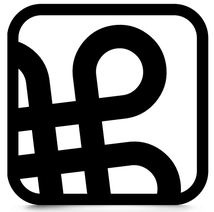


Macoun

GPU Datenverarbeitung mit Metal

Rolf Wöhrmann

Ablauf

- Motivation & Hintergrund
- Einführung in die Metal APIs
- Metal Shader Sprache
- Einbettung in die Gesamtanwendung
- Zusammenfassung

Motivation & Hintergrund

Motivation

- GPU als massiv paralleler Prozessor vs. CPU
- Anwendung in rechenintensiven Bereichen
  - Datenanalyse, Scientific Computing, Mustererkennung etc.
  - Insbesondere Augmented & Virtual Reality, Artificial Intelligence
- Optimal für parallelisierbare Algorithmen

Hintergrund

- Apples Einführung von Metal als Ablösung von OpenGL
- Vorgestellt auf WWDC 2014
- Proprietäre Hardware für iOS
- Fokus auf iOS, aber auch macOS Support

Minimum iOS Hardware

- Minimum Hardware:
  - iPhone 5S
  - iPad mini 2, iPad Air, iPad Pro
  - iPod touch 6th generation
- Info.plist UIRequiredDeviceCapabilities “metal”
- Nur arm64 Devices

Minimum Mac Hardware

- MacBook (Early 2015)
- MacBook Air (Mid 2012)
- MacBook Pro (Mid 2012)
- Mac mini (Late 2012)
- iMac (Late 2012)
- Mac Pro (Late 2013)

Einführung in die Metal APIs

“Hello Metal”

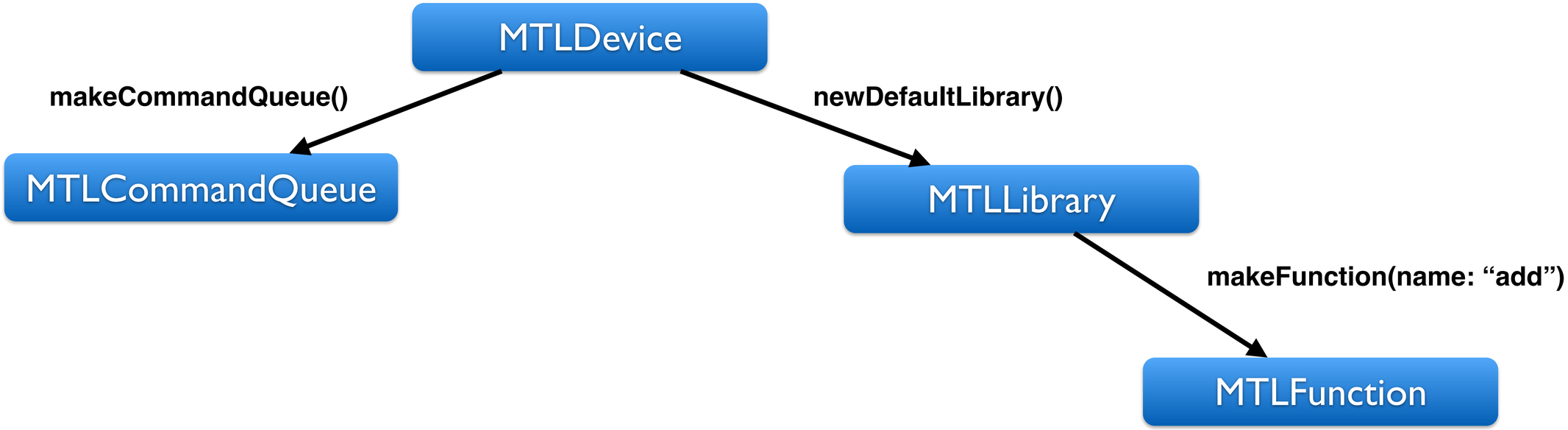
```
#include <metal_stdlib>
using namespace metal;


kernel void add(const device float *in [[ buffer(0) ]],
                device float *out [[ buffer(1) ]],
                uint id [[ thread_position_in_grid ]]) {

    out[0] = in[0] + in[1];
}
```

Demo

Grundlegende Klassen: Setup

- MTLDevice
- MTLCommandQueue
- MTLLibrary
- MTLFunction

Grundlegende Klassen: Setup



Grundlegende Klassen: Setup

```
import Foundation
import Metal

class MetalEngine {

    var device: MTLDevice
    var commandQueue: MTLCommandQueue
    var library: MTLLibrary
```

