Go on GPU

**Changkun Ou**

changkun.de/s/gogpu

GopherChina 2023
Session “Foundational Toolchains”
2023 June 10
Agenda

- Basic knowledge for interacting with GPUs
- Accelerate Go programs using GPUs
- Challenges in Go when using GPUs
- Conclusion and outlooks
Agenda

● Basic knowledge for interacting with GPUs
  ○ Motivation
  ○ GPU Driver and Standards
  ○ Render and compute pipeline
  ○ Vulkan/Metal/DX12/OpenGL

● Accelerate Go programs using GPUs

● Challenges in Go when using GPUs

● Conclusion and outlooks
Motivation of GPU Acceleration

Improve system computation performance

Increase amount of concurrency

Processing large amount of data

Machine learning, deep learning, graphics rendering, etc.
CPU Architecture

- L2 Cache
- Exec Unit
  - L1 Cache
- Exec Unit
  - L1 Cache
- Exec Unit
  - L1 Cache
- Exec Unit
  - Cache
- L2 Cache
- Command Queue
- Controller
- RAM
- Disk
GPU Architecture

- **SIMD Exec Unit**
- **Cache**
- **Texture**
- **Tessellate**
- **Culling Rasterizer**
- **Z-buffer**
- **Work Scheduler**
- **Command Queue**

**Cores for executing shader programs (programmable), in parallel**

**Graphics-specific fixed functions (non-programmable) and compute resources**
Collaboration between CPU and GPU: Render Scenario

CPU and GPU works together as a leader-follower relationship, CPU is responsible for preparing the GPU work and relevant resources, and GPU is responsible for the execution.

For instance, in a gaming scenario, we often aim for 30 fps and GPU renders per 33.3ms.
Graphics Rendering Pipeline

Rendering pipeline focuses on executing Render Passes. In each of the rendering pass, the associated render task can be customized using shaders. In GPU, except rasterizer unit, it also includes ray tracing unit.
Collaboration between CPU and GPU: Compute Scenario

As general purpose computing involves, GPU starts to detach from rendering pipeline, and can be considered as an external computing resource. CPU can schedule compute tasks to GPU at any time.
Compute Pipeline
Command Submission Model

Command encoders convert API commands into hardware commands

Hardware commands stored in command buffers

Depending on the task, there are different types of command encoders (render, compute, blit)

Command buffer can be created on different threads, they are explicitly and concurrently submitted to command queue
Graphics Standards

Conventional Gold Standards: OpenGL series

A successor: Vulkan

Khronos Groups cannot consider all realistic requirements in all manufactures

OS designers starts to customize and design their own standards

Web platform builds standard on top them
Agenda

- Basic knowledge for interacting with GPUs

- **Accelerate Go programs using GPUs**
  - Overall workflow of GPU acceleration
  - Two examples
  - Design Questions to Consider

- Challenges in Go when using GPUs

- Conclusion and outlooks
Basic Workflow to Add GPU Acceleration in Go Program

1. **Initialization** (device, shader compile)
2. **Prepare Resources needed for GPU**
3. **Encode compute commands in Command Encoders**
4. **Submit**
5. **Wait Until Finish**

---

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];
}
Example 1: Matrix Multiplication

Matrix multiplication is almost the fundamental compute unit for many modern scientific computation, it is also a classic performance improvement problem to solve.

Example: Feedforward propagation in neural network is done via matrix multiplication; many other linear solvers rely on matrices, etc.

\[
\begin{pmatrix}
4 & 5 \\
5 & 2 \\
\end{pmatrix}
\times
\begin{pmatrix}
4 \\
2 \\
\end{pmatrix}
= 
\begin{pmatrix}
4 \\
2 \\
\end{pmatrix}
\]
Example 1: Matrix Multiplication

Matrix multiplication is almost the fundamental compute unit for many modern scientific computation, it is also a classic performance improvement problem to solve.

Example: Feedforward propagation in neural network is done via matrix multiplication; many other linear solvers rely on matrices, etc.

\[
\begin{array}{c}
\begin{array}{c}
4x5 \\
\hline
5x2 \\
\hline
4x2 \\
\end{array}
\end{array}
\times
\Rightarrow
\begin{array}{c}
\begin{array}{c}
4x2 \\
\hline
\end{array}
\end{array}
\]
Example 1: Matrix Multiplication

Matrix multiplication is almost the fundamental compute unit for many modern scientific computation, it is also a classic performance improvement problem to solve.

Example: Feedforward propagation in neural network is done via matrix multiplication; many other linear solvers rely on matrices, etc

\[
\begin{bmatrix}
\end{bmatrix}
\times
\begin{bmatrix}
\end{bmatrix}
=
\begin{bmatrix}
\end{bmatrix}
\]
Example 1: Matrix Multiplication

Matrix multiplication is almost the fundamental compute unit for many modern scientific computation, it is also a classic performance improvement problem to solve.

