GPU Computing

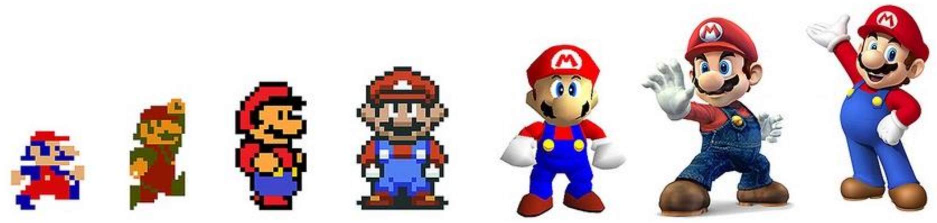
Lecture 2

Young-Ho Gong

Contents

Computer Graphics

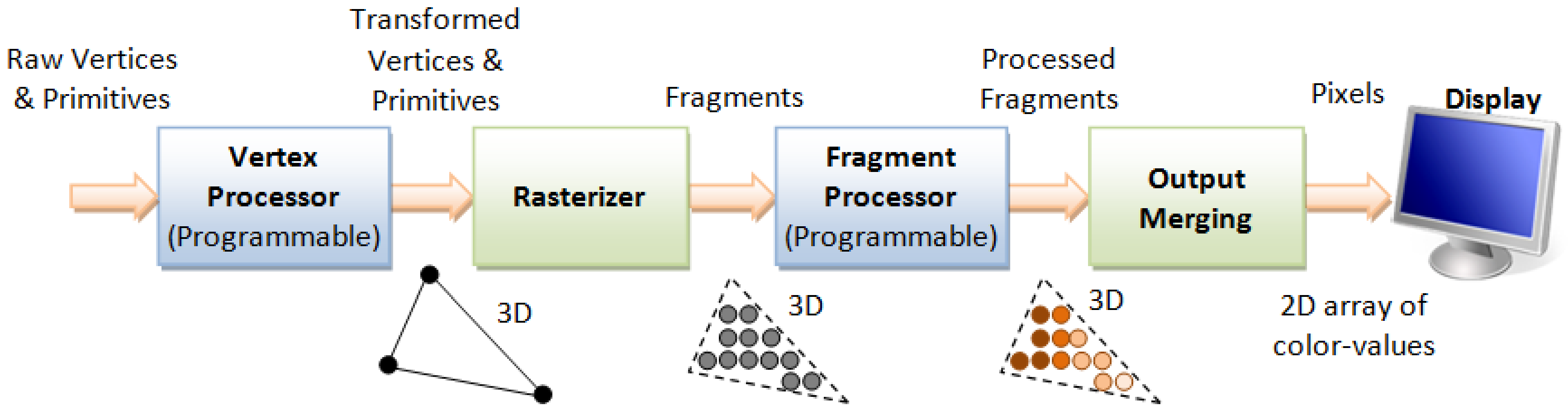
- Make great images
  - Intricate shapes
  - Complex optical effects
  - Seamless motion



- Make them fast
  - Invent clever techniques
  - Use every trick imaginable
  - ***Build monster hardware***
- We use the monster hardware as computing devices

Graphics Pipeline

- Sequence of operations
  - take the vertices and textures of your meshes all the way to the pixels in the render targets


Raw Vertices & Primitives
Transformed Vertices & Primitives
Fragments
Processed Fragments
Pixels
Display
Vertex Processor (Programmable)
Rasterizer
Fragment Processor (Programmable)
Output Merging
3D
3D
3D
2D array of color-values

3D Graphics Rendering Pipeline: Output of one stage is fed as input of the next stage. A vertex has attributes such as (x, y, z) position, color (RGB or RGBA), vertex-normal (n_x, n_y, n_z), and texture. A primitive is made up of one or more vertices. The rasterizer raster-scans each primitive to produce a set of grid-aligned fragments, by interpolating the vertices.

