doi:10.3772/j.issn.1002-0470.2022.02.006

PPTM:一种面向异构系统的主动式任务映射方法①

龚施俊②*** 鄢贵海* 李晓维*

(*中国科学院计算技术研究所计算机体系结构国家重点实验室 北京 100190)

(**中国科学院大学 北京 100049)

摘 要 在数据高速增长的背景下,异构计算作为满足新兴应用不断提高的算力需求的有效途径,涌现了许多异构加速系统。在这些异构加速系统中,高效的任务映射是充分发挥加速器潜能提升应用程序性能的关键之一。先前工作提出了许多基于有向无环图如何最小化应用程序整体执行时间和最小化异构多处理器之间通信开销等高效的任务映射方法,这些工作通常采用将任务映射到加速器上来提高整个应用的性能。但某些应用程序如果将所有子任务全部映射到加速器上执行,会带来额外的通信开销,进而可能达不到提升性能的预期,甚至造成整个应用程序的性能下降。因此,本文提出了一种基于预测的主动式任务映射算法(PPTM)来应对这样的场景,实现高效的任务映射。实验表明,本文算法能够更准确感知计算任务的运行时状态,大幅提高应用程序的整体性能。

关键词 异构计算;异构加速系统;任务映射;主动式;预测算法;加速器

0 引 言

随着半导体芯片制程工艺发展,晶体管尺寸不断逼近物理极限,摩尔定律逐步放缓[1-4],实现性能提升愈发依赖于新的体系结构设计方法。领域专用体系结构(domain-specific architecture, DSA)针对特定问题域定制体系结构,具有显著的性能和能效优势。DSA 通常被称为“加速器”,因为与通用 CPU 上执行整个应用程序相比,它们只会加速某些应用程序。典型加速器有 GPU、TPU[4-5]和用于软件定义网络(software define network, SDN)的处理器[6]等。

类似于通用 CPU 需要操作系统驱动,之前的工作中,针对不同的加速场景或加速器涌现了许多成熟高效的异构加速系统或者框架,如 OpenCL[7-8]、CUDA[9]、基于 OpenVX[10]的 AMDOVX[11]以及 TensorFlow[12]等。这些系统运行时将识别加速可行性并决定如何利用加速器的责任转移给程序员,这对程序员提出了更高的要求,需要程序员熟悉加速器的底层实现细节。这导致程序员水平的个体差异对程序性能的影响巨大,差别甚至达到数千倍[1],其主要原因是异构系统环境下的任务映射的差异。

为实现在异构系统中高效的任务映射,之前的工作提出了很多基于任务在不同处理上的执行性能和如何最小任务之间的通信开销等的调度算法[13-18]。这些工作一般采用计算流图来表示应用,每个节点代表一个计算任务,然后将这些任务在满足某些限制条件下,如最小化通信开销、最小化执行时间等,将其调度到加速器上执行。但是,我们发现针对某些应用场景,基于执行时间快慢来进行任务调度无法摊销计算节点之间的通信开销,同时在特地处理数据规模时,相比于 CPU 加速器对计算核能带来的加速比有限。在这种情况下,将计算节点映射到加速器上执行带来的性能增益将因整个应用程

① 国家自然科学基金(61872336,61572470,61532017,61432017,61521092,61376043)和中国科学院青年创新促进会(404441000)资助项目。

② 男,1992 年生,博士生;研究方向:领域专用计算机体系结构;联系人,E-mail: gongshijun@ict.ac.cn。

(收稿日期:2020-12-25)

序存在主机端(HOST)与加速器设备端的通信开销而被部分抵消掉,大幅降低了预期的性能提升。

为了解决上述问题,本文提出了一种基于预测的主动式任务映射算法(prediction-based proactive task mapping algorithm, PPTM)。首先,通过预测算法动态调整计算节点的运行时属性,包括不同计算平台的执行时间和平台之间的通信开销等。然后,基于计算节点的性能增益和额外开销数据来设计任务映射算法,完成计算流图中任务节点在异构平台上的映射。最后,通过一个案例研究来验证该算法的有效性。

本文的主要贡献包括以下 4 个方面。

(1)提出了一种面向异构加速系统的高效任务映射方法。该算法可以精确捕捉任务运行时特征来完成应用程序在异构加速平台上的任务映射,从而提高应用整体性能。

(2)发现在一个应用程序的计算流图中,虽然将计算节点映射到加速器上执行能带来较大的加速比,但是需要根据整个应用程序的计算流图来考虑其映射行为,否则可能无法带来性能增益。

(3)发现针对整个应用程序,并不是降低的通信开销越多,性能提升就越大,而是需要针对特定应用以及运行时数据来权衡。

(4)案例研究。本文以面向流式应用的加速器为例,来评估 PPTM 及其他映射算法的性能。这一案例研究的结论也可以推广到面向其他应用领域的加速器上。

1 相关工作

1.1 异构加速系统与任务映射

在数据高速增长的背景下,异构计算是满足新兴应用不断提高算力需求的有效途径[19]。为了更好地发挥专用加速器的性能以及给用户提供更加优化的编程接口,涌现了很多成熟高效的异构加速系统。OpenCL 提出了面向异构设备上编程的行业标准,通过它可以将应用程序的计算密集型部分卸载到加速设备上从而提高应用程序性能,它的目的是为异构系统提供统一的开发平台[20]。OpenCL 将主机与加速设备的交互方式进行了统一的抽象,并提供统一的编程语言。用户可以在程序池中选择由设备商实现的 kernel 程序来完成对加速设备的调用,然后在主机或设备上创建存储单元,通过主机端与设备端的数据复制来完成设备端程序与主机端程序之间的数据交互,从而实现对底层加速器的透明化管理。CUDA 作为英伟达的商业化异构加速系统,其最终目的是支持面向通用计算的 GPU(general-purpose computing on graphics processing units, GPGPU)。CUDA 提供统一的开发套件,并封装实现了很多高效计算库(cuFFT、cuBLAS 等)以及提供了统一的高效成熟的 NVCC 编译器,相比于 OpenCL 依赖于设备商的驱动实现,对用户更加友好[9]。TensorFlow[21]作为 Google 开源的数据流图科学计算库,主要用于机器学习等人工智能领域。通过提供众多机器学习领域抽象核函数实现,用户可以选择其构建特定的机器学习算法,降低了机器学习算法的设计门槛。在底层可以对接 CUDA 等加速器运行时系统,来支持异构计算[15]。

