计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2022,31(11):130−138 [doi: 10.15888/j.cnki.csa.008780]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于 HXDSP 的 OpenCL 运行时任务调度①

顾经纬[1], 宁成明[1], 郑启龙[1,2]

[1](中国科学技术大学 计算机科学与技术学院, 合肥 230027)
[2](中国科学技术大学 高性能计算安徽省重点实验室, 合肥 230027)
通信作者: 郑启龙, E-mail: QLZheng@ustc.edu.cn

摘　要: OpenCL 是一种开源免费的异构计算框架, 被各类架构处理器广泛采用. HXDSP 是中国电子科技集团公司第 38 研究所自主研发的国产高性能 DSP 芯片. 为了解决 HXDSP 异构计算平台调度困难和硬件利用不充分, 本文针对 OpenCL 运行时任务调度系统展开研究, 设计了 OpenCL 运行时期间的任务图自动化提取方法, 并结合 HXDSP 硬件特性和 OpenCL 执行模型特性对经典的静态调度算法 HEFT 进行改进, 提出了一种异构双粒度最早完成时间优先调度算法 HDGEFT, 并在 HXDSP 异构计算平台上设计实验验证算法. 实验结果表明经过特殊设计的调度算法在执行效率上有明显优势.

关键词: OpenCL; 异构计算; 任务调度; HXDSP; 内核

引用格式: 顾经纬,宁成明,郑启龙.基于 HXDSP 的 OpenCL 运行时任务调度.计算机系统应用,2022,31(11):130–138. http://www.c-s-a.org.cn/1003-3254/8780.html

Task Scheduling of OpenCL During Operation Based on HXDSP

GU Jing-Wei[1], NING Cheng-Ming[1], ZHENG Qi-Long[1,2]

[1](School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China)
[2](Key Laboratory of High Performance Computing of Anhui Province, University of Science and Technology of China, Hefei 230027, China)

Abstract: OpenCL is an open source and free heterogeneous computing framework, which is widely used in architecture processors. HXDSP is a domestic DSP chip independently developed by the 38th Research Institute of China Electronic Technology Corporation. To solve the scheduling difficulties and insufficient hardware utilization of the HXDSP heterogeneous computing platform, this work studies the task scheduling system of OpenCL during operation. The automatic task graph extraction method during the operation of OpenCL is designed, and the classic static scheduling algorithm HEFT is improved by the combination of the hardware characteristics of HXDSP and the execution model characteristics of OpenCL. Thus, a heterogeneous dual-granularity earliest finish time (HDGEFT) scheduling algorithm is proposed, and experiments are designed on the HXDSP heterogeneous computing platform for verification. The experimental results reveal that the specially designed scheduling algorithm has great advantages in execution efficiency.

Key words: OpenCL; heterogeneous computing; task scheduling; HXDSP; kernel

1　引言

实践证明, CPU 架构在面对海量数据和特殊计算时存在性能瓶颈, 依靠增加主频或堆积核心数量会使硬件结构复杂, 计算功耗增加, 且不能解决性能瓶颈[1]. 一种解决方案是搭建异构计算系统, 异构计算系统定义为由不同指令集和基础架构计算资源构成的系统, 计算资源包含 CPU, GPU, DSP, FPGA 等芯片. 异构系统协调硬件设备间计算, 从而提升系统的性能[2].

① 基金项目: 国家核高基重大专项 (2012ZX01034-001-001)
收稿时间: 2022-02-13; 修改时间: 2022-03-14; 采用时间: 2022-04-02; csa 在线出版时间: 2022-07-07

HXDSP 是中国电子科技集团公司第 38 研究所研制 BWDSP[3,4] 系列处理器下的一款国产 DSP 处理器芯片. HXDSP 芯片通常搭配 CPU 芯片, 构成 HXDSP 异构系统. HXDSP 异构计算框架选取 OpenCL 作为载体.

OpenCL 是异构并行编程模型, 异构编程模型还有 CUDA, Merge[5], C++AMP[6], Lime[7]. OpenCL 将计算任务抽象为 3 个层次: 内核, 工作组, 工作项. 当研究对象为内核在设备上的分配策略时, 问题为异构系统下的任务调度问题. 该问题被证明是 NP-hard 问题[8], 可通过启发式或元启发式算法解决, 启发式调度算法包括: 基于优先级列表调度算法[9], 基于聚类调度算法[10], 基于任务复制调度算法[11]; 元启发式算法包括: 遗传算法[12], 粒子群优化算法[13], 蚁群优化算法[14]. 基于优先级列表的调度算法 HEFT[15] 被实践证明简单高效, 被选取为研究对象.

基于 OpenCL 执行模型特性的任务调度存在大量研究, 如文献 [16] 通过机器学习方式通过代码文件判断内核在 CPU 和 GPU 间的最优分配, 文献 [17] 通过内核代码特征, 选择 CPU 执行, GPU 执行或在两者之间划分任务的策略, 文献 [18] 在多 DSP 架构针对 OpenCL 工作组和工作项两个粒度设计了一种双层调度器, 文献 [19]研究了在多设备协同执行同一个内核的有效性.

现有 OpenCL 任务调度研究局限于开发平台闭源性质, 研究点侧重于内核中工作组粒度上任务分配, 或在内核粒度上的研究不同处理器间选择问题, 并未从不同粒度对计算任务进行规划. 内核间的任务调度符合传统任务调度模型, 而 OpenCL 内核内符合单指令多线程, 本文算法综合考虑两种粒度提出新的思路.