3D Graphics Pipeline

Vertex Transform & Lighting

1

↓

Triangle Setup & Rasterization

↓

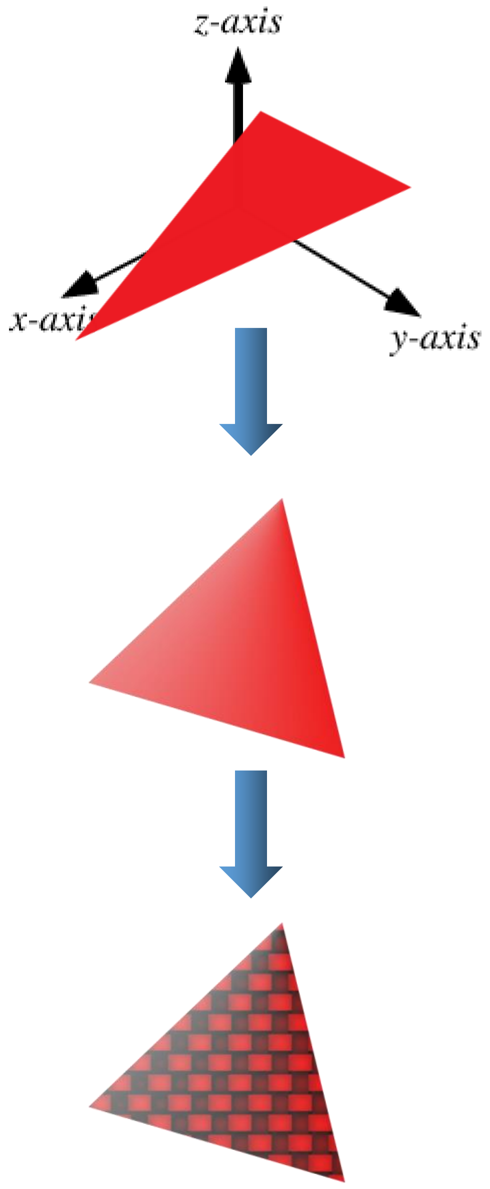
Texturing & Pixel Shading

↓

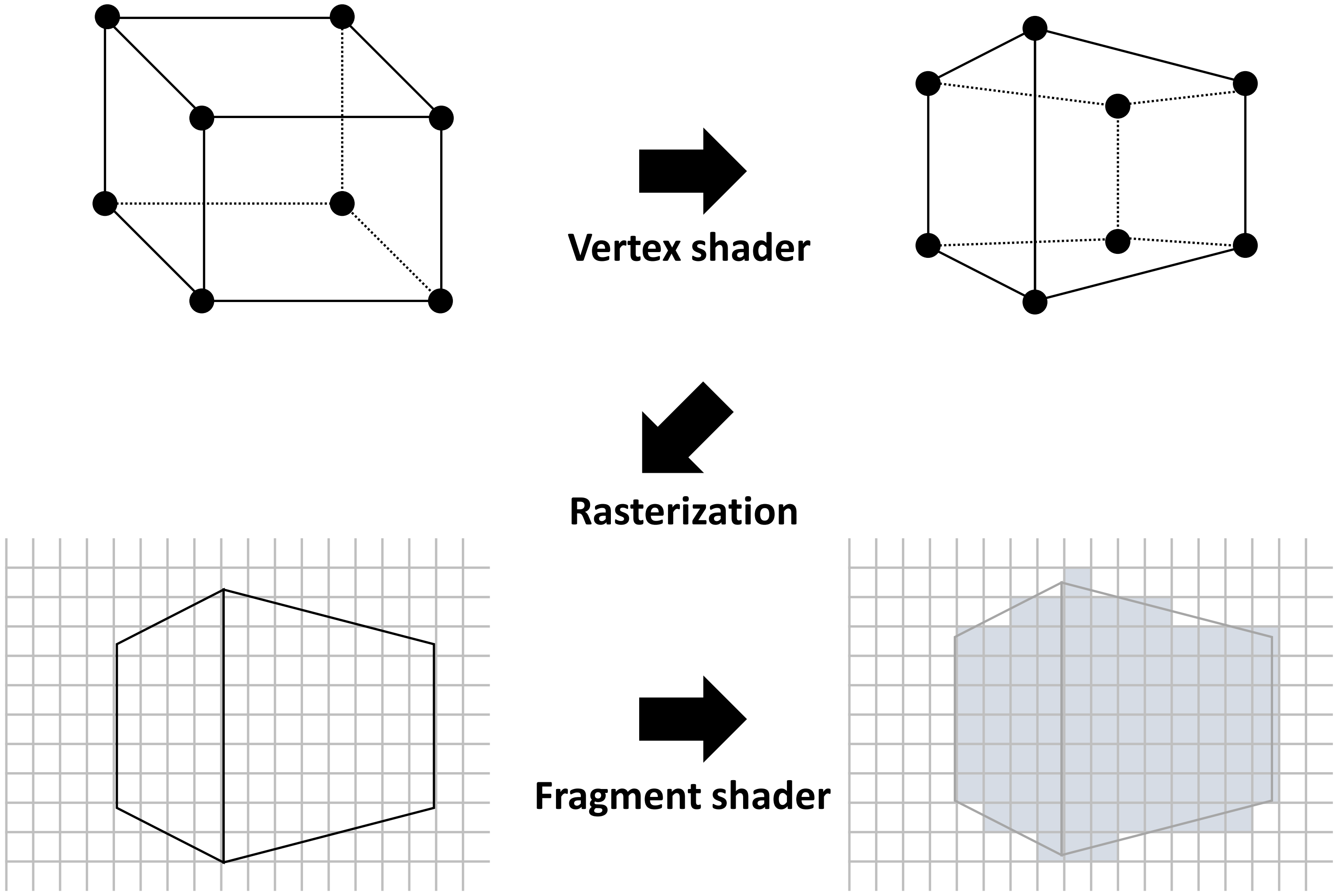
Depth Test & Blending

↓

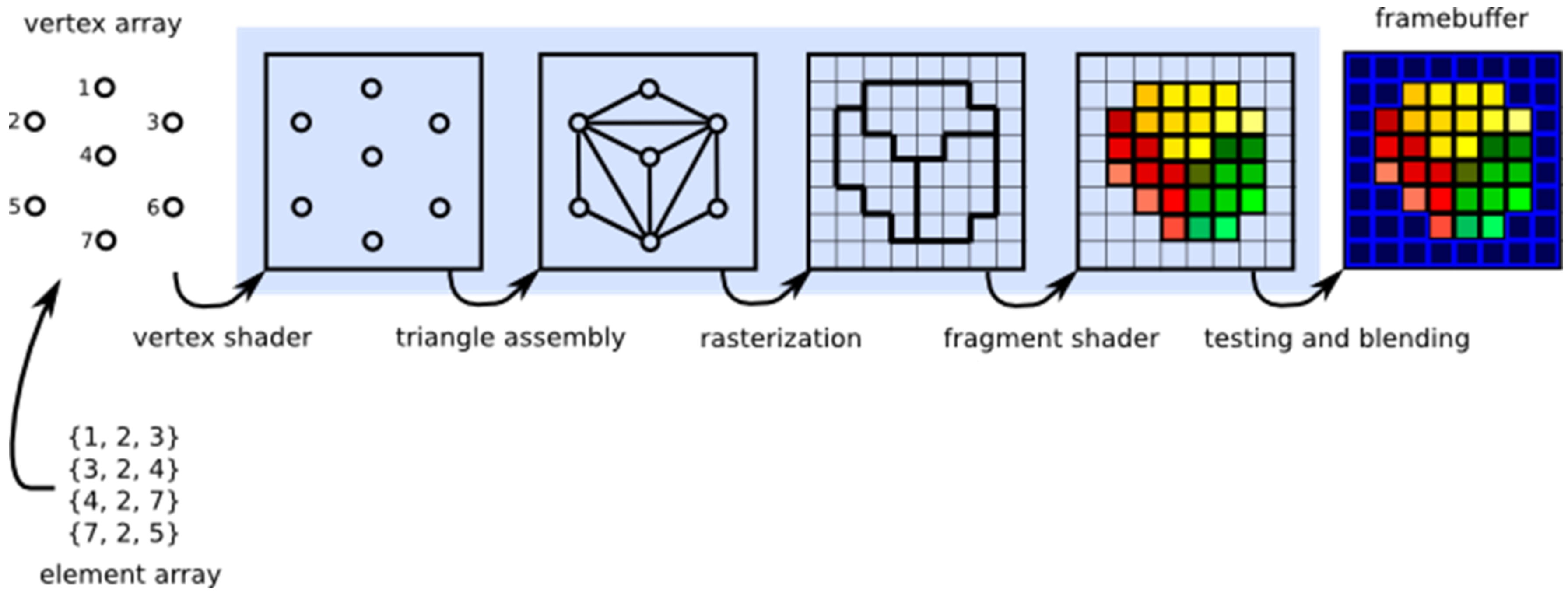
Framebuffer



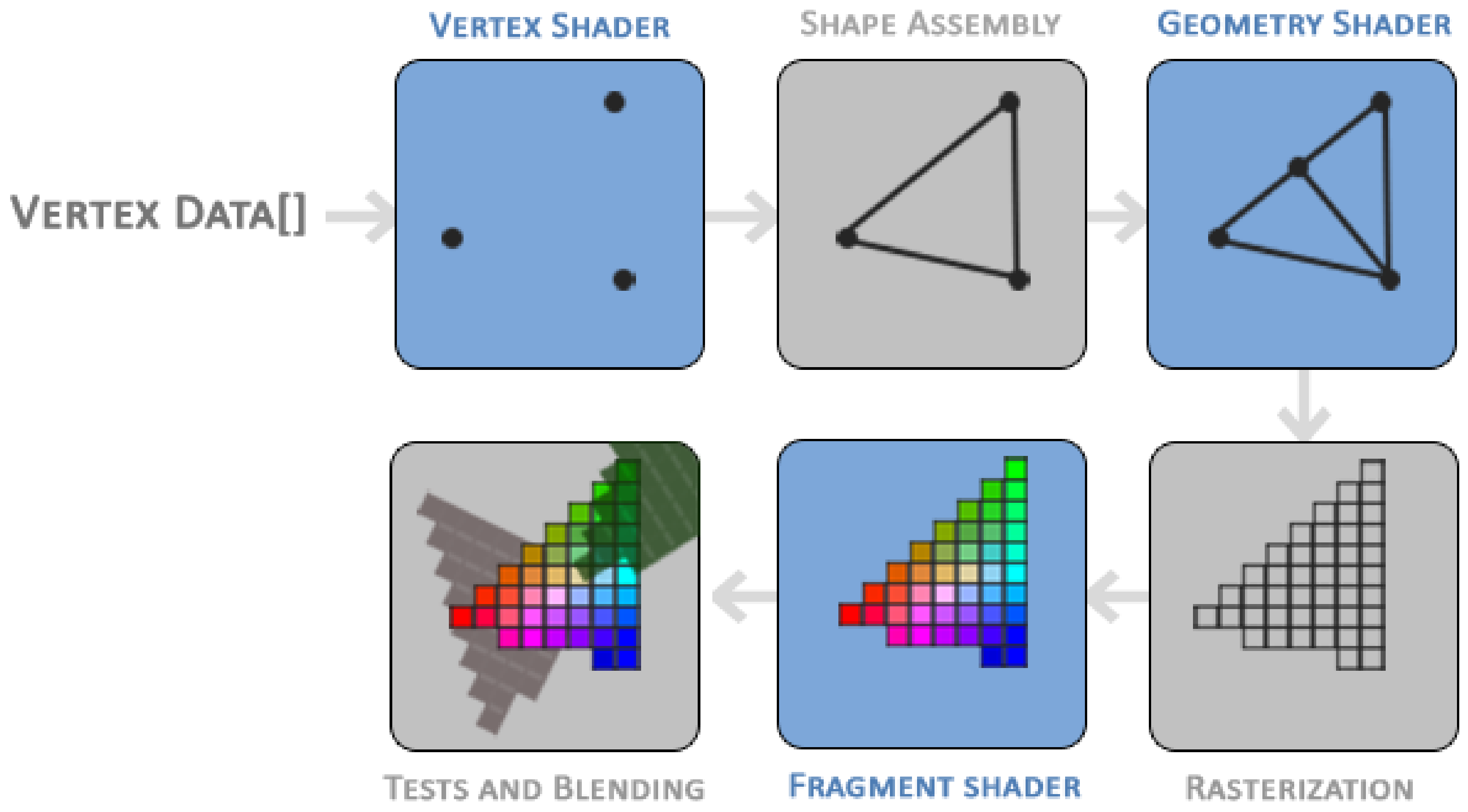
3D Graphics Pipeline



3D Graphics Pipeline

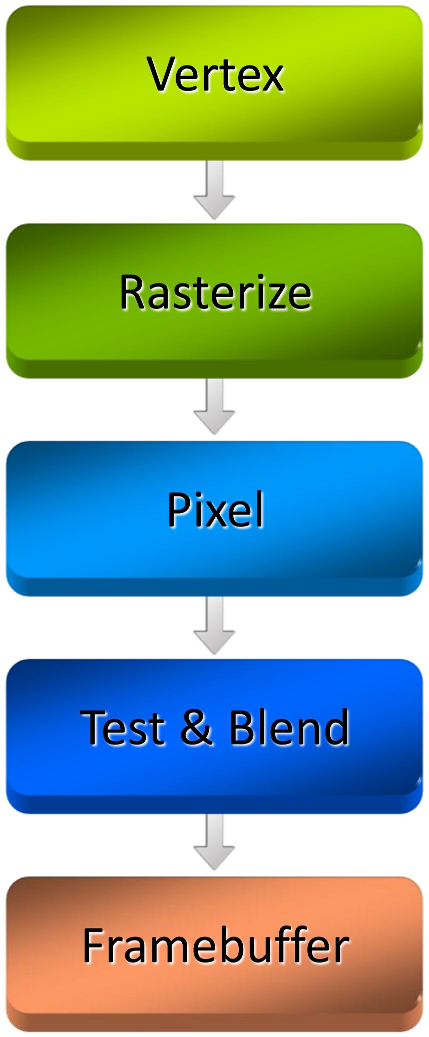


3D Graphics Pipeline



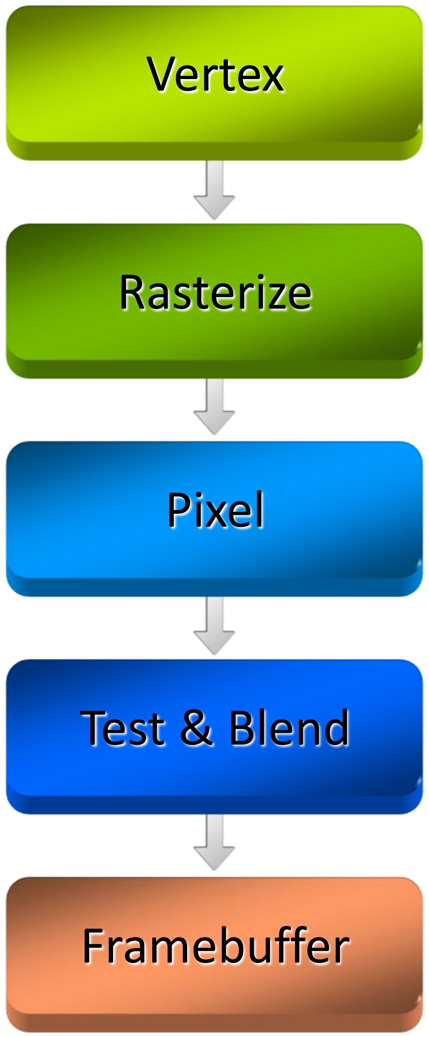
The Graphics Pipeline

- Key abstraction of real-time graphics
- One chip/board per stage
- Fixed data flow through pipeline



