（4）工程文件输出与离线交叉编译集成

最终工程输出为可直接编译的C源码与头文件集合,并包含面向RTEMS的构建脚本与必要配置，使得静态内存表、静态调度表与执行骨架在离线阶段完成编译与链接，形成无需文件系统与动态装载支持的可部署镜像。

3. 面向嵌入式操作系统的轻量化异构推理框架设计

3.1 方案概述

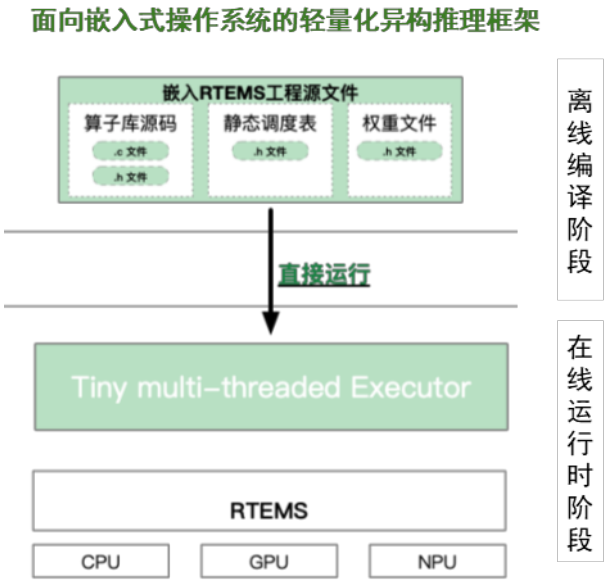


图 3-42 面向嵌入式操作系统的轻量化异构推理框架设计

面向嵌入式操作系统的人工智能推理任务，系统通常运行在具有严格时序约束和受限计算与存储资源的环境中，对推理执行过程的确定性、资源占用的可预测性以及软件栈整体可靠性提出了更高要求。在此类环境下，推理框架的执行路径与时序行为需要尽可能减少对运行期动态决策的依赖，以保证系统行为的可分析性与可控性。然而，面向通用操作系统设计的推理框架通常依赖动态调度、运行期资源管理以及复杂的系统服务支持，其执行控制在较大程度上受操作系统调度与资源管理机制影响，如图 3-42 所示，执行过程需经过多层操作系统服务与动态调度路径，难以在嵌入式操作系统环境中满足对确定性执行和资源上界可控性的要求。因此，有必

要面向嵌入式操作系统的运行特性，构建一种运行机制更加简洁、执行行为可验证、适用于实时场景的轻量化推理运行框架。

本项目拟设计并实现一种面向嵌入式操作系统的轻量化异构推理运行时框架。该框架的核心思路是将推理执行过程中涉及的调度决策、计算依赖解析以及内存与参数布局等原本依赖运行期动态完成的过程，前移至编译阶段统一完成，并固化为静态执行计划，使运行时本身成为编译产物的一部分。生成的执行框架采用固定的执行顺序、静态内存布局以及预定义的控制逻辑，在模型加载后即可直接运行，在仅依赖最小操作系统支持的前提下，无需依赖线程调度、文件系统或动态内存管理等复杂系统服务。通过上述设计，运行时在推理阶段不再承担复杂的决策与资源管理功能，其执行行为完全由编译期生成的静态执行计划所决定，从而有效降低运行时开销，并消除由操作系统调度和动态资源分配机制引入的不确定性。该运行模式能够为嵌入式操作系统环境下的实时推理任务提供可分析、可验证的执行时序特性以及明确的资源使用上界，满足如 RTEMS 等实时操作系统对高可靠性与确定性执行的要求。在具体实现过程中，本项目将参考成熟深度学习编译体系在算子表示、计算图优化以及代码生成等方面的设计思想，并结合嵌入式实时系统的运行约束，对编译期与运行期的功能边界进行重新划分，从而构建适用于实时场景的静态化推理执行框架。

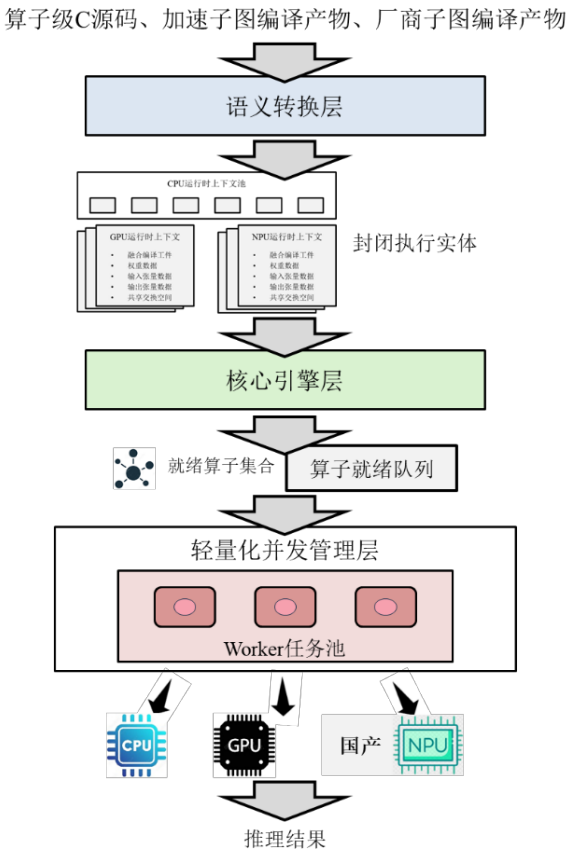


图 3-43　面向嵌入式操作系统的轻量化异构推理框架的三层架构

如图所示，面向嵌入式操作系统的轻量化异构推理框架采用三层架构设计，自上而下依次完成从模型语义收敛、到推理调度组织、再到并发执行控制的分层抽象，通过明确各层的能力边界与职责分工，实现对异构计算资源的统一组织与确定性推理执行：

（1）语义转换层：模型语义到可调度实体的收敛层。语义转换层负责对 TVM 编译阶段输出的模型产物进行解析与重组，将其转化为与目标运行环境强绑定的封闭式可调度数据结构。该层依据编译阶段注入的异构属性标签（面向 CPU、GPU、NPU 等计算资源），明确算子的执行归属，并完成算子内核、张量元数据以及参数/常量的统一封装，从而实现模型语义向运行时可执行语义的一次性映射。

（2）调度引擎层：推理执行时序的组织层。调度引擎层依据编译阶段生成的静态调度表与算子间依赖关系描述，对推理过程进行逻辑层面的组

织与推进。该层负责解析算子的先后关系与并行约束，判定算子的就绪状态，并在顺序或并行执行路径之间做出受编译期约束的调度决策，从而保证推理过程的逻辑正确性与整体执行效率。调度引擎层仅处理算子级别的执行时序与依赖关系，不直接参与算子内部计算，也不感知具体计算资源的底层实现细节。

（3）轻量化并发管理层：受控并行执行与资源协调层。轻量化并发管理层面向多核与异构计算环境，负责将核心引擎层给出的调度结果映射为受控、可预测的并发执行行为。该层通过同步屏障、事件通知及受控任务分派等机制，实现跨计算资源的任务协调与执行控制，在保证算子级时序一致性与系统确定性的前提下，充分挖掘多核与异构平台的并行计算能力。该层仅承担并发管理与执行协调职责，具体算子的执行由对应计算平台的驱动或运行接口完成，其实现方式可根据目标嵌入式操作系统与硬件平台进行工程化适配。

通过上述三层结构的协同，该框架在不引入复杂运行期机制的前提下，实现了异构推理任务在嵌入式操作系统环境中的统一组织与确定性执行，为后续异构执行适配与系统优化提供了清晰且可扩展的架构基础。

3.2 语义转换层

3.2.1 基本思路

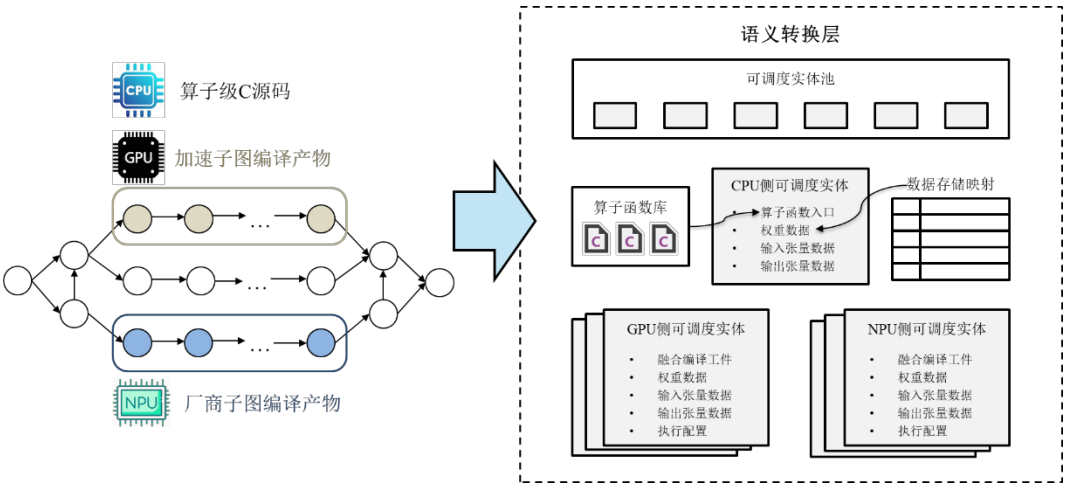


图 3-44 语义转换层的基本思路

语义转换层的输入来源于编译阶段生成的异构模型产物。针对嵌入式操作系统环境下的异构计算平台，模型在编译阶段通常已依据目标硬件特性完成子图划分与算子映射，不同计算资源对应的编译产物在组织形式、语义侧重点以及可用信息维度上均存在显著差异。因此，在语义转换层中，有必要对来自不同编译路径的模型产物建立统一的输入建模视角，以明确其共性与差异，为后续语义收敛与实体组装提供基础。

如图 3-44 所示，面向 CPU 与 GPU 等通用计算资源，编译阶段通常生成以计算图或融合子图为核心的加速编译产物，该类产物保留了较为完整的算子级描述信息，包括算子类型、输入输出张量关系、参数配置以及一定程度的融合与优化结果，其表示形式更偏向通用计算语义，适合在运行期进一步解析与组织，但整体仍以编译语义为中心，其结构与接口并未直接面向运行期调度与执行。相比之下，面向国产 NPU 等专用加速器的编译产物通常由厂商工具链生成，其表示形式更贴近底层硬件执行模型，往往以融合编译工件、二进制执行单元或专用描述文件的形式存在，内部已隐含具体的执行流程与资源使用方式，对外暴露的算子级结构信息相对有限，但在执行效率和硬件适配性方面具有明显优势。由于不同厂商工具链在表示方式与接口设计上的封闭性和差异性，上述两类编译产物在语义结构与组织形式上难以直接统一。综上所述，语义转换层所面对的输入模型呈现出显著的异构特征：一方面，不同计算资源对应的编译产物在表示层级与语义粒度上存在差异；另一方面，这些编译产物均携带推理执行所需的关键信息，只是其组织方式和显式程度不同。基于此，语义转换层需要对来自不同编译路径的模型产物建立统一的输入建模视角，相关异构编译产物的输入模型及其特征将在 3.2.2 节中进行详细介绍，以作为后续语义解析与可调度实体构建的输入基础。

