

# Go on GPU

欧长坤

*changkun.de/s/gogpu*

GopherChina 2023

Session "Foundational Toolchains"

2023 June 10

# 关于我

- 慕尼黑大学博士 (@mimuc/staff) 研究方向人在环路优化 (human-in-the-loop optimization)
- Sixt 高级工程师 (@sixt/pricing-yield) 全球自动定价系统
- 感兴趣机器学习、图形学、认知和社会心理学、哲学、系统编程
- 业余活跃在开源社区 (@golang/{ios,android}, @fyne-io/member, @talkgo/member, @golang-design/owner, ...)



# 大纲

- 与 GPU 进行交互的基本知识
- 在 Go 程序中支持 GPU 加速
- 使用 Go 进行 GPU 计算的挑战
- 总结

# 大纲

- 与 GPU 进行交互的基本知识
  - 使用 GPU 的动机
  - GPU 驱动和标准
  - 渲染管线和计算管线
  - Vulkan/Metal/DX12/OpenGL
- 在 Go 程序中支持 GPU 加速
- 使用 Go 进行 GPU 计算的挑战
- 总结

# 使用 GPU 加速计算的动机

提高系统计算性能

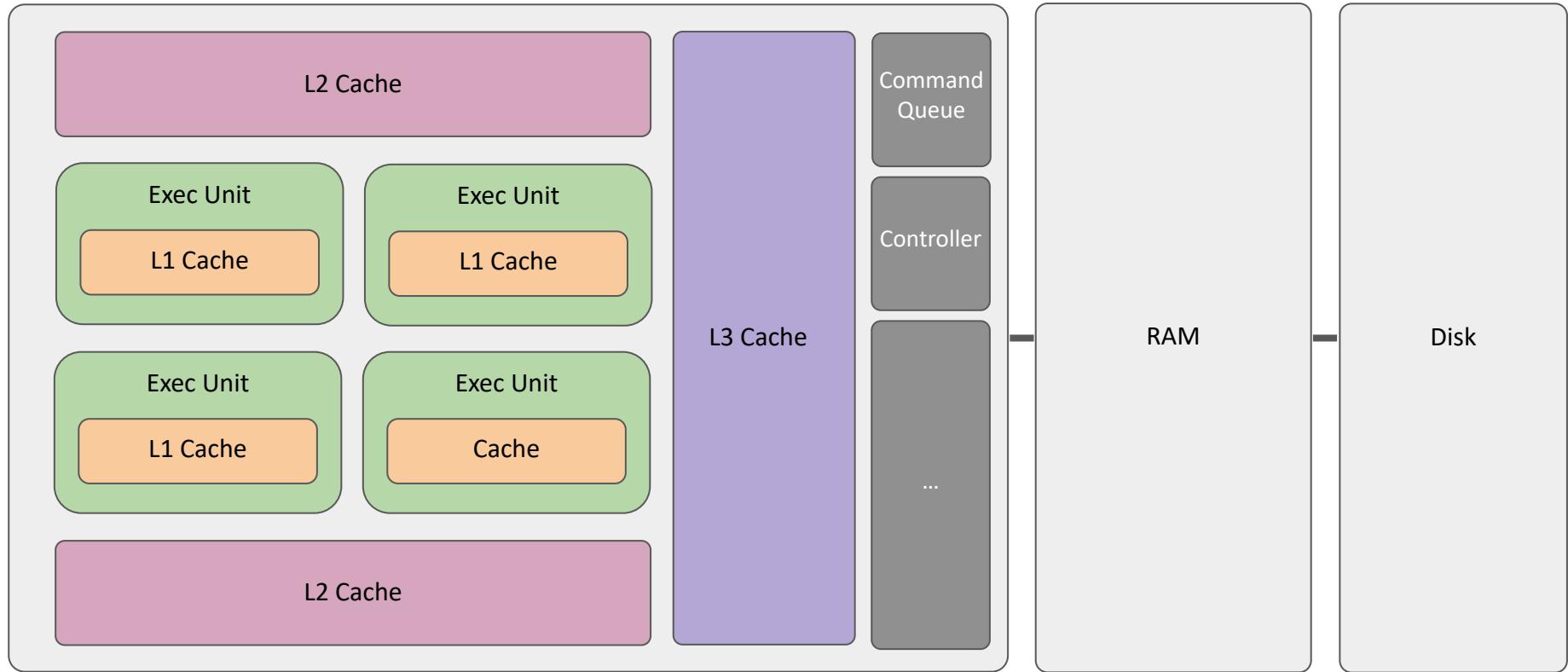
更大的并发量

大规模数据处理

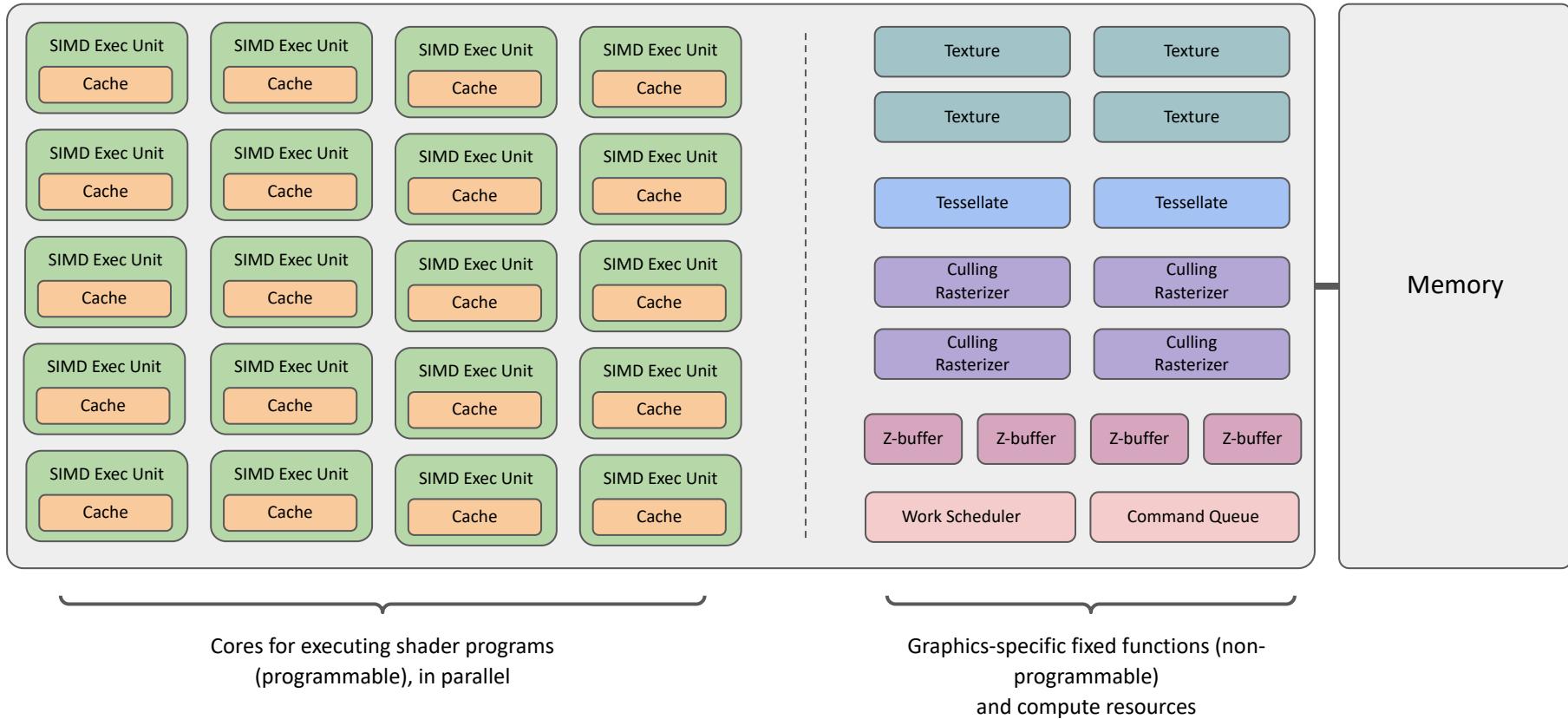
机器学习、深度学习、图形学渲染等等



# CPU 的架构

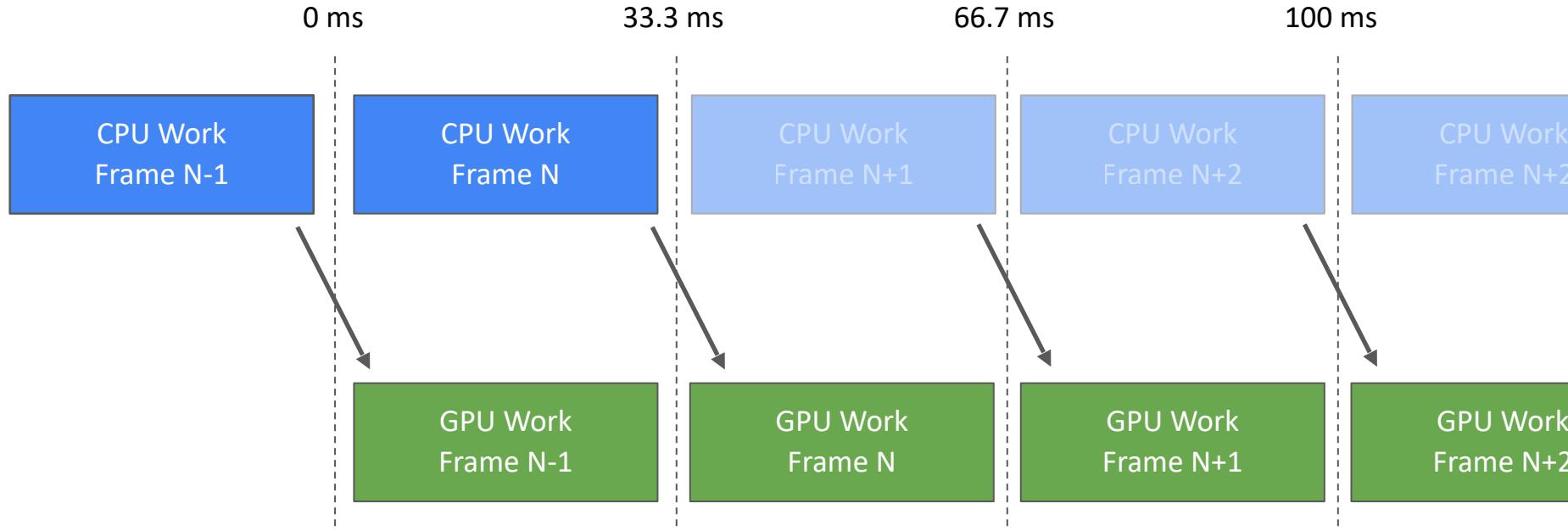


# GPU 的架构



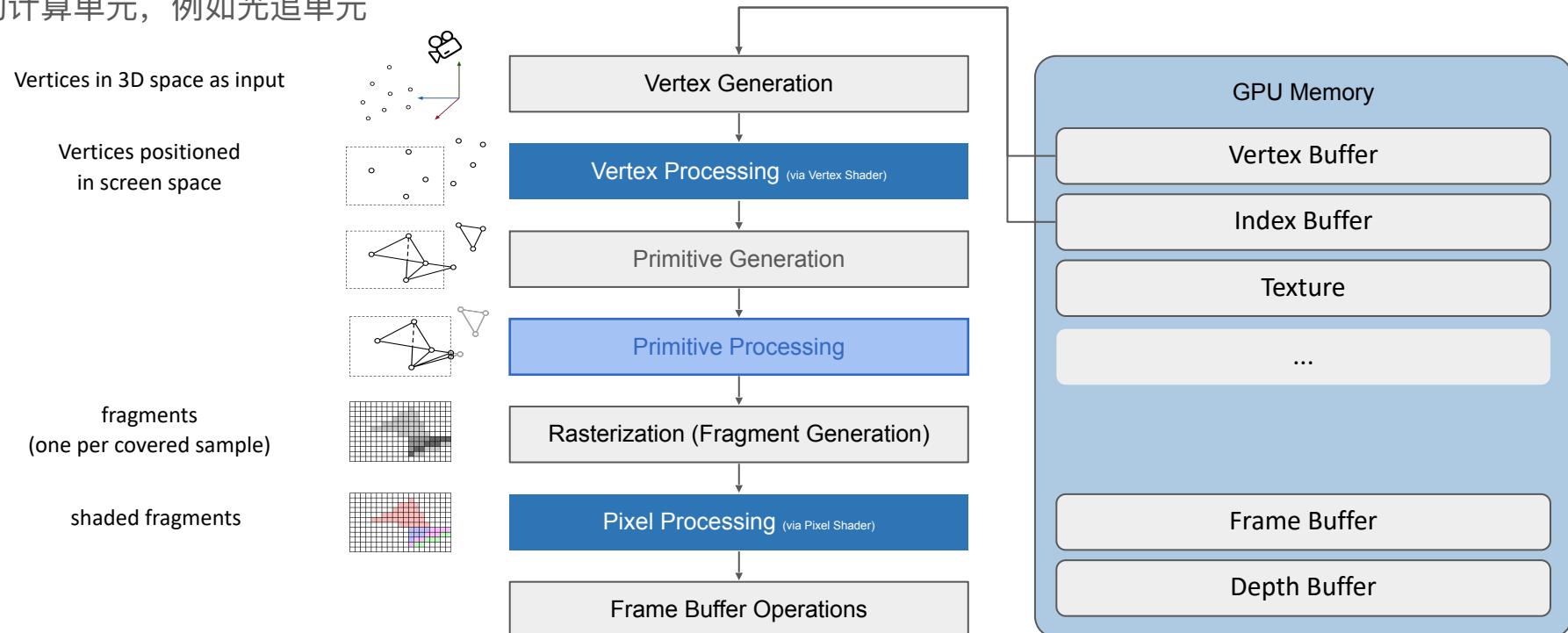
# CPU 与 GPU 之间的协作: 渲染场景

CPU 和 GPU 之间是一个“主仆”关系，CPU 负责准备 GPU 执行任务的步骤和相关资源，GPU 负责执行常见的游戏场景，以 30fps 为例，希望 GPU 在 33.3ms 之内完成一帧的渲染