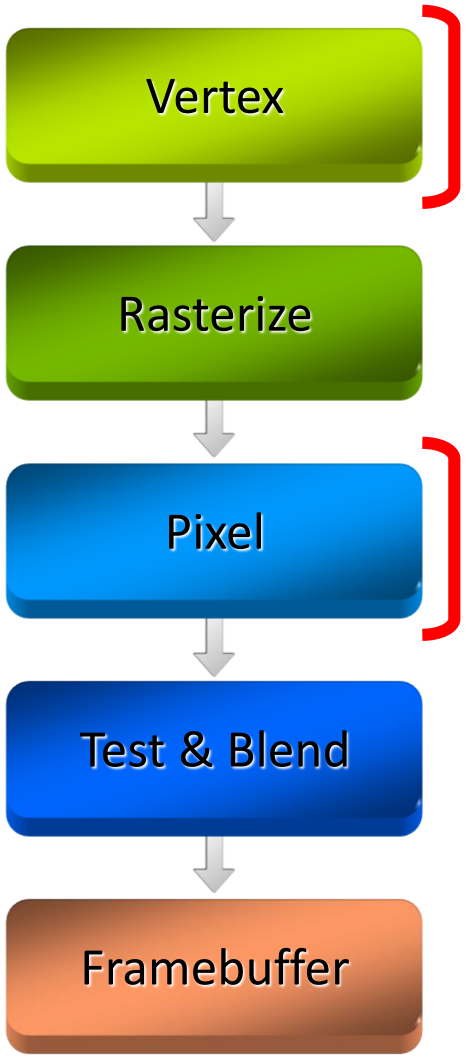
The Graphics Pipeline – 2nd gen

- Everything fixed function, with a certain number of modes
- Number of modes for each stage grew over time
- Hard to optimize HW
- Developers always wanted more flexibility



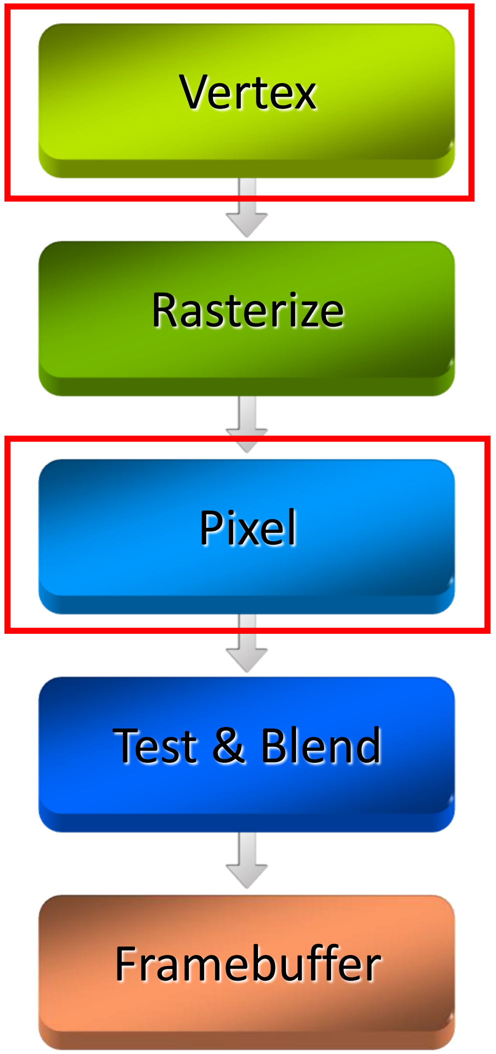
The Graphics Pipeline – 3rd gen

- Remains a key abstraction
- Vertex & pixel processing became programmable, new stages added
- GPU architecture increasingly centers around shader execution



The Graphics Pipeline – 4th gen

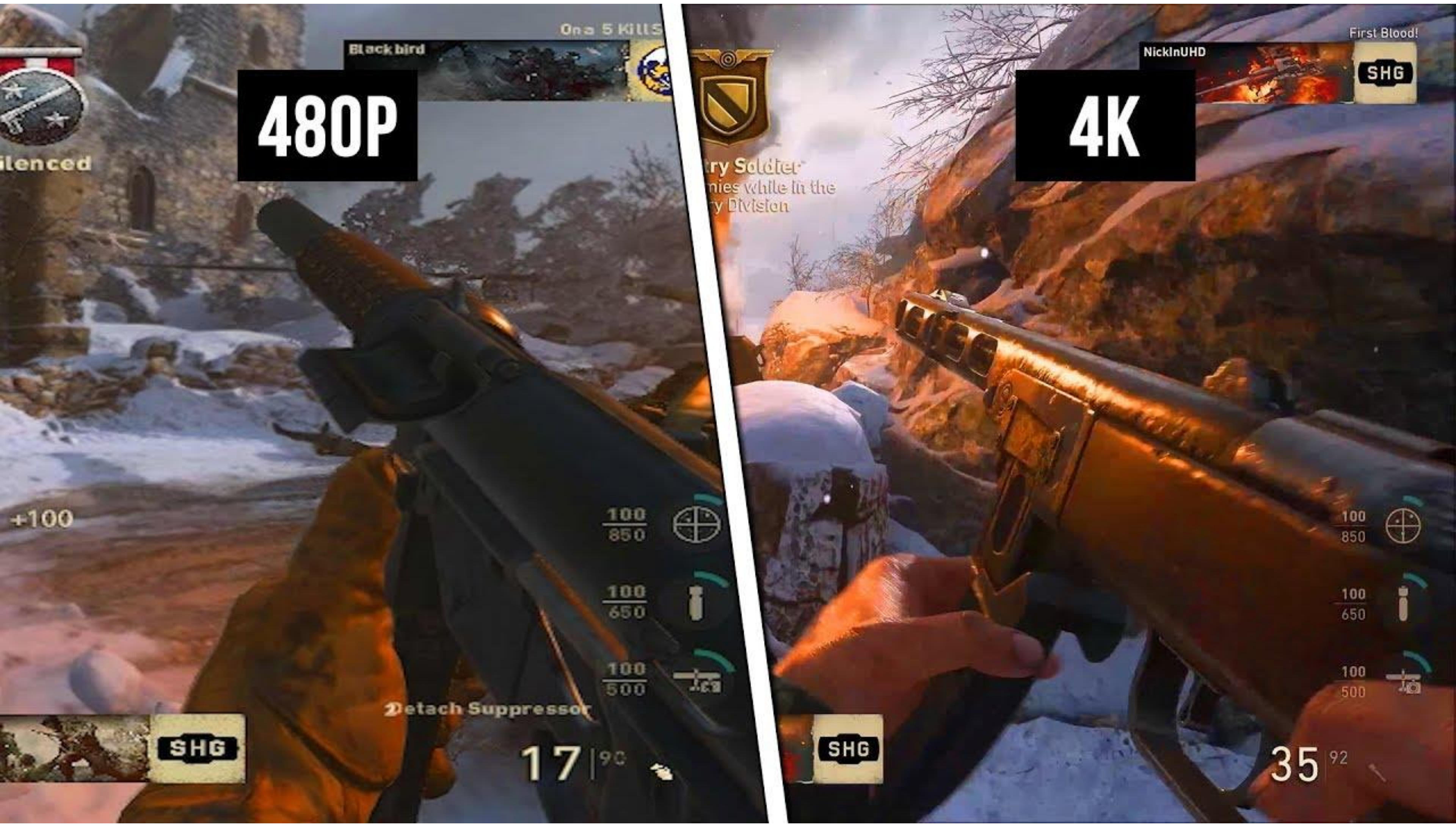
- Exposing a (at first limited) instruction set for some stages
- Expanded to full ISA
  - CUDA
- Limited instructions & instruction types and no control flow at first
- Optimization?