在完成对异构编译产物的统一建模与语义解析之后，语义转换层的核心输出是一组封闭式可调度实体，用于承载推理执行所需的完整执行语义

并作为运行期调度与执行的基本单元。该可调度实体以算子为中心，对其执行所需的函数入口、输入输出张量描述、权重与常量数据以及必要的执行属性进行统一封装，从而将原本分散于计算图结构、算子定义和设备相关配置中的模型语义信息，收敛为结构稳定、接口一致的数据结构。通过对可调度实体进行封闭化设计，运行期调度引擎无需再感知高层模型结构或编译产物差异，即可基于实体中显式描述的执行依赖与属性进行调度与触发执行，从而显著简化运行时组织逻辑并增强执行行为的确定性与可预测性。相关封闭式可调度实体的数据结构设计及其语义组成将在 3.2.3 节中进行详细说明，作为后续算子组装与调度执行的统一输出模型。

在语义转换层中，算子组装器作为连接异构编译产物输入模型与封闭式可调度实体输出模型的关键机制，其设计目标是在运行时初始化阶段完成针对静态推理图与已确定执行配置的模型执行语义最终构建，并严格早于任何推理调度与实际执行过程。算子组装器以 3.2.2 节所定义的异构编译产物输入模型为输入，针对不同编译路径和表示形式的模型产物，按照统一的执行语义抽象，对其中携带的算子描述、张量信息、参数与常量数据以及异构属性标签进行解析与重组，并将这些信息映射到 3.2.3 节所定义的封闭式可调度实体数据结构中。通过在模型加载与运行时初始化阶段一次性完成上述组装过程，算子组装器有效消除了推理执行阶段对编译产物差异和高层模型语义的感知需求，使后续调度引擎仅需面向结构统一、语义完备的可调度实体开展调度与执行控制，从而在保证执行正确性的同时提升运行期行为的确定性与可预测性。算子组装器的整体设计思想及其在运行时初始化阶段的组装流程将在 3.2.4 节中进行详细阐述。

综上所述，语义转换层通过对异构编译产物的统一建模、封闭式可调度实体的结构化定义以及运行时初始化阶段的算子组装机制，实现了从编译阶段模型产物到运行期执行对象之间的语义收敛与形态转换。该层在运行时初始化阶段一次性完成执行语义的构建，使推理执行阶段无需再解析

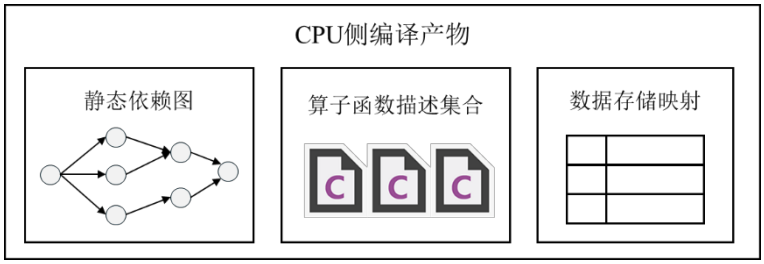
高层模型结构或感知编译产物差异，从而显著简化运行时组织逻辑并消除动态决策带来的不确定性。通过向下游提供结构稳定、语义完备的可调度实体，语义转换层为后续调度引擎层与轻量化并发管理开展确定性调度与受控执行提供了统一且可靠的输入基础，构成整个面向嵌入式操作系统的轻量化异构推理运行时框架中承上启下的关键组成部分，其相关方案的具体实现将在 3.2.5 节中进行详细概述。

3.2.2 异构编译产物的输入模型

在语义转换层中，算子组装器在运行时初始化阶段以编译阶段生成的模型产物作为输入，对其进行解析与重组，以构建运行期可调度的执行实体。由于面向嵌入式操作系统的异构计算平台通常同时包含 CPU、GPU 及国产 NPU 等多类计算资源，不同编译路径所生成的模型产物在表示形式、语义粒度及信息显式程度等方面存在显著差异，难以直接采用统一的处理方式。因此，有必要在语义转换层中为异构编译产物建立明确的输入模型，从执行语义的角度对各类编译产物所提供的关键信息进行抽象与归纳，屏蔽其底层表示差异，为后续算子组装与可调度实体构建提供统一、清晰的输入基础。基于上述考虑，下面将分别对面向 CPU、GPU 及国产 NPU 的编译产物输入模型进行说明。

（1）CPU 侧编译产物的输入模型

在语义转换层中，CPU 侧编译产物输入模型用于对 TVM 等通用编译框架生成的模型产物进行统一抽象与规范化表示，以支撑后续算子组装与可调度实体的构建。由于轻量化实时运行时需要对编译产物进行深入解析并消除其与具体编译框架之间的耦合关系，因此有必要从执行语义的角度对 CPU 侧编译产物所包含的关键信息进行结构化建模。



3-45 CPU 侧编译产物的输入模型

如图 3-45 所示，CPU 侧编译产物输入模型从 TVM 编译输出中抽象并规范化出以下三个核心组成部分：

1. 静态依赖图。该依赖图以拓扑有序的图结构形式刻画算子实例之间的有向数据依赖关系，是输入模型中对执行控制语义的集中表达。语义转换层通过解析该依赖图，可显式获取算子之间的执行顺序约束与同步关系，为后续算子组装与调度执行提供确定的偏序基础，从而保证推理执行过程的正确性与可重复性。

2. 算子函数描述集合。该部分对编译产物中包含的算子实例进行统一描述，明确算子的标识信息、语义类型以及对应的可执行函数入口。通过对算子函数接口的规范化抽象，输入模型在保持与编译期语义一致性的同时，为运行时提供稳定、解耦的算子调用接口，使算子能够在轻量化运行时环境中独立加载与执行。

3. 数据存储映射关系。输入模型通过数据存储映射明确算子输出张量及其元数据信息在内存中的静态归属关系，用于描述算子间的数据传递路径与访问方式。该映射关系使语义转换层能够在初始化阶段直接建立算子之间的数据引用关系，从而避免运行期额外的数据拷贝与动态内存分配，实现对内存使用的可控管理。

通过上述抽象建模，CPU 侧编译产物输入模型在控制面（执行依赖与顺序约束）与数据面（张量及存储关系）上对编译产物所包含的执行语义进行了清晰分离，为语义转换层后续的算子解析、可调度实体构建以及确定性调度执行提供了统一、稳定的输入基础。

（2）GPU 编译产物的输入模型

在语义转换层中，GPU 侧编译产物输入模型用于对 TensorRT、OpenCL 等 GPU 编译工具链生成的模型产物进行统一抽象与结构化建模，以支撑后续算子组装与可调度实体的构建。如图 3-46 所示，相较于 CPU 侧编译产物，GPU 编译产物通常在编译阶段已完成更深度的算子融合与硬件相关优化，其表示形式中隐含了大量与 GPU 执行模型和资源使用方式相关的信息。因此，有必要从执行语义的角度对 GPU 编译产物中显式或隐式包含的关键信息进行规范化抽取，以屏蔽底层硬件与工具链差异。

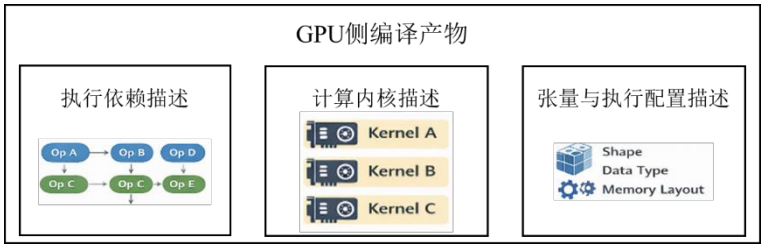


图 3-46 GPU 侧编译产物的输入模型

GPU 侧编译产物输入模型主要由以下三个组成部分构成：

1. 执行依赖描述。GPU 编译产物通常以融合计算图或执行序列的形式隐含算子之间的执行顺序与依赖关系。输入模型通过对编译产物中相关信息的解析，将其抽象为显式的执行依赖描述，用于刻画算子实例之间的先后关系与同步约束，为后续调度与执行控制提供必要的控制语义基础。

2. 计算内核描述集合。该部分对 GPU 编译产物中包含的计算内核进行统一描述，明确每个算子或融合算子对应的可执行内核入口及其语义标识。计算内核描述封装了已经针对 GPU 架构优化后的执行实现，使运行时无需再感知高层算子结构或具体编译过程，即可直接触发相应内核执行。

3. 张量与执行配置描述。输入模型通过张量描述信息刻画各算子输入输出张量的形状、数据类型及存储布局，并结合执行配置描述显式表达与 GPU 执行相关的资源使用方式，如线程组织、内存分配策略等。通过将上述信息统一建模，语义转换层能够在运行时初始化阶段完成对 GPU 执行环境的必要配置，从而避免推理执行阶段的动态解析与决策。

通过上述抽象建模，GPU 侧编译产物输入模型在控制面（执行依赖）、执行面（计算内核）与配置面（资源与张量描述）上对 GPU 编译产物所包含的执行语义进行了规范化表达，为语义转换层后续的算子组装与可调度实体构建提供了统一、稳定的输入基础。

（3）国产 NPU 编译产物的输入模型

在语义转换层中，国产 NPU 编译产物输入模型用于对各类厂商专用工具链生成的 NPU 编译产物进行统一抽象与结构化建模，以支撑异构推理执行场景下算子组装与可调度实体的构建。如图 3-47 所示，相较于 CPU 与 GPU 编译产物，国产 NPU 编译产物通常具有更强的硬件特化特征，其表示形式高度贴近底层硬件执行模型，并在编译阶段已隐含具体的执行流程、资源分配方式及内存访问策略。因此，有必要在语义转换层中对 NPU 编译产物中所携带的执行语义信息进行规范化抽取，以屏蔽厂商工具链差异并保持运行时的统一处理方式。

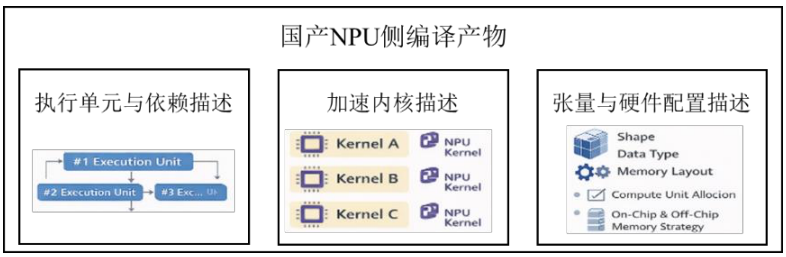


图 3-47 国产 NPU 侧编译产物的输入模型

国产 NPU 编译产物输入模型主要由以下三个组成部分构成：

1. 执行单元与依赖描述。国产 NPU 编译产物通常以高度融合的执行单元或编译工件形式存在，其内部已确定算子或算子组的执行顺序与依赖关系。输入模型通过对相关信息的解析，将这些隐含的执行关系抽象为显式的执行依赖描述，用于刻画不同执行单元之间的先后关系与同步约束，为后续调度与执行控制提供必要的控制语义基础。

