第5章　底层软件系统

【本章导读】

本章阐述深度学习系统的底层支撑软件，包括图计算运行时、硬件抽象运行时、驱动系统与 GPU 固件。结合内存管理、同步机制、事件流控制与异构执行等关键技术，建立了从模型执行到硬件调度的完整软件通路，并对不同硬件架构的适配进行了深入分析。

5.1　运行时系统概述


视频讲解

运行时系统是连接高层计算逻辑与底层硬件执行的关键组成，广泛应用于深度学习、高性能计算和异构计算中。它由图计算运行时与硬件抽象运行时协同构成，前者负责解析计

算图、管理任务依赖与调度顺序，并屏蔽底层复杂性，支持静态与动态图执行；后者则统一管理底层硬件资源，如设备、内存、线程与数据通信，提供低延迟、高带宽的访问接口，支持多种异构设备的高效协作。二者共同实现任务划分、资源优化和设备适配，显著提升系统在复杂多硬件环境下的执行效率与稳定性。

1. 系统结构

运行时系统结构是从高层逻辑到底层硬件执行的关键组件，它分为多个层次，每一层都有明确的功能分工和协作方式，以实现任务分解、优化、资源管理和执行，详细的运行时系统结构如图 5.1 所示。

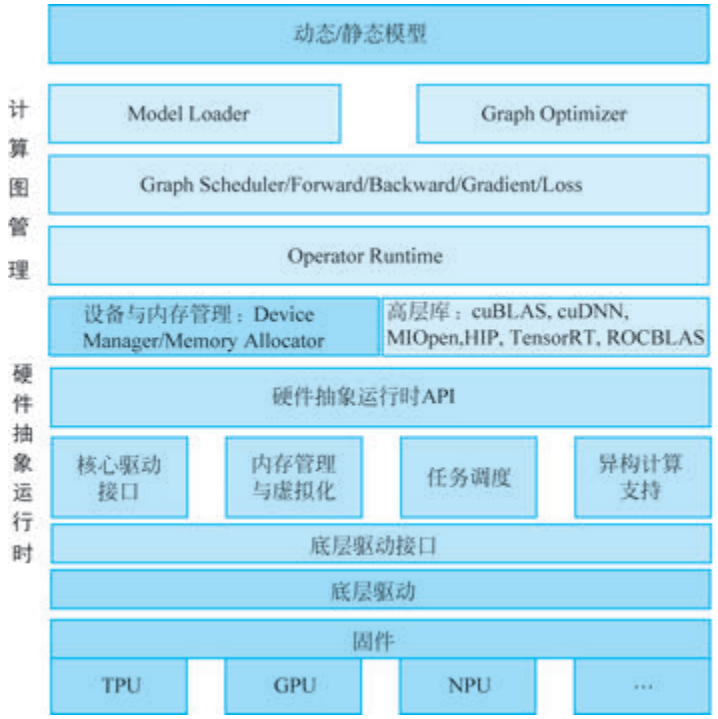

图 5.1 运行时系统结构

图 5.1 展示了从模型到底层硬件的完整运行时工作流程，划分为多个功能层，每一层在整个系统中扮演不同的角色，具体分为以下几个主要模块。

1) 动态/静态模型

该模块是用户与运行时系统交互的入口，负责接收用户定义的计算任务(如深度神经网络模型)，支持动态计算图(如 PyTorch)与静态计算图(如 TensorFlow 静态模式)两种形式。动态计算图在运行时构建并执行计算图，灵活性强；而静态计算图则在运行前进行图结构与算子优化，更适合高效部署。该模块提供高层抽象，屏蔽底层执行细节，是整个运行时流程的起点。

2) 计算图管理

计算图管理承担着模型加载、图结构优化与任务调度的核心职责。模型加载器(Model Loader)负责解析和加载用户模型(如 ONNX、TF SavedModel)，并转换为内部统一表示；计算图优化器(Graph Optimizer)对计算图进行结构性优化，包括操作融合、拓扑排序与内

存复用等策略，以提高整体运行效率；图调度器（Graph Scheduler）则根据优化后的图调度计算任务，包括前向传播、反向传播和梯度计算等操作；最终，操作运行时（Operator Runtime）通过调用高性能库或推理引擎（如 cuDNN、MIOpen、TensorRT 等）完成具体算子的执行，实现高效的图计算执行。

3）设备与内存管理

这一层负责管理系统中的计算资源与内存资源。设备管理器（Device Manager）动态识别可用的硬件设备（如 CPU、GPU、NPU），并根据资源状态与任务需求进行任务调度与设备分配，实现多设备并行与异构计算支持；内存分配器（Memory Allocator）负责主机与设备之间的内存传输、分配与回收，同时实施内存复用策略，避免碎片化，并充分利用共享内存、寄存器等硬件特性以提升计算效率。

4）硬件抽象运行时

硬件抽象运行时通过统一的抽象接口隐藏底层硬件差异，使上层系统能跨平台部署。该层通过硬件抽象 API 提供驱动控制、虚拟地址管理、任务调度与异构计算支持，确保图执行能够无缝映射到多种硬件环境中；同时，底层驱动接口（如 CUDA、HIP、OneAPI）与具体硬件驱动交互，提供实际的设备控制能力，是整个运行时系统与硬件之间的连接纽带。

5）驱动与固件

底层驱动接口负责向硬件发送低级命令并接收状态反馈；底层驱动是硬件厂商提供的核心程序，实现数据传输（如 DMA）、硬件控制、指令执行等功能，并确保与操作系统的交互。固件是运行在硬件内部的小型程序，负责解释和执行来自驱动的微指令，管理硬件内部资源，并在某些加速器中实现算子的底层控制逻辑。

6）底层硬件

底层硬件是执行所有计算任务的物理基础，涵盖通用处理器（CPU）、图形处理器（GPU）、张量计算加速器（如 Tensor Core）、定制推理芯片（如 NPU、TPU）及可编程硬件（如 FPGA）等。不同类型的硬件各自承担不同角色，如 CPU 适合串行逻辑和主控流程，GPU 擅长并行计算，FPGA 和 TPU 更适用于低功耗、高吞吐场景。它们在运行时系统调度下完成计算图中各算子的具体执行任务，支撑整个系统的高性能运行。

运行时系统结构自上而下形成清晰分层，每层各司其职，协同实现从模型加载到跨硬件执行的高效运行时支持。

2. 工作流程

以下以深度学习推理为例描述运行时系统的典型工作流程。

① 构建计算图：用户通过高层框架定义计算逻辑，计算图运行时解析任务并构建计算图。

② 图优化与调度：运行时对计算图进行优化（如操作融合、节点重排序）。将优化后的任务分配到不同的设备。

③ 算法库调用：计算图运行时调用适配的算法库（如 cuDNN、OneDNN）执行具体操作。

④ 硬件资源管理：硬件抽象运行时分配设备内存、调度线程并启动计算内核。

⑤ 数据传输与同步：硬件抽象运行时完成主机与设备、设备间的数据传输与同步。

⑥ 结果返回：执行完成后，将结果从设备传回主机并返回给用户。

3. 典型运行时

以下是一些典型的图计算运行时和硬件抽象运行时的列举和说明,并对它们的背景、特点和应用场景进行了说明。它们分别代表当前主流深度学习推理与训练框架中的核心运行时机制。

1) 图计算运行时

(1) ONNX Runtime。

ONNX Runtime 是一个高度优化的跨平台推理引擎,专为支持 ONNX 格式的模型而设计,可无缝集成 PyTorch、TensorFlow 等主流框架。它支持在多种硬件上执行,包括 CPU、GPU(CUDA 和 ROCm)等,并通过图优化(如操作融合)和量化推理(如 INT8)提升模型执行效率。其良好的扩展性允许用户添加自定义算子,使其广泛应用于模型部署与推理优化场景中,特别适用于多硬件平台下的统一模型运行。

(2) PyTorch Runtime(TorchScript)。

PyTorch 的运行时系统以动态图机制著称,并通过 TorchScript 提供将模型转换为静态图的能力,从而在保留灵活性的同时提升执行效率。在分布式训练方面,PyTorch 支持多种策略(如 DataParallel 与 DistributedDataParallel)和通信库(如 NCCL),适合多 GPU 并行训练。其运行时体系结构兼顾训练与推理,尤其适用于需要高度可定制逻辑和动态图控制流的模型开发与部署任务。

(3) TensorFlow Runtime(XLA Runtime)。

TensorFlow 的运行时以静态图为基础,依赖 XLA(加速线性代数)进行图优化和高性能执行。XLA 提供算子融合、内存优化等高级技术,支持在 CPU、GPU、TPU 等设备上进行高效推理与训练。通过图的提前编译与硬件感知优化,TensorFlow Runtime 可有效提高执行效率,特别适用于规模化部署、高吞吐推理与 TPU 上的训练任务。

(4) TVM Runtime。

TVM 是一个面向深度学习推理的模型编译框架,其运行时用于加载和执行编译好的模型模块,具备极强的跨硬件适配能力。它通过对目标硬件(如 CPU、GPU、FPGA、ASIC)的自动调优生成高效执行代码,并支持算子级定制与融合优化。TVM Runtime 的模块化设计使其适用于构建高性能、低功耗的模型部署系统,特别适合在资源受限或定制化硬件平台上的高效推理应用。

2) 硬件抽象运行时

(1) CUDA Runtime(NVIDIA)。

CUDA Runtime 是 NVIDIA 提供的硬件抽象运行时,广泛用于管理基于 NVIDIA GPU 的并行计算任务。它通过屏蔽底层 GPU 架构细节,为开发者提供统一的 API,支持内存分配(如 cudaMalloc)、内核启动(如 cudaLaunchKernel)以及主机与设备间的数据传输(如 cudaMemcpy)。此外,它与 cuBLAS、cuDNN 等高性能库深度集成,构成完整的软件栈。CUDA Runtime 是深度学习、科学计算、图像处理等领域实现高性能 GPU 加速的核心支撑组件。

(2) ROCm Runtime(AMD)。

ROCm(Radeon Open Compute)是 AMD 面向异构计算推出的运行时平台,支持 AMD 自家 GPU 及通过 HIP 与 CUDA 的兼容性。在功能层面,ROCm 提供与 CUDA 类似的接

口，如 hipMalloc、hipMemcpy 和 hipLaunchKernel，同时具备开放源码、可扩展的特点。ROCm Runtime 支持与 MIOpen、ROCBLAS 等 AMD 加速库结合，适用于在开源生态中使用 AMD GPU 进行深度学习、科学仿真等高性能计算任务。

(3) OneAPI Runtime(Intel)。

OneAPI 是 Intel 推出的统一异构计算平台，其运行时通过 Data Parallel C++(DPC++)接口实现跨架构编程，支持 CPU、GPU、FPGA 等设备的统一访问。运行时同时集成了高性能计算库如 OneDNN、OneDAL，用于优化深度学习、数据分析等任务。OneAPI Runtime 的关键优势在于提供统一的开发接口和开放的扩展能力，使得异构应用在 Intel 全栈硬件上具备良好的移植性与高性能表现，广泛应用于企业级 AI 与 HPC 场景。

(4) OpenCL Runtime。

OpenCL 是一种开放标准的异构计算框架，其运行时提供跨厂商的设备支持，包括 NVIDIA、AMD、Intel 的 CPU/GPU 以及多种 FPGA 平台。OpenCL Runtime 支持设备初始化、内存管理、任务调度等底层功能，并允许用户通过自定义内核开发并行任务。凭借其高度的通用性和平台无关性，OpenCL 在图像处理、科学模拟、加密计算等异构任务中广泛应用，尤其适合需要在多个硬件平台间迁移的场景。

4. 两层运行时的结合

图计算运行时与硬件抽象运行时通过紧密协作，实现了从高层模型表达到底层设备执行的高效映射机制。结合示例如下。

① ONNX Runtime + CUDA/ROCm Runtime：ONNX Runtime 调用 CUDA 或 ROCm 提供的硬件抽象接口(如 cudaLaunchKernel)来执行模型推理。

② PyTorch Runtime+CUDA/ROCm：PyTorch 使用动态计算图和 TorchScript 编译图，通过 CUDA 或 ROCm 加速底层操作(如卷积、矩阵乘法)。

③ TensorFlow Runtime+XLA+CUDA/TPU：TensorFlow 通过 XLA 优化计算图，调用 CUDA Runtime 或 TPU 的硬件抽象运行时进行加速。

二者结合后，图计算运行时可通过统一接口调用底层运行时提供的资源，利用 GPU、TPU 等异构硬件实现高性能推理与训练，从而构建出跨平台、可扩展、高效能的深度学习执行环境。

5. 优缺点

图计算运行时与硬件抽象运行时协同工作，具备显著优势。优点在于它通过图优化和异步执行实现高性能，提升硬件利用率；支持多种设备(如 CPU、GPU、FPGA)，具备良好适配性；同时支持静态图和动态图，扩展性强；为用户提供统一接口，自动管理资源，降低开发门槛。但也存在不足，如系统架构复杂，多设备并行下调试难度大；主机与设备之间的数据传输可能成为瓶颈；支持多平台会增加实现和维护的复杂度。整体来看，该体系适用于高性能异构计算，但在调试与跨平台兼容性方面仍有挑战。

6. 总结

运行时系统是承接高层计算逻辑与底层硬件执行的核心桥梁，广泛用于深度学习、高性能与异构计算场景。它由图计算运行时与硬件抽象运行时协同构成，分别负责模型调度优化与底层资源管理，具备高抽象性、操作融合、动态调度和高效通信等能力。未来的发展重点包括提升性能(如混合精度、并行计算)、增强异构支持(覆盖 CPU、GPU、FPGA 等)以及

优化数据传输效率。总体来看,运行时系统在提升系统性能、跨硬件适配和降低使用复杂度方面发挥关键作用,是智能计算平台不可或缺的核心组成部分。


视频讲解

5.2 图计算运行时

图计算运行时是深度学习系统中的高层调度与执行核心,负责将用户定义的模型逻辑表示为计算图,并对其进行优化和调度执行。它支持静态图和动态图两种模式,分别适用于高效部署与灵活开发,通过算子融合、常量折叠、内存复用等技术提升执行效率。同时,图计算运行时能够分析算子间依赖,实现并行化调度与分布式部署,并提供易用的接口(如TensorFlow 的 tf.function、PyTorch 的动态图机制),将高层模型表达高效映射到底层硬件执行,是构建高性能深度学习系统的关键组成。

1. 运行时架构

图计算运行时一般用于深度学习计算图的推理和训练过程,旨在高效调度节点操作,同时优化内存和硬件资源利用。图计算运行时基本框架如图 5.2 所示。

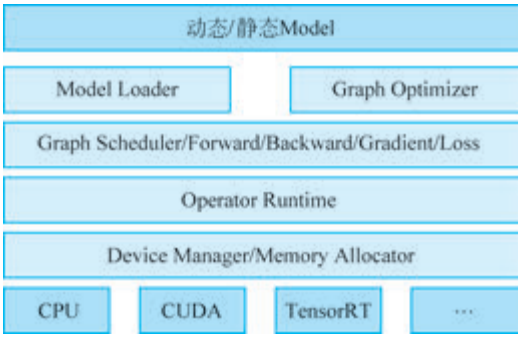

图 5.2 图计算运行时基本框架

以下是图计算运行时各核心组件的分段描述,展示其在深度学习模型执行过程中的关键职责与协同机制。

(1) 动态/静态 Model。

用户定义的深度学习模型,可以采用动态计算图(如 PyTorch Eager 模式)或静态计算图(如 TensorFlow Graph 模式)。动态模型灵活、易调试,适合研究开发;静态模型结构固定、易于优化和部署,适合生产环境。该层的主要输出是计算图的初始表示,为后续图优化与执行做准备。

(2) 模型加载器(Model Loader)。

Model Loader 负责加载模型结构和权重,支持从文件(如 ONNX、SavedModel、TorchScript)或网络加载,并将其转换为系统内部统一的中间表示(IR),为后续的图优化提供一致的数据结构基础。

(3) 计算图优化器。

图优化器负责对原始计算图进行结构性优化,以减少不必要的计算和数据传输开销。典型优化策略包括操作融合(将多个连续操作合并为一个计算节点)、静态形状推断(在编译阶段推导张量维度)以及常量折叠(提前计算常量表达式)。通过这些优化,计算图在执行时