在之前的异构加速系统中,程序员的主要工作是将一个应用程序划分成不同的计算任务,调用异构加速系统提供的核函数来实现特定的计算任务,并通过在加速器上执行来提高整个应用程序的性能。因此,一个应用程序在异构平台上如何高效地进行任务映射就显得至关重要,这关系到整个应用程序的实际性能,对程序员提出了更大的挑战。

之前的工作中,涌现了诸多针对异构多处理器的任务调度优化算法。文献[15]根据异构处理器的任务执行时间,提出了 2 种启发式算法来最小化应用程序在异构平台上的执行时间。而文献[17]和[14]则为了能最小化任务之间的通信开销提出了相关的任务映射算法。当然,还有很多调度算法[16,22-29],但是这些工作通常是考虑不同应用在加速器设备上的差异表现来决定最合适的任务映射方式,这与本文提出的在特定的应用程序中有些任务被映射到加速器上会降低应用程序整体性能还是有所差异的。

1.2 计算流图

计算流图通常是一个有向无环图,其中的每一

个节点都类似于一个函数调用,代表一个计算核,节点间的有向边代表了数据的输入输出关系[2]。其定义如下:

定义 1 计算流图由二元组 $G=(V, E)$ 表示,其中 V 为节点或顶点的有限集合,每个节点表示一个计算核;E 为有向边的有限集合,边表示计算核之间通信或依赖关系。每条边是一个有序二元组 $e=(v_1, v_2)$,其中 $v_1, v_2 \in V$。如果 $e=(v_1, v_2) \in E$,则边 e 从 v_1 指向 v_2,且称 v_1 是 v_2 的依赖。

通过将应用表达成计算流图,可以方便地描述计算之间的依赖关系,然后通过合适的调度算法可以大幅减少加速器中的控制,还可以提高计算并行度。因此在之前的异构多处理器任务映射算法研究中,基于计算流图的计算任务表示方式被大量运用。此外,在成熟的异构加速系统 TensorFlow 中也使用计算流图作为计算任务的表达形式。在本文的算法设计中,将基于计算流图来表达应用程序中的计算任务,并基于此来优化调度。

2 问题描述

在异构加速系统中,通过计算流图表达的应用程序,往往存在一些计算节点无法在加速器上执行。因为加速器通常是领域专用的,只会加速整个应用程序中的一部分。这导致在整个计算流图中存在"栅栏"[30-31]节点。所谓栅栏节点,就是在计算流图中,当前节点与其依赖节点支持运行的平台不同。这里只考虑主机(HOST)和加速器(设备)两种计算平台,因此,栅栏节点表示这些计算核节点没法在加速器上执行,而其他节点表示既可以在 CPU 上也可以在加速器上执行。所以,栅栏节点致使 HOST 和加速器设备被迫进行数据交互,从而带来通信开销,导致整个应用程序性能下降。在一些特殊应用场景中,计算流图中存在的栅栏节点可能会导致其依赖节点或者被依赖节点映射到加速器上执行反而无法带来性能增益。为了更清晰地说明这个问题,本文设计了如图 1 所示的计算流图。

图中给出了一个基于由 10 个计算核节点组成的有向无环图来表达的应用程序,并用黑色标识栅

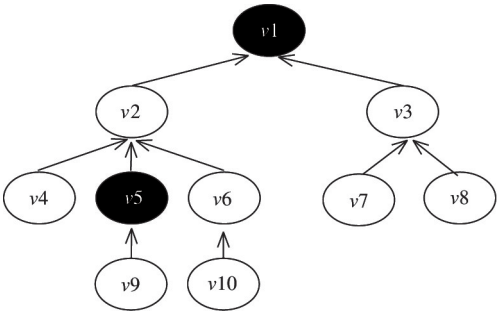

图 1 计算流图示例

栏节点。如图 1 所示,v_1 和 v_5 是栅栏节点,并且将是任务映射时需要特别考虑的点。因为,栅栏节点与其依赖节点存在 HOST 与加速器设备的数据交互,同时由于依赖关系,必须等待依赖节点执行完成才能开始下一步的计算,这将带来额外的通信开销从而降低整个应用程序的性能。为了降低栅栏节点对整个应用带来的性能影响,对其依赖节点完成高效的任务映射将是至关重要的。

用 C 表示计算流图中 2 个依赖节点之间的通信延迟的有限集合,如果 $c_1 \in C$,则称 c_1 是计算核节点 v_1 与被依赖节点之间的通信开销。假设,如果 2 个依赖节点映射的计算平台相同,则它们之间的通信延迟为零。用 S 表示计算核节点运行在加速器上时相对于 CPU 能够带来的性能增益的有限集合,如果 $s_5 \in S$,则称 s_5 代表节点 v_5 的性能增益。由于其无法在加速器上执行,令其等于零,即无性能增益。在图 1 中,v_9 节点如果要调度到加速器上执行,需要将结果数据从加速器上读回(假设待处理数据已经提前在平台准备好),这将带来额外的通信开销。很容易可以得出,如果 $s_9 < c_9$,节点 v_9 并不适合调度到加速器上执行。基于以上观察,首先,假设在计算流图中调度节点到加速器上执行带来的通信开销用 Φ 表示,节点在加速器上执行带来的增益用 X 表示。然后,给出如下推论:

在计算流图中,将节点映射到加速器上执行并不总能带来性能增益,假设只考虑通信开销,需要满足条件,即在加速器上执行的增益能够覆盖通信开销,即 $X \geqslant \Phi$。

图 1 中,在处理特定数据大小时,CPU 完全可以把处理数据全部放到内存中,访存不会成为 CPU

的性能瓶颈而导致 CPU 处理性能降低。因此,在接下来的讨论中,假设整个应用程序处理的数据规模是能全部装载到主机的内存中的。计算核节点 v_9,由于其被依赖节点 v_5 不能在加速器上执行,如果整个通信开销超过了加速器能够带来的性能增益,在执行调度的时候,如果能将 v_9 调度到 CPU 上来执行,对于提升应用的整体性能将是更优的调度策略。