2. 加速内核描述集合。该部分对 NPU 编译产物中包含的硬件特化加速内核进行统一描述，明确各执行单元对应的可执行内核入口及其语义标识。

加速内核描述封装已针对特定 NPU 架构优化完成的执行实现，使运行时无需感知高层算子结构或具体编译细节，即可直接触发 NPU 上的推理执行。

3.张量与硬件配置描述。输入模型通过张量描述信息刻画各执行单元输入输出张量的形状、数据类型及其在 NPU 上的存储布局，并结合硬件配置描述显式表达与 NPU 执行相关的资源使用方式，如计算单元分配、片上与片外存储使用策略等。通过对上述信息的统一建模，语义转换层能够在运行时初始化阶段完成对 NPU 执行环境的必要配置，避免推理执行阶段对厂商特定接口与配置细节的动态解析。

通过上述抽象建模,国产 NPU 编译产物输入模型在控制面(执行依赖)、执行面（加速内核）与配置面（硬件与张量描述）上对 NPU 编译产物所包含的执行语义进行了规范化表达，为语义转换层后续的算子组装与可调度实体构建提供了统一、稳定的输入基础。

3.2.3 封闭式可调度实体数据结构

封闭式可调度实体数据结构是语义转换层的核心输出，用于将编译阶段生成的异构模型语义信息封装为统一、稳定的运行期执行描述对象，从而为调度引擎提供直接可用的执行单元。每个可调度实体对应一个独立的推理执行单元，完整描述其在目标计算资源上执行所需的全部执行语义，包括计算内核入口、输入输出张量描述、权重与常量数据以及相关执行配置等。通过对上述信息的统一封装，运行期无需再解析高层计算图结构或动态构建执行语义，即可基于可调度实体直接触发推理执行。

可调度实体的设计遵循静态性与封闭性原则：实体在运行时初始化阶段构建完成后，其内部结构与语义内容保持不变，且执行所需的关键信息均以显式方式封装在实体之中，运行期不再依赖对编译产物差异或高层模型语义的动态解析。这种设计有效降低了运行期的控制复杂度与决策开销，为推理执行行为的确定性与可预测性提供了基础保障。同时，由于可调度

实体对执行语义进行了统一建模，不同平台上的运行时系统能够在保持一致执行语义的前提下对其进行适配与调用。

（1）可调度实体的统一语义组成

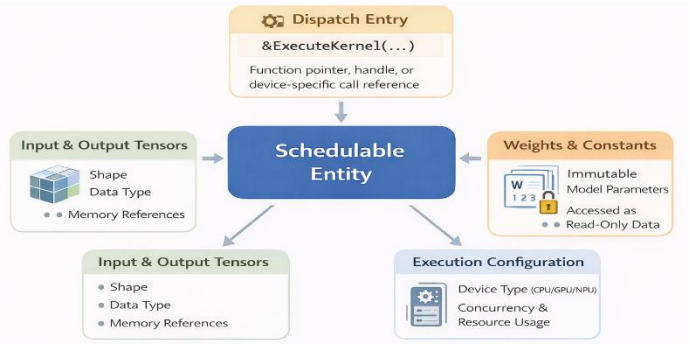


图 3-48 可调度实体的统一语义组成

可调度实体的核心目标是在运行时提供结构化、完备的执行语义描述，使调度引擎能够在不感知高层模型结构的情况下直接组织与触发推理执行。如图 3-48 所示，每个可调度实体由以下几个关键语义要素构成：

1. 执行入口：指向具体的计算内核或算子执行入口，可表现为函数指针、执行句柄或设备相关调用描述，用以明确执行单元的实际计算逻辑。

2. 输入输出张量：对执行单元所涉及的输入与输出张量进行统一描述，包括张量形状、数据类型、存储布局及其在内存中的引用关系，为执行过程中的数据访问提供明确约定。

3. 权重与常量数据：用于封装模型中的权重参数与常量信息，这些数据在模型加载与初始化阶段完成绑定，并在执行过程中作为只读数据被计算内核直接访问。

4. 执行配置描述：用于表达与目标计算资源相关的执行属性与约束信息，如执行设备类型（CPU/GPU/NPU）、并行度参数及资源使用方式等，为运行时在具体平台上正确触发执行提供必要的上下文信息。

通过上述语义要素的统一封装，可调度实体在初始化完成后即可作为独立的执行描述对象被调度引擎直接使用，从而避免运行期对执行语义的重复解析，提升推理执行的效率与确定性。

（2）面向通用操作系统的平台执行绑定

在通用操作系统平台上，可调度实体的执行绑定主要用于建立实体与底层计算资源（CPU/GPU/NPU）之间的执行关联关系，使运行时系统能够借助操作系统提供的线程管理、内存管理与设备驱动机制完成推理执行。语义转换层在初始化阶段为可调度实体配置与目标平台相匹配的执行上下文信息，包括执行设备类型、并行度参数及资源使用约定等。

在执行绑定过程中，系统会为每个可调度实体分配特定的资源，如 CPU 核心、GPU 计算单元等，并设置相应的执行属性（如线程数、并行度等）。在执行时，调度引擎将根据这些绑定信息，调度可调度实体到合适的计算资源上进行计算。同时，操作系统提供的内存管理机制将确保数据在内存中的正确布局和高效访问。由于操作系统调度的动态特性，通用操作系统平台能够提供较大的灵活性和资源共享能力，但相应的可能会引入一些不确定性和调度开销。

（3）面向嵌入式操作系统的平台执行绑定

相较于通用操作系统平台，面向嵌入式操作系统的平台执行绑定更加注重执行行为的确定性、资源使用的可控性以及系统运行的可靠性。由于嵌入式系统通常具有严格的时序约束和受限的计算与存储资源，可调度实体在该类平台上的执行绑定需要在运行时初始化阶段完成尽可能多的执行语义固化工作，从而减少推理执行阶段的动态决策与系统干预。

在嵌入式操作系统环境中，语义转换层根据平台特性为可调度实体建立明确、静态的执行绑定关系，包括目标硬件资源类型、执行顺序约束以及必要的资源使用配置。通过在初始化阶段完成上述绑定，可调度实体在执行阶段能够沿预先确定的执行路径被调度引擎直接触发，而无需依赖复杂的运行期调度策略。

为进一步降低运行期不确定性，嵌入式平台下的执行绑定通常结合静态内存布局、受控的调度机制以及最小化的系统调用路径进行设计。推理

执行过程中，操作系统主要承担基础的资源管理与中断处理职责，而不参与高层执行语义的解析与决策，从而有效减少上下文切换与调度抖动对执行时序的影响，确保推理过程满足实时性与高可靠性的要求。

3.2.4 算子组装器的设计思想

算子组装器是语义转换层中的核心组成部分，其主要职责是在运行时初始化阶段，对编译阶段生成的异构计算图语义信息进行解析与重组，将算子描述、张量关系、计算内核入口及相关执行配置等关键信息统一封装为结构稳定、语义完备的可调度实体，从而完成从编译期逻辑模型到运行期执行描述对象的关键转换。算子组装器的设计遵循静态化执行、硬件适配性与运行时简化的原则，通过在模型加载与初始化阶段预先固化算子的执行路径、资源使用方式及必要的执行语义，使推理执行阶段无需再进行高层模型解析或复杂的运行期决策，进而降低系统整体复杂度并提升执行行为的确定性。

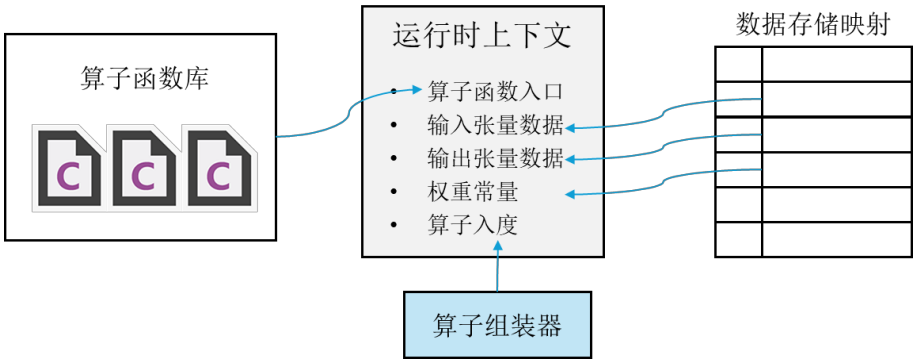


图 3-49 可调度实体的组装示意图

可调度实体在运行时初始化阶段的组装示意流程如图所示。具体而言，算子组装器在初始化阶段依据调度解释器输出的就绪算子序列，面向语义转换层中已统一建模的异构编译产物输入模型，对其中对应的算子或执行单元进行解析，提取其静态化的执行语义信息。随后，算子组装器向运行时上下文池申请可用的上下文槽位，并基于已构建的可调度实体完成执行上下文的装配。装配过程主要包括：绑定算子或执行单元对应的计算内核

入口，关联权重与常量数据所在的只读存储区域，解析输入张量的来源（如工作区、常量区或外部 I/O 绑定缓冲区），并配置输出张量的目标写入位置，以形成完整、可直接触发执行的运行时上下文实例。

从整体流程上看，算子组装器的工作可概括为三个关键步骤：解析异构编译产物输入模型、提取并组织执行单元的静态执行信息，以及构建与目标计算平台适配的执行上下文。通过上述过程，算子组装器将编译阶段计算图中的逻辑描述映射为硬件资源上的可执行实体，为后续调度引擎提供结构统一、接口稳定的执行对象。其核心思想在于通过在初始化阶段预先固化执行路径与资源使用方式，避免推理执行阶段的动态决策与资源冲突，从而提升推理执行的确定性与整体运行效率。

（1）算子执行入口项的组装

算子执行入口项的组装是算子组装器在运行时初始化阶段完成的首个关键步骤，其目标是在可调度实体中明确描述每个算子或执行单元对应的可执行实现。编译阶段，算子的计算逻辑通常被转换为与目标计算平台相关的执行形式，例如 CPU 平台上的函数实现、GPU 平台上的计算内核或国产 NPU 平台上的加速内核工件。算子组装器通过解析异构编译产物输入模型，提取上述执行实现的入口信息，并将其统一封装为可调度实体中的执行入口描述，为后续推理执行提供明确的调用依据。如图 3-50 所示，在 CPU 平台场景下，算子执行入口通常表现为标准的函数指针或调用接口，其执行语义相对直接；而在 GPU 或 NPU 场景下，执行入口描述往往还需要包含与内核调用相关的附加信息，如执行句柄、设备调用参数或必要的执行配置。算子组装器在该阶段并不绑定具体的硬件资源实例，而是将执行入口与其适用的计算资源类型及执行约束进行关联，并封装进可调度实体之中。

