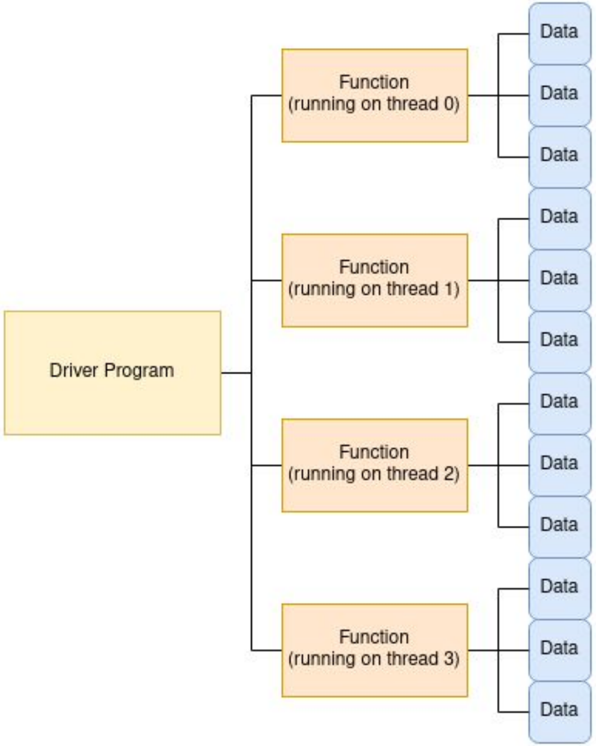
GPU Programming and the Graphics Pipeline

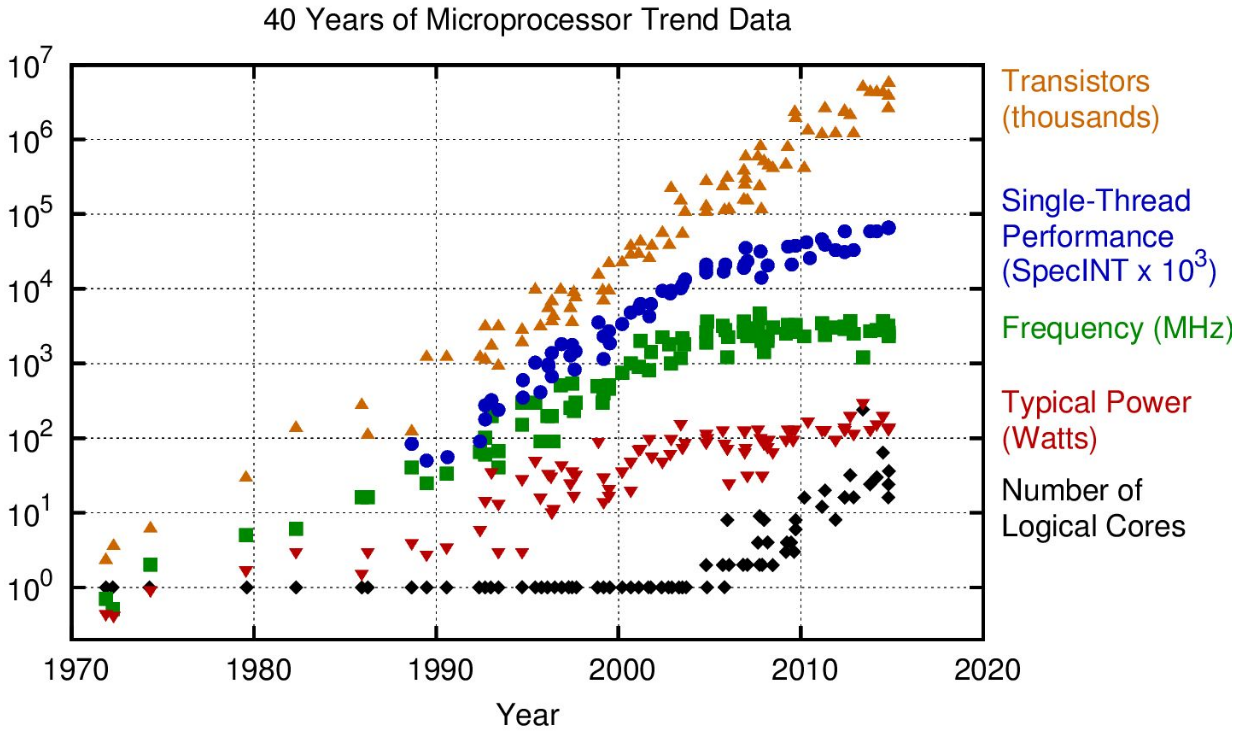
CSE 113, Devon McKee

Single-Program-Multiple-Data (SPMD) Parallelism

```
File: thread.cpp

#include <iostream>
#include <thread>

#define NUM_THREADS 4
#define SIZE 1024

using std::thread;

void worker(int *data, int start, int offset) {
    for (int i = start; i < start + offset; i++)
        data[i] = i;
}

int main() {
    int data[SIZE] = {0};
    thread threads[4];
    int chunk_size = SIZE / NUM_THREADS;
    for (int i = 0; i < NUM_THREADS; i++)
        threads[i] = thread(worker, data, chunk_size * i, chunk_size);
    for (int i = 0; i < NUM_THREADS; i++)
        threads[i].join();
    return 0;
}
```