Why GPUs scale so nicely

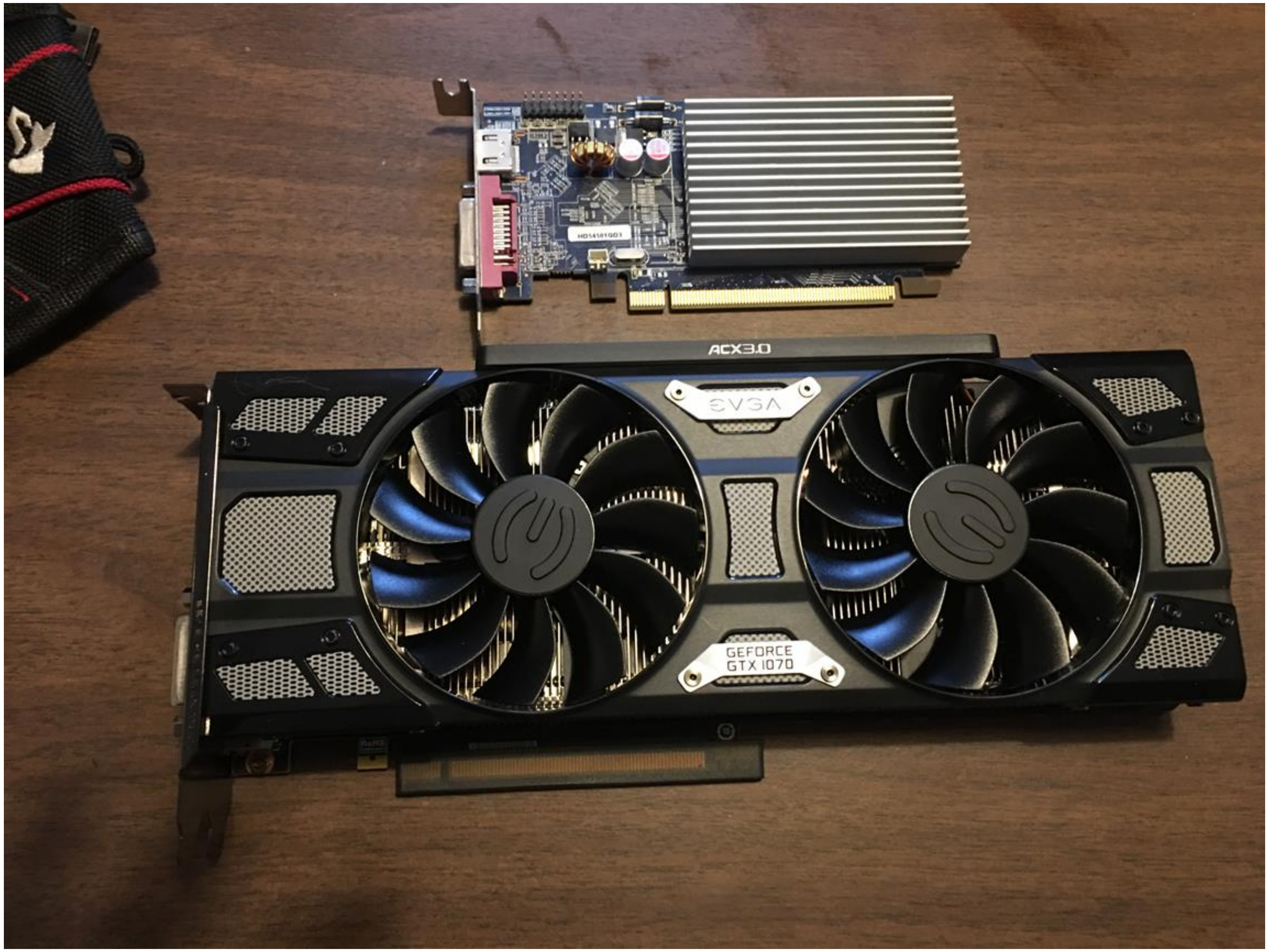
- Workload and Programming Model provide lots of parallelism
- Applications provide large groups of vertices at once
  - Vertices can be processed in parallel
  - Apply same transform to all vertices
- Triangles contain many pixels
  - Pixels from a triangle can be processed in parallel
  - Apply same shader to all pixels
- Very efficient hardware to hide serialization bottlenecks

Why GPUs scale so nicely



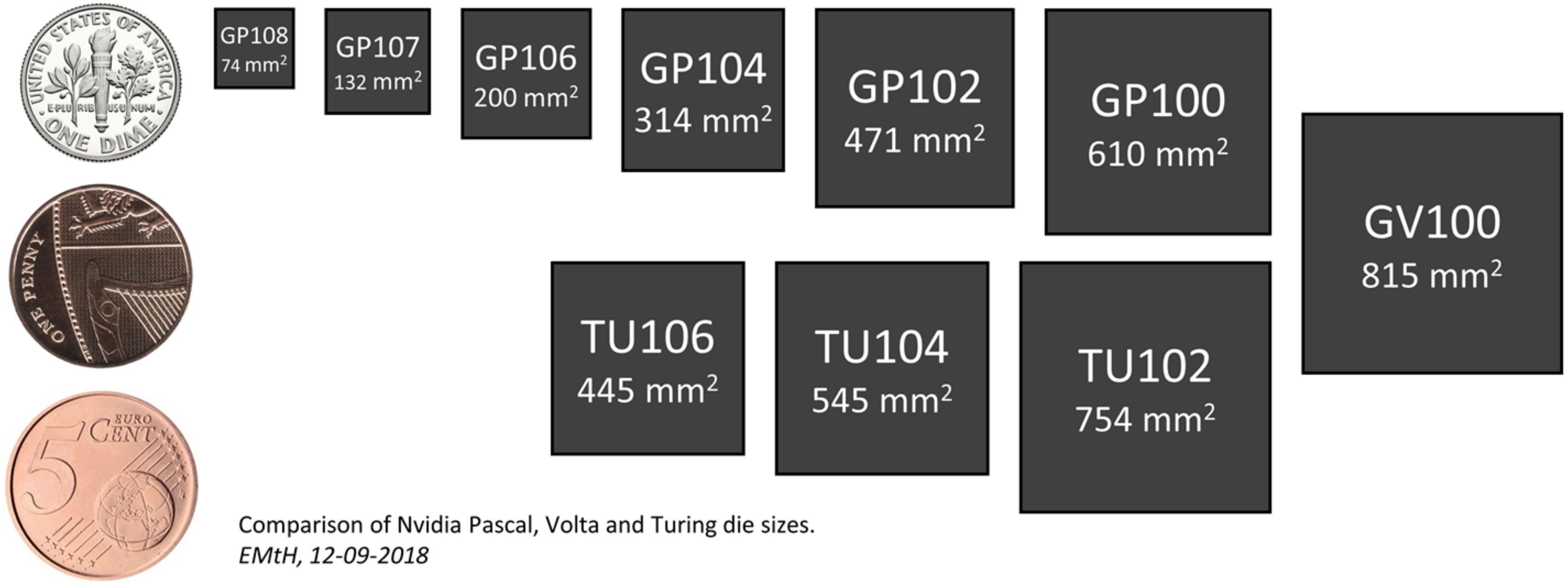
With Moore's Law...

- Old vs. New GPU



With Moore's Law...