在现有的异构加速系统中,运行时系统将识别加速可行性和决定如何利用加速器的责任转移给程序员,如 OpenCL 需要实现专门的计算核,CUDA 和 TensorFlow 提供了核函数库供用户调用,这对程序员提出了更高的要求。为了获得更高的性能需要,程序员需要了解加速器的底层细节,如加速器的存储体系和处理性能等。除此之外,这些特性都会随着应用运行时动态地改变,这极大地增加了调度优化的复杂度。

针对存在的问题,本文提出了基于预测的主动式任务映射算法,即通过预测算法动态调整计算节点的运行时属性,完成计算流图中任务在异构平台的映射。

3 PPTM 算法设计

在之前的工作中,面向多处理器的计算流图调度优化通常会考虑任务在处理器上的执行时间和通信开销[14-15],如果是实时任务还会考虑如何满足延迟要求,在此基础上来最大化应用程序性能。文算法设计只考虑计算核节点在加速器上执行所能带来的性能增益,用 X 表示,以及为此需要付出的额外开销。这些开销可以是数据通过 PCIe 在主机端和设备端的传输延迟以及程序额外编译时间等,统称为通信开销并用 Φ 表示。因此,PPTM 算法主要解决的问题可以抽象成,对于给定的应用程序 A 所表达的计算流图 $G=(V, E)$,根据节点在加速器上性能增益 X 和为了在加速器上执行所带来的通信开销 Φ,求得一个最优的图节点在异构平台上的映射方案 G',也即图节点的最优调度序列 $(v'_1, v'_2, \cdots, v'_n) \in V$,可以表示为

$G'_i=(v'_{i1}, v'_{i2}, \cdots, v'_{in})=f(G, X, \Phi)$,使得 $P(G'_i)=\max(P(G'_i))$,其中 $G'_i \in G'_i$,在这里,使用 P 函数去表示求一个计算流图的性能,f 函数表示根据给定的限制条件获得一个映射方案 G'。

同时,参考之前的工作,针对多处理器的任务映射问题通常是基于给定的处理器性能数据。然而,不管是处理器性能还是通信开销,这些都会随着任务的不同,而在运行时动态地变化。如果调度算法只根据历史给定的数据来进行任务映射,将无法适应实际的应用场景,如加速器性能在实际场景中没法全部发挥、PCIe 带宽实际上没有那么高等。因此,本文基于一元线性回归模型,针对计算节点在 CPU 和加速器上的执行时间以及节点之间的通信开销,分别实现了对应的预测模块。首先通过线性拟合给定的历史数据得到一个初步预测模型,然后会根据运行时节点性能数据,不断调整整个预测模型。相关模型如下:

$$\begin{cases} CX_i = b_{Ci}I + a_{ci} + \varepsilon_{ci},\ \varepsilon_{ci} \sim N(0, \sigma^2) \\ \sigma^2 = \dfrac{(CX_i - cx_i)^2}{n-2} \end{cases} \tag{1}$$

$$\begin{cases} AX_i = b_{Ai}I + a_{Ai} + \varepsilon_{Ai},\ \varepsilon_{Ai} \sim N(0, \sigma^2) \\ \sigma^2 = \dfrac{(AX_i - ax_i)^2}{n-2} \end{cases} \tag{2}$$

$$\begin{cases} C_i = b_{Ci}I + a_{Ci} + \varepsilon_{Ci},\ \varepsilon_{Ci} \sim N(0, \sigma^2) \\ \sigma^2 = \dfrac{(C_i - c_i)^2}{n-2} \end{cases} \tag{3}$$

其中,CX_i 代表计算核节点 i 在 CPU 上执行时的预测时间。假设节点 i 能在加速器上执行,则 AX_i 代表其在加速器上执行时的预测时间,相应的 cx_i 和 ax_i 表示实际的节点在 CPU 和加速器上的执行时间的测量值。I 代表节点处理数据的规模。C_i 则表示节点与依赖节点的通信开销的预测值,对应的 c_i 则为实际测量值。因此,对于计算节点在加速器上的性能增益,可以简单地表示为 $X_i = AX_i - CX_i$。

求解该问题的难点在于,修改某个节点的调度行为之后,会影响到其依赖与被依赖节点的性能,从而产生调度的传播性影响,因此获取一个最优的任务映射方案,将是一个 NP 问题。对此,本文给出一个基于性能增益传递的启发式的求解算法,也就是在决定节点是否映射到加速器上执行时,不只是根

据当前节点的性能增益,而是累计其相关子节点的性能增益来判断性能增益是否能覆盖通信开销。如果性能增益能覆盖通信开销则将其映射到加速器上执行,否则映射到 CPU 上执行。如果将其映射到 CPU 上执行,则接着需要递归重新调度其相关子节点。当然,如果在计算流图中,节点的输入是栅栏节点,在计算性能增益时将会考虑相关的通信开销。虽然,本文提出的求解算法是局部优化的,但是在考虑的应用中,栅栏节点作为一个分界点,栅栏节点上下游数据肯定来源于 CPU。因此,只考虑其父节点是否是栅栏节点,然后再来递归修改子节点的任务映射算法,也能求出一个比较全局优化的映射方案。

