# Apple GPU

## Apple M1架构

Apple没有公开其GPU架构，网上的资料也比较少。为了进行分析，需要结合逆向工作得到M1上的GPU指令集，然后基于指令来构造相关测试用例分析硬件细节。目前根据我们对已有资料的分析和总结，基于开源项目mesa，打造了一个能够在Apple M1上编译和运行OpenGL的框架。借助此框架，能够观察OpenGL程序编译出来的指令和执行过程，从而推断Apple GPU的一些基本特性。

### 名词解释

AGX: 苹果GPU的代号

Warp：线程束

Execution Unit：执行单元, 简称EU

[Arithmetic Logic Units](https://en.wikipedia.org/wiki/Arithmetic_Logic_Unit) ：算术逻辑单元，简称ALU

Stack Pointer：栈指针，简称SP

Program Counter：程序计数器，简称PC

Execution Mask：执行掩码，简称EM

General Purpose Register：普通寄存器，简称GPR

ThreadGroup：线程组，简称TG

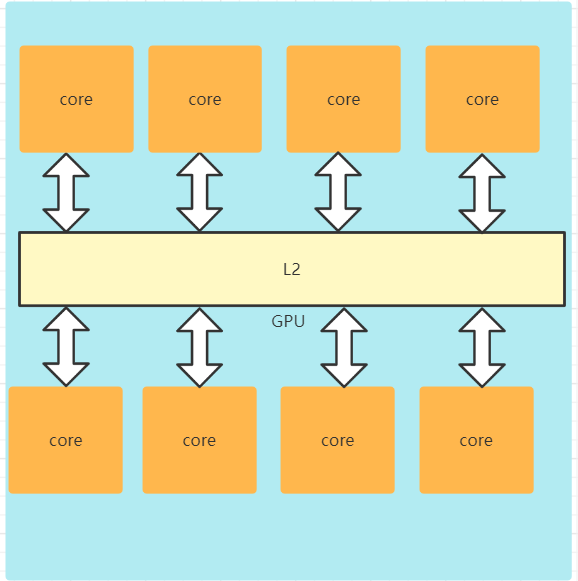
## G13 GPU架构

### 概述

Apple AGX GPU基于典型的SIMD架构打造，每个warp有32个线程，这32个线程也被称作SIMD Group。指令调度的基本单元是warp，也就是32个线程会执行同一条指令。每个warp会调度到一个Execution Unit上执行。每个Execution Unit包含8条ALU计算单元。如果一个ALU单元能在一个cycle内执行完一个FP32的计算，一个warp的FP32计算指令理论上能在4个cycle内执行完毕。

根据公开资料，M1 GPU一共有8个核，每个核包含16个EU，每个EU包含8个ALU。所以M1 GPU一共有128个EU，1024个ALU，最大可以允许24576个线程同时执行，即每个EU最多能保持 6个warp同时执行。

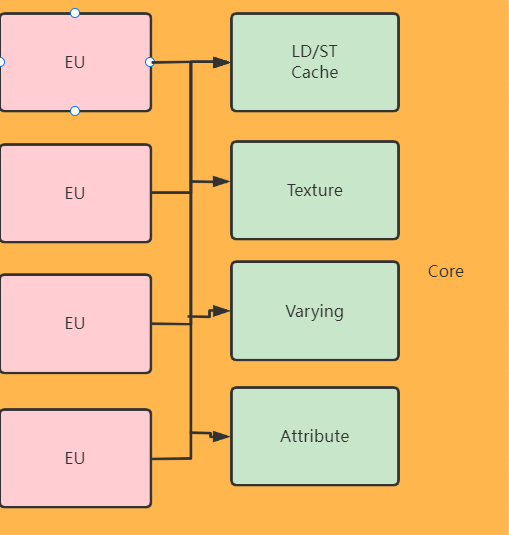
M1上每个线程能使用的最大寄存器数目是256 个WORD，对应的线程数目是384，据此可以推断出寄存器文件大小最少为384x256x2Bytes=192Kb。当线程数目为1024时，使用的寄存器数目是104，寄存器文件大小为104x1024x2Bytes=208Kb。所以每一个TG需要的寄存器文件有208Kb。

