doi:10.3772/j.issn.1002-0470.2016.12.001

基于 OpenCL 的流式应用程序在 MPSoC 上的动态并行度伸缩调度①

黄 姗②[*,**,***] 石晶林[*,**,***] 萧 放[*,**]

([*]中国科学院计算技术研究所无线通信技术研究中心 北京 100190)

([**]北京市移动计算与新型终端重点实验室 北京 100190)

([***]中国科学院大学 北京 100049)

摘 要 分析了嵌入式系统应用程序的复杂化和多样化趋势,面向嵌入式系统常见的流式应用程序,提出了基于开放运算语言(OpenCL)的统一编程框架,并在此框架的基础上设计一个运行时系统,在应用程序可用计算资源发生变化的场景下,该系统可在线调整应用程序的并行度,并进行动态调度。实验结果显示,与已有的 Flextream 动态调度系统相比,该调度系统在性能上最高可以提升 17%,在动态调度的时间开销上最多可以降低 7%。

关键词 多处理器片上系统(MPSoC),开放运算语言(OpenCL),编程框架,并行度伸缩,运行时系统

0 引 言

近年来,嵌入式系统的应用程序越来越复杂和多样化[1-3],多处理器片上系统(multiprocessor system on chip,MPSoC)因其灵活可编程的特点得到了广泛的应用。出于对性能、功耗效率的要求,MPSoC 上集成了多种异构的计算资源。因此,MPSoC 上的软件编程问题变得越来越复杂。程序员需要对不同类型的计算单元进行编程,甚至需要了解微结构信息从而充分地利用硬件资源的计算能力,提升程序的执行效率。为了降低软件编程的难度,为程序员提供统一的异构计算资源编程接口,苹果公司率先提出了开放运算语言(open computing language,OpenCL)[4]编程框架,使用该编程框架的程序可以在任意的支持 OpenCL 编程框架的异构平台上编译和执行。另一方面,为了提升资源利用率,多个应用程序会以一种动态的方式共享一个 MPSoC 的计算资源。例如,一个视频处理程序占据整个 MPSoC 的计算资源,当另一个拍照程序被启动时,视频处理程序不得不动态地调整它的任务映射,因为一部分计算资源需要被分配给拍照程序。这意味着,系统需要在线地改变任务并行度以及任务映射,来适应变化的计算资源。文献[5]提出了一种半静态的方法,准备多个可能场景下的并行度和任务映射方案,然后运行时系统根据实际情况进行动态的选择。这种方法能适应的场景会受到预先考虑的场景的限制,也会消耗大量的存储资源去保存这些并行度和任务映射方案。文献[6]提出了一种动态调整数据级并行度的方法,但并没有考虑其它类型的并行度调整。文献[7]只考虑了动态改变任务映射,但并没有相应的调整并行度。

综合上述两个方面的因素,本文提出了一种基于 OpenCL 编程框架的流式应用程序在异构多核片上系统的实现框架,并基于此实现了一个运行时系统的框架设计,支持当应用程序在可用计算资源发生变化的场景下,在线调整应用程序的并行度并动

① 国家自然科学基金(61431001)和北京市青年拔尖人才(2015000021223ZK31)资助项目。

② 女,1988 年生,博士生;研究方向:通信基带芯片设计,专用矢量 DSP 处理器设计,片上多核调度系统设计;联系人,E-mail: huangshan@ict.ac.cn

(收稿日期:2016-09-07)

态地调度应用程序,主要贡献在于:在流式应用程序的同步数据流(synchronous data flow,SDF)模型基础上,表征不同类型的并行度,给出局部调整并行度的方法;提出了一种根据可用的计算资源,动态调整并行度,并重新分配计算资源的算法;提出了基于 OpenCL 编程框架的流式应用程序在 MPSoC 平台上进行动态调度的运行时系统。

1 OpenCL 简介

OpenCL[4]是一个在异构平台下的编程框架,异构平台可以由通用处理器(central processing unit,CPU)、图像处理器(graphic processing unit,GPU)、数字信号处理器(digital signal processor,DSP)、可编程门阵列(field programmable gate array,FPGA)以及应用专用集成电路(application specific integrated circuit,ASIC)等组成。OpenCL 一方面提供定义和控制异构平台的应用编程接口(application programming interface,API),另一方面提供基于 C 语言标准扩展的对每个异构处理单元进行编程的 OpenCL C 语言。

OpenCL 编程框架对异构平台上的计算资源进行了层次化的抽象。一个异构计算平台由一个主控制单元(host)和多个计算设备(device)组成。一个异构计算平台的 host 通过 API 管理和控制各个 device,每个 device 执行一段特定的计算任务,被称为核心(kernel)。一般而言,host 实现在 CPU 上,kernel 程序在 device 上执行。一个多核心的 CPU 也可以同时作为 device,即通过操作系统的多线程执行具体的计算任务。

Host 通过为每个 device 创建和维护任务队列来决定 device 执行哪些具体的计算任务,队列里的命令可以是实例化一个 kernel 程序执行的命令,也可以是一条同步命令,或者是数据传输命令。除了显示的添加同步命令之外,host 也可以通过定义任务队列的执行模型来控制队列的执行顺序,命令队列有两种基本的执行顺序,一是串行执行,即严格按照命令入队列的顺序执行;另一种是乱序执行,即当前面的命令没有满足启动条件的情况下允许执行已经满足启动条件的命令先执行。

