

# 图形处理器(GPU)的历史、现状和展望

以Nvidia为例

## Contents

---

### 1 引言

### 2 图形处理中的GPU

- 2.1 从诞生到可编程
- 2.2 传统GPU架构
  - 2.2.1 顶点处理器
  - 2.2.2 片元装配和光栅化
  - 2.2.3 片段处理器
  - 2.2.4 像素引擎 (Pixel Engine) 和光栅操作 (Raster Operation)
- 2.3 Tesla架构
  - 2.3.1 流水线组件
  - 2.3.2 纹理处理簇
- 2.4 Fermi架构
  - 2.4.1 更完备的TPC
  - 2.4.2 曲面细分
- 2.5 Turing 架构
  - 2.5.1 Mesh Shading
  - 2.5.2 RT Core
  - 2.5.3 深度学习超采样 (DLSS)

### 3 通用计算中的GPU

- 3.1 可编程管线时代的通用计算
- 3.2 Tesla管线时代的通用计算
  - 3.2.1 流式多处理器的并行原理
  - 3.2.2 CUDA (Compute Unified Device Architecture)
- 3.3 Turing管线时代及之后的通用计算
  - 3.3.1 Tensor Core

### 4 展望

## 1 引言

GPU（Graphics Process Unit，图形处理单元）是一种专门的电子元件，最初用于视频卡上或嵌入在主板、手机、个人计算机和工作站上以加速计算机图形和图像处理工作。在图像处理领域，最初，GPU主要用于图形渲染和加速计算机游戏的图形效果。它们专注于处理三角形网格、纹理映射、光照等图形相关任务。随着技术的进步，GPU引入了可编程着色器，如顶点着色器和像素着色器。这允许开发人员对图形渲染的各个阶段进行自定义处理，从而实现更高级的图形效果。GPU的发展推动了更多高级渲染技术的出现，例如几何着色器、细分着色器、计算着色器等。这些技术提供了更多的灵活性和效果，使得图形渲染更加逼真和真实。

随着人们对算力的不断提高，GPU的并行结构也逐渐被人广泛用于更多任务中去。人们开始探索将GPU用于通用计算任务，称为通用GPU（GPGPU）。GPU的并行处理能力和高带宽内存使其在许多科学计算、数据处理和机器学习等领域表现出色。而如CUDA、OpenCL等通用GPU编程框架的出现，使并行计算的开发变得更简单。最近几年，GPU在人工智能和深度学习方面取得了巨大成功。深度神经网络的训练和推断任务可以高度并行化，GPU的强大计算能力使其成为进行大规模深度学习的首选工具。

NVIDIA 是一家全球领先的图形处理器和人工智能计算技术公司，其在显卡领域的发展经历了多个阶段和重要里程碑，从而在图形性能、光线追踪、并行计算和深度学习领域都有着最前沿的技术和产品，拓展了应用领域。是二十世纪最成功的公司之一。

本文将以英伟达的显卡产品发展历程为例，总结图形处理器的发展历程，概述图形处理器的主要功能原理和应用，展望未来可能的发展。

## 2 图形处理中的GPU

在显卡诞生之前，人们在电脑上对三维世界的探索需要巧妙地操纵2D图形，通过缩放贴图对象、一行一行地扭曲像素来创造一种有深度的幻觉。当处理器的计算能力变得越来越强大，多边形网格才真正被引入，这对动画/游戏行业来说有着革命性的影响。在GPU诞生之前，这些多边形都是在CPU上处理的。20世纪末出现了许多3D加速卡，可以代替CPU，将传入的顶点数据和纹理光栅化为2D像素显示在屏幕上。但在此之前CPU仍然需要创建所有对象的可见列表，并将3维空间的物体变换到2位坐标。这个时候的CPU尚未进入多核时代，在创建场景之外，CPU仍然需要在单独的线程中执行游戏逻辑和底层操作系统相关的其它后台任务。

### 2.1 从诞生到可编程

在1999年8月31日，英伟达推出了GeForce 256，并首次提出了“GPU”的概念：“集成了变换，光照，三角形设置/裁剪和渲染引擎的单芯片处理器，能够每秒至少处理1000万个多边形”，也首次将图形处理器提升到和硬盘、内存、CPU类似的地位。GeForce 256可以同时处理4个像素的光栅化和纹理着色，支持顶点变换和正方形环境贴图。虽然GeForce 256的主频只有120 MHz，支持的贴图数量只有4，甚至一开始使用的是频率为166 MHz 32MB的SDR显存（在后来又推出了32MB DDR版本的GeForce 256，读写速度的提高大幅提升了性

能），但它仍然是当时最快的图形处理器，对多边形的处理速度超过了当时最顶尖的CPU。

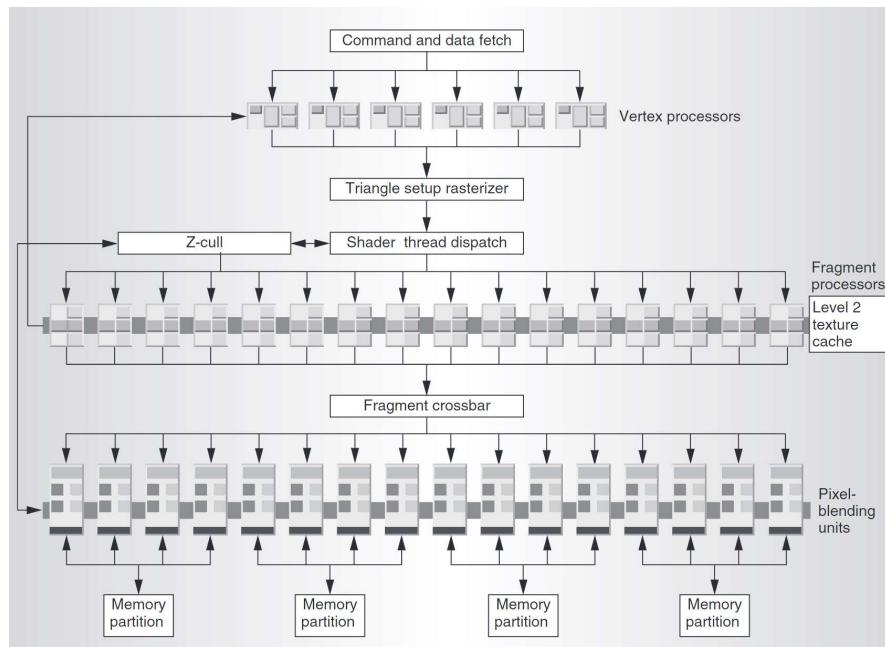
固定的渲染管线极大地限制了GPU的作用，所以提高可编程性成为了图形API和GPU发展的主要方向。在2002年，NVIDIA推出的GeForce 3加入了顶点着色器和可配置的片元管线，也拥有了经典的多重采样抗锯齿（Multisample Anti-aliasing）五点型抗锯齿（Quincunx Anti-aliasing）和功能。2003年推出的GeForce FX系列支持了32位可编程的片元着色器，也是NVIDIA第一代支持DX 9的硬件设备。