根据以上资料猜测出Apple的一种core之间的连接方式如下所示：

图表 1

图1所示了8个core通过L2 cache连接在一起，很容易看出来core可以根据需要进行扩展，比如m1 pro有16个core，m1 max有32个core，都可以继续往L2上面添加，Core内部的结构可以保持不变。

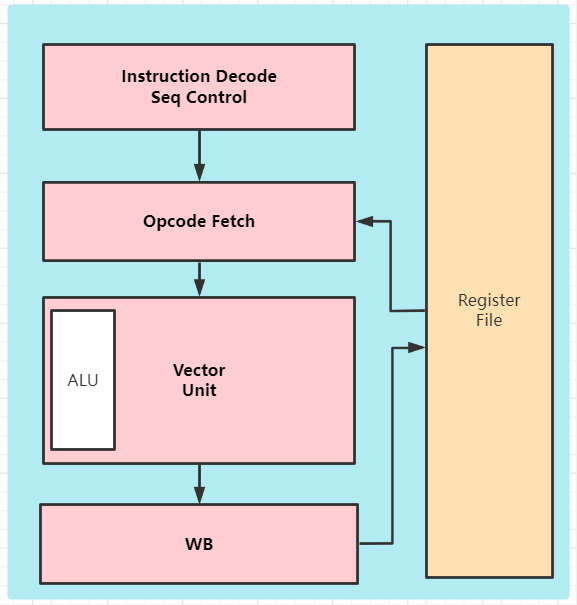
core内部的结构如下所示：



图表 2

图2所示为core内部的简要架构图，空间所限，只展示了4个EU，剩余的部分可以继续扩展。EU之间会share LD/ST，Texture，Varying，Attribute模块，L1 cache一般会配置为share memory或者普通的cache。

EU内部的架构图大概如下所示:



图表 3

图3所示是一个典型的4 state流水线，Vector Unit部分只画出了部分ALU，同样可以根据需求进行扩展。指令从Instruction Cache进入EU后，首先会在Instruction Decode模块进行解码，同时这个模块也会做指令顺序的控制。然后是Opcode fetch，得到每条指令的Opcode之后，进入ALU中进行运算，最后计算结果通过WB模块写入到寄存器中。

以上给出了GPU架构的主要模块，需要注意的是，公开的资料中并无L1和L2 cache的大小，以及内部的一些模块的细节，我们只是基于合理猜测给出一些基本框图，详细的数据需要进一步探索和测试。

### 架构特点

1. 生成的底层指令里面没有看到同步的逻辑，猜测硬件能够自动处理指令调度，能够大大减少编译器设计的复杂性。
2. 浮点ALU能够无缝进行clamps，saturate，negates, absolute 等计算。
3. 16bit和32bit之间的类型转换能够无缝进行。
4. ALU指令的cycle数不同。基于这点需要编译器在生成指令时评估指令周期。
5. 可能支持指令多发射，以及乱序执行。

### Apple M1 GPU指令集

指令集提供了针对硬件的可编程接口，它是对硬件的重要抽象。通过对指令集的研究能够深入了解硬件的细节。对指令进行高度抽象后，我们一般采用三地址码的方式来描述硬件指令：(操作符，结果，操作数1，操作数2)。

#### 寄存器

从功能上看，每个warp都有一个SP寄存器，一个PC寄存器，一个EM寄存器，以及128个32bit普通寄存器。所以程序中的变量能使用的寄存器范围就是r0-r127。低16bit使用r0l-r127l来访问。高16bit使用r0h-r127h来访问。如果需要访问64bit的变量，需要使用2个32bit的普通寄存器。

由于寄存器的数量有限，每个warp在运行的时候都被分配了一定数量的寄存器，如果程序中使用的寄存器数目超过了限制，那么超出的访问是无效的。编译器需要对这种情况进行处理，以保证使用的寄存器数目在合理的范围。