Grundlegende Klassen: MTLDevice

```
init?() {
    guard let _device = MTLCreateSystemDefaultDevice() else {
        print("No Metal support!")
        return nil
    }
    device = _device

    commandQueue = device.makeCommandQueue()

    guard let _library = device.newDefaultLibrary() else {
        print("Cannot create Metal library")
        return nil
    }
    library = _library
```

Grundlegende Klassen: MTLCommandQueue

```
init?() {
    guard let _device = MTLCreateSystemDefaultDevice() else {
        print("No Metal support!")
        return nil
    }
    device = _device

    commandQueue = device.makeCommandQueue()

    guard let _library = device.newDefaultLibrary() else {
        print("Cannot create Metal library")
        return nil
    }
    library = _library
```

Grundlegende Klassen: MTLLibrary

```
init?() {
    guard let _device = MTLCreateSystemDefaultDevice() else {
        print("No Metal support!")
        return nil
    }
    device = _device

    commandQueue = device.makeCommandQueue()

    guard let _library = device.newDefaultLibrary() else {
        print("Cannot create Metal library")
        return nil
    }
    library = _library
```

Grundlegende Klassen: MTLFunction

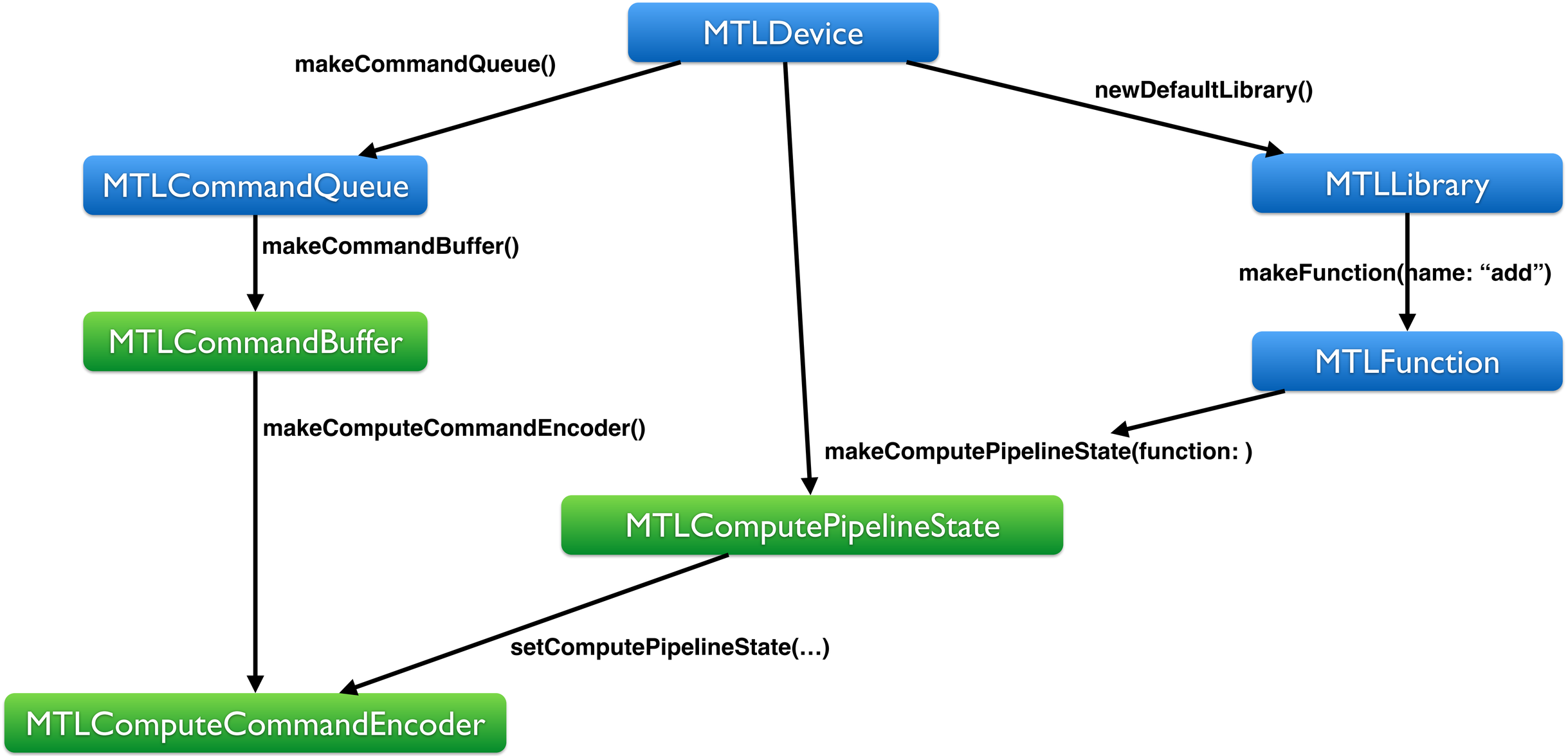
```
guard let _library = device.newDefaultLibrary() else {
    print("Cannot create Metal library")
    return nil
}
library = _library

guard let _addFunction = library.makeFunction(name: "add") else {
    print("Cannot create function")
    return nil
}
addFunction = _addFunction
```