DirectX 9是21世纪初最广泛使用、最有影响力的图形API之一，后续NVIDIA发布的显卡都需要适用于DX 9的设计。直到GeForce 7800 GTX的发布之前，GeForce的主要升级都在于制程的提升、显存的大小和带宽提升、核心的频率和并行管线的数量增加。

## 2.2 传统GPU架构

GeForce 6800是这个时代NVIDIA最成功的GPU之一，我们可以从它的设计来总结传统的专注于图形处理的GPU的最终形态。

对于从命令流中获得的每个顶点，GPU会开辟一个线程运行一个独立的顶点着色器（对每个光栅化后的片段同理）。每个线程有专有的只写的输出寄存器，将结果传送给下一阶段。除了这些输入输出寄存器，每个线程也有私有的临时寄存器、只读的程序变量和访问过滤后的、重采样的纹理贴图的权限。

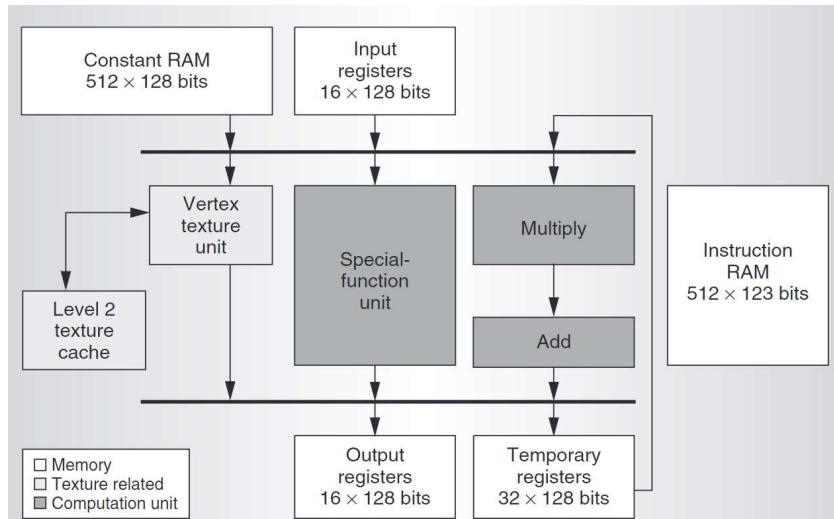


**Figure 1:** GeForce 6800 block diagram.

传统的GPU架构含有六个独立的可编程顶点处理器，读入的数据与指令可以分发给任一闲置的顶点处理器，所以并行性几乎是完美的。顶点阶段的处理结果按照应用既定的顺序被重新组装，送到三角形装配和光栅化单元。对于每个图元，光栅器找到组成的像素片段并将它们送到片段处理器。因为着色阶段所有处理器是彼此独立的，16个可编程的片段处理器可以完美地并行处理所有的任务。最后，交叉条将来自片段处理器的颜色和深度结果分发到十

六个固定功能的像素混合单元，这些单元执行帧缓冲操作，如颜色混合、抗锯齿和模板测试和更新。任一片段着色器的处理结果可以被送到任何位置的帧缓冲中。

### **2.2.1** 顶点处理器



**Figure 2:** GeForce 6800 Vertex Processor

顶点处理器可以执行非常大的指令字（为什么是123bit，很奇怪）。如上图所示，每个处理器的数据路径由一个向量乘加单元、一个标量特殊函数单元和一个纹理单元组成。向量单元可以同时执行四个数学运算。特殊函数单元实现如正弦、余弦对数等超越函数。计算单元可以从 $512 * 128$  bit的常量RAM、最大达 $32 * 128$ bit的临时寄存器，或是 $16 * 128$ bit的输入寄存器中获取操作数，将结果写入临时寄存器或是 $16 * 128$  bit的输出寄存器中。此外顶点处理器还支持实例化操作。

顶点处理器维护了一个与实现无关的编程模型，使用线程来使数据路径看起来具有统一的延迟，并使用计分板技术(scoreboarding)来隐藏纹理获取的延迟。顶点处理器的实现是完全多指令多数据(multiple instruction, multiple data, 简称MIMD)的。

为什么需要一个统一的延迟？因为可以为编程人员提供一个简单的抽象层，方便将计算任务分解为相同延迟进行的线程，使编程模型更易于理解与实现；可以隐藏不同指令和操作之间的  
真实延迟，可以简化程序的设计和分析；且线程间可以独立执行，不受其它线程的延迟影响，提高计算性能和吞吐量。

### 2.2.2 片元装配和光栅化

这一阶段的行为由API唯一确定，不要求可编程性，所以可以通过固定功能的单元高效地完成。输入是我们熟悉的齐次坐标向量。同时在这一阶段，装配好的图元需要进行一次视锥剔除以减少绘制数量。之后再进行视口变换和光栅化，得到一系列片段，输入片段处理器的数组。

页友好的光栅化顺序。第一次见到这个神奇的顺序还是在光线追踪实现的黑魔法上，通过这样的方式遍历像素可以获得很大的提升。没想到原来是这里先用上了。

值得注意的是，NVIDIA自GeForce 3系开始就在这一阶段引入了一个Z-Cull单元，以快速去除粗粒度上被遮挡的片元。而片段处理器阶段指向Z-Cull单元的目的正是更新其中缓存的部分深度值。

### 2.2.3 片段处理器

片段处理器接收输入的数据（位置、颜色、深度和10种通用的4\*FP32的属性），并对三角形的各项属性进行插值和纹理采样。GeForce 6800的片段处理器可以将结果输出给至多4个目标缓冲。和顶点处理器很类似，片段处理器也拥有常量/临时寄存器资源，和类似的MIMD功能。