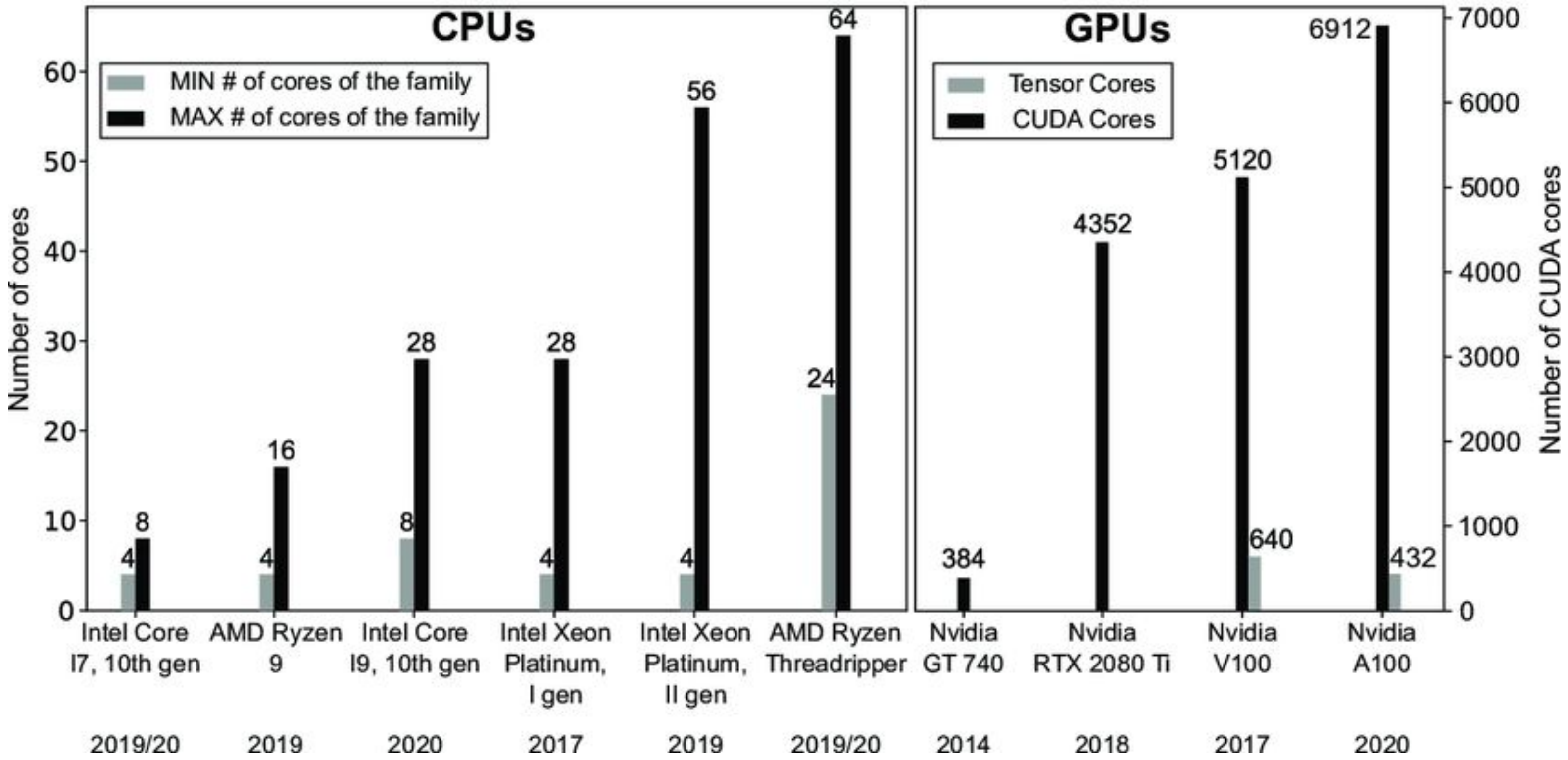
Limits of CPU SPMD Parallelism



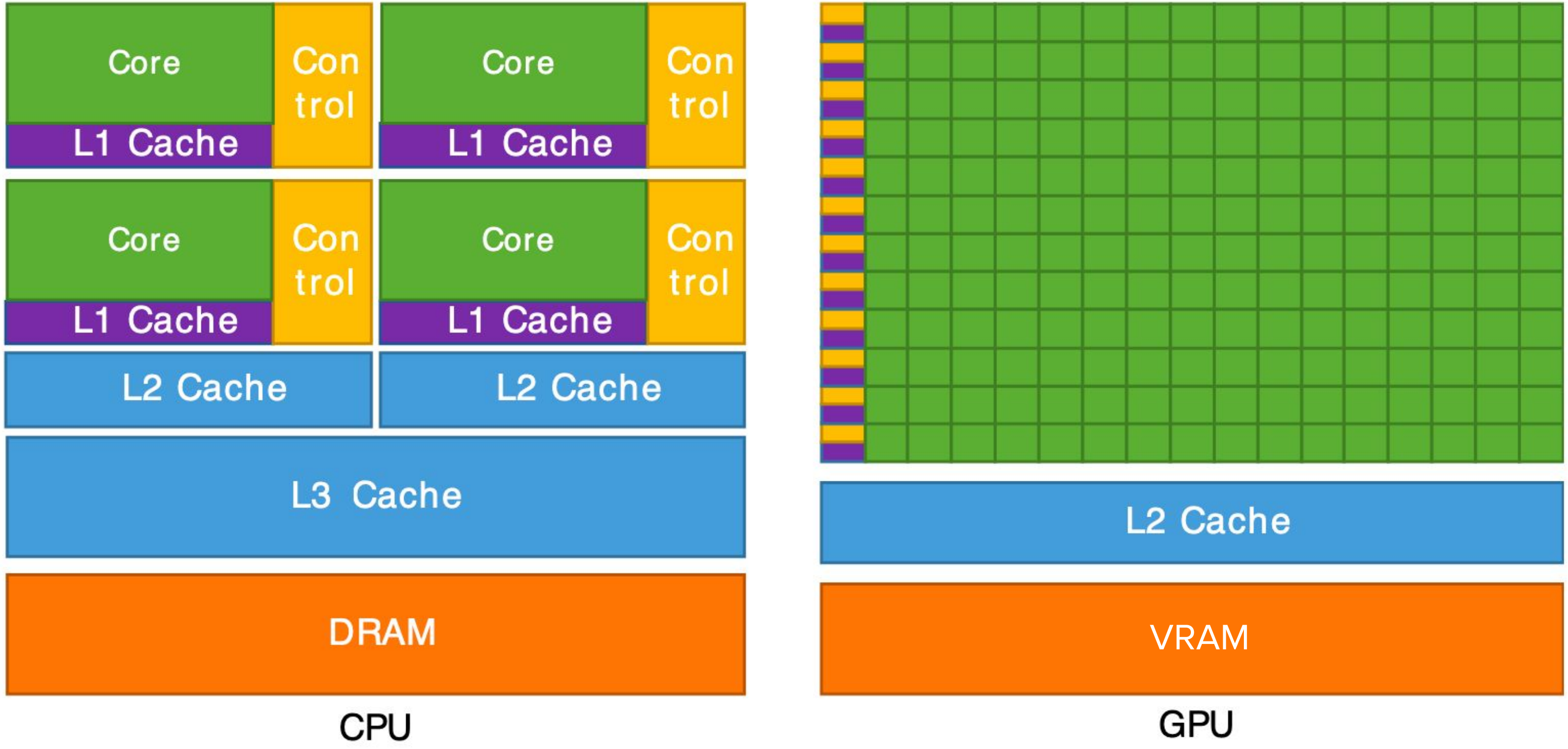