本文从特定的 HXDSP 硬件结构出发, 针对异构编程模型 OpenCL 展开运行时的自动化任务调度研究, 设计了一种通过任务队列在 OpenCL 运行时获取任务图的方法, 通过将内核分解与调度算法 HEFT 结合, 得到一种更加高效的静态任务调度算法. 本文组织如下: 第 1 节介绍异构计算中相关问题研究现状. 第 2 节介绍 HXDSP 硬件平台和 OpenCL 抽象模型. 第 3 节介绍在 OpenCL 应用中获取任务图方法. 第 4 节介绍在 HXDSP 异构系统中的 OpenCL 运行时任务调度算法设计. 第 5 节通过实验验证算法的可行性和有效性. 最后对本文工作进行总结和展望.

2 OpenCL 和 HXDSP 相关知识

2.1 HXDSP 结构

HXDSP1042 设备内部集成了 2 个 eC104 处理核心, 6 组 serdes 接口, 2 个 DDR 控制器, 1 个以太网接口, 一些低速外设接口. 在性能上, eC104 包含 4 个执行宏, 支持 SIMD 和 VLIW, 工作时的核心频率为 500 Mhz, 达到 30 GOPS 和 8 GFMACS 的运算能力. 在存储空间上, 程序空间和数据空间在物理上分离, 单核心含 32 KB 字指令 cache 及 1 536 KB 字数据存储器, 数据存储器被划分为 6 个 block, 每个 block 大小为 256 KB, 双内核共享 256 KB 字指令存储器. HXDSP 数据存储空间通过统一地址空间 NUMA 方式实现互访, 用户可访问程序存储空间主要包括共享指令存储器, 内核私有程序存储器, 加载核 ROM 存储空间和 DDR 空间.

研究对象 HXDSP 异构计算平台抽象为图 1 所示硬件结构, FPGA 板卡上搭载 1 张 DSP 加速板卡, DSP 加速板卡包含 4 个 HXDSP1042 芯片和 1 块 DDR 存储, FPGA 充当主设备, 也可视为 CPU, 运行主机端管理程序, 驱动 DSP 进行计算和 DDR 上数据的存取.

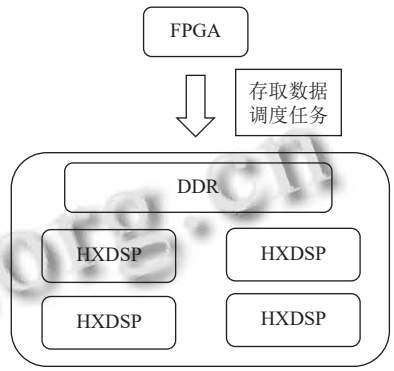


图 1 HXDSP 异构系统结构图

2.2 OpenCL 抽象模型

OpenCL 框架可分为 4 个模型: 平台模型, 执行模型, 存储模型, 编程模型.

OpenCL 执行模型分为两个模块: 主机端执行的管理程序, 设备端执行的内核. 内核是计算任务的描述, 通过 OpenCL C 语言进行编写. 内核代码在执行时会创建索引空间 NDRange, N 的维度可取{1, 2, 3}, 索引空间中的点有独特的全局 ID, 用于实例化工作项, 实现单指令多数据. 由于硬件限制, 索引空间中的工作项不能同时实例化, 工作项间被组织为工作组, 工作组是工作项实例化的集合. 在 HXDSP 异构系统中, 内核被映

射到 HXDSP1042 芯片上, 工作组被映射到芯片上的 eC104 处理核心集合, 工作项映射到单个处理核心.

OpenCL 存储模型将设备中的存储器抽象为 4 层结构. 全局内存: 索引空间中所有节点可读写; 全局常量: 索引空间中所有节点可读不可写; 本地内存: 工作组内的工作节点可读写, 对其他工作组不可见; 私有内存: 只属于工作节点. 软件层面的内存模型是建立在硬件存储系统之上, 在 HXDSP 硬件模型中, 芯片内部只有 1 536 KB 字数据存储器, 所以全局存储和全局常量可能不能全部加载到 DSP 内部, 全局存储和全局常量被映射到 DDR, 本地内存映射到数据存储器 block, 私有内存映射到 DSP 核心中的寄存器组.

3 OpenCL 任务图获取

OpenCL 规范明确了计算中涉及到的概念对象和操作对应的 API 接口, 框架具体行为由提供商依据规范实提供. OpenCL 规范不包含任务调度概念, 由用户管理全部计算流程, 复杂的过程导致用户承担巨大工作量, 且程序在并行性上往往存在优化空间. 文献 [20] 发现目前供应商不能发现顺序队列中内核任务级并行性, 对内核重新调度能实现更优的执行效果, 这也证明了 HXDSP 异构计算平台上 OpenCL 运行时任务调度系统的可行性和必要性.

实现 OpenCL 的运行时自动化任务调度抽象为两个关键问题: 在 OpenCL 应用程序中获取任务图, 节点映射到计算设备上的策略. 本节讨论问题 1: OpenCL 任务图建立的依据和方法, 这是进行任务调度必需的前驱条件, 第 4 节讨论问题 2.