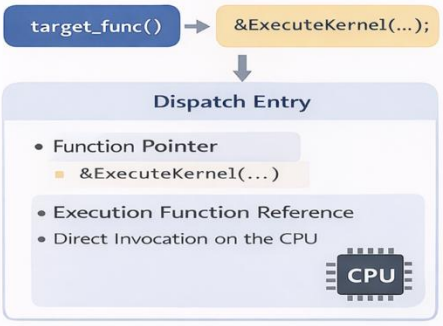


图 3-50 CPU 平台上的算子执行入口的识别抽象

通过完成执行入口项的统一组装，调度引擎在推理执行阶段无需再解析编译产物或区分底层编译路径，即可基于可调度实体中已明确的执行入口触发相应计算内核的执行。该过程通过在初始化阶段固化执行入口的映射关系，避免了运行期对执行实现的动态解析与高层决策，为后续调度与执行提供了稳定、可预测的基础。

（2）I/O 张量元数据与执行缓冲区映射

在推理执行过程中，输入与输出张量构成算子计算所依赖的核心数据载体。算子组装器的第二个关键任务是在运行时初始化阶段，对计算图中定义的张量元数据进行解析，并建立其与运行时执行缓冲区之间的映射关系，从而为算子或执行单元提供明确、稳定的数据访问约定。如图 3-51 所示，张量元数据用于描述张量的形状、数据类型及存储布局等属性，而执行缓冲区则对应于运行时用于存放实际数据的内存区域。算子组装器通过解析异构编译产物输入模型，提取每个算子相关的输入与输出张量元数据，并结合目标计算平台的内存访问约束与布局要求，为其建立指向相应执行缓冲区的引用关系。该过程并不涉及运行期的数据分配或拷贝，而是在初始化阶段完成张量访问路径的静态确定。

对于每个可调度实体，算子组装器将输入与输出张量的元数据描述及其对应的执行缓冲区引用，与执行入口及相关执行配置一并封装到实体之中。通过上述映射，调度引擎在推理执行阶段无需再解析高层张量关系或

动态决策内存访问方式，即可按照既定的访问约定直接触发计算内核对内存中数据的读写操作，从而保证执行过程的正确性、高效性与可预测性。

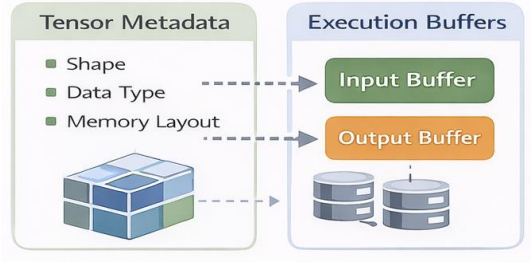


图 3-51 数据存储映射示意图

（3）资源绑定与执行配置注入

资源绑定与执行配置注入是算子组装器在运行时初始化阶段完成的关键步骤，其目的是在可调度实体中明确描述算子或执行单元对目标计算平台的执行要求与资源使用约定。算子组装器根据语义转换层已统一建模的执行语义，结合目标平台的硬件类型（如 CPU、GPU 或国产 NPU），为每个可调度实体注入与其执行相关的资源属性与配置参数，从而为后续推理执行提供清晰、稳定的执行约束。

资源绑定在该阶段并不涉及对具体硬件资源实例的运行期分配，而是用于描述执行单元适用的计算资源类型及其执行方式约定；执行配置则用于表达与执行相关的参数信息，如并行度约束、内存访问要求及必要的执行属性等。在 GPU 或 NPU 场景下，上述配置还可包含与硬件加速相关的执行特征描述。算子组装器将这些资源绑定信息与执行配置统一封装进可调度实体，使调度引擎在推理执行阶段无需再解析底层硬件差异或进行高层资源决策，即可基于既定约定触发执行，从而在保证执行正确性的同时提升运行行为的确定性与整体执行效率。

（4）可调度实体的完整性与封闭化校验

在算子组装器完成可调度实体各项执行语义的组装后，需要对生成的可调度实体进行完整性与封闭化校验，以确认其作为运行期执行单元的自洽性与可用性。完整性校验用于验证可调度实体是否已包含推理执行所必

需的全部关键信息，包括执行入口、输入输出张量描述、数据缓冲区引用以及执行配置等；封闭化校验则用于确认实体中的执行语义已在初始化阶段被充分固化，运行期不再依赖对编译产物、高层模型结构或执行语义的动态解析。通过上述校验过程，算子组装器确保每个可调度实体均具备结构稳定、语义完备的执行描述，从而为后续调度与执行阶段提供可靠、可预测的运行基础。

最终，在完成完整性与封闭化校验后，算子组装器输出结构稳定、语义完备的可调度实体，作为运行期推理执行的基本输入单元。通过在运行时初始化阶段一次性完成执行语义的解析、固化与校验，算子组装器有效将编译期模型语义与运行期执行机制解耦，使推理执行阶段无需再进行高层语义解析或复杂的动态决策，从而显著降低运行时开销，并提升推理执行过程的确定性、可预测性与系统整体运行可靠性。

3.2.5 方案实现

下面将对语义转换层的关键数据结构及函数功能进行详细论述：

（1）SchedulableEntity：可调度实体数据结构实现

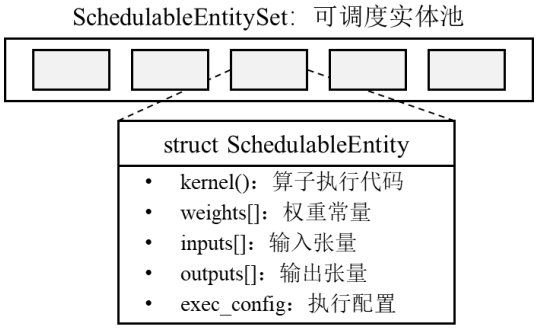


图 3-52 可调度实体数据结构

功能概述：如图所示，SchedulableEntity 是语义转换层输出并被运行时系统直接使用的最小可调度执行描述单元，用于承载单个算子或执行单元在一次推理执行中所需的完整执行语义。该实体在运行时初始化阶段由算子组装器构建完成，将编译阶段生成的计算内核、权重常量以及张量

与执行缓冲区之间的映射关系统一封装为结构稳定的执行视图。SchedulableEntity 的职责仅限于描述算子的执行入口及其数据访问约定，不涉及计算图调度、依赖解析或内存规划等高层逻辑，其输入输出均基于预先准备的执行缓冲区完成，从而支持零拷贝的数据传递，并为后续调度与执行提供封闭且可预测的执行基础。。

核心项说明：SchedulableEntity 的执行语义输入主要由以下四类信息构成，这些输入项均在运行时初始化阶段由算子组装器完成解析与绑定，并在实体生命周期内保持静态不变：

1.kernel()：算子执行入口。指向编译工具链生成的算子或融合子图的执行入口，可表现为函数指针、执行句柄或设备相关的内核标识。其调用参数约定在运行时初始化阶段即被固化，通常采用输入指针数组+输出指针数组+权重只读块的形式，使其能够在运行时任务或工作线程上下文中被直接调用，实现高效、确定的算子执行。

2.inputs[]/outputs[]：I/O 张量指针与缓冲区映射。分别表示算子执行所需的输入与输出张量指针集合，用于建立计算内核与运行时执行缓冲区之间的直接访问关系。每个张量项包含其对应的形状、数据类型及存储布局等元数据信息，并在初始化阶段完成与物理执行缓冲区的绑定。运行期内核通过这些指针直接对内存中的数据进行读写，无需额外的数据拷贝或动态解析，从而提供稳定、高效的数据访问路径。

3.weights[]：常量数据引用。用于记录当前算子执行过程中所需的权重、偏置及常量参数在只读存储区域中的基址信息。权重数据在模型加载与初始化阶段完成绑定，运行期以零拷贝方式只读访问，不在可调度实体内部复制或修改，保证执行过程中的数据一致性与内存使用的可控性。

4.exec_config：执行配置与资源约定。用于描述可调度实体在目标计算平台上的执行要求与约束条件，包括适用的计算资源类型（CPU/GPU/NPU）、并行度参数以及必要的执行属性。该配置不涉及具体硬

件资源实例的运行期分配，而是作为执行语义的一部分在初始化阶段被注入，用于指导运行时在正确的平台与执行环境中触发计算内核。

（2）SchedulableEntitySet：可调度实体池

功能概述：SchedulableEntitySet 是运行时系统中用于集中管理全部可调度实体的实体集合结构，其作用是在模型加载与运行时初始化阶段，为算子组装器与调度引擎提供统一、稳定的可调度实体存储与索引空间。该集合在初始化阶段完成一次性构建，其内容在推理执行过程中保持只读状态，不发生动态增删或结构性变化，从而避免运行期的并发修改与一致性问题。

核心项说明：1.entities[]：可调度实体数组。用于顺序存储系统中全部 SchedulableEntity 实例的实体数组或索引化容器。每个实体在运行时初始化阶段由算子组装器构建完成后被加入该集合，其在集合中的存储位置在整个推理执行过程中保持稳定，用于支持调度与执行阶段通过实体索引或句柄进行快速、确定的访问。

2.entity_count：实体数量。用于记录当前模型中包含的可调度实体总数。该字段在初始化完成后保持不变，为调度解释器、并发管理层及算子就绪队列提供明确的集合边界信息，避免运行期对集合状态进行动态查询或一致性判定。

3.entity_id：映射关系。用于维护编译阶段算子标识或输入模型中执行单元标识与 SchedulableEntity 在集合中索引位置之间的映射关系。该映射在初始化阶段构建完成，使运行时系统能够在不解析高层模型结构的情况下，通过稳定标识快速定位对应的可调度实体。

（3）OperatorExecutionAssembly()：算子组装器函数

功能概述：OperatorExecutionAssembly()是语义转换层在实现层面的核心组装函数，其主要职责是在运行时初始化阶段，面向统一建模后的异构编译产物输入模型，按照调度解释器给出的算子或执行单元顺序，完

成可调度实体的构建与注册。该函数在初始化阶段依次完成执行入口解析、I/O 张量映射、执行配置注入以及完整性与封闭化校验，并将构建完成的可调度实体纳入运行时管理结构，使推理执行阶段能够直接基于已固化的执行语义触发执行，从而为后续调度与推理运行提供结构稳定、语义完备的执行对象。其伪代码流程如图 3-53 所示：