更高效，内存开销更小，整体推理或训练速度显著提升。

(4) 图调度器。

图调度器基于计算图中节点之间的依赖关系，确定各操作的执行顺序，并生成运行时的调度计划。调度方式包括顺序执行(拓扑排序)和并行执行(对无依赖节点并发分配)。通过合理调度，可充分利用多核 CPU、GPU 或分布式计算资源，最大化系统并发度和执行效率。

(5) 前向传播模块。

前向传播模块按照优化后的计算图逐节点执行操作，计算每层的输出结果，同时缓存必要的中间状态(如激活值)，以备后续反向传播使用。它严格遵循依赖关系调度执行，并配合缓存策略减少内存使用，是模型推理和训练中执行路径的核心环节。

(6) 反向传播模块。

反向传播模块根据前向图自动构建反向计算图，并按照链式法则计算各参数的梯度。它依赖于自动微分机制(如 PyTorch 的 autograd)自动完成梯度传播逻辑，同时支持在多设备或分布式训练环境中进行梯度同步，确保参数更新一致性。

(7) 梯度更新模块。

在训练过程中，梯度更新模块根据计算得到的梯度使用优化算法(如 SGD、Adam)对模型参数进行更新。它支持正则化(如 L2)、动态学习率调整，以及混合精度训练下的梯度缩放机制，从而提升模型收敛速度与数值稳定性。

(8) 操作运行时。

操作运行时是实际执行计算图中节点操作的模块，如矩阵乘法、卷积等核心算子。它会根据设备类型选择合适的实现路径，调用高性能计算库(如 cuDNN、MKL、ROCBLAS)，并支持用户自定义算子。该模块在执行效率和硬件适配方面起着关键作用。

(9) 设备管理器。

设备管理器负责调度和管理系统中的所有计算设备，包括 CPU、GPU 等。在多设备场景下，它根据当前资源利用率和性能指标选择最佳设备，并将任务动态分发，实现跨设备负载均衡与高效调度。

(10) 内存分配器。

内存分配器承担模型执行过程中内存资源的动态或静态管理。它支持如 TensorFlow 的内存池机制，实现预分配和复用，减少频繁分配与释放带来的开销，同时负责 GPU 设备内存、共享内存等多级内存的协调管理，有效提升内存使用效率并降低峰值内存占用。

这些组件共同构成了图计算运行时的核心支撑体系，从图结构优化到操作调度、设备管理再到底层执行，实现了深度学习任务从高层建模到高效执行的完整闭环。

2. 工作流程

(1) 计算图加载：加载已构建的计算图，执行拓扑排序以确定节点依赖关系。

(2) 计算图优化：应用优化器对计算图进行优化，如操作融合和常量折叠。

(3) 生成调度计划：调用图调度器生成执行计划(顺序或并行)，训练时期根据前向传播，产生反向传播及梯度更新运算。

(4) 执行操作：调用操作运行时执行节点计算，使用设备管理器和内存分配器优化资源利用。

(5) 结果回传：操作执行完成后，将结果回传至下游节点或用户。

3. 优缺点

图计算运行时通过操作融合、内存复用和混合精度计算等优化手段，大幅提升执行效率，并支持动态图和静态图，兼顾开发灵活性与推理性能。同时具备良好的跨平台能力，可运行在 CPU、GPU、TPU 等多种设备上，并支持分布式训练和插件扩展，屏蔽底层硬件差异。然而，其缺点也较明显，如静态图调试难度高，动态图性能不如静态图，部分运行时对特定硬件依赖性强，分布式任务协调复杂，不同框架间兼容性有限。这使得图计算运行时在性能与适应性之间仍需权衡。

4. 总结

图计算运行时是连接深度学习框架与底层硬件的关键组件，负责解析、优化和调度计算图，以实现高效执行。其发展正朝着动态图与静态图融合、多设备协同、图优化自动化以及跨框架兼容性方向推进，既提升了模型执行效率，也增强了系统的灵活性与可移植性。在实际应用中，图计算运行时广泛用于高性能推理、分布式训练和异构硬件适配，成为支持现代深度学习系统在云端、边缘和多平台环境中高效运行的重要基础设施。

5.3 硬件抽象运行时

硬件抽象运行时系统是位于高层图计算运行时与底层硬件之间的核心桥梁，负责统一管理和调度计算资源、内存空间、线程执行和设备通信，以屏蔽不同硬件架构的差异性，提供一致的编程与执行接口。典型代表如 NVIDIA 的 CUDA Runtime 和 AMD 的 ROCm Runtime，它们面向 GPU 平台，通过抽象计算单元、共享内存、寄存器等底层资源，使上层框架无须关注硬件细节即可高效运行模型。在功能层面，HARS 支持主机与设备之间的数据传输(如 cudaMemcpy、hipMemcpy)、显式或统一内存分配、线程块和网格的并行任务组织，以及在多 GPU 环境下的设备互操作。同时，它还集成丰富的性能分析与调试工具(如 CUDA 的 Nsight、ROCm 的 rocprof)，帮助开发者识别瓶颈、优化调度。此外，HARS 具备良好的跨平台编译能力，如 CUDA 使用 PTX 中间表示适配不同架构的 NVIDIA GPU，ROCm 则基于 HIP 实现对 AMD GPU 的高效支持，并逐步扩展至更多异构平台。通过提供统一高效的硬件接口、调度机制和开发工具，硬件抽象运行时系统不仅显著提升了底层设备的计算利用率，还极大降低了上层 AI 框架在多平台环境中的部署难度，是现代智能计算系统不可或缺的基础组件。

1. 运行时架构

硬件抽象运行时通过分层设计实现高效的硬件资源管理、任务调度和数据传输。它在高层库与底层硬件之间构建了一个桥梁，使开发者能够以更简单的方式使用复杂的硬件资源，是现代高性能计算和深度学习框架的重要基础。未来，硬件抽象运行时将在跨平台支持、性能优化和异构计算领域继续发挥关键作用。硬件抽象运行时基本结构如图 5.3 所示。

根据图示，硬件抽象运行时位于高层库和底层驱动之间，主要进行内存、任务等管理。以下是详细的分层功能说明。

1) 高层库

高层库是直接为深度学习框架和开发者提供服务的优化算子库，封装了大量常用的数学与神经网络操作，如矩阵乘法、卷积、归一化、激活函数等。典型代表包括 NVIDIA 的

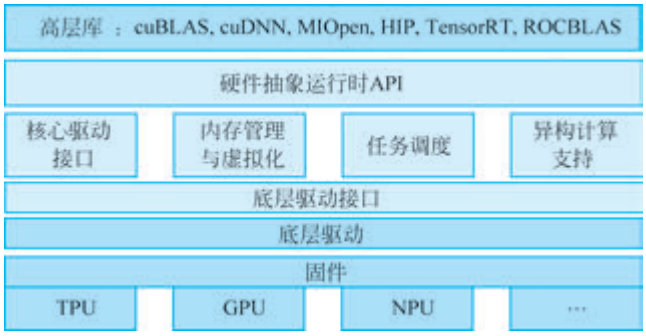

图 5.3 硬件抽象运行时基本结构

cuBLAS(线性代数)、cuDNN(深度学习),以及 AMD 的 MIOpen 和 ROCBLAS。这些库基于底层硬件架构进行深度调优,并通过调用硬件抽象运行时提供的 API(如 cudaMemcpy、cudaLaunchKernel)实现高效计算,从而为用户屏蔽了底层复杂性,提升了开发效率与跨平台可移植性。

2) 硬件抽象运行时

这一层是整个运行时系统的核心层,向高层库提供统一的程序接口,向下封装底层驱动功能。其模块包括:

① 核心驱动接口:用于设备初始化与信息查询,如 cudaSetDevice()、cudaGetDevice(),支持异构系统中多设备识别与控制。

② 内存管理与虚拟化:负责主机与设备之间的内存分配与数据传输,如 cudaMalloc()、cudaMemcpy(),并支持统一内存模型(Unified Memory)以简化大规模内存操作。

③ 任务调度:将高层算子任务调度为可执行的 GPU 核函数,并通过线程块与网格模型在硬件上高效执行,如使用 cudaLaunchKernel()。

④ 信号和同步:管理信号对象,实现任务间的同步。

⑤ 异构计算支持:提供统一接口封装多种异构设备,如 CPU、GPU、FPGA,典型代表如 OneAPI 和 HIP,支持跨架构协同与数据交互。

该层通过模块化设计,构建出对上层友好、对硬件高效的通用接口体系,是异构计算生态的关键支撑。

3) 底层驱动接口

底层驱动接口是硬件抽象运行时系统的最底层执行核心,直接控制物理设备,管理内存、线程、任务队列等底层资源。CUDA 的 cuDriver API 和 ROCm 的 HSA(Heterogeneous System Architecture)是典型实现,它们向上提供调用接口,支持运行时层的内核调度与内存访问,同时向下对接硬件微架构,是实现精细控制与硬件利用最大化的基础。

2. 工作流程

以下以 CUDA 为示例描述工作流程(ROCm 类似)。

(1) 初始化:加载驱动程序,初始化运行时环境。

(2) 内存分配:分配主机和设备内存(如 cudaMalloc)。

(3) 数据传输:将数据从主机传输到设备内存(如 cudaMemcpy)。

(4) 核函数调用:在设备上启动 GPU 核函数,执行并行计算。

(5) 同步与结果获取：同步主机和设备任务，获取计算结果。

(6) 清理与释放资源：释放设备内存，关闭运行时环境。

3. 对外接口

AMD ROCm 平台上 ROCR(ROCm Runtime)运行时典型对外 API 的分类说明如表 5.1 所示。

表 5.1 ROCR 运行时典型接口种类

分　　类	描　　述
错误处理	获取错误状态码对应的字符串描述
运行时初始化与关闭	初始化或关闭 HSA 运行时环境
系统与代理信息	查询系统或代理(agent)的相关信息
信号与同步	创建信号并进行同步操作
调度与队列管理	创建和销毁调度队列，用于任务分发
内存管理	分配和复制内存资源
代码对象与可执行文件	创建代码对象和可执行文件
扩展功能	AMD 特定的内存池操作扩展
虚拟内存管理	管理虚拟内存地址的保留和映射

这些 API 函数定义在 ROCm 的官方文档和源代码中，读者可以参考 ROCR 运行时 API 文档获取详细信息。

4. 底层驱动用户态接口

表 5.2 所示为 AMD ROCm 平台中 ROCt Thunk 层(libhsakmt)提供的典型驱动对上 API 的分类说明。

表 5.2 底层驱动用户态接口概述

分　　类	描　　述
初始化与关闭	打开或关闭与内核驱动(KFD)的连接
系统信息获取	获取或释放系统属性信息
拓扑信息查询	查询节点、内存、缓存及 I/O 链路属性
事件管理	创建、销毁、设置、重置事件及查询事件状态
队列管理	创建、更新、销毁队列及设置计算单元掩码
内存管理	分配、释放、注册、注销内存及映射/取消映射到 GPU
共享内存管理	共享内存及在多个节点间注册共享句柄
调试支持	注册/注销调试、监视及获取设备/队列数据
性能计数器	获取性能计数器属性、注册/注销跟踪、获取/释放跟踪访问、启动/查询/停止跟踪
其他功能	获取时钟计数器、查询指针信息、获取队列信息等

完整的 API 列表及其详细说明可在 libhsakmt 的版本控制文件中找到。

5. 优缺点

硬件抽象运行时系统具备显著优势，能够实现硬件级优化，最大限度发挥 GPU 的计算性能，同时支持多线程与多 GPU 调度，适用于 AI 与高性能计算场景。CUDA 在 NVIDIA 平台具有成熟生态，而 ROCm 也在逐步拓展跨平台能力。两者均提供完善的开发工具链，涵盖编译、调试和性能分析。然而，它们也存在一些限制，如平台依赖性强(CUDA 仅限

NVIDIA，ROCm 对非 AMD 支持不足），开发门槛较高，需要深入理解并行计算模型与硬件架构，同时在主机与设备之间的内存管理仍需显式操作，增加了开发复杂度。

6. 总结

硬件抽象运行时系统（以 CUDA 和 ROCm 为例）是高性能计算领域的关键技术，能够屏蔽底层硬件复杂性，提升开发效率和跨平台能力。CUDA 在 NVIDIA 生态中广泛应用，凭借成熟的工具链和优化能力占据主导地位；ROCm 则通过支持多硬件平台逐步扩大影响力。未来，随着异构计算的普及，这类运行时系统将在高性能计算、人工智能、大数据分析等领域发挥更重要的作用。

5.4 ONNX Runtime

ONNX Runtime 是微软推出的高性能推理引擎，旨在支持 ONNX 格式模型的高效执行。作为 ONNX 生态的核心组件，它实现了跨框架、跨平台和跨硬件的推理支持，允许用户在 PyTorch、TensorFlow 等训练框架之间自由切换，并部署到 CPU、GPU、VPU 等多种设备上。其核心优势包括图优化（如常量折叠、算子融合）、低延迟、高吞吐，以及对多语言和平台的广泛支持。通过插件式 Execution Provider 架构，ONNX Runtime 可接入 CUDA、DirectML 等设备级后端，并支持将子图委托给 TensorRT、OpenVINO 等推理引擎执行，成为工业界部署深度学习模型的统一推理平台。

1. 基本结构

ONNX Runtime 的系统结构分为多个模块，每个模块负责特定的功能，从模型加载到推理执行，具体结构如图 5.4 所示。

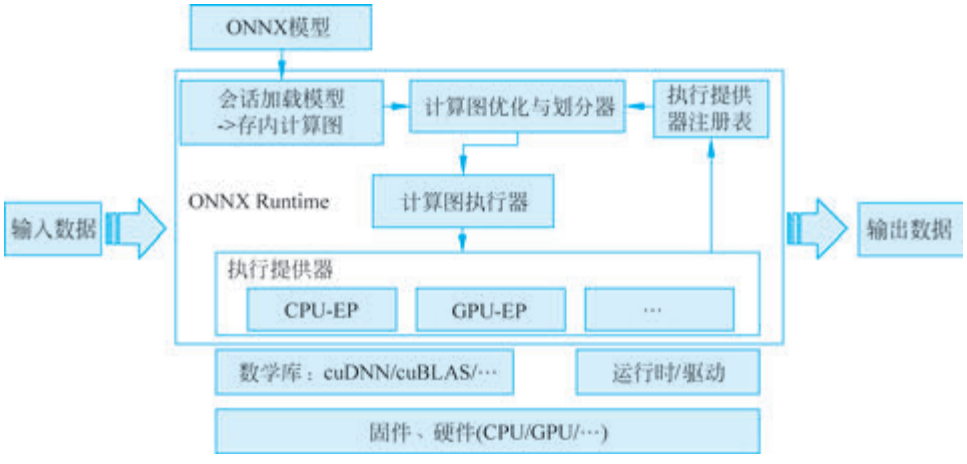

图 5.4 ONNX Runtime 基本结构

以下是对 ONNX Runtime 系统结构中各模块的分段描述：

1）输入/输出数据

推理输入数据，可以是图像、语音、文本等张量格式，由用户提供给 ONNX Runtime，作为模型计算的起点。推理的最终输出结果，可能是分类概率、边界框、文本序列等，由 ONNX Runtime 返回给用户应用程序。

2) ONNX 模型

ONNX 模型是系统的起点,包含完整的计算图、节点定义、权重参数、输入输出信息等。它以标准的 .onnx 格式存储,确保模型可以跨深度学习框架使用。此模型在加载后被解析为内存图结构,供后续优化与执行。

3) 会话加载模型→存内计算图(In-Memory Graph)

该模块负责在会话(Session)初始化阶段加载 ONNX 模型,并将其解析为内存中的计算图结构,包含所有计算节点、输入输出张量、初始权重等。在这一阶段,系统会整理模型的拓扑结构和节点依赖,为图分区和执行做好准备。它是后续图划分和运行器操作的核心数据结构。

4) 计算图优化与划分器(Graph Optimizer and Partitioner)

此模块负责将 In-Memory Graph 进行优化和子图划分。它根据各执行提供器(Execution Provider,EP)的算子支持能力和约束条件,对计算图进行子图划分,如算子融合、裁剪等。划分出的子图将传递给执行器处理。

5) 计算图执行器(Graph Executor)

该执行器执行已划分的多个子图,支持并行执行以提高吞吐性能。它协调不同子图间的依赖关系,确保正确的数据传递顺序,并调度执行提供器执行具体计算任务。

6) 执行提供器

EP 是负责模型实际计算的执行单元,代表不同的硬件后端,如 CPU、CUDA(GPU)、TensorRT 等。每个 EP 实现其所支持的 ONNX 算子的硬件优化版本,并向执行提供器注册表(Provider Registry)注册自己,以便运行器调度使用。