2 流式应用程序建模分析

本节阐述流式应用程序的图建模,利用图的节点来表示特定的计算任务,利用边来表示任务之间的数据依赖关系。不同类型的并行度可以通过图的结构进行表征,基于这个结构,定义并行度调整的方法。

2.1 流式应用程序的 SDF 模型

同步数据流[8](SDF)图模型经常用来对流式应用程序进行建模。在 SDF 模型中,两个存在数据依赖关系的节点用有向边相连,节点表示两个特定的计算任务,有向边表示数据的流动方向,产生数据的节点被称为生产者,使用数据的节点被称为消费者。图 1 为流式应用的数据流图模型。

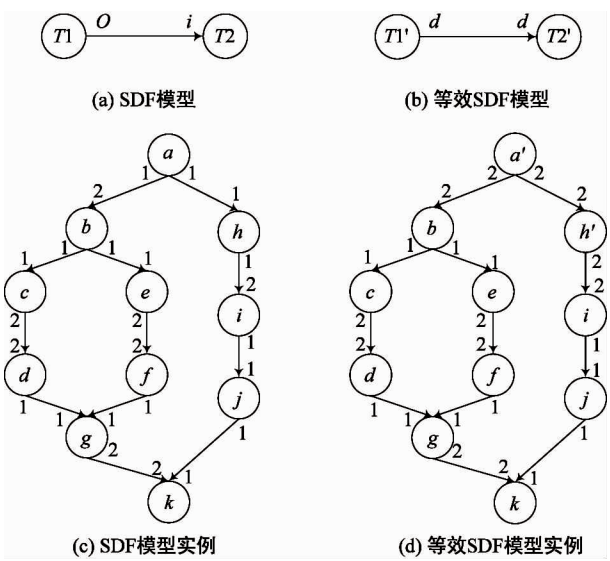


图 1 流式应用程序的数据流图模型

如图 1(a) 所示,有向边相连的两个节点代表的计算任务 T_1 和 T_2 每一次被执行都产生或消耗确定数据量,分别用 o 和 i 表示,被标记在有向边的起点和终点处。对于 SDF 模型,每对有向边相连的节点,需要存在整数 r_1 和 r_2 满足公式

$$o \times r_1 = i \times r_2 \tag{1}$$

其中 r_1 和 r_2 分别表示相连的两个节点 T_1 和 T_2 的执行次数。如图 1(b)所示, T'_1 和 T'_2 分别是 T_1 和 T_2 执行 r_1 和 r_2 次之后的等效节点, T'_1 产生的数据量

和 T'_2 消耗的数据量相等,用 d 来表示,本文称这种具有相等的生产率和消耗率的 SDF 模型为等效 SDF 模型。图 1(c) 和图 1(d) 分别是 SDF 模型和等效 SDF 模型的一个实例,节点 a 和 h 需要被重复执行两次,转化成节点 a' 和 h', 相应的数据率也要变化。

后文将使用等效 SDF 模型所定义的数据流图 $G=(V,E)$ 来对流式应用程序进行刻画,其中 V 表示节点的集合, E 表示有向边的集合。对于每一个节点 $v \in V$, 使用该节点对应的计算任务的工作负载对其进行量化,节点工作负载可以通过静态分析或动态评估得到;对于每一条边 $e \in E$, 使用有向边传递的数据量对其进行量化,即等效模型中的 d。

2.2　并行度的表征

在数据流图模型 G 上添加特殊节点可以显式地表征并行度,如图 2(a)所示。进一步地,图 2(b)表示数据并行,图 2(c) 表示任务并行,图 2(d)表示流水并行。数据并行是指同时对不同的输入数据进行相同计算任务,任务并行是指同时执行了两个或多个相互独立的没有数据依赖关系的任务。数据并行和任务并行是通过添加拆分节点 S 和合并节点 J 显示的表现在任务流图中,通常被形象化地表现为一种水平并列的模式,本文使用水平并行(horizontal parallelism,HP)来描述。

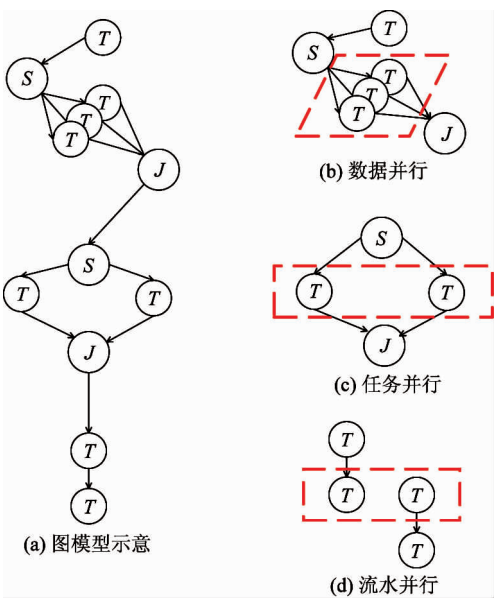


图 2　数据流图模型及其并行度表征

由于流式应用程序会迭代执行多次来处理源源不断到达的需要处理的数据,所以数据流图 G 实际上被一个隐式的外部循环所包含。这个特征使得流式应用程序得以非常自然地利用流水线的实现方式。通过将外层隐式的循环进行展开,原本属于不同迭代周期的任务节点就可以并行执行。这种通过循环展开获得流水并行的技术最早被应用在指令层次,称为软流水技术[9]。Gordon 等人将这种软流水技术应用在粗粒度的任务层次,挖掘了任务级的流水并行[10]。如图 2(d) 所示,两个级联的任务节点通过循环展开即可构成流水并行的结构,本文使用垂直并行(vertical parallelism,VP)来描述。