M1还有256个32bit常量寄存器，u0-u255。它们分为高位和低位进行访问。这些寄存器里面保存的值在所有线程里面都是相同的。

#### 寄存器缓存

M1 GPU上设计了寄存器缓存，能够保存最近使用过的寄存器内容，便于加速后面的访问。这种加速是通过在指令读写寄存器中编码hint标志位来完成的。Hint标志一般位于dst或者src操作数中，用于表明值会被再次用到。除此之外，还有个discard标志位，一般位于src操作数中，用于无效register cache中的数据，即读完所有操作数后，值不会被写回到寄存器文件。

#### 指令系统

M1 GPU指令属于可变长指令，从2 bytes到12 bytes都有可能。指令的编码长度通过bit L来标记，如果L是0，表示指令的最后2或者4字节需要跳过。其中12Bytes长度的指令需要跳过4字节，其他的跳过2字节。所有指令的编码都是以little endian从右到左表示。

#### 指令类型

目前我们已知的M1 GPU指令有74条，使用这些指令，能够完成一些从简单到复杂的场景渲染或者计算，然而这并不能代表M1 GPU上的所有指令。我们按照类型，将已知的指令罗列如下。

##### Mov指令

Mov指令主要完成从source到dest的数据搬运，source可以是一个立即数，也可以是寄存器。主要有下面3种：

1. mov dst， imm16 // 48 bit
2. mov dst， imm32 // 64 bit
3. get\_sr dst，special reg // 32 bit

##### 整数算术指令

整数算术指令主要完成整数的加减乘除运算，以及整数之间的类型转换。

1. iadd dst，src1， src2 // 64 bit
2. imadd dst，src1，src2，src3 // 64 bit
3. convert dst，src // 48 bit

##### 移位指令

移位指令主要完成算术/逻辑左移和右移

1. bfi dst，src1，src2，src3 // 64 bit
2. bfeil dst，src1，src2，src3 // 64 bit
3. extr dst，src1，src2，src3 // 64 bit
4. shlhi dst，src1，src2，src3 // 64 bit
5. shrhi dst，src1，src2，src3 // 64 bit
6. asr dst，src1，src2 // 64 bit
7. asrh dst，src1，src2 // 64 bit

##### 位操作指令

位操作指令主要完成与非或，以及位反转等操作。

1. bitop dst, src1, src2 // 48 bit
2. bitrev dst，src1 // 48 bit
3. popcount dst, src1 // 48 bit
4. ffs dst, src1 // 48 bit

##### 浮点运算指令

浮点运算指令主要完成浮点数的加减乘除以及special function相关计算。

1. fmadd dst，src1，src2，src3 // 64 bit
2. fmadd16 dst，src1，src2，src3 // 64 bit
3. fadd dst，src1，src2 // 48 bit
4. fadd16 dst，src1，src2 // 48 bit
5. fmul dst，src1，src2 // 48 bit
6. fmul16 dst，src1，src2 // 48 bit
7. floor dst，src1 // 48 bit
8. ceil dst，src1 // 48 bit
9. trunc dst，src1 // 48 bit
10. rint dst，src1 // 48 bit
11. rcp dst，src1 // 48 bit
12. rsqrt dst，src1 // 48 bit
13. rsqrt\_special dst，src1 // 48 bit
14. sin\_pt\_1 dst，src1 // 48 bit
15. sin\_pt\_2 dst，src1 // 48 bit
16. log2 dst，src1 // 48 bit
17. exp2 dst，src1 // 48 bit
18. dfdx dst，src1 // 48 bit
19. dfdy dst，src1 // 48 bit

##### 控制流指令

控制流指令主要完成程序执行流程控制逻辑，比如程序中断、返回、跳转和函数调用等等。

1. ret dst // 16 bit
2. stop // 16bit
3. trap // 16 bit
4. call dst // 16 bit
5. jump\_incomplete // 32 bit
6. jmp\_exec\_any // 48 bit
7. jmp\_exec\_none // 48 bit
8. call // 48 bit

##### SIMD执行流指令

Simd执行流指令主要用来进行simd的执行控制，保证不同线程能执行不同的代码逻辑。

1. pop\_exec dst // 48 bit
2. if\_icmp dst，src1，src2，src3 // 48 bit
3. if\_fcmp dst，src1，src2，src3 // 48 bit
4. while\_icmp dst，src1，src2，src3 // 48 bit
5. while\_fcmp dst，src1，src2，src3 // 48 bit
6. else\_icmp dst，src1，src2，src3 // 48 bit
7. else\_fcmp dst，src1，src2，src3 // 48 bit