7) GPU-EP

GPU-EP(不同厂商,名词不同)是专为 GPU 构建的 EP 实现,调用底层 GPU 库(如 cuDNN、cuBLAS)来执行加速计算。它负责将子图转换为 GPU 上的高效操作,并通过驱动与硬件通信完成执行。

8) 执行提供器注册表

该模块管理系统中所有已注册的 EP,为图划分和执行阶段提供 EP 能力查询与匹配支持。

9) 数学库

数学库是计算所依赖的基础库,如 cuDNN 提供卷积、归一化操作,cuBLAS 提供矩阵运算支持。数学库为 GPU-EP 提供核心的高性能数值计算能力,通过运行时 API 间接调用底层驱动完成任务执行。

10) 运行时/驱动

运行时/驱动是连接 EP 与底层硬件的桥梁,其中运行时提供针对特定硬件平台的高级编程接口(如 CUDA Runtime、ROCm Runtime),负责任务管理、内存分配和调用调度;驱动则直接与硬件通信,将这些调用翻译成设备可执行的指令,管理数据传输、硬件资源和执行状态,从而完成计算并将结果返回给上层。

11) 固件与硬件

固件是嵌入在硬件内部的低级控制程序,用于初始化设备、管理资源并执行基本控制逻辑,确保硬件能正确响应驱动下发的指令;硬件是实际执行计算任务的物理设备,如 CPU、

GPU、NPU、FPGA 等，通过固件和驱动的协同控制完成深度学习所需的各类运算。

这个系统架构实现了模型从标准表示到硬件执行的完整流程，通过模块化的设计有效支持多平台、高性能推理。

2. 执行流程

以下是 ONNX Runtime 推理执行流程的分段描述，系统性地呈现从模型加载到输出结果的全过程。

1）模型加载与解析

ONNX Runtime 的推理流程从加载. onnx 文件开始。首先，使用 onnxruntime. InferenceSession 加载模型，解析其包含的 ModelProto 数据结构。解析过程中会提取模型的计算图、节点结构、权重参数、输入/输出信息等。同时，还会校验模型的合法性，如验证 ONNX opset 版本是否兼容，并检查所有节点是否受支持的执行提供器覆盖。

2）图优化与分区

加载完成后，系统进入图优化阶段。优化器会进行常量折叠、算子融合、死节点移除等静态图优化操作，简化计算图，提高执行效率。随后，图划分器会分析计算图中的每个节点，依据硬件支持能力（如是否支持 GPU/CPU）将其分配给对应的执行提供器（如 CPU、CUDA、TensorRT），划分出多个可独立执行的子图。

3）执行计划生成

图优化和分区完成后，执行规划器（Execution Planner）生成整个模型的执行计划。该计划定义了各个子图和节点的调度顺序，并分配所需计算资源。此阶段还包括张量内存管理：Tensor 分配器（Tensor Allocator）会智能分配并复用中间张量内存，以节省资源并提升推理效率。

4）推理执行

计算图的执行由 ONNX Runtime 内部的执行器（Executor）负责。该模块根据拓扑顺序协调子图执行，如果多个执行提供器同时参与，还要负责跨设备数据传输。每个子图在执行时由其绑定的执行提供器负责调度，具体算子由各 EP 提供的算子实现，这些实现通常基于 cuDNN、cuBLAS、MKL 等硬件优化库。

5）输出结果处理

所有计算完成后，系统从内存中提取模型输出张量，由输出管理器（Output Manager）整合这些结果返回给上层应用。应用可继续对这些张量执行后处理操作，如将 logits 映射为标签、进行分类后排序或提取关键点坐标等，从而完成完整的推理任务。

这个流程展示了 ONNX Runtime 如何将模型结构、高效执行策略与多硬件支持整合为一体，形成一个高度模块化且高性能的推理引擎。

3. 编译

在 ONNX Runtime 中，大多数模型推理流程并不需要显式编译。对于常规场景，如使用默认的 CPU Execution Provider 或内置的 GPU 加速（如 CUDA），ONNX Runtime 提供了预编译的动态库和算子实现，模型可以直接加载并执行，无需用户干预。这种即开即用的机制大大简化了部署流程，使得 ONNX Runtime 成为跨平台、高效、低门槛的推理引擎。

然而，在某些特定情况下，推理过程可能会触发“编译”行为。首先是自定义算子场景，用户若使用了标准 ONNX 不支持的操作，需要自行用 C++编写并编译为共享库，再注册到

ONNX Runtime。其次,部分硬件加速后端如 TensorRT、OpenVINO 或 Vulkan 等,可能在首次加载时执行与后端相关的即时或离线编译流程,生成平台专用的执行表示并进行缓冲,后续通过缓存机制复用优化结果。此外,针对如 FPGA 等硬件,ONNX Runtime 也可能触发专用的编译流程,以生成底层硬件配置文件或中间表示。

综上,虽然 ONNX Runtime 本身是预编译的运行时引擎,绝大多数使用场景无须额外编译,但在涉及定制化或硬件深度适配时,其架构也支持按需编译机制,实现更高的性能与兼容性。

4. 典型应用场景

ONNX Runtime 广泛应用于高性能推理场景,包括语音识别、图像处理和自然语言处理等需要低延迟的任务。它支持在云平台和边缘设备上部署,兼容多种硬件(如 CPU、GPU、FPGA),无须修改模型即可迁移运行。开发者可直接运行从 PyTorch、TensorFlow 等框架导出的模型,适合用于生产部署、跨平台迁移和长期维护的 AI 系统。

5. 优缺点

ONNX Runtime 是一个高性能的推理引擎,支持多种操作系统和硬件平台,能与多种训练框架(如 PyTorch、TensorFlow)配合使用,实现跨框架、跨平台的高效部署。它提供多语言接口,支持图优化、算子融合、自定义算子扩展等技术,提升推理效率,非常适合云端和边缘设备部署。

但 ONNX Runtime 也有局限,如对某些特殊算子或动态图结构支持不足,调试不如原生框架灵活,且性能优化依赖特定硬件加速器。整体来看,ONNX Runtime 是部署阶段的有力工具,但导出规范和调试能力仍有提升空间。

6. 总结

ONNX Runtime 是为解决模型在不同框架和硬件间迁移与部署难题而诞生的高性能推理引擎。它不仅支持多种训练框架导出的 ONNX 模型,还提供高效的图优化能力和广泛的硬件后端支持(如 CPU、GPU、TensorRT、OpenVINO 等),大幅提升推理性能与部署灵活性。作为连接训练与推理的重要桥梁,ONNX Runtime 在工业级深度学习部署中发挥着关键作用。


视频讲解

5.5 硬件驱动系统

显卡驱动程序是现代计算系统中不可或缺的系统软件组件,处于操作系统与 GPU 硬件之间的关键中间层,负责协调主机(Host)与显卡设备(Device)之间的通信与协作。基于 CUDA(NVIDIA)和 ROCm(AMD)架构的显卡驱动程序,不仅提供统一的编程接口,屏蔽底层硬件差异,还通过资源管理与任务调度机制,最大化 GPU 的计算性能和带宽利用率。其主要作用包括:对 GPU 设备进行初始化和属性查询、动态分配与回收全局内存、调度线程块并启动 GPU 内核任务(kernel)、实现主机与设备间以及多 GPU 之间的高效数据传输(支持异步传输和 P2P 通信,如 NVLink 和 Infinity Fabric),同时也负责处理运行时错误、提供调试与诊断信息,并支持性能监控与硬件级优化。典型接口如 cudaMalloc()、cudaMemcpy()、cudaLaunchKernel()、cudaDeviceEnablePeerAccess()等,均由运行时库封装驱动接口实现并提供给上层运行时系统与深度学习框架调用。通过这些功能,显卡驱动程序不仅支撑起

图形渲染与 AI 推理/训练等高负载场景，还为异构计算环境中的多设备协同、性能调优和系统稳定性提供了坚实基础，是构建高性能 GPU 加速平台的核心组成。

1. 驱动架构

GPU 驱动架构一般分为如表 5.3 所示的 3 层。

表 5.3 GPU 一般驱动架构

层 级	主要模块	功 能
用户空间层	用户级 API(如 CUDA/ROCm Runtime API)、调试工具接口	提供用户访问 GPU 功能的高级接口，屏蔽底层硬件复杂性
内核空间层	设备管理模块、内存管理模块、任务调度模块、通信模块	实现硬件资源管理、任务分配、地址映射和数据传输的核心逻辑
硬件控制层	硬件接口模块(如 PCIe 驱动、内存控制器接口)	直接控制 GPU 硬件资源，支持与主机和设备之间的通信与操作

1) 用户空间层

用户空间层为开发者提供高层接口和工具支持，主要包括用户 API、调试分析工具和运行时库。开发者通过如 cudaMalloc()、cudaMemcpy()、cudaLaunchKernel()(CUDA)或对应的 HIP 接口(ROCm)进行内存管理、任务调度与数据传输。运行时库(如 CUDA Runtime、HIP)封装了底层驱动调用，简化了编程过程。同时，工具如 NVIDIA Nsight、nvidia-smi 或 ROCm Profiler 提供性能监控、调试与诊断功能。该层将用户请求转换为系统调用，屏蔽硬件细节，为高效开发提供了良好支撑。

2) 内核空间层

内核空间层是显卡驱动的核心控制中心，运行在操作系统内核态，直接负责设备初始化、内存管理、任务调度和数据通信。设备管理模块控制 GPU 启动与资源分配，内存模块处理显存的物理分配与虚拟映射，调度模块将计算任务分配至 GPU 执行单元，通信模块则实现主机与多 GPU 间的高效数据传输。该层连接用户请求与硬件执行，是确保 GPU 稳定高效运行的关键中介。

3) 硬件控制层

硬件控制层直接对接 GPU 物理硬件，是驱动系统中最底层的执行部分。它通过 PCIe、NVLink 等硬件接口模块实现主机与设备、设备之间的低延迟通信；通过内存控制模块管理显存资源和访问安全；通过任务控制模块精确调度流式多处理器(SM 或 CU)执行指令。该层将内核空间的指令转换为硬件动作，确保计算任务在硬件上高效、可靠地完成，是实现 GPU 全面功能的最终执行环节。

2. 工作流程

以下是显卡驱动程序的完整工作流程，分为初始化、任务执行和数据传输与结果返回 3 个阶段，每个阶段用一段话进行描述。

1) 初始化

在系统启动后，GPU 首先由固件完成底层硬件模块的初始化(如电源管理、内存控制器、PCIe 接口等)。随后，操作系统加载 GPU 驱动程序，初始化内核空间中的设备管理模块，识别并注册 GPU 设备。接着，用户空间的运行时库(如 CUDA Runtime 或 ROCm HIP)被加载，准备好高级接口供开发者调用。整个初始化阶段为后续的资源分配与任务执

行打下基础,确保软硬件协同运行环境的就绪。

2) 任务执行

用户通过高级 API(如 cudaLaunchKernel()或 hipLaunchKernel())发起计算请求,运行时库将该请求解析为底层内核调用,并配置内存管理和数据传输操作(如 cudaMalloc()、cudaMemcpy())。驱动内核空间中的调度模块随后接管,负责将计算任务分配到 GPU 上,并由硬件控制层启动相应的计算单元(如 SM、CU)进行实际执行。这一阶段体现了从用户请求到硬件任务执行的完整映射与调度过程。

3) 数据传输与结果返回

任务执行完成后,内核空间通过直接内存访问(DMA)机制将结果数据从 GPU 内存传输回主机内存,支持同步或异步的数据传输策略。在传输完成后,用户空间运行时收到返回信号,包括任务执行状态和计算结果。该阶段实现了从硬件执行结果回传到用户可用输出的闭环,标志着一次 GPU 计算任务的完整生命周期结束。

3. 与底层通信方式

GPU 驱动、固件和硬件之间的通信是 GPU 系统工作的关键。通信方式的设计旨在高效、安全地完成任务传递、硬件控制和状态反馈。如图 5.5 以某厂家的 GPU 为例,详细描述驱动与固件、硬件的通信方式及其主要任务。

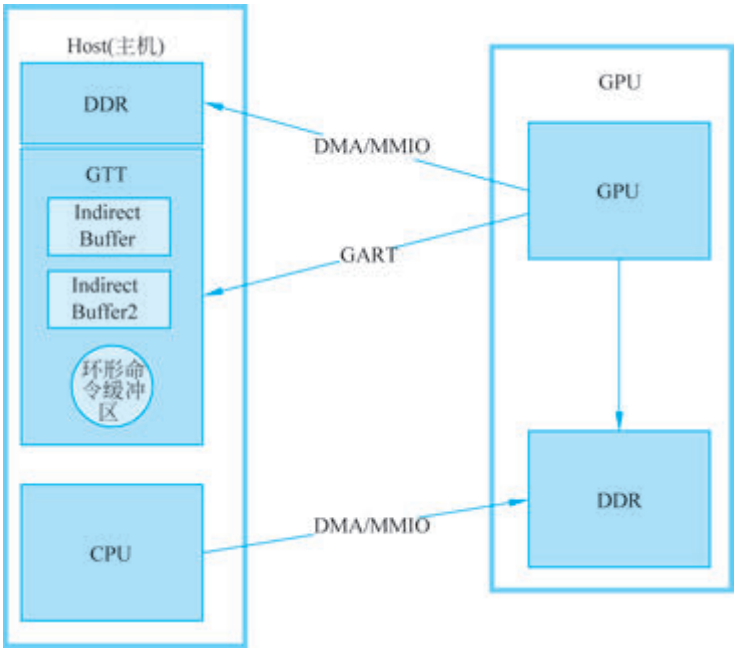

图 5.5 通信方式基本结构

1) 驱动与固件的通信方式

以下是 GPU 驱动与固件之间常见通信机制的分段描述,涵盖 4 种典型方式:基于寄存器的通信、基于共享内存的通信、中断机制和消息队列机制。

(1) 基于寄存器的通信。

在寄存器通信机制中,驱动通过内存映射 I/O(Memory-Mapped I/O,MMIO)方式直接访问 GPU 的控制寄存器,以完成任务配置和状态读取。GPU 固件持续监听这些寄存器

的值，当寄存器发生变化时即触发任务执行或设备响应。主要任务包括：驱动向寄存器写入内核启动配置（如线程块、网格维度）以发起任务请求，固件通过寄存器反馈任务执行的结果或错误状态，同时支持设备初始化、复位等控制操作。这种通信方式延迟低、控制精确，适合小规模、高实时性的指令传递。

（2）基于共享内存的通信。

该机制利用图形地址重映射表（Graphics Address Remapping Table，GART）和图形转换表（Graphics Translation Table，GTT）支持 Host 与 GPU 对相同物理内存的访问，通过共享内存区实现驱动与固件之间的数据交互。共享内存通常作为任务队列或中间缓冲区，驱动将任务描述（如矩阵乘法参数、数据传输结构）写入共享内存，固件轮询解析任务并执行，同时将结果或进度写回共享区域。GTT 主要实现主机侧内存映射，而 GART 实现 GPU 显存地址重定向，共同支持异构系统中的高效共享访问。这种通信方式适合传输大批量任务数据，带宽利用率高，支持复杂配置同步。

（3）中断机制。

中断机制用于固件向驱动发起主动通信，典型场景包括任务完成通知、错误上报或设备状态变更。GPU 固件在检测到特定事件［如计算完成、纠错码（Error Correcting Code，ECC）错误］后，触发中断信号传递至 CPU，驱动的中断处理函数响应该事件并读取必要状态或执行后续任务。这种方式保证了关键事件的实时反馈，避免驱动轮询浪费资源，适合用于高优先级、异步事件通知。驱动也可借助中断机制实现多任务调度与系统监控，如温度、电源状态等告警通知。

（4）消息队列机制。

驱动与固件之间通过共享内存中的环形缓冲区（Ring Buffer）构建异步消息队列，用于高效传输任务描述和状态信息。驱动将待执行的任务写入消息队列，固件按照队列顺序处理任务，并将执行结果、状态更新等反馈写回队列。该机制支持多条任务排队、并行处理与按序消费，具有良好的异步性与吞吐能力。消息队列也可用于维护多 GPU 协作中的状态同步与命令广播，是复杂调度系统中常用的通信方式之一。以下详细描述基于环形缓冲区的通信机制，其基本原理如图 5.6 所示。

