Aladdin 工作原理

February 5, 2018

1 Introduction

Aladdin 工作的核心原理是通过数据依赖图 (Data Dependency Graph) 表示加速器的行为。Graph 中每个节点表示预定义的操作指令，边对应存储和寄存器依赖。硬件架构的各种设计调整均通过图结构的变换、节点属性的修改等方法体现。在此基础上，根据用户定义的硬件资源约束，使用图遍历算法 (BFS)schedule 该 graph，计算加速器的执行情况。最后，通过合适的 binding 策略将每个操作与实际的硬件资源对应起来，结合基本硬件资源的功耗模型，计算整个加速器的功耗。

2 Graph Construction

目标应用使用 c/c++ 描述，Aladdin 依据代码执行的动态指令流构建动态数据依赖图。Aladdin 使用 LLVM IR，因为 LLVM IR 是机器无关的，能更好体现算法本身的行为，避免机器相关的指令如寄存器分配等的影响。图中的每个节点表示 LLVM IR 中的指令, 而边表示节点之间的依赖包括存储/寄存器依赖等。

首先使用 LLVM 的编译器前端 clang 将 c 代码翻译成 LLVM 的 IR 表示, 然后通过在 IR 中添加指令获取函数 (profiling functions) 来定位和提取 trace。但这样修改后的 IR 无法真正在机器上运行, 因此还需要一个 just-in-time 编译器/解释器来执行修改后的 IR 并生成运行时指令流。指令流中包括了指令的各项信息如指令 ID、操作码、操作数、虚拟寄存器 ID、存储地址 (Load/store 指令)、代码块 ID 等。(instruction IDs, opcodes, operands, virtual register IDs, memory addresses (for load/store instructions) and basic block IDs)

3 Graph Transformation

3.1 Graph Optimiztion

最初生成的 DDDG 还包含一些辅助节点和依赖边, 与硬件行为无关。例如为了解决 SSA 形式的变量赋值问题而添加的 PHI 节点和循环模块的索引变量依赖等。为了使 DDDG 主要表达加速器的硬件行为, 无关节点会被删除或将 latency 设置为 0, 无关的边会被移除，由此可获得完全体现实际计算行为的 DDDG。

3.2 Hardware Transformation

3.2.1 Array

代码中的 array(数据) 被映射为对应的存储资源，包括 Memory(Scratchpad) 和 Register，访问不同的存储资源有不同的能耗和 latency。

如果 array 被映射为 Memory，则在 schedule 时其 load 和 store 都需要作为一个独立 node 考虑，其访问有固定的 latency(scratchpad 访问地址是已知的)，同时要考虑占用的端口资源 (port)。用户可以设置 Memory 的尺寸、位宽、端口数、Partition 策略等。

当 array 被映射为 Register 时，处理情况与 Memory 类似，但其 latency 为 0，即可以在同一周期中访问到其中数据。

3.2.2 Mem Access Optimizaton

根据指令 trace 中的地址和对 memory 的设置，指令流中的数据可以被映射到相应的地址中，并计算 graph 中的 load/stroe 操作的访存地址，根据这些信息可以对访存操作进行优化，移除冗余的 laod/store 指令，主要有以下三个方面：

Shared load: 如果两个 load 指令访问的是同一块地址且两个 load 间没有向该地址写入的 store 指令，那么 load 指令可以合并为一个。

Store buffer: 如果一个 store 节点是一个访问相同地址的 load 节点的直接前驱，则该 load 指令可以被移除，即使用数据 buffer 避免了写回再读出的浪费。

Repeated store: 如果两个 store 指令访问同一块地址且两者间没有数据依赖，则保留后一个 store 即可。

3.2.3 Loop

当循环中的计算没有 Loop-Carried Dependeces 时，通过将 graph 中循环索引变量的依赖、顺序执行时跳转指令的依赖移除，循环可以被展开，多个循环体内部计算可以并行进行。

3.2.4 Function

Graph 中每个节点的属性中包含其属于的函数 ID，在 schedule 和 binding 的时候可以依据此信息区分硬件中的不同模块。

3.2.5 Pipeline

Loop 和 funciton 调用可以被 pipelined 执行，Pipeline 的实现是通过将 pre_block 的 branch 节点 (表示该 loop 执行完成) 与 next_block 的执行节点之间的 control edge(即 pre 中操作全部执行完才开始执行 next) 移到 pre_block 的第一个非孤立节点和 next_block 的执行节点之间 (pre 中的第一个操作执行完即开始执行 next 中的操作)，由此两个 Loop 中的操作可以依次流水执行，每个 node 占用一个 pipeline stage。

4 Schedule

Schedule 负责将 Graph 中的每个节点分配至相应的周期中，Aladdin 中主要使用了两个策略，一个是 ASAP(As-Soon-As-Possible)，另一个是 Resouce-Constrained Scheduling。

依据 ASAP 策略，当 Graph 中一个节点的所有前驱节点 (predecessors) 执行完成时立即执行该节点。

实际中还要考虑硬件资源的约束，即对 ASAP 的 schedule 结果再做修正。主要有以下几个方面的考虑：

1. 时序约束：每个 cycle schedule 的节点的最大 latency 不能超过用户设定的 cycle time，否则将该 node schedule 到下一个周期。

2. 资源约束：每个 scheduled node 需要有可用的硬件资源。Aladdin 在做 schedule 过程中会记录计算资源和存储资源的使用情况，当没有可用的计算单元或访存端口时，操作需要等待。

5 Binding

Schedule 将 graph 中的每个 node 分配至各个 cycle，Binding 则负责将每个操作与实际的硬件资源对应起来，主要考虑的是时间上的复用。由于复用需要占用 MUX 资源，Aladdin 中对复杂的 function unit 如乘法器等进行复用，简单的 unit 如加法器、移位器等不复用。

6 Dumpstats

通过 schedule 和 binding，加速器执行中每个 cycle 的行为以及总的资源使用情况已知，结合每个操作对应的功耗可计算完整的功耗。功耗模型中考虑了不同 latency 下的动态功耗和静态功耗。