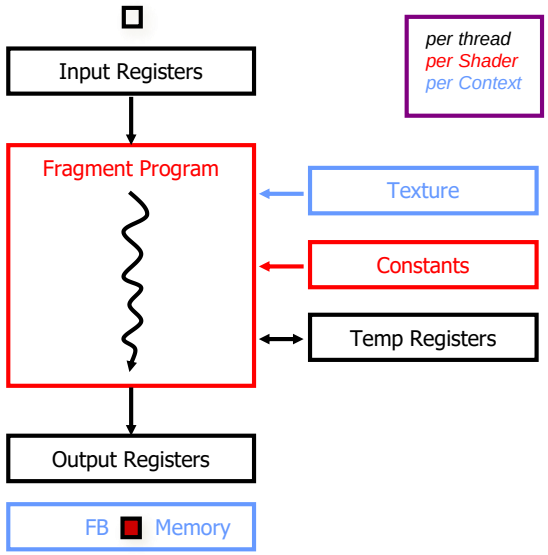
Exploring the use of GPUs to solve compute intensive problems

The birth of GPGPU but there are many constraints

GPUs and associated APIs were designed to process graphics data

Previous GPGPU Constraints

- Dealing with graphics API
  - Working with the corner cases of the graphics API
- Addressing modes
  - Limited texture size/dimension
- Shader capabilities
  - Limited outputs
- Instruction sets
  - Lack of Integer & bit ops
- Communication limited
- No user-defined data types



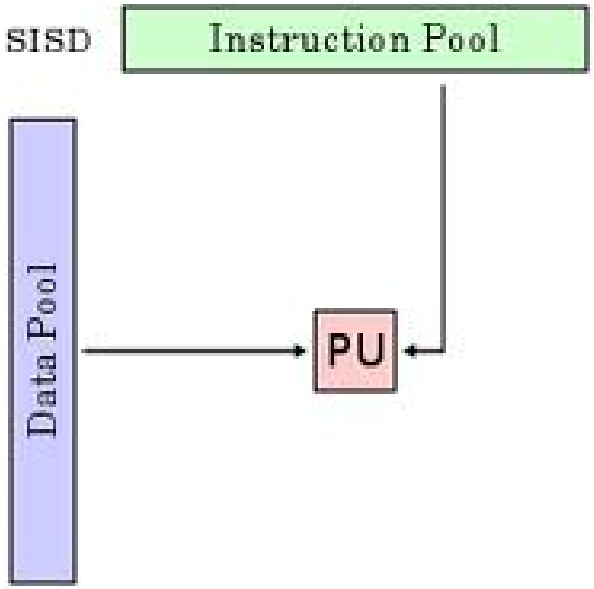
The Birth of GPU Computing

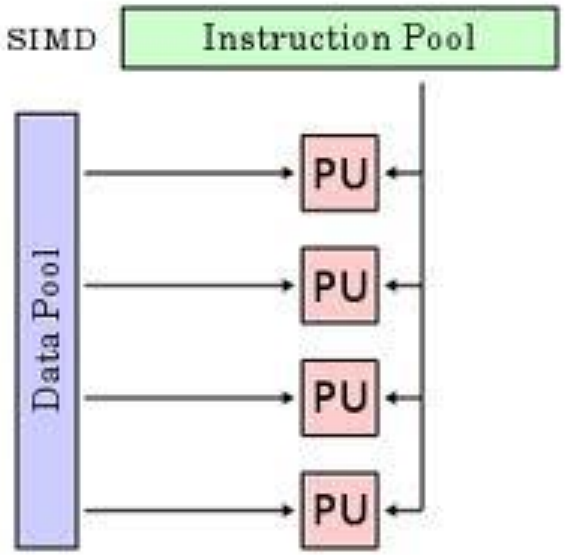
- **Step 1**: Designing high-efficiency floating-point and integer processors.
- **Step 2**: Exploiting data parallelism by having large number of processors
- **Step 3**: Shader processors fully programmable with large instruction cache, instruction memory, and instruction control logic.
- **Step 4**: Reducing the cost of hardware by having multiple shader processors to share their cache and control logic.
- **Step 5**: Adding memory load/store instructions with random byte addressing capability
- **Step 6**: Developing CUDA C/C++ compiler, libraries, and runtime software models.

A Quick Glimpse on: Flynn Classification

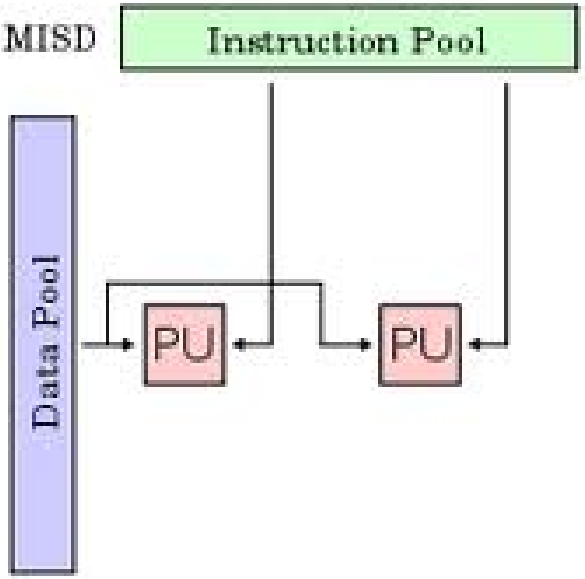
- A taxonomy of computer architecture
- Proposed by Micheal Flynn in 1966
- It is based two things:
  - Instructions
  - Data

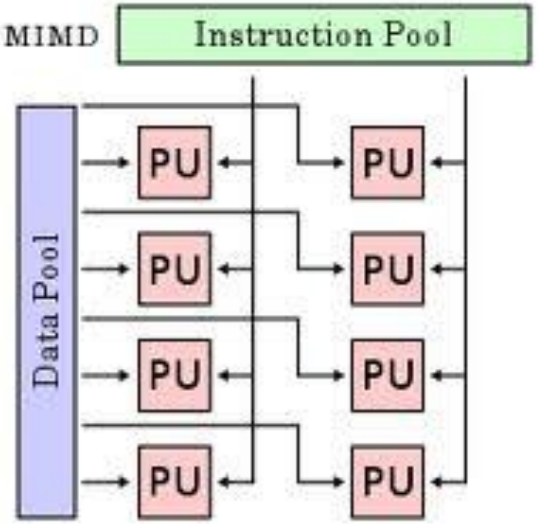
	Single instruction	Multiple instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD

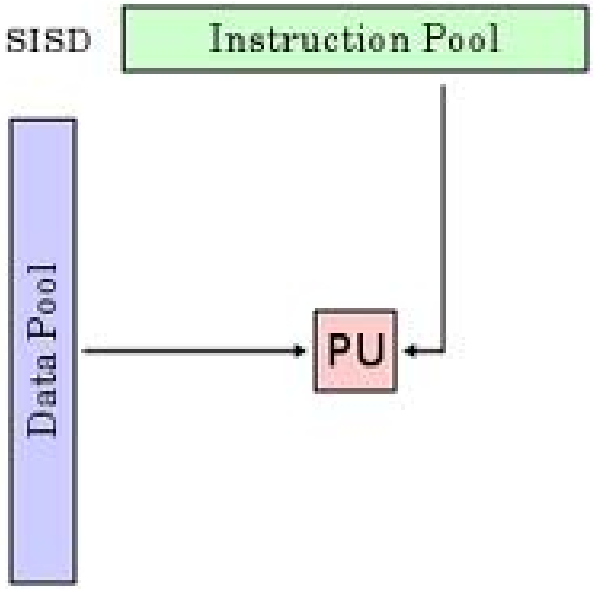


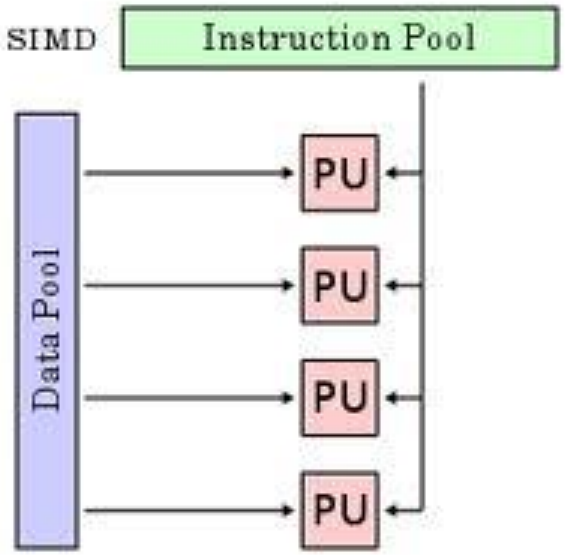


PU = Processing Unit

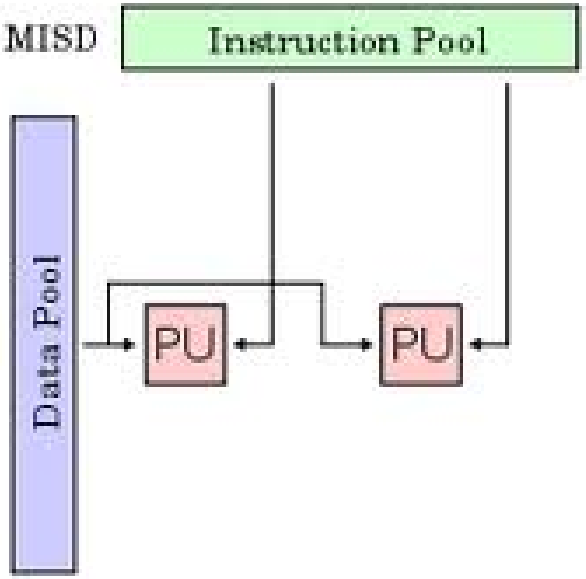


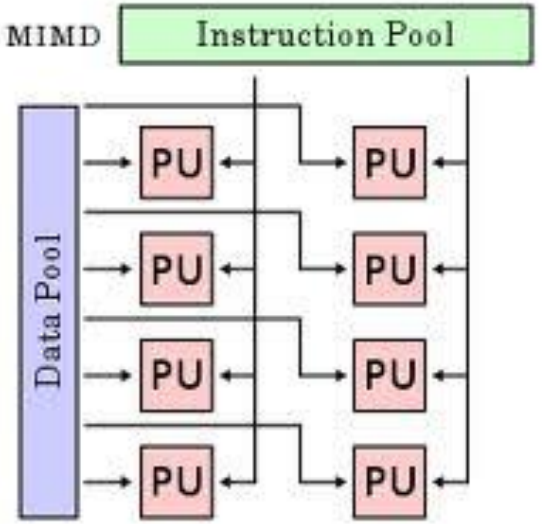




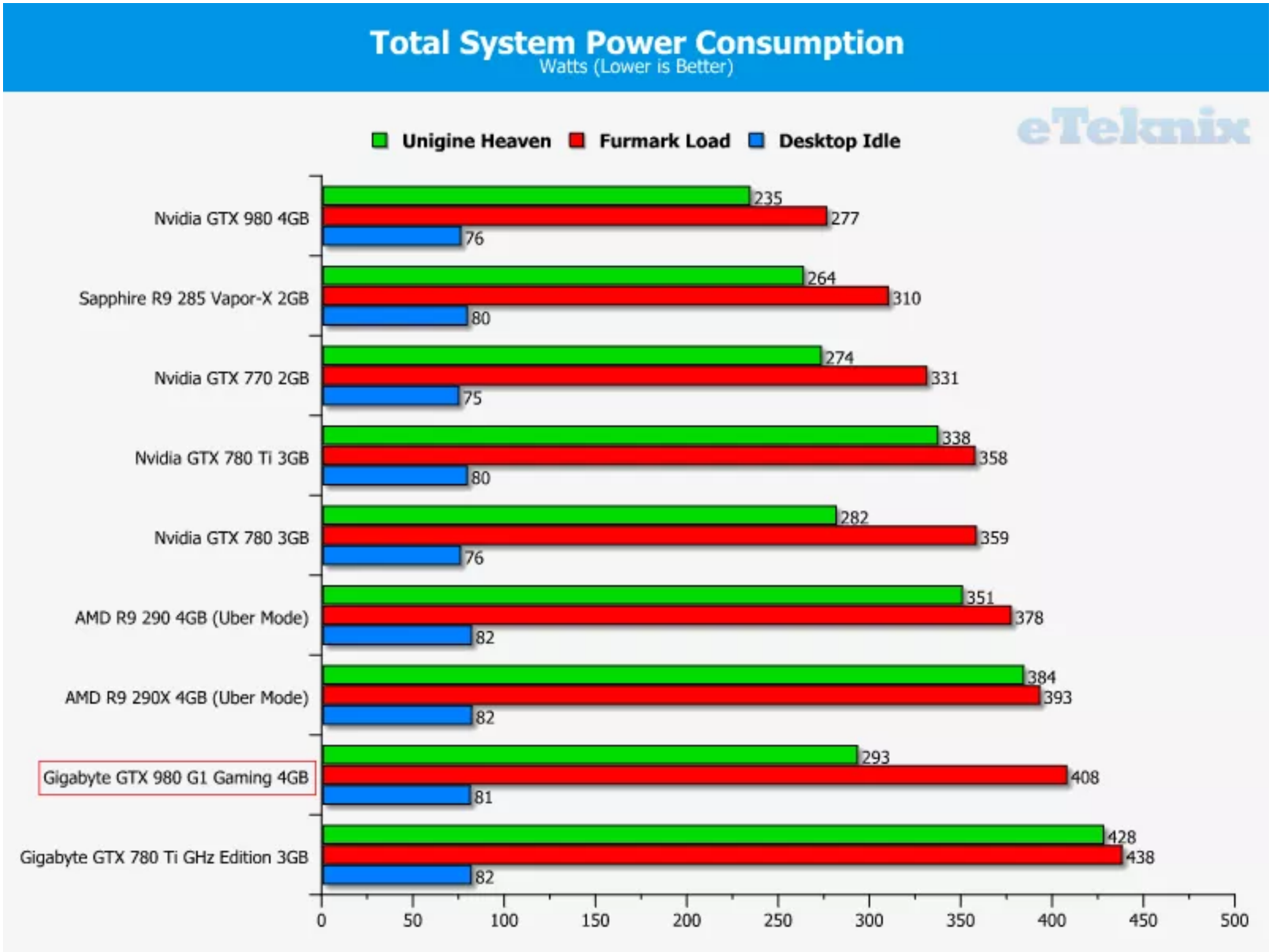


Which one is closest to GPU?