环形缓冲区是一种高效的内存通信机制。它在硬件调度架构中扮演着核心角色，尤其适合需要高吞吐、低延迟、无锁通信的异构计算环境。其基本结构包括一段固定大小的共享内存、一个由驱动维护的写指针和一个由固件维护的读指针，两者共同管理数据在缓冲区中的流动。缓冲区被初始化在 GPU 的全局内存中，驱动通过设置物理地址或 GPU 虚拟地址让固件可访问，并将读写指针初始化为起始位置。

通信过程中，驱动首先检查空闲空间是否足够（避免覆盖尚未读取的任务），然后将任务描述（如任务 ID、参数地址、数据块信息）写入缓冲区，并推进写指针。固件则周期性地读取写入任务，解析任务内容并控制 GPU 硬件启动核函数、配置 DMA、执行内存复制等操作。任务完成后，固件将执行状态或中间结果写入缓冲区的另一部分区域（或状态缓存区），并可通过中断机制向驱动发出完成信号。

驱动在响应中断或定时检查状态后，读取缓冲区中的反馈信息，更新任务状态并将结果返回给用户态或上层中间件系统。整个通信过程支持循环利用，写指针到达末尾后自动回绕到起始地址；同时由于读写操作由不同实体控制（驱动写，固件读），避免了锁机制带来的

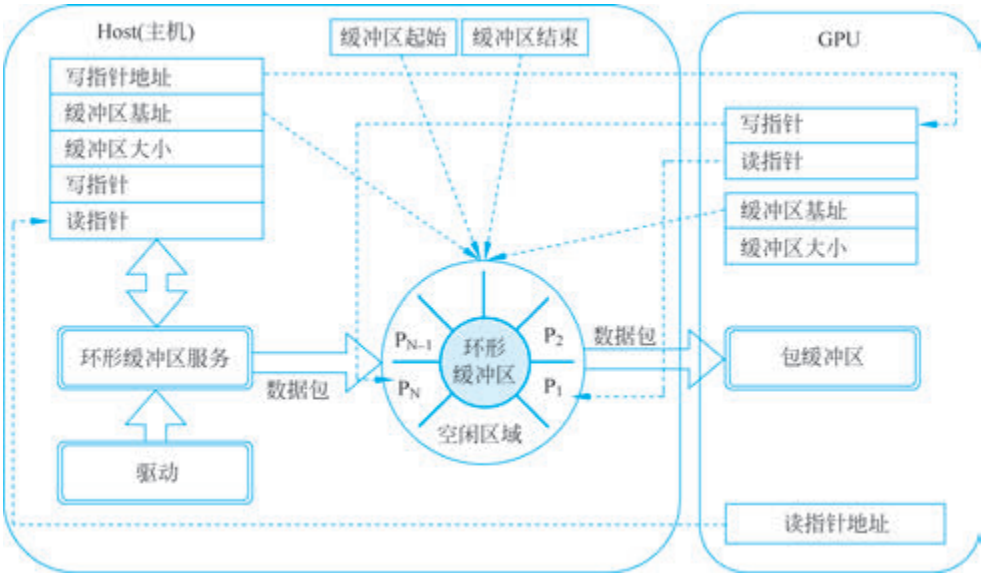

图 5.6 环形缓冲区通信基本原理

性能开销,实现无阻塞、高并发通信。

此外,环形缓冲区可扩展支持多个任务并发、优先级队列、状态分类(如成功、失败、中间状态),甚至支持固件回写新的命令请求,实现双向交互。它也可嵌入更复杂的 GPU 驱动框架中,如 Command Processor、Task Scheduler、Async Execution Engine 等模块,支撑多流并发、GPU 多引擎调度和统一任务池管理,是现代 GPU 软件栈中不可或缺的底层通信机制。

总之,这 4 种机制在实际系统中往往协同使用,如寄存器发起任务、中断反馈完成、共享内存存放数据、消息队列维护任务状态,共同构成 AMD GPU 驱动与固件高效通信的底层基础。

2) 驱动与硬件的通信方式

以下是 GPU 驱动与硬件之间常见通信方式的分段描述,涵盖控制寄存器、直接内存访问(Direct Memory Access,DMA)、总线通信(如 NVLink/PCIe)及时钟与电源控制 4 种机制。

(1) 控制寄存器。

驱动通过内存映射 I/O(MMIO)方式访问硬件控制寄存器,设置特定的指令与配置值,触发 GPU 执行相关操作。典型任务包括初始化 GPU 模块、启动核函数(如通过配置网格维度、线程块大小等参数),以及控制 DMA 通道的行为。硬件在侦测到寄存器变更后立即响应并执行对应功能,这种通信方式反应迅速、控制精细,是 GPU 任务调度与资源初始化的核心手段之一。

(2) 直接内存访问。

DMA 是一种无需 CPU 介入的高速数据传输机制,驱动通过配置 DMA 通道控制主机与设备之间或设备内部的内存复制。主机到设备的数据加载、设备结果回传主机,以及在多 GPU 环境下的点对点通信(P2P)都依赖于 DMA 实现高带宽、低延迟的数据流。通过异步

DMA 操作，GPU 可在数据传输与计算任务之间实现流水线式并行，提高整体执行效率。

（3）NVLink/PCIe。

驱动依托 NVLink 或 PCIe 总线与 GPU 建立通信，负责配置链路、优化带宽与延迟，协调主机与设备、设备与设备之间的传输。PCIe 是最常见的通用 GPU 接口，支持主机与显存的数据交互；而 NVLink 提供更高带宽的专用互连，适用于多 GPU 间的大规模数据传输，如深度学习模型的多卡训练同步等任务。驱动层可管理链路拓扑、通道复用、通信优先级等策略，保障传输的稳定与高效。

（4）时钟与电源控制。

驱动通过访问 GPU 的时钟管理单元和电源管理模块（如 PMU），动态调节核心频率与电压，实现按需供能（DVFS）策略。在低负载场景下，驱动降低频率以节能；在高负载或高温条件下，则触发主动降频或散热机制，以保障系统稳定运行。这些功能通常与温度传感器、功率监控单元协同工作，实现对 GPU 功耗与热设计（TDP）的全面管理，是实现智能节能与系统保护的重要通信路径。

这些通信机制共同构成了 GPU 驱动与硬件之间的完整控制链路，既涵盖了任务调度与数据交互，也支持对能耗和热状态的精细管理，是实现高性能、稳定 GPU 系统运行的关键基础。

3）驱动与固件、硬件通信的核心特点

高效性：使用寄存器和 DMA 通道等硬件加速通信，减少延迟。

实时性：中断机制确保任务完成和错误能够实时通知驱动。

灵活性：共享内存和消息队列允许复杂任务参数和结果的传递，支持多任务并行。

可靠性：寄存器通信和中断结合，实现了驱动和硬件间的错误反馈与恢复机制。

4. 通信消息的应用场景

表 5.4 总结了通信消息在 GPU 驱动与固件之间的典型应用场景，展示了从设备启动、任务调度到错误处理等各环节中，不同类型的消息在系统运行中的关键作用。

表 5.4 驱动与固件消息典型应用场景

场　景	通 信 消 息	示 例 任 务
设备启动与初始化	硬件初始化消息	配置内存控制器、初始化计算单元
动态性能优化	时钟与电源管理消息	根据负载调整时钟频率、切换高性能模式
任务执行与调度	任务控制消息、硬件资源管理消息	启动内核任务，分配线程块
主机与设备数据传输	数据传输控制消息	配置 DMA 通道，传输模型参数或计算结果
实时任务完成通知	中断管理消息	通知驱动任务已完成，准备启动下一个任务
硬件状态监控与优化	状态监控与反馈消息	监控温度、电压，调整硬件参数以提高性能
错误处理与恢复	错误报告消息	捕获 ECC 校验错误，通知驱动重新分配内存或重启任务

5. 总结

显卡驱动程序是 GPU 计算生态的重要组成部分，负责管理硬件资源、协调任务执行和提供高性能接口。它通过模块化设计实现设备管理、内存管理和任务调度等核心功能，连接了硬件和高层应用。虽然 CUDA 和 ROCm 的具体实现有所不同，但在概念和功能上具有

较高的一致性。未来,显卡驱动程序的发展将重点关注跨平台兼容性、支持新型异构计算架构和提升编程易用性。

5.6 显卡固件系统

GPU 固件是嵌入式运行在 GPU 上的底层软件系统,位于驱动程序与硬件之间,起到初始化、控制与调度硬件资源的关键作用。它在设备上电启动后由启动引导代码加载,负责完成计算单元、内存控制器、电源管理模块等硬件的初始化与自检,并持续管理硬件运行状态,包括时钟频率调整、温度监控和功耗控制。固件还承担任务调度功能,根据驱动提交的任务请求,将任务分配至合适的硬件资源上执行,并协调资源使用以提升并发效率。同时,它管理与主机之间的数据通信,并支持多 GPU 间的点对点交换。在异常发生时,固件具备错误检测和热重启能力,可及时反馈错误信息给驱动并尝试自动恢复。作为硬件抽象层的重要组成部分,GPU 固件保障了系统运行的稳定性、高效性与可扩展性,是现代 GPU 架构不可或缺的核心组件。

1. 固件架构

GPU 固件是 GPU 硬件与驱动之间的重要桥梁,负责硬件初始化、任务调度、数据管理、状态监控和错误处理等功能。固件的软件架构通常采用模块化和分层设计,以满足复杂任务的高效管理和硬件资源的灵活调度需求,如图 5.7 所示。当然,每个 GPU 场景的实现细节会有不同,本章以一个概念性的固件系统来展示 GPU 通用计算固件系统的工作原理,以帮助读者理解整体的系统架构。

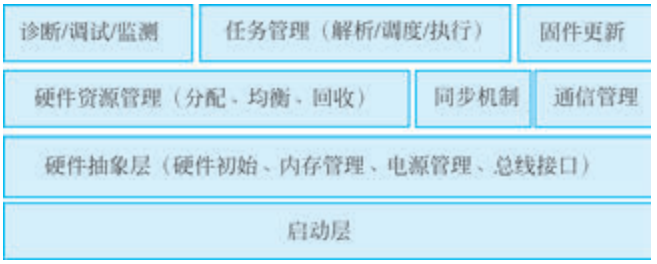

图 5.7 固件的软件架构

整个 GPU 固件架构由启动层引导系统运行,硬件抽象层屏蔽硬件差异,资源管理与任务管理模块确保并行执行效率,同步机制与通信管理提供高效调度能力,固件更新和诊断/调试/监测则确保系统可维护、可扩展。这种模块化设计确保固件能够在面对复杂任务负载、多设备协作和动态运行环境时,依然保持高性能、稳定和安全,是现代 GPU 系统不可或缺的核心控制平台。

1) 启动层(Boot Layer)

启动层是 GPU 上电后的第一个执行阶段,负责整个系统的基本初始化工作。它设置 GPU 时钟频率、总线接口、内存控制器以及电源管理模块,确保硬件各子系统在上层固件启动前已准备就绪。此外,启动层还负责加载主固件映像,并完成安全验证(如签名校验)以防止恶意固件篡改,是 GPU 系统启动的核心保障机制。

2）硬件抽象层（HAL）

硬件抽象层是固件与底层硬件之间的中介层，向上层提供统一的接口，屏蔽底层硬件实现差异。它包含图形/计算单元初始化、显存管理、电源状态调节、总线通信控制（PCIe、NVLink）等模块。该层确保不同 GPU 架构或代际变更对上层逻辑透明，是整个固件系统运行的硬件控制核心。

3）硬件资源管理

该模块对 GPU 内部的资源（如计算单元、内存带宽、调度队列等）进行全局调度与生命周期管理。它根据任务优先级、资源使用情况动态分配资源，避免冲突和拥塞，同时在任务完成后回收空闲资源，确保多任务执行的公平性和高效性，是并行计算场景下的基础支撑层。

4）任务管理（解析/调度/执行）

任务管理是固件架构中的核心功能，负责从驱动接收任务请求后，进行解析、调度和下发执行。它可处理图形渲染命令、通用计算任务（如 GPGPU 核函数），并依据任务类型和优先级调度至对应的硬件单元。任务管理系统的高效与否，直接决定了 GPU 整体性能和响应能力。

5）同步机制

GPU 内部任务常处于并行状态，必须依赖同步机制保障数据一致性和执行顺序。固件通过信号量、屏障（Barrier）、原子操作等机制协调多个线程/核函数之间的通信，防止写冲突、读写不一致，是多线程安全的保障基础。

6）通信管理

通信管理负责驱动与固件、GPU 与主机系统（CPU），以及 GPU 间的数据交互。它通过控制总线（如 PCIe、NVLink）和 DMA 通道，优化数据传输路径、调度带宽分配，并实现低延迟通信调度。该模块对大规模异构计算（如多 GPU 训练）尤为关键，是系统级协同计算的枢纽。

7）固件更新

为提升系统功能与安全性，GPU 固件需支持可更新机制。固件更新模块支持本地或远程升级、版本回滚和故障恢复，确保更新过程的可靠性与容错能力。它为 GPU 提供生命周期内的长期演进能力，是应对新功能需求和修复漏洞的重要保障。

8）诊断/调试/监测

该模块提供对 GPU 运行状态的全面监控能力，包括电压、温度、功耗、频率等运行参数。它支持硬件级调试接口和错误码回溯，便于开发和维护人员排查故障。同时，它还能分析任务性能瓶颈和资源使用情况，为固件优化与系统评估提供数据支撑。

2. 工作流程

1）执行任务流程

（1）任务接收：固件接收驱动程序传递的任务请求。检查任务参数是否合法。

（2）资源分配：根据任务需求分配 GPU 的计算资源、内存资源。优化任务执行顺序，减少硬件资源冲突。

（3）任务调度与执行：将任务分配到计算单元（如 SM 或 CU），并启动任务执行。管理任务之间的依赖关系。

(4) 通信支持：管理任务执行中的主机与 GPU、GPU 间数据传输。

(5) 状态监控：在任务执行过程中监控硬件状态(如功耗、温度)，并动态调整配置以优化性能。

(6) 任务完成与结果返回：任务执行完成后，将结果写入指定内存地址并通知驱动程序。

2) 初始化流程

(1) 硬件自检：在 GPU 上电时，固件启动并对硬件模块进行自检(如内存、计算单元)。

(2) 时钟和电源初始化：设置 GPU 的默认时钟频率和电压。

(3) 内存初始化：初始化显存控制器与地址映射/页表等机制，为后续显存分配与访问提供基础。

(4) 通信接口初始化：配置主机与 GPU 的通信接口(如 PCIe 或 NVLink)。

(5) 核心功能加载：加载任务调度模块、电源管理模块等核心功能。

3. 任务调度机制

GPU 固件的任务调度机制采用多级调度队列架构，旨在实现高效的资源利用、任务并行处理和优先级控制。调度系统通过动态监测计算单元、内存、带宽等硬件资源状态，合理分配任务并在多核资源上并发执行，同时支持核函数计算、数据传输、同步操作等多类型任务。该机制具备优先级队列管理与抢占能力，确保高优先级任务能够实时响应；同时内置错误检测与恢复机制，提高系统的容错性和鲁棒性。其结构设计灵活，可根据任务类型与运行环境动态调整调度策略，确保 GPU 在高负载和复杂并发场景下保持稳定、高效的运行状态。

任务调度系统主要由全局任务队列、任务接收与分类模块、分类/优先级队列、任务解析器、任务分配器、任务执行器、状态监控模块、反馈模块及任务状态表等组件构成。这些组件协同工作，完成任务从接收、解析、资源分配、执行、状态监控到反馈的完整生命周期管理，实现任务调度的全流程闭环控制，支撑现代 GPU 在图形渲染、通用计算及异构协作等多样场景下的高性能执行需求。任务调度机制的基本流程如图 5.8 所示。