2.3　并行度调整方法

基于流式应用程序的图建模和对不同并行类型的表征,本节阐述并行度调整的方法。在应用程序执行的过程中,分配给该应用程序的硬件资源有可能发生变化,这就需要在线动态调整应用程序的并行度。静态编译阶段可以使用复杂度较高的分析优化算法,获得最优或接近最优的结果,而动态并行度调整由于占用运行时的时间,不能使用复杂度太高的算法。以一个优化的静态分析优化的结果作为动态调整的起点,一方面可以降低动态调整算法的复杂度,另一方面也可以保证性能的损失在可接受的范围内。本文提出的动态并行度调整是基于一个应用程序能够使用异构平台上所有的计算资源的场景下的静态分析结果。因此,本文的并行度调整根据不同的并行度类型,对流图模型进行局部节点合并。

图 3 是通过局部的节点合并进行并行度调整的 API 定义,分为水平并行度调整和垂直并行度调整两种类型。水平并行度调整是将包含在同一对 S-J 节点对 v_{sp} 和 v_{jo} 中的两个节点 v_i 和 v_j 合并成一个行的节点 v_k, 并通过置 v_k 的前驱节点 $In(v_k)$ 和后继节点 $Out(v_k)$ 分别为 S-J 节点对的 v_{sp} 和 v_{jo} 将新节点 v_k 连接到图中。垂直并行度调整的 API 将有向边相连的两个节点 v_i 和 v_j 合并成新的节点 v_k, 并通过置 v_k 的前驱节点 $In(v_k)$ 和后继节点 $Out(v_k)$ 分别为原 v_i 的前驱节点和原 v_j 的后继节点将新节点 v_k 连接到图中。

```
// 水平并行度调整
// 节点 v_i,v_j 包含在一对S-J节点对 v_sp, v_jo 中
function HPScale (v_i, v_j, v_sp, v_jo)
    v_k ← v_i ∪ v_j
    In(v_k) ← v_sp
    Out(v_k) ← v_jo
endfunction
```

(a) 水平并行度调整API

```
// 垂直并行度调整
// (v_i,v_j) ∈ E
function VPScale(v_i, v_j)
    v_pre ← In(v_i)
    v_suc ← Out(v_i)
    v_k ← v_i ∪ v_j
    In(v_k) ← v_pre
    Out(v_k) ← v_suc
endfunction
```

(b) 垂直并行度调整API

图 3　并行度调整 API

3　基于 OpenCL 的实现框架

本节阐述基于 OpenCL 编程框架的流式应用程序的实现框架。首先需要确定经过数据流图建模的应用程序和异构平台的计算资源之间的映射关系，然后根据这个映射关系通过主控单元 host 的逻辑管理和调度各个 device 执行所承载的计算任务。

3.1　数据流图模型到异构平台的映射

图 4 展示了一个流式应用程序的数据流图模型 G 映射到一个异构平台的方法，模型的节点被映射到异构平台的 device 处理资源上（虚线），例如 GPU 和 DSP；图模型的边实例化为先入先出（FIFO）的数据结构，映射到异构平台的不同存储资源上（点划线）。

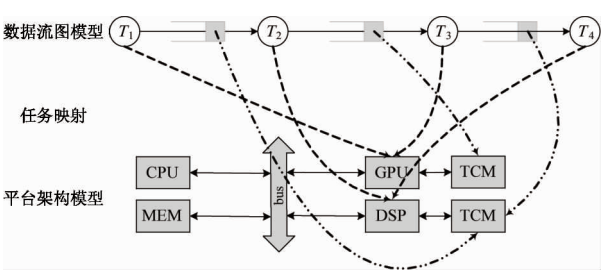


图 4　数据流图模型到计算平台的映射

3.2　基于 OpenCL 的实现框架

图 5 是流式应用程序基于 OpenCL 的实现框架图。主控制的功能主要包括根据应用程序模型和任务映射，调用 OpenCL 的 API 库函数控制和调度各个 device 完成相应的计算任务。在线的并行度调整和动态任务调度需要主控单元调用并行度调整的 API 函数（2.3 节）来完成。各个 device 上执行的计算任务用 OpenCL C 语言实现，通过 OpenCL C 编译器产生对应的可执行文件。这个过程可以静态地完成，也可以由主控单元动态地调用编译组件完成，如图 5 中的虚线箭头和虚线框所示。

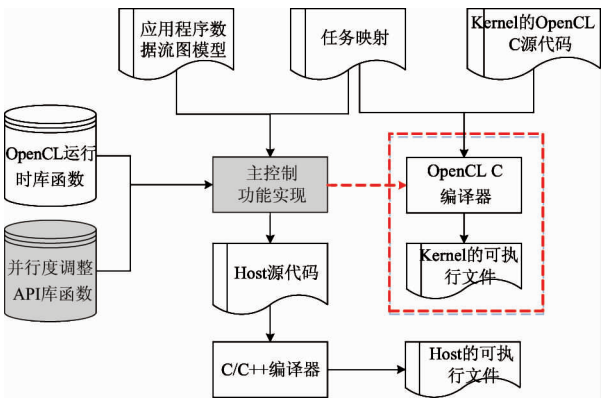


图 5　实现框架图

4　运行时系统

本节描述运行时系统对流式应用程序的调度，以及在线调整并行度进行动态调度的方法。

4.1　流水线调度的基本方法

由于流式应用程序经常需要重复执行多次来处理源源不断到达的数据，所以一个典型的流式应用程序的顶层有一个隐式的循环。根据这个特征，Gordon 等人将软流水技术调度应用在粗粒度的任务层次[10]，通过对整个流式应用程序顶层的隐式循环进行展开，来自不同循环迭代次数的任务就可以并行执行，流式应用程序因为本身的特征可以很自然地从流水并行中获得巨大的数据吞吐收益。