图 2 给出了 PPTM 算法的流程图,图中使用的性能数据来自于预测模块,为简洁起见图中没有画出预测模块。

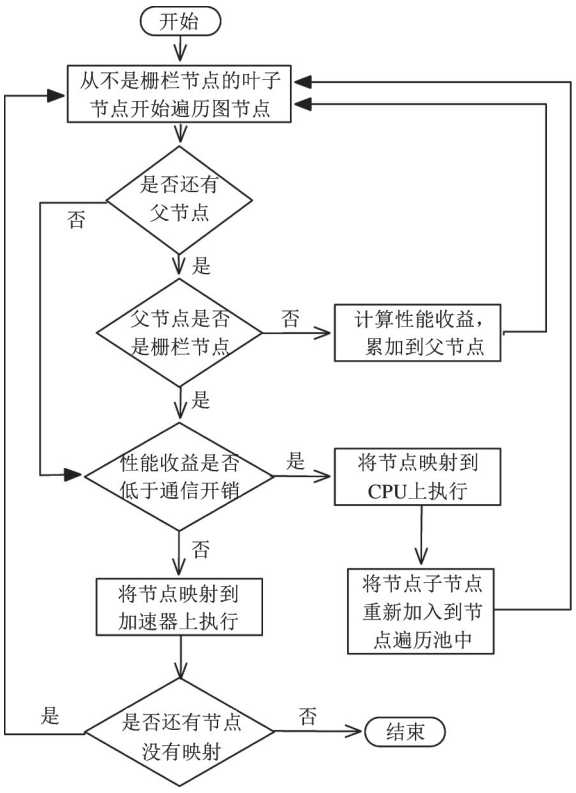

图 2　PPTM 算法流程图

如图 2 所示,从应用程序计算流图的非栅栏节点的加速器可执行的叶子节点开始遍历整个图节点。首先判断节点是否有父节点,如果没有,则说明调度来到了根节点,进入调度流程。否则,判断其父节点是否是栅栏节点,如果不是栅栏节点,将根据预测得到的加速器性能增益累加到其父节点的性能增益上,然后接着遍历图节点。否则,接着判断节点与父节点的通信开销是否高于性能增益,如果是,将节点映射到 CPU 上执行,并将其相关的子节点重新加入到节点遍历池中进行重新映射。否则,将节点映射到加速器上执行,接着判断是否整个图已经遍历完成,如果遍历完成则结束算法,否则继续算法执行。算法 1 给出了 PPTM 的伪代码实现。

算法 1　主动式任务映射算法

输入:计算流图 G,节点性能增益集 X, 通信开销集 Φ;
输出:完成了任务映射的计算流图 G'

1) *PPTM*(G, X, Φ);
2) $NS \leftarrow$*getNodeStack*(); /* 获得计算流图 G 的深度优先遍历栈 */
3) While NS is not NULL /* 当栈中还有节点时,继续进行调度 */
4) $Vi \leftarrow$*getNode*(NS); /* 从计算流图中获取一个节点 */
5) If root node of Vi is *NULL*
6) goto *Schedule*;
7) End
8) If root node of Vi is not fence node
9) $Xi \leftarrow$ *getSpeed*(Vi); /* 获取节点 Vi 的性能增益数据 */
10) *accXToRootNode*(); /* 将节点 Vi 的性能增益累加到其父节点上 */
11) *mapNodeToCPU*(Vi); /* 将节点 Vi 的映射到 CPU 上 */
12) Else
13) $Xi \leftarrow$ *getSpeed*(Vi); /* 获取节点 Vi 的性能增益数据 */
14) $\Phi i \leftarrow$ *getOverHead*(Vi); /* 获取节点 Vi 的通信开销数据 */
15) Schedule: If $Vi < \Phi i$
16) map*NodeToCPU* (Vi); /* 将节点 Vi 的映射到 CPU 上 */
17) set*SonNodeToStack*(Vi); /* 将节点 Vi 的子节点重新加入到遍历栈中 */
18) Else
19) map*NodeToACC* (Vi); /* 将节点 Vi 的映射到加速器上 */
20) End
21) End
22) End while
23) Return G'

4 示例研究

为了验证算法设计的有效性,并为将来面向加速器的运行时系统设计提供参考,本文将针对之前的工作,面向流式应用的加速器平台 ShuntFlow[3],也即面向计算核的处理器(kernel processing unit, KPU),设计和实现运行时系统(kernel operating system, KOS),并实现本文提出的任务映射方法。首先介绍 ShuntFlow 加速器的硬件结构,然后介绍整个 KOS 的架构以及相关实现细节。

4.1 核处理器 KPU

图 3 给出了 ShuntFlow(KPU)架构图[32]。KPU 加速器主要加速实现了一系列流式应用中的聚合算子,这些算子在流式数据挖掘、金融风控等领域被广泛应用[33],可以被看成是多个计算核,从而可以将 KPU 抽象成由多个计算核组成的加速器平台。表 1 给出了一个 KPU 加速器实现的计算核的相关信息。

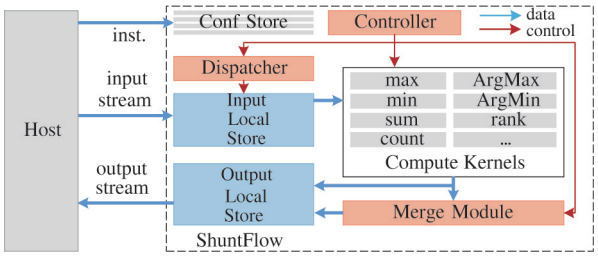

图 3 ShuntFlow(KPU)加速器架构图

表 1 KPU 计算核实现信息

名称	加速比	输入数
Max	30 ~ 50	2
Min	30 ~ 50	2
Sum	20 ~ 30	2
Add	20 ~ 30	2
Sub	20 ~ 30	2
Rank	70 ~ 90	1

在本文的实例中,KPU 总共实现了 6 类计算核,其中加速比是对比 CPU 不考虑数据传输延迟时,KPU 能够带来性能的提升,同时可以看出不同计算核能够实现的加速比是不一样的。输入数代表计算核需要几个输入数据,包括参数在内。

4.2 KOS 实现

首先,给出 KOS 针对 KPU 加速器的架构,在这里只给出支持 KPU 和 CPU 两个平台的实现,多平台的实现可以很方便地进行扩展。然后,将详细介绍针对 KPU 加速器怎么实现 PPTM 算法进行性能优化。最后,介绍了一些应用,用于下一节的测评。

图 4 给出了 KOS 针对 KPU 的架构图,接下来,将简单介绍一些模块的功能。

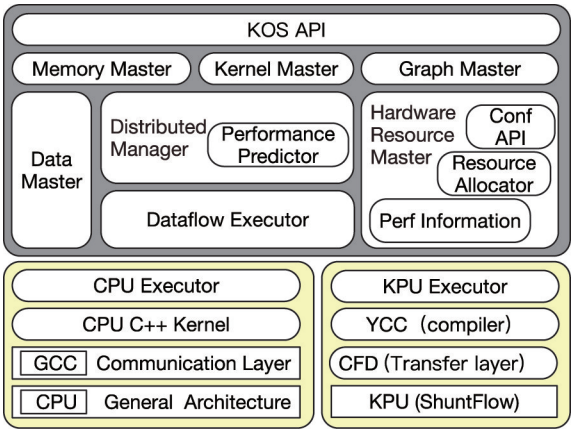

图 4 KOS 架构图

KOS API 主要包括计算流图构建、数据注册、计算核注册以及加速器相关属性注册等接口。针对 KPU 需要注册的硬件属性,包括特定数据大小传输开销和硬件上计算核实现信息(见表 1)等。