##### 选择指令

选择指令主要用来完成整数和浮点数的比较选择

1. fcmpsel dst，src1，src2，src3，src4，src5 // 80 bit
2. icmpsel dst，src1，src2，src3，src4，src5 // 80 bit

##### SIMD ballot指令

Ballot指令主要完成每个active 线程的结果比较和设置

1. icmp\_balloc dst，src1，src2，src3 // 64 bit
2. icmp\_quad\_balloc dst，src1，src2，src3 // 64 bit
3. fcmp\_balloc dst，src1，src2，src3 // 64 bit
4. fcmp\_quad\_balloc dst，src1，src2，src3 // 64 bit
5. simd\_shuffle dst，src1，src2 // 48 bit
6. simd\_shuffle\_down dst，src1，src2 // 48 bit

##### 内存操作指令

内存操作指令主要完成内存的读取和写入，以及同步等操作

1. wait // 16 bit
2. ld/st\_tile dst // 64 bit
3. ld\_var dst // 64 bit
4. uniform\_store dst，src1 // 64 bit
5. device\_load dst，src1，src2 // 64 bit
6. device\_store dst，src1，src2 // 64 bit
7. stack\_store dst，src1 // 64 bit
8. stack\_load dst，src1 // 64 bit
9. stack\_get\_ptr dst // 64 bit
10. stack\_adjust // 64 bit
11. threadgroup\_load dst，src1，src2 // 64 bit
12. threadgroup\_store dst，src1，src2 // 64 bit
13. texture\_sample dst，src1，src2 // 64 bit
14. texture\_load // 64 bit
15. threadgroup\_barrier // 16 bit

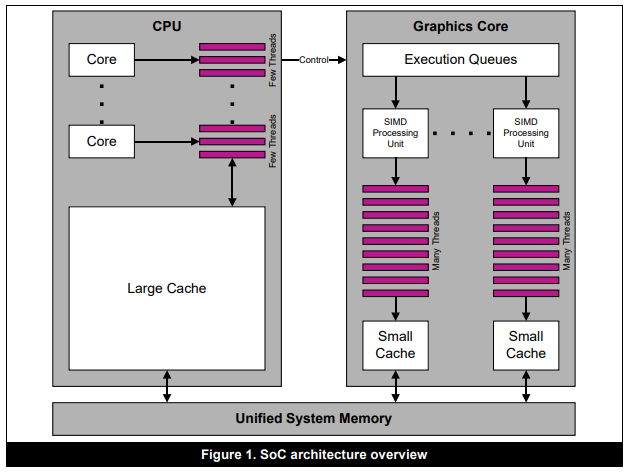
# 移动端GPU介绍

**GPU**是**Graphics Processing Unit的简称**，中文名字是图形处理单元。它的功能最初与名字一致，是专门用于绘制图像和处理图元数据的特定芯片，后来渐渐加入了其它很多功能，比如用于科学计算和人工智能任务。下面我们主要介绍GPU的图形功能。

## 移动SOC架构

GPU从设备上来说又可分为桌面端和移动端，桌面端主要是为了PC服务，移动端主要是为手机等移动产品服务，这两者的架构又完全不同。

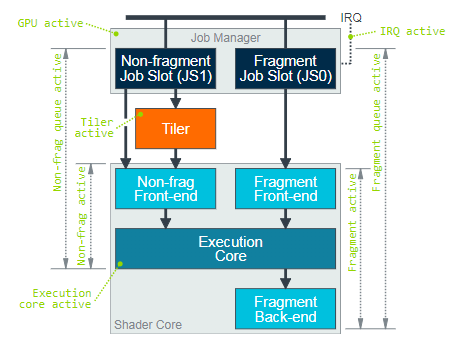
移动端的GPU一般是集成在一块SOC上面，对性能和功耗都有比较高的要求。一个典型的SOC架构如下图所示：