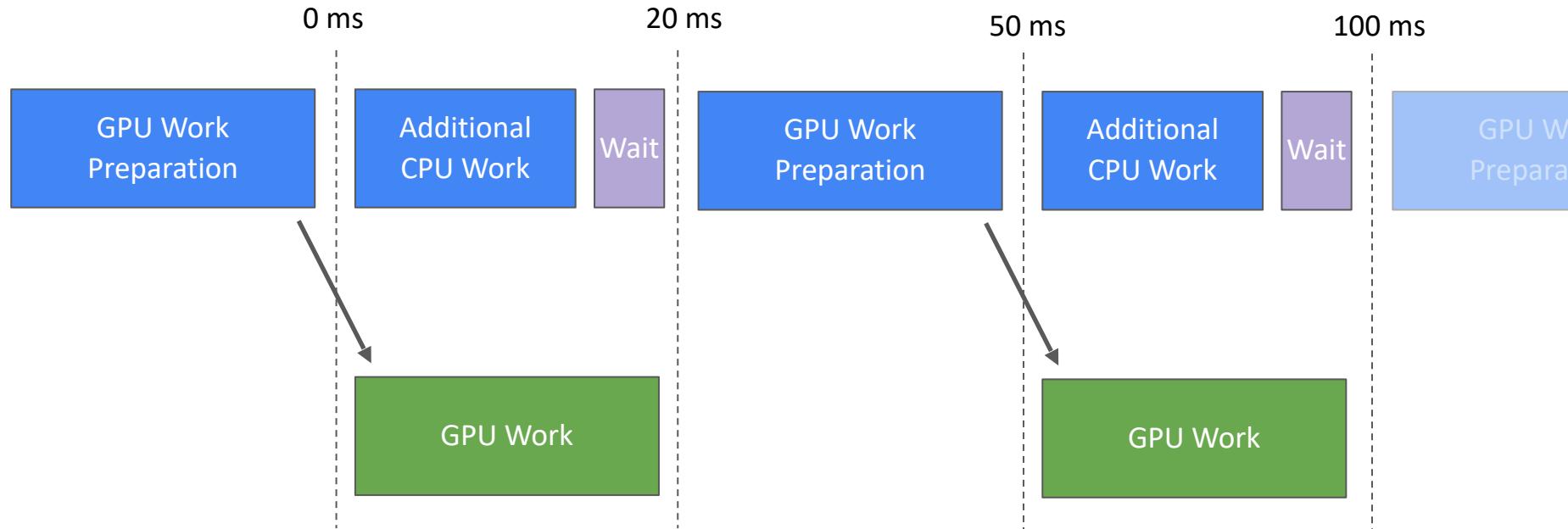
# 图形渲染管线

渲染管线关注在执行 Render Pass，每个 Pass 包含多个阶段，每个阶段都具有可定制的着色器，GPU 包含专用的计算单元，例如光追单元

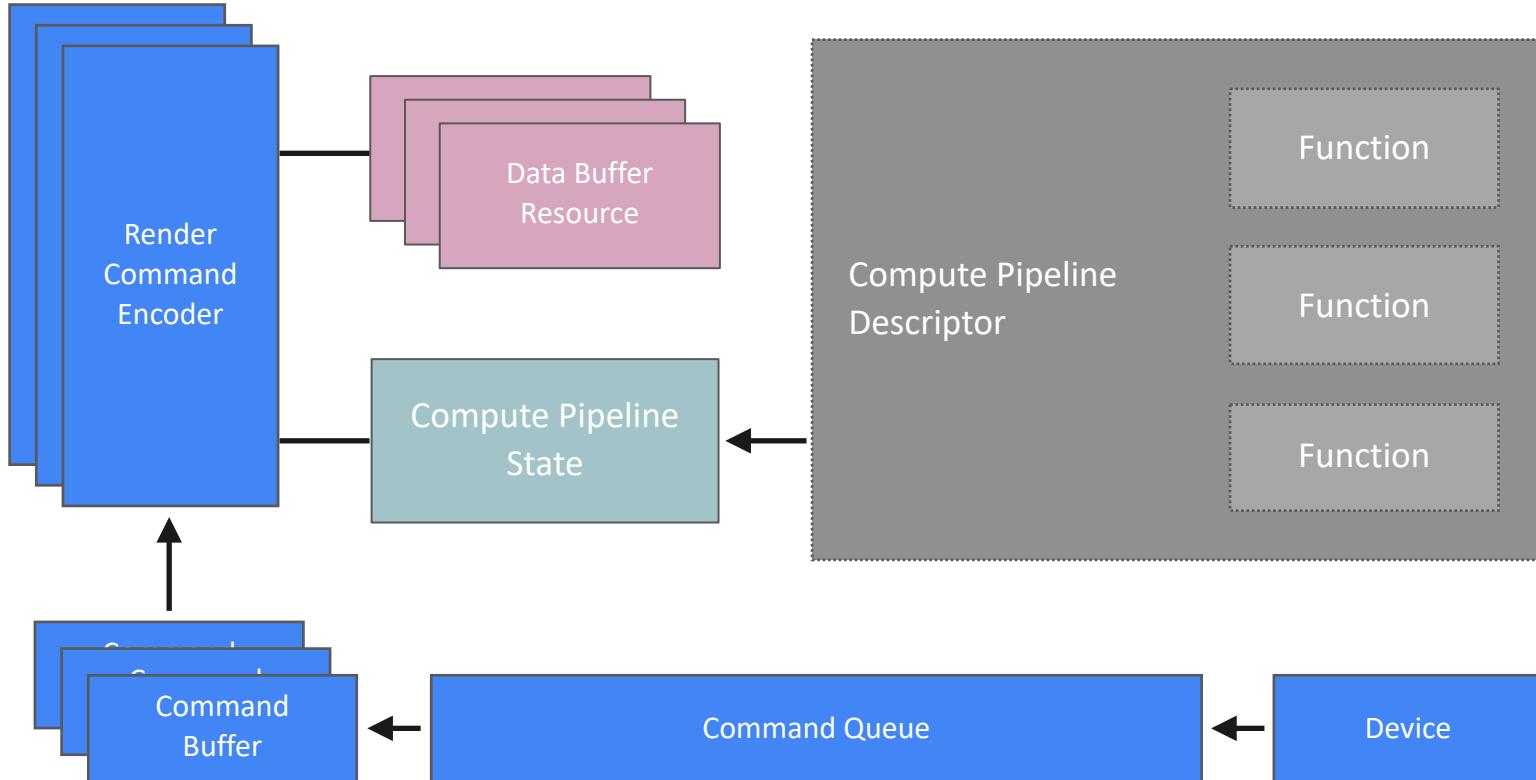


# CPU 与 GPU 之间的协作: 计算场景

随着通用计算的发展, GPU 开始摆脱渲染流水线的限制, 可以直接当做一个外部计算资源, 随时待命等待 CPU 调度的计算任务



# 通用计算管线



# 指令提交模型

指令编码器 (command encoder) 将 API 指令转译为硬件指令，包装了所有的任务状态

硬件指令直接存储在指令缓存上

根据任务的不同，具有不同类型的指令编码器（计算编码器、渲染编码器等等）

指令缓存可以在不同线程上进行创建，可以显式的、并发的提交给指令队列

# 业界的各类图形计算标准

早年的老牌 OpenGL 系列标准

以及后来的继承者 Vulkan

标准制定者并没有实现全 对需求的把握也不够全面

平台厂商开始根据自身需求设计自己的标准

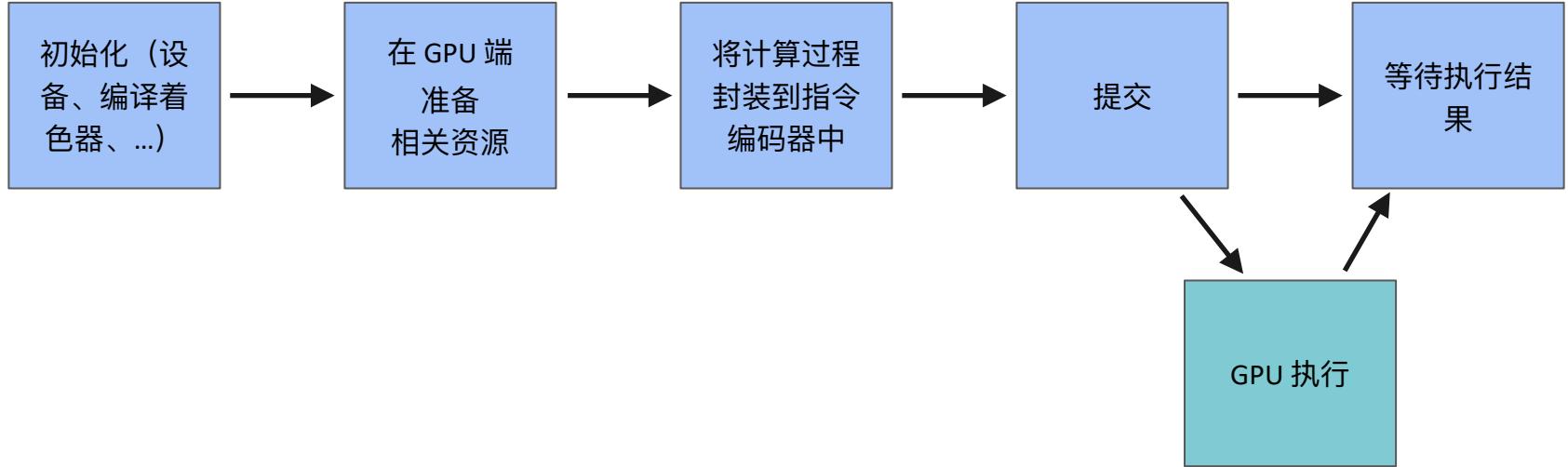
Web 平台上的各类基于以上标准的通用标准



# 大纲

- 与 GPU 进行交互的基本知识
- 在 Go 程序中支持 GPU 加速
  - 使用 GPU 进行加速的基本思路
  - 两个例子
  - 需要考虑的问题
- 使用 Go 进行 GPU 计算的挑战
- 总结