流水线调度首先根据计算任务到处理资源的映射计算每个任务的所属于的迭代周期，如公式

$$I_{v_j} = \begin{cases} \max\limits_{(v_i,v_j)\in E} I_{v_i} + D, map_{v_i} \neq map_{v_j} \\ \max\limits_{(v_i,v_j)\in E} I_{v_i}, \ map_{v_i} = map_{v_j} \end{cases}$$
$$v_i, \ v_j \in V; \ map_{v_i}, \ map_{v_j} \in P \qquad (2)$$

所示。数据流图源节点的迭代周期为 0，其余对于数据流图模型的每一条边 $(v_i, v_j) \in E$，v_j 的迭代周期取决于其所有前驱节点 v_i 中迭代周期最大的一个，如果节点 v_i 和 v_j 的任务被分配在不同的处理资源上，那么就需要在最大迭代周期的基础上加一个迭代距离 D，map_{v_i} 和 map_{v_j} 分别表示任务 v_i 和 v_j 映

射的处理资源,P 表示计算平台上所有处理资源的集合。如果计算处理资源支持任务计算和数据传输并行执行,即一个处理器在启动了 DMA 数据传输任务之后可以不等待数据传输任务结束即开始执行后续的计算任务,则 D 为 2,否则 D 为 1。如果节点 v_i 和 v_j 的任务被分配在相同的处理资源上,那么 v_j 的迭代周期就等于 v_i 的迭代周期的最大值。

根据已经计算好的每个节点所属于的迭代周期,根据公式

$$Buf_{(v_i,v_j)} = (I_{v_j} - I_{v_i} + 1) \times weight_{(v_i,v_j)} \quad (v_i,v_j) \in E \tag{3}$$

可以计算出每条有向边所需要的存储空间,用以进行数据传递,其中 $weight_{(v_i,v_j)}$ 每条有向边传递的数据量,即图模型中的 d。

一个典型的流水线调度需要三个阶段,分别为流水的建立期(Prologue)、核心期(Steady State)和退出期(Epilogue)。在流水的建立期,通过逐步添加各个迭代周期的计算任务到各个 device 的任务队列,在每两个存在数据依赖的任务之间添加足够的缓存空间,使得数据的消费者节点并不直接依赖于其生产者当前产生的数据,而是使用生产者上一次产生的缓存数据。流水调度进入核心期后,分配在不同计算资源上的任务都可以彼此独立地执行,不等待其它计算资源完成任何的任务计算,由此便获得了流水并行。退出期即当没有新数据到达时,逐步释放掉已经完成的计算任务。图 6 是一个简单的流水线调度不同时期的示意图。

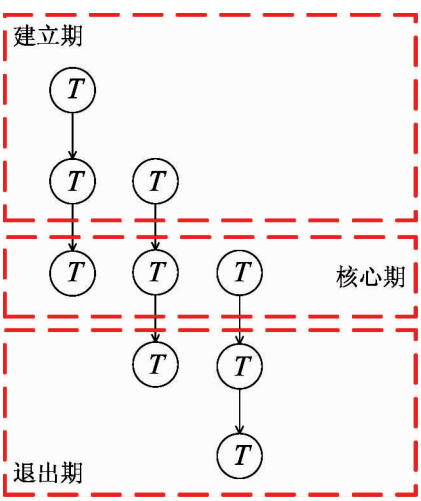


图 6　流水线调度时期示意图

4.2　运行时系统调度流程

图 7 为运行时系统工作的全流程,系统启动之后初始化 device 设备,根据任务映射关系计算图模型中每个节点的迭代周期,并为每个有向边分配存储空间。初始化设备同时置变量 resChange 为 FALSE,表示系统初次启动,随后进入流水线调度的建立期。在建立期中,host 根据每个计算节点的迭代周期,向 device 的任务队列中添加新的迭代周期的任务并执行整个任务队列,直到所有迭代周期的节点都添加到任务队列中,并在任务队列的末尾添加栅障同步命令。之后,流水进入核心期,device 的任务队列不再添加新的任务,只要还有新数据到来并且计算资源不变(! dataEnd && ! resChange),任务队列的任务就被循环执行。如果不再有新数据(dataEnd)需要处理,则进入流水的退出期。在流水的退出期中,主控 host 根据每个计算节点的迭代周期,从 device 的任务队列中删除某个迭代周期的任务并重新执行任务队列,直到任务队列为空。最后调用 OpenCL 提供的 API 接口函数,释放相关任务队列、存储资源以及 device,结束整个应用程序的执行。

当应用程序处于核心期时,如果计算资源有所变化,则运行时系统进入动态并行度伸缩调度的工作流程。首先进入并行度伸缩调度的决策期,如图 7的点划线框所包含的流程所示,包括调用并行度伸缩的算法,之后重新计算图模型中每个节点的迭代周期和分配每条有向边的存储资源,并置变量 resChange 为 TRUE。随后,运行时系统进入并行度伸缩调度的切换期,切换期分别由流水的退出期和建立期组成,完成从上一个核心期到下一个核心期的转换工作。