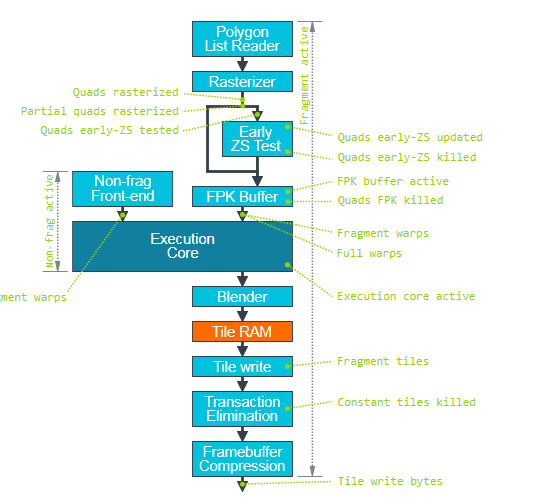
左边是CPU，右边是GPU，CPU可以向GPU发送控制命令，来驱动GPU运行。GPU的核心实现在SIMD Processing Unit，也就是向量计算核，也有部分厂商把他称作SIMT，这两者有些细微差别。

## Mali GPU架构

Mali GPU是典型的tile base架构，总体上包括Non Fragment部分和Fragment部分。Non Fragment主要用来处理顶点，Fragment部分主要用来处理像素。基于unified shading的架构，Non Fragment和Fragment最终都会使用Shader Core模块进行计算，减少了不同workload造成的shader core stall。

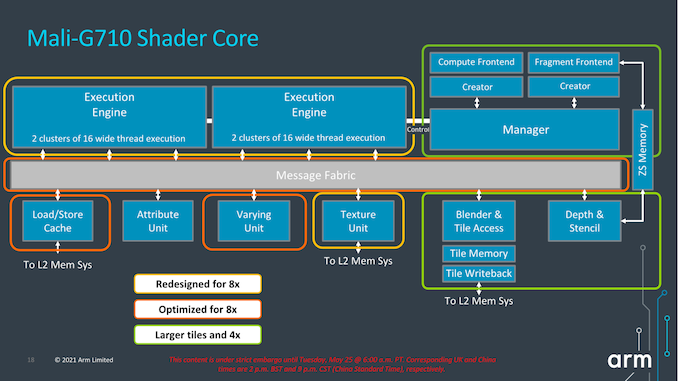


上图中所有作业统一由Job manager调度，他负责将不同的工作负载分配到GPU的不同调度单元上。该Job manager有两个调度队列公开给外部driver使用。Non Fragment队列可以处理vertex shading, tiling，geometry shading, tessellation shading, and compute shading工作。Fragment队列用于处理光栅化、EarlyZ、FPK、Fragment shading、Blender、Tile write等工作，具体处理流程如下图所示。



基于以上架构，在做性能测试分析时，需要对Fragment和Non Fragment分开处理，找到瓶颈。

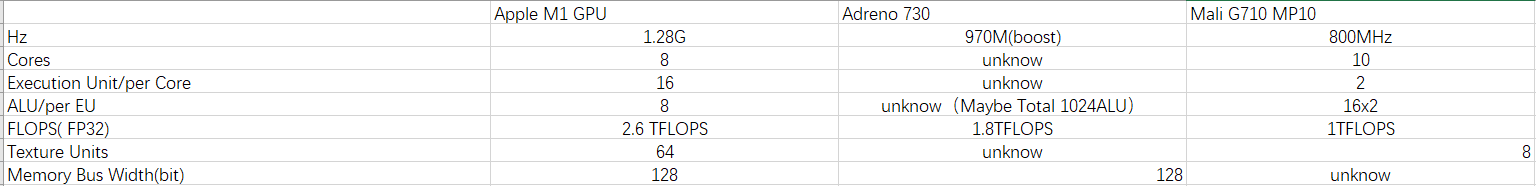
### Mali G710架构



图表 4 G710 core

Arm mali G710每个core内部有2个Execution Engine，相比Valhall 架构，shadercore的性能提升一倍。

### GPU Cores对比



图表 5 GPU Core对比

通过图5的对比可以发现，M1的GPU选择了较多的EU，mali选择了较大的ALU，两者都能够共享core外的其他硬件资源，core越大，共享的资源越多，计算能力越强。选择较多的EU能够进一步加强调度的灵活性，对于计算量较小的任务更加友好。