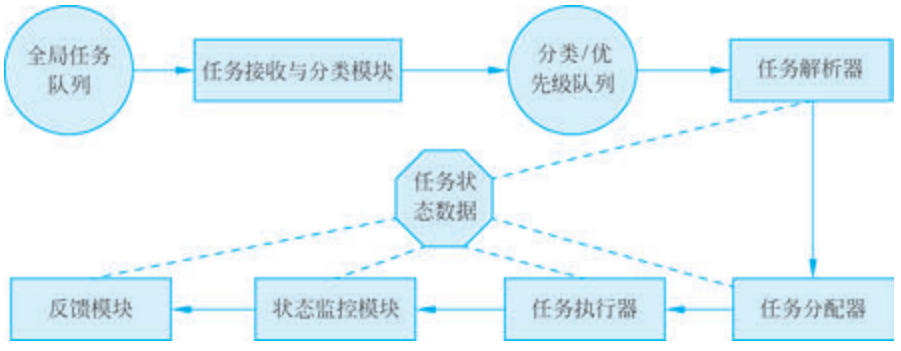

图 5.8 任务调度机制的基本流程

1) 全局任务队列

全局任务队列是任务调度的起点，用于集中存储驱动提交的所有任务描述符。驱动通过寄存器或共享内存将任务请求写入该队列，任务内容包括类型(如核函数、DMA 传输)、优先级、数据地址、执行参数等。该队列保证任务接收的顺序性和完整性，为后续的分类和

调度流程提供统一入口。

2）任务接收与分类模块

该模块从全局任务队列中读取任务，并根据任务描述符中的信息进行初步分类。一方面，按任务优先级划分，将高优先级任务优先送入处理通道；另一方面，按任务类型分类为计算类、传输类或同步类任务，为后续分配资源与优化执行路径做准备，是任务调度结构中的关键分流单元。

3）分类/优先级队列

此阶段将任务进一步细化管理，优先级队列确保高优任务能更快获得执行机会，而类型队列（如计算队列、DMA 队列）有助于按功能模块并行调度。队列调度器会根据当前资源状态及策略（如抢占、轮转）动态选择队头任务交由解析器处理，平衡效率与响应性。

4）任务解析器

任务解析器负责从队列中取出待执行任务，读取并提取任务核心信息，包括类型识别（如 kernel vs DMA）和参数（如线程块大小、内存地址、数据量等）。解析结果被结构化处理，供后续资源分配模块高效调用，是调度逻辑向执行逻辑的关键转换环节。

5）任务分配器

任务分配器根据解析后的任务需求和当前硬件资源状态（通过任务状态数据库实时更新）动态分配计算单元（如 SM/CU）、寄存器、共享内存、DMA 通道等。它需精确评估当前负载情况，确保资源合理分配，既不浪费也不造成饥饿，是任务可执行性的核心保障机制。

6）任务执行器

该模块根据任务分配器的指令配置硬件寄存器、载入核函数或 DMA 指令，并启动硬件资源执行任务。它负责控制底层执行流、执行时间窗口与资源调度，同时与状态监控模块协同，实时记录任务执行状态。其作用相当于“指挥员”，确保任务在硬件上按计划执行。

7）状态监控模块

状态监控模块持续跟踪任务执行状态（进行中、完成、错误等）及硬件运行状态（温度、电压、频率、功耗）。它通过底层传感器接口和硬件反馈机制不断刷新任务状态数据库，为系统调度、错误处理和能耗控制提供实时数据支持，是系统可靠运行的重要后盾。

8）反馈模块

反馈模块根据状态监控与数据库信息，构建反馈包发送给驱动程序。常见方式包括中断机制（触发立即响应）和共享内存更新（供驱动轮询读取）。反馈内容包括任务完成状态、错误类型、性能统计等，用于驱动决定是否重试、报告错误或调度新任务。

9）任务状态数据

作为系统中枢记录器，该数据实时更新每个任务的生命周期状态（待分配、执行中、完成、失败），并记录资源使用情况，供任务分配器、状态监控模块、反馈模块等使用。它提供任务调度逻辑中的全局上下文，是整个任务执行过程的动态信息基座。

这套结构化的调度系统实现了从任务接收到执行再到反馈的高效闭环机制，通过模块协作、实时状态更新与优先级控制，使 GPU 固件能够高效、稳定地管理复杂的并发任务。

4. 调度队列的组织

GPU 的任务调度队列采用了多级组织结构，通过不同维度的队列分类实现任务的高效调度和资源分配，调度队列组织形式如图 5.9 所示，每家 GPU 厂家实现方式会有不同，总

体都采用多级多优先级方式实现。

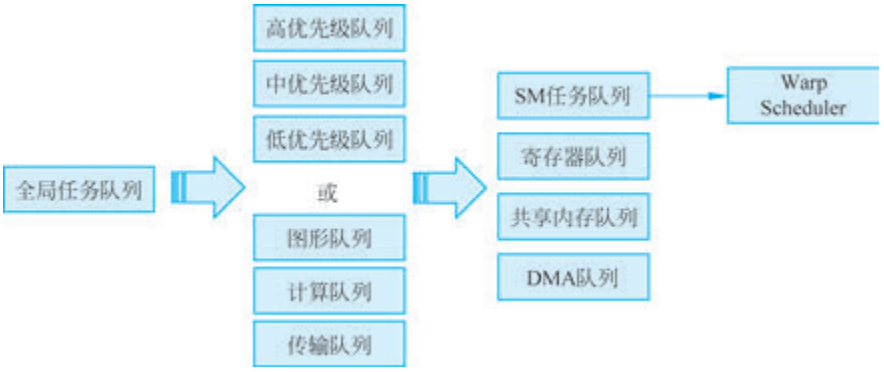

图 5.9 GPU 调度队列组织形式

如图 5.9 所示,GPU 调度系统采用三级队列结构。第一层是全局任务队列,负责接收驱动提交的所有任务;第二层是分类队列,根据任务优先级和功能类型进行初步分流;第三层是硬件资源队列,将任务映射到具体的执行资源上(如 SM、DMA、寄存器等),形成从逻辑到物理的调度通路。这种多级结构可提升调度系统的可扩展性与调度粒度控制能力。

1) 全局任务队列

全局任务队列是调度的入口,采用先进先出(First-In First-Out,FIFO)机制,保障任务按提交顺序进入调度系统。每个任务描述符包含任务类型、优先级、参数等元信息,便于后续分类与解析。队列中的任务被实时轮询或触发机制取出,导向优先级队列或类型队列进行进一步处理。

2) 分类队列

优先级队列用于根据任务的紧急程度(高、中、低)划分调度优先权,确保高优先级任务能够抢占资源并快速启动,适用于实时图像处理、低延迟计算等场景。任务类型队列将任务按功能类别划分为图形队列、计算队列、传输队列,便于不同任务利用专用硬件资源并行执行。该层次优化了任务路径规划,降低了资源争用与冲突,提高了整体吞吐效率。

3) 硬件资源队列

SM 任务队列,每个 SM 维护独立任务队列,Warp Scheduler 从中分派 warp 任务,适用于计算密集任务如深度学习核函数。寄存器队列,协调线程块对寄存器的分配,保证任务在编译约束与运行时资源动态变化下仍可执行。共享内存队列,为需线程间通信的任务动态分配共享内存,优化局部数据重用与带宽利用。DMA 队列,集中管理 GPU 内部或 GPU 与主机间的数据传输任务,利用 DMA 引擎实现高带宽、低 CPU 负担的数据搬移,是计算与传输并行调度的关键支点。

这种多层次、可组合、资源感知的调度队列系统,构建了 GPU 固件中任务调度的高效通道,使得在多任务、高负载、异构协同场景下,系统依然能维持高性能、低延迟与高度可控的运行态。

5. 调度优化策略

GPU 固件的调度优化策略围绕提高任务执行效率、系统响应能力与运行稳定性展开,主要包括 4 方面:首先,优先级调度机制确保关键任务优先执行,支持高优先级任务在资源

受限时进行资源抢占，提升系统实时性；其次，通过任务合并将多个小任务打包为大任务，减少调度频率和上下文切换开销，提升吞吐率；接着，动态负载均衡机制可根据当前硬件使用状态智能调整资源分配，支持多 GPU 系统中任务的跨设备迁移与调度，避免资源瓶颈或空转；最后，系统内建容错机制，在任务出错时支持自动重试或回滚，并生成错误日志反馈驱动，实现稳定、可靠的任务执行链路。这些优化策略相互配合，构建出一个高效、自适应、可恢复的 GPU 调度系统。

6. 硬件资源管理

GPU 的硬件资源管理是确保 GPU 硬件资源在多任务、多线程和高负载环境中高效分配、使用和回收的关键模块。它是 GPU 固件和架构的重要组成部分，为任务调度和执行提供基础支持。

1）硬件资源的分类

GPU 内部的硬件资源可分为计算资源、内存资源、带宽资源、电源与热管理资源及调度资源五大类。计算资源包括流处理器、张量核心和专用单元，负责各种计算任务的执行；内存资源涵盖显存、共享内存与多级缓存，支撑数据存储与高速访问；带宽资源连接各个处理单元与内存体系，决定数据传输效率，涵盖显存带宽、内部总线和主机互连带宽（如 PCIe 或 NVLink）；电源与热管理资源通过动态电压频率调整（DVFS）和功耗控制，确保 GPU 稳定运行；调度资源则管理任务的提交与切换，包括执行队列和上下文切换机制。以上资源共同构成 GPU 任务调度与执行的基础，协调运行以实现高性能、低延迟和高能效的计算目标。

2）硬件资源管理的主要功能

GPU 硬件资源管理旨在高效分配与协调计算、内存等资源以支撑多任务并发执行。系统结合静态与动态机制，依据任务需求灵活分配流处理器、张量核心与显存，并通过负载感知实现资源均衡与热管理，避免热点和资源浪费。任务完成后，资源回收机制及时释放显存与计算单元，防止泄露。为提升效率，GPU 固件引入 DVFS 调节功耗，采用显存压缩、预取等内存优化技术，并支持多任务动态调度，保障关键任务响应和系统吞吐。尽管机制完善，仍面临资源冲突、带宽瓶颈、功耗限制和高实时性需求等挑战，需依赖智能调度、带宽优化和快速响应策略加以应对。

7. 总结

GPU 固件系统是 GPU 硬件功能的核心控制组件，它在 GPU 启动、任务执行、资源管理和错误恢复中扮演关键角色。通过硬件抽象和优化，固件为驱动程序和上层应用提供了稳定高效的接口。然而，固件的设计复杂性和硬件依赖性也对其扩展和维护提出了挑战。随着 GPU 架构的发展，固件将更加注重支持异构计算和优化资源管理，从而进一步提升 GPU 的计算能力和可靠性。

5.7 底层软件系统关键技术


视频讲解

5.7.1 内存管理技术

现代高性能计算框架（如 CUDA 和 ROCm）对内存管理与虚拟化进行了高度优化，以

便有效利用硬件资源,减少开发复杂性,提升计算效率。以下是内存管理与虚拟化的核心原理和具体接口说明。


视频讲解

1. 内存结构

GPU 被用作处理通用计算任务的高性能计算设备。为了实现高效的计算,GPU 设计了一套复杂的内存结构,如图 5.10 所示,支持并行处理、多线程访问和高效的数据传输。分层的内存结构目的是在大规模并行计算中最大限度地提高带宽利用率和数据访问效率。

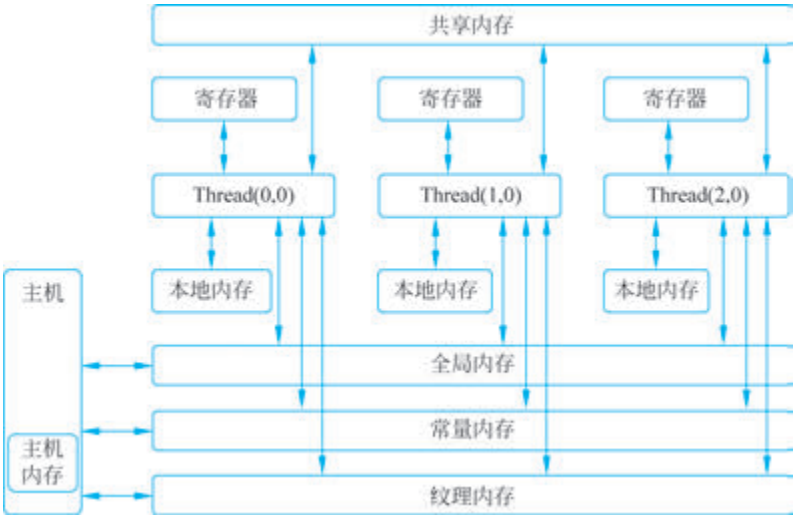

图 5.10 GPU 内存基本结构

1) 全局内存(Global Memory)

全局内存是 GPU 中容量最大的存储区域,通常为几吉比特至数十吉比特,所有线程都可以访问。其访问延迟较高(数百个时钟周期,具体取决于架构及缓存命中情况),但适合存储大规模输入数据、输出结果和中间计算值。为了提高带宽利用率,应通过内存对齐和访问合并(Coalescing)策略减少访问开销。在核函数中频繁访问的全局内存应尽可能被缓存或搬运到共享内存中以降低延迟。

2) 共享内存(Shared Memory)

共享内存是线程块(Thread Block)内部的共享存储区域,容量较小(如 48KB 或 96KB,依 GPU 架构而异),但具有极低访问延迟(数十个周期)。它适合线程间数据协作与复用,如矩阵计算中的分块乘法。使用共享内存能显著减少对高延迟全局内存的访问,是提升并行计算性能的关键手段之一。

3) 寄存器(Registers)

寄存器是每个线程的私有高速存储空间,访问延迟极低,适用于存放中间变量和局部数据。每个线程可用寄存器数量有限(通常几十个),超出部分将被“溢出”至本地内存,从而导致性能下降。因此优化寄存器使用、减少变量生命周期重叠是提升计算效率的重要策略。

4) 常量内存(Constant Memory)

常量内存是一块只读的共享存储区,容量通常为 64KB,适用于存储在程序执行过程中不变的参数(如卷积核、查找表)。它支持广播机制,即多个线程可同时读取相同地址的数据,在同一线程束(warp)内访问相同地址时支持广播机制,延迟很低。由于其只读特性,适

合用于共享全局常量而非线程间可变数据。

5）纹理内存（Texture Memory）

纹理内存是专为处理图像和二维数据设计的内存通道，具备专用缓存结构，优化了随机访问、插值和边界处理等图像相关操作。尤其在图形处理、图像滤波或深度学习中的输入预处理阶段具有显著性能优势。它们由硬件纹理单元支持，支持特定的取样方式与空间访问模式。

6）主机内存（Host Memory）

主机内存是CPU管理的系统内存，GPU与其通过PCIe或NVLink等互连总线进行通信。主机内存通常用于存储输入数据、读取结果、模型权重等。在数据传输密集型任务中，优化主机到设备的数据复制路径（如使用异步DMA传输、Pinned Memory）是降低数据通信瓶颈的重要手段。

通过合理规划和分配上述各类内存资源，可显著提升GPU应用的内存访问效率、计算性能与系统响应能力，是优化CUDA或GPU计算任务不可或缺的关键技术手段。

2. 关键优化技术

GPU内存子系统的性能优化依赖于多层次的关键技术。全局内存优化主要通过访问合并（Coalesced Access）提升带宽利用率，即将多个线程对相邻地址的访问整合为单次事务；同时通过共享内存缓存热点数据以减少全局内存调用频率。共享内存优化强调线程块内部的数据复用，避免重复加载，并通过合理划分共享内存区域，降低线程间的冲突访问。寄存器优化聚焦于控制每个线程的变量数量，减少寄存器溢出至慢速内存，并通过调节寄存器使用支持更高线程并发度。在数据传输优化方面，使用异步内存复制（如cudaMemcpyAsync()或hipMemcpyAsync()）实现数据搬移与计算的并行，结合Pinned Memory（页锁定内存）可显著提升主机与设备间的通信效率。这些技术协同作用，有效提升了GPU程序的整体性能与执行吞吐率。

3. 虚拟内存技术

虚拟内存是一种由硬件内存管理单元（Memory Management Unit，MMU）与操作系统协同实现的内存管理机制，其核心在于为每个进程提供独立的虚拟地址空间，通过页表将虚拟地址映射到实际的物理内存位置。该机制实现了地址空间抽象，使应用程序感知到的是连续的内存，而物理内存可以是非连续分布的。虚拟内存支持按需分配，仅在访问时分配内存页，从而节省物理资源。同时，它通过分页管理方式控制内存映射与迁移，并实现进程间的地址空间隔离与访问保护，提升了系统的灵活性、安全性与可扩展性。该机制在CPU（Host）和GPU（Device）架构中均有应用，但在具体实现与性能优化上存在差异。