4.3　在线并行度伸缩算法

本小节详述在线并行度伸缩的算法细节,即图 7中的灰色标记的内容。

流式应用程序使用 2.1 节中的图模型 $G = (V, E)$ 来表示,集合 P 是异构平台上所有处理资源的集合,初始的任务映射用数组 Map 表示,如下式所示:

$$Map = \{map_{v_1},\cdots,map_{v_{|V|}}\}$$
$$v_1,\cdots,v_{|V|} \in V,\ map_{v_1},\cdots,map_{v_{|V|}} \in P \tag{4}$$

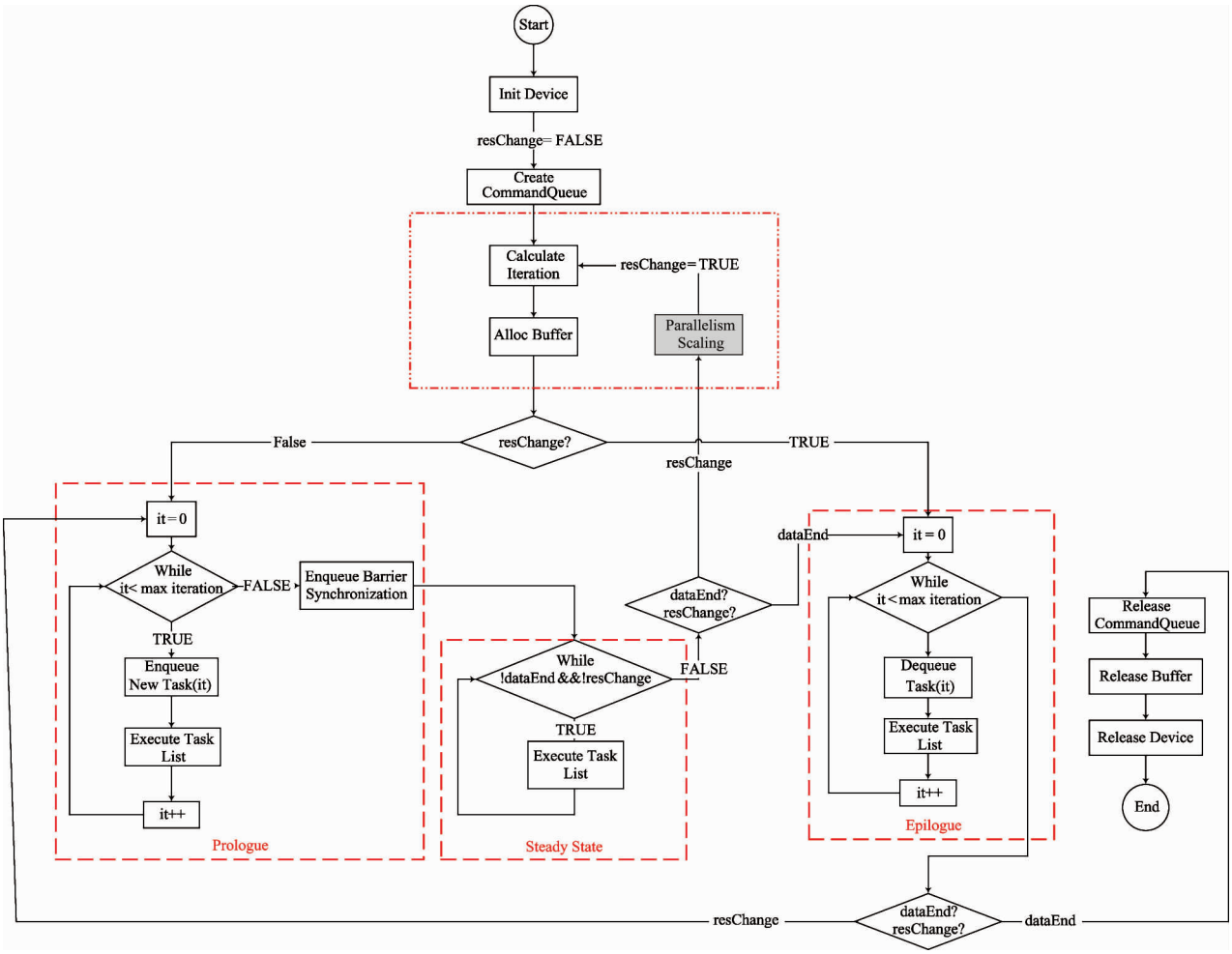


图 7　运行时系统流程图

N_r 表示处理资源变化后仍然可用的计算单元个数，满足 $N_r \leqslant |P|$。图 8 为并行度调整的算法流程，为了清晰起见，整个算法流程分成 4 个步骤，下面将逐步阐述。

步骤 1：决定哪些处理单元被保留继续执行该应用程序。通过对初始任务映射中的处理器按照每个处理器承载的计算任务个数做升序排列，序列前 N_r 个处理资源被保留，并组成集合 P_{reserve}。因为静态任务的映射结果是最优解或者近优解，每个处理资源的任务负载总和是趋近于平衡的，保留承载任务个数少的处理器即保留了单个任务负载较高的任务，通过重新分配多个负载较低的任务更有利于新的任务映射达到负载均衡。与此同时，在静态任务映射中被分配到集合 P_{reserve} 上执行的节点任务组成集合 V_{reserve}，其余组成集合 V_{victim}。