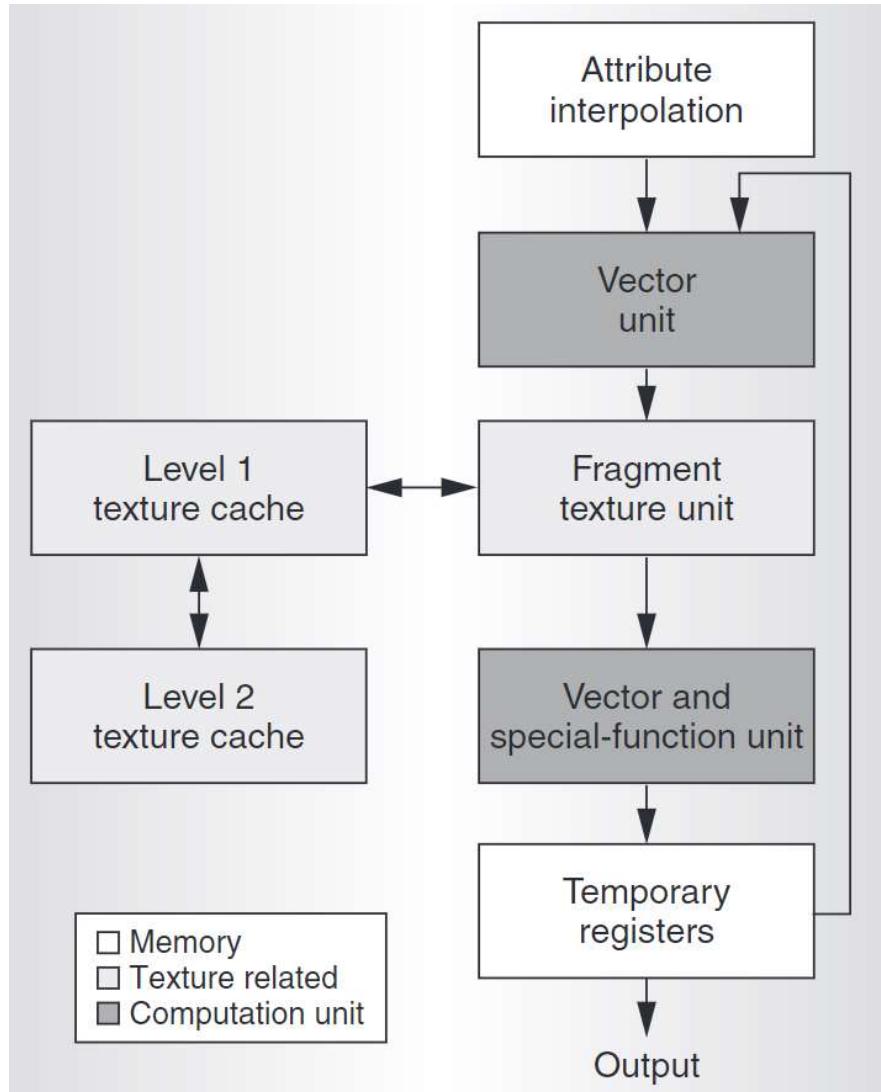


Figure 3: GeForce 6800 Fragment Processor

由于颜色成分中对Alpha值的不同处理（Alpha Component），fig. 3中计算单元被分成了两个部分，在“Vector and special-function unit”中支持各种不同的向量运算。每个片段处理器还支持流水线指令级并行，每个时钟周期内每像素可以执行最多6条DX 9指令。

纹理单元是片段着色器的重要组成部分，为了减小内存阻塞，纹理单元实现了管线式的二级缓存，并对纹理格式进行固定比率的有损压缩以提高细粒度

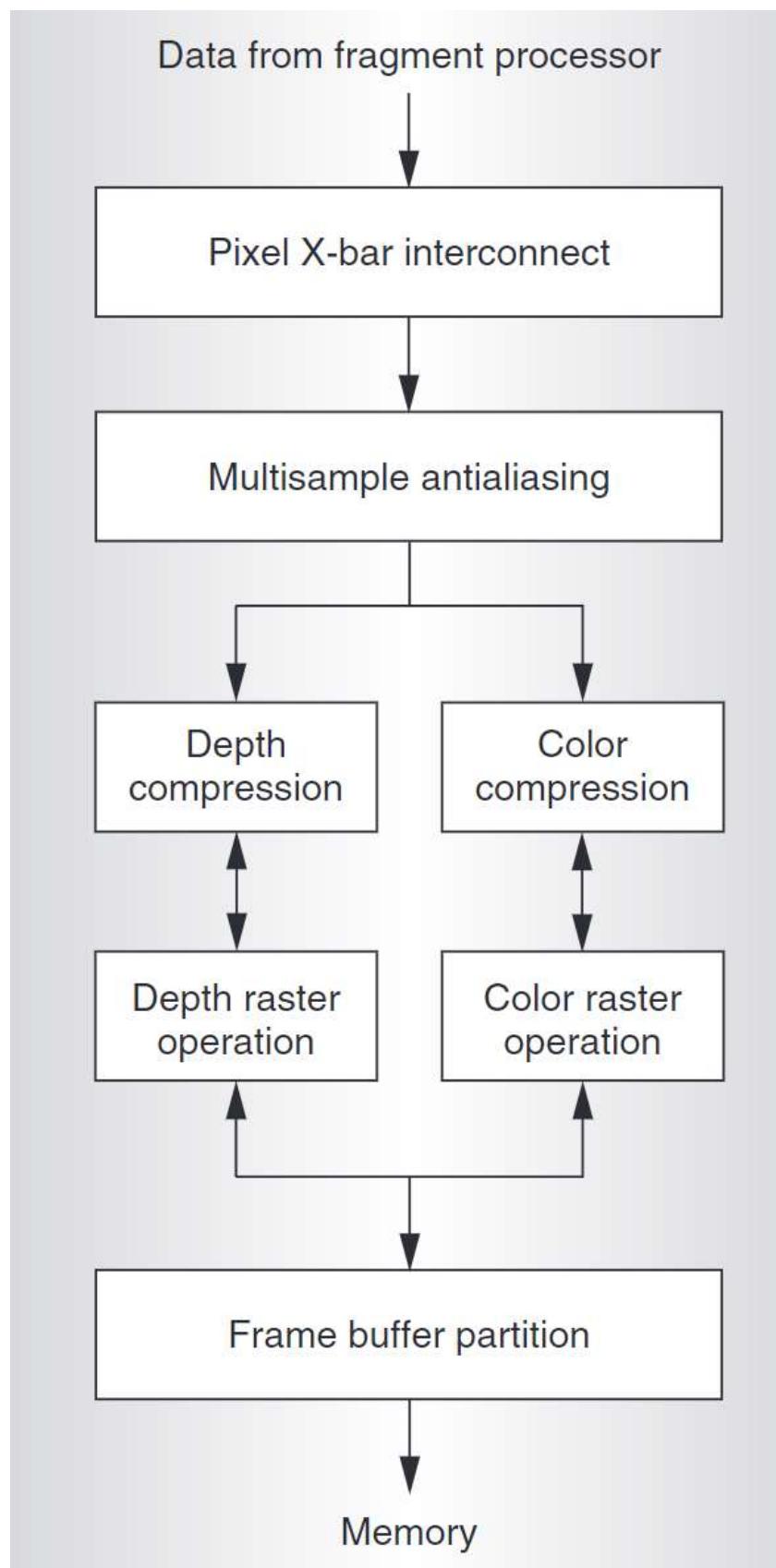
的访问和随机可寻址性。 GeForce 6800支持最近采样、双线性采样、三线性采样和各向异性滤波。