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

CPU vs. GPU: Cores

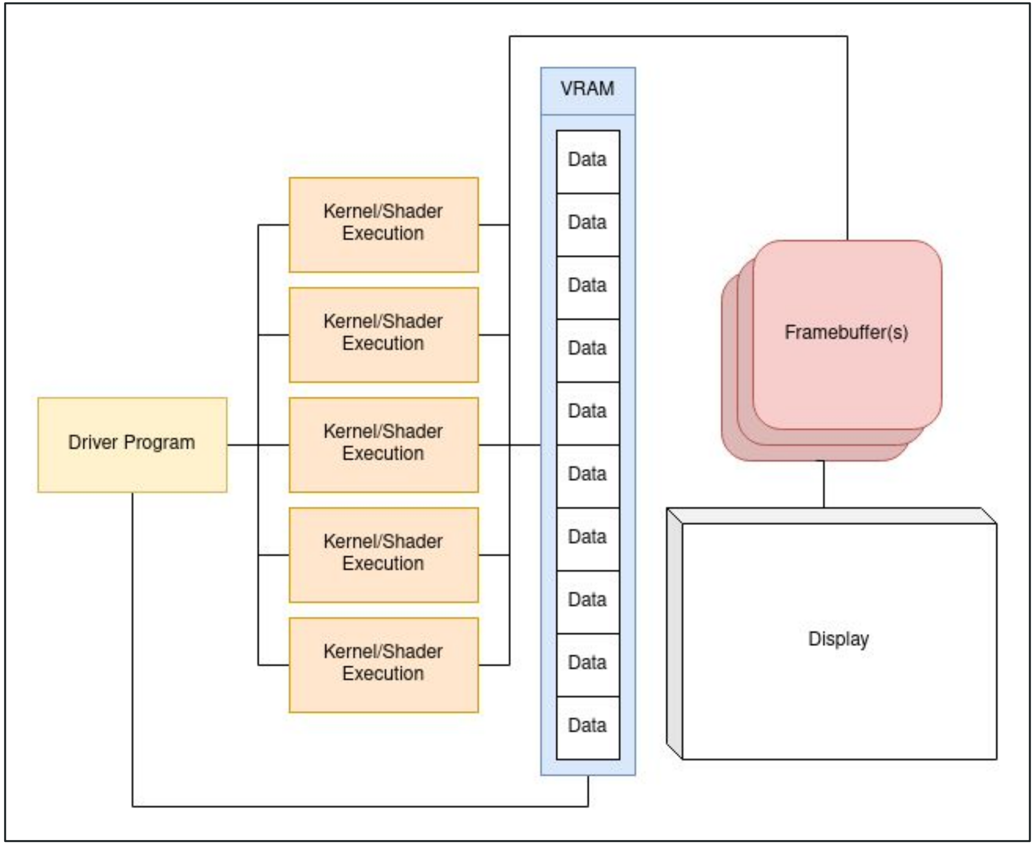
NVIDIA RTX 4090: 16384 cores!