步骤 2：检测并行度。通过函数 DetectParallel 检测集合 V_{victim} 中的每个任务是否存在并行度。遍历集合 V_{victim} 中的每个节点 v，首先将 v 添加集合 V_{scale} 中，判断如果 v 属于一对 *S-J* 节点对中，那么 v 是水平平行度的一部分，那么则遍历与它同属于这个 *S-J* 结构的每个兄弟节点 v_s，如果 $v_s \in V_{\text{reserve}}$，那么节点 v_s 和承载其执行的计算单元 map_{v_s} 组成一个节点资源对 (v_s, map_{v_s})，进而构成节点资源对集合 $\text{VP}_{\text{neighbor}} = \{(v_s, map_{v_s})\}, v_s \in V_{\text{reserve}}$；如果 $v_s \in V_{\text{victim}}$，则节点 v_s 构成集合 V_{scale}。同理的，判断如果 v 不属于 *S-J* 结构中，那么 v 只可能是垂直并行度的一部分，那么则遍历它的前驱节点和后继节点，如果其前驱或者后继节点属于集合 V_{reserve}，那么前驱或者后继节点则与其映射的处理资源构成节点资源对集合 VP_{neighbor}。在本步骤中，输出的集合 V_{scale} 表示的是需要调整并行度并重新分配计算资源的节点的集合。

步骤 3:调整并行度。函数 ScaleParallel 将上一个步骤中形成的集合 V_{scale} 中的任务节点和集合 $VP_{neighbor}$ 中的节点进行合并,达到局部的调整水平或垂直并行度的效果,合并的节点将由对应节点资源对的计算资源承载执行。方法是首先将集合 V_{scale} 根据任务复杂度进行降序排列,依次考虑排序后的每个计算节点 v_i,同时选择当前 $VP_{neighbor}$ 中任务负载最小的处理资源 p 所对应的节点资源对 (v_j,p),将 v_i 与 v_j 进行节点合并,同时保证合并之后处理资源 p 的任务负载不能超过阈值(Th)的 10%。这个阈值 Th 表示了一种理想的情况,即所有的任务在剩余的 N_r 个处理资源上绝对平均的分配。设置这样一个约束的原因是局部的调整并行度可以降低同步以及通信的开销从而提升性能,但仍然需要兼顾全局的负载均衡。此外,在进行了节点合并之后,需要检查并删除无用的 S-J 节点对。

步骤 4:为不能调整并行度的节点分配处理资源。算法进行到这里,集合 V_{victim} 中还余下一些没有进行合并的计算节点,这里采取贪心算法将剩余节点分配给计算资源集合 $P_{reserve}$。

```
Input: G = (V, E), P, Map, Nr
// 步骤1
Th ← weight(V)/Nr
Preserve ← ∅ , Vreserve ← ∅
SortByNumberofTasksAscending(P)
Preserve ←{pi, 1≤i≤Nr}
Vreserve ← {vi, mapvi ∈ Preserve }
Vvictim ← V\ Vreserve
// 步骤2和步骤 3
for each v ∈ Vvictim do
VPneighbor←∅ , Vscale←∅
{type,VPneighbor,Vscale } = DetectParallel(In(v), Out(v))
ScaleParallel(G, VPneighbor, Vscale, type)
if type == HP then
RemoveSJWithOneVexInside(In(v), Out(v))
end if
end for
// 步骤4
if Vvictim≠∅ then
Vscale←∅
SortByWeightDescending(Vvictim)
SortByWorkloadAscending(Preserve)
for each vv ∈ Vvictim do
Vscale←Vscale U { vv }
if weight(Vscale) ≥Th then
pr = NextProcessor(Preserve)
mapVscale ← pr
Vvictim←Vvictim \ Vscale
Vscale←∅
end if
end for
end if
```

(a) 并行度调整算法

```
Function DetectParallel(v)
if |In(v)| == 1 && |Out(v)| == 1 && In(v) == S && Out(v) == J
then
type ← HP
for each (In(v), vs) ∈ E do
if vs  Vreserve then
VPneighbor ←VPneighbor U {(vs,mapvs)}
else
Vscale←Vscale U {vs}
end if
end for
else
type ← VP
Vscale←Vscale U {v}
for each vi ∈ {In(v) U Out(v)} do
if vi ∈ Vreserve then
VPneighbor←VPneighbor U {(vi,mapvi)}
end if
end for
end if
return (type, VPneighbor, Vscale)
end function
Function ScaleParallel(G,VPneighbor, Vscale, type)
for each vi ∈ Vscale do
(v, p) = ProcessorWithMinWorkload(VPneighbor)
if workload(p)+weight(vi) ≤ Th*1.1 then
if type == HP then
HPScale(v, vi)
else
VPScale(v, vi)
end if
Vvictim ← V \ { vi }
end if
end for
end function
```

(b) 检测和调整并行度子函数

图 8　并行度调整算法

该算法的复杂度主要对于集合 V_{victim} 中的每个节点,DetectParallel 函数需要遍历其水平或者垂直的邻居节点,ScaleParallel 函数需要通过排序算法选择一个邻居节点进行合并。然而这个过程被限制在

了数据流图模型的一个局部结构,所以涉及到的节点个数有限,使得该算法能够在有效的时间内完成。而全局的排序算法主要集中在第 4 步,出于复杂度的考虑,步骤 4 使用了贪心算法,保证算法整体可行。