Memory Master 采用 BFC 算法来完成 KOS 内存管理,统一管理 KOS 中的内存分配,回收内存碎片,降低内存分配开销。

Kernel Master 主要负责接收注册的计算核信息,并提供给计算流图构建模块计算流图,并对节点进行标识。

Graph Master 主要负责计算核节点的创建和释放,并根据计算核节点负责管理计算流图。

Hardware Resource Master 提供加速器资源以及性能数据配置接口,并负责对加速器资源的分配和释放以及为性能预测器提供性能数据。

Data Master 负责管理源数据和结果数据,同时提供接口,可以在不同节点或者平台之间完成数据格式的转换。

Distributed Manager 主要负责根据平台信息

等完成计算流图的调度,同时为了优化调度实现性能预测器。针对 KPU,需要满足 KPU 能够支持的最大计算核并行执行的资源限制,同时能够根据注册的通信开销、计算核加速比等信息,针对应用不同的数据大小等完成性能预测。

Dataflow Executor 实现记分板算法,完成最小调度单位,计算流子图在不同平台的执行。

CPU&GPU Executor 负责针对特定的平台完成子图的解析,然后根据子图的依赖关系,并行或者串行调度计算核节点到特定平台完成计算并返回计算结果。

YCC&CFD 针对特定加速器平台 KPU 实现的编译器和驱动层,主要负责将计算核节点解析成平台二进制指令,然后通过特定的传输协议完成与加速器的交互。

4.2.1 性能优化——PPTM 算法实现

如前面 PPTM 算法的设计所述,PPTM 算法要完成计算节点的执行时间预测,不仅需要初始的性能配置数据,同时还需要获取任务的特定属性,如输入输出数据大小等,然后根据任务的运行时数据,才能提高性能预测的精度。因此,本文首先根据任务的输入数据大小,通过最小二乘法在初始的性能数据上,训练得到一个初始的节点性能预测模型。类似于执行时间预测,根据节点输出数据大小,在初始的通信开销数据上,训练得到一个初始的节点通信开销预测模型。然后,这些模型会在运行时,根据实际预测的测量值,不断调整整个模型来提高预测的精度。

基于以上获取的执行时间,根据之前描述的性能增益求取模型,可以很容易计算出计算节点的性能增益数据。然后,基于性能增益和通信延迟数据,KOS 将很容易实现 PPTM 算法,完成任务映射。首先,基于深度优先遍历获得一个节点栈,方便从计算流图中的叶子节点开始遍历整个图。然后,得到应用程序的计算流图 G 的一份拷贝 G',作为映射算法的输入。最后,实现之前给出的 PPTM 算法的伪代码,得到优化的计算流图 G''。

4.2.2 应用示例

本文选取金融量化投资领域的一些实际应用作为示例应用程序,当然这些计算核在风险控制以及流式数据聚合中也被广泛使用[34]。同时,使用现实世界的股票的股价(price)、每日开盘价(open)、收盘价(close)、交易量(volume)以及回报率(returns)作为数据源,通过调用相关计算核可以实现对股票的量化交易因子挖掘。参考量化投资领域中因子挖掘算法[32],构造了如下 5 个应用。

$$RANK\left(ADD\left(\begin{matrix}SUM\left(AVG\left(SUB\left(SUM(open,5),SUM(close,5)\right)\right),5\right),\\ ADD\left(MAX\left(SUM(returns,5),5\right),MAX\left(SUM(price,5),5\right)\right)\end{matrix}\right)\right),\quad \text{应用(1)}$$

$$RANK\left(ADD\left(\begin{matrix}RANK\left(AVG\left(MAX\left(SUM\left(ADD(open,close),5\right),5\right)\right)\right),\\ ADD\left(MAX\left(AVG\left(RANK(volume)\right),5\right),RANK\left(AVG\left(ADD\left(SUM(price,5),returns\right)\right)\right)\right)\end{matrix}\right)\right),\quad \text{应用(2)}$$

$$RANK\left(AVG\left(ADD\left(\begin{matrix}RANK\left(MAX\left(ADD\left(MAX(volum,5),SUM(returns,5)\right),5\right)\right),\\ RANK\left(AVG\left(ADD\left(AVG\left(MIN(open,5),MAX\left(RANK(close),5\right)\right)\right)\right)\right)\end{matrix}\right)\right)\right),\quad \text{应用(3)}$$

$$RANK\left(SUB\left(AVG\left(ADD\left(\begin{matrix}MAX\left(ADD\left(SUB(open,5),close\right),5\right),\\ MAX(returns,,5),\\ RANK\left(AVG\left(ADD\left(AVG\left(RANK(volume)\right),SUM\left(RANK(price),5\right)\right)\right)\right)\end{matrix}\right)\right)\right)\right),\quad \text{应用(4)}$$

$$MAX\left(ADD\left(MIN\left(AVG\left(SUB\left(RANK\left(ADD\left(\begin{matrix}MAX(volume,5),\\ MAX(close,5)\end{matrix}\right)\right),AVG\left(RANK\left(MIN(price,5)\right)\right)\right)\right)\right)\right)\right),\quad \text{应用(5)}$$

以上的 5 个应用,是本文通过抽取量化投资领域中因子挖掘算法[32]的常见操作,然后抽象成计算核,最后对计算核进行随机组合而成。其中 AVG 是求平均数,在加速器上没有实现,本文会在 CPU 上给出其实现,因而其是栅栏节点。根据 KOS 计算流图的定义,图 5 和图 6 给出了应用(4)和应用(5)的计算流图形式,其他应用的计算流图可以很容易类比得到。