1. **初始化阶段：准备执行环境**：
   - 按照调度器指定的顺序，遍历每个算子（`operator`）。这意味着每个算子都会按照预定的顺序逐个进行处理。
2. **执行入口解析**（`parse_execution_entry`）：
   - 对每个算子，解析其执行入口，确定执行方法或函数。这个步骤帮助识别算子的执行方式，并根据输入模型对其进行初始化。
3. **I/O 张量映射**（`map_input_tensors`，`map_output_tensors`）：
   - 对算子的输入张量进行映射，确保从输入模型中正确提取相关的数据。类似地，也要确保输出张量能够正确映射。
4. **执行配置注入**（`inject_execution_config`）：
   - 向算子的执行上下文中注入必要的执行配置。这些配置可能包括线程数、执行优化标志等参数，以支持高效执行。
5. **完整性与封闭化校验**（`validate_integrity`）：
   - 在执行算子之前，进行完整性和封闭性校验，确保该算子能够独立正确执行，并且没有外部依赖冲突。如果检查失败，抛出异常。
6. **创建运行时上下文**（`create_runtime_context`）：
   - 根据算子的输入、输出张量以及执行配置，创建该算子的运行时上下文。运行时上下文包含执行该算子所需的所有信息。
7. **将算子注册为可调度实体**（`register_operator`）：
   - 将该算子的运行时上下文注册到调度状态中，确保算子可以被调度解释器识别并纳入调度管理。
8. **调度队列管理**（`add_to_pending_queue`）：
   - 检查该算子是否满足就绪条件。如果是，将该算子加入到挂起队列中，等待调度执行。
9. **内存管理**（`allocate_memory`）：
   - 为算子的输入和输出张量分配内存，确保在执行过程中数据能够正确访问。
10. **初始化完成后返回**：
    - 在处理完所有算子之后，函数返回一个表示初始化完成的状态信息（例如`"Assembly Complete"`）。

图 3-53 算子组装器的伪代码工作流程

3.3 调度引擎层

3.3.1 基本思路

在完成语义转换层对异构编译产物的统一建模与可调度实体构建之后，运行时系统进入以执行控制为核心的调度引擎层。调度引擎层以上游

输出的封闭式可调度实体为基本调度对象，负责在运行期根据实体间的执行依赖关系以及目标计算平台的运行约束，组织并触发推理执行过程。与语义转换层侧重执行语义的解析与固化不同，调度引擎层关注的是在既定执行语义约束下，对执行顺序与触发时机进行受控管理，从而保证推理执行的正确性、确定性与整体运行效率。

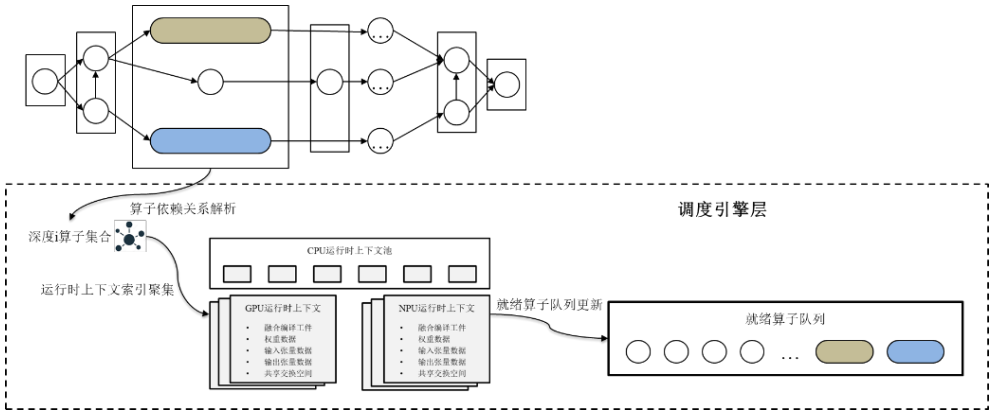


图 3-54 调度引擎层的基本思路

如图 3-54 所示，调度引擎层并不直接解析高层模型结构或编译产物细节，而是以语义转换层输出的可调度实体集合为输入，对其进行运行期组织与调度控制。由于可调度实体已在初始化阶段完成执行语义的封闭化描述，调度引擎层仅需依据实体中显式表达的执行依赖、资源约定及就绪条件，对执行过程进行调度与触发，从而有效避免运行期对模型语义的再次解析与感知。具体而言，调度引擎层的核心职责可概括为以下三个方面：

1. 算子执行依赖关系解析与维护。调度引擎层依据可调度实体中显式描述的依赖信息，对实体之间的前驱—后继关系进行解析，并在运行期维护对应的依赖状态。当某一可调度实体的前驱实体执行完成后，其依赖计数随之更新，从而为判断该实体是否具备执行条件提供依据。该过程不涉及高层计算图的遍历，而是基于初始化阶段已固化的依赖关系进行轻量化状态维护。

2. 可调度实体索引聚集与访问管理。为支持高效、可预测的调度决策与执行触发，调度引擎层通过对可调度实体集合进行索引化管理，实现对

实体的快速定位与访问。调度引擎仅通过稳定的实体标识或索引引用对应的可调度实体，而不直接操作其内部执行语义，从而在执行控制层面保持清晰的职责边界，并避免运行期对实体内容的结构性干预。

3.就绪算子队列的更新与调度触发。在运行期，当可调度实体的执行依赖条件被满足后，调度引擎层将其加入就绪算子队列，作为后续执行触发的候选对象。并发管理模块或工作线程从就绪算子队列中获取待执行实体，并依据既定的调度策略触发其执行入口。由于进入就绪队列的实体均已具备完整且封闭的执行语义，该过程无需额外的运行期语义检查，从而有效降低调度开销并提升执行时序的可预测性。

综上所述，调度引擎层通过对可调度实体执行依赖的受控维护、实体索引的集中管理以及就绪算子队列的有序更新，实现对推理执行过程的确定性调度与触发控制。该层在整体架构中承接语义转换层输出的可调度实体，并向下游并发管理与执行机制提供清晰、稳定的调度接口，与语义转换层共同构成面向嵌入式操作系统的轻量化异构推理运行时框架的核心执行控制路径，其具体实现方案将在后续小节中进一步展开说明。

3.3.2 算子依赖关系解析

算子依赖关系解析是调度引擎层中的一项关键机制，其目标是在既定执行语义约束下，明确算子或可调度实体之间的前驱–后继关系，为推理执行过程中的调度与触发提供可靠依据。在该过程中，系统依据编译阶段与初始化阶段已固化的计算图结构，对算子之间的数据依赖关系进行解析，确定哪些算子必须顺序执行、哪些算子在依赖条件满足后可以并行触发，从而保证推理执行过程的正确性与整体效率。

算子依赖关系解析主要包括依赖关系识别与依赖状态维护两个方面。在依赖关系识别阶段，系统通过分析算子输入与输出张量之间的连接关系，明确每个算子所依赖的前驱算子集合，并据此建立稳定的依赖描述；在运行期，调度引擎层通过维护对应的依赖计数或就绪状态，判断算子是否具

备执行条件。当某一算子的所有前驱执行完成后，其依赖条件被满足，便可进入就绪状态，参与后续的调度与执行。通过上述方式，调度引擎层在不进行高层计算图遍历或复杂运行期决策的前提下，实现了对算子执行顺序与并行性的受控管理。

（1）静态算子计算图的结构化查询

静态算子计算图的结构化查询是算子依赖关系解析的基础环节，其目标是在运行时初始化阶段，通过对计算图结构的静态分析，提取并明确算子之间的输入–输出依赖关系。计算图由算子节点及其间的数据流连接构成，通过对节点及边的结构化查询，可以系统性地识别算子之间的前驱–后继关系，从而确定算子执行所需满足的依赖条件。

在该过程中，系统对计算图进行一次性遍历，解析每个算子的输入张量来源及输出张量的使用去向，进而明确哪些算子在依赖条件满足后可以并行执行，哪些算子必须等待前驱算子完成后才能触发执行。通过将计算图中的依赖关系固化为稳定的调度描述信息，运行期无需再进行计算图遍历或动态依赖分析，从而避免潜在的依赖冲突，确保推理执行过程的正确性与确定性。

（2）算子拓扑深度的计算原则

图 3-55　算子拓扑深度的计算示意图

算子拓扑深度计算用于对计算图中算子的执行先后关系进行度量，是调度引擎在初始化阶段确定执行顺序与依赖层级的重要依据。拓扑深度基

于计算图的拓扑结构进行计算，每个算子的深度由其输入依赖及前驱算子的深度共同决定，用以反映该算子在整体执行路径中的相对位置。

如图 3-55 所示，拓扑深度的计算遵循以下基本原则：对于不存在前驱算子的算子，其拓扑深度初始化为 0；对于存在前驱算子的算子，其拓扑深度取所有前驱算子深度的最大值并加 1。通过上述递推规则，系统能够为计算图中的每个算子确定唯一的拓扑深度值，并据此形成符合依赖约束的执行顺序。在此基础上，拓扑深度不仅用于保证算子执行顺序的正确性，也为并行执行提供参考依据：拓扑深度较小且依赖已满足的算子可优先或并行触发执行，而拓扑深度较大的算子需等待其前驱算子完成后方可进入就绪状态，从而为推理执行过程中的高效调度与并行性管理提供支撑。

通过计算拓扑深度，系统不仅能够确定算子执行的顺序，还能为并行执行提供依据。例如，深度较小的算子可以先行执行，深度较大的算子需要等其前驱算子完成后才能执行。这为推理任务的并行性和高效调度提供了关键支持。

（3）就绪算子索引集合的构建

就绪算子索引集合的构建是算子依赖关系解析过程中的最后一个关键环节，其目的是在运行期为调度引擎维护一组当前满足执行条件的算子或可调度实体。通过前述依赖关系解析与拓扑深度计算，系统能够明确各算子的前驱依赖状态，从而判断哪些算子已具备执行条件并可被立即触发。

在初始化阶段，系统依据算子依赖关系与入度信息，识别所有不存在未完成前驱依赖的算子，将其对应的索引或标识加入就绪算子索引集合中。该集合仅保存可直接参与执行调度的算子标识，用于支持调度引擎在运行期对可执行对象的快速定位与访问，而不涉及对算子内部执行语义的解析。

在推理执行过程中，随着算子逐步完成执行，其后继算子的依赖状态随之更新。当某一算子的所有前驱依赖被满足后，其索引将被加入就绪算子索引集合，从而进入可执行状态。通过对就绪算子索引集合的受控维护，

调度引擎能够在不进行高层计算图遍历或复杂运行期分析的前提下，持续获得当前可执行算子的候选集合，为后续调度与执行触发提供稳定、高效的执行路径，确保推理过程的正确性与整体运行效率。

核心引擎层的功能目标在于忠实践行编译阶段预设的调度策略，将静态确定的调度依赖关系转化为运行期可执行的算子调度序列。该层以系统接口层输出的运行时上下文为输入，通过对算子执行依赖的维护与执行状态的更新，输出严格遵循 DAG 约束的算子调用顺序，从而确保推理过程在实际运行中保持与理论调度模型一致的确定性执行行为。核心引擎层以调度解释器为核心组件，负责加载并解析序列化调度表，维护算子级执行状态，并持续跟踪 DAG 的执行进度以识别当前可执行的算子集合。