# 在 Go 程序中支持 GPU 加速基本流程



# 准备工作: 驱动 - 以 Metal 为例

```
package mtl // import "changkun.de/x/gopherchina2023gogpu/gpu/mlt"

/*
#cgo CFLAGS: -Werror -fmodules -x objective-c
#cgo LDFLAGS: -framework Metal -framework CoreGraphics
#include "mtl.h"
*/
import "C"

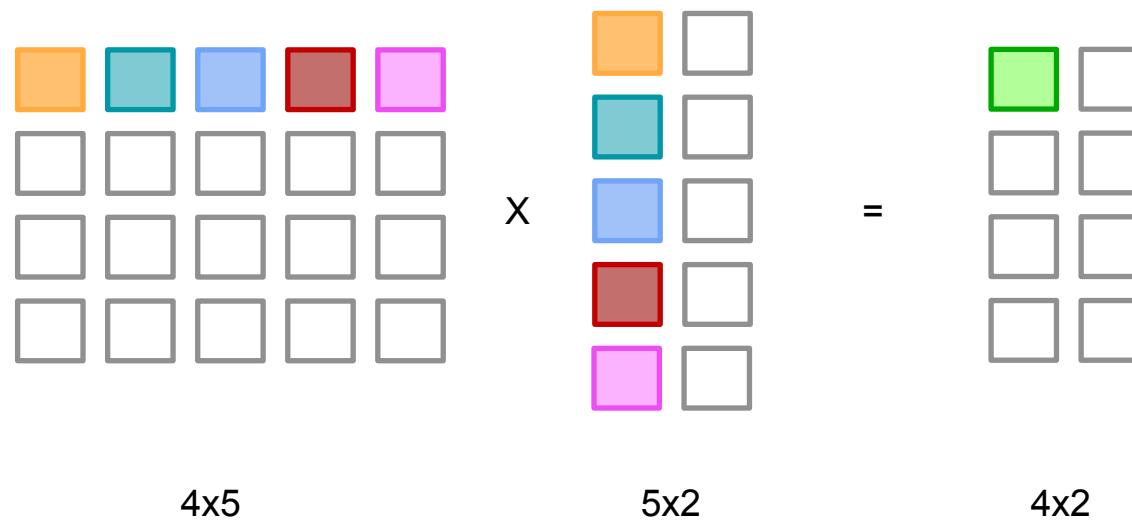
type Device struct
func (d Device) MakeCommandQueue() CommandQueue

@import Metal;
#include "mtl.h"
void * Device_MakeCommandQueue(void * device) {
    return [(id<MTLDevice>)device newCommandQueue];
}
...
...
```

# 例子1：矩阵乘法

矩阵乘法几乎支撑了所有的现代科学计算，它也是一个非常经典的性能优化问题

例如：神经网络层与层之间的运算是通过矩阵完成的，科学计算中大量的求解器均依赖矩阵，等等



# 例子1：矩阵乘法

矩阵乘法几乎支撑了所有的现代科学，它的计算是一个非常经典的性能优化问题

例如：神经网络层与层之间的运算是通过矩阵完成的，科学计算中大量的求解器均依赖矩阵，等等

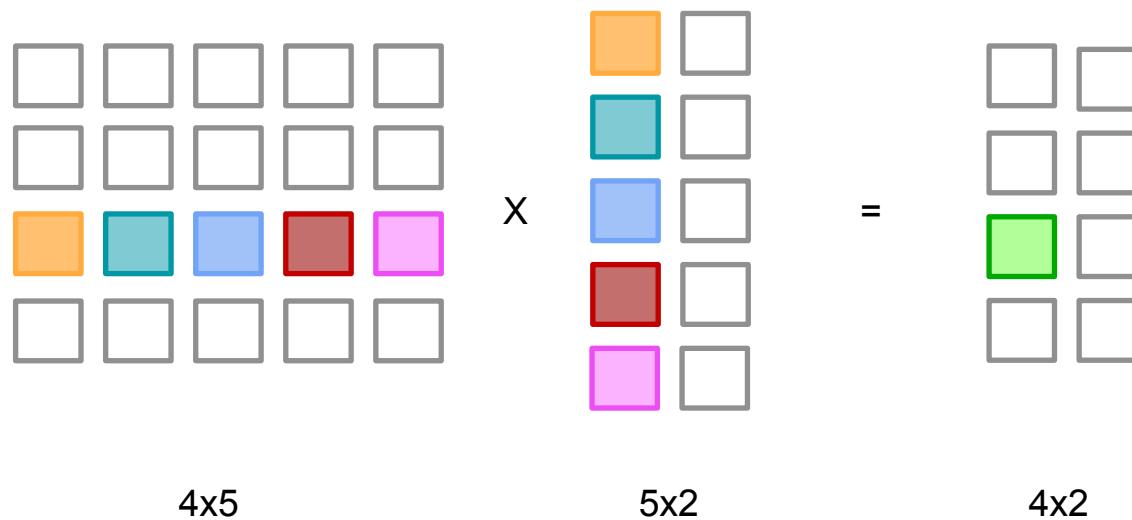
$$\begin{matrix} & \begin{matrix} \square & \square & \square & \square & \square \end{matrix} \\ \begin{matrix} \square & \square & \square & \square & \square \end{matrix} & \times & \begin{matrix} \text{orange} & \square \\ \text{teal} & \square \\ \text{blue} & \square \\ \text{red} & \square \\ \text{pink} & \square \end{matrix} \\ & \begin{matrix} \square & \square & \square & \square & \square \end{matrix} \\ & \begin{matrix} \square & \square & \square & \square & \square \end{matrix} \end{matrix} = \begin{matrix} & \begin{matrix} \square & \square \end{matrix} \\ \begin{matrix} \square & \square & \text{green} & \square \\ \square & \square & \square & \square \\ \square & \square & \square & \square \\ \square & \square & \square & \square \end{matrix} \end{matrix}$$

$4 \times 5$        $5 \times 2$        $4 \times 2$

# 例子1：矩阵乘法

矩阵乘法几乎支撑了所有的现代科学，它的计算是一个非常经典的性能优化问题

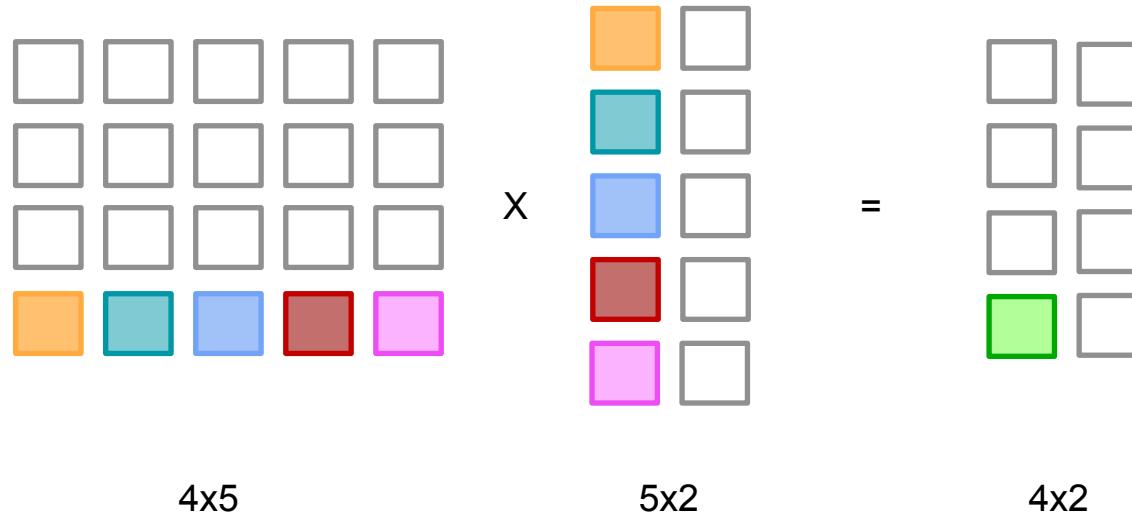
例如：神经网络层与层之间的运算是通过矩阵完成的，科学计算中大量的求解器均依赖矩阵，等等



# 例子1：矩阵乘法

矩阵乘法几乎支撑了所有的现代科学，它的计算是一个非常经典的性能优化问题

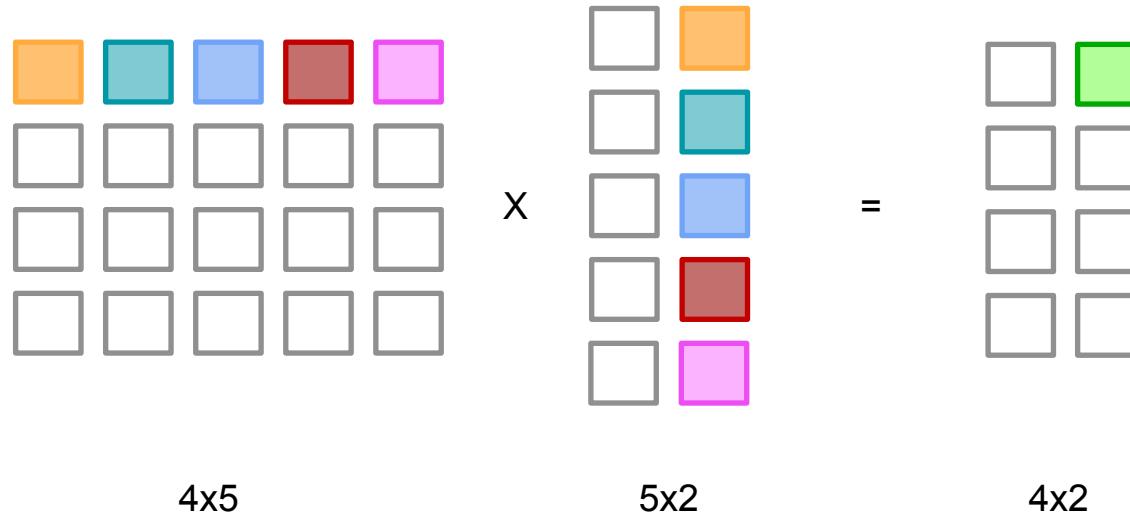
例如：神经网络层与层之间的运算是通过矩阵完成的，科学计算中大量的求解器均依赖矩阵，等等