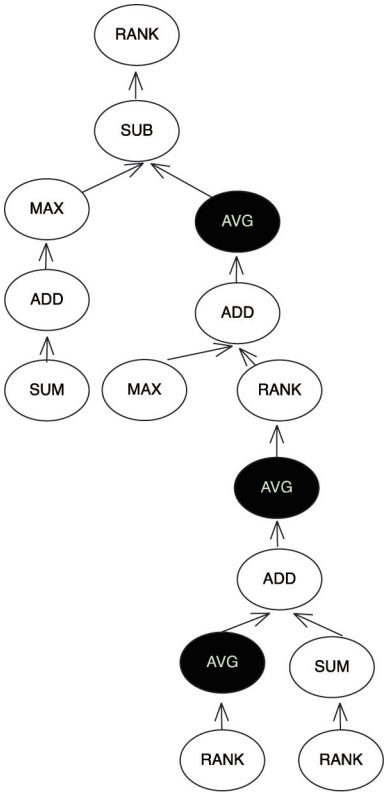

图 5　应用(4) KOS 计算流图

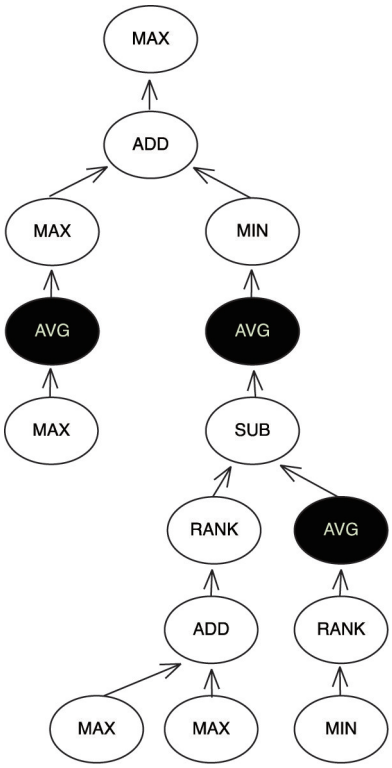

图 6　应用(5)计算流图

如图中所示,由于 AVG 无法在加速器上加速,显示为黑色节点。限于篇幅,这里只给出了 5 个应用示例,来证明本文系统的高效性。具体的分析将在下一节进行详细描述,其结论可以简单地推广到其他应用中。

5　测评与分析

5.1　测试环境与测试方案

5.1.1　测试环境

KOS 的测试是在一台服务器上完成的,加速器 KPU 通过 PCIex8 接口连接。加速器实现在 DE5a-Net 的 FPGA 板卡上,综合频率为 150 MHz。服务器的配置如表 2 所示。KOS 的测试用例采用前面描述的 5 个应用。

表 2　测试环境参数表

	部件	型号
硬件	CPU	IntelXeon E5-2634 v4 2.2 GHz
	内存	Samsung DDR3 1333 MHz, 32 GB
软件	OS	CentOS 6.3 64 位
	Kernel	LinuxKernel 2.6.32

5.1.2　测试方案

PPTM 的设计目的是通过运用提出的主动式任务映射算法,能够根据系统的运行时数据动态地调整计算流图中计算节点的映射关系,从而提高整个应用程序在异构加速系统中的性能。因此,本文总共实现了 3 种执行模式,如下所述。

Accelerator-Free(A-Free)模式　应用程序不使用加速器进行加速,而是直接将计算流图中的所有节点全部映射到 CPU 上执行,做为基础的对比数据。

DirectMapping(D-Map)模式　采用传统的任务映射方法,即在映射应用程序计算流图中节点时采用加速器优先映射,也就是只要加速器支持的计算节点都映射到加速器上执行,来做为主要的对比算法实现。

PPTM 模式　将运用提出的任务映射算法,来调度计算流图在异构加速系统中的执行方式。

基于以上的 3 种执行模式,在不同的浮点数据集(data1-32M,data2-320M)上对不同的应用进行了多次实验,并统计了相关性能数据。

5.2 数据分析

5.2.1 性能分析

表3和表4分别给出了在不同数据集上,5个应用在不同执行模式下的性能数据。

表3 数据集1下应用性能数据

	A-Free /μs×10^5	D-Map /μs×10^5	PPTM /μs×10^5	D-Map对比A-Free提升	PPTM对比D-MAP提升
应用(1)	2.3406	1.5414	1.3246	34.15%	14.06%
应用(2)	2.8528	2.2193	2.1282	22.21%	4.10%
应用(3)	2.9908	2.1404	2.0624	28.43%	3.64%
应用(4)	2.8701	2.1995	2.1475	23.36%	2.37%
应用(5)	2.7307	2.0780	1.9876	23.90%	4.35%
均值	2.7570	2.0375	1.9301	26.41%	5.70%

表4 数据集2下应用性能数据

	A-Free /μs×10^6	D-Map /μs×10^6	PPTM /μs×10^6	D-Map对比A-Free提升	PPTM对比D-MAP提升
应用(1)	2.3311	2.0095	1.5674	14.15%	22.00%
应用(2)	2.7988	2.8955	2.6744	3.19%	7.63%
应用(3)	2.9218	2.7385	2.6579	8.43%	2.94%
应用(4)	2.7991	2.8364	2.1316	5.16%	24.85%
应用(5)	2.7267	2.7021	2.4871	1.05%	7.96%
均值	2.8088	2.6364	2.3037	6.40%	13.80%

从表中可以看出,在数据集1上,通过D-Map映射方式,将计算流图中的节点调度到加速器上执行可以带来巨大的性能增益,5个应用平均性能提升了26.41%。但是在数据集2上时,随着数据规模的加大,主机与加速设备的通信开销变大,采用D-Map的任务映射方法,对某些应用,带来性能增益有所下降,特别是对应用(5)。从图5中可以看到,应用(5)中左子树上的栅栏节点AVG所依赖的子节点,将其映射到加速器上执行,根据提出的算法,其累积的性能收益相比于其他栅栏节点偏少,如果将其调度到加速器上执行,带来的性能增益甚至不能抵消通信开销,从而导致整个应用程序性能增益下降。本文实验中,在数据集2上,5个应用通过D-Map映射方法在异构加速平台上带来的平均性能增益不足7%。

从表中可以看到,通过PPTM任务映射算法,在不同的数据规模时,对比D-Map映射方式都能带来性能增益。在数据集1上,通信延迟还比较小,因此对于大多数应用通过D-Map映射方法就能够带来很好的性能增益。本文提出的PPTM算法,相比于D-Map模式,带来了2.37%~14.06%的性能提升。而在数据集2时,随着通信延迟的增加,PPTM任务映射算法,相比于D-Map映射方式,平均能够带来13.8%的性能提升。特别是针对应用(4),从图4可以看到,栅栏节点存在于叶子节点到根节点的关键路径上,尤其是最下面的2个栅栏节点之间只隔了1个节点。如果按照D-Map的任务映射方法,将这个节点调度到加速器上执行,将导致主机与加速设备之间反复进行数据通信,从而带来严重的通信开销。因此,通过PPTM任务映射算法,将其映射到CPU上执行,反而带来了24.85%的性能提升。