3.3.3 调度解释器驱动的可调度实体聚集

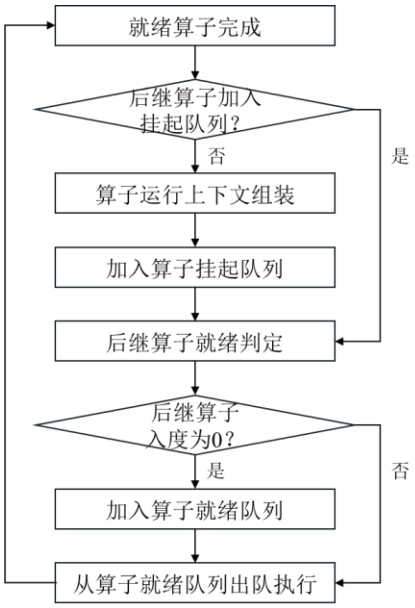


图 3-56 调度解释器的工作流程

如图 3-56 所示，当某一工作线程完成一个运行时上下文的执行后，调度解释器被触发以更新调度状态。首先，系统判断该上下文对应的后继算子是否已在算子挂起队列中完成注册；若尚未存在，则由算子组装器为该后继算子创建并组装对应的运行时上下文，并将其加入算子挂起队列。随后，调度解释器依据已固化的依赖关系，对算子挂起队列中相关后继算子

的入度计数进行更新，并据此执行就绪判定：当某一算子的入度递减至零时，表明其所有前驱算子均已执行完成，该算子已满足执行条件。

在就绪判定成立后，系统将对应算子的运行时上下文从算子挂起队列转移至算子就绪队列，等待后续的调度与执行触发。通过上述过程，调度解释器实现了对算子执行依赖状态的持续维护与就绪算子索引的动态聚集，使得算子调度能够在严格遵循拓扑依赖关系的前提下有序推进。同时，该机制在算子执行、状态更新与调度触发之间形成闭环，使运行时系统能够对整体计算流程进行精确控制，在保证确定性执行的同时，提高并行资源的利用效率。

3.3.4 方案实现

调度引擎层的整体实现方案是在系统初始化阶段创建并常驻一个调度器任务（SchedulerTask），该任务在整个推理执行周期内持续运行，并在任意 Worker 线程完成某个可调度实体对应的运行时上下文执行后，以事件驱动方式被激活。当调度器任务被唤醒时，其内部的调度解释器首先从完成算子队列中取出已执行结束的可调度实体，依据静态调度表执行最后一次使用判定，对已不存在后继依赖的输出张量回收或释放对应的存储块，以实现运行时内存复用。接着，调度解释器遍历这些已完成可调度实体的后继实体：若后继实体尚未存在于挂起队列中，则由算子组装器为其创建并组装对应的运行时上下文，并将其加入算子挂起队列（PendingOperatorQueue）；接着，对相关后继实体的依赖计数（entry_count)逐一递减,当检测到某一可调度实体的依赖计数降为 0 时，表明其已满足执行就绪条件，调度解释器便将其运行时上下文从挂起队列转移至算子就绪队列（ReadyOperatorQueue）。通过上述机制，调度器任务在运行时形成了回收—判定—组装—推进—入队的闭环调度流程：既负责可调度实体依赖状态的动态维护与运行时上下文的受控构建，也承担就

绪状态触发与执行派发，是系统在单核模式下的主执行循环入口，同时也是多核并行环境中协调任务流转与资源回收的核心调度单元。

（1）ReadyOperatorQueue：就绪算子队列

功能概述：ReadyOperatorQueue 是调度引擎层与并发管理层之间的关键交互队列，用于维护当前已满足执行依赖条件、可被立即触发执行的可调度实体集合。调度解释器在依赖计数更新与就绪判定完成后，将入度归零的可调度实体对应运行时上下文推入 ReadyOperatorQueue；并发管理层的工作线程从该队列中获取待执行对象并触发其执行入口。由于进入就绪队列的对象均已完成执行语义固化与上下文装配，该队列不承担模型语义解析与资源规划职责，仅提供可执行对象的稳定供给，从而降低运行期调度开销并提升执行时序的可预测性。

（2）CompleteOperatorQueue：完成算子队列

功能概述：CompleteOperatorQueue 用于承载已执行完成的可调度实体完成事件，是调度引擎层推进 DAG 执行与触发资源回收的输入通道。工作线程在完成某个可调度实体对应运行时上下文的执行后，将该实体的完成标识推入 CompleteOperatorQueue；调度器任务被事件驱动唤醒后，调度解释器从该队列批量取出已完成实体，执行最后一次使用判定与工作区块回收，并依据静态调度表推进其后继实体的依赖计数更新与就绪判定。通过该队列，运行时在执行完成、状态回写、依赖推进、就绪入队的链路上形成闭环，避免工作线程承担调度推进与回收逻辑，从而维持职责边界清晰并提升系统整体确定性。

（3）SchedulingInterpreter()（调度解释器运行函数）

功能概述：调度器任务的核心执行例程，其职责是在调度事件触发后，对算子完成队列（CompleteOperatorQueue）中的节点进行解析与处理。函数的核心作用包括：1.执行最后一次使用判定，对已无后继依赖的输出张量进行工作区块回收，维持内存占用的动态平衡；2.根据静态调度表更新

调度状态机（ScheduleState）中的依赖计数信息，对所有相关后继节点递减入度；3.对入度归零的节点进行就绪判定，并将其运行时上下文转移至就绪算子队列（ReadyOperatorQueue）中，以供并发管理层的 Worker 线程调度执行。通过该过程，调度解释器实现了从资源回收到依赖推进再到任务入队的闭环逻辑，确保算子执行在多线程环境下的有序性与高效性。

1. **初始化**:
   - `ready_enq = 0`：初始化一个计数器，用于统计新增的就绪节点数量。
   - `complete_nodes = sched_state.get_completed_nodes()`：获取所有已完成的计算节点集合。
2. **遍历所有已完成节点**:
   - 对每个已完成的节点 `node` 进行以下处理:

     a. **标记该节点为"已完成"状态**:

     - `sched_state[node].status = DONE`：将当前节点的状态设置为"已完成"（`DONE`）。

     b. **执行"最后一次使用"判定**:

     - `if sched_state.is_last_use(node)`：判断该节点的输出是否是最后一次被使用。
     - 如果是最后一次使用，则释放该节点的输出内存块：`mem_mgr.release(node.output_blocks)`。

     c. **遍历该节点的后继节点，更新它们的依赖计数**:

     - 对每个后继节点 `succ` 进行以下操作:

       i. **如果后继节点尚未注册**:

       - `if not sched_state.exists(succ)`：检查该后继节点是否已经存在于调度状态中。如果不存在，则:
         - `ctx = Assembler.build_runtime_context(succ)`：构建该后继节点的运行时上下文。
         - `sched_state.register(succ, ctx)`：将该后继节点及其上下文注册到调度状态中。
         - `pending_queue.push_unique(ctx)`：将该节点的上下文加入挂起队列，确保该节点在未来执行。

       ii. **递减后继节点的入度计数**:

       - `if sched_state[succ].indegree > 0`：检查后继节点的入度是否大于0。
       - `sched_state[succ].indegree -= 1`：递减该后继节点的入度计数。

       iii. **如果入度为0且节点状态不是"就绪"**:

       - `if sched_state[succ].indegree == 0 and sched_state[succ].status != READY`：判断后继节点的入度是否已经降为0，并且节点状态不是"就绪"（`READY`）。
       - 如果满足条件，表示该后继节点已满足就绪条件:
         - `sched_state[succ].status = READY`：将该节点的状态更新为"就绪"。
         - `pending_queue.erase(succ)`：从挂起队列中移除该节点。
         - `ready_queue.push(succ)`：将该节点加入到就绪队列中。
         - `ready_enq += 1`：增加新增就绪节点的计数。
3. **清空完成节点标记**:
   - `sched_state.clear_completed_nodes()`：清空已完成节点的标记，防止重复处理。
4. **返回新增就绪节点的数量**:
   - `return ready_enq`：返回本轮中新增的就绪节点数量。

图 3-57　调度解释器的伪代码工作流程

输入参数：1.ScheduleState*sched_state：调度状态机实例，记录各算子的执行状态、依赖计数（entry_count）及节点间的动态调度信息。

2.PendingOperatorQueue*pending_queue：算子挂起队列指针，用于维护尚未满足就绪条件但已部分组装完成的运行时上下文。

3.ReadyOperatorQueue*ready_queue：算子就绪队列指针，用于接收新识别的就绪节点，等待执行分派。

4.MemoryManager*mem_mgr：内存管理句柄，负责释放或回收不再被引用的张量存储空间及工作区块。

5.ScheduleTable*sched_table：静态调度表，提供后继节点索引及算子间依赖关系描述，用于驱动入度更新与就绪判定。

其伪代码流程如图 3-57 所示。

3.4 轻量化并发管理层

3.4.1 基本思路

轻量化并发管理层是本系统的核心组成模块之一，主要负责在多核及异构计算资源环境下，对可调度实体的并发执行进行统一协调与受控管理。该层以调度引擎层输出的就绪可调度实体为输入，在不引入复杂运行期调度策略的前提下，通过轻量化的任务分发、执行绑定与并行度控制机制，实现对 CPU、GPU、NPU 等计算资源的高效利用，并确保推理执行过程在时序与资源使用上的稳定性与可预测性。

如图 3-58 所示，轻量化并发管理层位于调度引擎层与底层计算资源之间，承担着将就绪可调度实体高效映射为并发执行任务的关键职责。该层以上游调度引擎层输出的就绪算子队列为输入，通过同步任务分发机制将可调度实体转化为具体的 Worker 执行任务，并统一纳入 Worker 任务池进行管理。在执行过程中，不同 Worker 根据任务类型与绑定策略，分别触发算子级或子图级的并行执行，并将计算任务分派至 CPU、GPU 或 NPU 等异构计算资源上完成。与此同时，Worker 内部遵循统一的任务工作流程，对

任务获取、执行与完成状态进行受控管理，从而保证并发执行过程在时序与资源使用上的稳定性与可预测性。基于上述设计，轻量化并发管理层的具体实现将从同步任务分发与协调机制、算子级并行执行机制以及子图级并行执行机制三个方面展开说明。