Problem With GPUs: Power



Source: http://www.eteknix.com/gigabyte-g1-gaming-geforce-gtx-980-4gb-graphics-card-review/17/

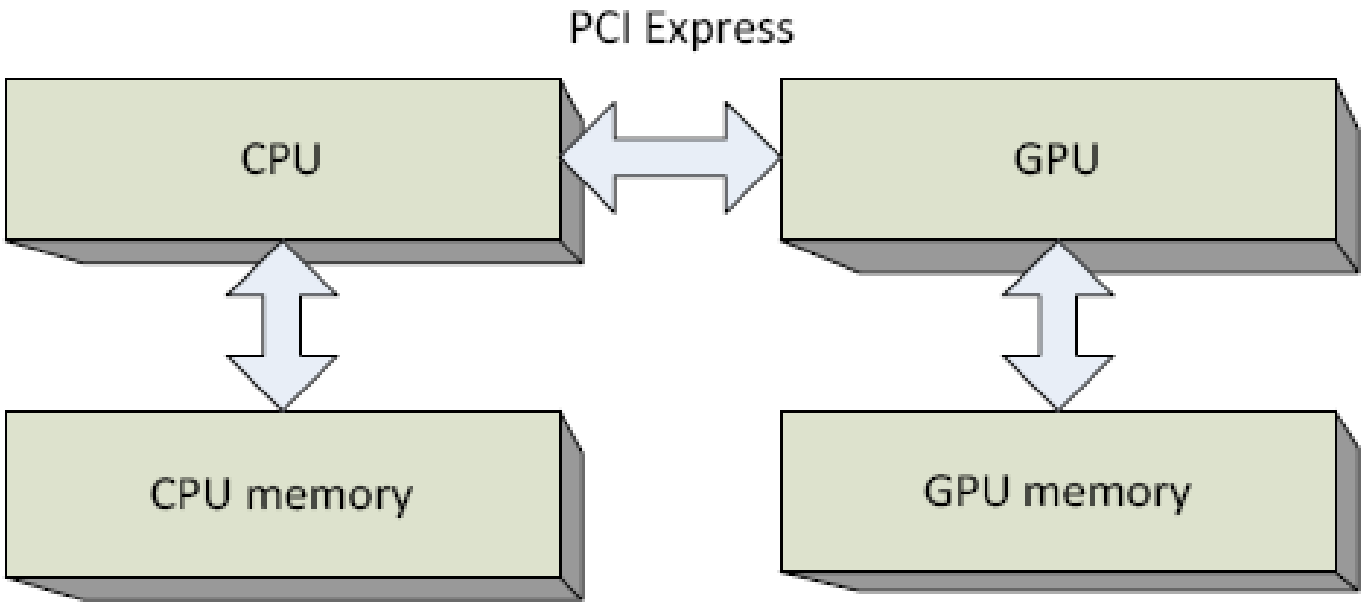
Problems Faced by GPUs

- Need enough parallelism
- Under-utilization
- Bandwidth to CPU

Still a way to go

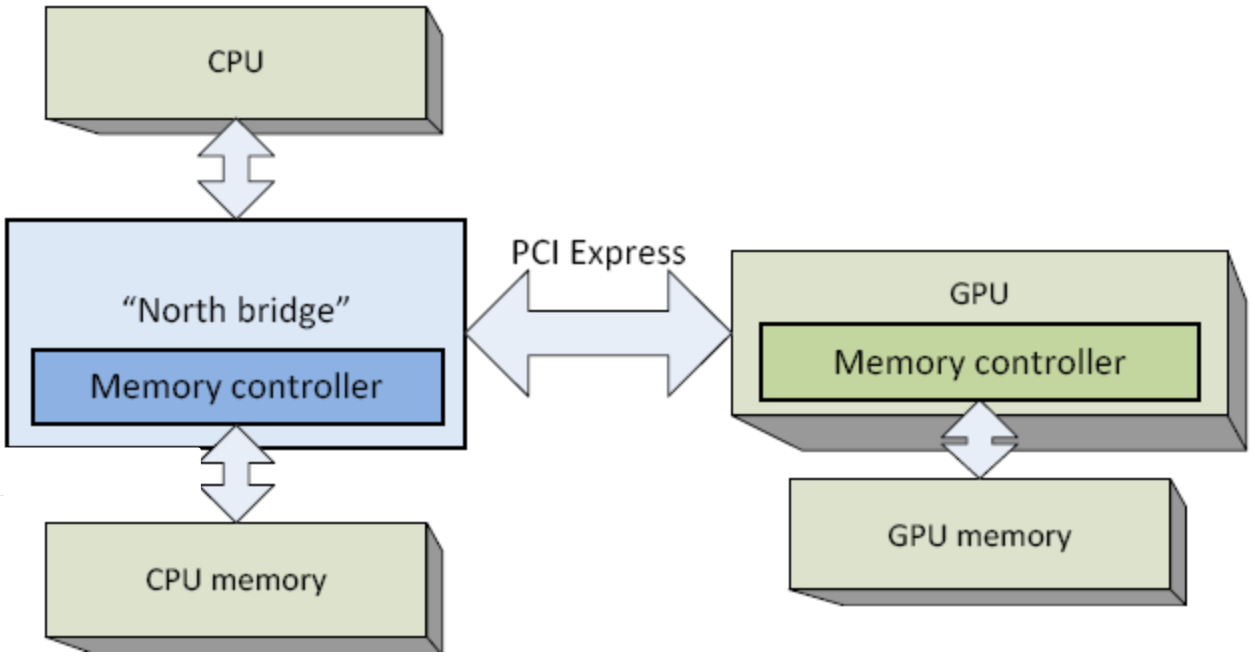
Let's Take A Closer Look: The Hardware

Simplified View



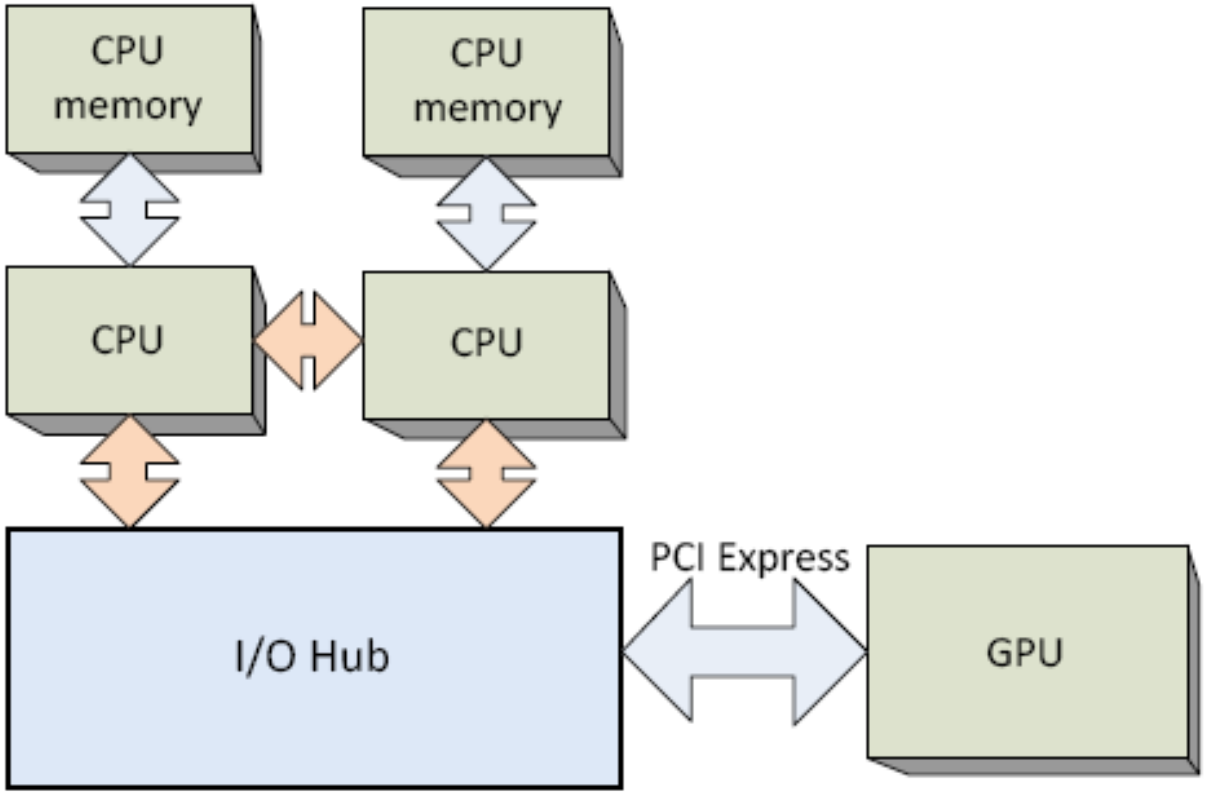
Source: “The CUDA Handbook” by Nicholas Wilt .. Copyright (c) by Pearson Education Inc.

A Closer Look …



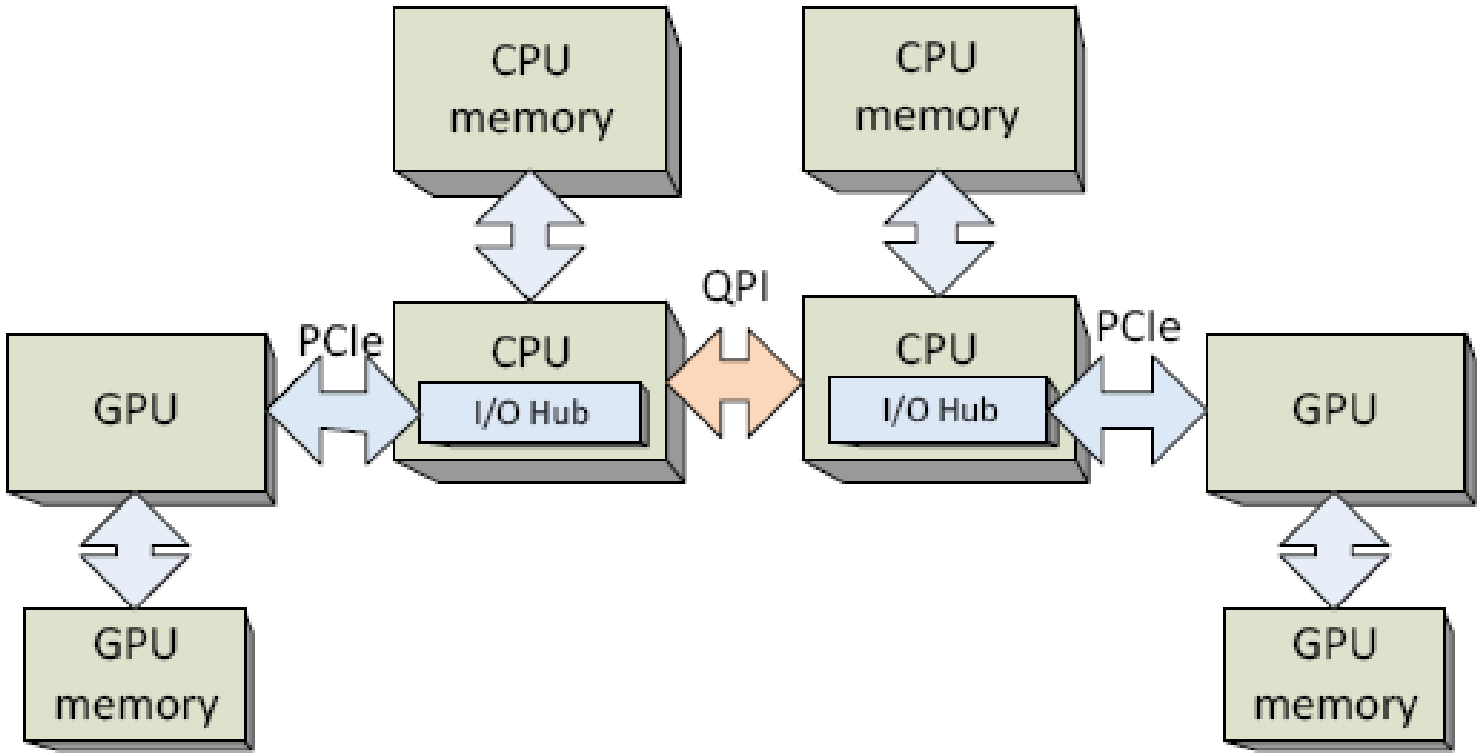
Source: "The CUDA Handbook" by Nicholas Wilt .. Copyright (c) by Pearson Education Inc.

How about multi-CPU? Also: memory controller in CPU

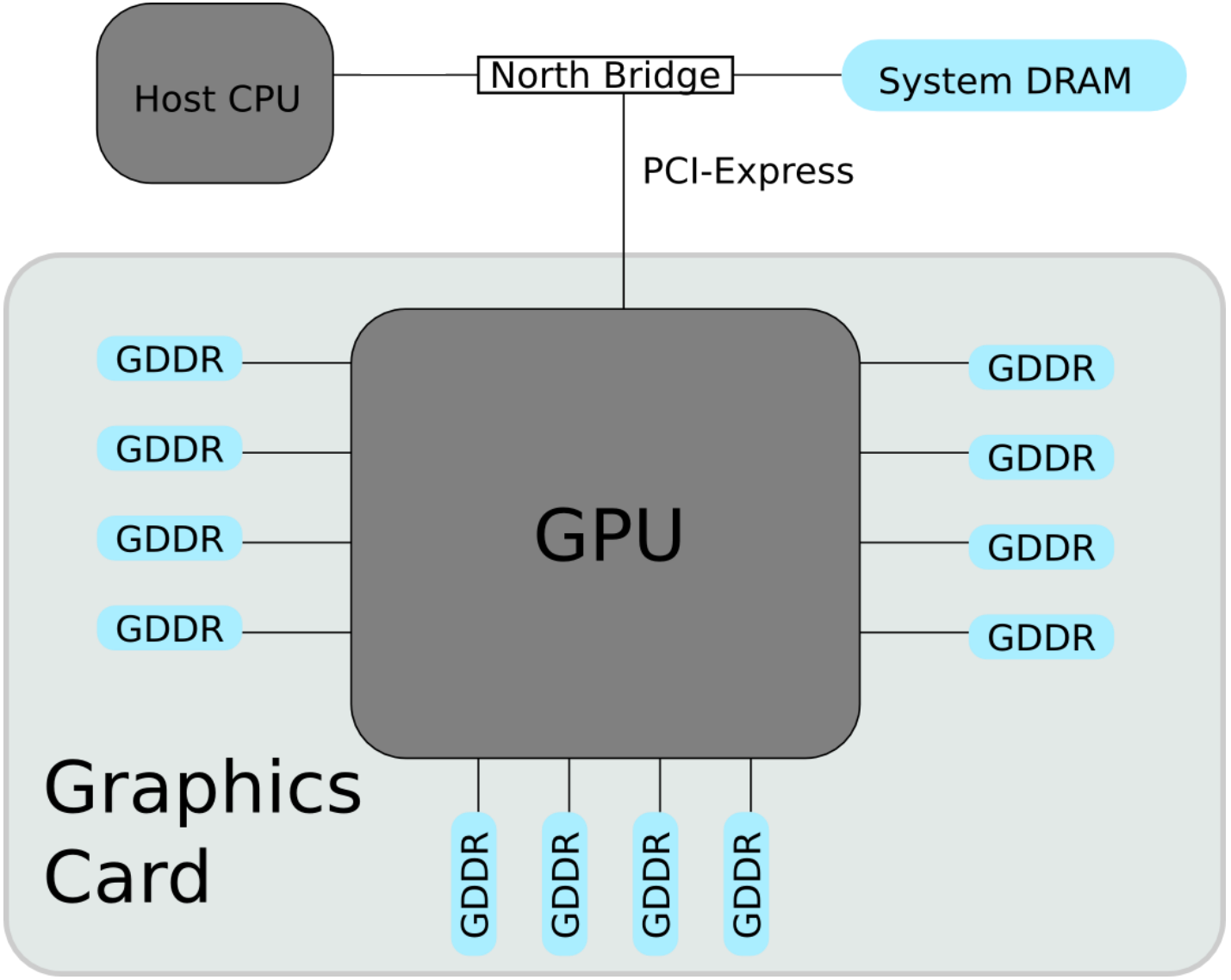


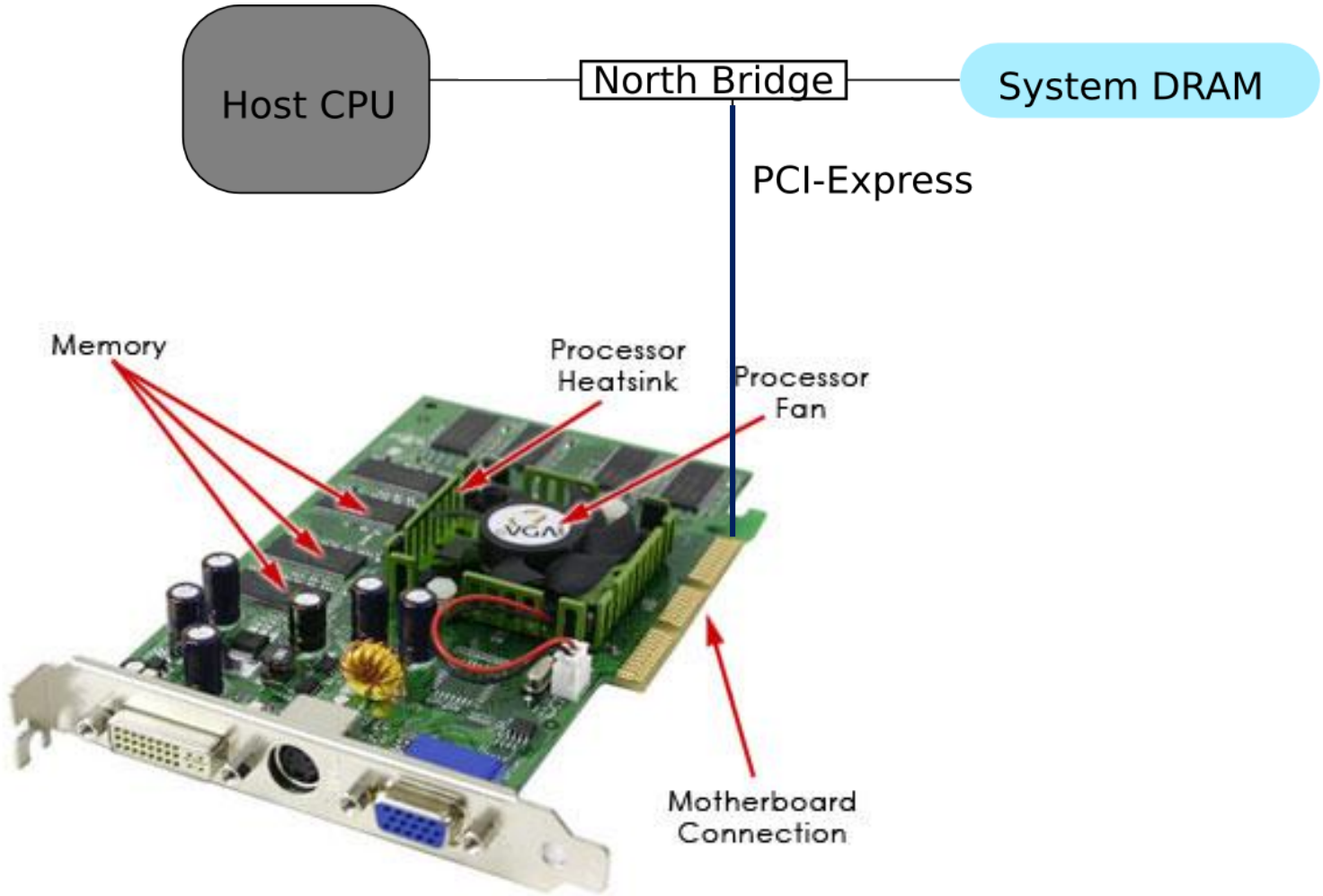
Source: “The CUDA Handbook” by Nicholas Wilt .. Copyright (c) by Pearson Education Inc.