除此之外可以看出M1的texture unit是arm的8倍，这也能够提升游戏场景中越来越多的纹理贴图操作。

# MESA架构介绍

Mesa，也称为 Mesa 3D 和 Mesa 3D 图形库，是 OpenGL、Vulkan 和其他图形 API 规范的开源软件实现，包括了完整的驱动，编译器以及软件模拟器实现。基于mesa，能够在各种硬件平台上运行OpenGL程序，并且能够有限利用硬件本身提供的加速卡，比如GPU，来加速OpenGL程序的执行。如果硬件本身没有加速卡实现，那么Mesa也能够利用软件模拟器来运行OpenGL程序。

## Mesa编译

m1上编译mesa有两种方式：

1. 编译arm64版本
2. 编译x86\_64版本

这两种的编译配置文件稍微有所不同，我们以arm版本为准进行编译。

### 环境设置

配置zsh，保证运行的是arm版本的brew。

export PATH=/opt/homebrew/bin:$PATH

export LD\_LIBRARY\_PATH=/opt/homebrew/lib

然后安装x11，xext等arm64版本的包到/opt/homebrew

### 编译选项

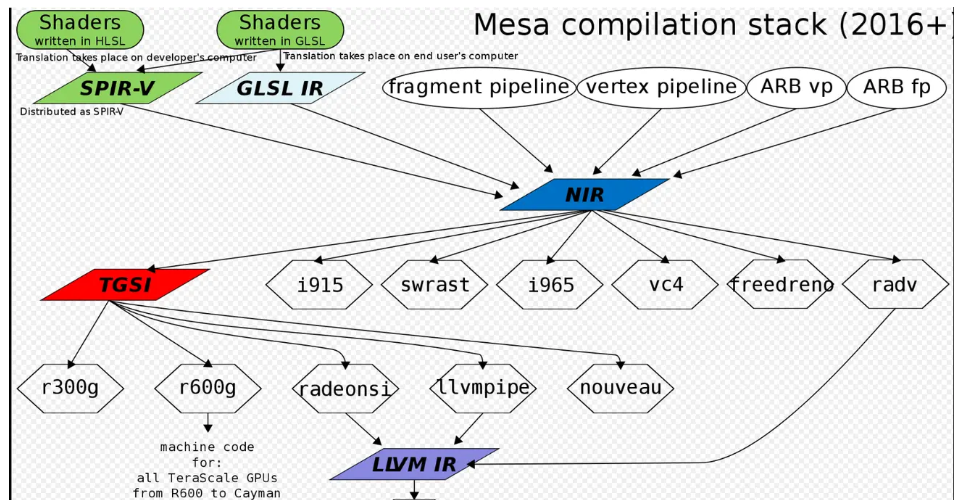
1. Gallium-driver：[asahi, swrast]
2. Glx: [gallium-xlib]