# 例子1：矩阵乘法

矩阵乘法几乎支撑了所有的现代科学，它的计算是一个非常经典的性能优化问题

例如：神经网络层与层之间的运算是通过矩阵完成的，科学计算中大量的求解器均依赖矩阵，等等



# 例子1：矩阵乘法

矩阵乘法几乎支撑了所有的现代科学，它的计算是一个非常经典的性能优化问题

例如：神经网络层与层之间的运算是通过矩阵完成的，科学计算中大量的求解器均依赖矩阵，等等

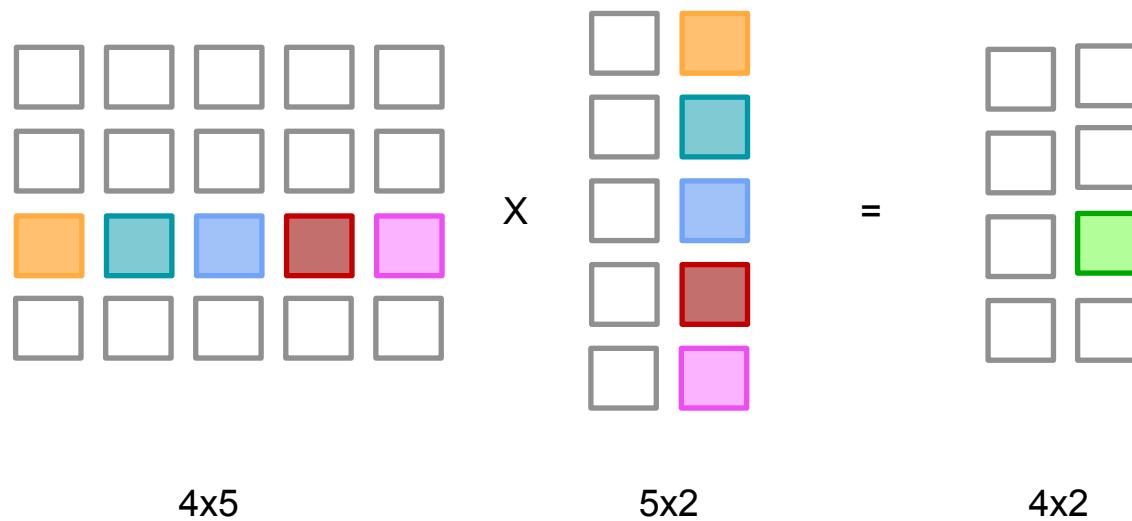
$$\begin{array}{c} \begin{array}{ccccc} \square & \square & \square & \square & \square \\ \text{orange} & \text{teal} & \text{blue} & \text{red} & \text{pink} \\ \square & \square & \square & \square & \square \\ \square & \square & \square & \square & \square \end{array} & \times & \begin{array}{cc} \square & \text{orange} \\ \square & \text{teal} \\ \square & \text{blue} \\ \square & \text{red} \\ \square & \text{pink} \end{array} & = & \begin{array}{cc} \square & \square \\ \square & \text{green} \\ \square & \square \\ \square & \square \end{array} \end{array}$$

4x5                    5x2                    4x2

# 例子1：矩阵乘法

矩阵乘法几乎支撑了所有的现代科学，它的计算是一个非常经典的性能优化问题

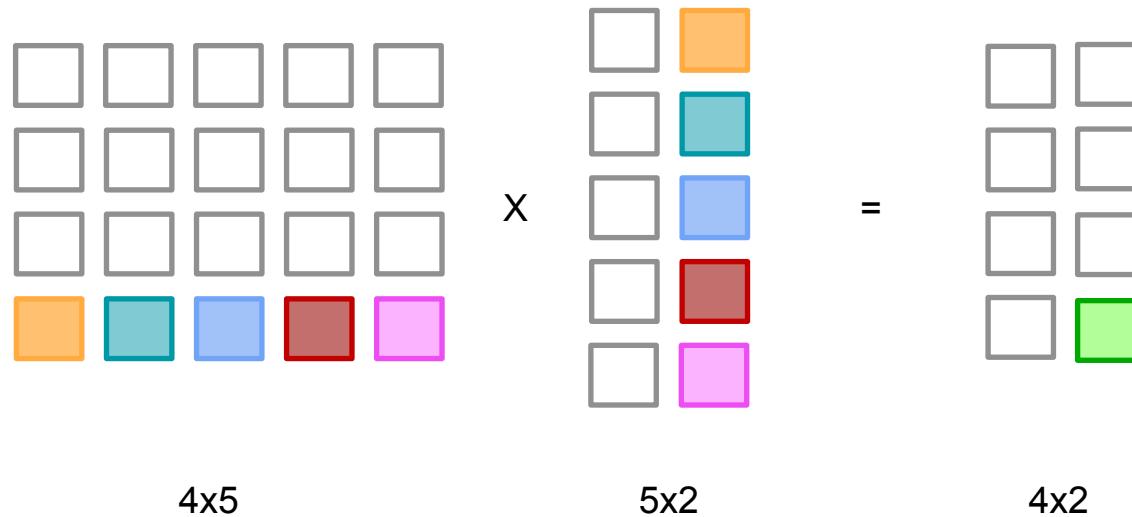
例如：神经网络层与层之间的运算是通过矩阵完成的，科学计算中大量的求解器均依赖矩阵，等等



# 例子1：矩阵乘法

矩阵乘法几乎支撑了所有的现代科学，它的计算是一个非常经典的性能优化问题

例如：神经网络层与层之间的运算是通过矩阵完成的，科学计算中大量的求解器均依赖矩阵，等等



# 例子1：矩阵乘法

```
// Mat represents a Row x Col matrix.
type Mat[T Type] struct {
    Row  int
    Col  int
    Data []T
}

// MulNaive applies matrix multiplication of two given matrix, and returns
// the resulting matrix: r = m*n. This is a O(n^3) implementation.
func (m Mat[T]) MulNaive(n Mat[T]) Mat[T] {
    r := Mat[T]{Row: m.Row, Col: n.Col, Data: make([]T, m.Row*n.Col)}
    for i := 0; i < m.Row; i++ {
        for j := 0; j < n.Col; j++ {
            sum := T(0)
            for k := 0; k < m.Col; k++ {
                sum += m.Get(i, k) * n.Get(k, j)
            }
            r.Set(i, j, sum)
        }
    }
    return r
}
```

# 例子1：矩阵乘法

```
// Mat represents a Row x Col matrix.
type Mat[T Type] struct {
    Row  int
    Col  int
    Data []T
}

// MulNaive applies matrix multiplication of two given matrix, and returns
// the resulting matrix: r = m*n. This is a O(n^3) implementation.
func (m Mat[T]) MulNaive(n Mat[T]) Mat[T] {
    r := Mat[T]{Row: m.Row, Col: n.Col, Data: make([]T, m.Row*n.Col)}
    for i := 0; i < m.Row; i++ {
        for j := 0; j < n.Col; j++ {
            sum := T(0)
            for k := 0; k < m.Col; k++ {
                sum += m.Get(i, k) * n.Get(k, j)
            }
            r.Set(i, j, sum)
        }
    }
    return r
}
```

1

# 例子1：矩阵运算

```
import "changkun.de/x/gopherchina2023gogpu/gpu/mlt" // Metal driver 1
var (
    //go:embed mul.metal
    mulMetal string
    device    mtl.Device
    cq        mtl.CommandQueue
    lib       mtl.Library
    funcMul   mtl.Function
    funcMulCPS mtl.ComputePipelineState
)
func init() {
    // 0. Initialization
    device = try(mtl.CreateSystemDefaultDevice())
    cq = device.MakeCommandQueue() 2

    lib = try(device.MakeLibrary(mulMetal, mtl.CompileOptions{
        LanguageVersion: mtl.LanguageVersion2_4,
    })) 3
    funcMul = try(lib.MakeFunction("mul"))
    funcMulCPS = try(device.MakeComputePipelineState(funcMul))
}
```