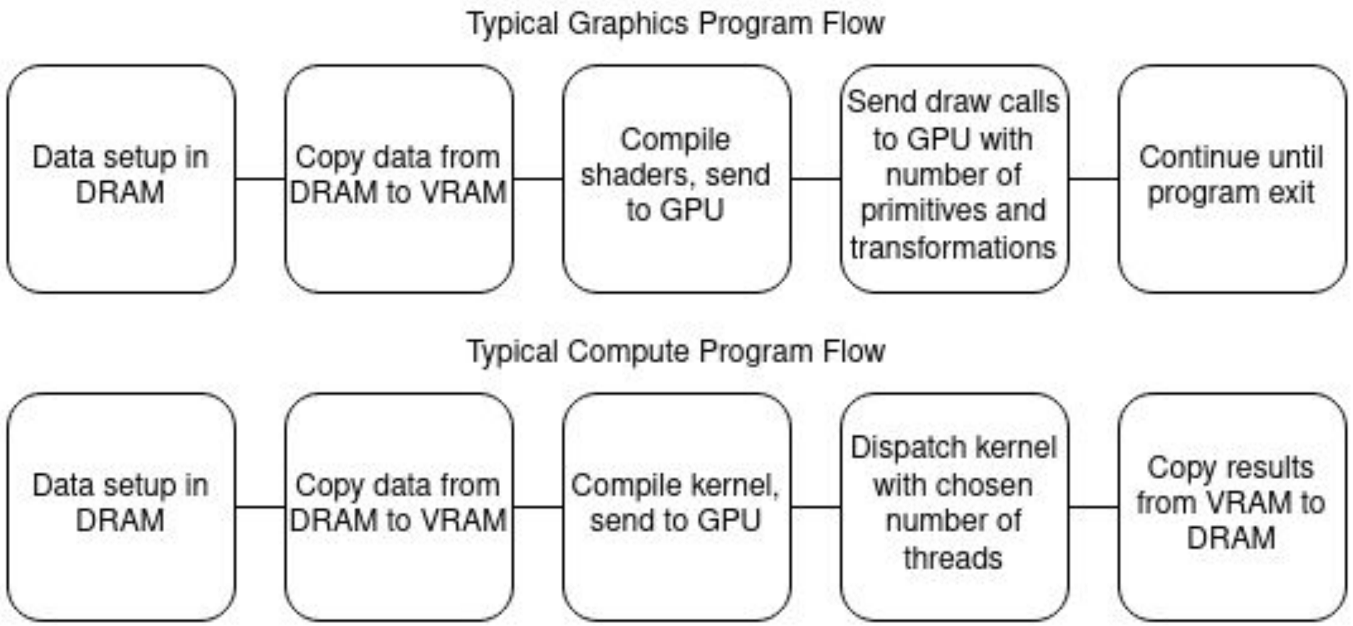
CPU vs. GPU: Architecture



GPU Programming



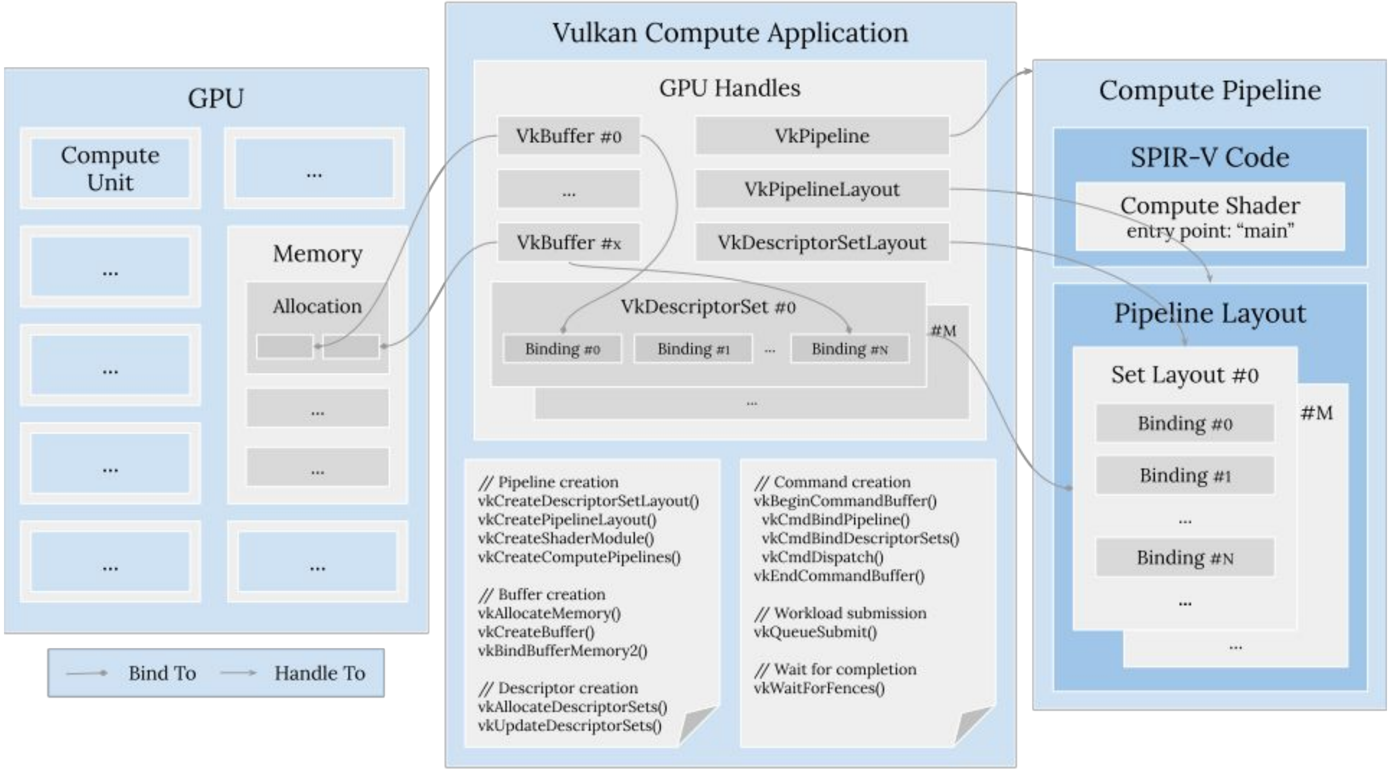
GPU Programming: Graphics vs. Compute



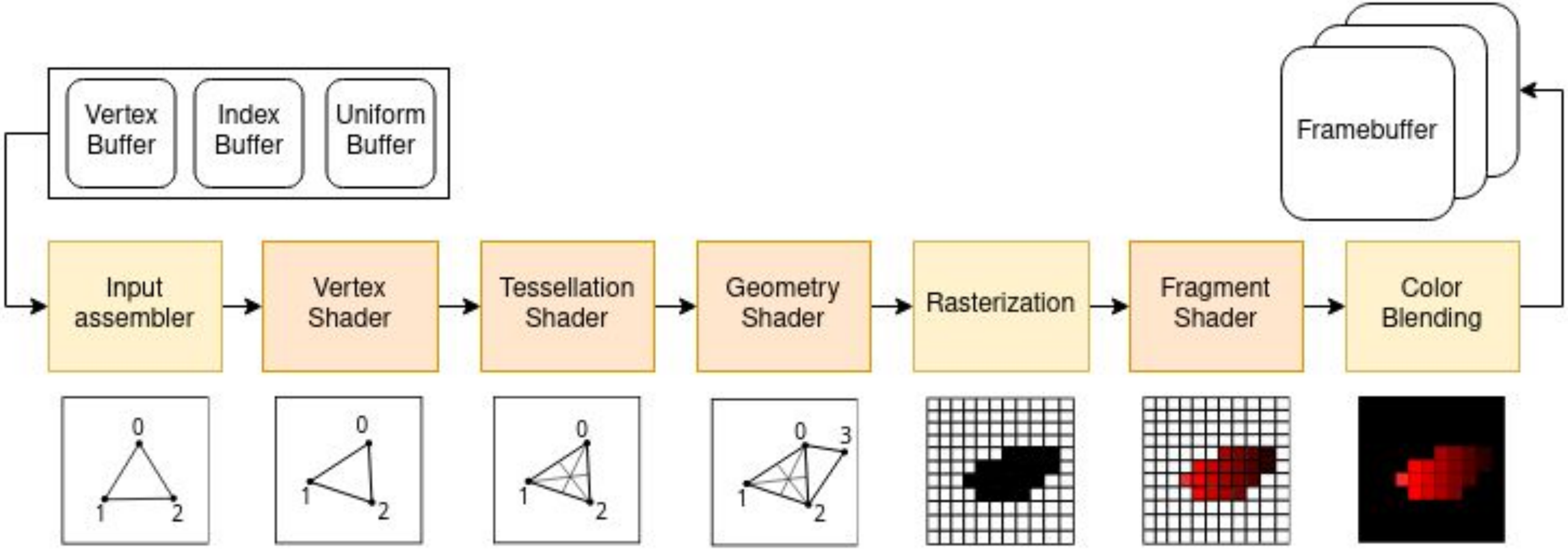
CUDA: Vector Add

```
#include <stdio.h>
#include <assert.h>

__global__ void vectorAdd(int* a, int* b, int* c, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    c[idx] = a[idx] + b[idx];
}

int main() {
    int n = 1<<24;
    int val = 1024;

    int *h_a, *h_b, *h_c; // Pointers to host memory
    int *d_a, *d_b, *d_c; // Pointers to device memory

    // Allocating host memory
    h_a = (int*)malloc(sizeof(int) * n);
    h_b = (int*)malloc(sizeof(int) * n);
    h_c = (int*)malloc(sizeof(int) * n);

    // Allocating device memory
    cudaMalloc(&d_a, sizeof(int) * n);
    cudaMalloc(&d_b, sizeof(int) * n);
    cudaMalloc(&d_c, sizeof(int) * n);

    // Setting up host memory
    for (int i = 0; i < n; i++) {
        h_a[i] = i;
        h_b[i] = 1024 - i;
    }

    // Copying memory from host to device
    cudaMemcpy(d_a, h_a, sizeof(int) * n, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, sizeof(int) * n, cudaMemcpyHostToDevice);

    // Executing kernel
    int blockSize = 1024; // Number of threads in each thread block
    int gridSize = (int)ceil((float)n / blockSize); // Number of thread blocks in grid
    vectorAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

    // Copying results from device to host
    cudaMemcpy(h_c, d_c, sizeof(int) * n, cudaMemcpyDeviceToHost);

    // Checking results
    for (int i = 0; i < n; i++)
        assert(h_c[i] = val);

    // Deallocating device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    // Deallocating host memory
    free(h_a);
    free(h_b);
    free(h_c);

    return 0;
}
```