#### **2.2.4 像素引擎（Pixel Engine）和光栅操作（Raster Operation）**

GeForce 6800拥有16个像素引擎，这些固定功能的单元执行深度/模板检测和更新，以及颜色混合。

每个像素引擎连接到一个特定的内存分区，当抗锯齿启用时，像素引擎会将每个片段的颜色和深度信息扩展至多个样本。GeFORCE 6800支持MSAA和SSAA。SSAA会为每个子样本重新执行片段着色器以得到准确颜色，而MSAA需要对额外的遮盖性样本进行类似的深度测试和写入。

为什么要把深度测试放到最后。看到过的很有意思的讨论。从前面的架构介绍可以看到，顶点着色器和片段着色器几乎是百分百并行。而深度测试是一个原子性操作，涉及到原子丢弃，原子交换（比较后）。这就涉及到两个电路。而且输入像素着色器的片段需要保持一定的顺序，所以当一个片元正在进行原子交换时，后续的片元就必须等待。造成较低的吞吐量。而现有的z-cull单元是基于粗粒度的保守估计的深度测试，只能剔除部分像素，可以保持比较高的吞吐量。而针对深度的优化还可以先生成只含深度的贴图信息（称为z pre-pass），给第二个pass节省时间，这里就涉及到片段着色器与纹理单元单宽的取舍问题。



**Figure 4:** GeForce 6800 Pixel Engine

## 2.3 Tesla架构

在GPU发展的初期，顶点处理器和像素片段处理器以不同的速度发展：顶点处理器是为低延迟、高精度的数学运算而设计的，而像素片段处理器是为高延迟、低精度的纹理过滤而优化的。顶点处理器传统上支持更复杂的处理，所以它们比片段处理器更早支持可编程。在发展的过程中，由于需要更强大的编程通用性，这两种处理器类型在功能上已经趋同。另一方面，因为GPU通常处理比顶点更多的像素，所以片段处理器的数量通常是顶点处理器的三倍，然而工作负载不能很好地平衡，这导致更低的效率。例如对于小三角形，顶点着色器工作量大，片段着色器工作量小，反之同理。且Dx 10中增加了更复杂的几何单元处理（例如引入了几何着色器（Geometry Shader）），使得选择固定的处理器比例变得困难。这些都影响设计统一架构的决定。

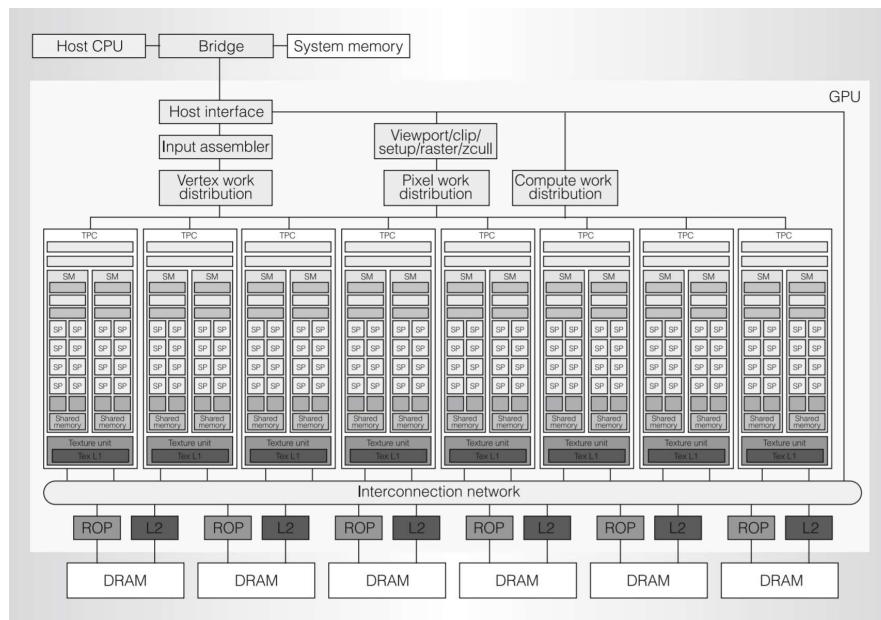


Figure 5: Tesla Architecture

Tesla架构统一且扩展了顶点和像素处理器，构成了可扩展的处理器列。在这里分为流处理器和其它部分介绍。

### 2.3.1 流水线组件

- 主机接口（**Host Interface**）负责收发来自CPU的指令（Command），从显存中获取数据，检查指令连贯性，并处理GPU的上下文切换。
- 输入装配器（**Input Assembler**）组装顶点数据和顶点属性，传给顶点任务分配器。
- 顶点/像素/计算任务分配器（**Vertex、Pixel、Compute Work Distribution**）负责将顶点/片元/计算着色器的任务分发给流处理器完成。
- 纹理处理簇（**Texture Processing Clusters, TPC**）包含了一个纹理单元和两个流处理器，在下一小节会重点介绍。