I/O Hub inside the CPU



Source: “The CUDA Handbook” by Nicholas Wilt .. Copyright (c) by Pearson Education Inc.


Host CPU
North Bridge
System DRAM
PCI-Express
GDDR
GDDR
GDDR
GDDR
GPU
GDDR
GDDR
GDDR
GDDR
Graphics Card
GDDR
GDDR
GDDR
GDDR



source: http://static.ddmcdn.com/gif/graphics-card-5.jpg

The Interconnection: CPU-GPU and GPU-GPU

PCIe

Crossfire

About Connections

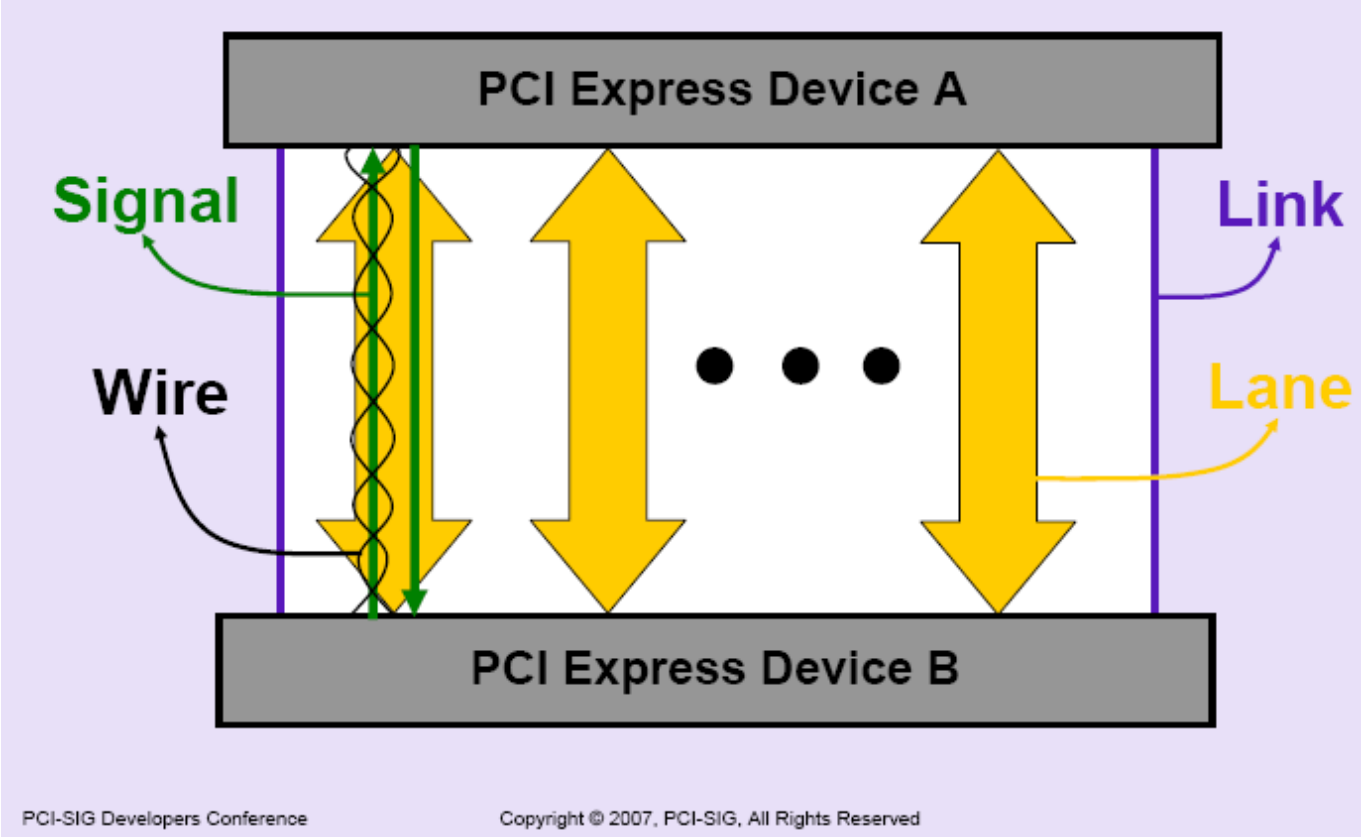
SLI

NVLINK

PCIe

- **Peripheral Component Interconnect**
- Developed by Intel
- PCI Express architecture is a high performance, IO interconnect for peripherals.
- A serial point-to-point interconnect between two devices
- Data sent in packets
- Each lane enables 250 MBytes/s bandwidth per direction.
- Synchronous
- No shared bus but a shared switch

PCIe

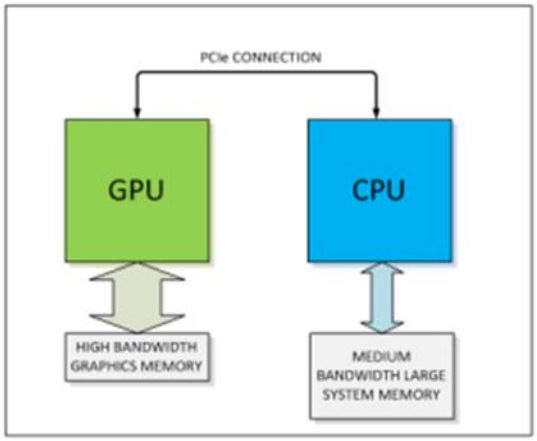


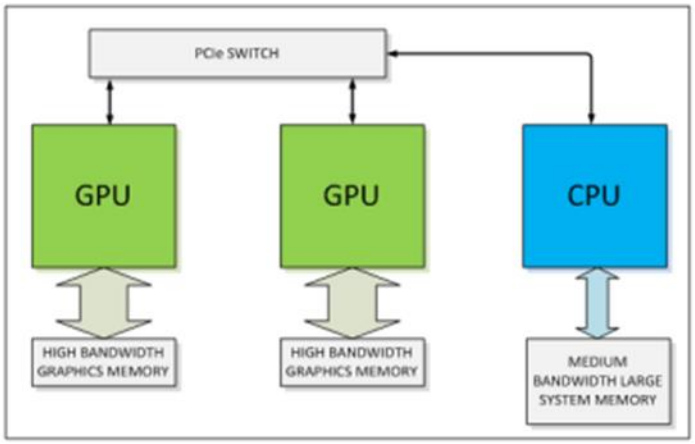
Link Width	x1	x2	x4	x8	x12	x16	x32
Aggregate BW (GBytes/s)	0.5	1	2	4	6	8	16

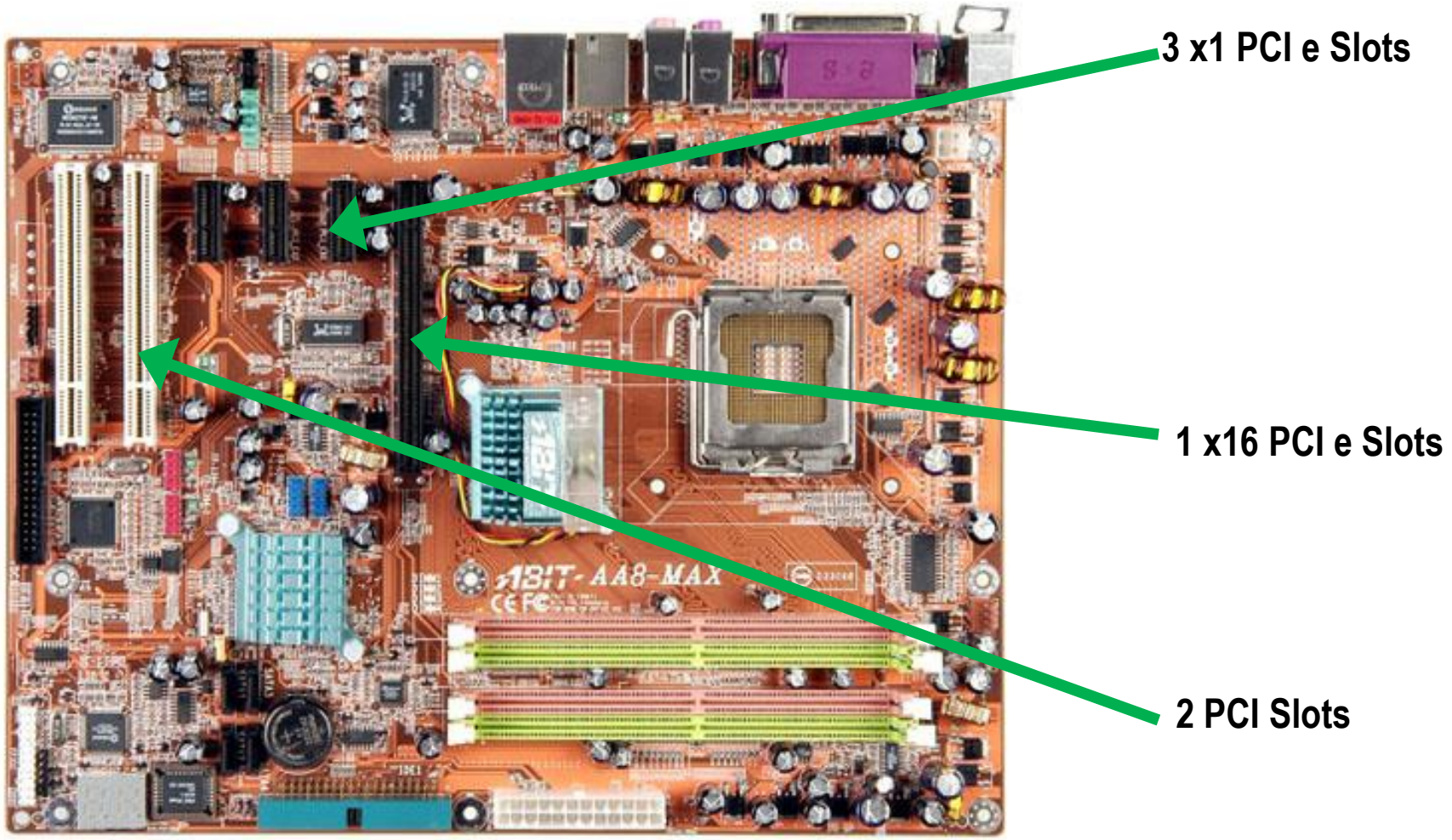
Speed for v3.0

Speed of PCIe

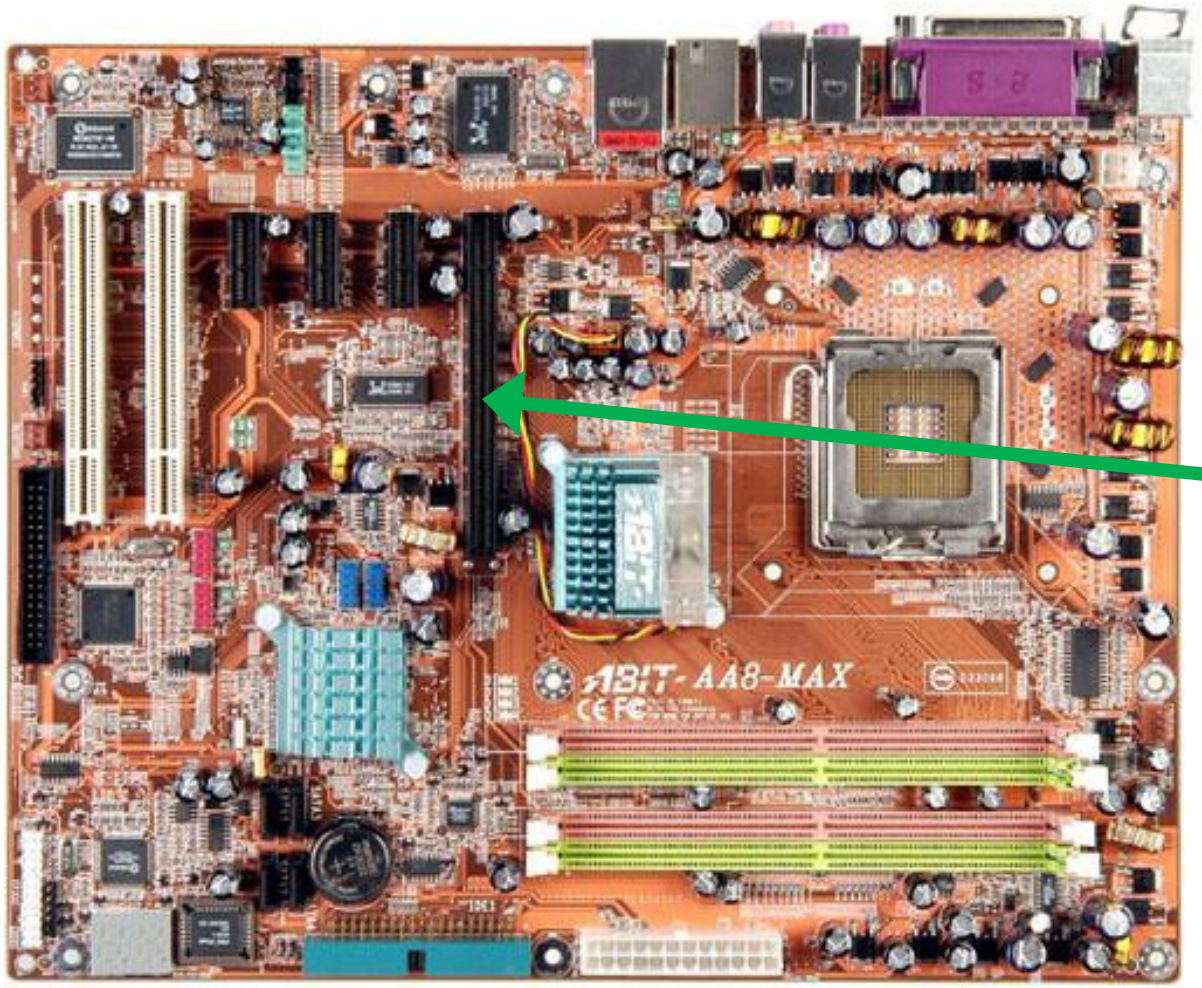
Version		Speed (x1)
1.0	2.5 GT/s	250 MB/s
2.0	5 GT/s	500 MB/s
3.0	8 GT/s	984.6 MB/s
4.0	16 GT/s	1969 MB/s
5.0(expected in 2019)	32 or 25 GT/s	3938 or 3077 MB/s







Source: National Instruments



1 x16 PCI e Slots

Source: National Instruments

SLI

- **Scalable Link Interface**
- Developed by NVIDIA
- Enables inter-GPU communication of up to 1GB/s
- Consumes no bandwidth over the PCIe bus.
- 2,3, or 4 graphics cards