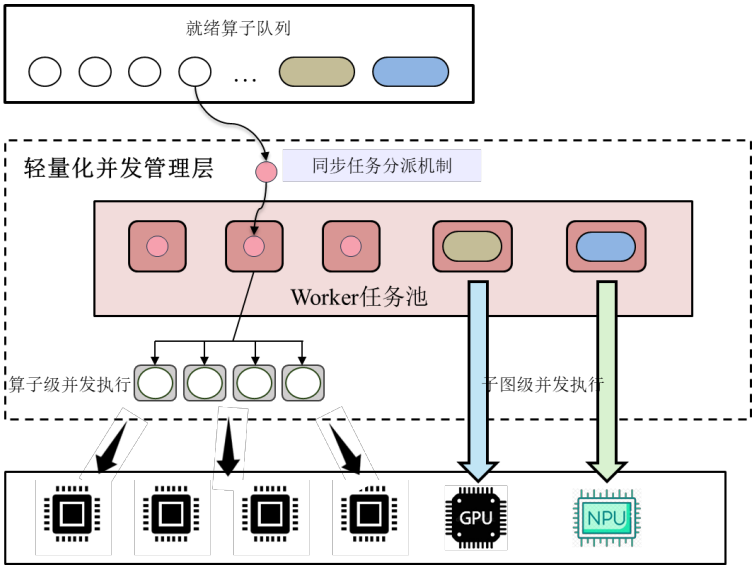


图 3-58　轻量化并发管理层的基本思路

1. 同步任务分发与执行协调。并发管理层从就绪算子队列中获取已满足执行条件的可调度实体，并将其分发至对应的 Worker 线程或执行单元。任务分发过程以事件驱动方式进行，避免轮询或复杂调度逻辑带来的额外开销。在任务执行过程中，并发管理层负责协调各 Worker 的执行节奏与状态回收，确保执行完成事件能够被及时反馈至调度引擎层，从而形成稳定的就绪分发—执行完成—状态回写闭环。

2. 算子级并行执行控制。在算子级粒度上，并发管理层支持多个彼此不存在依赖关系的可调度实体并行执行。该并行性完全基于调度引擎层已完成的依赖判定结果，不在运行期进行额外的依赖分析，从而降低并发控制复杂度。通过对 Worker 数量、并发上限及资源占用情况的受控管理，并发管理层在提升执行吞吐率的同时，有效避免资源争用与执行抖动，保证系统行为的可分析性。

3. 子图级并行执行支持。针对编译阶段已完成子图划分的模型，并发管理层进一步支持子图级并行执行机制。在满足跨子图依赖约束的前提下，不同子图可作为较粗粒度的可调度实体并行执行，并优先绑定至对应的异构计算资源（如 GPU 或 NPU）。该机制能够减少频繁的算子级调度与上下文切换开销，提升异构资源利用效率，适用于高吞吐或流水化推理场景。

通过上述机制，轻量化并发管理层在保持执行依赖正确性与运行时行为确定性的前提下，实现了对多核及异构计算资源的高效并发利用。该层与调度引擎层协同工作，将复杂调度决策前移并固化于初始化阶段，在运行期仅承担轻量、可控的并发执行管理职责，为整个轻量化异构推理运行时框架提供了稳定可靠的执行支撑。

3.4.2 同步任务分发与协调机制

并发管理层面向多核执行场景设计，其核心目标是在保证系统执行确定性与实时性的前提下，最大化底层硬件资源的并行利用效率。该层以调度引擎层维护的全局就绪队列为输入，通过一组协同运行的 Worker 任务

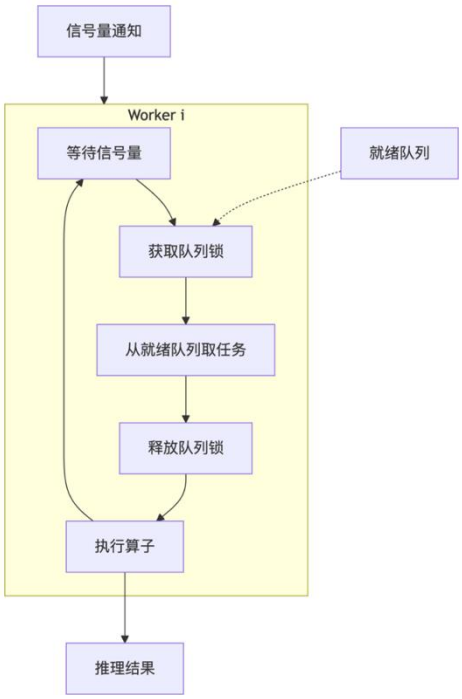


图 3-59 Worker 任务工作流程

对可调度实体进行并行调度与执行，其输出是一个在多核计算资源上高效运行、时序一致且行为可预测的推理执行过程。

并发管理层主要承担以下两项核心工作内容。第一，在系统初始化阶段创建 Worker 任务池，生成若干优先级一致、功能对等的静态 Worker 任务。每个 Worker 均封装统一的执行逻辑：等待工作信号、获取就绪任务并触发算子执行，从而在多核环境下实现算子级任务的并行调度与协同执行。通过避免任务优先级差异与复杂运行期调度决策，该机制有助于实现计算核心之间的负载均衡，并维持整体执行行为的确定性。

第二，并发管理层采用基于计数信号量与自旋锁的同步仲裁机制，实现多核竞争环境下任务分发与队列访问的高效协调。具体而言，计数信号量作为任务就绪通知手段，用于驱动 Worker 的阻塞与唤醒；自旋锁用于保护全局就绪队列的并发访问，确保临界操作的原子性。如图 3-59 所示，该机制构成一个高效的生产者–消费者模型：调度引擎层在识别到新的就绪可调度实体后，将其加入就绪队列并递增信号量计数；Worker 任务作为消费者阻塞等待信号量通知，被唤醒后竞争队列锁并快速获取任务，随后立即释放锁以缩短临界区驻留时间。算子执行过程完全在锁外独立完成，执行结束后，Worker 再次检测信号量状态以尝试获取新任务；若队列为空，则自动进入阻塞等待。通过上述机制，实现了低延迟、高并发且可持续流动的任务流水线执行模式。

3.4.3 算子级并行执行机制

算子级并行执行机制是轻量化并发管理层提升推理执行吞吐率的关键手段之一，其目标是在不破坏执行确定性与依赖约束的前提下，充分挖掘可调度实体之间以及算子内部的潜在并行性。针对不同运行平台与执行环境的能力差异，本系统在算子级并行执行层面明确区分并支持两种不同层次的并行形式，以实现对并行行为的可控管理。

第一层并行体现在多个可调度实体的并行执行。当调度引擎层判定多个算子之间不存在数据依赖关系时，这些算子将以独立的可调度实体形式进入就绪队列，并由并发管理层分派至不同的 Worker 线程或执行单元并行触发执行。该层并行完全建立在调度引擎层已固化的依赖判定结果之上，并发管理层不在运行期进行额外的依赖分析或调度决策，从而在保证执行正确性的同时，降低运行期调度复杂度并维持系统行为的确定性。

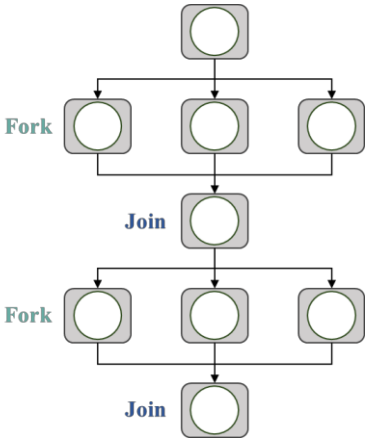


图 3-60 算子内部并行加速示意图（Fork-Join 机制）

第二层并行体现在单个算子执行过程中的内部并行加速。在通用操作系统平台上，如图 3-60 所示，部分算子实现可通过 OpenMP 等并行编程机制，在算子内部利用多核 CPU 资源进行并行计算，以进一步缩短单次算子执行时间。然而，在嵌入式操作系统平台上，受限于运行环境与系统支持能力，OpenMP 等通用并行运行时通常不可用或不被推荐使用。因此，本系统将算子内部并行视为一种可选且平台相关的优化手段，其启用与否在编译阶段或平台配置阶段即被明确约束，而不作为运行期并发管理层的默认并行策略。通过对算子级并行执行的分层建模与显式区分，系统能够在不同平台条件下灵活选择合适的并行方式，在嵌入式场景中优先依赖算子间并行，在通用平台上结合算子内部并行优化，从而在性能与可控性之间取得平衡。

综上所述，算子级并行执行机制通过对并行层次的明确区分，实现了对推理执行并行性的可控表达：在运行时层面，系统优先通过多个可调度实体的并行调度来挖掘算子间的天然并行性；在算子实现层面，则将算子内部并行加速视为平台相关的可选优化手段，并在编译或配置阶段进行显式约束。通过这种分层建模与职责隔离的设计，轻量化并发管理层既能够在嵌入式操作系统平台上避免对通用并行运行时的依赖，保证执行行为的确定性与可分析性，又能够在通用操作系统环境中充分利用算子内部并行能力以提升整体性能，从而在不同平台条件下实现性能与可控性之间的平衡。

3.4.4 子图级并行执行机制

子图级并行执行机制是轻量化并发管理层在异构计算平台环境下提升整体执行效率的重要手段，其核心目标是在保证跨设备依赖关系正确性的前提下，实现面向不同计算设备的推理子图并行执行。由于模型在编译阶段已根据目标硬件特性完成子图划分与算子映射，不同子图在语义上对应不同的计算资源（如 CPU、GPU、NPU），因此运行期的子图级并行执行本质上体现为多设备之间的并行协同执行。

在运行时，当调度引擎层判定多个子图之间不存在未满足的跨子图依赖关系时，并发管理层可将这些子图作为较粗粒度的可调度实体并行触发执行。每个子图整体绑定至其对应的计算设备，由相应的执行后端负责具体算子或融合子图的执行过程。并发管理层在该过程中不介入子图内部的执行细节，而仅负责子图级任务的分发、执行状态跟踪以及完成事件的回收，从而避免在运行期引入跨设备的复杂调度逻辑。通过子图级并行执行机制，不同计算设备能够在同一推理过程中并行工作，充分发挥异构平台的整体算力优势。同时，由于子图划分与设备映射均在编译阶段完成，运行期只需遵循已固化的执行边界与依赖约束，该机制在提升吞吐率的同时，

仍然保持了执行行为的确定性与可分析性，特别适用于多设备协同推理与流水化执行场景。

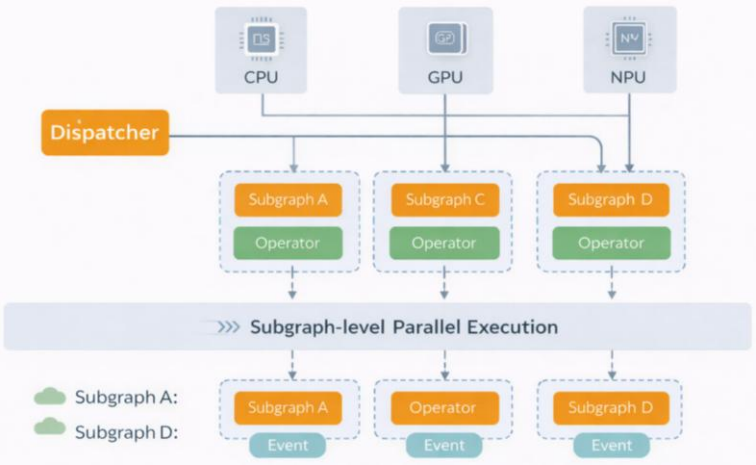


图 3-61 子图级并行执行示意图

综上所述，子图级并行执行机制通过将编译期完成设备划分的推理子图作为运行期调度与并发执行的基本单位，实现了多设备环境下的高效协同执行。该机制与算子级并行执行形成互补：前者侧重于跨设备的粗粒度并行，后者侧重于同一设备内的细粒度并行控制。通过在轻量化并发管理层中对两类并行机制进行清晰分层与职责隔离，系统能够在嵌入式与通用操作系统平台上灵活适配不同硬件条件，在保证执行正确性与确定性的同时，最大化异构计算资源的整体利用效率。