Metal Shader & Xcode

- Xcode kompiliert alle *.metal Shader Files in eine DefaultLibrary
- Befindet sich als default.metallib File direkt im App Ordner
- Zeitersparnis gegenüber Runtime Kompilierung
- Einfache Handhabung im Projekt

Grundlegende Klassen: Commands

- MTLCommandBuffer
- MTLComputeCommandEncoder
- MTLComputePipelineState


MTLDevice
makeCommandQueue()
newDefaultLibrary()
MTLCommandQueue
MTLLibrary
makeCommandBuffer()
makeFunction(name: “add”)
MTLCommandBuffer
MTLFunction
makeComputeCommandEncoder()
makeComputePipelineState(function:)
MTLComputePipelineState
setComputePipelineState(…)
MTLComputeCommandEncoder

MTLCommandBuffer

```
func compute1(a: Float, b: Float) -> Float? {

    let commandBuffer = commandQueue.makeCommandBuffer()
    let commandComputeEncoder = commandBuffer.makeComputeCommandEncoder()

    var compState: MTLComputePipelineState

    do {
        try compState =
             device.makeComputePipelineState(function: addFunction)
    } catch {
        print("Cannot create ComputePipelineState")
        return nil
    }

    commandComputeEncoder.setComputePipelineState(compState)
```

MTLComputeCommandEncoder

```
func compute1(a: Float, b: Float) -> Float? {

    let commandBuffer = commandQueue.makeCommandBuffer()
    let commandComputeEncoder = commandBuffer.makeComputeCommandEncoder()

    var compState: MTLComputePipelineState

    do {
        try compState =
             device.makeComputePipelineState(function: addFunction)
    } catch {
        print("Cannot create ComputePipelineState")
        return nil
    }

    commandComputeEncoder.setComputePipelineState(compState)
```