Crossfire

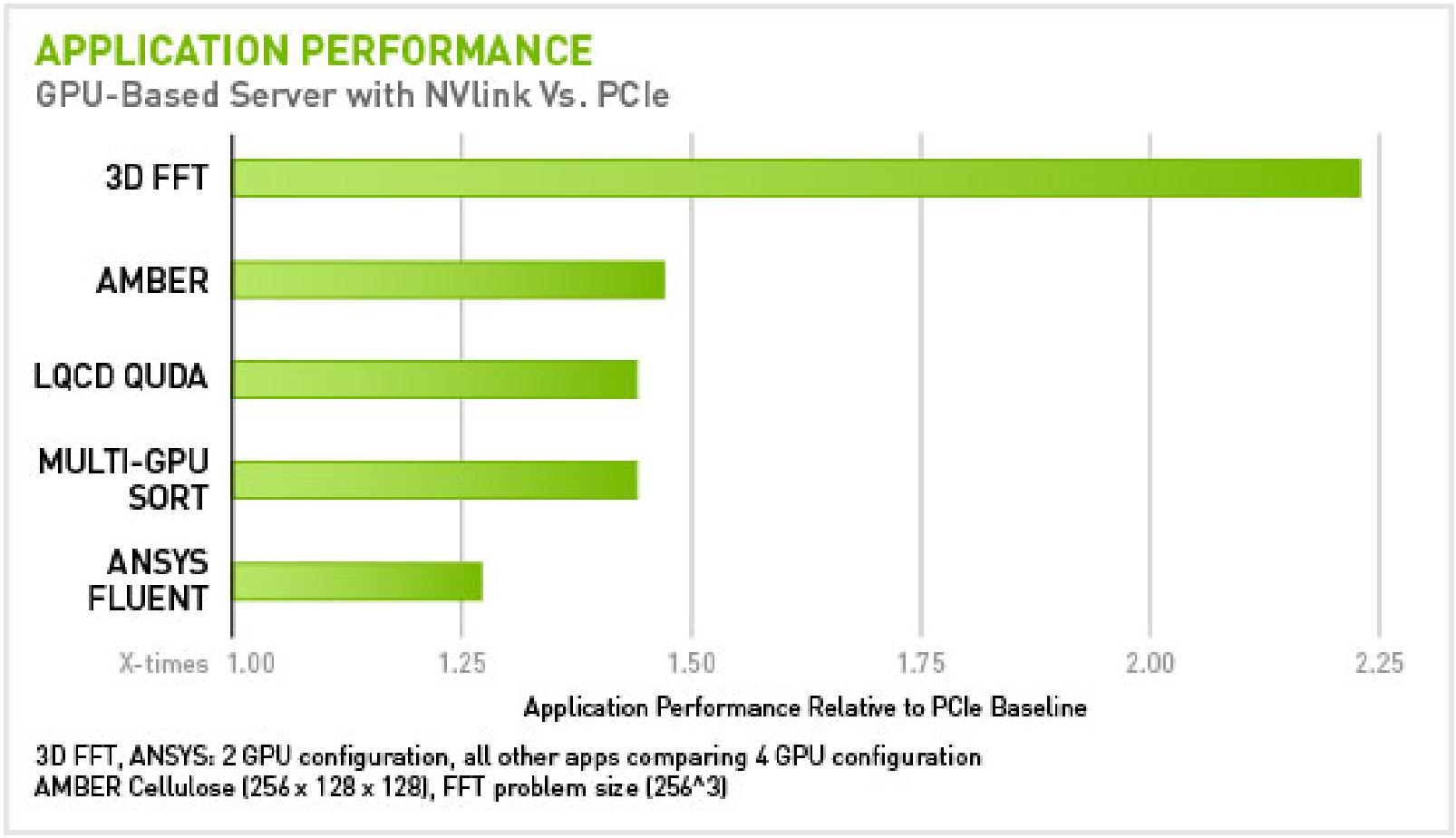
- From AMD
- Very similar technology as SLI

Source: http://www.pcworld.com/article/2023630/how-to-trick-out-your-gaming-pc-with-multiple-graphics-cards.html

NVLINK

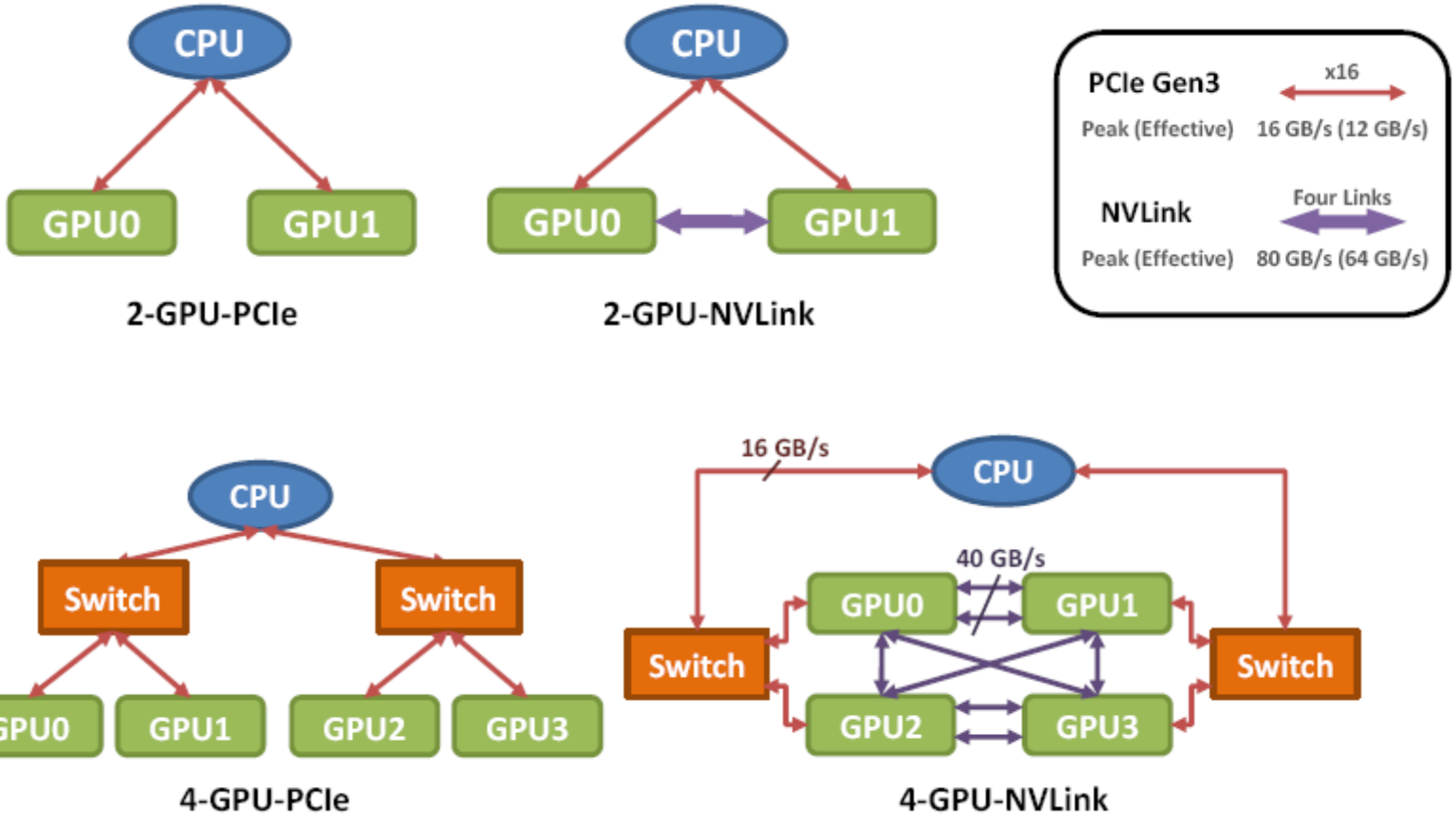
- From NVIDIA
- Starting from PASCAL chips
- higher-bandwidth alternative to PCI Express 3.0
- GPU-to-GPU connections
- Also expected: CPU-GPU
- Allows data sharing at rates 5 to 12 times faster than the traditional PCIe.
- Next generation will support coherence among chips

NVLINK



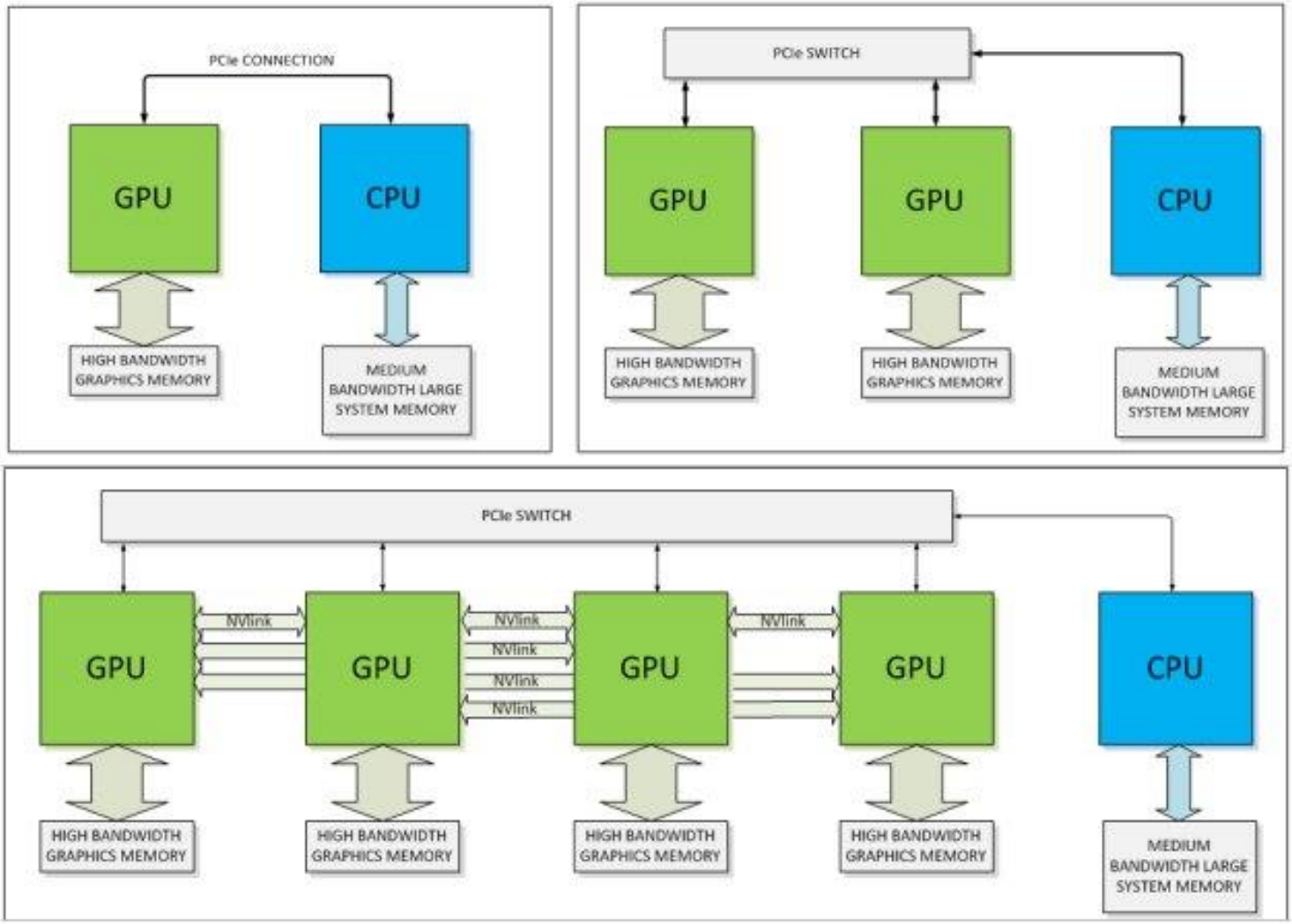
source: http://www.nvidia.com/object/nvlink.html

NVLINK



source: NVIDIA® NVLink TM High-Speed Interconnect: Application Performance whitepaper, November 2014.

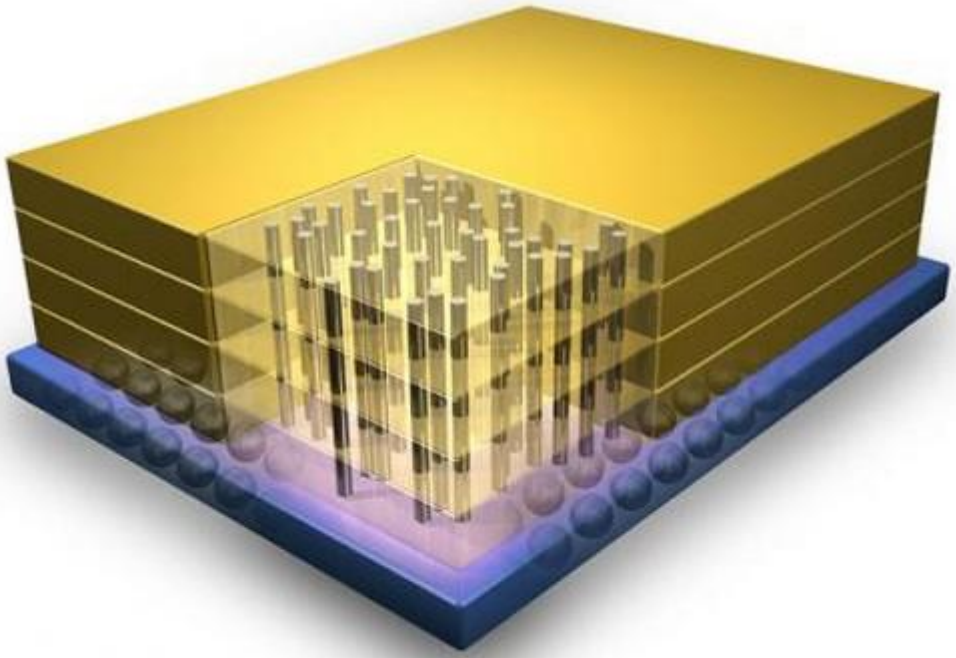
NVLINK



Source: http://gadgets.ndtv.com/laptops/news/nvidia-announces-nvlink-architecture-3d-stacked-memory-pascal-gpu-500335

Memory Hardware
2D → 3D

Stacked Memory



3D instead of 2D

Source: http://www.blogcdn.com/www.engadget.com/media/2011/12/ibmtoproduce2.jpg

Stacked Memory

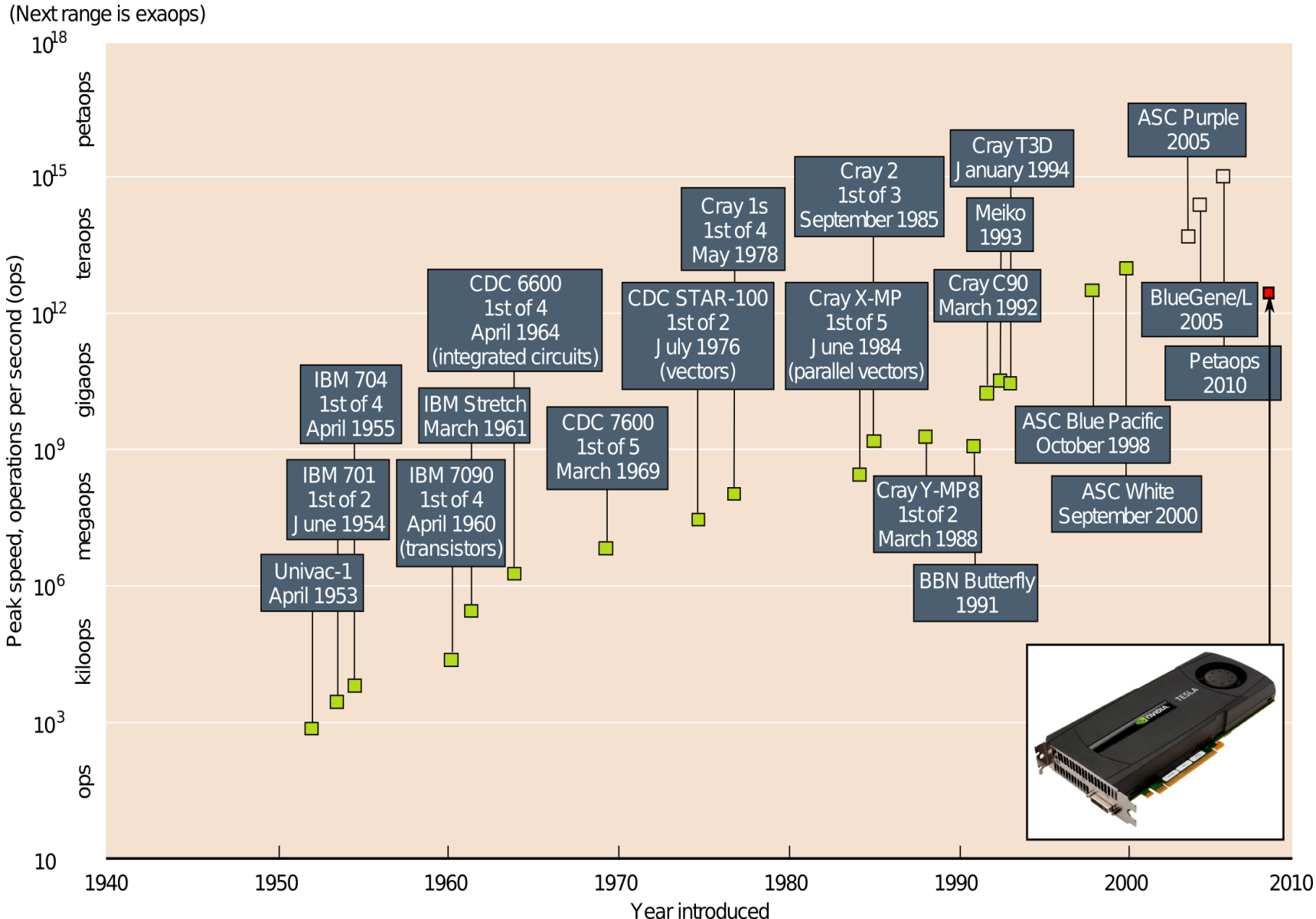
- Higher chip capacity
- Increases bandwidth not latency, in general
- Different flavors:
  - Hybrid memory cubes
  - High bandwidth memory