# 例子1：矩阵运算

```
// Mul is a GPU version of math.Mat[T].Mul method and it multiplies
// two matrices m1 and m2 and returns the result.
func Mul[T math.Type](m1, m2 math.Mat[T]) math.Mat[T] {

    // 1. Allocate GPU buffers
    a := device.MakeBuffer(unsafe.Pointer(&m1.Data[0]), uintptr(math.TypeSize[T]
() * len(m1.Data)), mtl.ResourceStorageModeShared)
    defer a.Release()
    b := device.MakeBuffer(unsafe.Pointer(&m2.Data[0]), uintptr(math.TypeSize[T]
() * len(m2.Data)), mtl.ResourceStorageModeShared)
    defer b.Release()
    out := device.MakeBuffer(nil, uintptr(math.TypeSize[T] () * m1.Row*m2.Col),
mtl.ResourceStorageModeShared)
    defer out.Release()
    dp := device.MakeBuffer(unsafe.Pointer(&params{
        ColA: int32(m1.Col),
        ColB: int32(m2.Col),
    }), unsafe.Sizeof(params[T]{}), mtl.ResourceStorageModeShared)
    defer dp.Release()

    ...
}
```

1

# 例子1：矩阵运算

```
#include <metal_stdlib>
using namespace metal;

struct params { uint colA; uint colB; };

kernel void mul(device const float* inA      [[ buffer(0) ]],
                device const float* inB      [[ buffer(1) ]],
                device      float* out       [[ buffer(2) ]],
                device const params& params [[ buffer(3) ]],
                uint                 index    [[thread_position_in_grid]]) {

    uint i = index / uint(params.colB);
    uint j = index % uint(params.colB);
    float sum = 0.0;
    for (uint k = 0; k < params.colA; k++) {
        float a = inA[i * int(params.colA) + k];
        float b = inB[k * int(params.colB) + j];
        sum += a * b;
    }
    out[index] = sum;
}
```

1

2

3

# 例子1：矩阵运算

```
// Mul is a GPU version of math.Mat[T].Mul method and it multiplies
// two matrices m1 and m2 and returns the result.
func Mul[T math.Type](m1, m2 math.Mat[T]) math.Mat[T] {
    ...
    // 2. Create command buffer, command encoder, and set buffer
    cb := cq.MakeCommandBuffer()
    defer cb.Release()

    ce := cb.MakeComputeCommandEncoder() 1
    ce.SetComputePipelineState(fn.funcMul.cps)
    ce.SetBuffer(a, 0, 0)
    ce.SetBuffer(b, 0, 1)
    ce.SetBuffer(out, 0, 2)
    ce.SetBuffer(dp, 0, 3)
    ...
}
```

# 例子1：矩阵运算

```
// Mul is a GPU version of math.Mat[T].Mul method and it multiplies
// two matrices m1 and m2 and returns the result.
func Mul[T math.Type](m1, m2 math.Mat[T]) math.Mat[T] {
    ...
    // 2. Create command buffer, command encoder, and set buffer
    cb := cq.MakeCommandBuffer()
    defer cb.Release()
    ...

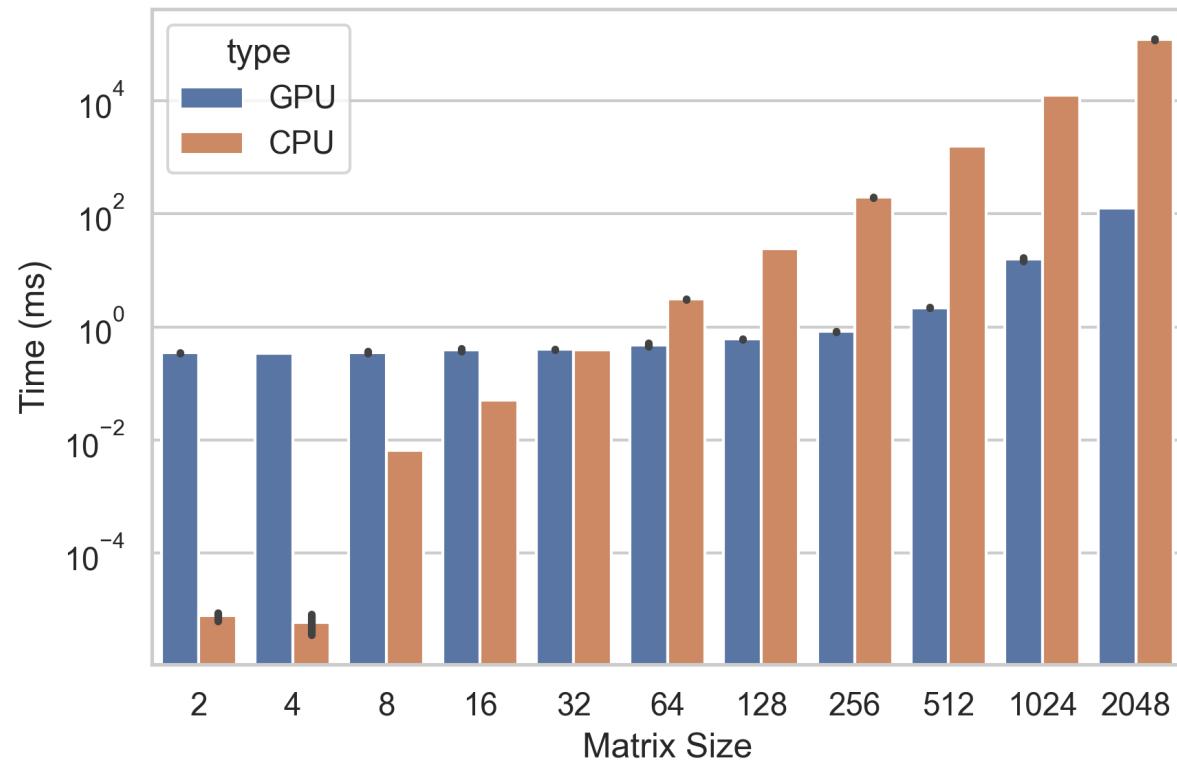
    // 3. Dispatch threads and commit command encoders in the command buffer
    ce.DispatchThreads(
        mtl.Size{Width: m1.Row * m2.Col, Height: 1, Depth: 1},           1
        mtl.Size{Width: 1, Height: 1, Depth: 1})
    ce.EndEncoding()
    cb.Commit()
    cb.WaitUntilCompleted()
    ...
}
```