MTLComputePipelineState

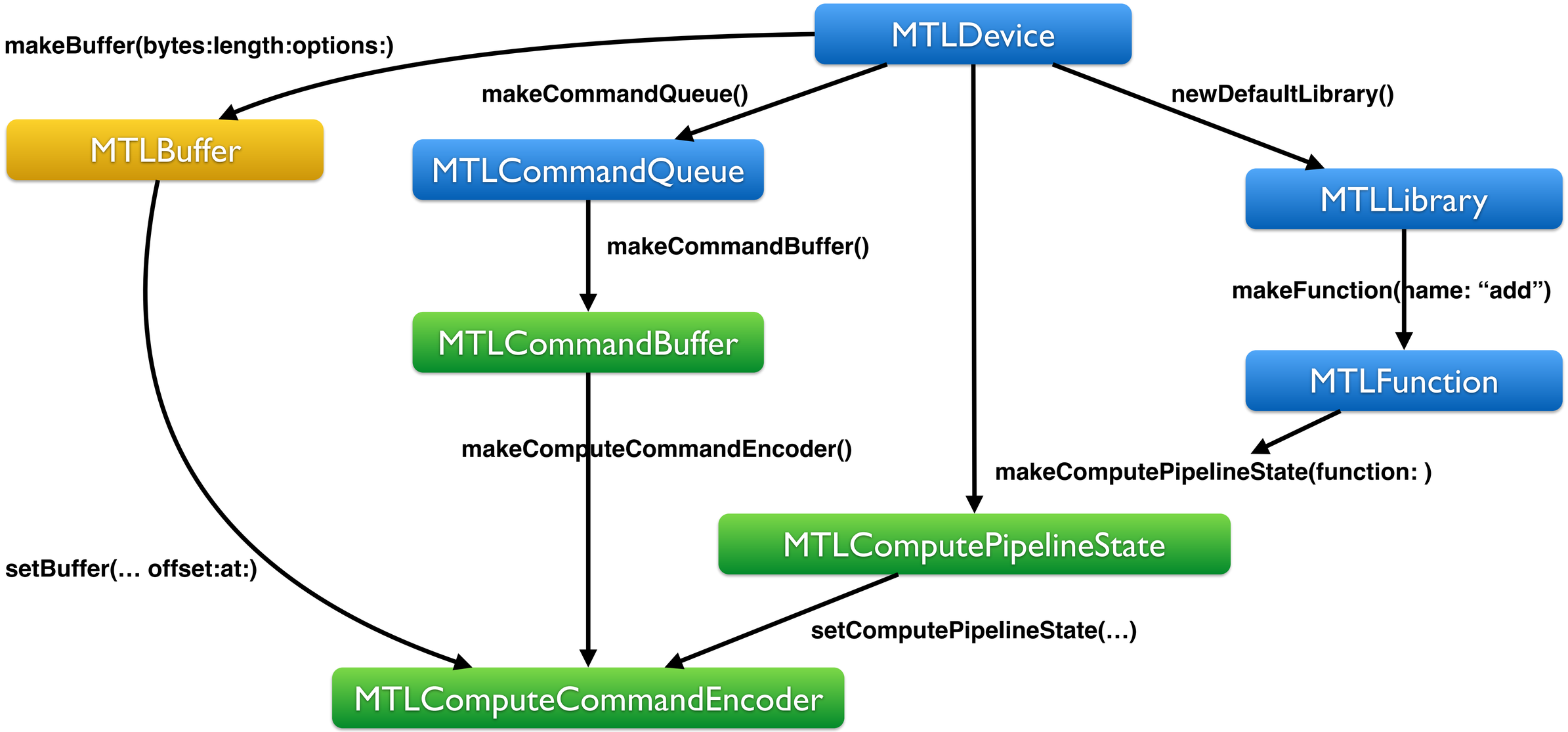
```
func compute1(a: Float, b: Float) -> Float? {

    let commandBuffer = commandQueue.makeCommandBuffer()
    let commandComputeEncoder = commandBuffer.makeComputeCommandEncoder()

    var compState: MTLComputePipelineState

    do {
        try compState =
            device.makeComputePipelineState(function: addFunction)
    } catch {
        print("Cannot create ComputePipelineState")
        return nil
    }

    commandComputeEncoder.setComputePipelineState(compState)
```

Grundlegende Klassen: Commands

```
func compute1(a: Float, b: Float) -> Float? {

    let commandBuffer = commandQueue.makeCommandBuffer()
    let commandComputeEncoder = commandBuffer.makeComputeCommandEncoder()

    var compState: MTLComputePipelineState

    do {
        try compState =
             device.makeComputePipelineState(function: addFunction)
    } catch {
        print("Cannot create ComputePipelineState")
        return nil
    }

    commandComputeEncoder.setComputePipelineState(compState)
```

Grundlegende Klassen: Daten

- MTLBuffer


MTLDevice
makeBuffer(bytes:length:options:)
makeCommandQueue()
newDefaultLibrary()
MTLBuffer
MTLCommandQueue
MTLLibrary
makeCommandBuffer()
makeFunction(name: “add”)
MTLCommandBuffer
MTLFunction
makeComputeCommandEncoder()
makeComputePipelineState(function:)
MTLComputePipelineState
setBuffer(… offset:at:)
setComputePipelineState(…)
MTLComputeCommandEncoder

MTLBuffer - Input

```
enum BufferIndex: Int {
    case Input = 0
    case Output = 1
}

var inData = [Float](repeating: 0.0, count:2)
inData[0] = a
inData[1] = b

let inBuffer = device.makeBuffer(bytes: &inData,
    length: MemoryLayout.size(ofValue: Float()) * 2,
    options: MTLResourceOptions.cpuCacheModeWriteCombined)

commandComputeEncoder.setBuffer(inBuffer, offset: 0,
   at: BufferIndex.Input.rawValue)
```

MTLBuffer - Input

```
enum BufferIndex: Int {
    case Input = 0
    case Output = 1
}

var inData = [Float](repeating: 0.0, count:2)
inData[0] = a
inData[1] = b

let inBuffer = device.makeBuffer(bytes: &inData,
    length: MemoryLayout.size(ofValue: Float()) * 2,
    options: MTLResourceOptions.cpuCacheModeWriteCombined)

commandComputeEncoder.setBuffer(inBuffer, offset: 0,
   at: BufferIndex.Input.rawValue)
```