Modern GPU Hardware
NVIDIA

Modern GPU Hardware

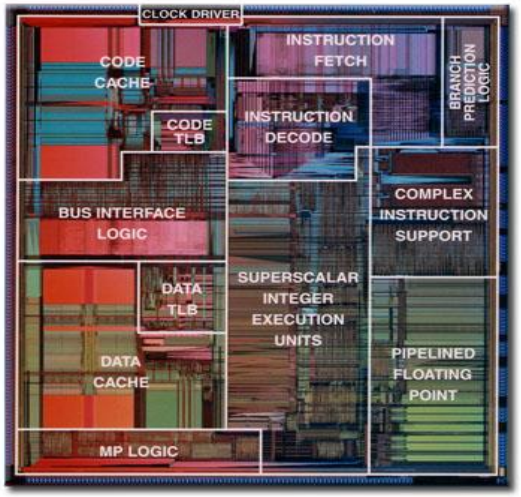
- GPUs have many parallel execution units (called cores in general, and SP or CUDA cores in NVIDIA lingo) and higher transistor counts, while CPUs have few execution units and higher clock speeds
- GPUs have much deeper pipelines (several hundreds stages vs 10-20 for CPUs)
- GPUs have significantly faster and more advanced memory interfaces as they need to shift around a lot more data than CPUs

Single-Chip GPU vs Supercomputers

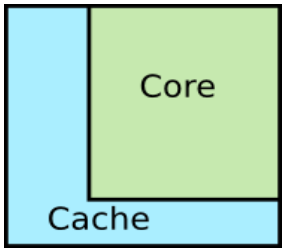


Evolution of Intel Pentium

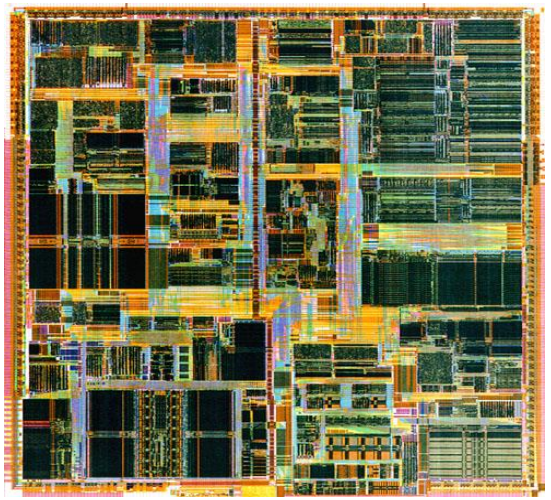
Pentium I

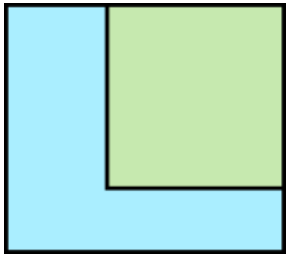


Chip area breakdown

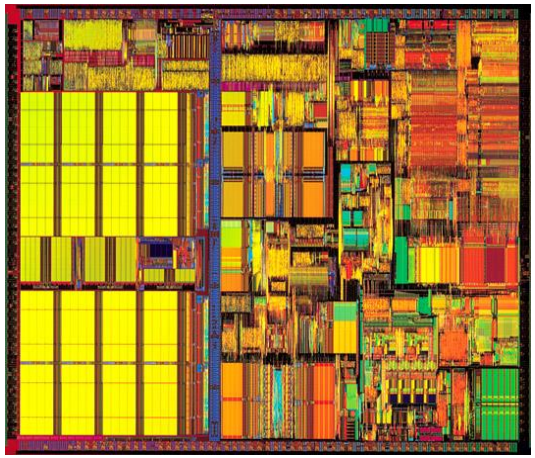


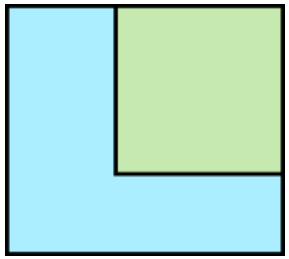
Pentium II



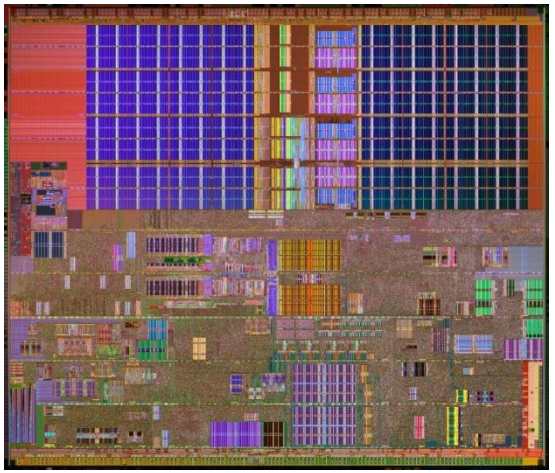


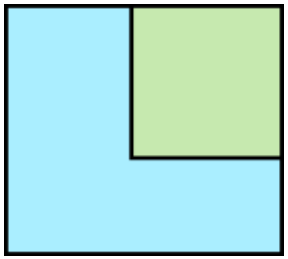
Pentium III





Pentium IV

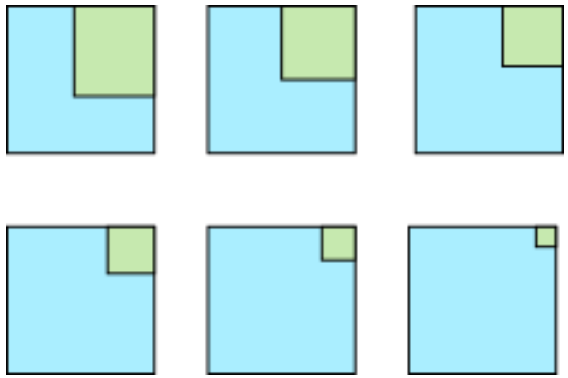




source: http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/GPU.ppt

Extrapolation of Single Core CPU

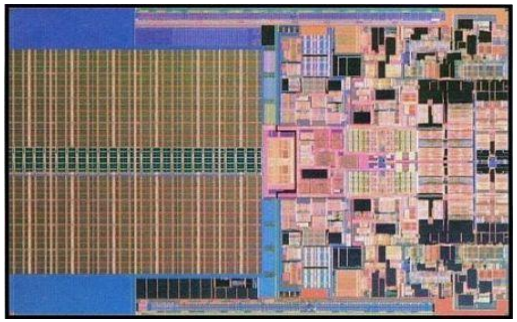
If we extrapolate the trend, in a few generations, Pentium will look like:



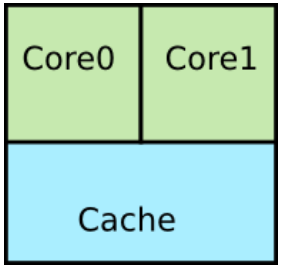
source: http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/GPU.ppt

Evolution of Multi-core CPUs

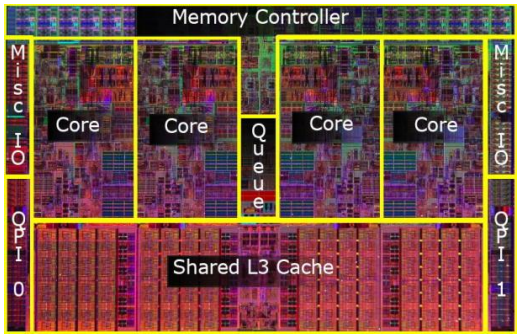
Penryn

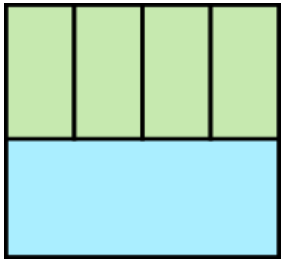


Chip area breakdown

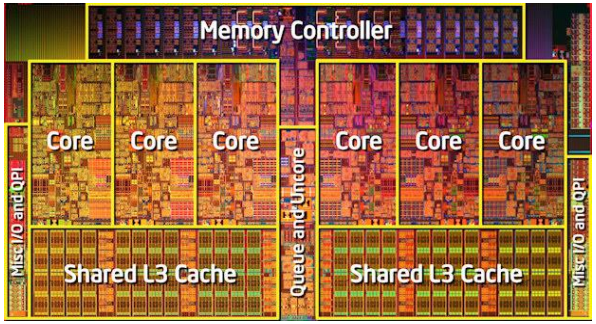


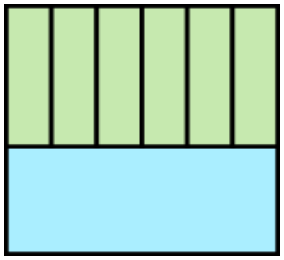
Bloomfield



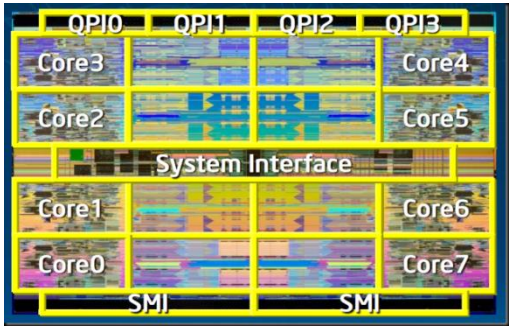


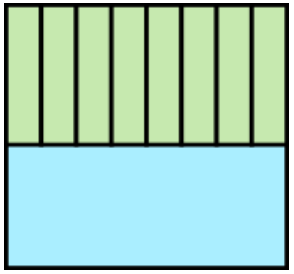
Gulftown





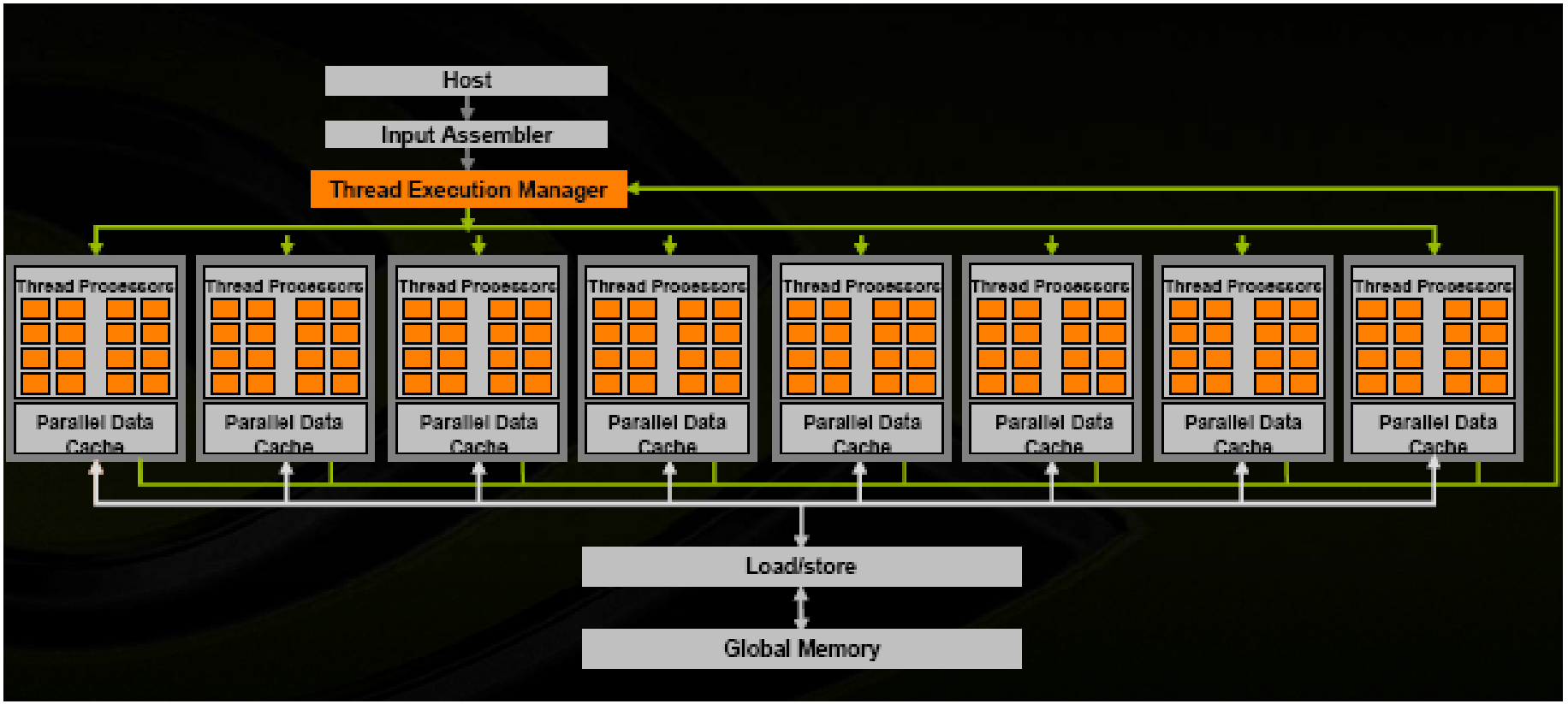
Beckton





source: http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/GPU.ppt

This is how we expose GPU as parallel processor.



Scalar vs Vector vs Threaded

Scalar program

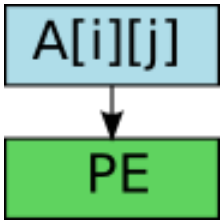
```
float A[4][8];

for(int i=0;i<4;i++){

    for(int j=0;j<8;j++){
        A[i][j]++;
    }
}
```

Vector Program

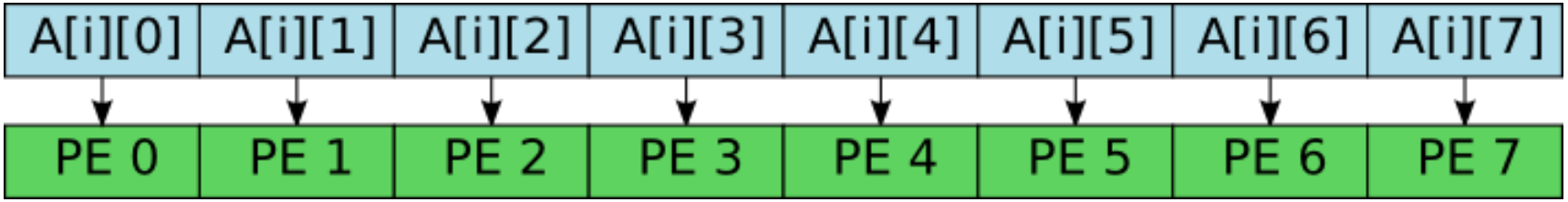
Vector width is exposed to programmers.

Vectorizing compilers
are doing good job,
if helped by the programmer!