- 视口/裁剪/装配/光栅化/深度剔除单元  
(**Viewport/clip/setup/raster/zcull**)

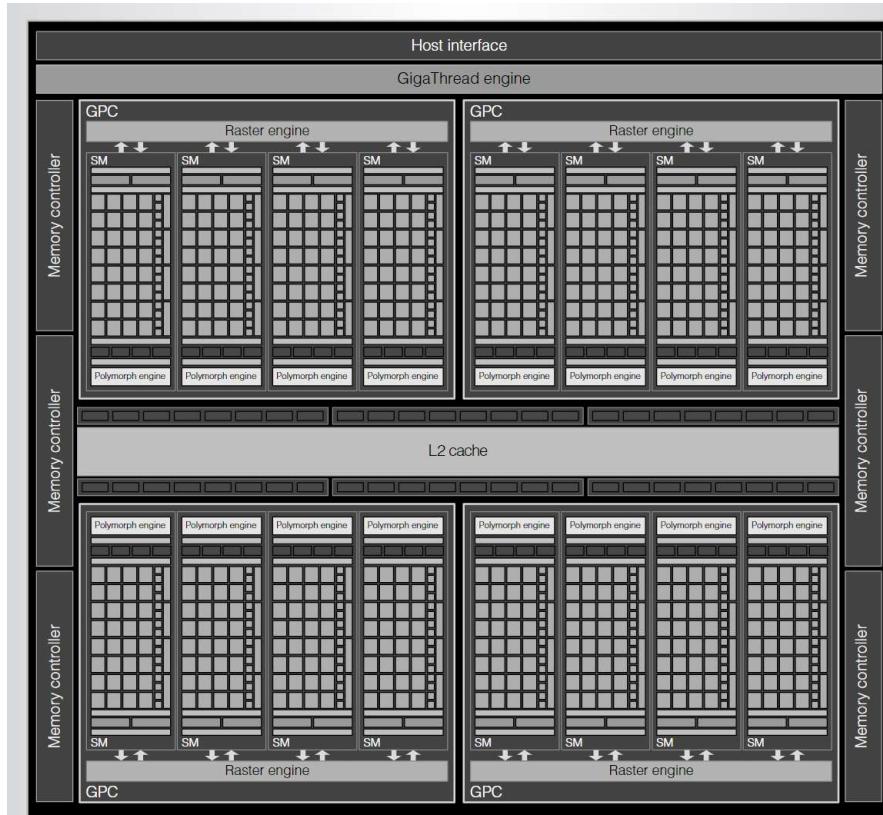
**block**) 组成了传统的光栅化单元，将顶点或几何着色器的结果进行转换和剔除。几何原语被分为一个较小的像素块，既方便快速判断剔除，也通过块地址快速选择处理的流处理器。处理完的像素会被打包通过流处理器控制器分配TPC进行片段着色器的工作。 - 光栅运算处理器 (**Raster Operations Processor**) 功能与传统管线非常类似。每个ROP被分配了一个特定的内存区块，TPC通过互连网络向ROP传送着色像素。 - **L2 Cache、Memory Controller** 和 **DRAM** 组成的存储系统。每个 DRAM 搭配了一个 Memory Controller、L2 Cache 和一个ROP，处理从TPC的SM里发来的数据吞吐请求和ROP写入帧缓冲的像素数据。

### 2.3.2 纹理处理簇

纹理处理簇由以下几个单元组成： - 几何控制器 (**Geometry Controller**) 负责顶点属性在芯片内的输入输出，它会把顶点/几何着色器的结果送到光栅化单元，或者保存到内存中。 - 流处理器控制器 (**SM Controller**) 负责将各种计算任务拆分打包成Warp，交给TPC中所属的某个SM来计算。此外，SMC还负责协调 - 纹理单元 (**Texture Unit**) 包含4个纹理地址生成器和8个滤波单元，通过分析指令中的纹理坐标输出经过插值的纹理值，存在纹理一级缓存中。 - 流式多处理器 (**Streaming Multiprocessor**) 是负责底层运算最主要的单元，和我们熟悉的CPU组成也十分类似。大致可以分为： - 指令缓存会将来自SMC的工作指令会被缓存在其中，分批执行。 - 流处理器 (SP) 执行最基本的浮点型标量运算和各种指数运算，类似于ALU。而特殊函数单元 (SFU) 负责处理更复杂的运算。这和传统管线是大致相同的。 - 多线程调度是任务并行的关键，负责将warp任务分配给计算单元完成。 - 常量缓存和共享内存。是所有处理器共享的存储单元，提供最快速的访问支持。

抛开合并后的计算单元，Tesla架构对传统图形管线并没有非常大的革新。所以在图形处理部分就暂时介绍到这里。而该架构引入的并行编程模型和调度原理会在通用计算章节下做进一步介绍。

## 2.4 Fermi架构



**Figure 6: Fermi Architecture**

Tesla架构统一了不同着色器执行的硬件单元，而光栅管线仍然和传统管线一样作为独立单元在管线中，随着SM数量的增加和效率的提升，顶点着色器输出的片元数量也在增多。而传统光栅单元作为管线中单一的小水管就容易堵塞。而升级的思路也很简单，将光栅处理普及到每个TPC中。

此外，Fermi架构实现了DirectX 11的所有硬件功能，尤其瞩目的是细分着色器与计算着色器。这就引入了更复杂的几何处理。

#### 2.4.1 更完备的TPC

- Fermi架构拆分传统管线中的光栅单元，将顶点获取、曲面细分、视口转换、属性设置和流式输出功能构成了新的几何引擎（PolyMorph Engine）内置于每个SM中。
- 包含边配置、光栅化和Z-cull的Raster Engine只在每个TPC中服务于所有SM。
- 每个SM中都内置了四个纹理单元。

#### 2.4.2 曲面细分

曲面细分是在GPU中为模型添加细节的重要方式。曲面细分的流程大致分为：1. 细分控制着色器（Tessellation Control Shader），决定patch细分的程度。2. 细分原语生成器（Tessellator），根据控制着色器的参数执行细分过程。3. 细分计算着色器（Tessellation Evaluation Shader），处理细分后的所有顶点。