3.1 OpenCL 任务同步机制

OpenCL 任务图的建立在对 OpenCL 同步机制的正确理解之上. OpenCL 的同步机制分为主机端同步和设备端同步. 主机端同步机制描述命令队列中各个命令之间的同步关系, 包含内核间的相关依赖关系. 设备端的同步机制描述工作组内部的同步关系, 内核内部的同步机制与任务图获取无关. 主机端的同步机制分为 3 种: 命令队列执行模式, 基于事件的同步方式, 显式的同步命令.

命令队列有两种执行模式: 顺序执行和乱序执行. 默认情况下, 命令队列中的命令是 FIFO 操作, 先发送的命令先执行, 在创建命令队列对象时, 选择参数属性 CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, 目标设备将不定序的执行其接受到的命令.

第 2 种方式是基于事件的同步方式, 每一个 OpenCL 命令在入队列时都可以关联一个事件对象以及一个等待事件列表, 前者代指当前命令执行完成与否, 可被用于其他等待事件列表, 后者描述了该命令需要等待的事件集合, 只有事件集合中的命令全部执行完毕, 当前命令才能执行.

第 3 种方式是通过显式同步命令进行同步. OpenCL 中的同步命令包括标记命令, 栅栏命令和等待命令. 标记命令是对一组事件的标签, 用来记录事件的状态. 栅栏命令入队之后, 后续入队的命令等待栅栏命令执行完成. 等待命令会暂停命令队列执行, 直到其等待的事件列表执行完成.

3.2 任务图构造方法

任务图通常通过有向无环图 DAG 表示, 形式化表示为 $G=\{T, E, W, C\}$, T 指任务节点集合, E 指节点间的有向边集合, E 中的边表示依赖关系, W 指任务在处理器上的执行开销, C 指节点间的通信开销, 当节点在同一个处理器上时, 通信开销为 0.

对于任务在处理器上的处理开销, 可通过测量一小部分工作组的执行开销推算完成内核需要的全部时间. 前沿研究的方法是性能建模[21], 例如通过深度神经网络对基本块在时钟周期级别进行预测[22], 通过随机森林预测内核在 GPU 上执行时间[23], 通过卷积神经网络预测嵌入式 GPU 上程序执行效率[24].

对于任务节点间的通信开销, 通过硬件 IO 通道传输速度和传输的数据量的比值进行估算.

对于节点间和节点与数据间的依赖关系, 可以通过在 OpenCL 运行时添加任务调度模块实现, 任务调度模块负责构造任务图和调度任务执行. 在 OpenCL 中, 用户对设备发送的命令请求通过命令队列传递, 命令队列相关接口 API 中接受命令队列对象, 数据对象, 内核对象, 事件对象等, 完成命令的提交. 通过在命令对象接口 API 下添加额外操作, 在 API 入口处完成对传入对象的关系分析, 并将有效信息提交至任务调度模块, 即可构造运行时的任务图信息.

任务节点之间的依赖关系通过两个部分进行确定, 通过 OpenCL 任务同步机制确定命令之间的控制依赖关系, 然后在控制依赖图的基础上分析任务内存对象之间的依赖获得任务图. 根据 OpenCL 任务同步机制, 命令之间的控制依赖分析应该从命令队列执行模式,

命令依赖的事件集和显式的同步点 3 个方面进行分析. 情况 1, 命令对象为顺序执行模式时, 后入队列的命令对象逻辑上依赖于先入队列的命令, 当命令队列被设置为无序状态时, 队列中的命令逻辑上是并行的. 情况 2, 一个命令在入队列时可以显式的关联一个事件列表, 命令对象和事件列表对象一起被传递给命令队列, 事件列表中的对象集合的命令被当前命令依赖. 情况 3, 当显式的同步命令被提交给命令队列时, 同步命令前的命令集合被同步点后入队的命令所依赖. 在命令队列接口下建立控制流图的过程如算法 1 所示.

算法 1. 命令队列构造控制流图

输入: command_queue
输出: control_flow_graph

```
1. if command_queue_type==inorder then
2.    for command in command_queue do
3.       add edge from prev(command) to cur(command)
4.    for command in command_queue do
5.       if command_type==kernel or
6.          command_type==transmission or
7.          command_type==wait then
8.          if wait_event_list!=NULL then
9.             for event in wait_event_list do
10.               command_waited=getCommand(event)
11.               add edge from command_waited to command
12.      if command_type==barrier or
13.         command_type==marker then
14.         if wait_event_list!=NULL then
15.            for event in wait_event_list do
16.               command_waited=getCommand(event)
17.               add edge from command_waited to command
18.         else
19.            command_waited=getExitCommandBefore(command)
20.            add edge from command_waited to command
```

由控制流图构造任务图还需要边的权值, 即明确内核间数据传输量. OpenCL 用户习惯使用顺序队列简化操作, 这种行为导致不必要的依赖关系被加入到控制流图中, 为此需要在控制依赖的基础上进行数据依赖分析, 分析被内核对象修改的参数, 结合内核对象间的依赖关系, 可确定内核对象间数据的交换量. 数据依赖分析对内核参数的读写属性进行确认, OpenCL 中参数的读写属性体现在内存对象的参数修饰符上, 通过以下 3 种情况确定命令对数据的访问方式.

1) 若内核程序的参数的地址空间修饰符为 CL_MEM_READ_ONLY, 则所有命令对该内存对象的访问权限为只读, 若内存对象的访问类型为 CL_MEM_WRITE_ONLY, 则所有命令对该内存对象的访问权限为只写.