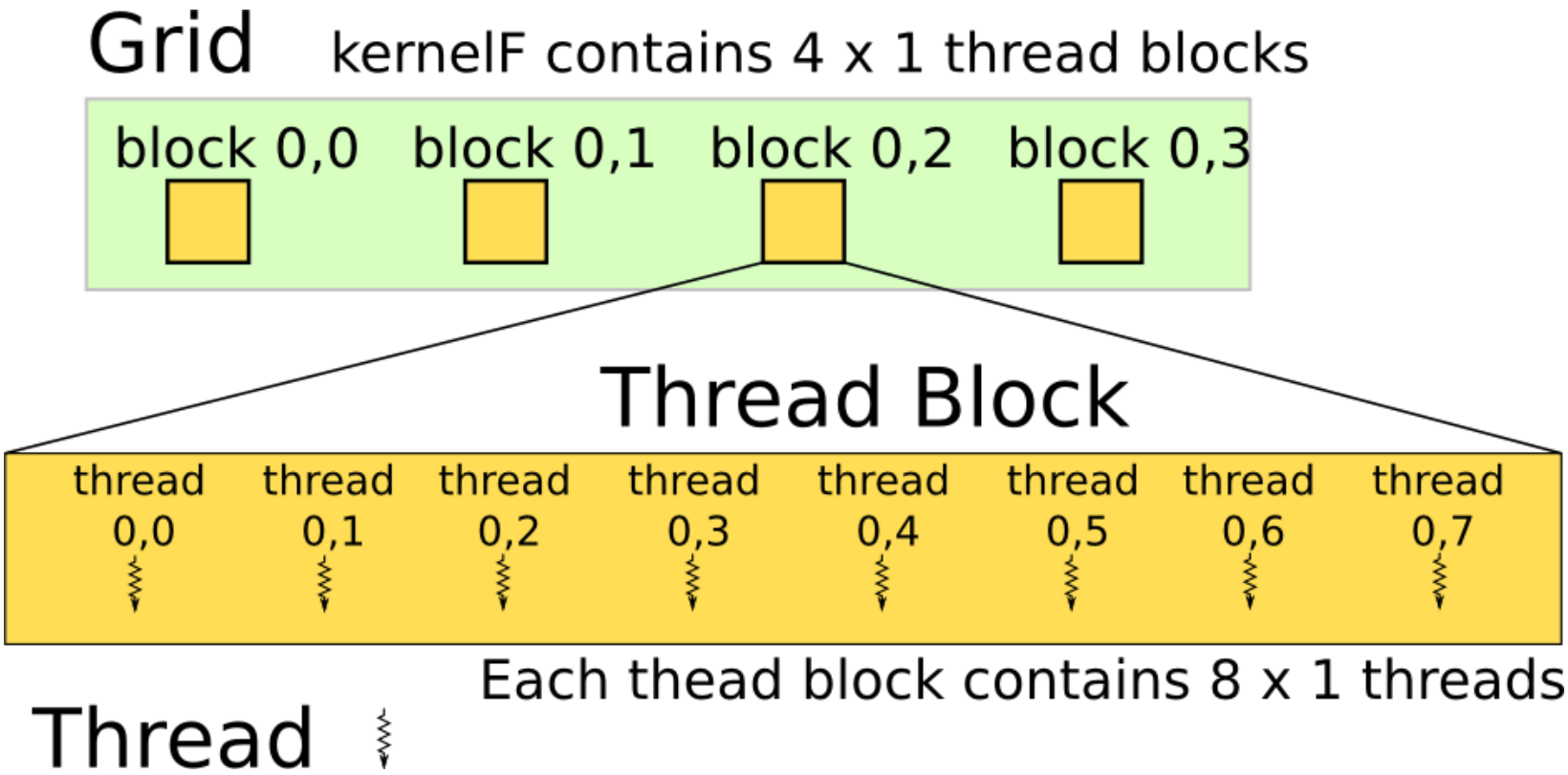
Scalar



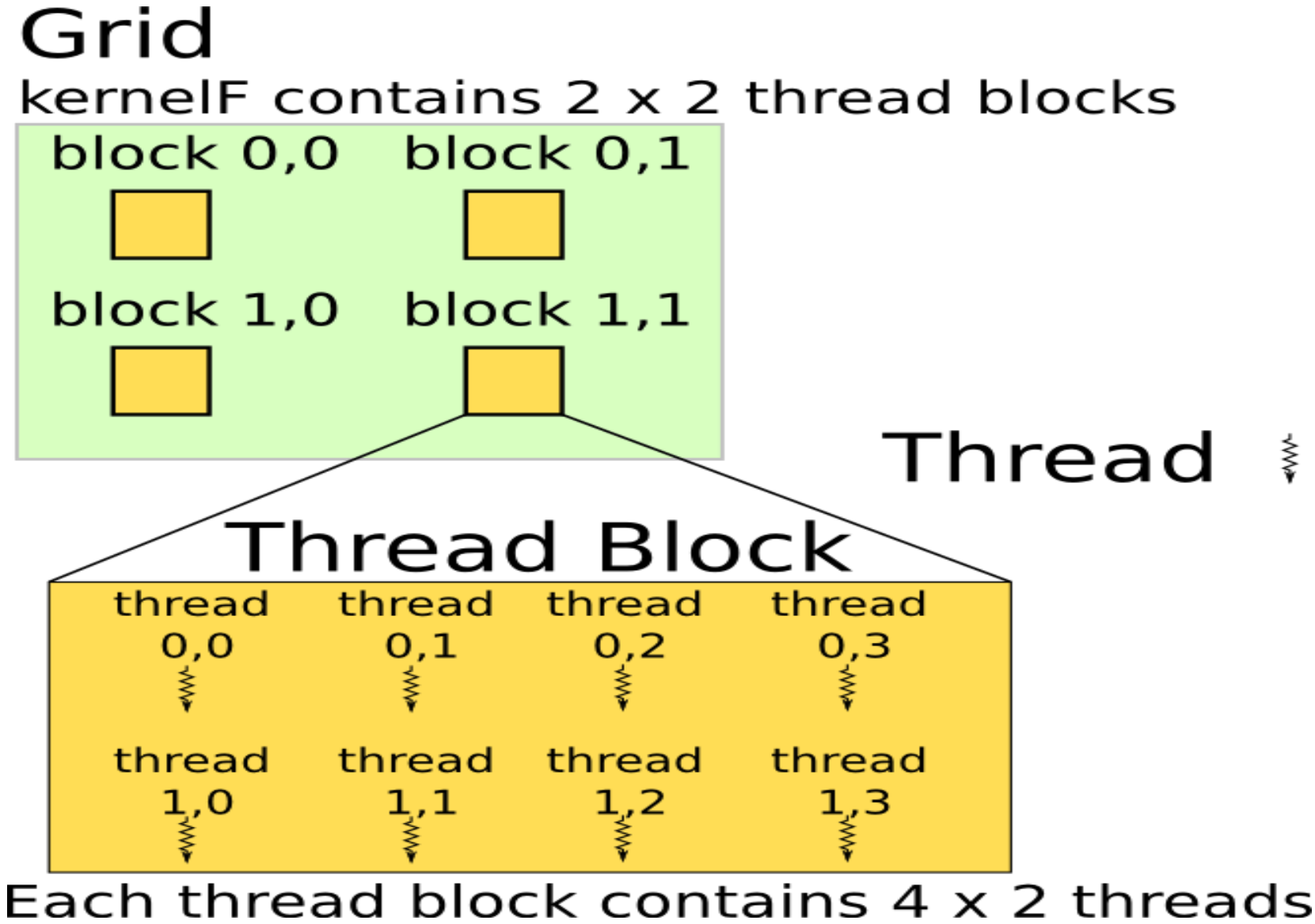
Vector of width 8



Multithreaded: (4x1) blocks – (8x1) threads



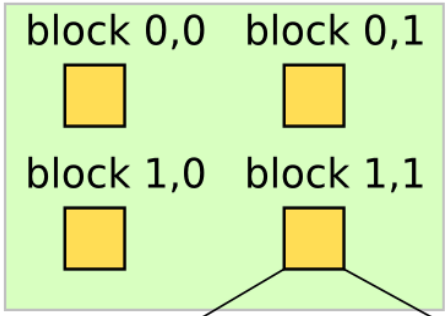
Multithreaded: (2x2)blocks – (4x2) threads

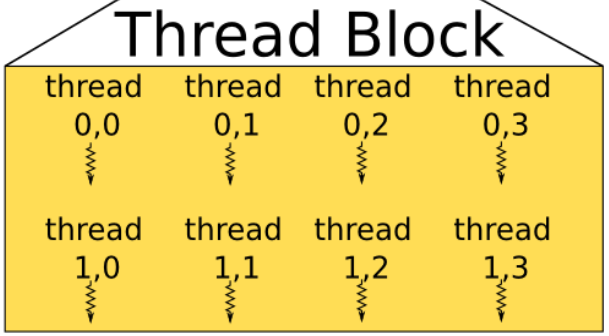


Scheduling Thread Blocks on SM

Grid

kernelF contains 2 x 2 thread blocks

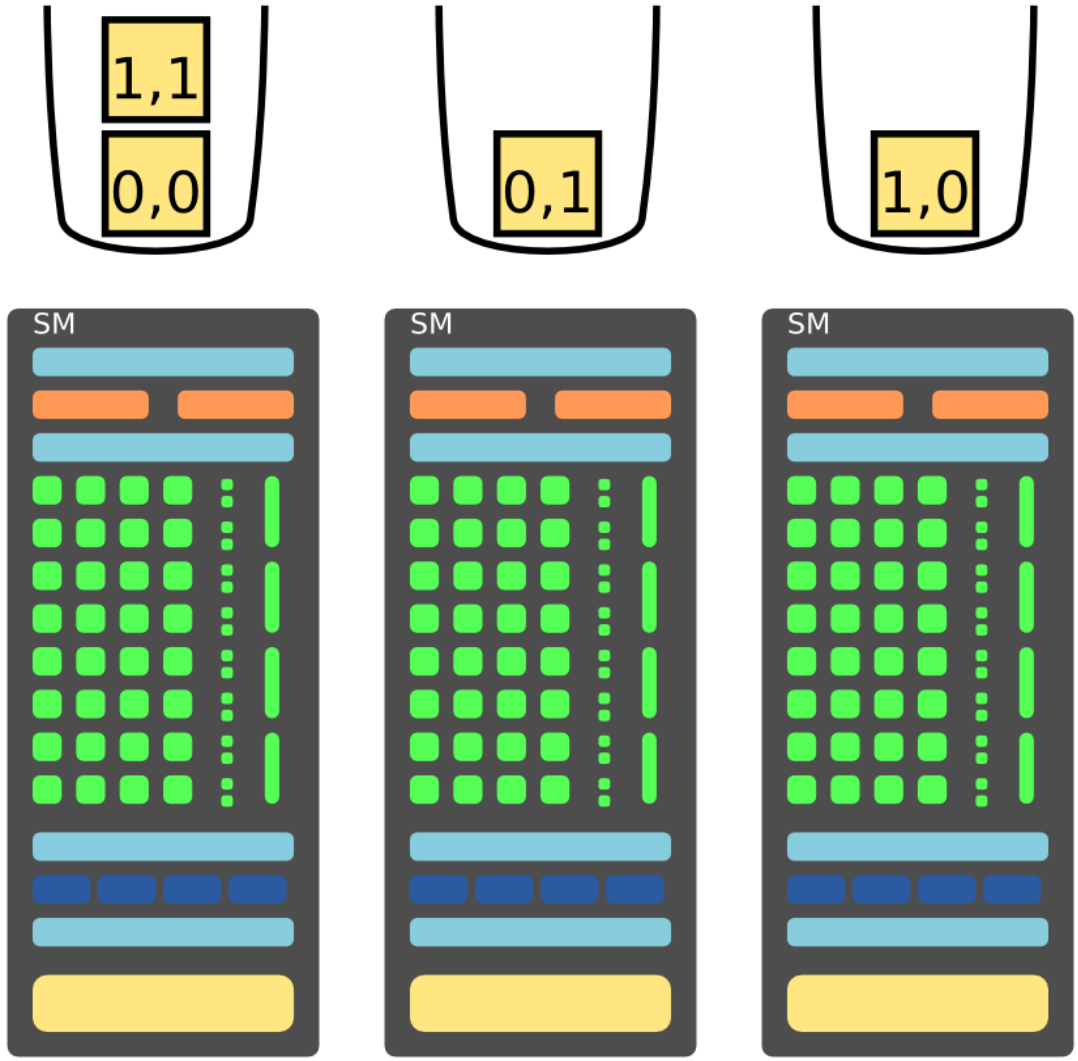




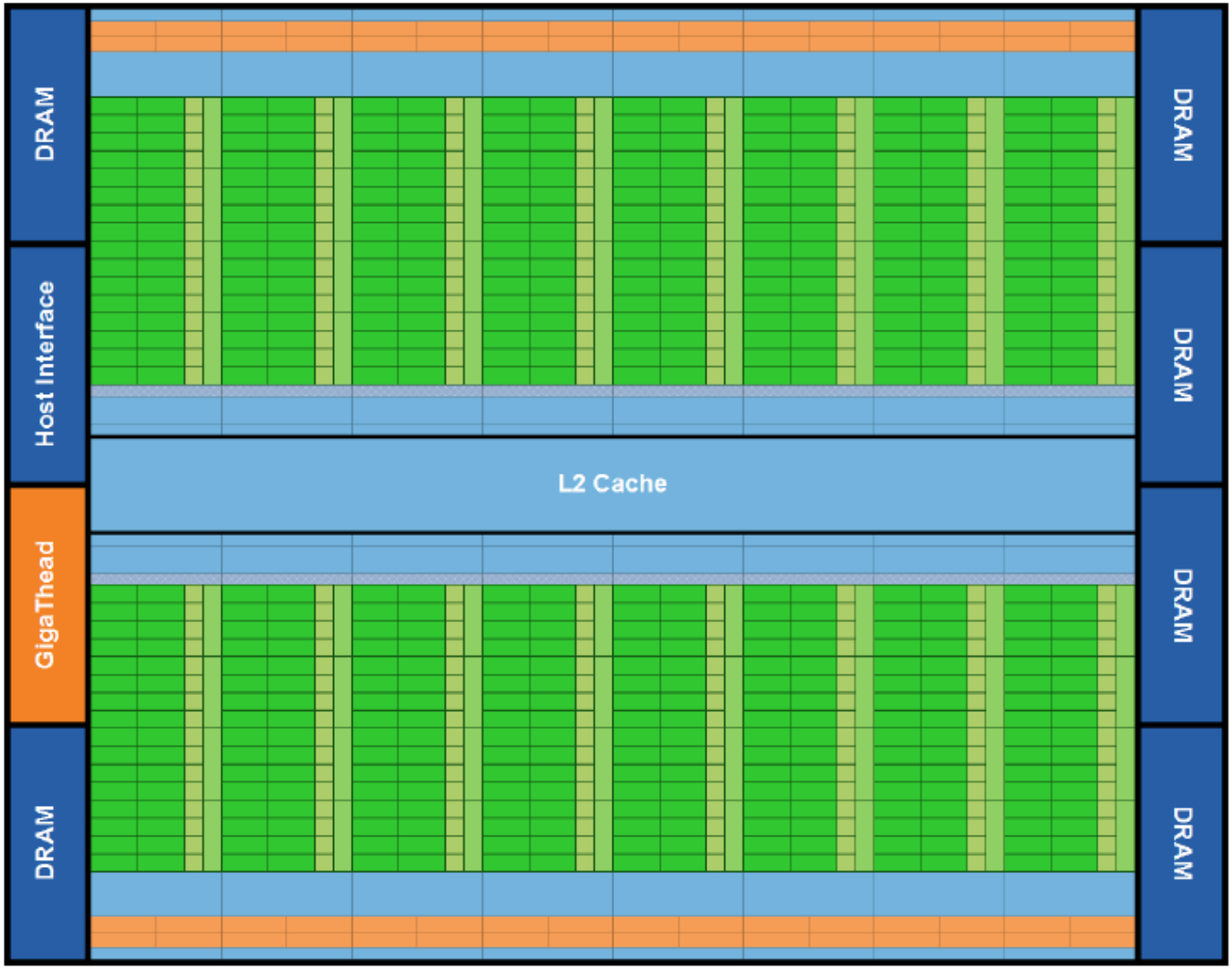
Each thread block contains 4 x 2 threads

Example:

Scheduling 4 thread blocks on 3 SMs.



GPU FERMI



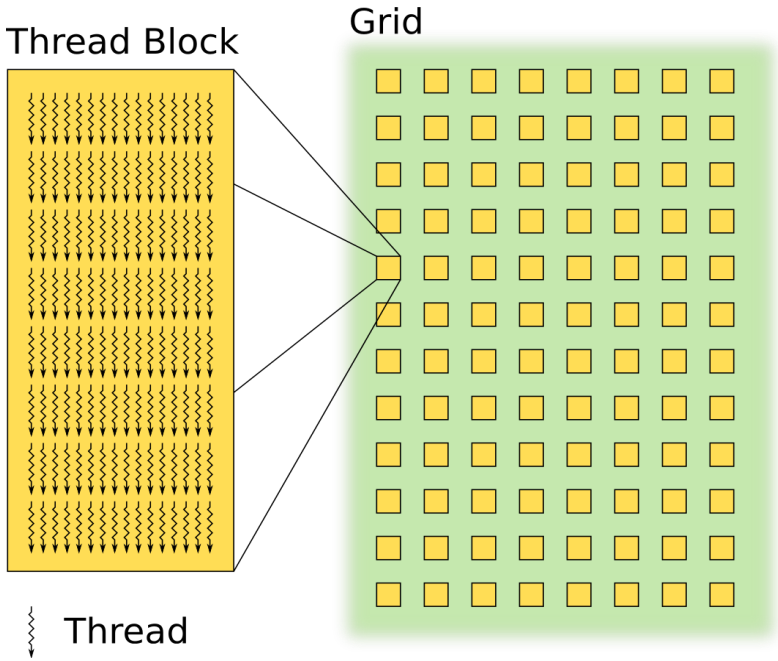
32 cores/SM

~3B Transistors

Glimpse at the programming model …
Linking the hardware to the software …
What does the programmer see?