控制着色器通过layout指定要输出的顶点数，也即一个patch要开的线程数。一个patch内的所有线程可以访问这个patch的所有输入（来自顶点着色器）和输出。而控制着色器输出的顶点作为细分计算着色器输入的数据，并为Tessellator设置gl\_TessLevelInner和gl\_TessLevelOuter。

原语生成器会根据输入的信息，在抽象的几何原语上插入抽象patch中相对位置的归一化坐标，作为新的顶点。

而细分计算着色器通过gl\_TessCoord插值后得到新的顶点数据。

## 2.5 Turing 架构

Turing架构是新特性井喷的一代，在图形方面有开天辟地的硬件光追、突破传统的mesh shading、横空出世的DLSS等，同时Turing架构也是人工智能发展史上重要的代表。

### 2.5.1 Mesh Shading

随着传统光栅管线中功能越来越多，几何引擎的调度和任务分发也变得越来越复杂。每次打包Warp所需要的开销也变得扎眼起来。就像之前将光栅化各个阶段全都打包塞进一个SM统一处理一样，将繁琐的许多几何着色器(顶点、曲面细分、几何)整合也变得非常合情合理。

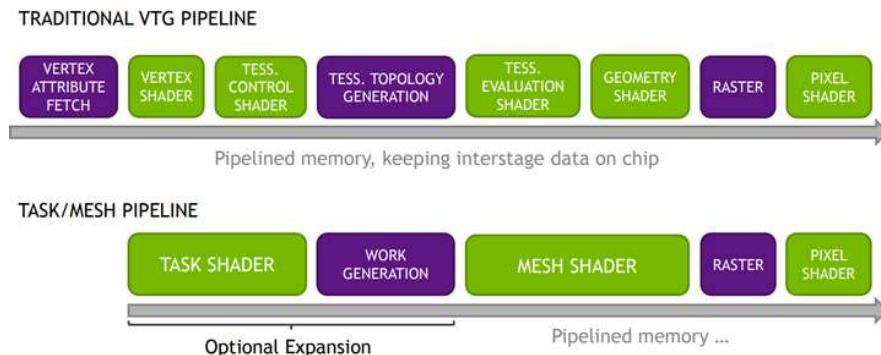


Figure 7 Turing Mesh Shading Pipeline.

Mesh Shader接管了传统管线的顶点变换、细分和其他的处理流程。Mesh Shader和计算着色器一样是协作式编程模型，每个线程具体对应什么也可以由使用者决定的，而不像传统的着色器那样固定每个线程的输出。

而Task Shader可以在Mesh Shader执行之前决定所处理的数据是否需要剔除、选择合适的LOD。

### 2.5.2 RT Core

Turing架构的一大开创就是在流处理器中引入了光线追踪核心（RT Core）以加速层次包围体积的遍历和三角形的相交测试。

相比较传统的GPU光线追踪需要在Shader中计算大量的相交测试，Turing架构可以将所有的BVH遍历和射线-三角形相交测试交给光线追踪核心处理，从而节省了SM在每条射线上花费数千个指令槽的时间，这对于整个场景来说可

能是巨大的指令数量。RT核心包括两个专门的单元。第一个单元执行层次包围体积测试，第二个单元执行射线-三角形相交测试。SM只需要发射射线探针，RT核心进行BVH遍历和射线三角形测试，并向SM返回命中或未命中。SM就能在很大程度上被腾出来做其他图形或计算工作。

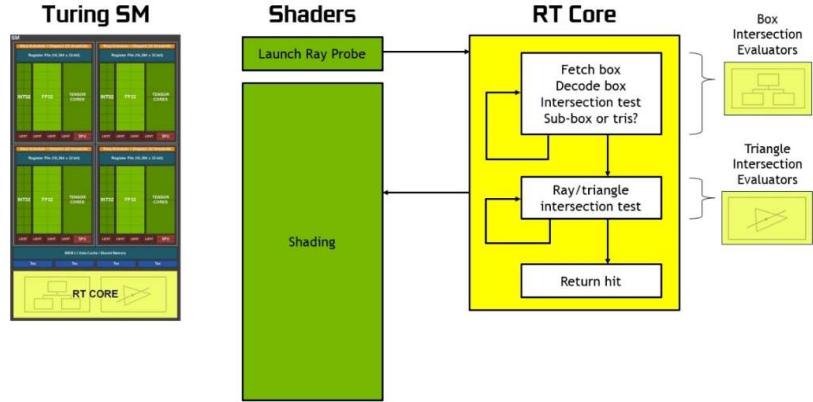


Figure ?Turing Ray Tracing with RT Cores.

### 2.5.3 深度学习超采样（DLSS）

在传统渲染中，后处理和抗锯齿是渲染管线中的重要一环。例如，时域抗锯齿(TAA)是一种使用着色器实现的算法，它使用运动向量结合两帧来确定对前一帧的采样位置，是当今使用的最常见的图像增强算法之一。然而TAA也存在着难以解决的问题，如鬼影和抖动等。

DLSS是典型的引入新功能和传统管线性能和效果相互取舍的例子。DLSS1.0版本主要依赖一个卷积自编码器神经网络来进行图像超分。首先利用当前帧和运动向量进行边缘增强和空间抗锯齿。第二个阶段再通过网络生成超分辨率图像。这样降低渲染目标的分辨率，减少传统管线和光线追踪的开销，将细节生成交给DLSS。

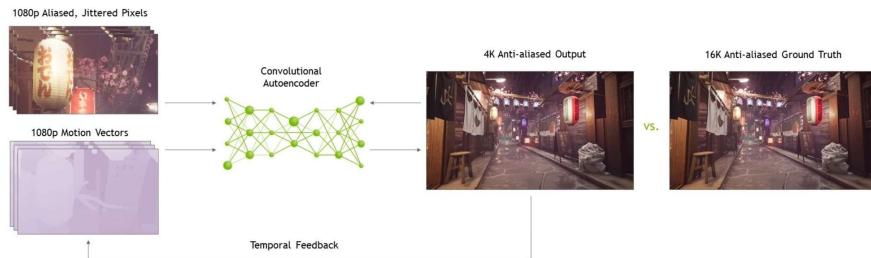


Figure ?Turing DLSS.