1）虚拟内存在Host中的工作原理

在主机（Host）系统中，虚拟内存由操作系统与硬件协同管理，提供了高效、安全且抽象的内存访问机制。操作系统负责将每个进程申请的虚拟地址空间映射到实际物理内存，并通过分页机制（Page）进行精细化管理；而硬件中的内存管理单元则负责地址转换，并借助页表缓存（TLB）提升翻译效率。当程序如通过malloc()或mmap()申请内存时，操作系统分配虚拟地址，但并不会立刻对应物理内存；只有在进程真正访问该地址时，才通过页表查找进行地址转换，如无映射则触发页面错误（Page Fault），此时系统会分配物理内存并更新页表。若物理内存不足，操作系统还可将不常用内存页换出至磁盘（Swap），实现更大容量

的地址空间。虚拟内存还确保了进程之间的隔离,不同进程即使使用相同的虚拟地址,也映射到不同的物理位置。对程序开发者而言,虚拟内存屏蔽了底层物理细节,使其专注于虚拟地址操作,从而简化了开发与内存管理。与此同时,通过优化访问模式、避免频繁页面错误,可有效提升程序运行性能。

2)虚拟内存在 GPU(Device)中的工作原理

在 GPU(Device)端,虚拟内存的引入是为了解决传统显式内存管理带来的编程复杂性,特别是在通用计算任务(GPGPU)中。现代 GPU 架构中,每个线程可以使用虚拟地址访问内存,地址转换由 GPU 内部的内存管理单元(MMU)完成,将虚拟地址映射到 GPU 的物理内存。GPU 虚拟内存支持独立页表,页面大小通常为 4KB 或 64KB,并可通过主机协作管理页表内容及内存迁移。

开发者可以通过统一内存(Unified Memory)接口(如 cudaMallocManaged()或 hipMallocManaged())分配主机-设备共享的虚拟地址空间,从而省去手动的数据传输操作。当 GPU 线程访问某一虚拟地址时,如果对应的物理页尚未加载,主机会触发页面迁移机制,将相应数据页从主机或其他位置迁移至当前 GPU 的物理内存中。类似地,当主机访问尚在 GPU 的数据页时,也会执行反向迁移。

在程序开发中,使用统一内存极大地降低了主机与设备协作的复杂度,开发者可以在主机和 GPU 代码中透明访问相同内存地址。但从性能角度看,频繁的页面迁移会带来延迟,因此仍需通过合理划分计算任务与优化访问模式来减少迁移次数,提升整体计算效率。虚拟内存机制使 GPU 能够更好地支持大型数据结构、动态内存需求和多任务共享环境,是现代 GPGPU 编程的重要基础。

3) Host 和 Device 虚拟内存的对比

表 5.5 对比了主机(Host)和设备(Device)虚拟内存的关键属性,展示了两者在地址空间管理、页表机制、页大小支持、数据迁移策略以及应用场景等方面的差异。

表 5.5 Host 和 Device 虚拟内存关键属性比较

属　　性	Host 虚拟内存	Device 虚拟内存
地址空间	每个进程独立的虚拟地址空间	GPU 支持独立的虚拟地址空间
管理单元	操作系统和硬件协作,使用 MMU 和页表	主机协助 GPU 的 SMMU(或类似模块)进行管理
页大小	通常为 4KB、2MB 或更大	通常为 4KB 或 64KB
内存迁移	使用 Swap 将页面换出到磁盘	按需迁移页面至 GPU 或主机内存
应用场景	多任务程序执行、文件映射	GPGPU 计算,主机和设备协作
开发复杂性	操作系统管理,用户无须关心物理内存细节	开发者需关注数据迁移对性能的影响

4) 主机与设备协作的虚拟内存技术

主机与设备协作的虚拟内存技术通过统一地址空间和智能数据迁移机制,显著提升了异构计算系统的编程便捷性与运行灵活性。统一内存(Unified Memory)是一种由 CUDA 和 ROCm 提供的机制(如 cudaMallocManaged()或 hipMallocManaged()),使主机和 GPU 共享一个虚拟地址空间。开发者无须显式区分主机和设备内存,而是由运行时系统根据访问行为动态更新 GPU 页表,并在主机与设备间自动迁移所需页面。这大大简化了主机与设备的协作逻辑,提高开发效率,但若访问模式不优化,频繁的数据迁移可能会带来显著的

性能损耗。

相比之下，显式内存管理结合虚拟地址机制则更适用于高性能计算任务。在这种模式下，开发者依然手动控制内存的分配与数据传输，但通过虚拟地址抽象，减少了物理地址管理的复杂性。这种方式为性能敏感型应用提供了更高的控制力，能够精确调度数据流动，避免不必要的迁移开销，是在追求极致性能时的常用策略。两种机制可根据应用需求灵活选择，形成效率与控制的平衡。

5）优缺点

虚拟内存的主要优点是简化了编程，用户无须管理物理地址，同时通过资源隔离保障进程安全，并支持按需分配，提高内存利用率。但它带来性能开销，如地址翻译和页面迁移可能导致延迟，此外 GPU 虚拟内存依赖主机支持，增加了系统实现的复杂度。

4. 各级内存的分配和使用

GPU 的内存层次结构分为多个级别，每一级内存都有特定的功能和访问机制，与主机（Host）和设备（Device）的程序交互密切相关。如表 5.6 所示详细描述每一级内存与 Host/Device 程序的关系、分配与使用的阶段。

表 5.6 各级内存分配使用说明

内存级别	与 Host 的关系	与 Device 的关系	分配阶段	使用阶段
全局内存	Host 显式分配并传输数据	Device 线程直接访问	主机分配	主机/设备使用
共享内存	不可直接访问	线程块内线程共享	设备分配（内核启动时）	设备线程使用
寄存器	不可直接访问	每个线程的私有存储	编译期分配	设备线程使用
常量内存	Host 初始化常量值	所有线程只读访问	主机初始化	主机/设备使用
主机内存	主机直接分配和操作	通过传输间接使用	主机分配	主机使用/设备传输
纹理内存	主机分配和初始化	设备优化访问	主机分配	设备线程使用

每一级内存的分配和使用阶段与主机和设备程序紧密结合，设计合理的内存使用策略可以显著提升 GPGPU 程序的性能。

5. 总结

GPU 内存管理与虚拟化技术通过分层结构和统一内存机制，在提升性能的同时简化了异构计算开发。内存管理方面，借助共享内存与寄存器减少对高延迟全局内存的访问，同时支持动态分配与异步数据传输，提高执行效率。虚拟化方面，CUDA 的 Unified Memory 与 ROCm 的 HMM 提供了统一地址空间与透明数据迁移机制，支持跨 CPU/GPU 的数据共享与访问。在实际应用中，如深度学习可减少主机与设备间的数据传输，科学计算可利用共享内存提升效率，而分布式计算则受益于跨设备的高效内存管理。这些技术共同推动了 CUDA 与 ROCm 在异构计算中的高效、可扩展应用。

5.7.2 同步机制

GPU 同步机制是一种用于协调并行计算中各线程、线程块及主机与设备之间执行顺序的关键技术，旨在保障共享资源访问的安全性和计算过程的正确性。它通过控制执行时机，避免多个计算单元同时访问同一数据而导致的竞争条件、数据冲突或结果不一致。具体而

言,同步机制可用于线程间协调,确保共享内存中的数据被正确读取和写入;用于线程块间同步,管理跨块的数据依赖关系;同时用于主机与设备之间的同步,确保任务完成后再进行数据传输或下一步计算。同步还直接关联到内存一致性问题,保障数据在各并行单元之间保持统一视图,是构建可靠 GPU 并行程序的基础机制。GPU 的同步机制分为不同粒度和层次,用于协调线程、线程块以及主机与设备之间的任务和资源依赖。以下从分类、实现方式以及底层原理逐步详述这些同步机制。

1. 同步机制分类

1) 线程级同步

线程级同步适用于同一线程块(Thread Block)中的线程协作,主要用于共享内存中的数据读写同步。最常见的方法是 __syncthreads(),它在所有线程达到同步点之前阻塞后续执行,确保线程间对共享数据的访问一致。该机制延迟低,执行仅在线程块内部,底层由 SM 内部的线程块屏障控制逻辑跟踪线程状态,并在所有线程到达同步点后释放执行。

2) 线程块间同步

线程块间同步用于不同线程块之间的执行协调,通常涉及全局内存和核函数边界。常用方式包括:通过全局内存共享数据,并显式设计同步逻辑来保障数据一致性;以及使用 cudaDeviceSynchronize()实现主机线程的阻塞等待,确保所有线程块任务执行完成。底层由 GPU 控制器统一调度核函数执行状态并通知主机。

3) 主机与设备同步

主机与设备同步用于协调核函数执行和数据传输顺序,确保主机在设备任务完成后再进行后续操作。常用方式包括事件同步(如 cudaEventRecord()和 cudaEventSynchronize())用于标记和等待任务完成,以及流同步(如 cudaStreamSynchronize())用于同步特定计算流。底层依靠事件状态或流执行状态的硬件跟踪机制实现。

4) 数据传输同步

该类同步用于保障主机与设备或设备之间数据传输完成的时序正确性。同步传输使用 cudaMemcpy()等阻塞式调用,确保传输完成后才继续执行;异步传输则使用 cudaMemcpyAsync()搭配流同步,在计算与数据搬运之间实现并行,底层由 DMA 控制器与流调度器管理传输任务。

5) 高级同步机制

用于复杂依赖场景和大规模并发任务调度。

(1) 原子操作:如 atomicAdd 或 atomicCAS,由硬件提供原子性保障,适用于多个线程对同一数据的竞态访问。

(2) 分布式锁:使用全局内存中的锁变量和 atomicCAS 实现线程互斥访问,控制访问顺序。

(3) 任务图(CUDA Graphs):通过显式任务依赖关系建图,图调度器自动调度执行,免去手动同步。

(4) 双缓冲技术:计算和数据加载在两组缓冲区中交错进行,提升并行度,减少同步等待时间。

这些同步机制从低层粒度到系统级调度覆盖了 GPU 并行计算的各个关键点,是确保执行顺序、数据一致性和性能优化的基础。

2. 同步机制实现与底层原理

GPU 同步机制的底层实现依赖于硬件与指令级支持，确保线程协调、任务调度与数据一致性的高效运行。线程屏障同步通常在同一个线程块内部实现，通过共享内存和专用寄存器跟踪线程状态；Warp Scheduler 会在所有线程达到同步点后统一释放，确保同步点前后的执行顺序一致。全局同步则依赖 cudaDeviceSynchronize()等 API，主机线程阻塞在硬件层面保障所有线程块完成任务后才继续下一步操作。流与事件同步则基于流调度器与事件计时器的配合实现，硬件会跟踪流中各操作的完成状态，并通过事件标记关键时间点，实现跨核函数或数据流的精准同步。对于原子操作，GPU 硬件通过硬件原子指令和内存子系统内置的原子仲裁机制，保障对共享变量的原子读写操作，其实现依托 CUDA 的 PTX 或 ROCm 的指令集提供的原子指令。这些机制共同构成了 GPU 在高并发计算中实现高性能与一致性的同步基础，汇总说明如表 5.7 所示。

表 5.7 GPU 同步机制实现原理

同步机制	适用范围	实现方式	底层原理
线程屏障	线程块内	__syncthreads()	Warp Scheduler 监控线程状态，SM 内部屏障控制逻辑记录线程到达状态
全局内存同步	线程块间	全局内存配合 cudaDeviceSynchronize()	全局内存保证中间数据的一致性，阻塞主机等待 GPU 完成所有任务
设备同步	线程块间	cudaDeviceSynchronize()	硬件阻塞任务调度，确保所有核函数执行完成
事件同步	主机与设备	cudaEventRecord()、cudaEventSynchronize()	事件通过硬件计时器记录任务完成状态
流同步	主机与设备、流内任务	cudaStreamSynchronize()	流调度器监控流任务完成状态
数据传输同步	主机与设备	cudaMemcpy()、cudaMemcpyAsync()	内存控制器协调主机与设备间数据传输，流同步确保异步传输任务完成
原子操作	线程间	atomicAdd、atomicCAS	锁定内存地址执行原子操作，硬件保证操作的不可分割性
分布式锁	线程内或线程间	全局内存标志变量＋atomicCAS	标志变量状态由硬件控制，确保线程独占资源
双缓冲	数据流处理	双缓冲区交替读写	一组缓冲区用于计算，另一组同时加载数据，避免等待
任务图同步	复杂任务依赖	cudaGraphCreate()、cudaGraphLaunch()	图调度器根据任务依赖关系自动管理执行顺序

3. 优化策略与挑战

GPU 同步优化的核心在于减少开销、提高并行度。常用策略包括：尽量避免全局同步，用更细粒度的方式(如线程块内同步或事件)；利用异步操作实现计算与数据传输并行；减少线程竞争，优化线程任务分配；使用任务图自动管理依赖关系，减少手动同步。而面临的主要挑战包括：同步延迟高，频繁阻塞影响性能；依赖关系复杂，多流或多核函数协调困

难；以及资源竞争，高并发访问易造成瓶颈。这要求开发者在同步设计中兼顾效率与正确性。

4. 总结

GPU 底层系统的同步机制通过 线程屏障、事件、流同步、设备同步、数据传输同步等技术实现了任务的协调与正确性保障。其核心特点如下。

① 多级同步：从线程到主机设备的全栈同步。

② 异步优化：通过流和事件提高并行性。

③ 灵活高效：支持多种同步模式以适应不同任务需求。

这种机制为 GPU 在高性能并行计算场景中的正确性和效率提供了坚实基础。

5.7.3 事件和流管理

在 GPU 计算中，事件(Event)和流(Stream)是管理任务执行、协调任务之间依赖关系的两大核心机制。这两者广泛用于异步任务执行、多任务并行优化，以及任务之间的同步。以下从概念、作用、实现细节、底层原理和应用场景等方面详细说明。

1. 事件

事件是 GPU 编程中用于标记和监控设备任务状态的一种机制，主要作用是在主机和 GPU 之间实现高效的异步同步与状态查询。主机可以通过事件检查核函数或数据传输是否完成，从而决定是否启动后续操作。事件还可用于性能分析，记录任务起止时间以评估执行效率。此外，在复杂的任务链中，事件也可作为依赖管理工具，用于通知下游任务当前任务的完成情况，从而构建灵活的执行流控制逻辑。

1) 事件在运行时中的表现形式和实现原理

在运行时系统中，事件作为一种逻辑同步对象，用于标记 GPU 上任务(如核函数或数据传输)的完成状态，并由用户通过 API 进行操作与查询。从表现形式来看，事件是运行时中的一个抽象对象，用户可通过 cudaEventCreate 创建事件，通过 cudaEventRecord 记录事件，将其插入特定流中绑定到某个任务；事件对象由运行时维护，用于记录任务的完成状态；在启用计时功能时，事件还可关联时间信息，用于性能测量。事件的创建、记录与查询均由运行时统一管理。

从实现原理来看，当调用 cudaEventCreate 时，运行时系统分配事件 ID 并初始化事件元数据；调用 cudaEventRecord 则将事件插入 GPU 的流队列中，用于标记该流中此前所有任务完成的时间点；当用户调用 cudaEventQuery 时，运行时通过驱动查询事件对应任务的执行状态；若调用 cudaEventSynchronize，则会阻塞主机线程，直到事件完成为止。这种机制提供了精细的任务状态控制能力，是构建异步执行与任务依赖关系的基础工具之一。

2) 事件在驱动中的表现形式和实现原理

在 GPU 驱动层，事件并不以显式对象存在，而是作为与任务状态关联的间接管理结构体现。驱动通过追踪与事件绑定的任务完成情况，实现对事件行为的支持。其核心结构是任务状态表，记录每个已提交任务的执行状态，并为运行时的事件查询和同步提供基础。

实现原理方面，当运行时通过如 cudaEventCreate 请求创建事件时，驱动会分配内部资源(如状态表条目)用于记录该事件所关联任务的执行状态。通过 cudaEventRecord，驱动将任务插入指定流队列，并将其绑定至状态表项。当该任务由硬件执行完成后，驱动从 GPU 固件获取完成信号，更新对应任务状态。此后，运行时可通过驱动提供的查询接口(如

cudaEventQuery)读取状态表项判断事件是否完成,或通过 cudaEventSynchronize 发起阻塞调用,由驱动在确认任务完成后才唤醒主机线程。

这种事件机制的设计将高层事件抽象与底层任务调度解耦,确保了事件同步的低开销、可扩展性和硬件状态的精确反映。

3) 事件在固件中的表现形式和实现原理

在 GPU 固件层,事件以任务完成状态的硬件标志位形式存在,并通过与驱动配合间接实现事件机制。固件不直接管理"事件"对象,而是通过标志寄存器与中断机制来反映任务的完成状态,从而支撑事件的触发与反馈流程。