5 实验与结果分析

5.1 实验平台与基准程序

本文选择 parallella 开发板[11]作为实验平台。Parallella 开发板上集成了一个 ARM A9 中央处理器,和一个 Epiphany 的协处理器,包含 16 个计算单元,每个计算单元包含一个紧耦合的 32KB 的 SRAM 存储器。Parallella 板上集成集成了 1GB 大小的 DDR3 存储器作为内存,其中的 32MB 的存储空间作为被主控制单元 ARM A9 处理器和协处理器 Epiphany 共享。

Parallella 平台支持 OpenCL 的编程框架,主控制处理器 ARM A9 上搭载 Linux 的操作系统,其中 OpenCL 的 API 基于 Epiphany 提供的软件开发包[12](epiphany software development kit, eSDK)实现。Host 处理器 ARM A9 通过 API 函数管理和控制 Epiphany 的每个计算单元执行特定的计算任务。

本文选择 StreamIT 基准程序作为评估该编程框架和运行时系统的基准程序[13]。StreamIT 是由美国麻省理工大学计算机系统实验室研发的针对流式应用程序的高层编程语言,它同时收集了多个典型的流式应用程序作为基准测试程序集合。

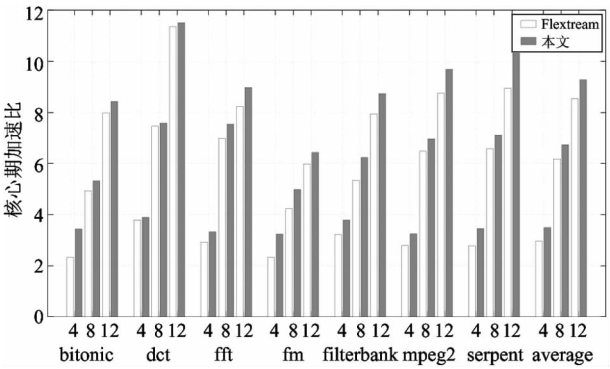
5.2 实验设计与结果分析

本文假设 16 个计算单元全部可用的场景为静态任务映射起点[10]作为动态并行度伸缩的起点。为了说明本文方法的有效性,本文将与 Flextream[7]的结果进行对比,为了保证对比的公平性,我们将 Flextream[7]的算法在 Parallella 的开发板上进行了实现,对比将从流水核心期性能、动态伸缩调度时间开销和运行时系统整体性能及吞吐能力三个方面展开。流水核心期的性能是评估的首要指标,因为一个基于流水调度方法实现的系统的性能主要取决于流水核心期的性能。本文提出了动态的并行度伸缩和调度的方法,动态变化的时间开销也是衡量该运行时系统的一个关键因素。本文以一个连续变化的场景来评估系统整体的性能和吞吐能力。

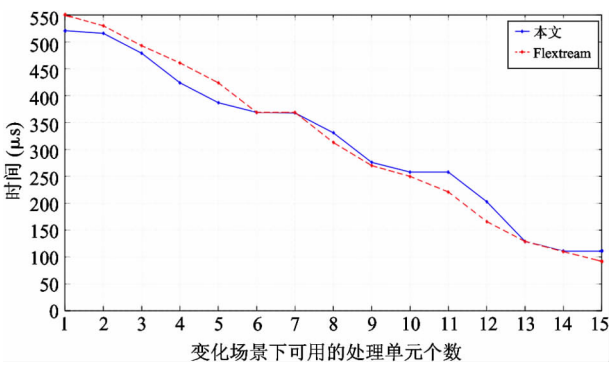
(1) 流水核心期性能。图 9(a) 表示了基准程序的流水核心期在 12/8/4 个处理单元上执行相对于在 1 个处理单元上执行的加速比,它们都是从 16 个处理单元的场景动态调整过来的,与 Flextream[7]相比,本文可以获得最多 17% 的核心期性能提升。这是因为本文在决策期采用的算法首先考虑了适当的调整并行度,避免了在处理资源减少的情况下过度并行导致的不必要通信和同步的开销,从而提升了性能。

(2) 动态伸缩调度时间开销。除了核心期的性能之外,完成一次动态调整切换的时间开销也是衡量系统性能的重要指标,此处选择基准程序 MPEG2 作为基准程序。根据第 4 节的阐述,运行时系统通过决策期和切换期两个阶段完成对应用程序的动态调度。决策期如图 7 中点划线包含的框所示,包含并行度调整、迭代周期和缓存需求计算等。图 9(b) 展示了决策期从 16 个可用计算单元的场景分别到 1 ~ 15 个可用计算单元的场景所需的时间开销。因为静态任务映射是基于 16 个可用计算单元的,所以越接近 16 个计算单元的变化场景,所需的时间开销越小。同样的,本文与 Flextream[7]的算法的时间开销进行对比。在场景变化比较大的情况下,例如从 16 个计算单元到 4 个可用计算单元场景变化,本文的时间开销更小。这是因为并行度调整算法(图 8)中步骤 2 和步骤 3 局部合并了一些节点,从而降低了步骤 4 中的排序算法的时间复杂度。然而当场景变化比较小时,例如从 16 个计算单元到 1 个计算单元的场景变化,本文的算法带来的局部搜索和比较的复杂度超过了全局排序降低的复杂度,则会消耗更多的时间。图 9(c) 展示了切换期的时间开销,从 16 个可用计算单元到 1 到 15 个可用资源 15 个场景变化。切换期的调度分为流水退出期和流水建立期两个阶段,图 9(c) 进行了分别的比较。退出期的时间开销主要取决于前一个场景的资源分配和流水阶段总数,因为本实验所设计的 15 个场景的起点都是基于 16 个可用计算单元,所以退出期调度的