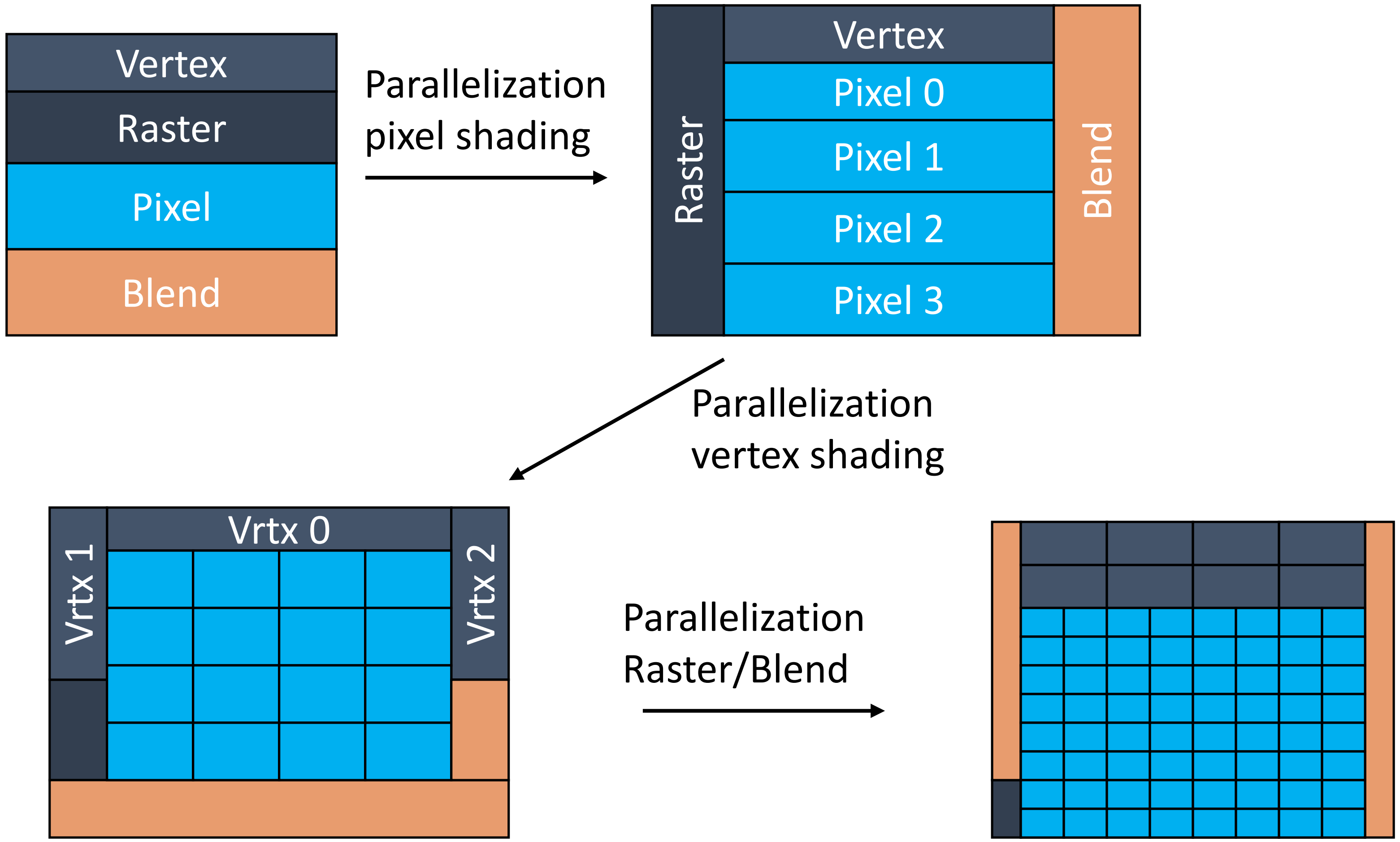
- Low-end → High-end GPU



Comparison of Nvidia Pascal, Volta and Turing die sizes.
EMtH, 12-09-2018

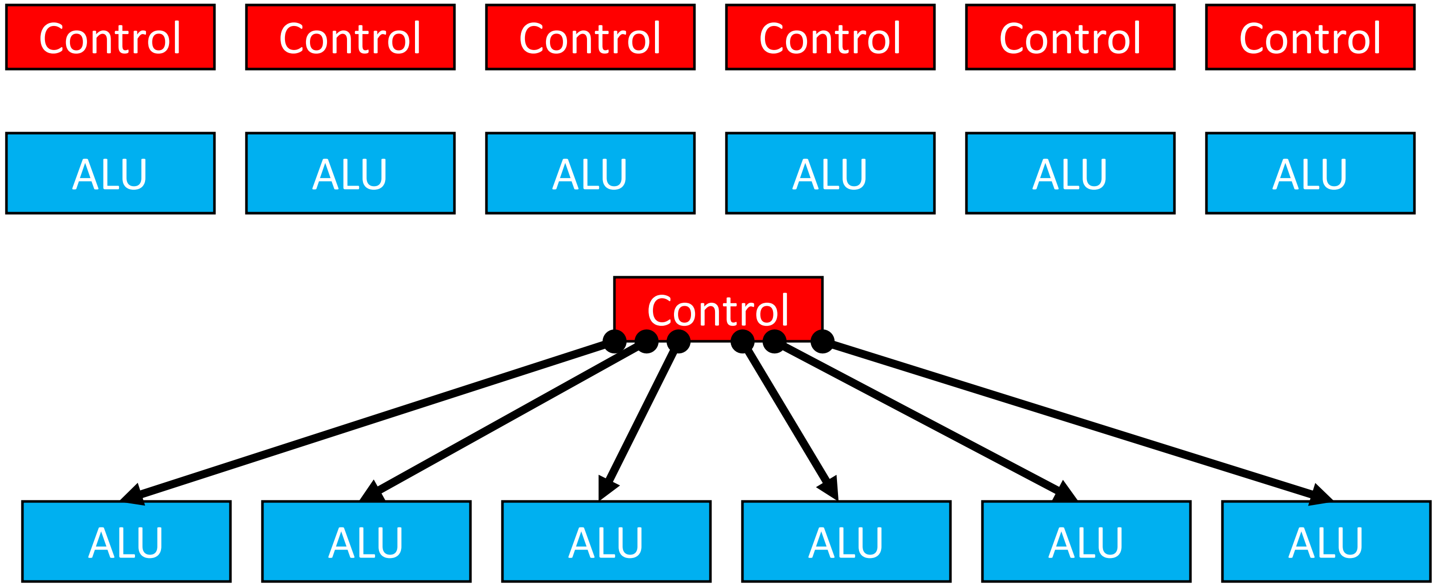
With Moore's Law...

- We get more and more transistors !



More Efficiency

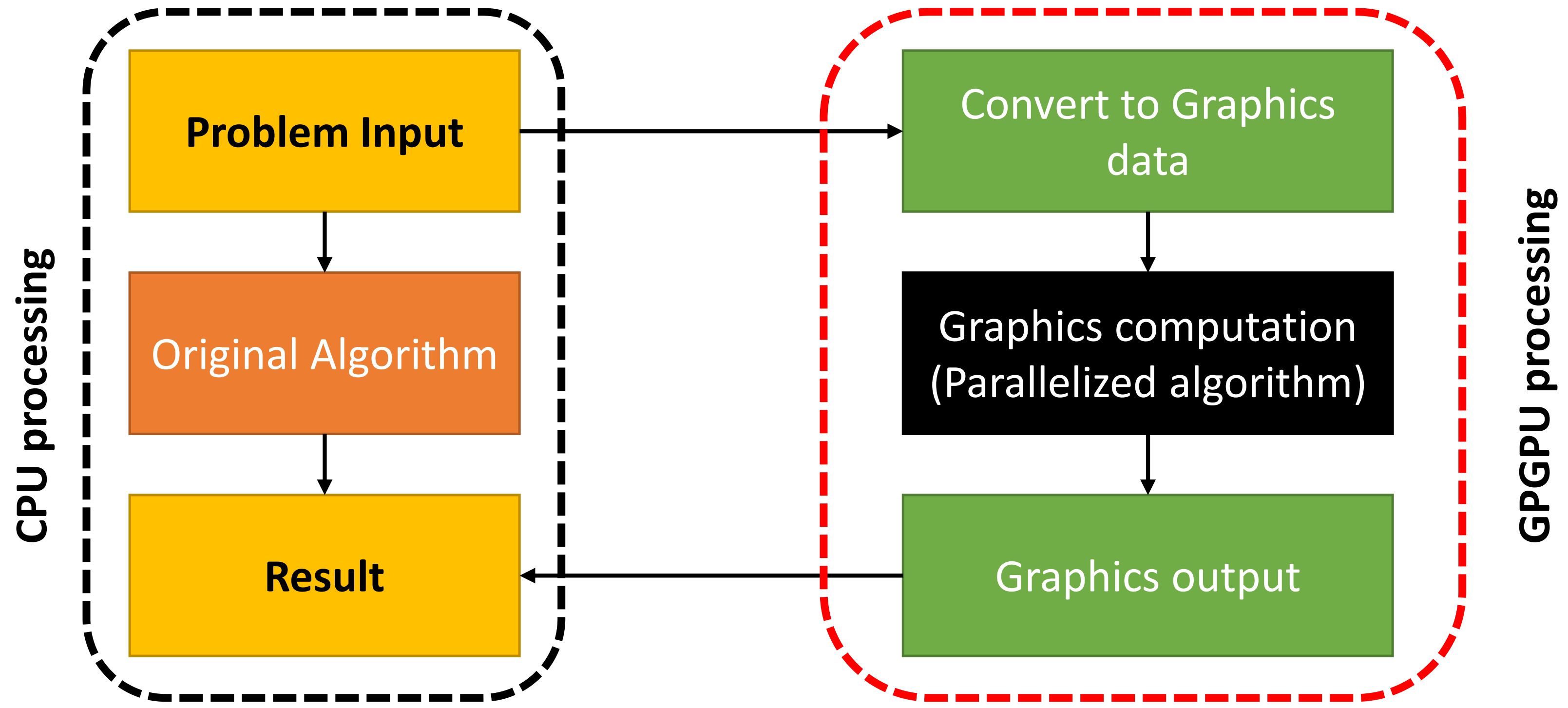
- Note that we do the same thing for lots of pixels/vertices
  - In a CPU manner (Single control unit for each ALU)



- A warp = 32 threads launched together
  - Usually, execute together as well

Early GPGPU

- GPGPU: General Purpose Graphics Processing Unit



- Very tedious, limited usability
- Still had some very nice results (Performance)
- This was the lead up to CUDA

CUDA

- Compute Unified Device Architecture
- General purpose programming model
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel co-processor
- Targeted software stack
  - Compute oriented drivers, language, and tools
- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Interface designed for compute - graphics free API
  - Data sharing with OpenGL buffer objects
  - Guaranteed maximum download & readback speeds
  - Explicit GPU memory management

CUDA

- Compute Unified Device Architecture

GPU Computing Applications

Libraries and Middleware

cuDNN TensorRT	cuFFT, cuBLAS, cuRAND, cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL, SVM, OpenCurrent	PhysX, OptiX, iRay	MATLAB Mathematica

Programming Languages

C	C++	Fortran	Java, Python, Wrappers	DirectCompute	Directives (e.g., OpenACC)

CUDA-enabled NVIDIA GPUs

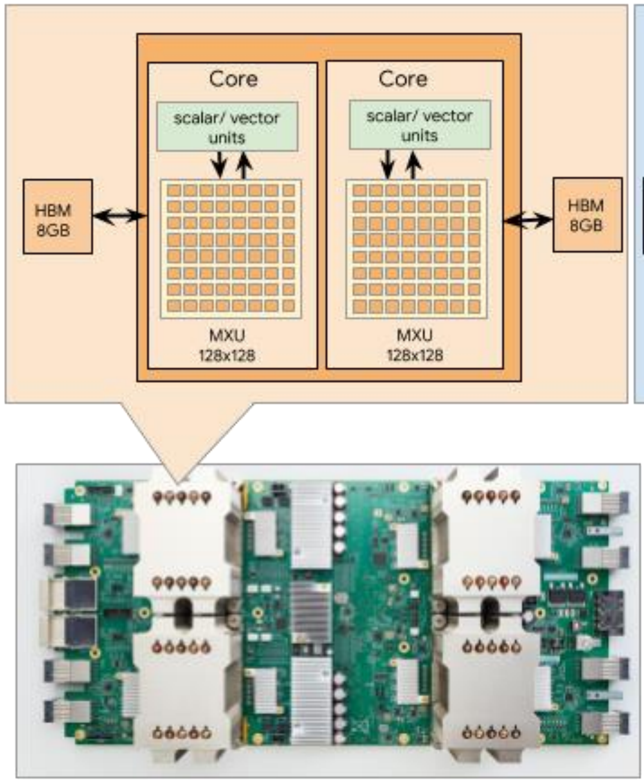
Turing Architecture (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier	GeForce 2000 Series	Quadro RTX Series	Tesla T Series
Volta Architecture (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier			Tesla V Series
Pascal Architecture (Compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series
Maxwell Architecture (Compute capabilities 5.x)	Tegra X1	GeForce 900 Series	Quadro M Series	Tesla M Series
Kepler Architecture (Compute capabilities 3.x)	Tegra K1	GeForce 700 Series GeForce 600 Series	Quadro K Series	Tesla K Series
	EMBEDDED	CONSUMER DESKTOP, LAPTOP	PROFESSIONAL WORKSTATION	DATA CENTER