MTLBuffer - Input

```
enum BufferIndex: Int {
    case Input = 0
    case Output = 1
}

var inData = [Float](repeating: 0.0, count:2)
inData[0] = a
inData[1] = b

let inBuffer = device.makeBuffer(bytes: &inData,
    length: MemoryLayout.size(ofValue: Float()) * 2,
    options: MTLResourceOptions.cpuCacheModeWriteCombined)

commandComputeEncoder.setBuffer(inBuffer, offset: 0,
   at: BufferIndex.Input.rawValue)
```

MTLBuffer - Output

```
var outData = [Float](repeating: 0.0, count:1)
let outDataBytes = MemoryLayout.size(ofValue: Float())

let outBuffer = device.makeBuffer(bytes: &outData,
  length: outDataBytes,
  options: MTLResourceOptions.cpuCacheModeWriteCombined)

commandComputeEncoder.setBuffer(outBuffer, offset: 0,
  at: BufferIndex.Output.rawValue)
```

MTLBuffer - Output

```
var outData = [Float](repeating: 0.0, count:1)
let outDataBytes = MemoryLayout.size(ofValue: Float())

let outBuffer = device.makeBuffer(bytes: &outData,
  length: outDataBytes,
  options: MTLResourceOptions.cpuCacheModeWriteCombined)

commandComputeEncoder.setBuffer(outBuffer, offset: 0,
  at: BufferIndex.Output.rawValue)
```

MTLBuffer - Output

```
var outData = [Float](repeating: 0.0, count:1)
let outDataBytes = MemoryLayout.size(ofValue: Float())

let outBuffer = device.makeBuffer(bytes: &outData,
  length: outDataBytes,
  options: MTLResourceOptions.cpuCacheModeWriteCombined)

commandComputeEncoder.setBuffer(outBuffer, offset: 0,
  at: BufferIndex.Output.rawValue)
```

Ausführung auf dem GPU

- Configure Dispatch Threadgroups
- Commit CommandBuffer
- Completion Handling
- Get Output Data

Dispatch Threadgroups

```
commandComputeEncoder.dispatchThreadgroups(
                          MTLSize(width: 1, height: 1, depth: 1),
  threadsPerThreadgroup: MTLSize(width: 1, height: 1, depth: 1))

commandComputeEncoder.endEncoding()

commandBuffer.commit()
commandBuffer.waitUntilCompleted()

let data = NSData(bytesNoCopy: outBuffer.contents(),
                          length: outDataBytes, freeWhenDone: false)
data.getBytes(&outData, length:outDataBytes)

return outData[0]
```

Dispatch Threadgroups

- 3D Grid von Threads
- Logische Größe des Grids aus Sicht Algorithmus
- Runtime Größe des Grids für einen Threadgroup
- Physikalische Anzahl der GPU Cores
- Entscheidender Einfluss auf Performanz

Commit & Wait

```
commandComputeEncoder.dispatchThreadgroups(
                            MTLSize(width: 1, height: 1, depth: 1),
  threadsPerThreadgroup: MTLSize(width: 1, height: 1, depth: 1))

commandComputeEncoder.endEncoding()

commandBuffer.commit()
commandBuffer.waitUntilCompleted()

let data = NSData(bytesNoCopy: outBuffer.contents(),
                          length: outDataBytes, freeWhenDone: false)
data.getBytes(&outData, length:outDataBytes)

return outData[0]
```

MTLCommandBuffer Execution API

```
- (void)enqueue;
- (void)commit;
- (void)addScheduledHandler:(MTLCommandBufferHandler)block;
- (void)waitUntilScheduled;
- (void)addCompletedHandler:(MTLCommandBufferHandler)block;
- (void)waitUntilCompleted;
```

Get Output Data

```
commandComputeEncoder.dispatchThreadgroups(
                            MTLSize(width: 1, height: 1, depth: 1),
  threadsPerThreadgroup: MTLSize(width: 1, height: 1, depth: 1))

commandComputeEncoder.endEncoding()

commandBuffer.commit()
commandBuffer.waitUntilCompleted()

let data = NSData(bytesNoCopy: outBuffer.contents(),
                          length: outDataBytes, freeWhenDone: false)
data.getBytes(&outData, length:outDataBytes)

return outData[0]
```