2) 若内核程序的参数的地址空间修饰符为 CL_MEM_READ_WRITE, 此时命令对该内存对象既可读可写, 根据内核程序的参数修饰符进一步确定命令对该内存对象的访问权限: 如果参数修饰符为 read_only, 那么内核程序对该数据的访问权限为只读, 如果参数修饰符为 write_only, 那么内核程序对该数据的访问权限为只写, 如果参数修饰符为 read_write, 那么内核程序对该数据即可读可写.

3) 若内核程序的参数的地址空间修饰符为 constant, 那么内核程序对该数据的访问是只读的.

在控制流图的基础上, 根据内核对象对内存对象的访问权限可以确定内核任务间的数据依赖关系, 建立任务图的过程如算法 2 所示.

算法 2. 构造任务图

输入: control_flow_graph
输出: task_graph

```
1. for command in control_flow_graph do
2.    if command in control_flow_graph then
3.       for mem_obj in command_mo_list do
4.          if mem_obj_type==CL_MEM_READ_ONLY |
5.             Cl_MEM_COPY_HOST_PTR then
6.             input_command=createCommand(mem_obj)
7.             add edge from input_command to command
8.          if mem_obj_type==CL_MEM_READ_WRITE &&
9.             command_access_mo_type!=__write_only then
10.            for command_prev_type==kernel do
11.               if command_prev_type==kernel then
12.                  if mem_obj in command_prev_mo_list&&
13.                     command_prev_access_mo_type!=__read_only
14.                     then
15.                     add edge from command_prev to command
16.               else if command_prev_type==input then
17.                  if mem_obj==getMemObj(input) then
18.                     add edge from command_prev to command
19.                  else
20.                     traverse command of prev(command_prev)
```

矩阵分块乘加是矩阵乘法的加速实现, 将输入矩阵平分为 4 个子矩阵, 然后对 4 个子块分别进行矩阵乘法, 最后将各个子块运算结果合并. 使用 OpenCL 实现矩阵分块乘加, 并依据算法对矩阵分块乘进行任务图提取操作, 将结果进行可视化后得到如图 2 的计算任务图, 节点上的数字代表 HXDSP 执行时间开销, 边上的数字代表数据传输开销.

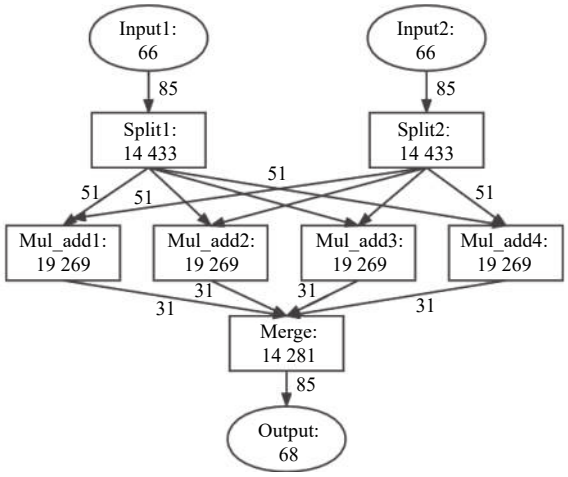


图 2 OpenCL 提取矩阵分块乘加任务图

4 HDGEFT 调度算法

本节依据 HXDSP 架构特征和 OpenCL 计算任务特性, 基于异构调度算法 HEFT, 结合 OpenCL 内核分解技术, 改进得到新的调度算法 HDGEFT (heterogeneous double-granularity earliest finish time) 异构双粒度最早完成时间优先调度算法.

4.1 HEFT 算法

异构调度算法 HEFT 包含两个步骤.

1) 任务选择阶段. 计算每个任务的向上权值 $rank_u$, 根据该值对任务节点降序排序, 得到任务分配的优先级序列, 任务优先级一经确定不再改变.

2) 任务分配阶段. 依据步骤 1) 得到的 $rank_u$ 序列进行任务分配, 计算任务 t_i 在不同处理器 p_j 上最早执行完成的时间 $EFT(t_i, p_j)$, 选择能最早完成任务的处理器并将任务分配到该处理器上.

HEFT 算法中涉及的相关公式及符号定义如下.

任务节点 t_i 的向上权值 $rank_u$ 定义为从出口节点向上遍历到任务节点 t_i 的最长距离, 计算公式如下:

$$rank_u(t_i)=\bar{w}_i+\max_{t_j\in succ(t_i)}\left(rank_u+c_{ij}\right) \tag{1}$$

其中, $\bar{w}_i$表示任务节点 i 在所有处理器上的平均处理时长; $succ(t_i)=\{t_j \mid e_{ij}\in E\}$代表任务节点 t_i 的后继节点集合. 若任务节点 t_i 的后继节点集合为空, 那么 t_i 也是出口节点, 记为 t_{exit}.