Other Architectures

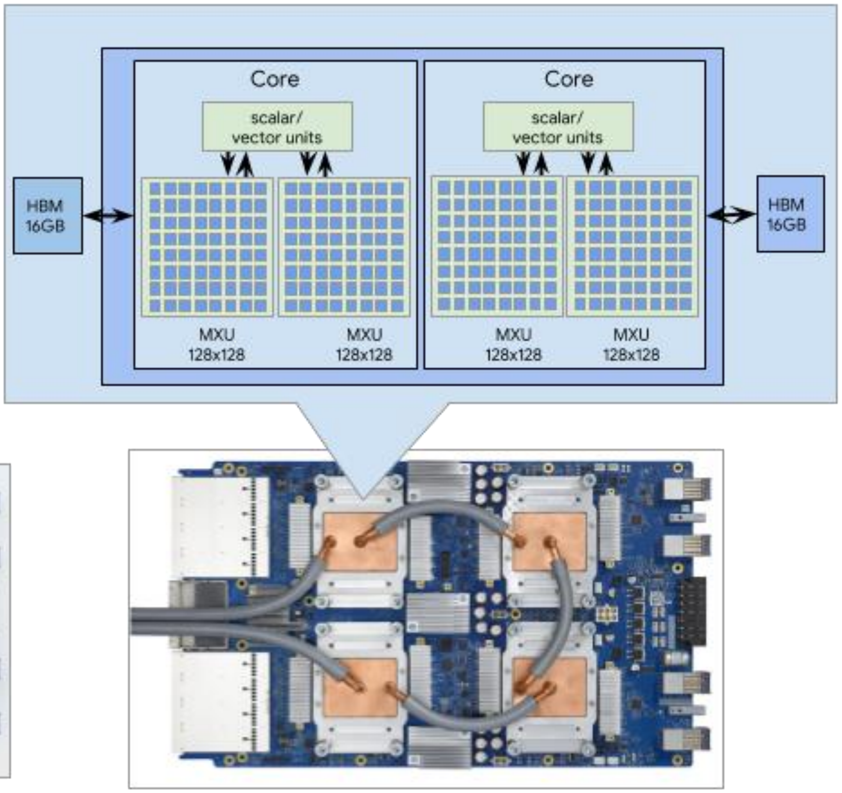
- IBM Watson Architecture
  - Traditional design
  - 90 * IBM Power 750 system (3.5GHz IBM Power7 8-core CPU)
  - Each core can execute 4 threads → 90 * 8 * 4 = 2,880 threads totally
  - 2,880 threads + 16 TB RAM
- It works as a question answering computing system
  - w/ Deep learning technology

Other Architectures

- AlphaGo Architecture
  - Deep Learning → so much computations
  - Cloud computing required → scalable architecture
  - Oct 2015: 1202 CPU + 176 GPU (CUDA) – fake
  - Mar 2016: 1920 CPU + 280 GPU (CUDA) – fake
  - May 2016: AlphaGo uses **TPU !**



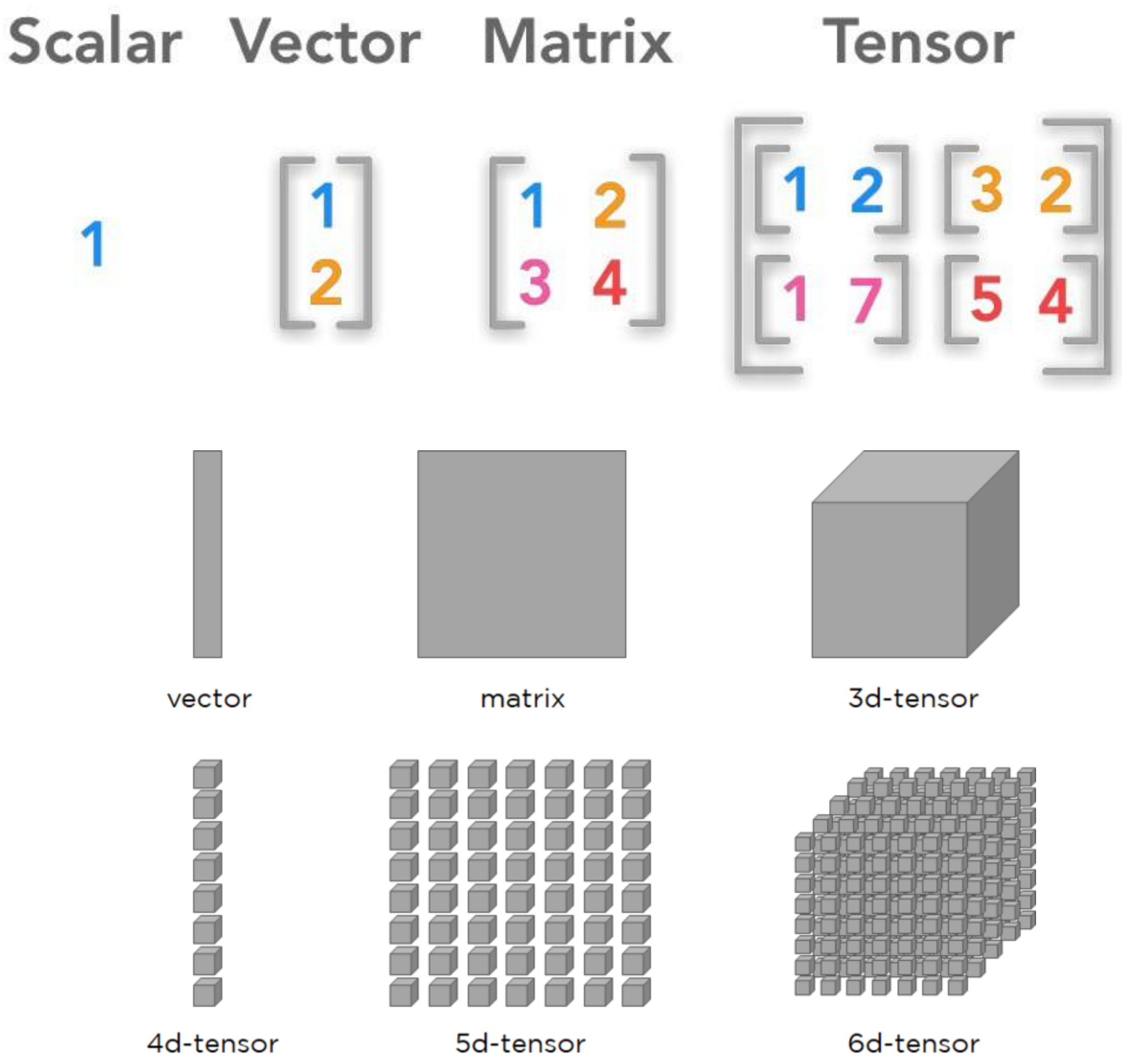
TPU v2 - 4 chips, 2 cores per chip



TPU v3 - 4 chips, 2 cores per chip

Other Architectures

- Al
  -
  -
  -
  -
  -



$$1 \quad \begin{bmatrix}1\\2\end{bmatrix} \quad \begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} \quad \begin{bmatrix}\begin{bmatrix}1 & 2\end{bmatrix} & \begin{bmatrix}3 & 2\end{bmatrix}\\\begin{bmatrix}1 & 7\end{bmatrix} & \begin{bmatrix}5 & 4\end{bmatrix}\end{bmatrix}$$

Intermediate Summary

- Graphics and Parallel Computing History
- 3D Graphics Pipeline
- HW Development
- Early GPGPU
- CUDA
- Other Parallel Architectures

Next step

- CUDA Install in your computer

- If your desktop/laptop has an Nvidia GPU
  - Install Visual Studio 2019
  - Install latest CUDA SDK
  - You can start CUDA programming by generating new CUDA project