3.4.5 方案实现

并发管理层在系统初始化阶段根据可用 CPU 核心数创建同一优先级的 Worker 任务（WorkerTask），并使其初始状态阻塞于共享的计数信号量之上。调度引擎层作为任务生产者，在识别到新的就绪可调度实体后，将其对应执行任务入队至全局就绪队列，并递增计数信号量以触发 Worker 唤醒。Worker 作为任务消费者，被唤醒后竞争就绪队列锁获取任务，随后立即释放锁并在锁外独立执行算子；执行完成后，Worker 将结果上报至完成队列，并重新进入等待状态。通过该机制，系统构建了一个低延迟、事件

驱动的生产者-消费者执行流水线，在提升多核利用率的同时，保持了执行行为的确定性与可分析性。

并发管理层的核心数据结构为 WorkerPool，用于统一管理 Worker 任务的创建、调度与生命周期状态。WorkerPool 负责维护 Worker 的标识与运行状态，提供集中化的启动、停止与控制接口，并作为并发执行的组织载体，与调度引擎层协同完成任务分发与执行回收。

（1）WorkerPool

功能概述：WorkerPool 是并发管理层中的核心管理单元，其主要职责是在系统初始化阶段统一创建并配置 Worker 任务（通常与可用 CPU 核心数一致），并协调其在运行期围绕共享计数信号量与就绪队列进行协同工作。WorkerPool 本身不参与调度决策，仅负责为 Worker 提供一致的执行环境与受控的并发访问机制，从而在多核环境下实现负载均衡与确定性执行。

核心项说明：1.num_workers：Worker 数量（通常等于可用 CPU 核数）

2.worker_ids[]：Worker 任务 ID 列表（用于定位与控制）。

3.worker_state[]：任务状态（IDLE/RUNNING/SLEEPING/STOPPING 等）。

4.priority/affinity[]：统一优先级与可选核亲和配置。

5.sem_task_available：共享计数信号量句柄（任务到达通知）。

6.ready_queue/complete_queue 引用：就绪与完成队列的访问句柄。

7.rq_lock：全局就绪队列自旋锁引用（保障并发访问原子性）。

8.worker_entry：Worker 统一入口函数指针（等待→取任务→执行→上报循环）。

9.policy_flags：运行策略位（如是否允许自旋等待、回退策略等）。

10.metrics：运行指标（已派发/完成计数、平均等待时间、最后心跳）。

（2）WorkerPoolInit()（Worker 工作池初始化函数）

功能概述：WorkerPoolInit()在系统启动阶段完成 Worker 工作池的创建与初始化，包括确定 Worker 数量及其优先级/亲和策略，构建共享计数信号量与就绪队列锁，并创建并启动各 Worker 任务。Worker 启动后立即进入阻塞等待状态，直至调度引擎层投放首批就绪任务。该函数同时完成就绪队列、完成队列与 Worker 内部状态的绑定与初始化，为后续并发执行提供稳定的运行基础，其伪代码工作流程如图 3-62 所示：

1. **创建 Worker 池**:
   - `pool = newWorkerPool()`：创建一个新的工作池（`WorkerPool`）。
2. **配置工作池参数**:
   - `pool.num_workers = min(max(1, cfg.requested_workers), cpu_count())`：根据配置的请求数量（`cfg.requested_workers`）和系统的 CPU 核心数（`cpu_count()`）计算实际创建的工作线程数。确保至少有 1 个工作线程。
   - `pool.priority = cfg.priority`：设置工作池的优先级为配置中指定的优先级。
   - `pool.policy = cfg.policy_flags`：设置工作池的调度策略。
   - `pool.sem = semaphore_create(0)`：创建一个初始计数为 0 的共享信号量（用于同步工作线程）。
   - `pool.rq_lock = spinlock_create()`：创建一个自旋锁，用于保护就绪队列的访问。
   - `pool.ready_q = ready_queue`：将就绪队列设置为函数输入的 `ready_queue`。
   - `pool.pending_q = pending_queue`：将挂起队列设置为函数输入的 `pending_queue`。
3. **初始化工作池的每个线程状态**:
   - 使用一个循环，遍历每个工作线程:
     - `pool.state[i] = IDLE`：将工作池中的每个线程的状态初始化为 `IDLE`（空闲状态）。
4. **创建并启动工作线程**:
   - 在循环中，创建并启动每个工作线程:
     - `tid = task_create("worker_" + i, WorkerEntry, (pool, i), priority=pool.priority, stack=cfg.stack_size)`：为每个工作线程创建一个新的任务，任务名为 `"worker_" + i`，并将工作池及线程索引作为参数传递给 `WorkerEntry` 函数，设置任务的优先级和堆栈大小。
     - `if tid == INVALID: return NULL`：如果任务创建失败（`tid` 为无效值），则返回 `NULL`，表示初始化失败。
     - `if cfg.affinity_enabled: task_set_affinity(tid, cfg.affinity[i])`：如果启用了 CPU 亲和性配置（`cfg.affinity_enabled`），则设置该线程的 CPU 亲和性，以确保线程在指定的 CPU 核心上运行。
     - `pool.id[i] = tid`：将工作线程的 ID 存储在工作池的 `id` 数组中。
     - `task_start(tid)`：启动任务，使其开始执行，并阻塞在信号量上，直到其他部分唤醒该任务。
5. **返回工作池**:
   - `return pool`：初始化完成后，返回创建的工作池。

图 3-62　Worker 工作池初始化函数伪代码工作流程

3.WorkerRoutine()（Worker 工作函数）

功能概述：WorkerRoutine()是并发管理层中单个 Worker 任务的常驻执行循环，其职责是在运行期以事件驱动方式持续获取并执行可调度实体对应的运行时上下文。该函数在共享计数信号量上阻塞等待任务到达通知，被唤醒后以最小临界区访问全局就绪队列，获取一个可执行的运行时上下文；随后立即释放队列锁，并在锁外独立触发算子内核的执行。算子执行完成后，Worker 将对应的运行时上下文推送至完成队列，用于通知调度引擎层推进依赖状态与资源回收，并继续进入下一轮等待。该执行循环同时支持系统关停与空唤醒等异常场景的处理，确保在多核并发环境下实现低开销、可预测且具备确定性行为的任务执行，其伪代码工作流程如图所示：

1. **初始化线程状态**:
   - `set_state(wid, SLEEPING)`：将工作线程（`wid`）的状态设置为 `SLEEPING`（休眠状态），表示线程开始时没有任务可执行，处于等待状态。
2. **进入主循环**:
   - `while not pool.shutdown:`：进入一个循环，持续运行直到工作池关闭（`pool.shutdown` 被设置为 `True`）。
3. **等待任务通知**:
   - `semaphore_wait(pool.sem)`：等待信号量的通知，意味着线程会在此阻塞，直到有任务通知它执行任务。
4. **支持优雅关停**:
   - `if pool.shutdown: break`：检查工作池是否请求关闭。如果请求关闭，则退出循环，停止工作线程。
5. **进入临界区，获取任务**:
   - `lock(pool.rq_lock)`：获取就绪队列的自旋锁，进入临界区，确保对队列的操作是线程安全的。
   - `ctx = pool.ready_q.pop()`：从就绪队列中弹出一个任务（`ctx`），准备执行。
   - `unlock(pool.rq_lock)`：释放锁，退出临界区。这样可以避免任务在执行过程中被其他线程干扰。
6. **空唤醒/竞争失败的处理**:
   - `if ctx == NULL: continue`：如果没有任务可执行（`ctx == NULL`），则表示空唤醒或任务竞争失败，跳过当前循环并继续等待下一个任务。
7. **设置线程为运行状态并执行任务**:
   - `set_state(wid, RUNNING)`：将工作线程的状态设置为 `RUNNING`（运行状态），表示线程正在处理任务。
   - `rc = ctx.kernel(ctx.inputs, ctx.in_count, ctx.outputs, ctx.out_count, ctx.weights)`：调用任务的执行函数（`ctx.kernel`），传递输入、输出和权重等参数，执行算子的工作。执行结果保存在 `rc` 中。
8. **上报任务完成**:
   - `ctx.exec_rc = rc`：将任务执行的返回码（`rc`）保存到任务上下文中。
   - `pool.complete_q.push(ctx)`：将任务上下文（`ctx`）推送到完成队列中，表示任务已完成。
9. **准备下一轮任务**:
   - `set_state(wid, SLEEPING)`：将工作线程的状态重新设置为 `SLEEPING`，准备进入下一轮任务的等待。
10. **线程停止**:
    - `set_state(wid, STOPPING)`：如果循环结束（如池关闭），将工作线程的状态设置为 `STOPPING`，表示线程即将终止。

图 3-63 Worker 工作函数伪代码工作流程

(三) 创新点

1.面向存储层次结构的编译优化

针对嵌入式异构平台中存储层次约束与带宽瓶颈对端到端推理性能的主导影响，提出存储层次结构导向的编译期约束与优化技术，将存储访问对齐、冲突约束与数据复用收益等因素纳入统一的可判定约束空间，在编译阶段给出满足约束的内核候选与执行形态，降低无效访存与搬运开销，提升端到端推理效率并降低时延抖动。

2.面向异构计算资源协同的规则驱动编译

针对国产异构 SoC 部署中后端能力差异难以统一刻画、子图划分缺少可判定规则与可比较代价约束，导致映射边界碎片化、跨后端交互开销难受控、异构协同效率偏低的问题，提出面向异构计算资源协同的规则驱动编译技术。该技术以协同可行性与端到端效率为牵引，建立统一的后端能力与代价描述空间，将算子与复合模式支持条件、布局与数据类型约束、资源边界与交互代价纳入可判定的规则集合，并在规则约束下给出稳定的映射与边界选择，形成统一的多后端协同执行形态；同时生成面向端侧部署的硬件特定 C 代码与一致的调用接口，最终提升异构计算资源的利用效率和端到端推理效率。

3.面向实时嵌入式操作系统的多核确定性并行编译

针对多核推理对通用运行时并行库与运行期调度机制的依赖易引入不确定性，难以满足 RTEMS 对确定性与可验证性的要求的问题，提出面向 RTEMS 的多核确定性并行编译技术。该技术以并行行为可预测为牵引，将算子内并行的任务划分与同步约束在编译阶段确定下来，生成可在 RTEMS 上直接集成的纯 C 并行内核与静态并行回放入口，使端侧多核执行能够复现既定并行行为而不依赖 OpenMP 等复杂运行时。最终降低推理时延抖动并提升多核可用性。