# 例子1：矩阵运算

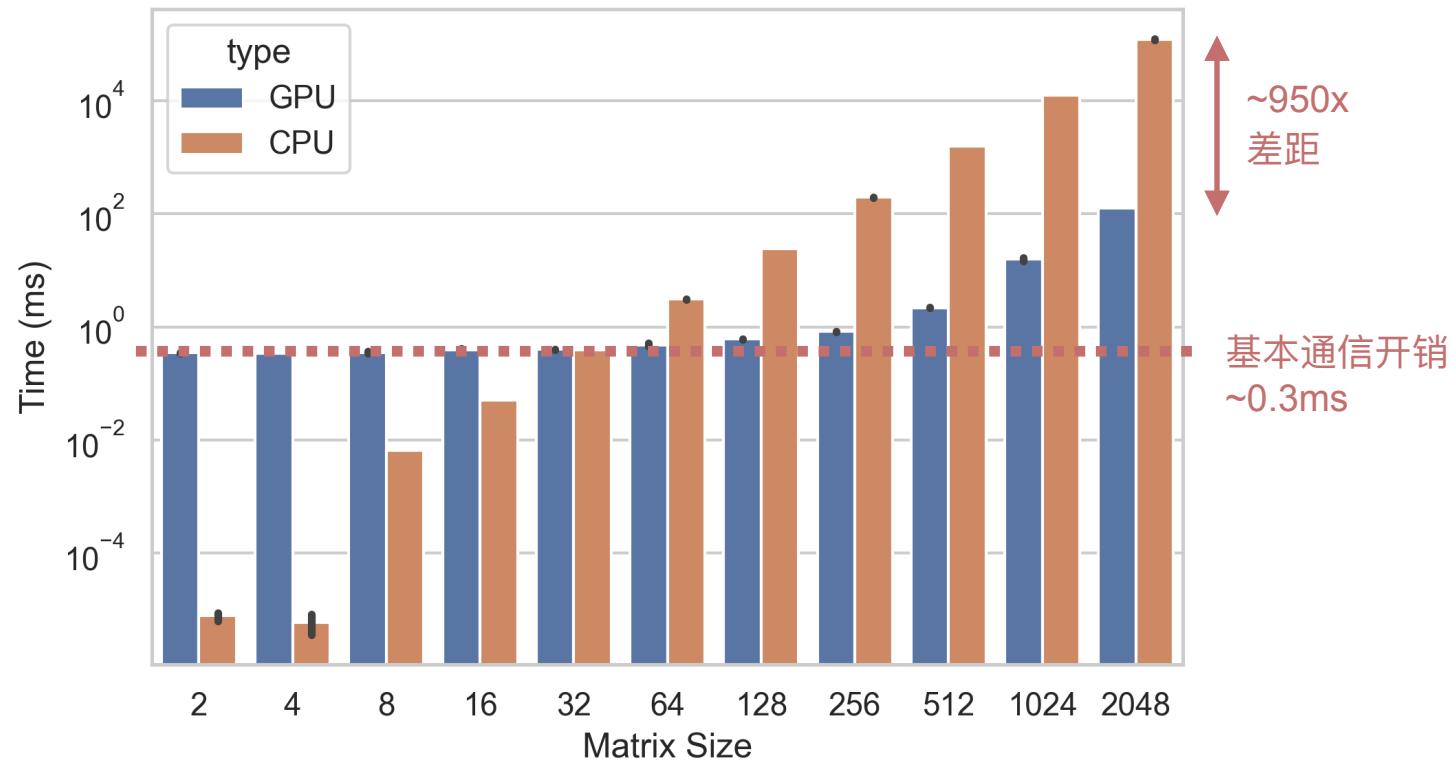
```
// Mul is a GPU version of math.Mat[T].Mul method and it multiplies
// two matrices m1 and m2 and returns the result.
func Mul[T math.Type](m1, m2 math.Mat[T]) math.Mat[T] {
    ...
    out := device.MakeBuffer(nil, uintptr(math.TypeSize[T]()*m1.Row*m2.Col), ①
    mtl.ResourceStorageModeShared)
    ...

    // 4. Copy data from GPU buffer to CPU buffer
    data := make([]T, m1.Row*m2.Col)
    copy(data, unsafe.Slice((*T)(out.Content()), m1.Row*m2.Col)) ②
    return math.Mat[T]{
        Row:  m1.Row,
        Col:  m2.Col,
        Data: data,
    }
}
```

# 例子1：矩阵运算



# 例子1：矩阵运算



## 例子2：图像处理

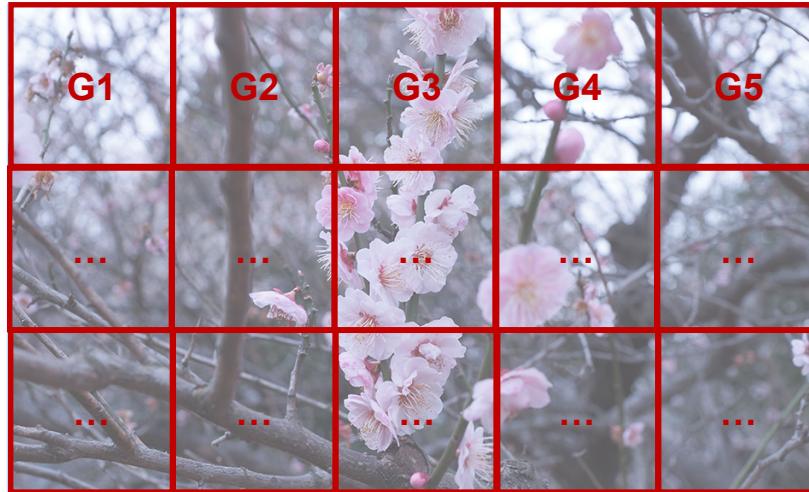


原始图像

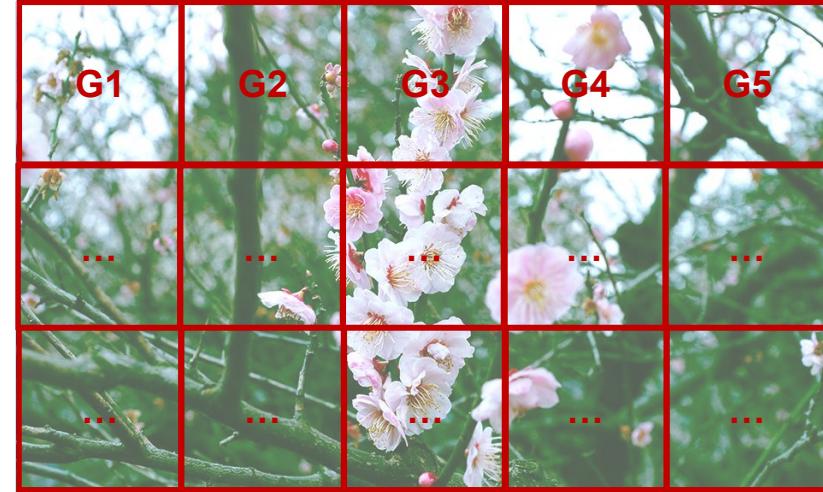


处理后

## 例子2：图像处理中并行的粒度



原始图像



处理后

CPU 计算并行的粒度 = CPU 缓存行大小的一半

GPU 计算并行的粒度 = 像素级并行

## 例子2：图像处理

```
kernel void procPixel(device const float* img      [[ buffer(0) ]],  
                      device      float* out      [[ buffer(1) ]],  
                      device const params& params [[ buffer(2) ]],  
                      uint          index     [[thread_position_in_grid]]) {  
    float brightness = clamp(params.brightness) - 0.5;  
    float contrast = clamp(params.contrast) - 0.5;  
    float saturation = clamp(params.saturation) - 0.5;  
    float temperature = clamp(params.temperature) - 0.5;  
    float tint = clamp(params.tint) - 0.5;  
    float r = srgb2linear(img[index * 4 + 0]);  
    float g = srgb2linear(img[index * 4 + 1]);  
    float b = srgb2linear(img[index * 4 + 2]);  
    color c = color{r, g, b};  
    c = apply_temperature_tint(c, temperature, tint);  
    c = apply_brightness(c, brightness);  
    c = apply_contrast(c, contrast);  
    c = apply_saturation(c, saturation);  
    out[index * 4 + 0] = clamp(linear2srgb(c.r));  
    out[index * 4 + 1] = clamp(linear2srgb(c.g));  
    out[index * 4 + 2] = clamp(linear2srgb(c.b));  
    out[index * 4 + 3] = img[index * 4 + 3];  
}
```

1

2

## 例子2：图像处理



原始图像



处理后 (CPU)



处理后 (GPU)

432ms



6.39ms



~64x  
差距



# 为代码增加 GPU 的支持需要考虑什么？

时机	频率	考虑要点
初始化	仅一次	着色器编译
资源加载	少量	拷贝时间
指令编码	频繁	调度策略
资源共享	频繁	同步回调

# 大纲

- 与 GPU 进行交互的基本知识
- 在 Go 程序中支持 GPU 加速
- **使用 Go 进行 GPU 计算的挑战**
  - Cgo 的开销
  - 基础设施的积累
  - 设计 GPU 的统一抽象
  - 着色器的编写和调试
- 总结

# 挑战 1: Cgo 的成本

所有涉及显卡相关的操作或多或少涉及系统调用，最简单的方式就是引入 Cgo

```
/*
#cgo CFLAGS: -Werror -fmodules -x objective-c
#cgo LDFLAGS: -framework Metal -framework CoreGraphics
#include "mtl.h"
*/
import "C"
```

又例如: <https://github.com/go-gl/gl> 如 <https://github.com/go-gl/glfw>

# 挑战 1: Cgo 的成本

所有涉及显卡相关的操作或多或少涉及系统调用，最简单的方式就是引入 Cgo

```
/*
#cgo CFLAGS: -Werror -fmodules -x objective-c
#cgo LDFLAGS: -framework Metal -framework CoreGraphics
#include "mtl.h"
*/
import "C"
```

又例如: <https://github.com/go-gl/gl> 如 <https://github.com/go-gl/glfw>

这会导致很多问题的产生:

1. 无法交叉编译
2. 调用性能受损
3. 难以维护、non-reproducible build

# 挑战 1: Cgo 的成本

Windows 令人意外的可以不需要使用 Cgo:

```
var (
    LibGLESv2          = syscall.NewLazyDLL("libGLESv2.dll")
    _glCreateShader = LibGLESv2.NewProc("glCreateShader")
    ...
)
```

# 挑战 1: Cgo 的成本

近年来存在社区的工作开始推进去 cgo 化，通过汇编传递系统调用来消除 cgo

从而达到优化内存、调用机制以及支持交叉编译的目的

<https://github.com/ebitengine/purego>

```
libc, err := purego.Dlopen("/usr/lib/libSystem.B.dylib", purego.RTLD_NOW|purego.RTLD_GLOBAL)
if err != nil { panic(err) }
var puts func(string)
purego.RegisterLibFunc(&puts, libc, "puts")
```

## 挑战 2: 基础设施的匮乏

几乎没有成熟的框架\*

在社区内被人熟知的 GUI 框架包括: Fyne, Ebitengine, GioUI 相对完善 (但无人使用的) 3D 引擎 g3n/engine

## 挑战 2: 基础设施的匮乏

几乎没有成熟的框架\*

在社区内被人熟知的 GUI 框架包括: Fyne, Ebitengine, GioUI 相对完善 (但无人使用的) 3D 引擎 g3n/engine

但这些框架都有明显的局限性:

1. Fyne 整个底层渲染引擎依靠 OpenGL/ES, 而且只能进行 2D 渲染, 没有直接暴露给用户的 GPU 支持
2. Ebitengine 底层渲染通过 OpenGL+Metal 来对全平台进行支持, 依然没有公开 GPU 相关的 API
3. Gioui 在几乎所有主流标准之上设计了一套抽象, 但只设计了为 GUI 2D 渲染相关的抽象, 不适用于更复杂的 3D 场景或者纯计算场景
4. g3n/engine 依赖 OpenGL 实现渲染

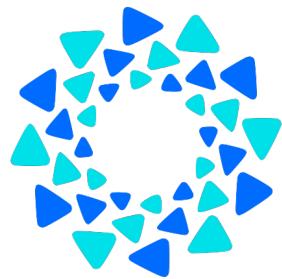
\*<https://github.com/changkun/awesome-go-graphics>

## 挑战 3: 缺少统一的抽象

几乎没有能够同时涵盖渲染和计算相关的统一抽象

设计这类抽象的难度不亚于重新设计一套标准，相对简单的方式是直接借鉴 WebGPU 的设计

社区内的热门 GUI 框架都有根据自身需求对 GPU 驱动进行不同级别的抽象



# 例子1: GUI 图形框架 Fyne 的图形驱动接口

```
package fyne

// Driver defines an abstract concept of a Fyne render driver.
// Any implementation must provide at least these methods.
type Driver interface {
    // CanvasForObject returns the canvas that is associated with a given CanvasObject.
    CanvasForObject(CanvasObject) Canvas

    // Device returns the device that the application is currently running on.
    Device() Device
    // Run starts the main event loop of the driver.
    Run()

    // StartAnimation registers a new animation with this driver and requests it be started.
    StartAnimation(*Animation)
    // StopAnimation stops an animation and unregisters from this driver.
    StopAnimation(*Animation)

    ...
}
```

## 例子2: GUI 图形框架 Ebitengine 的图形驱动接口



```
type Graphics interface {
    Initialize() error
    Begin() error
    End(present bool) error
    SetTransparent(transparent bool)
    SetVertices(vertices []float32, indices []uint16) error
    NewImage(width, height int) (Image, error)
    NewScreenFramebufferImage(width, height int) (Image, error)
    IsGL() bool
    IsDirectX() bool
    MaxImageSize() int

    NewShader(program *shaderir.Program) (Shader, error)

    // DrawTriangles draws an image onto another image with the given parameters.
    DrawTriangles(dst ImageID, srcts [graphics.ShaderImageCount]ImageID, shader ShaderID,
        dstRegions []DstRegion, indexOffset int, blend Blend, uniforms []uint32, evenOdd bool) error
}
```

# 例子3: GUI 图形框架 GioUI 的图形驱动接口



```
type GPU interface {
    Release()
    // Frame draws the graphics operations from op into a viewport of target.
    Frame(frame *op.Ops, target RenderTarget, viewport image.Point) error
    ...
}

// Device represents the abstraction of underlying GPU APIs such as OpenGL,
// Direct3D useful for rendering Gio operations.
type Device interface {
    BeginFrame(target RenderTarget, clear bool, viewport image.Point) Texture
    EndFrame()
    NewComputeProgram(shader shader.Sources) (Program, error)
    NewVertexShader(src shader.Sources) (VertexShader, error)
    NewFragmentShader(src shader.Sources) (FragmentShader, error)
    NewPipeline(desc PipelineDesc) (Pipeline, error)
    BeginCompute()
    EndCompute()
    DispatchCompute(x, y, z int)
    Release()
    ...
}
```

# 挑战 4: 着色器的编写和调试

着色器语法多为 C/C++ 的变种，且不能使用常规的手段进行调试

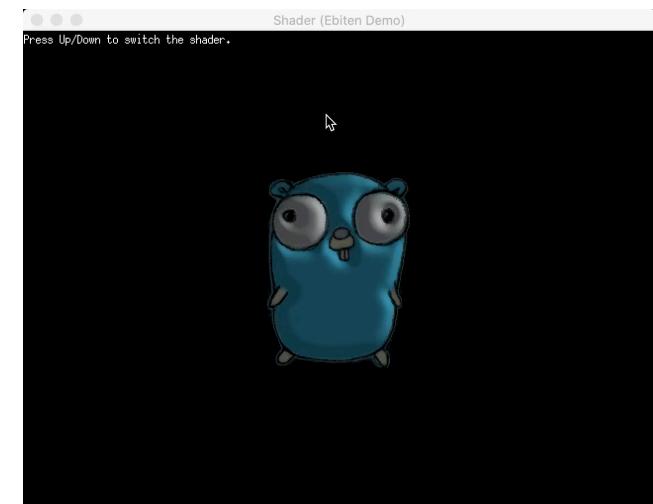
目前社区内唯一见到对着色器有定制的是 Ebitengine 的 Kage 语法

```
package main

// Uniforms
var Time float
var Cursor vec2
var ScreenSize vec2

// Fragment is the entry point of the fragment shader.
func Fragment(position vec4, texCoord vec2, color vec4) vec4 {
    lightpos := vec3(Cursor, 50)
    lightdir := normalize(lightpos - position.xyz)
    normal := normalize(imageSrc1UnsafeAt(texCoord) - 0.5)
    ambient := 0.25
    diffuse := 0.75 * max(0.0, dot(normal.xyz, lightdir))

    return imageSrc0UnsafeAt(texCoord) * (ambient + diffuse)
}
```



# 大纲

- 与 GPU 进行交互的基本知识
- 使用 Go 进行 GPU 计算的挑战
- 在 Go 程序中支持 GPU 加速
- 总结

# 总结与展望

- 目前 Go 语言本身只能够充当任务派发以及资源调度的作用
- 实际的 GPU 计算需要通过编写图形管线的着色器来执行
- Go 语言社区存在少量的扩展 Go 语言语法来实现着色器编写，并通过转译器翻译成目标平台的着色器代码
- Go 语言在 GPU 计算的基础设施还存在巨大的空白和机遇
  - 统一的抽象：消除平台间的差异
  - 根据计算任务规模自动选择计算架构：对任务的执行规模进行推断
  - 扩展 Go 语言的着色器语法
    - 通过 `//go:gpu` 标注函数可以作为着色器函数使用
    - 通过编译器自动分析一个函数是否支持直接转译到着色器函数，从而避免手动重新实现
  - debug 和 profiling 的工具

# 进一步阅读的参考资料

<https://developer.apple.com/documentation/metal>

<https://github.com/changkun/gopherchina2023gogpu>

<https://github.com/polyred/polyred>

<https://git.sr.ht/~eliasnaur/gio>

<https://github.com/fyne-io/fyne>

<https://github.com/hajimehoshi/ebiten>

<https://github.com/changkun/awesome-go-graphics>

<https://github.com/tinne26/kage-desk>

# Go on GPU

欧长坤

[changkun.de/s/gogpu](http://changkun.de/s/gogpu)

GopherChina 2023

Session "Foundational Toolchains"

2023 June 10