Example: Feedforward propagation in neural network is done via matrix multiplication; many other linear solvers rely on matrices, etc.
Example 1: Matrix Multiplication

Matrix multiplication is almost the fundamental compute unit for many modern scientific computation, it is also a classic performance improvement problem to solve.

Example: Feedforward propagation in neural network is done via matrix multiplication; many other linear solvers rely on matrices, etc.
Example 1: Matrix Multiplication

Matrix multiplication is almost the fundamental compute unit for many modern scientific computation, it is also a classic performance improvement problem to solve.

Example: Feedforward propagation in neural network is done via matrix multiplication; many other linear solvers rely on matrices, etc.
Example 1: Matrix Multiplication

Matrix multiplication is almost the fundamental compute unit for many modern scientific computation, it is also a classic performance improvement problem to solve.

Example: Feedforward propagation in neural network is done via matrix multiplication; many other linear solvers rely on matrices, etc.
Example 1: Matrix Multiplication

Matrix multiplication is almost the fundamental compute unit for many modern scientific computation, it is also a classic performance improvement problem to solve.

Example: Feedforward propagation in neural network is done via matrix multiplication; many other linear solvers rely on matrices, etc.

\[
\begin{pmatrix}
\begin{array}{cccc}
\text{ } & \text{ } & \text{ } & \text{ } \\
\text{ } & \text{ } & \text{ } & \text{ } \\
\text{ } & \text{ } & \text{ } & \text{ } \\
\text{ } & \text{ } & \text{ } & \text{ }
\end{array}
\end{pmatrix}
\times
\begin{pmatrix}
\begin{array}{cc}
\text{ } & \text{ } \\
\text{ } & \text{ } \\
\text{ } & \text{ } \\
\text{ } & \text{ }
\end{array}
\end{pmatrix}
= \begin{pmatrix}
\begin{array}{c}
\text{ } \\
\text{ } \\
\text{ } \\
\text{ }
\end{array}
\end{pmatrix}
\]

4x5 \quad \times \quad 5x2 \quad = \quad 4x2
Example 1: Matrix Multiplication

// 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
}
Example 1: Matrix Multiplication

```go
// 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 MulNaive(m 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
}
```
Example 1: Matrix Multiplication

```go
import "changkun.de/x/gopherchina2023gogpu/gpu/mtl" // Metal driver

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()

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

1. Initialize device and command queue
2. Compile and initialize GPU functions
Example 1: Matrix Multiplication

```go
import "changkun.de/x/gopherchina2023gogpu/gpu/mtl" // Metal driver

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()

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

Changkun Ou. 2023. Go on GPU. GopherChina 2023. Session "Foundational Toolchains"
Example 1: Matrix Multiplication

```go
// 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()

    ...
Example 1: Matrix Multiplication

```c
#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;
}
```

Changkun Ou. 2023. Go on GPU. GopherChina 2023. Session "Foundational Toolchains"
Example 1: Matrix Multiplication

```cpp
#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;
}
```

# Example 1: Matrix Multiplication

```cpp
#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 const 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;
}
```

Example 1: Matrix Multiplication

// 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()
    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)

    ...
}
Example 1: Matrix Multiplication

// 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},
        mtl.Size{Width: 1, Height: 1, Depth: 1})
    ce.EndEncoding()
    cb.Commit()
    cb.WaitUntilCompleted()
    ...
Example 1: Matrix Multiplication

// 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),
    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))
    return math.Mat[T]{
        Row:  m1.Row,
        Col:  m2.Col,
        Data: data,
    }
}
Example 1: Matrix Multiplication

![Graph showing matrix multiplication time for GPU and CPU across different matrix sizes.](image)
Example 1: Matrix Multiplication

~950x Faster
Basic costs ~0.3ms
Example 2: Image Processing

Original

After Processing

Image source: Yuki Koyama
**Example 2: Image Processing’s Parallelization Granularity**

<table>
<thead>
<tr>
<th>G1</th>
<th>G2</th>
<th>G3</th>
<th>G4</th>
<th>G5</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Original Image" /></td>
<td><img src="image2.png" alt="Original Image" /></td>
<td><img src="image3.png" alt="Original Image" /></td>
<td><img src="image4.png" alt="Original Image" /></td>
<td><img src="image5.png" alt="Original Image" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>G1</th>
<th>G2</th>
<th>G3</th>
<th>G4</th>
<th>G5</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="After Processing Image" /></td>
<td><img src="image2.png" alt="After Processing Image" /></td>
<td><img src="image3.png" alt="After Processing Image" /></td>
<td><img src="image4.png" alt="After Processing Image" /></td>
<td><img src="image5.png" alt="After Processing Image" /></td>
</tr>
</tbody>
</table>

**CPU Compute Granularity** = Half of CPU Cache Line Size

**GPU Compute Granularity** = Pixel Level Parallelization
Example 2: Image Processing' Kernel Function