随着DLSS的不断发展，到DLSS 3.5已经成为了NVIDIA新显卡不可缺少的技术优势和卖点。也可以看到深度学习的发展对传统渲染管线以及GPU架构的影响和融合，增加了更多可能的技术方向。

### 3 通用计算中的GPU

随着GPU性能的不断提高，将GPU用于更广泛的通用计算成为了一个主要趋势。相比CPU，GPU的浮点计算速度比CPU快得多，且能有更好的负载平衡——将计算任务分配给GPU可独立于CPU完成计算，而且GPU性能的增长曲线远远快于CPU。

不同于CPU的线性处理模式，GPU采用的模型通常是流式模型（Stream Processor），在一组输入数据上并行地执行一个特定的函数，输出一组结果。在这个过程中，函数通常被称为核（kernel），一组数据通常被称为流（stream）。数据以流的形式输入处理器，在核函数上被执行，结果输出到内存中。所有传入处理器的元素独立地被处理，这保证了GPU架构不用向使用者暴露任何并行单元和组成，就可以实现一个完全并行的编程模型。

#### 3.1 可编程管线时代的通用计算

在可编程管线诞生时，已经有相当一部分任务可以将数据抽象为“片段”，在片段着色器中并行计算。如Simon Green在2003年展示了用这样的技术在GPU上模拟布料。模拟算法的核心是基于GPU的双调排序和二分查找，这里就不再展开。而Purcell等人在这时已经开始了光线追踪的尝试。他们将光线追踪的过程分为四个核：光线生成、均匀网格遍历、三角形求交和着色。- 光线生成。给定相机参数，对于每个屏幕像素生成一条出射光线，并检测是否与场景包围盒相交。- 均匀网格遍历。选择均匀网格作为加速结构的理由是，各种加速结构的效率没有绝对优势，且均匀网格在GPU的实现上更为简单，在遍历时也有3维的DDA算法。

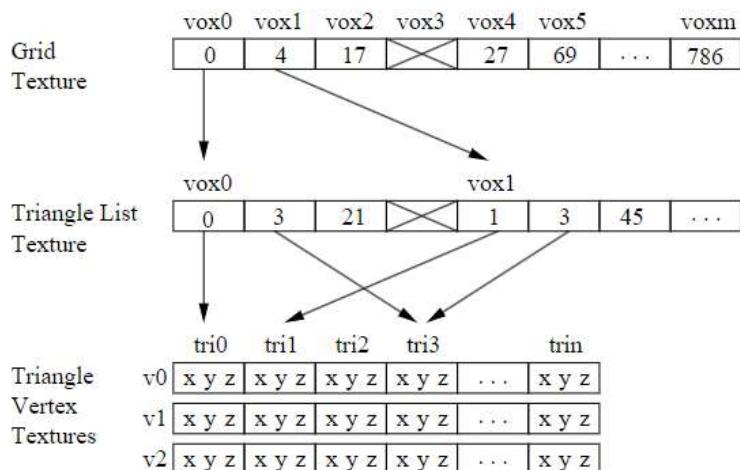


Figure ?The grid and triangle data structures stored in texture memory.

- 三角形求交。对于含有三角形的网格，对体素中包含的所有三角形进行相交测试。  
- 着色阶段。场景的材质和额外的顶点信息存在一系列的RGB纹理中。  
对于Path Tracer，散射的光线会被存在内存中以在场景中继续传播。

这样的光线追踪管线的表现相当不错，且还能在今天的一些基于Computer Shader的光追样例中看到架构/存储上的相似之处。

## 3.2 Tesla管线时代的通用计算

### 3.2.1 流式多处理器的并行原理

一个着色器或是一个CUDA核函数，都定义了一个线程如何对一个像素着色/计算得到一个结果。为了并行管理数百个不同类型的线程，流式多处理器是硬件级别并行的。当SMC取得一个着色器的所有指令后，会将这些指令以一个Warp，即32个线程为单位分发给SM。例如，8个 $2 \times 2$ 的像素块会被打包进入同一个Warp，共享相应着色器的指令。MT Issue每次从存在I Cache中指令取出一条令Warp中的线程执行。这也是SIMT（Single Instruction Multiple Thread）概念由来。

而由于每个线程输入不同，同一条指令的结果也不同。当遇到分支时，需要额外的开销来判断Warp中所有线程是否会进入同一分支，存在不可避免的不同分支时，所有线程将分类串行，这会引入无法避免的额外开销。而另一个非常大的时间消耗来自内存阻塞，在传统管线中，厂商会建议开发者在程序开始时请求相关资源，而在中间插入其它计算以减少等待时间。而在流式多处理器中，硬件设计者提供了一个足够大的指令缓存，通过切换Warp来隐藏内存读取的延迟。所以线程调度器还需要每两个周期从多个Warp中挑选合适的进行切换。

### 3.2.2 CUDA (Compute Unified Device Architecture)

Tesla架构解耦了计算单元和图形单元，使通用管线的自由度更高，也释放了更灵活、性能更强的可能性。CUDA就应运而生了。

CUDA是基于C/C++的扩展，用户可以将核函数写成C的函数形式，nvcc编译器将编译的含有CPU串行代码和并行的GPU核函数通过CUDA的运行时API，将GPU作为CPU的协处理器进行并行计算。

作为一个编程模型，我将会从线程和存储架构来介绍CUDA的设计思想。

CUDA的线程结构为Grid、Block和Thread。同一个Grid中的所有Block相互独立，而每个Block中的Thread可以通过共享内存传递数据，所以Block中的线程也被称为协作线程组（CTA，cooperative thread array）。每个Grid包含的Block数量和每个Block包含的Thread个数由核函数确定。而线程处理的数据则通常由线程在结构中的ID进行索引。