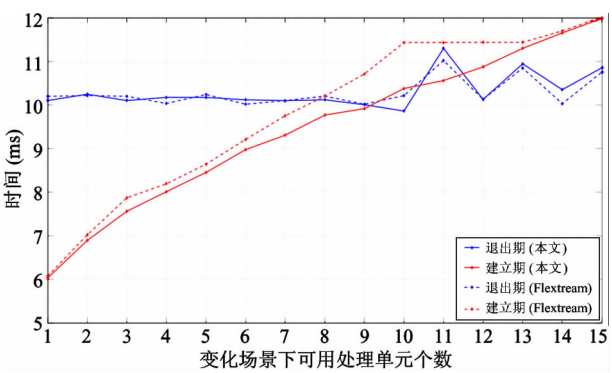
时间开销和 Flextream 的算法基本一致。然而由于本文的动态伸缩算法建立新场景时减少了不必要的并行度带来的开销,所以在流水建立期上与 Flextream 相比可最多减小 7% 的时间开销。



(a) 流水核心期性能



(b) 动态调度决策期时间开销



(c) 动态调度切换期时间开销

图 9　流水核心期性能及动态调度时间开销

(3) 运行时系统整体性能和吞吐能力。尽管动态调度的决策是基于一个静态的任务映射结果,但系统仍然可以从任意的一个场景迁移到另一个。本文评估了 MPEG2 基准程序在一个连续变化的场景下的运行效率,可用的计算单元数按照 16、13、9、5、7 依次变化,每个场景持续 200s 时间。在整个变化过程中,本文的运行时系统可以完成 432000 次完整的执行,Flextream 可以完成 374000 次。由于流式应用程序每一次完整地执行消耗和产生等量的数据,所以本文在整体上提供了更高的数据吞吐率。

6　结　论

本文首先提出了流式应用程序基于 OpenCL 编程框架的实现方法,并基于此设计了动态调度的运行时系统。当计算资源发生变化时,运行时系统通过决策期合理的调整并行度和任务分配,随后通过流水线调度的退出期和建立期完成动态切换。实验表明,相较于已有的 Flextream,本文可以最多提升 17% 的核心期性能,并降低 7% 的切换时间开销。

参考文献

[1] Liu L, Zhou Y, Tian L, et al. CPC-based backward compatible network access for LTE cognitive radio cellular networks. *IEEE Communication Magazine*, 2015, 53(7): 93-99

[2] Zhou Y, Liu L, Pan Z, et al. Two-stage cooperative multicast transmission with optimized power consumption and guranteed coverage. *IEEE JSAC on SEED*, 2014, 32(2):274-284

[3] Garcia V, Zhou Y, Shi J. Coordinated multipoint transmission in dense cellular networks with user-centric adaptive clustering. *IEEE Trans Wireless Comm*, 2014, 13(8):4297-4308

[4] The OpenCL Specification, Khronos OpenCL Working Group, https://www.khronos.org/opencl/. 2016

[5] Schor L, Bacivarov L, Yang H, et al. Expandable process networks to efficiently specify and explore task, data, and pipeline parallelism. In: Proceedings of the 2014 International Conference on Compilers, Architecture and Sythesis for Embedded Systems, New Dehli, India, 2014

[6] Choi Y, Li C, Silva D, et al. Adaptive task duplication using online bottleneck dectection for streaming applications for heterogeneous architecture. In: Proceedings of

the 9th Conference on Computing Frontiers, NY, USA, 2012. 163-172

[7] Hormati A, Choi Y, Kudlur M, et al. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In: Proceedings of the 18th International Conference on Parallel Architecture and Compilation Techniques, Raleigh, USA. 214-223

[8] Lee E A, Messerschmitt D G. Synchronous data flow. *Proceedings of the IEEE*, 1987, 75: 1235-1245

[9] Allan V, Jones R, Lee R, et al. Software pipelining. *ACM Computing Surveys*, 1995, 27: 367-432

[10] Gordon M, Thies W, Amarasinghe S. Exploiting coarse-grained task, data, pipeline parallelism in stream programs. In: Proceedings of the 12th International Conference on Architecture Support for Programming Languages and Operating Systems, San Jose, USA, 2006. 151-162

[11] E16g301 epiphany 16-core microprocessor. http://adapteva.com/docs/e16g301 datasheet.pdf

[12] Epiphany sdk reference. http://adapteva.com/docs/epiphany sdk ref.pdf

[13] StremI T. http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml

An openCL based streaming applications program's dynamic parallelism scaling scheduling on MPSoC

Huang Shan[*,**,***], Shi Jinglin[*,**,***], Xiao Fang[*,**]

([*] Wireless Communication Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

([**] Beijing Key Laboratory of Mobile Computing and Pervasive Device, Beijing 100190)

([***] University of Chinese Academy of Sciences, Beijing 100049)

Abstract

The complex and diversity trends of the application programs for embedded computing systems were analyzed. Then, a unified programming framework based on the open computing language (OpenCL) was proposed for embedded computing systems' common streaming application programs, and on the basis of the framework, a runtime system was designed. Under the variation of application programs' computing resources, the system on-line regulates programs' parallelism, and conducts dynamic parallelism scaling scheduling. The experimental results showed that, compared with the existing dynamic scheduling system of Flextream, the proposed scheduling system's performance was improved by 17%, and the runtime overhead of the dynamic scheduling was reduced by 7%.

Key words: multiprocessor system on chip (MPSoC), open computing language (OpenCL), programming framework, parallelism scaling, runtime system