Quick Glimpse At GPU Programming Model

Application → **Kernels** → **Threads** → **Blocks**



↓

Grid

Quick Glimpse At GPU Programming Model

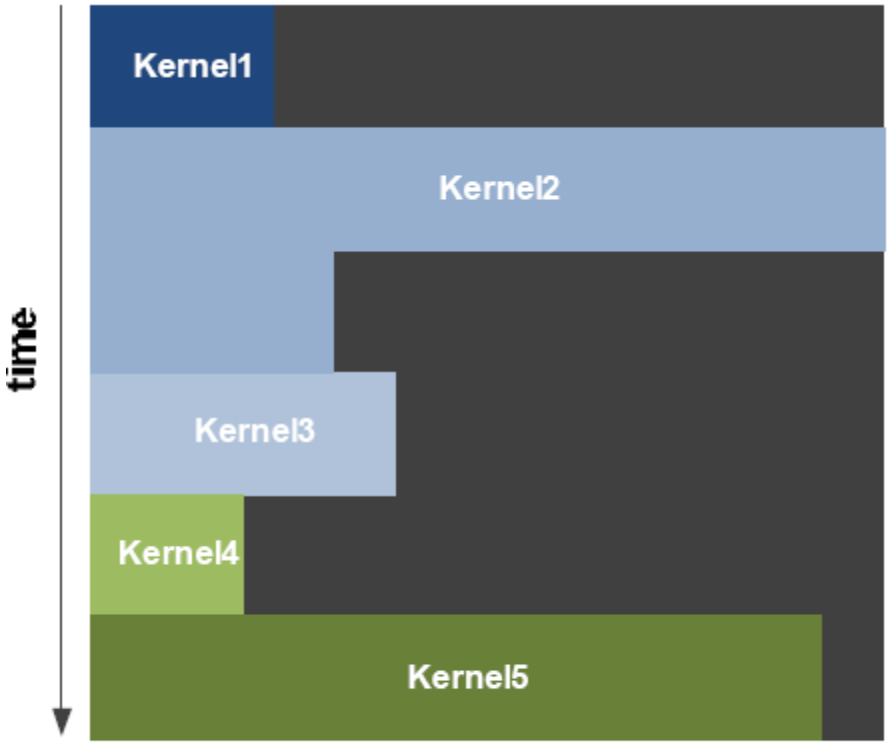
- Application can include multiple kernels
- Threads of the same block run on the same SM
  - So threads in SM can operate and share memory
  - Block in an SM is divided into warps of 32 threads each
  - A warp is the fundamental unit of dispatch in an SM
- Blocks in a grid can coordinate using global shared memory
- Each grid executes a kernel

Scheduling In Modern NVIDIA GPUs

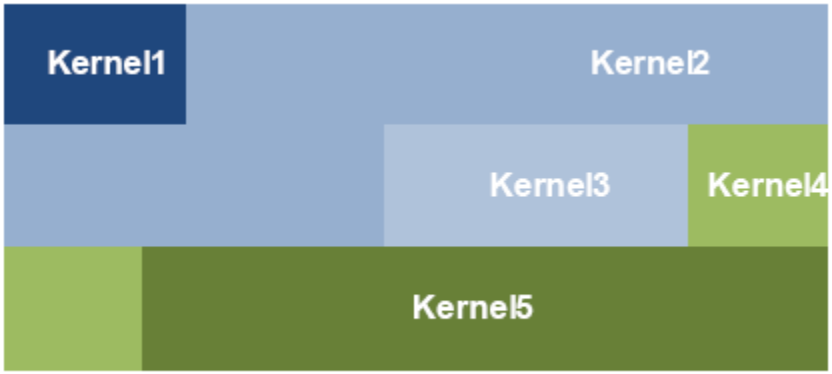
- At any point of time the entire device is dedicated to a single application (*well, more on that later!*)
  - Switch from an application to another takes ~25 microseconds
- GPU can simultaneously execute multiple kernels of the same application
- Two warps from different blocks (or even different kernels) can be issued and executed simultaneously

Scheduling In GPUs

- Two-level, distributed thread scheduler
  - At the device level: a global work distribution engine schedules thread blocks to various SMs
  - At the SM level, each warp scheduler distributes warps of 32 threads to its execution units.



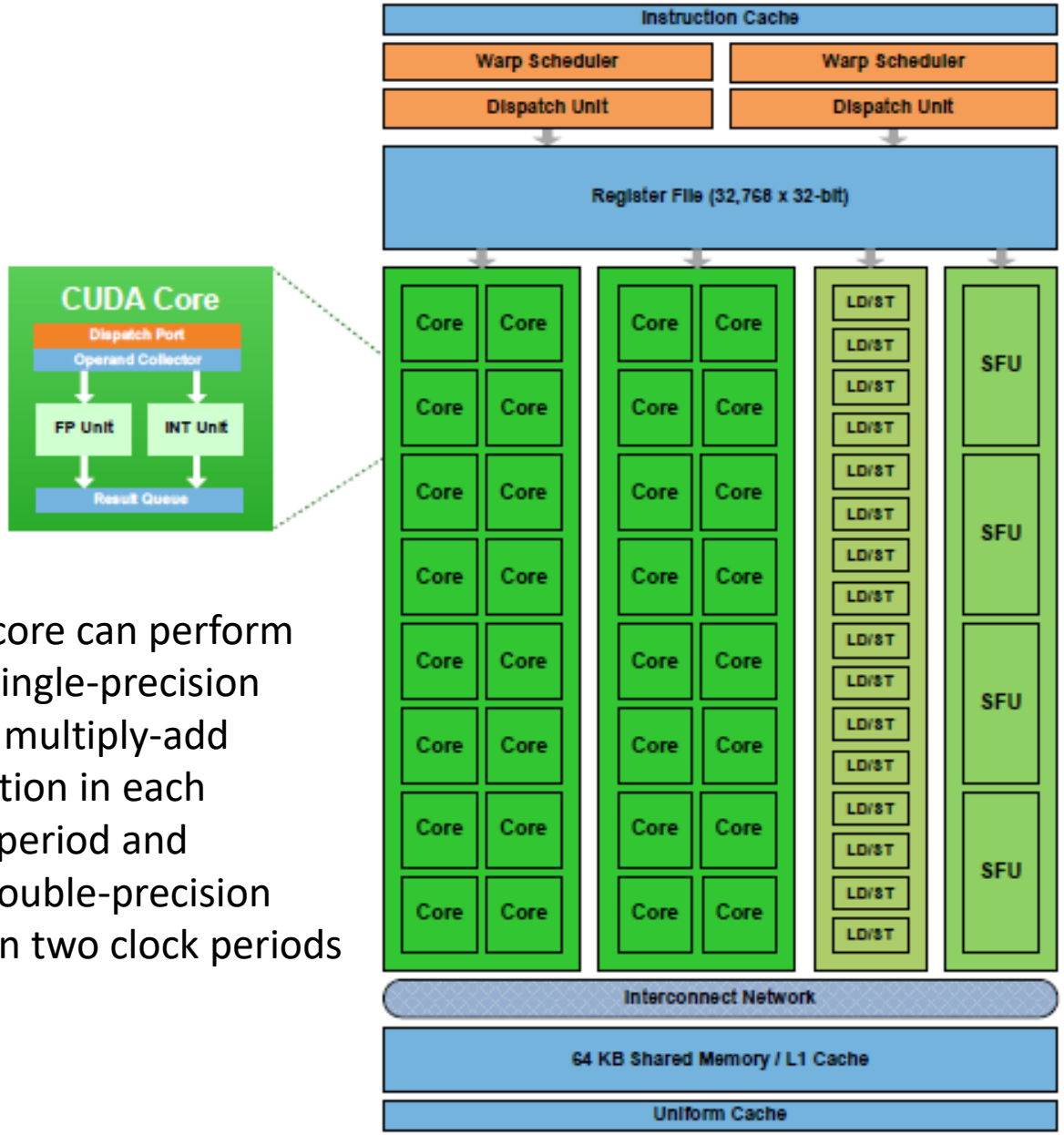
Serial Kernel Execution



Concurrent Kernel Execution

Source: Nvidia

An SM in Fermi



- 32 cores
- SFU = Special Function Unit
- 64KB of SRAM split between cache and local mem

Each core can perform one single-precision fused multiply-add operation in each clock period and one double-precision FMA in two clock periods

The Memory Hierarchy

- All addresses in the GPU are allocated from a continuous 40-bit (one terabyte) address space.
- Global, shared, and local addresses are defined as ranges within this address space and can be accessed by common load/store instructions.
- The load/store instructions support 64-bit addresses to allow for future growth.

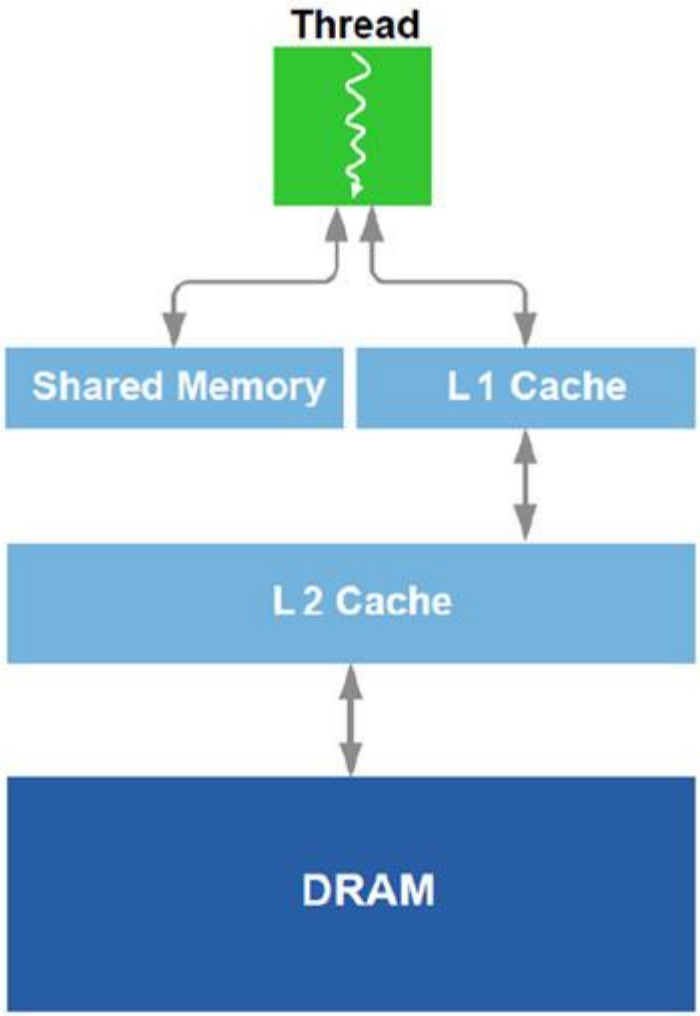
The Memory Hierarchy

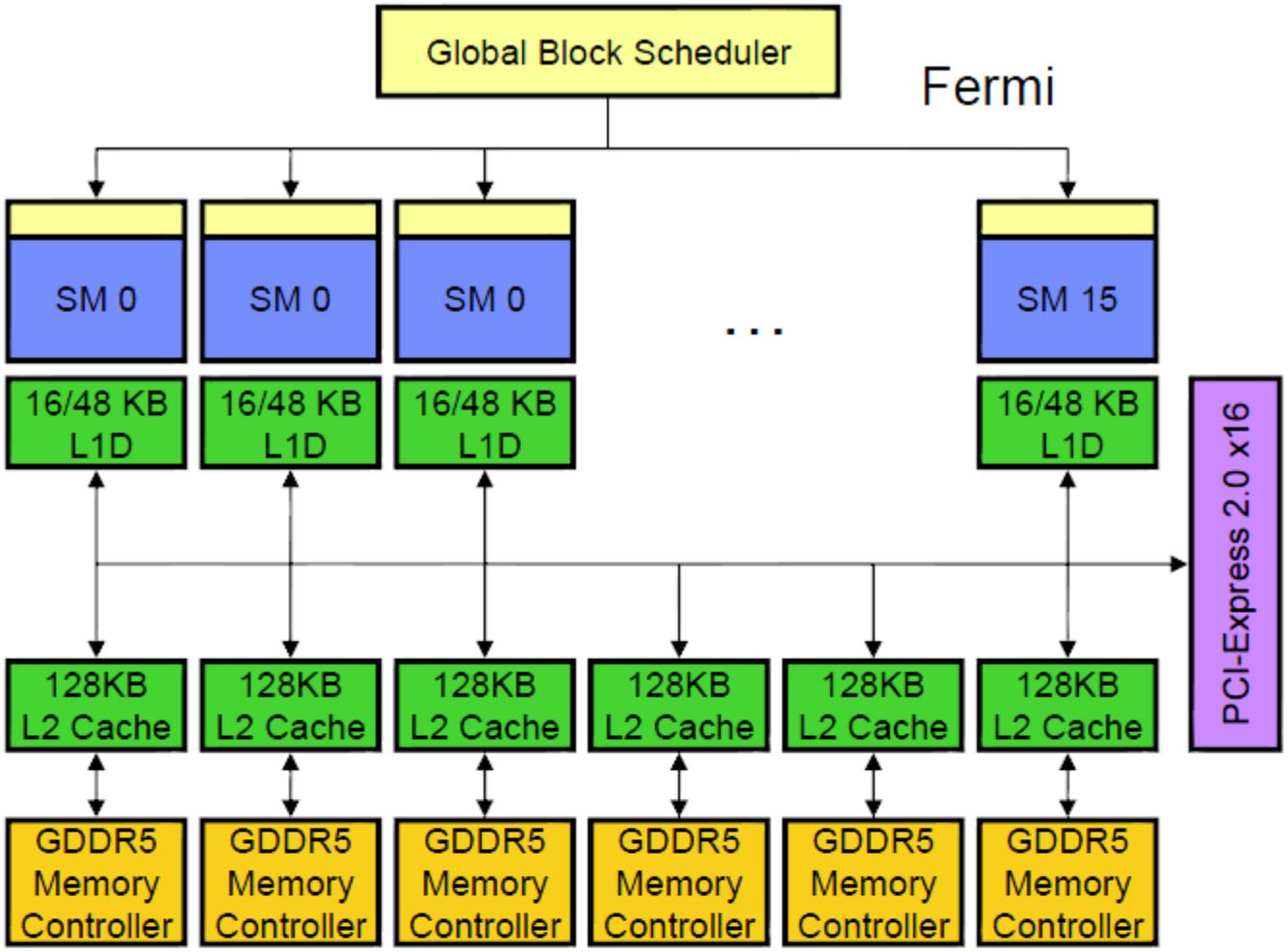
- Local memory in each SM
- The ability to use some of this local memory as a first-level (L1) cache for global memory references.
- The local memory is 64K in size, and can be split 16K/48K or 48K/16K between L1 cache and shared memory.
  - Separate cache/shared memory in Maxwell and Kepler and beyond
- Access latency to this memory is predictable → algorithms can be written to interleave loads, calculations, and stores with maximum efficiency.