Vulkan

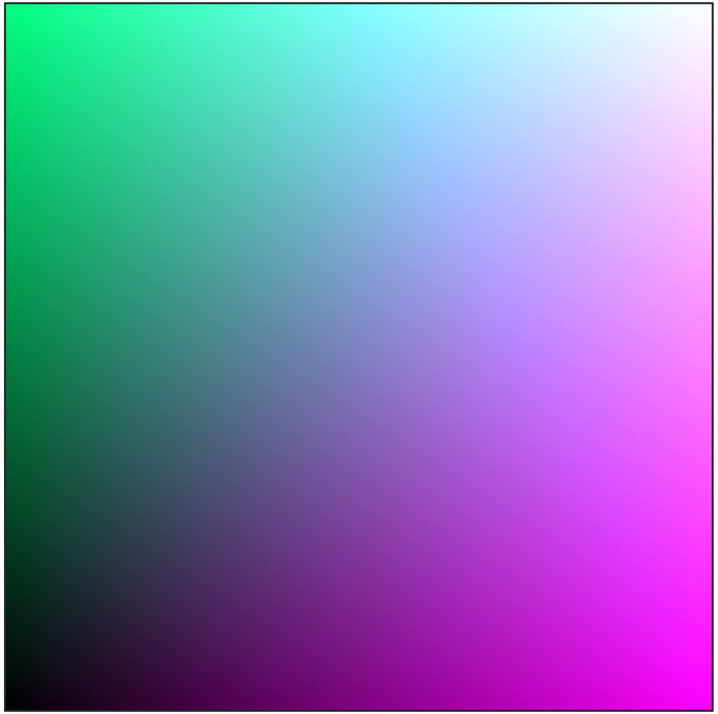

GPU
Compute Unit
...
Memory
Allocation
Bind To
Handle To
Vulkan Compute Application
GPU Handles
VkBuffer #0
VkBuffer #x
VkPipeline
VkPipelineLayout
VkDescriptorSetLayout
VkDescriptorSet #0
Binding #0
Binding #1
Binding #N
#M
// Pipeline creation
vkCreateDescriptorSetLayout()
vkCreatePipelineLayout()
vkCreateShaderModule()
vkCreateComputePipelines()
// Buffer creation
vkAllocateMemory()
vkCreateBuffer()
vkBindBufferMemory2()
// Descriptor creation
vkAllocateDescriptorSets()
vkUpdateDescriptorSets()
// Command creation
vkBeginCommandBuffer()
vkCmdBindPipeline()
vkCmdBindDescriptorSets()
vkCmdDispatch()
vkEndCommandBuffer()
// Workload submission
vkQueueSubmit()
// Wait for completion
vkWaitForFences()
Compute Pipeline
SPIR-V Code
Compute Shader
entry point: "main"
Pipeline Layout
Set Layout #0
Binding #0
Binding #1
Binding #N
#M