```c
kernel void procPixel(  
  device const float* img,  
  device float* out,  
  device const params& params,  
  uint index)  
{
  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];
}
```
Example 2: Image Processing

Original
Before Processing (CPU)
After Processing (GPU)

432ms
6.39ms
~64x Faster

Image source: Yuki Koyama
## What to Consider When Adding GPU Acceleration?

<table>
<thead>
<tr>
<th>When?</th>
<th>How Frequent?</th>
<th>Design Consideration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>Only once</td>
<td>Shader Compilation</td>
</tr>
<tr>
<td>Resource Loading</td>
<td>Less often</td>
<td>Memory Copy</td>
</tr>
<tr>
<td>Command Encoding</td>
<td>Frequent</td>
<td>Scheduling Strategy</td>
</tr>
<tr>
<td>Resource Sharing</td>
<td>Frequent</td>
<td>Sync Callback</td>
</tr>
</tbody>
</table>
Agenda

● Basic knowledge for interacting with GPUs
● Accelerate Go programs using GPUs

● Challenges in Go when using GPUs
  ○ Costs of Cgo
  ○ Fundamental infrastructure
  ○ Common abstraction
  ○ Writing and debugging shaders

● Conclusion and outlooks
Challenge 1: The cost of Cgo

Operations on GPU involves system calls, and easiest approach is to use Cgo:

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

For other examples: https://github.com/go-gl/gl and https://github.com/go-gl/glfw
Challenge 1: The cost of Cgo

Operations on GPU involves system calls, and easiest approach is to use Cgo:

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

For other examples: [https://github.com/go-gl/gl](https://github.com/go-gl/gl) and [https://github.com/go-gl/glfw](https://github.com/go-gl/glfw)

However, the use of Cgo will cause more problems to solve:

1. Cannot easily do cross compilation
2. Call performance drops
3. Hard to maintain, non-reproducible build
Challenge 1: The cost of Cgo

Surprisingly Windows platform do not need use Cgo:

```go
var (
    LibGLESv2    = syscall.NewLazyDLL("libGLESv2.dll")
    _glCreateShader = LibGLESv2.NewProc("glCreateShader")
    ...)
```
Challenge 1: The cost of Cgo

Recently, there are work exist in the community starts to remove the need of ego, and start to use assembly to pass arguments to system call directly, the project is:

https://github.com/ebitengine/purego

```go
if err != nil { panic(err) }
var puts func(string)
purego.RegisterLibFunc(&puts, libc, "puts")
```
Challenge 2: Lack of Foundational Infrastructure

Nearly No mature framework*

There are well known GUI frameworks: Fyne, Ebitengine, GioUI and a 3D engine g3n/engine
Challenge 2: Lack of Foundational Infrastructure

Nearly No mature framework*

There are well known GUI frameworks: Fyne, Ebitengine, GioUI and a 3D engine g3n/engine

These frameworks have common limitations:

1. In Fyne, the underlying rendering depends on OpenGL/ES, and only support 2D rendering. No exposed APIs to users.
2. In Ebitengine, the underlying rendering uses OpenGL+DirectX+Metal but no exposed APIs to users.
3. GioUI is the most comprehensive approach but only designed 2D rendering abstraction. No exposed APIs to users.
4. g3n/engine is depending on OpenGL’s rendering

*https://github.com/changkun/awesome-go-graphics
Challenge 3: Lack of Common Abstraction

There are no common abstractions in the Go world

Design an abstraction is as challenge as designing a new standard
Example 1: Fyne’s Graphics Driver Interface

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)

    ...
}
Example 2: Ebitengine’s Graphics Driver Interface

```go
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, srcs [graphics.ShaderImageCount]ImageID, shader ShaderID, dstRegions []DstRegion, indexOffset int, blend Blend, uniforms []uint32, evenOdd bool) error
}
```
Example 3: GioUI’s Graphics Driver Interface

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()
  ...
}
Challenge 4: Writing and Debugging Shaders

Syntax of Shaders are based on variation of C/C++. It is not possible to debug in a regular manner.

The only community work that extends Go syntax to shader is Kage in Ebitengine.

```go
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)
lighdir := 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)
}
```

Agenda

- Basic knowledge for interacting with GPUs
- Accelerate Go programs using GPUs
- Challenges in Go when using GPUs
- Conclusion and outlooks
Conclusion and Outlook

- In Go, it is only possible to schedule and manage resources for GPU tasks.
- The actual computation need to utilize shaders.
- There is a little work in Go to extend its syntax for writing shaders.
- There is a large gap and huge opportunity in Go to support GPU computation infra:
  - Common abstraction: cross platform.
  - Automatic inference for architecture selection depending on compute workload.
  - Extend Go syntax to support shaders:
    - E.g., using `//go:gpu` to mark a function that can be a shader function.
    - E.g., Automatically analyze if a function can be used in shader.
  - Debug and profiling toolchain.
References

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

Changkun Ou

changkun.de/s/gogpu

GopherChina 2023
Session “Foundational Toolchains”
2023 June 10