具体原理:当 GPU 上的某个任务(如核函数或数据传输)执行完成后,固件会更新对应的硬件寄存器标志位,将该任务标记为已完成。这一变化会触发中断信号或被驱动周期性轮询读取。驱动接收到中断后,依据固件提供的状态信息,更新任务状态表,从而使运行时能够通过事件机制感知任务已完成。

同时,固件内部还持续跟踪任务执行进度,确保任务的实际状态与外部可见状态一致。通过这种机制,固件在底层为 GPU 事件的状态管理提供了精准、实时的支撑,是整个事件系统中最靠近硬件的同步监控基础。

2. 流

流是 GPU 中用于控制任务执行顺序的机制,它本质上是一个按序执行的任务队列。所有提交到同一流中的任务(如核函数调用、数据传输)会依次执行,而不同流之间的任务可以并发运行,从而实现异步执行和任务并行。通过多个流同时调度任务,开发者不仅可以提升 GPU 的资源利用率,还能结合事件机制管理不同任务之间的依赖关系,实现高效、可控的并行计算流程。流机制是构建异步、并行 GPU 应用程序的基础工具之一。

1) 流在运行时中的表现形式和实现原理

在运行时系统中,流表现为一个逻辑任务队列,用于管理和控制设备端任务(如核函数或数据传输)的执行顺序。用户通过如 cudaStreamCreate、cudaStreamSynchronize 等 API 创建和操作流,对任务进行分配与同步。每个流由运行时维护为一个对象,包含任务队列、优先级、状态等信息,并统一登记在运行时的流表(Stream Table)中进行调度管理。

实现机制包括:调用 cudaStreamCreate 时,运行时为新流分配队列资源并在流表中注册;当用户向流中提交任务(如核函数或 cudaMemcpyAsync)时,运行时会将这些任务插入对应的流队列中;调用 cudaStreamSynchronize 则会阻塞主机线程,直到该流中的所有任务完成。整个过程中,运行时根据流的状态与队列信息协调任务执行与同步,是 GPU 异步并行机制的核心调度基础。

2) 流在驱动中的表现形式和实现原理

在 GPU 驱动层,流为硬件任务队列的管理单元,负责协调任务的提交、调度和执行。每个流在驱动中会被映射并调度到硬件任务队列,任务按顺序排入等待执行。同时,驱动维护一个流表(Stream Table),用于记录每个流的任务队列指针、优先级、状态等元数据。

实现机制包括:在运行时创建流时,驱动初始化对应的硬件任务队列,并在流表中登记;当任务(如核函数或异步数据传输)被提交时,驱动将其转换为低级硬件指令并加入流对应的 FIFO 队列;多个流之间的任务调度由驱动根据流优先级和当前硬件负载动态决策;当运行时请求同步(如 cudaStreamQuery 或 cudaStreamSynchronize)时,驱动提供查询

接口,以确认流内所有任务是否执行完成。整个机制实现了对多任务的有序调度和资源优化,是 GPU 多流并行执行的底层基础。

3) 流在固件中的表现形式和实现原理

在 GPU 固件层,流(Stream)以硬件任务队列的形式存在,任务队列用于存储待执行的底层硬件任务(如启动核函数、触发 DMA 传输等)。固件中的流调度器(Stream Scheduler)负责从多个流队列中提取任务,并将其调度到适当的硬件资源上执行,如计算单元(SM/CU)、内存控制器等。

实现机制包括:当驱动将任务提交到流后,固件从对应的流队列中提取指令并解析,随后将任务分发到空闲的硬件资源上执行。任务执行完毕后,固件会更新相关的任务状态寄存器,以便后续的状态查询或同步。同时,固件的流调度器会持续监控各流的优先级与硬件资源负载情况,动态在多个流之间进行调度,以最大化硬件资源利用率并保障高优先级任务的响应时效。这一机制实现了从指令级到硬件层的并发执行控制,是 GPU 异步流系统的核心支撑。

3. 事件与流的结合

在 GPU 异步执行体系中,事件与流的结合是实现多任务并行与依赖控制的关键机制。事件可作为流中某一任务完成的标记,允许其他流通过等待该事件来实现同步,从而管理跨流之间的执行顺序。例如,流 A 中的核函数执行完成后记录一个事件,流 B 可以通过等待该事件来确保在其任务执行前,流 A 的任务已完成。此外,在流内部,事件也常用于标记关键的同步点,使后续任务仅在前序任务完成后才启动。

在实际应用中,开发者通常通过创建多个流以提交相互独立或部分关联的任务,最大化并行执行;然后使用事件在流之间建立明确的依赖路径,保证任务以预期顺序进行。通过这种事件与流的协作机制,GPU 能够在保证数据一致性的同时,充分发挥硬件并发执行能力。

4. 总结

事件和流是 GPU 并行计算中的两种核心机制,协同构建起完整的任务调度与同步体系。事件是用于标记任务完成状态的抽象对象,由运行时管理,在驱动中通过任务状态表间接实现,在固件中则依赖硬件标志寄存器和中断机制完成状态反馈。流则是任务的调度容器,运行时将其管理为逻辑任务队列,驱动将其映射为任务管理单元,固件则将其对应为实际的硬件指令队列并执行。三层架构中,运行时提供接口和高层控制,驱动协调资源和调度策略,固件直接执行任务并返回状态。通过事件与流的灵活组合,开发者可以精准控制任务顺序、实现多任务并行、优化设备利用率,从而大幅提升 GPU 应用的性能与响应能力。


视频讲解

5.7.4 核函数装载启动

在 GPU 计算中,核函数启动的过程贯穿了从运行时到驱动程序再到固件和硬件的多个层级,每一层都有其具体的职责和接口。以下对核函数启动过程中的各层工作、接口及具体参数进行详细说明。

1. 核函数启动整体流程

GPU 核函数启动是一个从用户代码到底层硬件协同执行的完整流程,分为多个层次协作完成;在应用层,用户通过如 kernel <<< grid, block, sharedMem, stream >>> 语法提交核

函数，定义线程网格、线程块以及共享内存等配置；进入运行时层后，系统会解析调用参数、校验配置，并将任务封装为驱动接口调用；驱动层随后将该请求转换为底层硬件指令，进行资源可用性评估（如寄存器、共享内存需求是否满足），并将任务提交给固件执行；固件层解析指令、初始化线程网格，调度任务分发到具体硬件单元；最终在硬件层，GPU 执行核函数，通过线程调度器运行各线程块，并在任务完成后更新状态，逐层向上反馈。该流程实现了从抽象接口到并行执行的高效映射，是 GPU 编程模型中的核心机制。

2. 各层职责

1）应用层

在 GPU 编程中，应用层是用户与 GPU 系统交互的起点，其主要职责是通过高层接口配置和发起核函数执行请求。用户需明确指定线程网格（gridDim）与线程块（blockDim）的维度来组织并行计算，同时设定动态共享内存大小（sharedMem）和所使用的流（stream）以支持异步执行。典型接口如 CUDA 的 kernel <<< grid，block，sharedMem，stream >>> (args)或 HIP 的 hipLaunchKernelGGL()用于启动核函数，而 cudaMalloc/hipMalloc 和 cudaMemcpy/hipMemcpy 负责内存分配与数据传输。应用层还负责将核函数的输入参数(args)传入执行环境。这一层的核心任务是定义并提交计算任务的执行结构与资源需求，为运行时系统提供完整的调度与执行配置。

2）运行时层

在 GPU 编程架构中，运行时层作为连接用户接口和驱动层的桥梁，承担设备抽象、参数校验与任务封装的关键职责。当用户通过高层接口发起核函数调用时，运行时首先对参数进行合法性检查（如线程块是否超出硬件上限），然后将这些信息封装为标准格式，转交给驱动接口。运行时还维护任务队列与流管理，确保异步任务正确插入执行顺序。

核心接口包括 cudaLaunchKernel()（CUDA）或 hipModuleLaunchKernel()（HIP），这些接口接收经过编译的核函数指针、线程网格/线程块配置、共享内存大小、流句柄及核函数参数数组。在此过程中，运行时层将用户级配置解析为底层可识别的调用结构，并将其提交给驱动进行后续资源评估与执行调度。它是 GPU 核函数执行流程中实现平台抽象和统一调度的重要中间层。

3）驱动层

在 GPU 执行架构中，驱动层负责将运行时提交的核函数启动请求转换为具体的硬件执行命令，是连接高层 API 和底层硬件之间的关键调度枢纽。其主要职责包括解析运行时封装的调用信息，验证硬件资源可用性（如共享内存、寄存器数是否足够），并将任务构建为底层设备指令。

典型接口如 CUDA 的 cuLaunchKernel()，接收核函数句柄、线程网格和线程块配置、动态共享内存大小、任务所属流以及核函数参数数组。在核函数启动前，驱动会评估资源占用，并据此判断是否可调度。如果条件满足，驱动构建对应的设备任务描述，加入流管理队列，并将其提交至固件层执行。同时，驱动还负责提供任务执行状态查询接口，供运行时检测任务进度或完成状态。驱动层是核函数从软件逻辑向硬件控制转换的核心接口层。

4）固件层

在 GPU 核函数执行流程中，固件层作为驱动与硬件之间的直接控制组件，负责接收驱动提交的设备指令并协调实际的硬件资源调度与执行。其核心职责包括解析任务配置、初

始化线程网格与线程块结构、分配所需资源(如寄存器、共享内存、计算单元),并将核函数任务调度到 GPU 的执行引擎(如 SM)上运行。

固件通过任务接收接口接收驱动传来的核函数启动信息,包括核函数入口地址、网格与块维度、共享内存配置、参数地址等。它会在内部构建任务执行上下文,确保每个线程块能正确启动,并在执行过程中监控任务状态,如是否完成或中断。任务完成后,固件通过状态反馈接口通知驱动更新状态表。固件层的调度效率和资源分配能力直接影响整个 GPU 执行流程的并行性和稳定性,是 GPU 硬件执行控制的核心环节。

5) 硬件层

在 GPU 的核函数执行链条中,硬件层是最终的计算执行单元,直接负责将核函数的指令转换为实际的并行操作。该层不暴露显式接口,而是根据固件提交的任务描述信息和命令队列内容自动完成初始化与调度。其主要任务包括:分配每个线程块所需的寄存器和共享内存,根据线程网格配置启动计算单元(如 SM),并通过线程调度器安排 Warp 执行核函数代码。

每个线程块在启动时,会绑定到具体的执行引擎,硬件根据已准备好的参数缓冲区和任务描述信息访问核函数参数,并启动设备端指令执行。执行过程中,线程间通过共享内存协作,执行完的中间或最终结果被写回全局内存。任务完成后,硬件更新相关的任务状态标志位,供固件上报驱动。硬件层的并发度、调度策略和执行效率直接决定了 GPU 计算性能,是整个核函数执行流程的物理基础。

3. 核函数启动参数

运行时在启动核函数时,会将一组结构化参数传递给固件,确保核函数能在 GPU 上正确调度和执行。这些参数包括核函数的入口地址和参数地址,用于定位设备代码和输入数据;线程网格与线程块维度,决定并行执行结构;共享内存大小、全局内存和常量内存信息,用于配置内存资源;流 ID 和任务优先级,用于控制任务调度顺序;以及寄存器使用量和共享内存分区策略,用于分配硬件资源。固件接收这些参数后,完成资源初始化、任务入队和线程调度,实现核函数的并发执行。这一机制是 GPU 运行时与底层执行引擎协同工作的关键,表 5.8(仅示意)详细说明运行时传递的参数类型及其用途。

表 5.8 核函数启动参数说明

参数类别	参数名称	描述	数据类型
核函数基础信息	核函数入口地址	核函数的设备端起始地址,用于启动核函数执行	64 位地址
	核函数参数地址	核函数参数在设备全局内存中的地址	64 位地址
线程配置	网格维度(gridDim)	线程网格的 X、Y、Z 维度	3 个 32 位整型
	块维度(blockDim)	线程块的 X、Y、Z 维度	3 个 32 位整型
内存配置	共享内存大小	每个线程块动态分配的共享内存大小(字节)	32 位整型
	全局内存地址	全局内存中核函数所需数据的起始地址	64 位地址
	常量内存地址	核函数使用的常量内存的地址	64 位地址
调度信息	流 ID	核函数所属流的标识符	32 位整型
	任务优先级	所属流的调度优先级	32 位整型
硬件配置	寄存器数量	每个线程所需的寄存器数量	32 位整型
	L1/共享内存分区策略	指定共享内存与 L1 缓存的分配比例	枚举值

这些参数递给固件后，固件利用这些信息完成硬件资源配置，调度任务到 GPU 的计算单元并启动核函数执行。

4. 总结

核函数的启动过程是一次从高层抽象到底层执行的分层协同流程，涉及运行时、驱动、固件与硬件多个环节。运行时层负责接收用户调用，配置线程网格、内存与流；驱动层解析这些配置，进行资源检查与指令生成；固件层接收指令，分配计算资源并初始化执行环境；硬件层最终并行调度线程并执行设备端代码。整个过程通过参数逐层下传与指令精化，实现核函数从逻辑描述到物理执行的完整映射，是 GPU 编程模型中至关重要的执行链条。


视频讲解

5.7.5 异构可执行文件

异构可执行文件是一种将主机(Host)代码和设备(Device)代码统一封装的二进制格式，用于支持异构计算平台中的协同执行。它的核心作用是将控制逻辑(CPU 执行)与并行计算逻辑(GPU 执行)集中管理，便于在运行时根据硬件平台加载和调度对应的代码。这种统一封装方式提升了程序的跨平台适配能力和执行效率。

1. 文件的核心组件

1) 主机代码

主机代码运行在 CPU 上，负责程序的整体控制和设备调度。其主要功能包括设备初始化、内存分配与数据传输、核函数配置(如线程网格和线程块大小)、核函数的启动和任务流控制。主机代码是标准的可执行二进制，通常为 ELF(Linux)或 PE(Windows)格式，是异构应用的入口逻辑。

2) 设备代码

设备代码运行在 GPU 或其他加速器上，负责执行并行计算任务。这些代码通常以中间表示形式(如 NVIDIA 的 PTX、AMD 的 LLVM IR、OpenCL 的 SPIR-V)或目标设备的汇编/机器码(如 SASS)形式存在。设备代码可能直接嵌入主程序中，也可以以专用二进制镜像(如 CUDA 的 fatbin)附加在可执行文件中。

3) 元数据

元数据定义了主机和设备端之间的接口，确保两者能够正确通信和调度。它包括：

- 核函数的符号名与入口地址；
- 参数布局、寄存器需求、共享内存大小等资源配置；
- 支持的设备架构(如 SM_75、GFX908)；
- 驱动依赖与运行环境要求；
- 函数签名，用于主设备之间参数对接与调用匹配。

4) 数据段

数据段包含核函数执行所需的静态数据，包括常量表、初始的全局变量值、默认配置值等。GPU 执行过程中会从这些区域读取只读常量或预设参数，通常位于 Constant Memory 或 Global Memory 区域，是设备运行期必要的支持内容。

5) 链接信息

链接信息描述主机代码与设备代码之间的逻辑映射与依赖，类似传统可执行文件中的符号解析和重定位信息。在 CUDA 和 ROCm 中，这部分内容支持静态与动态链接形式，例

如将多个 PTX 或 GFX 中间代码打包为 fatbinary,或将 GPU 核函数链接到动态库中以支持运行时加载和设备端版本切换。

通过这种结构组织,异构可执行文件实现了跨架构部署、任务分离调度和统一编程接口支持,是现代异构计算系统高效运行的基础。

2. 文件的组织形式

1) 内嵌组织

内嵌组织将主机代码、设备代码和相关数据全部打包到一个可执行文件中(如 .exe 或 .elf)。主机代码存储在常规代码段(如 .text),设备代码则嵌入专用段(如.nv_fatbin),运行时由主机加载到设备执行。主机与设备的数据段(如.data、.bss、.nv.constant 等)也嵌入同一文件,元数据用于标识核函数和设备信息。这种方式部署简单、文件独立,适合平台固定的一体化部署,但更新不灵活,修改设备代码需整体重新编译,结构示例如图 5.11 所示。