$EFT(t_i, p_j)$ 表示任务节点 t_i 在处理器 p_j 上最早完成时间, 计算公式如下:

$$EST\left(t_i,p_j\right)=\max\left(avail[j],\max_{t_m\in pred(t_i)}\left(AFT(t_m)+c_{m,i}\right)\right) \tag{2}$$

$$EFT\left(t_i,p_j\right)=w_{i,j}+EST\left(t_i,p_j\right) \tag{3}$$

其中, EST 表示任务节点 t_i 在处理器 p_j 上最早可执行时间; $avail[j]$ 表示处理器 p_j 的最早空闲时间, 即任务节点可以被处理器 p_j 处理的最早时间; $pred(t_i)=\{t_j \mid e_{ji}\in E\}$代表任务节点 t_i 的前驱节点集合, 若任务节点 t_i 的前驱节点集合为空, 则 t_i 也是入口节点, 记为 t_{entry}; $AFT(t_i)$ 表示任务节点 t_i 的实际完成时间; $c_{m,i}$ 表示当前任务节点 t_i 与前驱任务节点 t_m 的通讯开销; $w_{i,j}$ 表示任务节点 t_i 在处理器节点 p_j 上的执行开销.

4.2 HDGEFT 算法设计

本小节阐述算法思路的出发点和算法具体步骤.

1) 调度结果中会存在如图 3 情况, 某 DSP 处理器在处理一个任务时, 从任务开始执行到结束执行期间, 旁边的 DSP 处理器都处于空闲状态. 如任务 T_j 被调度在 0 号 DSP 处理器上执行, 而在 T_j 的执行时间段内, 其余 3 块 DSP 处理器并未执行计算任务, 理论上, 如果空闲 DSP 的计算能力能够正向辅助当前的计算任务, 那么运行时系统有理由采取优化设计去利用空闲设备进行计算.

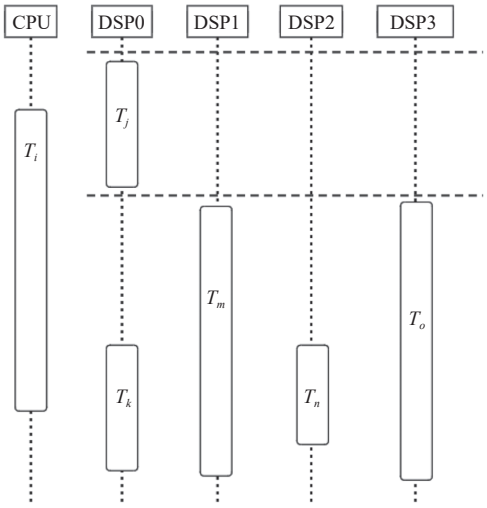


图 3 HXDSP 调度图

2) OpenCL 的执行模型特性使内核索引空间中的工作组是并行执行的, 工作组之间没有约束条件, 内核执行结果不依赖于工作组的执行顺序, 每一个输出数据由唯一的工作组产生, 工作组可以作为被调度的单位.

3) 内核执行过程可分为数据传输, 创建执行环境和内核任务执行 3 个过程, 观察 HXDSP 异构计算平台下的基本性能评测数据 (表 1)[25], 可以发现, 数据传输

时间对比内核任务执行时间有着巨大差距, 创建内核执行环境的开销较低, 这意味着数据传输开销不是性能瓶颈, 额外的计算核心可以产生正向作用.

表 1 基本性能评测表

向量长度	数据传输时间 (μs)	内核编译时间 (μs)	创建执行环境时间 (μs)	内核任务执行时间 (μs)
2 000	36	141	224	540
4 000	81	141	224	1 080
6 000	102	141	224	1 620
8 000	129	141	224	2 160

基于以上 3 点原因, 在依据 HEFT 调度算法在内核粒度上产生调度结果后, 可依据调度结果对符合条件的内核任务在工作组粒度上再进行一次任务调度, 新算法 HDGEFT 的具体步骤如下:

输入: 任务流图

输出: 任务在 DSPs 上的调度策略

1) 任务选择阶段.

2) 任务分配阶段. 前两步与 HEFT 算法相同.

3) 任务分解阶段. 在任务分配阶段得到分配结果后, 从前到后扫描, 寻找分配在 DSP 处理器上且满足任务执行期间存在空闲 DSP 计算资源的任务, 对该任务在空闲 DSP 集合上执行计算分解, 由于该操作使当前任务执行完成时间和当前处理器空闲时间这两个指标改变, 而任务在一个处理器上最早执行时间 EST 由处理器可用时间 *avail* 和依赖项执行结束时间 AFT 决定, 内核分解操作可能会对后续任务的最早执行完成时间 EFT 产生影响, 从而影响分配决策, 见式 (3).

因此对原分配序列应重新执行任务分配阶段和任务分解阶段, 迭代执行直至分配完最后一个任务.

4.3 工作组划分

文献 [19] 研究了 1CPU-2GPUs 架构上单 OpenCL 内核多设备协同计算相关问题, 通过方案设计和测试数据证明了使用多设备对单个内核计算实现加速的可行性, 其运行时设计为 DSPs 架构上的内核协同计算运行时设计提供了思路启发.

工作组划分方式需要满足每个工作组都被计算且仅计算一次, 划分工作组有简单可行的方式, 即从最高维度对工作组区间进行划分. 图 4 展示三维划分, 工作组索引具有连续性, 且整个工作空间被完全覆盖, 通过 4 个数据对象可以描述进行划分操作后各个 DSP 负责的工作空间: 数据的维度 work_dims, 各个维度上全局 ID 偏移量 global_work_offset, 各个维度上工作项的数量 global_work_size, 各个维度上一个工作组中工作项的数量 local_work_size. 不同的 DSP 设备从各自的全局偏移处开始计算, 在各个维度计算一定的步长, 每个 DSP 的工作区间不重叠, 所有 DSP 工作组的并集构成完整的任务空间.