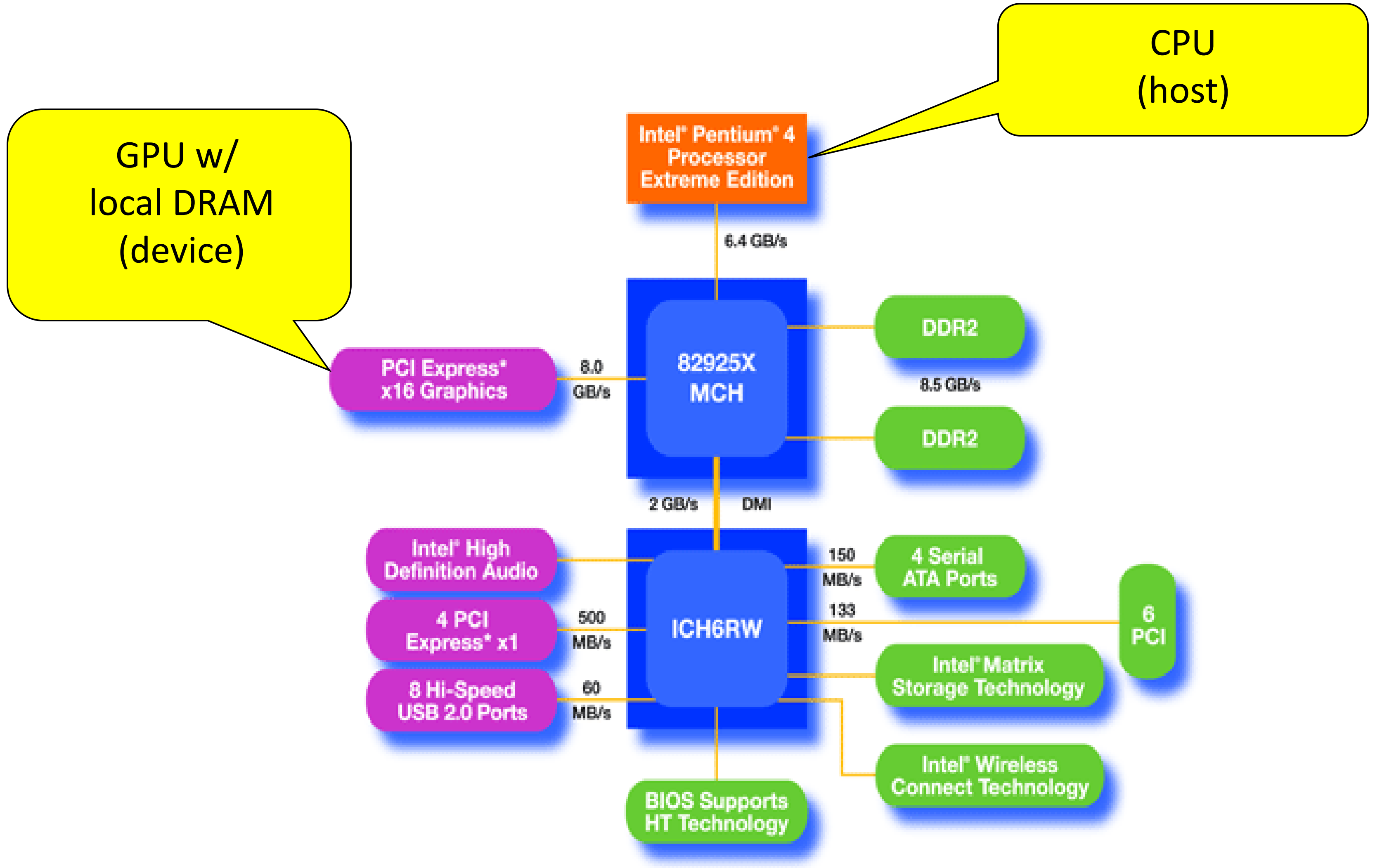
- Else,
  - You can start CUDA Programming using Colab by Google

Next step

- Very simple CUDA Program
- Simple CUDA model
  - Host, device
- CUDA memory copy scenario
- CPU memory copy functions
- CUDA memory copy functions
- Host-device memory copy functions

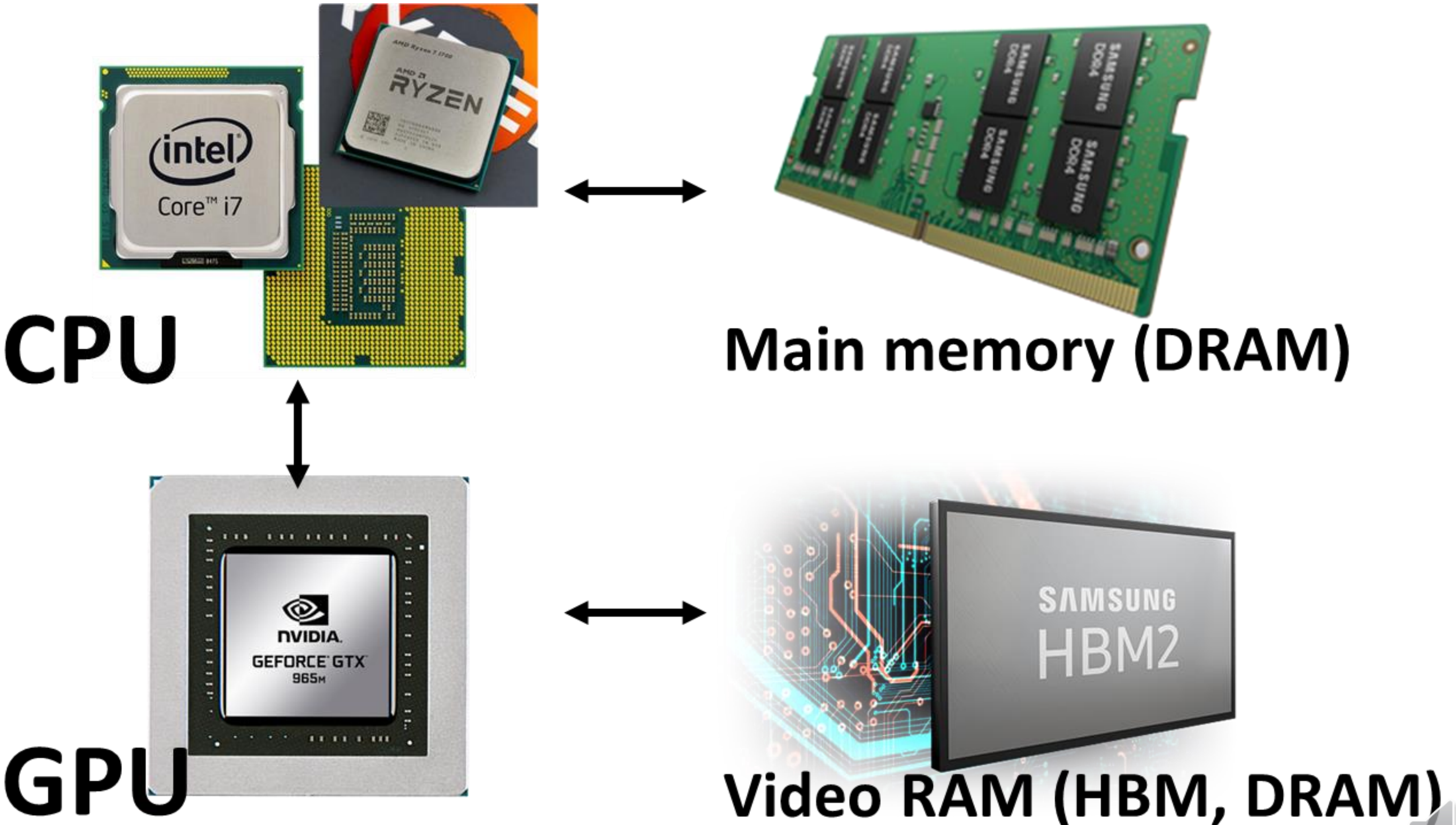
An Example of Physical Reality behind CUDA

- Old but basic architecture


CPU (host)
GPU w/ local DRAM (device)
Intel® Pentium® 4 Processor Extreme Edition
6.4 GB/s
82925X MCH
PCI Express* x16 Graphics
8.0 GB/s
DDR2
8.5 GB/s
DDR2
2 GB/s
DMI
ICH6RW
Intel® High Definition Audio
4 PCI Express* x1
500 MB/s
8 Hi-Speed USB 2.0 Ports
60 MB/s
150 MB/s
4 Serial ATA Ports
133 MB/s
6 PCI
Intel® Matrix Storage Technology
Intel® Wireless Connect Technology
BIOS Supports HT Technology