图 5.11 内嵌代码组织形式

2) 外部附加组织

外部附加组织将主机与设备代码分别存储,主机代码为可执行文件(如.exe 或.so),设备代码独立,为.cubin、.hsaco、.bin 等文件,由主机运行时动态加载。设备的数据段包含在这些独立文件中,主机通过 API 动态加载设备模块(如 cuModuleLoad),并根据需要通过 cuMemcpyHtoD/cuMemcpyHtoDAsync 等接口传输输入数据。这种方式便于更新和灵活部署,支持模块化开发,适合设备平台多样或迭代频繁的应用,但需额外管理设备文件和加载逻辑。

3) 多平台支持

多平台支持组织在同一可执行文件中打包多个平台的设备代码(如多个 PTX/SASS/GFX/SPIR-V 版本),运行时根据实际设备自动选择最优版本执行。各平台的数据段(常量、变量)也独立维护,元数据用于架构识别与调度。这种方式适合跨架构部署、统一分发,具有良好的兼容性与可移植性,但构建复杂,文件体积较大。

3. 总结

异构可执行文件通过统一封装主机与设备端代码,实现了跨平台、多架构下的高效协同执行。其常见组织方式包括内嵌(如 CUDA 的 fatbin)、外部附加(如动态加载的设备模块)以及多架构兼容(同时支持多种 GPU 架构)。不同平台如 CUDA、ROCm、OpenCL 虽在实现细节上有所差异,但都围绕提升异构计算性能、简化开发流程和增强硬件适配能力为目标。整体而言,异构可执行文件是现代异构系统中任务调度与资源协调的关键载体。

5.7.6 异构执行模型

异构执行是一种在多种硬件架构(如 CPU、GPU、FPGA、TPU 等)间协同完成计算任务的方式,旨在充分发挥各类硬件的性能优势。与传统仅依赖单一处理器的模式不同,异构执行通过将任务合理划分并调度至最适合的硬件上运行,提升整体系统的计算效率、资源利用率和能效。它广泛应用于科学计算、深度学习和高性能计算等对性能要求极高的领域,随着 GPU 和专用加速器的兴起,异构执行已成为现代计算架构的重要发展趋势。

1. 异构执行的核心架构

异构执行的核心架构由多个关键组件协同构成，确保不同类型的硬件设备能够高效协作执行任务。通常包括以下 4 部分。

(1) 主控设备(Host)：一般是 CPU，负责任务分配、设备初始化、内存管理、核函数调度等。运行高级控制逻辑和串行任务。

(2) 加速设备(Device)：GPU、FPGA、TPU 等加速器，负责执行高并行的计算密集型任务。提供线程并行、向量化计算等能力。

(3) 统一程序模型：通过统一的程序模型(如 CUDA、HIP、OpenCL 等)，编写跨设备的任务代码。将 Host 和 Device 代码整合在一个可执行文件中。

(4) 运行时系统：如 CUDA Runtime、ROCm Runtime，提供任务调度、内存管理、设备间通信等功能。

2. 典型模型

在典型的异构计算中，Host 端(通常为 CPU)负责任务的整体调度、数据管理和核函数的调用，而 Device 端(通常为 GPU)负责高并行的计算任务。图 5.12 描绘了 CPU 和 GPU 协同工作的整体执行流程。

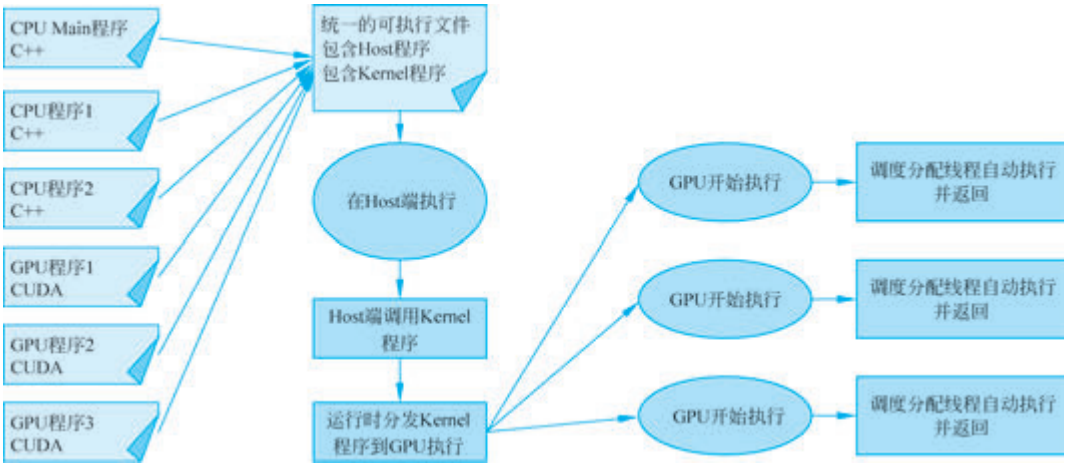

图 5.12 典型异构执行模型

该模型展示了异构执行的核心流程，通过统一可执行文件将 CPU 和 GPU 代码整合，并通过 Host 程序调用和调度 GPU 核函数执行任务。GPU 利用其高并行性和自动线程调度能力，显著提高了计算密集型任务的效率。该模型适用于需要大量并行计算的场景，如深度学习、科学计算和大规模数据处理。

该异构执行模型由 CPU 主程序和多个 GPU 核函数组成，并通过统一的可执行文件进行整合。主程序通常采用 C++编写，负责初始化设备、管理数据、调用核函数及收集结果；同时还可能包含若干辅助 CPU 模块，用于处理不适合 GPU 并行的任务。GPU 部分由多个核函数(如 GPU 程序 1、GPU 程序 2、GPU 程序 3)构成，采用 CUDA 等并行编程框架开发，专门用于执行高吞吐量的计算密集型任务。这些 CPU 和 GPU 程序通过静态链接或动态加载整合在一个统一的可执行文件中，由 Host 程序统一调度。

在运行机制上，主程序控制整个执行流程，根据需要调用不同的核函数并传递参数。每

个核函数在启动后被划分为线程块和线程,GPU 硬件会自动将任务分配到多个计算核心并并行执行,由 GPU 的线程块调度逻辑和 Warp Scheduler 等硬件单元协同管理执行。该模型支持多个核函数并行运行,有效提升了资源利用率和执行效率,特别适用于深度学习、图像处理、科学仿真等需要大规模并行计算的应用场景。

3. 典型工作流程

以下是对异构执行典型流程的描述,涵盖主控设备任务、加速设备任务、数据管理与任务同步 4 方面。

1) 主控设备(Host)任务

主控设备(通常为 CPU)负责整个异构执行流程的组织与调度。首先,它会初始化运行时库(如 CUDA 的 cuInit()或 ROCm 的 hsa_init()),并完成对 GPU 等加速设备的初始化与执行环境准备,如运行时上下文和任务队列的建立。随后,主控设备将复杂计算任务拆解为多个子任务,并调用设备管理接口指定执行目标。在核函数调度阶段,Host 会根据任务特性配置线程网格(Grid)和线程块(Block),并通过任务队列向 GPU 发出核函数启动指令。

2) 加速设备(Device)任务

加速设备(如 GPU)在接收到 Host 的调度命令后,加载对应核函数入口,解析任务参数与线程配置,并将任务分配到其内部的计算资源(如 SM 和 Warp)。每个线程块(Thread Block)被分派到某个 SM(Streaming Multiprocessor)上,而线程则被划分为 Warp,由硬件调度器进行高效调度与执行。所有线程块并行运行,完成计算后结果存储到全局内存,或通过统一内存模型传回 Host。

3) 数据管理机制

异构执行涉及频繁的数据交换,因此内存与数据管理至关重要。Host 通过接口如 cudaMalloc()或 hipMalloc()为加速设备分配内存,并通过 cudaMemcpy()或 hipMemcpy()在 Host 和 Device 间实现数据传输。为简化开发、提高效率,也可使用统一内存(Unified Memory)机制,使得 CPU 和 GPU 可共享访问内存,自动完成数据迁移。

4) 任务同步与依赖管理

为保证任务按预期顺序执行并协调 CPU 与 GPU 的协作,系统使用多种同步机制。通过 cudaStreamCreate()或 hipStreamCreate()创建多个异步任务流(Stream),可实现任务的并行与重叠执行。同时,借助事件机制(如 cudaEventRecord())在 Host 与 Device 间标记和同步任务完成状态。显式的任务依赖管理则用于定义操作顺序,确保数据一致性与流程正确性,是高性能异构执行的关键保障。

4. 优缺点

异构执行具有显著的性能优势,依靠 GPU 的大规模并行能力和 CPU 的控制能力协同工作,大幅提升计算效率与系统吞吐量,特别适用于大数据和高并行场景。同时,它具备良好的灵活性与扩展性,可支持多种硬件(如 CPU、GPU、FPGA)协同运行,并轻松扩展至多设备环境。然而,该模型存在一些局限,如开发难度较高,要求开发者熟悉硬件架构、内存管理与调度机制;此外,Host 与 Device 之间的数据传输可能带来额外开销,影响整体性能;不同硬件平台间的差异也可能导致软件移植复杂度增加。

5. 总结

异构执行是一种高效的计算模式,通过 CPU 和 GPU 的协同工作,充分利用了异构硬

件的特点。它在计算密集型任务中表现出极高的性能优势，但开发时需要处理硬件特性和数据传输等复杂问题。通过成熟的编程框架（如 CUDA 和 ROCm），可以显著降低开发成本，使异构执行成为高性能计算和深度学习等领域的主流方案。

5.8 GPU、NPU、TPU 对比分析

GPU、NPU 和 TPU 是 3 类主流的 AI 加速器，分别代表了通用计算、专用推理与定制化训练的不同方向。GPU 起源于图形处理，采用 SIMD/SIMT 并行结构，具备极强的通用性和灵活的编程能力，适用于深度学习训练、科学计算等广泛场景；NPU 专为神经网络推理与低精度计算优化，具有高能效和高吞吐，适用于边缘设备和轻量级部署，但底层控制能力相对有限；TPU 则由 Google 自研，最初围绕 TensorFlow 生态设计，采用 Systolic Array 架构，适合云端大规模训练和推理，性能极高但依赖封闭生态和固定平台。三者在算力结构、能效比、编程模型和部署环境方面各有优势，需根据具体任务需求选择合适方案。表 5.9 所示是 GPU、NPU、TPU 三类加速器的详细对比分析，适合系统性理解与技术选型。

表 5.9 GPU、NPU 和 TPU 多维度对比分析

对比维度	GPU	NPU	TPU
主要用途	图形渲染、通用并行计算（GPGPU）	AI 专用推理加速（部分产品支持训练）	Google 自研 AI 推理/训练
架构特点	多核 SIMD/Warp，通用可编程	专用张量计算单元＋AI 优化指令＋DMA	大规模 Systolic Array＋高速内存
并行方式	线程块＋Warp 并行	张量/向量并行、MAC 并行	大规模 Systolic 并行
编程模型	CUDA、OpenCL、DirectCompute	驱动＋API（如 NNAPI、AscendCL、SNPE）	XLA 编译器＋TPU Runtime
运算类型	支持通用计算（float，int，double）	高效支持 INT8、FP16、BF16、少量支持 FP32	主要支持 BF16、INT8、有限支持 FP32
可编程性	高（图形＋通用计算）	中（API 封装良好，受限于硬件指令）	低（依赖 XLA 编译模型）
通用性	通用性强，支持多种并行任务	针对 AI 优化，不适合图形/通用计算	专注于深度学习推理/训练，通用性较弱
架构代表	NVIDIA Ampere/Hopper、AMD CDNA	华为昇腾、寒武纪、Imagination、Intel NPU	Google TPU v2/v3/v4
软件生态	CUDA、cuDNN、OpenCL、TensorRT	MindSpore、TensorRT、NNAdapter、SNPE 等	TensorFlow/XLA，TPU Runtime
部署环境	数据中心、PC、移动端、嵌入式	手机、边缘设备、云端 AI 加速卡	Google Cloud（TPU Pods）
训练/推理适配	训练＋推理均可	以推理为主，也支持训练（如昇腾）	v2/v3 兼顾训练和推理，v4 更偏向训练

5.9 未来发展趋势

1. GPU 运行时的发展趋势

GPU 运行时系统作为连接高层应用与底层硬件的核心桥梁,正向更高层次抽象、更强智能调度与更广泛异构支持持续演进。在抽象层面,未来运行时将提供跨语言接口(如 Python、C++、Rust)和统一 API,简化硬件调用细节,并与深度学习框架(如 TensorFlow、PyTorch)深度集成,让开发者专注于算法实现。在分布式支持方面,运行时将强化多 GPU 和跨节点任务调度,集成高效通信机制(如 NCCL、RCCL),满足大模型训练与 HPC 场景需求。

在调度机制上,运行时将普遍采用基于任务图(Task Graph)的执行模型,自动进行依赖分析、算子融合和调度重排,提升并发性、降低同步开销(如 CUDA Graphs 所体现)。面向异构平台,运行时将统一支持 GPU、CPU、NPU、DPU 等加速器,实现跨架构一致性编程和资源协同。未来还将集成 AI 驱动的自动化调优系统,智能识别性能瓶颈、自动配置资源和优化核函数执行策略,降低开发调试成本。整体而言,GPU 运行时系统的演进将成为异构智能计算系统的调度核心和优化引擎,支撑大模型训练、科学仿真和边缘 AI 的规模化应用。

2. GPU 驱动的发展趋势

GPU 驱动正在从传统的硬件控制接口演变为智能化、跨平台、可扩展的系统核心。未来驱动将通过模块化设计,将内存管理、任务调度、通信与安全等功能拆分为独立子模块,提升维护性与适配能力,特别适用于多样化的硬件平台(如嵌入式系统和数据中心)。在架构支持方面,驱动将统一适配 x86、ARM、RISC-V 等平台,实现跨系统部署。智能化调度将成为关键,驱动将引入 AI 分析机制,根据负载特征动态调整资源分配与任务调度,提升多任务执行效率。同时,驱动将增强安全能力,引入虚拟化隔离、权限管理和零信任模型,应对敏感应用对安全性的要求。借助 PCIe 5/6、NVLink、CXL 等高速互连协议,GPU 驱动还将优化底层通信路径,实现更低延迟和更高带宽,支撑大模型训练与数据密集型任务。整体来看,GPU 驱动将演进为面向异构智能计算的关键控制中枢。

3. GPU 固件的发展趋势

GPU 固件作为驱动与硬件之间的核心控制层,未来将朝着更高抽象、更智能调度和更强异构协同方向演进。首先,在硬件抽象方面,固件将不仅执行底层命令,还将内建高层操作符(如矩阵乘法、卷积)的支持,从而减少驱动负担、降低延迟,并加速深度学习模型推理。其次,固件将实现对多种硬件模块(如 Tensor Core、视频编解码引擎、DPU)的统一调度,提升资源弹性和异构任务适应能力。接着,在功耗优化方面,固件将深度集成 DVFS、电源门控等技术,结合温度和负载感知实现智能动态调节,满足数据中心与边缘场景的低功耗需求。然后,在异构协同层面,固件将具备与 NPU、TPU、DPU 等设备通信与协同调度的能力,成为多芯融合架构中的协同控制中心。最后,AI 技术将被引入固件中,用于实时分析任务负载、能耗和缓存行为,并据此动态优化调度策略与资源配置。总体而言,GPU 固件正从执行接口转变为具备智能感知、策略决策和异构协调能力的关键调度中枢。

4. GPU 软件栈整体趋势

GPU 软件栈正朝着统一化、智能化、异构协同和安全可扩展的方向演进。未来，运行时、驱动与固件将通过更紧密的集成和标准化接口（如 CUDA、ROCm、OpenCL），构建统一的软件生态体系，提升开发效率与平台兼容性。同时，随着异构计算的普及，GPU 软件栈将全面支持 GPU、CPU、NPU 等多种计算单元的协同任务调度与数据共享。在自动化方面，AI 将被用于运行时性能分析、任务调度优化及安全策略生成，实现动态调优与自适应运行。此外，软件栈将采用模块化设计，便于按需扩展至多样化场景（如云端、边缘端），并通过沙箱化、虚拟化、权限隔离等机制增强整体系统的安全性，满足下一代智能计算平台对高性能与高可靠的双重需求。

5. 结论

GPU 的底层软件系统将朝着更高效、更智能、更统一的方向发展，以满足深度学习、高性能计算、科学仿真和边缘计算等不断增长的需求。未来的 GPU 软件栈将进一步解放开发者，使其专注于创新，而非底层硬件细节。