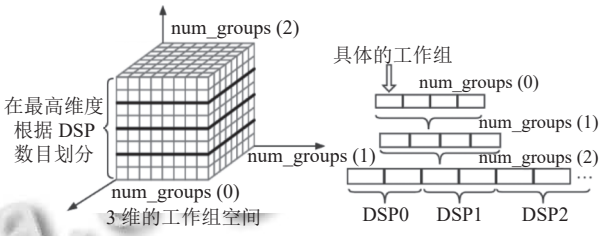


图 4 三维工作组划分

内核划分的另一个关键问题是内存一致性, 内核被分解到不同设备上执行计算, 输出数据存储在不同物理介质上, 最终输出由各部分整合得到. 由于工作组之间的独立性, 某一个输出数据由唯一确定工作组输出, 对不同设备输出数据做线性扫描可得到最终输出数据. 在 HXDSP 硬件系统上, DSP 设备的全局存储被映射到 DDR 存储上, DSP 可操作同一块物理介质, 可避免数据整合问题, 但会引入对 DDR 资源的竞争, 这种现象可通过工作组的 preload-poststore buffer 优化.

利用多个 DSP 设备协同计算一个内核任务的运行时流程如下: 编译内核环节, 使用目标编译器编译内核并将编译后机器码发送到参与计算的 DSP; 创建数据环节, 创建作为输入输出参数的数据对象, 将其发送到 DDR 上, 获得数据对象在 DDR 上的物理起始地址; 设置内核参数环节, 将参与计算的 DSPs 上内核参数地址设置为上一步中获得的物理地址, 不同 DSP 指向相同的 DDR 地址; 执行内核环节, 划分内核任务工作空间, 确定每个 DSP 设备全局偏移量和各个维度上工作项的工作项数目, 并驱动 DSP 的 RTOS 执行内核; 执行完成环节, 主机端监听设备信号, 在所有参与计算的 DSP 都向主机端发送子内核执行完成信号, 内核进入完成状态, 可进行下一步数据处理或后续任务调度.

5 实验

5.1 实验环境

本文相关研究及实验基于实验室前期搭建的 HXDSP 异构计算平台[21], 该平台实现参考一种开源的

面向 CPU 处理器的 OpenCL 实现-PoCL, 在其基础上面向 HXDSP 平台进行修改和调整, 其中充当从设备 HXDSP 有相关的开发工具链和模拟运行器, 可以完成 kernel 任务模拟计算. 实验中涉及的多 HXDSP 环境, 数据传输, 并行计算, 任务调度等环节都通过软件模拟实时系统实现了相应功能.

实验数据采用随机生成的任务图进行测试, 随机任务图的关键参数变量设置如下, 其中, V 指任务图中的节点数目, CCR 指计算通信比, out_degree 指任务图中的平均出度值, q 指参与计算的处理器集合, 固定为 1CPU 加 4HXDSP:

$SET_V = \{20, 40, 60, 80, 100, 120, 140, 160\}$

$SET_{\text{CCR}} = \{0.1, 0.2, 0.3\}$

$SET_{\text{out\_degree}} = \{1, 3, 5, 7, 9\}$

$SET_q = \{1, 4\}$

组合不同的参数获得实例, 在每一个不同的实例下生成 10 个随机任务图, 计算并对结果取算术平均值. 实验评价指标采用 *SLR* 和加速比 *Speedup*, 相关的指标和计算公式如下.

1) 调度总长度 *makespan*: 调度总长度等于出口节点的执行结束时间, 当有多个出口节点时, 调度总长度等于出口节点执行结束时间中的最大值, 定义如下:

$$makespan = \max\left(AFT\left(v_{\text{exit}}\right)\right) \tag{4}$$

2) *SLR* (schedule length ratio): 调度总长度与关键路径上的节点分别在最快处理器上执行时间之和的比. *SLR* 为一个无单位实数, *SLR* 值与算法性能成反比, 值越接近 1 效果越好, 定义如下:

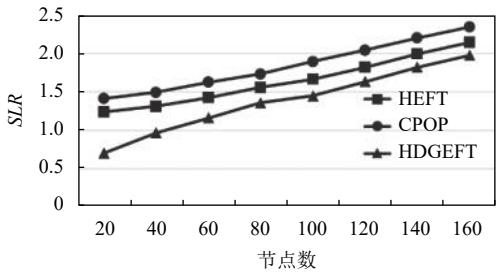
$$SLR = \frac{makespan}{\sum_{n_i \in CP_{\min}} \min_{p_j \in Q}\left(w_{i,j}\right)} \tag{5}$$

3) 加速比 *Speedup*: 在各个处理器上分别串行执行任务图中所有节点产生的执行开销之和, 取其中的最小值与调度总长度的比值. *Speedup* 为一个无单位实数, *Speedup* 与算法的效率成正比, *Speedup* 数值不会大于处理器数量, 定义如下:

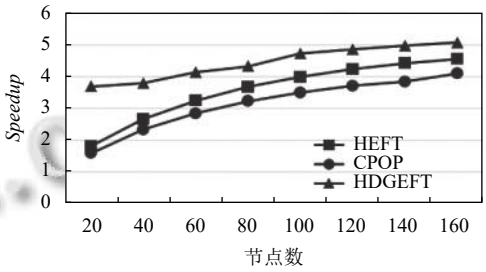
$$Speedup = \frac{\min_{p_j \in Q}\left(\sum_{n_i \in V} w_{i,j}\right)}{makespan} \tag{6}$$

5.2 实验结果分析

实验使用 HEFT, CPOP 算法进行对比, 使用 *SLR* 和加速比 *Speedup* 作为评价指标, 实验结果如图 5.