The Graphics Pipeline



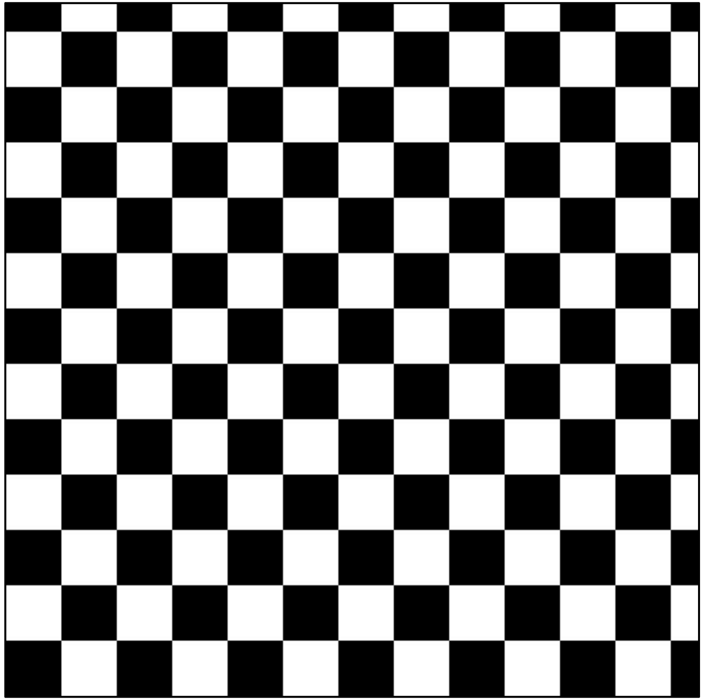
Fragment (Pixel) Shader

```
precision mediump float;

uniform vec2 u_resolution;

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec3 color = vec3(st.x, st.y, st.x + st.y / 2.0);
    gl_FragColor = vec4(color, 1.0);
}
```



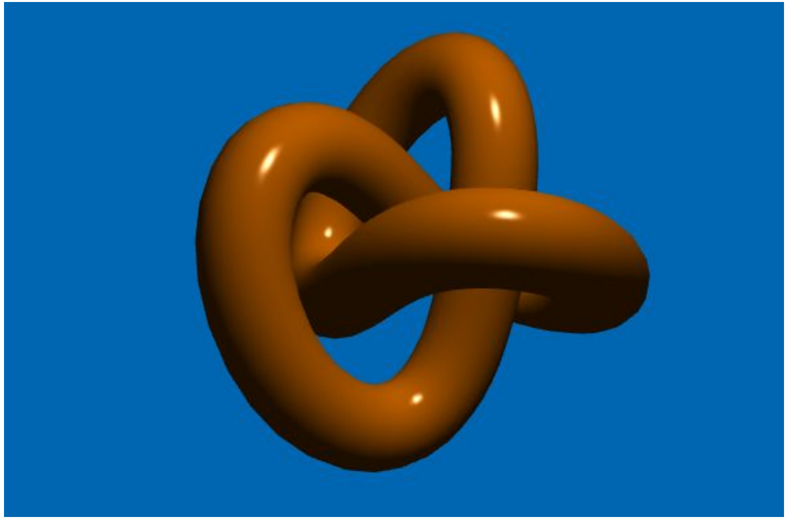
Fragment (Pixel) Shader

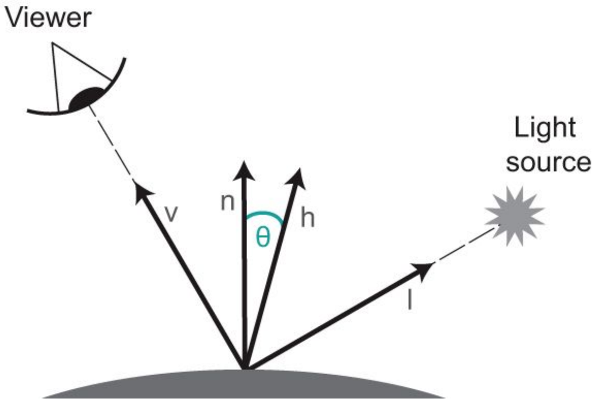
```
precision mediump float;

void main() {
    if (mod(floor(gl_FragCoord.x / 40.0), 2.0) == 0.0)
        if (mod(floor(gl_FragCoord.y / 40.0), 2.0) == 0.0)
            gl_FragColor = vec4(0, 0, 0, 1);
        else
            gl_FragColor = vec4(1, 1, 1, 1);
    else
        if (mod(floor(gl_FragCoord.y / 40.0), 2.0) == 0.0)
            gl_FragColor = vec4(1, 1, 1, 1);
        else
            gl_FragColor = vec4(0, 0, 0, 1);
}
```



Fragment (Pixel) Shader

```
File: phong.glsl

precision mediump float;
varying vec3 normalInterp;  // Surface normal
varying vec3 vertPos;       // Vertex position
uniform int mode;   // Rendering mode
uniform float Ka;   // Ambient reflection coefficient
uniform float Kd;   // Diffuse reflection coefficient
uniform float Ks;   // Specular reflection coefficient
uniform float shininessVal; // Shininess
// Material color
uniform vec3 ambientColor;
uniform vec3 diffuseColor;
uniform vec3 specularColor;
uniform vec3 lightPos; // Light position

void main() {
  vec3 N = normalize(normalInterp);
  vec3 L = normalize(lightPos - vertPos);

  // Lambert's cosine law
  float lambertian = max(dot(N, L), 0.0);
  float specular = 0.0;
  if(lambertian > 0.0) {
    vec3 R = reflect(-L, N);      // Reflected light vector
    vec3 V = normalize(-vertPos); // Vector to viewer
    // Compute the specular term
    float specAngle = max(dot(R, V), 0.0);
    specular = pow(specAngle, shininessVal);
  }
  gl_FragColor = vec4(Ka * ambientColor +
                      Kd * lambertian * diffuseColor +
                      Ks * specular * specularColor, 1.0);
}
```