#### 3.2.2.1 CUDA的内存模型

配合线程的结构，GPU内存也被分为不同的层级以减少带宽消耗，提高数据读取速度。

每个线程都有独立的寄存器组，以存储所需的局部变量。

每个Block共享一片内存区域，且包含一个一级缓存用来存储纹理数据。一个Block内的所有线程必定位于同一个SM中，而不同Block的线程是相互不可见的。

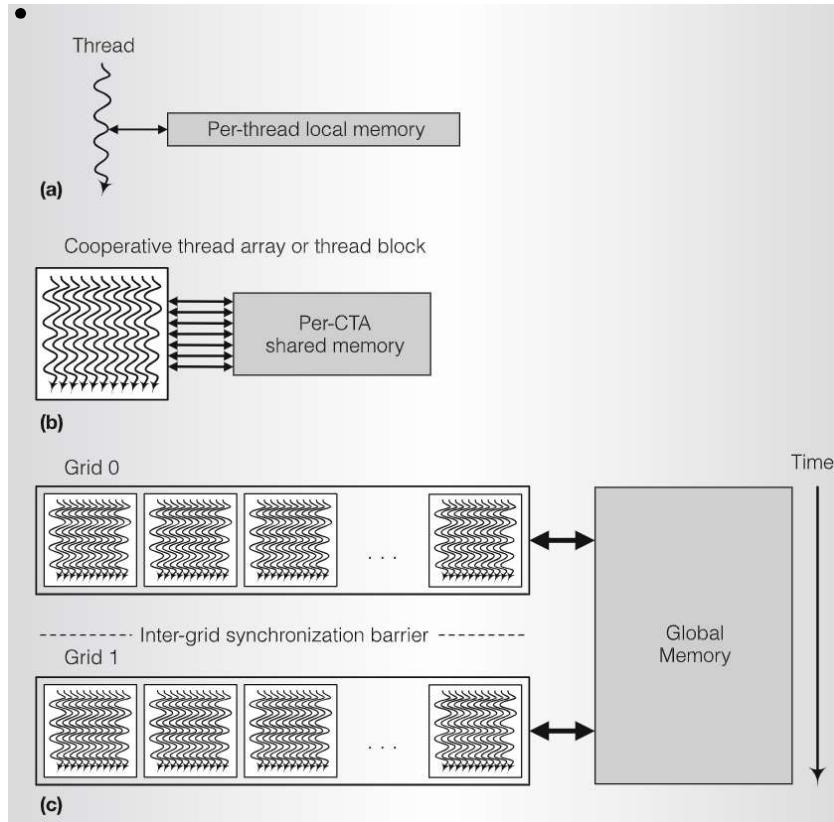


Figure. [CUDA Mem] CUDA Memory Architecture

而全局内存通过L2 Cache与SM相连，所有SM可以共同使用。不同Grid的执行是串行的，一个Warp中的连续线程访问连续的内存，可以被合并为一条内存读取指令。

### 3.3 Turing管线时代及之后的通用计算

#### 3.3.1 Tensor Core

张量核心（Tensor Core）是专门用于执行张量 / 矩阵操作的专门执行单元，该单元是深度学习中使用的根本计算功能。Tensor Core并不是在Turing架构中第一次被引入，但增强的速度和使小规模神经网络任务可以应用到实时任务中。Turing Tensor Core为深度学习神经网络训练和推断操作的核心矩阵计算提供了巨大的加速。在Turing架构中，Tensor Core添加了新的INT8和INT4精度模式，用于推理可以忍受量化且不需要FP16精度的工作负载。

简单地说，Tensor Core可以在一个周期内计算 $4 \times 4$ 的矩阵乘法，其中两个 $4 \times 4$  FP16矩阵相乘，然后将结果添加到 $4 \times 4$  FP16或FP32矩阵中，最终输出新的 $4 \times 4$  FP16或FP32矩阵。在利用两个Tensor Core时，warp调度器直接发出矩阵乘法运算，并且在从寄存器接收输入矩阵之后，执行 $4 \times 4 \times 4$ 矩阵乘法。待完成矩阵乘法后，Tensor Core再将得到的矩阵写回寄存器。

## 4 展望

总结NVIDIA显卡架构的发展历程，可以看到是一个算法和硬件相互促进、通用计算和传统光栅融合与取舍的过程。

从算法与硬件的角度来说，在管线发展早期，算法和API的进步使得厂商主动适配，加入相关硬件功能（如几何着色器、曲面细分等）。而随着GPU计算能力的提高、通用计算的可能性增加，以及特定硬件单元的加入，给图形API和传统渲染流程提供了更多选择。混合渲染管线（Hybrid Rendering Pipeline）就是最好的例子。

从通用计算和传统图形计算的角度来说，GPU的趋势是将传统的图形计算过程整合到通用的计算管线中，并提供一些功能特化的硬件以加速特定的图形计算过程，如RT Core。由于传统图形算法的效率和效果上限，盲目堆积图形特定硬件、增大纹理单元和显存给图形效果的提升已经到了边际效应的极限。通过更好地利用通用计算单元，尤其是人工智能算法（DLSS），可以从更特殊的场景表达、后处理效果、生成模型上给传统图形管线以新时代的可能。

要赶上最新的发展进程和可能的趋势，就要主动了解GPU的功能和发展，从而不落后于时代。

## 引用

- [GeForce6800] J. Montrym and H. Moreton, “The GeForce 6800,” in IEEE Micro, vol. 25, no. 2, pp. 41-51, March-April 2005,  
doi:10.1109/MM.2005.37.
- [GPU Gems] Randima Fernando. 2004. GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics. Pearson Higher Education.
- [GPU Gems2] Matt Pharr and Randima Fernando. 2005. GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems). Addison-Wesley Professional.
- [GPU Gems3] Hubert Nguyen. 2007. Gpu gems 3 (First. ed.). Addison-Wesley Professional.
- [Simon] Green, S. (2003, March). Stupid opengl shader tricks. In Game Development Conference (Vol. 2003).
- [Purcell] Purcell, Timothy J., Ian Buck, William R. Mark, and Pat Hanrahan. 2002. “Ray Tracing on Programmable Graphics Hardware.” ACM Transactions on Graphics 21(3), pp. 703–712.
- [Turing] NVIDIA TURING GPU ARCHITECTURE.
- [DLSS] Burgess, John. “Rtx on—the nvidia turing gpu.” IEEE Micro 40.2 (2020): 36-44.
- [Tesla] Lindholm, Erik, et al. “NVIDIA Tesla: A unified graphics and computing architecture.” IEEE micro 28.2 (2008): 39-55.
- [Fermi] Wittenbrink, Craig M., Emmett Kilgariff, and Arjun Prabhu. “Fermi GF100 GPU architecture.” IEEE Micro 31.2 (2011):50-59.

*formatted by Markdeep 1.16* ♦