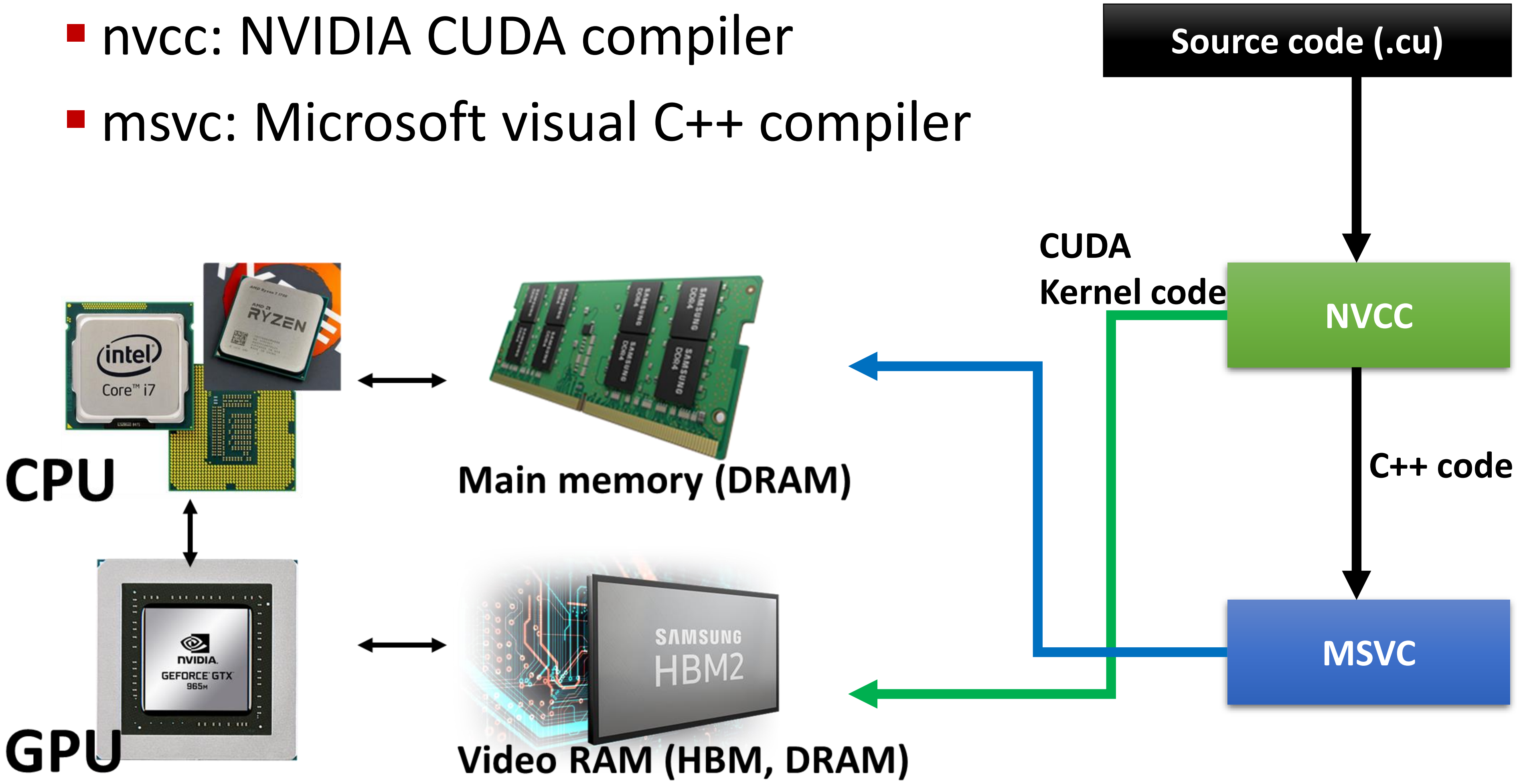
Very simple CUDA model

- Host: CPU + main memory
- Device: GPU + VRAM



Very simple CUDA model

- nvcc: NVIDIA CUDA compiler
- msvc: Microsoft visual C++ compiler

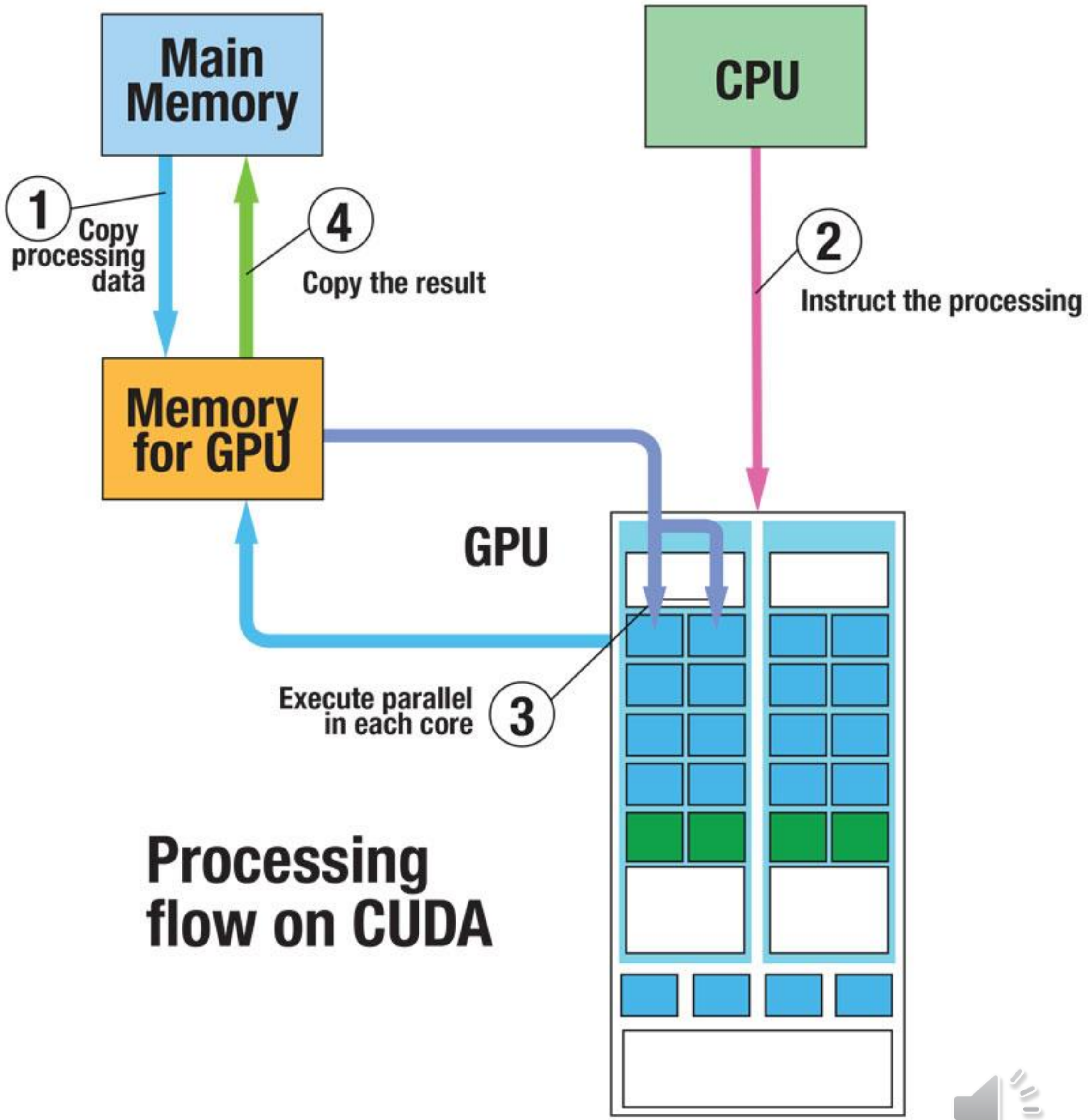


Simple CUDA Program Scenario

- Step 1. Serial code (Host)
  - Serial execution: **read data**
  - Prepare parallel execution
  - Copy data from main memory to VRAM
- Step 2. Kernel code (Device)
  - **Parallel Processing**
  - Read/Write data from VRAM to VRAM
  - **CUDA do NOT have any direct I/O features**
- Step 3. Serial code (Host)
  - Copy data from VRAM to main memory
  - Serial execution: **print data**

Simple CUDA Program Scenario

- Step 1. Serial code (Host)
  - Serial execution: **read data**
  - Prepare parallel execution
  - Copy data from main memory to VRAM
- Step 2. Kernel code (Device)
  - **Parallel Processing**
  - Read/Write data from VRAM to VRAM
  - **CUDA do NOT have any direct I/O feature**
- Step 3. Serial code (Host)
  - Copy data from VRAM to main memory
  - Serial execution: **print data**



Memory Spaces

- CPU and GPU have separate memory spaces
  - Data is moved across data bus (e.g., PCIe)
  - Use functions to allocate/set/copy memory on GPU
    - Very similar to corresponding C functions
- Pointers are just addresses
  - Can’t tell from the pointer value whether the address is on CPU or GPU
  - Must exercise care when dereferencing:
  - Dereferencing CPU pointer on GPU will likely crash
    - Same for vice versa

CPU Memory Allocation / Release

- Host (CPU) manages host (CPU) memory:
  - void* malloc (size_t nbytes)
  - void* memset (void * pointer, int value, size_t count)
  - void free (void* pointer)

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int * ptr = 0;
ptr = malloc(nbytes);
memset( ptr, 0, nbytes);
free(ptr);
```

GPU Memory Allocation / Release

- Host (CPU) manages device (GPU) memory:
  - cudaMalloc (void ** pointer, size_t nbytes)
  - cudaMemset (void * pointer, int value, size_t count)
  - cudaFree (void* pointer)

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int * dev_a = 0;
cudaMalloc( (void**)&dev_a,  nbytes );
cudaMemset( dev_a, 0, nbytes);
cudaFree(dev_a);
```

CUDA function rules

- Every library function starts with “cuda”
- Most of them returns error code (or cudaSuccess).
  - cudaError_t cudaMalloc(void** devPtr, size_t size);
  - cudaError_t cudaFree(void* devPtr);
  - cudaError_t cudaMemcpy(void* dst, const void* src, size_t size);
- Example:

  -
    ```
    if (cudaMalloc(&devPtr, SIZE) != cudaSuccess){
        exit(1);
    }
    ```

Data copies

- cudaError_t cudaMemcpy (void* dst ,

```
                        void* src,
                        size_t nbytes,
                enum cudaMemcpyKind direction);
```

  - Returns after the copy is complete
  - Blocks CPU thread until all bytes have been copied
  - Doesn’t start copying until previous CUDA calls complete
- enum cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
  - cudaMemcpyHostToHost (added !)

Assignment #1

- Cuda Memcpy
  - Host to device
  - Device to host

Summary

- Very simple CUDA program
- Simple CUDA model
  - Host, Device
- CUDA memory copy scenario
- CPU memory copy functions
- CUDA memory copy functions
- Host-device memory copy example

Thank you

Any questions?

E-mail: yhgong@kw.ac.kr
Lab: https://sites.google.com/view/yhgong/