Metal Shader Sprache

Aufbau & Syntax

- Wie C++14 mit u.a. folgenden Ausnahmen:
  - Lambda Ausdrücke, rekursive Funktionen
  - New & Delete
  - Vererbung & virtuelle Funktionen
  - Exceptions, goto
  - register, thread_local
- … also fast allem, was an C++ Spaß macht :-)
- Vollständige Liste in Apples “Metal Shading Language Guide”

Function Qualifier

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


kernel void add(const device float *in [[ buffer(0) ]],
                device float *out [[ buffer(1) ]],
                uint id [[ thread_position_in_grid ]]) {

    out[0] = in[0] + in[1];
}
```

Function Qualifier: kernel, vertex, fragment

- Data Processing:
  - kernel
- 3D Graphik:
  - vertex
  - fragment

Address Space Qualifier

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


kernel void add(const device float *in [[ buffer(0) ]],
                device float *out [[ buffer(1) ]],
                uint id [[ thread_position_in_grid ]]) {

    out[0] = in[0] + in[1];
}
```

Address Space Qualifier

- Memory Spezifikation hat großen Einfluss auf Performanz
  - **device:** Austausch GPU & CPU via MTLBuffer
  - **threadgroup**: Speicherbereich einer Threadgroup
  - **thread**: Speicherbereich eines Thread
    - Default für lokale Variablen einer kernel Funktion
  - **constant:** Read-only Speicher

Attribute Qualifier:
buffer, texture, sampler, threadgroup

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


kernel void add(const device float *in [[ buffer(0) ]],
                device float *out [[ buffer(1) ]],
                uint id [[ thread_position_in_grid ]]) {

    out[0] = in[0] + in[1];
}
```

Kernel Function Attribute Qualifier for Input Arguments

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


kernel void add(const device float *in [[ buffer(0) ]],
                device float *out [[ buffer(1) ]],
                uint id [[ thread_position_in_grid ]]) {

    out[0] = in[0] + in[1];
}
```

Kernel Function Attribute Qualifier for Input Arguments

- thread_position_in_grid
- thread_position_in_threadgroup
- thread_index_in_threadgroup
- threadgroup_position_in_grid
- threads_per_grid
- threads_per_threadgroup
- threadgroups_per_grid
- thread_execution_width

Demo 2

Einbettung in die Gesamtanwendung

Einbettung in die Gesamtanwendung

- Empfehlungen
- Debugging und Profiling von Metal Code
- Weiterführende Infos

Empfehlungen (1)

- Nur eine pre-kompilierte MTLLibrary mit allen kernel Funktionen
- Frühzeitig initialisieren:
  - MTLDevice, MTLLibrary, MTLFunction, MTLCommandQueue
- Möglichst wiederverwenden:
  - MTLBuffer, MTLComputePipelineState

Empfehlungen (2)

- Resources möglichst komplett vor dem Processing laden
- MTLCommandBuffer CompletionHandler verwenden
- Verschiedenen Dispatch Threadgroup Kofigurationen ausprobieren
- Speicherklassen optimieren

Empfehlungen (3)

- On-CPU vs. On-GPU Processing optimieren:
  - Frequenz der Kernel-Ausführung an Frame-Rate orientieren
  - Datenvolumen für Input & Output Buffer
  - Parallelisierbarkeit des Algorithmus

- Weiterführende Empfehlungen in “Metal Best Practices” von Apple

Debugging & Profiling auf dem GPU?

- Xcode GPU Frame Debugger
- Label properties
- Metal System Trace in Instruments
- Was tun für Data Processing?
- Keine Debug Print Statements
- Dafür: Custom Diagnostic Data Buffer

Weiterführende Infos

- Metal Programming Guide
- Metal Shading Language Guide
- Metal Best Practices
- WWDC Videos 2014-2016
- https://developer.apple.com/metal/

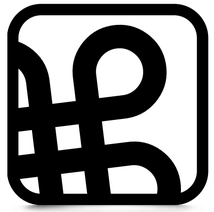
Zusammenfassung

Zusammenfassung

- GPU als massiv paralleler Prozessor vs. CPU
- Anwendung in rechenintensiven Bereichen
- Optimal für parallelisierbare Algorithmen
- Steile Lernkurve für API
- Dokumentation reichhaltig, aber mit Fokus auf 3D
- Gute Möglichkeit für Alleinstellungsmerkmale von iOS Apps

Fragen?

Vielen Dank



Macoun