(a) 调度算法 *SLR* 对比



(b) 调度算法 *Speedup* 对比

图 5 调度算法实验结果

由图 5 实验数据可得, 改进的 HDGEFT 算法在 OpenCL 任务调度中有明显的效率优势. 在部分结果中, *SLR* 在 HDGEFT 优化算法下的值能小于 1, 这意味着对关键路径上满足条件的节点进行内核任务分解, 算法正向效率提升大于通信开销.

在节点数目确定的情况下, 与 HDGEFT 调度效果最相关的参数是图节点的平均出度值, 出度值越大, 节点的依赖关系越多, 串行性增强, 并行性减弱, DAG 图中的节点会获得更大深度, 这也导致依据 HEFT 算法产生的调度结果中出现更多的处理器空闲间隙, 为 HDGEFT 算法提供了更多可分解的节点. 在节点数设置为 80, 选取 *Speedup* 作为评价指标的情况下, 测试不同依赖数目下的优化效果, 实验结果如表 2, 观察可得, 算法优化效果与出度值有明显正相关, 即内核间的串行性越强, 应用越能从 HDGEFT 算法中获得收益.

表 2 出度值性能评测表

出度值	*Speedup*		差值比
	HEFT	HDGEFT	
1	4.47	4.75	0.06
3	3.97	4.50	0.13
5	3.16	3.93	0.24
10	2.10	2.69	0.28
15	1.66	2.58	0.55
20	1.40	2.61	0.86

使用 OpenCL 应用任务图进行测试: MapReduce, 矩阵分块乘法, 特征融合. 在 HXDSP 异构计算平台上,

应用在不同策略下的执行时间如图 6, 观察可得, HDGETF 算法有不同程度的优化效果.

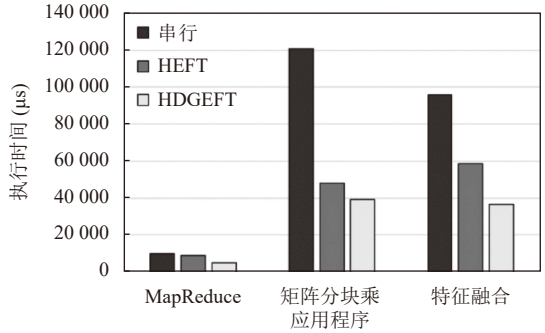


图 6 调度长度实验评估

测试应用中的矩阵分块乘法的任务图如图 2 所示, 该任务图使用 HDGEFT 算法调度后得到的调度图如图 7 所示, 任务图中的 split1 和 merge 内核任务满足内核任务分解条件, 被执行内核分解操作, 获得了仅靠 HEFT 任务调度无法获得的额外性能收益. split1 和 split2 任务是并发关系, 由于 split1 在分配列表中优先, 所以空闲 DSP 资源全部被分配给 split1, 但仅分解 split1 并不能加速 mul_add 任务, 因为 mul_add 也依赖于 split2, 观察可得, 将空闲的 DSP 资源平分给 split1 和 split2 是更优的行为, 可以使计算任务更快完成, 获得更大收益, 这为算法进一步改进提供了方向.

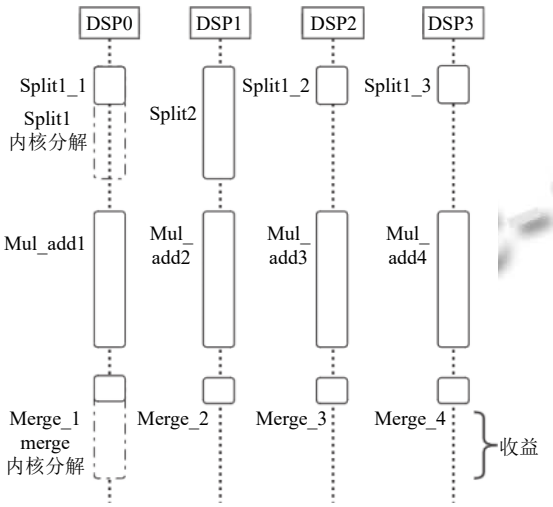


图 7 矩阵分块乘调度图

6 总结和展望

本文面向异构环境下的 OpenCL 运行时中任务调度问题, 给出一种获取 OpenCL 任务计算图的工程方法, 并结合 HXDSP 硬件架构, 基于 HEFT 算法提出了一种针对 OpenCL 任务进行优化调度的 HDGEFT 算法, 通过设计实验, 使用随机生成的任务图和具体的应用任务图验证了该算法的有效性.

在运行时进行 OpenCL 任务调度系统设计有许多改进方向, 下一步可添加机制完善运行时任务调度系统, 如其他类型处理器和 DSPs 之间对同一个内核的协同计算, 基于内核可分解特性的先验条件设计出更加完备的调度策略, 重叠计算操作和数据传输操作以获得更大的并发性, 这些措施可通过自动化实施提高异构系统的效能.