通过以上的数据分析可以看出,针对不同的应用程序,并不是将其计算流图中加速器支持的所有节点全部调度到加速器上执行,就一定能获得最佳的性能。因为针对某些应用,如果直接把节点映射到加速器上,会导致无法摊销的主机与加速设备之间的数据交互,从而带来严重的通信开销,反而无法提升整个应用程序的性能。因此,需要根据特定性

能数据,合理地进行任务映射,才能最大化应用程序在异构加速系统上的性能。为了进一步分析通信开销对性能的影响,接下来,将对应用在不同的数据集上的通信开销数据进行分析。

5.2.2 通信延迟分析

图 7 和图 8 分别给出了 D-Map 模式和 PPTM 模式在不同的数据集下,应用由于存在主机和加速器端的数据交互而带来的通信开销数据。图中,D-Map-C代表应用在 D-Map 模式下的通信开销,PPTM-C 则代表应用在 PPTM 模式下的通信开销。

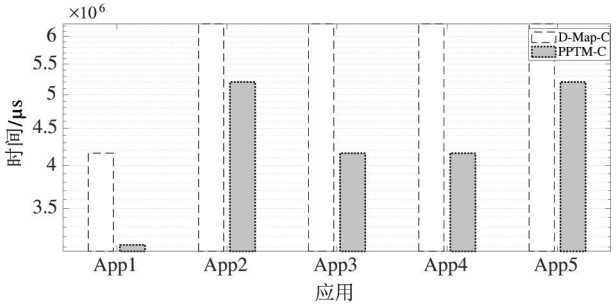

图 7 数据集 1 下应用通信开销数据对比

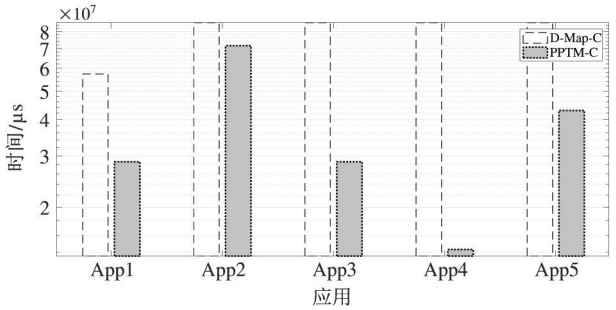

图 8 数据集 2 下应用通信开销数据对比

从图中可以看到,PPTM 任务映射算法,在不同的数据集上,其通信开销相比于 D-Map 映射方法都要低。而在本文的假设中,通信开销只存在于栅栏节点的上下节点需要数据交互时。因此,必然是 PPTM 任务映射算法改变了与栅栏节点相连的某些节点的映射方式。

从图 7 中可以看到,在数据集 1 上,PPTM 算法相比于 D-Map,平均降低了 24.9% 的通信开销。但是,从之前的性能分析可知,其带来的性能增益平均不到 6%。因为 PPTM 虽然降低了应用程序的整体通信开销,但同时也会失去将某些节点映射到加速器上执行带来的性能增益,这需要根据应用的运行时数据做出权衡。

从图 8 可以看出,当在数据集 2 上时,PPTM 算法相比于 D-Map,平均降低了 53.3% 的通信开销。同时,之前的性能分析表明,PPTM 带来了平均 13.8% 的性能增益,特别是应用(4),带来了 24.85% 的性能提升。从图 7 中可以看到,针对应用(4),PPTM 算法对比于 D-Map,降低了将近 83% 的通信开销。然而,对于应用(3),PPTM 算法降低了 66.67% 的通信开销,但性能增益微乎其微。

通过上面的分析可以看到,PPTM 算法通过改变栅栏节点子节点的映射方式必然会降低整个应用程序的通信开销,但并不是通信开销降低得越多带来的性能增益越高。除了因为改变节点映射方式后无法受益于加速器带来的性能增益,性能增益还跟具体的应用存在很强的关联性。

6 结 论

现有异构加速系统中,将计算通过加速设备来加速,在需要与主机端进行交互的应用场景中会带来相应的通信开销。如果无法合理地摊销主机与设备端的通信开销,在特定的数据规模时(CPU 性能瓶颈不在访存时),针对不同的应用,并不是将其计算流图中的加速器支持的所有节点全部调度到加速器上执行,就一定能获得最佳的性能。通过引入 PPTM 算法,在运行时动态地感知应用中计算节点的性能变化,实现更加高效的任务映射,能够提高应用程序的整体性能。最终测试充分验证了,针对不同的应用程序,本文提出的任务映射算法,能够实现更加高效的任务调度,从而提高应用程序性能。

参考文献

[1] HENNESSY J, PATTERSON D. A new golden age for computer architecture[J]. *Communication of the ACM*, 2019, 62(2): 48-60

[2] 鄢贵海,李晓维,孙凝晖. 软件定义加速器设计方法初探[J]. 中国计算机学会通讯, 2018, 14(11): 58-63

[3] 包云岗,张科,孙凝晖. 处理器芯片开源设计与敏捷开发方法思考与实践[J]. 中国计算机学会通讯, 2019,

15(10): 42-48

[4] CHEN Y T, CONG J, GILL M, G, et al. Customizable Computing[M]. Los Angeles: Morgan & Claypool Publishers, 2015:1-5

[5] JOUPPI N P, YOUNG C, PATIL N, et al. In-datacenter performance analysis of a tensor processing unit[C]// Proceedings of the 44th Annual International Symposium on Computer Architecture, New York, USA, 2017:1-12

[6] JOUPPI N P, YOON D B, KURIAN G, et al. A domain-specific supercomputer for training deep neural networks [J]. *Communication of the ACM*, 2020, 63(7): 67-78

[7] Kirkpatrick K. Software-defined networking[J]. *Communication of the ACM*, 2013, 56(9):16-19

[8] RUPP K. The OpenCL library ecosystem: current status and future perspectives[C]// Proceedings of the 4th International Workshop on OpenCL, New York, USA, 2016:25-26