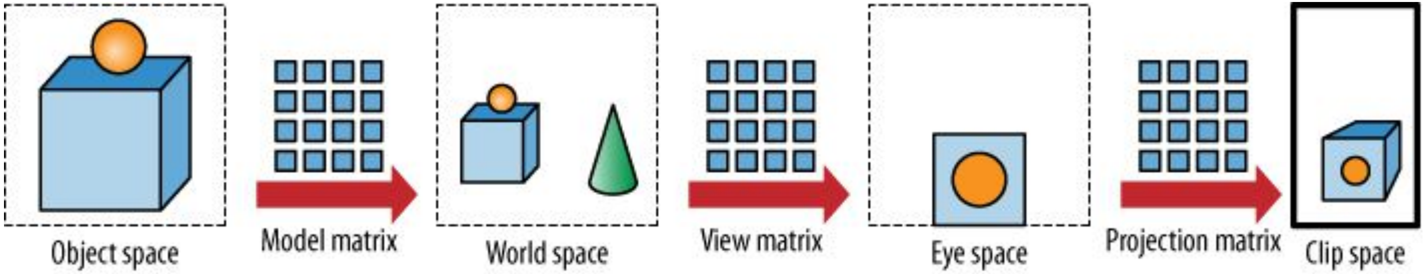


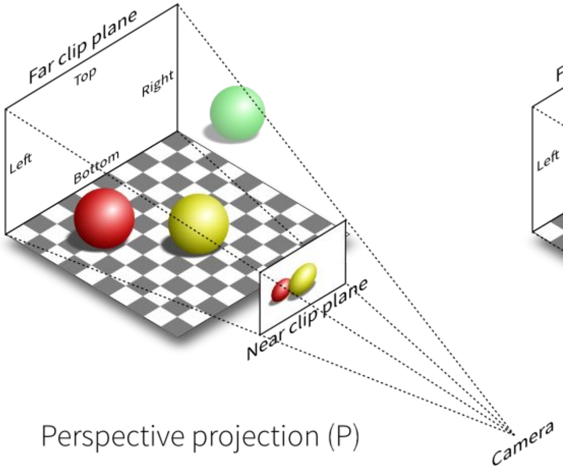


Vertex Shader

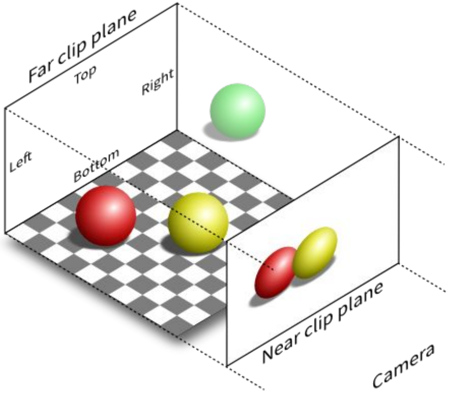
```
File: vertex.glsl

#version 410 core
in vec2 point;
in vec3 color;
out vec4 vColor;
uniform float radAng = 0;
uniform float time = 0;

vec2 Rotate2D(vec2 v) {
    float c = cos(radAng), s = sin(radAng);
    return vec2(c*v.x-s*v.y, s*v.x+c*v.y);
}
void main() {
    vec2 point_r = Rotate2D(point);
    point_r.x *= (sin(time));
    point_r.y *= (sin(time));
    vColor = vec4(sin(color.x + point.x + time),
        cos(color.y + point.y + time - 5),
        (cos(color.x + time)+sin(color.y + time))/2,
        1
    );
    gl_Position = vec4(point_r, 0, 1);
}
```

3D Graphics

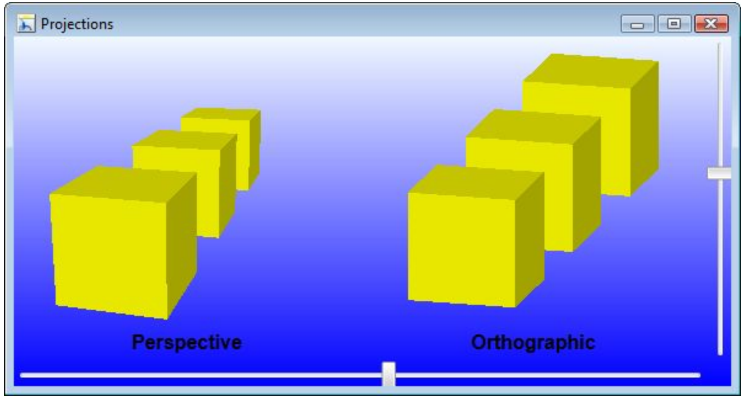




Perspective projection (P)



Orthographic projection (O)



Combining Graphics and Compute

GPU Resources

- Shaders:
  - https://thebookofshaders.com/
- Graphics/OpenGL:
  - https://learnopengl.com/
  - http://www.opengl-tutorial.org/
- CUDA:
  - https://developer.nvidia.com/blog/even-easier-introduction-cuda/
  - https://cuda-tutorial.readthedocs.io
- GPU Architecture
  - https://bit.ly/appendix-c