参考文献

1 Kwok YK, Ahmad I. Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Computing Surveys, 1999, 31(4): 406–471. [doi: 10.1145/344588.344618]

2 Shan A. Heterogeneous processing: A strategy for augmenting Moore's law. Linux Journal, 2006, 2006(142): 80–82.

3 邓文齐. 基于 BWDSP 的众核深度学习加速器的研究 [硕士学位论文]. 合肥: 中国科学技术大学, 2018.

4 蔡恒雨, 宁成明, 侯璇, 等. 国产 BWDSP 的并行通信接口设计. 小型微型计算机系统, 2021, 42(5): 897–904. [doi: 10.3969/j.issn.1000-1220.2021.05.001]

5 Piccardi M. Background subtraction techniques: A review. Proceedings of 2004 IEEE International Conference on Systems, Man and Cybernetics. The Hague: IEEE, 2004. 3099–3104.

6 Gregory K, Miller A. C++ AMP: Accelerated massive parallelism with Microsoft Visual C++. Microsoft Press, 2012.

7 Auerbach J, Bacon DF, Cheng P, *et al*. Lime: A Java-compatible and synthesizable language for heterogeneous architectures. ACM SIGPLAN Notices, 2010, 45(10): 89–108. [doi: 10.1145/1932682.1869469]

8 Ruan YL, Liu G, Li QH, *et al*. An efficient scheduling algorithm for dependent tasks. Proceedings of the 4th International Conference on Computer and Information Technology. Wuhan: IEEE, 2004. 456–461.

9 Abbasi SI, Kamal S, Gochoo M, *et al*. Affinity-based task scheduling on heterogeneous multicore systems using CBS and QBICTM. Applied Sciences, 2021, 11(12): 5740. [doi: 10.3390/app11125740]

10 Kwok YK, Ahmad I. Dynamic critical-path scheduling: An effective technique for allocating task graphs to

multiprocessors. Proceedings of the International Workshop on Languages & Compilers for Parallel Computing. Berlin: Springer, 2011.

11 Zhou L, Sun SX. Scheduling algorithm based on critical tasks in heterogeneous environments. Journal of Systems Engineering and Electronics, 2008, 19(2): 398–405. [doi: 10.1016/S1004-4132(08)60099-7]

12 Ma JT, Li WT, Fu T, *et al*. A novel dynamic task scheduling algorithm based on improved genetic algorithm in cloud computing. Wireless Communications, Networking and Applications. New Delhi: Springer, 2016. 829–835.

13 Hasan MZ, Al-Rizzo H. Task scheduling in Internet of Things cloud environment using a robust particle swarm optimization. Concurrency and Computation: Practice and Experience, 2020, 32(2): e5442.

14 Kumar AMS, Venkatesan M. Multi-objective task scheduling using hybrid genetic-ant colony optimization algorithm in cloud environment. Wireless Personal Communications, 2019, 107(4): 1835–1848. [doi: 10.1007/s11277-019-06360-8]

15 Topcuoglu H, Hariri S, Wu MY. Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Transactions on Parallel and Distributed Systems, 2002, 13(3): 260–274. [doi: 10.1109/71.993206]

16 Grewe D, Wang Z, O'Boyle MFP. OpenCL task partitioning in the presence of GPU contention. Proceedings of the 26th International Workshop on Languages and Compilers for Parallel Computing. San Jose: Springer, 2013. 87–101.

17 Ravi VT, Ma WJ, Chiu D, *et al*. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. Proceedings of the 24th ACM International Conference on Supercomputing. Ibaraki: ACM, 2010. 137–146.

18 Tian L, Meng C, Zhou FG. A two-level task scheduler on multiple DSP system for OpenCL. Advances in Mechanical Engineering, 2014, 6: 1–10.

19 Lee J, Samadi M, Park Y, *et al*. SKMD: Single kernel on multiple devices for transparent CPU-GPU collaboration. ACM Transactions on Computer Systems, 2015, 33(3): 9.

20 Jääskeläinen P, Korhonen V, Koskela M, *et al*. Exploiting task parallelism with OpenCL: A case study. Journal of Signal Processing Systems, 2019, 91(1): 33–46. [doi: 10.1007/s11265-018-1416-1]

21 彭云峰. 高性能计算应用程序的静态性能分析和建模方法研究. 计算机应用研究, 2021, 38(1): 204–208, 222.

22 Mendis C, Renda A, Amarasinghe SP, *et al*. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. Proceedings of the 36th International Conference on Machine Learning. Long Beach: PMLR, 2019. 4505–4515.

23 Braun L, Nikas S, Song C, *et al*. A simple model for portable and fast prediction of execution time and power consumption of GPU kernels. ACM Transactions on Architecture and Code Optimization, 2021, 18(1): 7.

24 Bouhali N, Ouarnoughi H, Niar S, *et al*. Execution time modeling for CNN inference on embedded GPUs. Proceedings of the 2021 Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools Proceedings. Budapest: ACM, 2021. 59–65.

25 宁成明, 蔡恒雨, 郑启龙, 等. HXDSP 异构计算框架的设计与优化. 小型微型计算机系统, 2022, 43(1): 179–185. [doi: 10.3969/j.issn.1000-1220.2022.01.028]

(校对责编: 牛欣悦)