配置完成后，使用ninja进行编译。

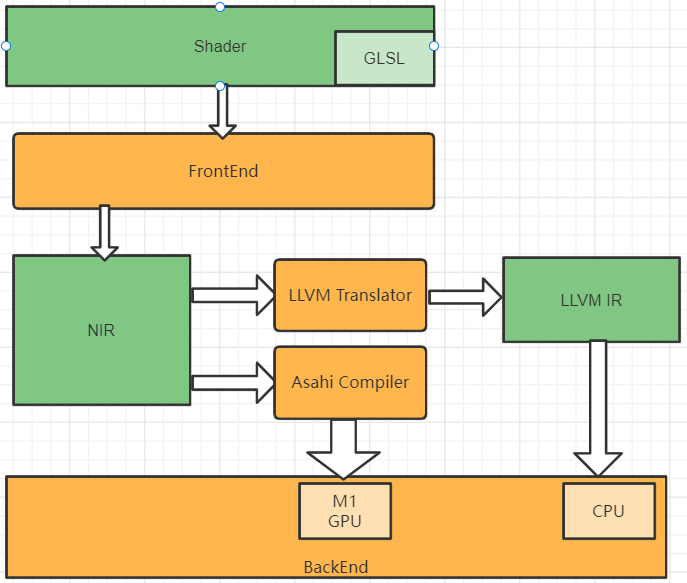
## Mesa 编译架构介绍

### NIR中间描述语言

不同的图形API使用了不同的shading launage，常见的有GLSL，HLSL，以及SPIRV，为了支持这些不同的语言，Mesa提出了一种统一的描述语言：NIR。通过将各种不同的shader编译到NIR来建立一个统一的平台，然后根据不同的硬件把NIR翻译到对应的硬件平台指令。



图表 6 NIR架构图



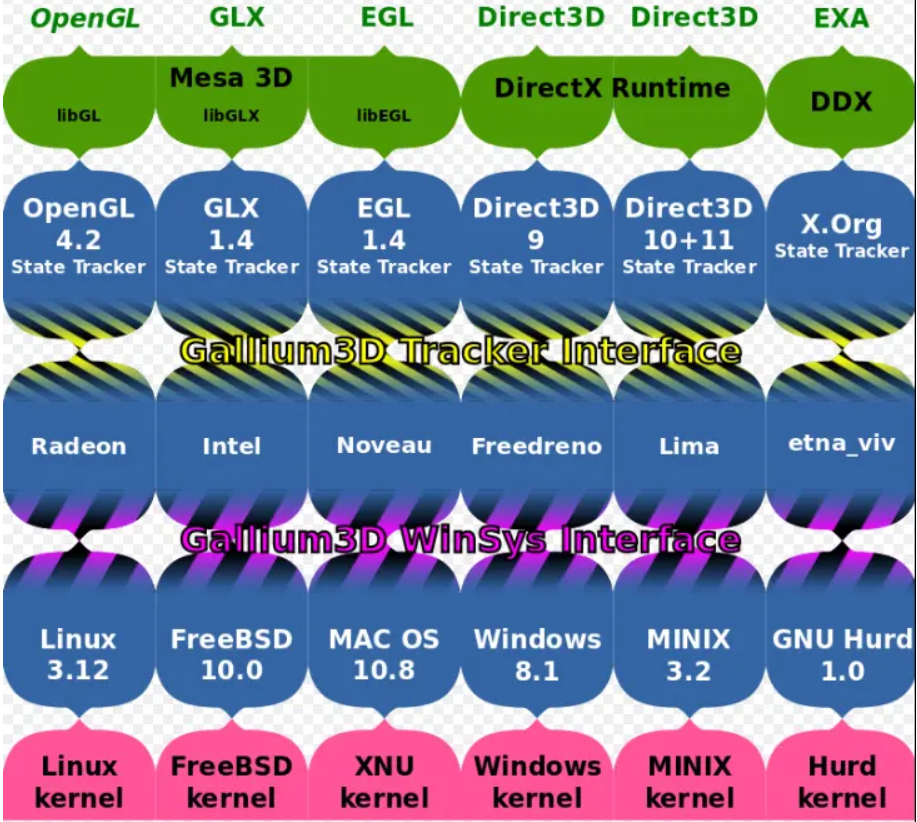
图表 7

上图是mesa编译OPENGL的架构图，基于此流程我们可以将OPENGL编译为在M1上运行的binary，用来探索Apple GPU的特性。

## Mesa驱动框架

为了简化GPU的驱动开发，mesa提出了Gallium开发框架。在mesa之前，驱动的开发都是由硬件厂商来完成，不同的设备厂商需要针对不同的操作系统开发不同的驱动，整个过程比较繁琐和冗长，很难做到完善支持。

为了解决上述问题，mesa提出了分层的概念，把device driver抽象成上下两部分，OS Winsys抽象了当前的操作系统，使得同一个device driver可以用于不同的操作系统。API State Tracker部分抽象了上层不同的GL接口，可以保证不同的接口都使用同一个驱动程序，大大降低了驱动的开发和适配工作。



图表 8 Gallium分层

从图6可以看出，Mesa经过分层后，往上由state tracker来对接不同的API，往下由winsys来对接不同的系统，完成了对系统和API的统一。