[9] Nvidia. CUDA toolkit release notes[EB/OL], https://docs. nvidia. com:NVIDIA,[2020-10-25]

[10] KHRONOS. Khronosopenvxregistry[EB/OL]. https://khronos. org:Khroonos,[2020-10-25]

[11] AMD. AMD OpenVX[EB/OL]. https://github. com/: GitHub,[2020-10-25]

[12] ABADI M, BARHAM P, JIANMIN C, et al. TensorFlow: a system for large-scale machine learning[C]// Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, Berkeley, USA, 2016: 256-283

[13] WANG Y, LIU D, QIN Z W, et al. Optimally removing intercore communication overhead for streaming applications on MPSoCs[J]. *IEEE Transacctions on Computers*, 2013, 62(2): 336-350

[14] YI W, SHAO Z, HENRY C B C, et al. Memory-aware task scheduling with communication overhead minimization for streaming applications on bus-based multiprocessor system-on-chips[J]. *IEEE Transactions on Parallel and Distributed Systems*, 2014, 25(7): 1797-1807

[15] TOPCOGLU H, HARIRI S, MINYOU W. Task scheduling algorithms for heterogeneous processors[C]// Proceedings of the 8th Heterogeneous Computing Workshop, Puerto Rico, USA, 1999: 3-14

[16] YAN W, KENLI L, HAO C, et al. Energy-aware data allocation and task scheduling on heterogeneous multiprocessor systems with time constraints[J]. *IEEE Transactions on Emerging Topics in Computing*, 2014, 2(2): 134-148

[17] 赵瑞姣,朱怡安,李联. 基于异构多核系统的混合关键任务调度算法[J]. 计算机工程, 2018, 44(2): 51-55

[18] 赵国亮,李云飞,王川. 异构多核系统任务调度算法研究[J]. 计算机工程与设计, 2014, 35(9): 3099-3106

[19] 顾钧. 异构计算系列文章(一):定义、场景及局限性[EB/OL]. https://www. infoq. cn/: InfoQ, (2020-03-28),[2020-10-25]

[20] 科普中国. OpenCL 科学百科词条[EB/OL]. https://baike. baidu. com/item/OpenCL/8477301? fr = aladdin: Baidu,(2020-04-14),[2020-10-25]

[21] Google. TensorFlow 指南[EB/OL]. https://tensorflow. google. cn/guide? hl = zh _ cn:Google,[2020-10-25]

[22] KOFLER K, GRASSO I, COSENZA B, et al. An automatic input-sensitive approach for heterogeneous task partition[C]// Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, New York, USA, 2013:149-160

[23] KARKOWSKI I, CORPORAAL H. Design space exploration algorithm for heterogeneous multi-processor embedded system design[C]// Proceedings of the 35th Annual Design Auttomation Conference, New York, USA, 1998: 82-87

[24] PLANAS J, BADIA R M, AYGUADE E, et al. SSMART: smart scheduling of multi-architecture tasks on heterogeneous systems[C]// Proceedings of the 2nd Workshop on Accelerator Programming Using Directives, New York, USA, 2015:1-11

[25] PECCERILLO B, BARRTOLINI S. Single-source library for enabling seamless assignment of data-parallel task-DAGs to CPUs and GPUs in heterogeneous architectures [C]// Proceedings of the 10th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms, New York, USA, 2019:1-4

[26] SUN E, SCHAA D, BAGLEY R, et al. Enabling task-level scheduling on heterogeneous platforms[C]// Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units, New York, USA, 2012:84-93

[27] RARAVI G, NELIS V. Task assignment algorithms for heterogeneous multiprocessors[J]. *ACM Transactions on Embedded Computing Systems*, 2014, 13(5): 159-185

[28] 裴颂文,吕春龙,宁钟,等. 异构多核计算系统的 Codelet 任务调度策略[J]. 计算机应用研究, 2019, 26(5): 1433-1436,1440

[29] 郭辉,王智广,周敬利. 异构分布式系统中基于负载均衡的容错调度算法[J]. 计算机学报, 2005, 28(11): 1807-1816

[30] CSDN 博主. 几分钟理解和使用 BFC[EB/OL]. https://blog.csdn.net:CSDN, (2019-08-02),[2020-10-25]

[31] GOETZ B, PEIERLS T, BLOCH J, et al. Java Concurrency in Practice[M]. 北京:机械工业出版社, 2012: 83-84

[32] GONG S J, LI J J, LU W Y, et al. ShuntFlow: an efficient and scalable dataflow accelerator architecture for streaming applications[C]//Proceedings of the 56th Annual Design Automation Conference 2019, New York, USA, 2019: 1164-1169

[33] KAKUSHADZE Z. 101 formulaic alphas[J]. *Wilmott Magazine*, 2016, 84: 72-80

[34] MUELLER R, TEUBNER J, ALONSO G. Streams on wires: a query compiler for FPGAs[J]. *VLDB Endowment*, 2009, 2(1): 229-240

PPTM: a proactive task mapping method for heterogeneous systems

GONG Shijun[*,**], YAN Guihai[*], LI Xiaowei[*]

([*] State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

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

Abstract

With the rapid growth of data, heterogeneous computing has emerged as an effective way to meet the ever-increasing computing power demands of emerging applications, and many heterogeneous acceleration systems have emerged. In these heterogeneous acceleration systems, efficient task mapping is one of the keys to making full use of the accelerator's potential to improve application performance. In the previous work, many efficient task mapping methods have emerged based on directed acyclic graphs to minimize the overall execution time of the application and minimize the communication overhead between heterogeneous multiprocessors. In these work, it is usually assumed that mapping tasks to accelerators will definitely bring performance benefits to the entire application. However, it was found that if some applications map all subtasks to the accelerator for execution, it will bring additional communication overhead, which may not meet expectations for performance improvements, and even cause the performance of the entire application to decrease. Therefore, a prediction-based proactive task mapping algorithm (PPTM) is proposed to deal with such scenarios and achieve efficient task mapping. Experimental results show that the prediction algorithm can more accurately perceive the runtime state of computing tasks, and then through the mapping algorithm, the whole performance of the application can be greatly improved.

Key words: heterogeneous computing, heterogeneous accelerating system, task mapping, proactive, prediction algorithm, accelerator