The Memory Hierarchy

- Fermi (and later generations) GPUs are also equipped with an L2.
- The L2 cache covers GPU local DRAM as well as system memory.
- The L2 cache subsystem also implements a set of memory read-modify-write operations that are atomic

Fermi Memory Hierarchy




Global Block Scheduler
Fermi
SM 0
SM 0
SM 0
SM 15
16/48 KB L1D
16/48 KB L1D
16/48 KB L1D
16/48 KB L1D
128KB L2 Cache
128KB L2 Cache
128KB L2 Cache
128KB L2 Cache
128KB L2 Cache
128KB L2 Cache
GDDR5 Memory Controller
GDDR5 Memory Controller
GDDR5 Memory Controller
GDDR5 Memory Controller
GDDR5 Memory Controller
GDDR5 Memory Controller
PCI-Express 2.0 x16

Deciphering the specs:
Geforce 1080 Ti SC2 (PASCAL)

Graphics Processing Clusters	6
Streaming Multiprocessors	28
CUDA Cores (single precision)	3,584
Texture Units	224
ROP Units	88
Base Clock	1,480MHz
Boost Clock	1,582MHz
Memory Clock	5,505MHz
Memory Data Rate	11 Gbps
L2 Cache Size	2816K
Total Video Memory	1,126MB GDDR5X
Memory Interface	352-bit
Total Memory Bandwidth	484 GB/s
Texture Rate (Bilinear)	331.5 GigaTexels/sec
Fabrication Process	16 nm
Transistor Count	12 Billion
Connectors	3 x DisplayPort 1 x HDMI
Form Factor	Dual Slot
Power Connectors	One 6-pin, One 8-pin
Recommended Power Supply	600 Watts
Thermal Design Power (TDP)	250 Watts
Thermal Threshold	91°C

group of SMs (here 2 are disabled)

Render Output Unit (or Raster Operation Pipeline)

Glimpse at Non-NVIDIA GPUs

AMD

- Heterogeneous compute compiler (HCC) for C++ programming
- For applications already developed in CUDA, they can now be ported into C++.
  - This is achieved using the new Heterogeneous-computing Interface for Programmers (HIP) tool that ports CUDA runtime APIs into C++ code.
- The architecture of AMD GPU is called Graphics-core next (GCN)
  - Used in all product lines from low-end to high-end

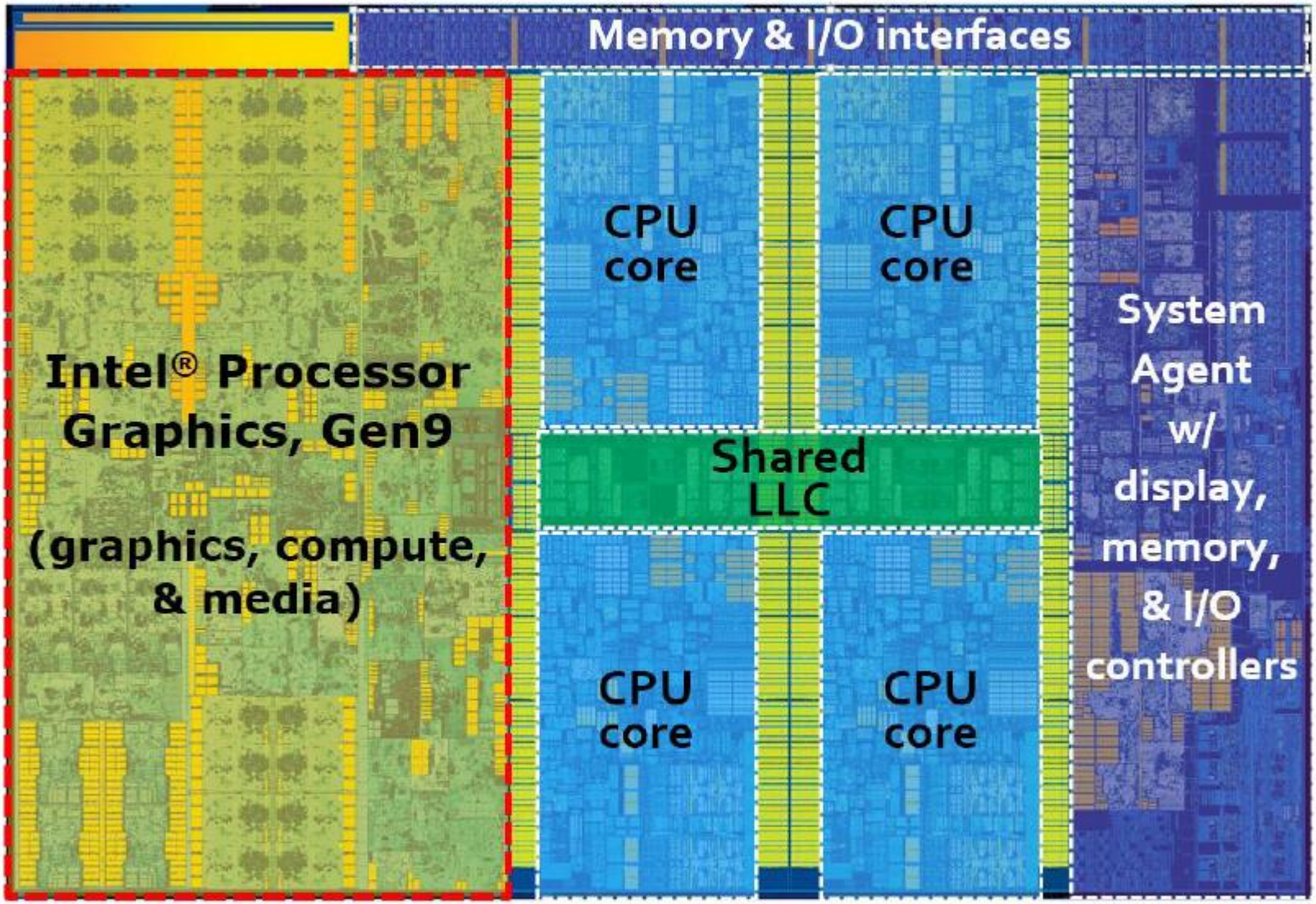
AMD Radeon R9 Fury X

Fiji GPU Architecture (AMD's Graphics Core Next (GCN) microarchitecture.)

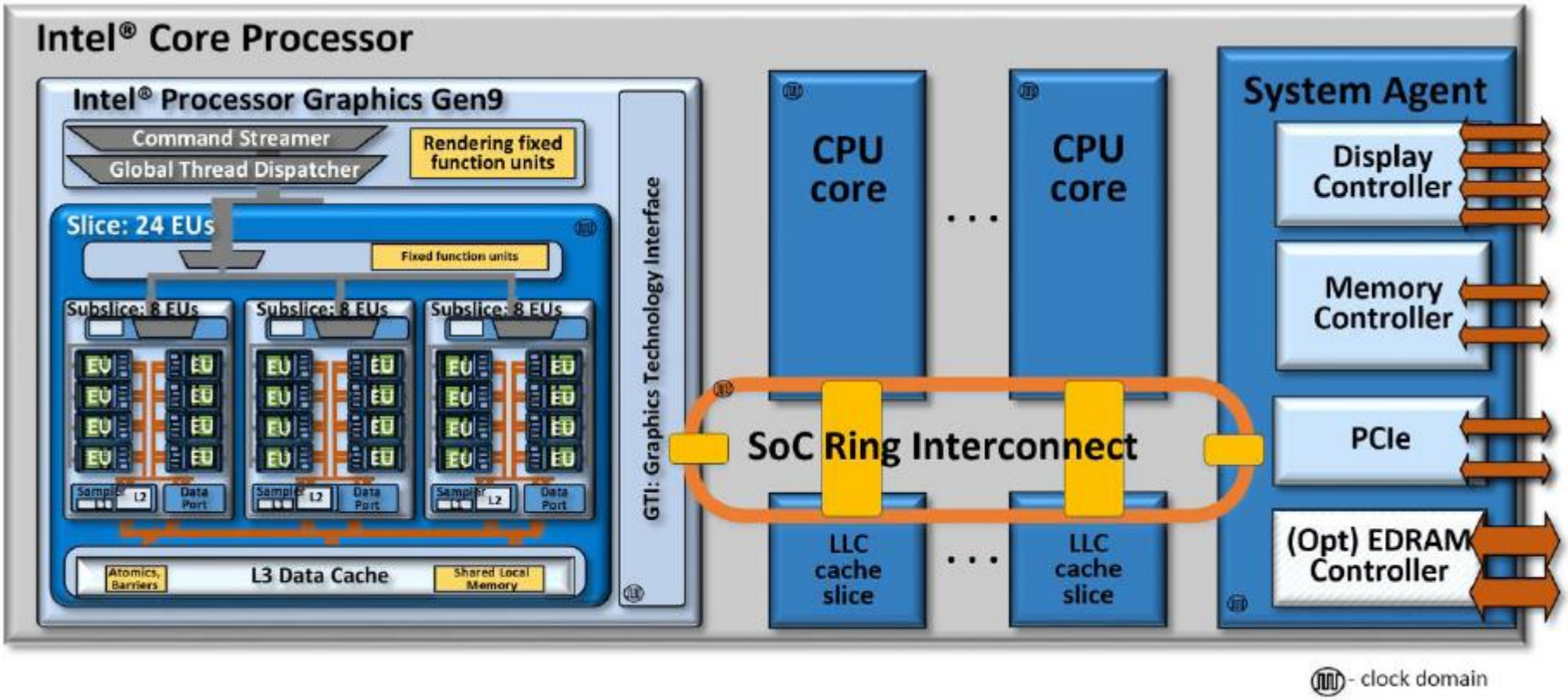
CU: Compute Unit

And now … Intel

Intel's Tick-Tock Cadence					
Microarchitecture	Process Node	Tick or Tock	Release Year		
Conroe/Merom	65nm	Tock	2006		
Penryn	45nm	Tick	2007		
Nehalem	45nm	Tock	2008		
Westmere	32nm	Tick	2010		
Sandy Bridge	32nm	Tock	2011		
Ivy Bridge	22nm	Tick	2012		
Haswell	22nm	Tock	2013		
Broadwell	14nm	Tick	2014		
Skylake	14nm	Tock	2015		
Kaby Lake	14nm	Tock	2016		



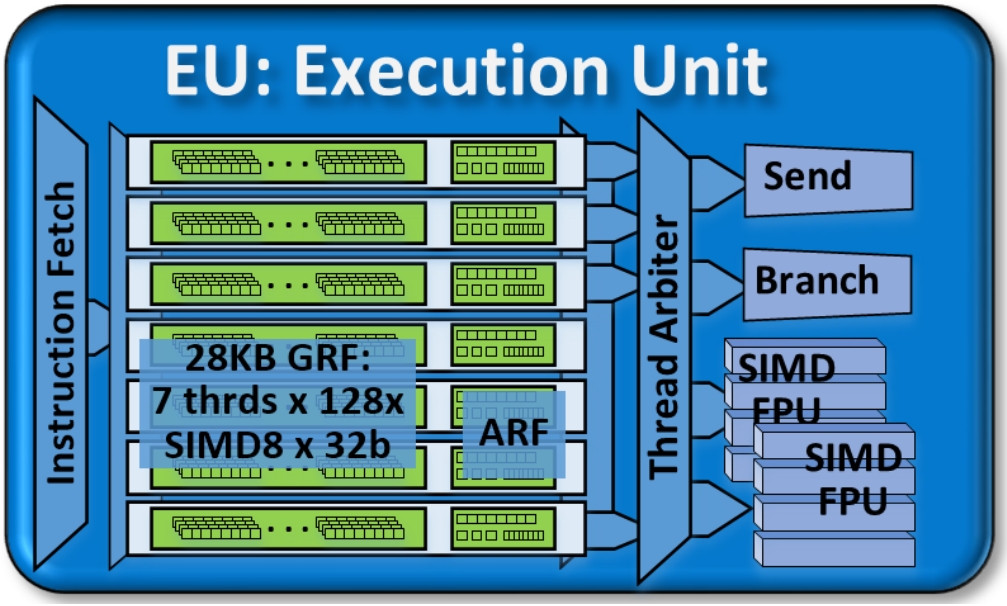
Source: The Compute Architecture of Intel® Processor Graphics Gen9 Whitepaper, August 2015



Source: The Compute Architecture of Intel® Processor Graphics Gen9 Whitepaper, August 2015

Intel Gen9 Embedded GPU

- Compute components called execution units (EU).
- Execution units are clustered into subslices.
- Subslices are gathered into slices.



GRF: general purpose register file

ARF: Architecture register file

FPU: Floating point unit
But supports both FP and integer ops.

EU

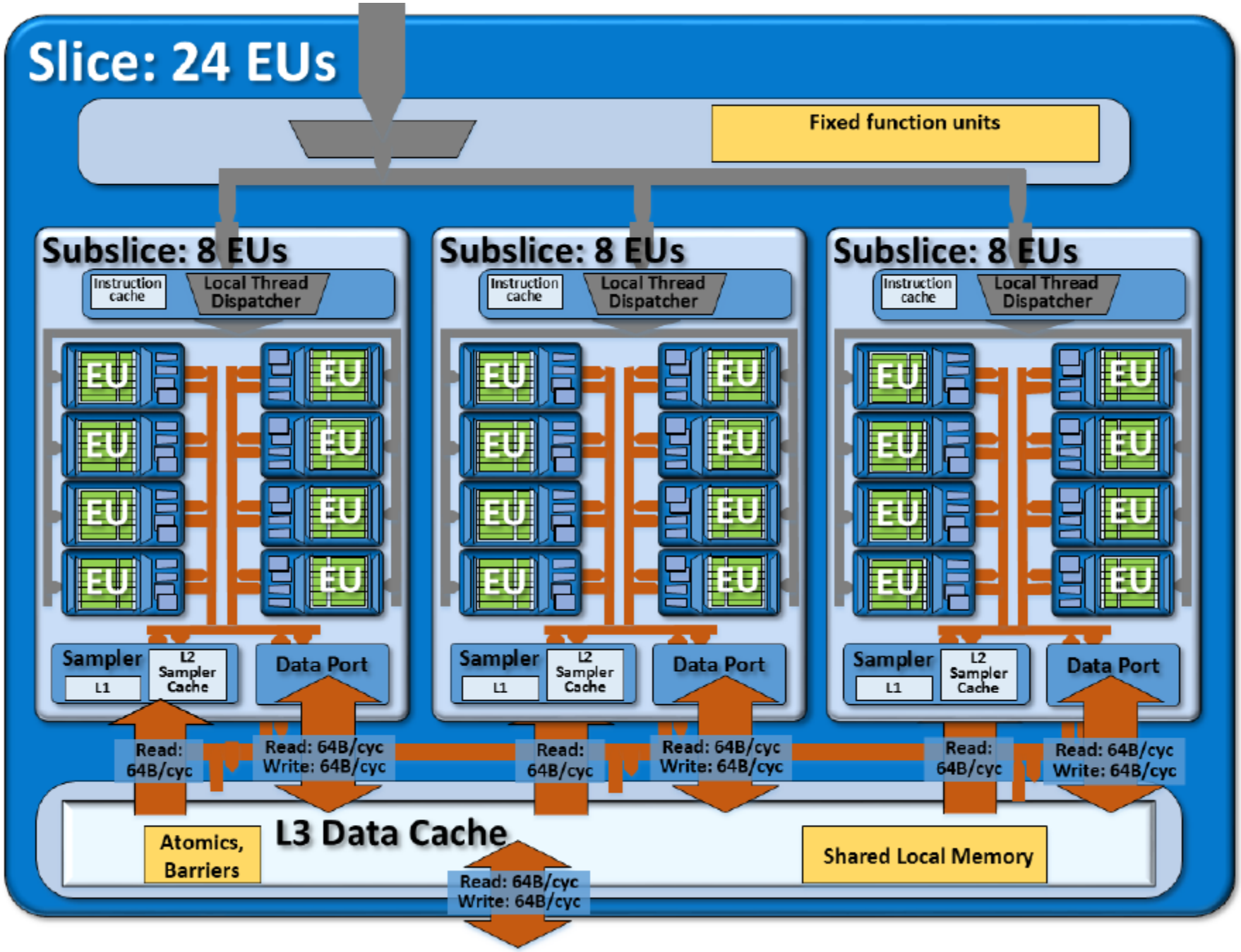
- SIMD ALUs
  - pipelined across multiple threads
- A combination of:
  - simultaneous multi-threading (SMT)
  - fine-grained interleaved multi-threading (IMT).
- For gen9-based products there are 7 threads
- Each EU thread:
  - has 128 general purpose registers
  - Each register stores 32 bytes
  - Accessible as a SIMD 8-element vector of 32-bit data elements.
  - Thus each gen9 thread has: 4 Kbytes of general purpose register file (GRF).

EU

- Depending on the software workload, the hardware threads within an EU may
  - all be executing the same compute kernel code
  - each EU thread could be executing code from a completely different compute kernel
- On every cycle:
  - an EU can co-issue up to four different instructions
  - must be sourced from four different threads.
  - The EU's thread arbiter dispatches these instructions to one of four functional units for execution
- Each FPU can SIMD:
  - execute up to four 32-bit floating-point (or integer) operations
  - or execute up to eight 16-bit integer or 16-bit floating-point operations.

Subslice

- Each subslice contains its own local-thread dispatcher
- The sampler is a read-only memory fetch unit.


Slice: 24 EUs
Fixed function units
Subslice: 8 EUs
Instruction cache
Local Thread Dispatcher
EU
Sampler
L1
L2 Sampler Cache
Data Port
Subslice: 8 EUs
Instruction cache
Local Thread Dispatcher
EU
Sampler
L1
L2 Sampler Cache
Data Port
Subslice: 8 EUs
Instruction cache
Local Thread Dispatcher
EU
Sampler
L1
L2 Sampler Cache
Data Port
Read: 64B/cyc
Read: 64B/cyc Write: 64B/cyc
Read: 64B/cyc
Read: 64B/cyc Write: 64B/cyc
Read: 64B/cyc
Read: 64B/cyc Write: 64B/cyc
Atomics, Barriers
L3 Data Cache
Shared Local Memory
Read: 64B/cyc Write: 64B/cyc

Slice

- Shared local memory:
  - a structure within the L3 complex
  - supports programmer-managed data
  - for sharing among EU hardware threads within the same subslice

Conclusions

- The design of state-of-the art GPUs includes:
  - Data parallelism
  - Programmability
  - Much less restrictive instruction set
- By looking at the hardware features, can you see how can you write more efficient programs for GPUs?