高技术通讯 2018 年 第 28 卷 第 11 – 12 期:885 ~ 898

doi:10.3772/j.issn.1002-0470.2018.11 – 12.001

基于网络负载特征感知的数据流指令调度机制研究①

冯煜晶②*** 欧 焱*** 叶笑春③* 范东睿*** 谭 旭*** 唐志敏***

(*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)

(**中国科学院大学计算机与控制学院 北京 100049)

摘 要 研究了数据流架构的指令调度策略,基于当前普遍采用的数据流指令调度机制,提出了支持模式切换的动态指令调度机制。由于数据流架构执行模式具有并行化特点,同一时刻存在大量并行传递的数据和并行的计算,网络传输负载呈现出非均匀的分布模式。局部网络传输压力过大导致数据流节点中的处理单元内部出现流水线停顿,片上网络(NoC)的局部传输效率降低,从而影响数据流架构的网络传输延迟、计算部件的利用率和整体的执行效率,因此针对原有的指令调度策略提出改进方案。针对网络负载的动态变化实时调整指令调度策略,从而达到缓解网络局部拥塞,提高网络传输效率的目的。本研究使用数据流模拟器对提出的机制进行验证,实验结果表明,采用本文提出的指令调度机制,数据流网络的传输延迟平均降低了 12.8%,计算部件的利用率平均提高了 14.4%,数据流架构的整体性能平均提高了 14.7%。

关键词 数据流架构,动态指令调度,片上网络(NoC),网络负载,单元利用率

0 引 言

科学计算是计算机负载应用中不可缺少的一部分,现实生活中的许多研究和探索都需要通过计算模拟,比如天体运动、全球气候模拟、DNA 测序等。这些应用都需要数以 E 级的数据作为计算基础,并且要求计算系统能够高并发、强实时地处理计算请求。传统的冯诺依曼结构由于控制逻辑复杂、“存储墙”等原因在计算大规模数据的时候展现出诸多不足。数据流体系结构作为专用计算结构的一个重要分支,在科学计算应用中发挥了越来越重要的作用[1-4]。数据流体系结构能够极大地挖掘指令之间的并行性,从而达到较高的计算并行度和能效比[5]。因此,相比于传统的冯诺依曼结构,数据流架构在科学计算领域能够取得更高的执行性能[6]。

数据流架构从执行控制模式上分为两种类型,一种是在数据流图的块内部通过程序控制寄存器来控制块内指令执行的顺序,流图中块与块之间采用数据流的方式执行,该方式称为操作触发的控制模式[7,8]。另一种叫做数据传输触发模式,当一条指令的所有源操作都准备好的条件下该条指令可以被触发执行[9-11]。数据传输触发模式摒弃了第一种方式并行度低下的弊端,有利于充分发挥数据流架构的指令级并行和线程级并行的执行特点,在数据流结构中被广泛使用[12-14]。在采用数据传输式触发的结构当中,指令调度决定了应用负载中的指令级并行度能否被充分挖掘[15]。指令调度可以采用硬件或者软件的方式来完成。硬件调度的方式在超线程处理器当中被广泛采用,该方式可以有效地检测

① 国家重点研发计划(2017YFC0803401),国家自然科学基金(61332009,61732018)和计算机体系结构国家重点实验室创新课题(CARCH3303, CARCH3407, CARCH3502, CARCH3505)资助项目。

② 女,1984 年生,博士;研究方向:计算机系统结构,异构加速系统;E-mail: fengyujing@ict.ac.cn

③ 通信作者,E-mail: yexiaochun@ict.ac.cn

(收稿日期:2018-03-15)

指令之间的依赖关系和资源冲突,但缺点是可扩展性差,随着指令窗口的增大,该机制所需的硬件开销也随之线性增加。数据流架构大都采用软件的方式来实现指令的静态调度。编译器负责完成指令在数据流处理单元阵列的调度,满足指令之间的传输距离最短和指令的并行度达到最优的要求。但是静态编译在适应实时动态场景方面相比硬件调度器又不够灵活。于是,Ramadass 等[16]提出采用静态调度和动态发射相结合的执行模式,编译器完成指令到处理单元阵列的静态调度映射,硬件负责实施指令发射到执行单元的操作。采用该方式,不仅有利于充分发挥数据流架构的执行效率,同时也保持了动态控制的灵活性和可扩展性。在数据流架构中,除了调度机制之外,另外一个对能效和性能影响较大的因素是数据流互连网络的延迟。如 Shen 等人[17]针对数据流互连网络的分析结果表明,数据流网络具有高注入率以及对延迟敏感的特征。一方面,互连网络的连通程度和总线套数影响了数据流处理器的功耗,功耗与能效指标直接相关;另一方面,互连网络的数据传输效率直接影响数据流结构的性能。如果数据在互连网络上的平均传输延迟增大 3 倍,数据流的平均性能会下降 2 倍,所以互连网络的拥塞状况越少越好。由此可见,指令调度策略和互连网络对于数据流架构的性能都是不可忽视的影响因素。但是,目前常用的数据流架构的调度算法大都没有考虑互连网络的数据传输效率对于整体性能的影响,而是只考虑了负载均衡、数据的传输距离和指令并行度的因素。经过对数据流负载分析后发现,编译器静态调度后的指令在动态执行过程中,在互连网络中会出现局部负载不均衡或者局部拥塞的情况。实验结果表明,这种情况会直接导致数据传输效率低,从而影响到数据流处理单元的执行效率。

本研究基于一款数据流架构,分析数据流互连网络的负载分布对于性能的影响,并且提出一种优化互连网络传输效率的硬件调度机制。该机制的硬件开销小,有助于提高互连网络的传输能力,减少局部拥塞的发生。在不改变数据流互连网络资源的条件下,可以有效地提高数据流架构的部件利用率和性能。

本文的结构如下,第 1 部分对数据流架构和执行模式进行介绍;第 2 部分讨论了相关工作,结合实验结果和当前研究现状提出了研究动机;第 3 部分提出基于网络负载特征感知的调度机制,并进行详细地介绍;第 4 部分介绍了实验方法和实验平台;第 5 部分针对实验结果进行分析;最后在第 6 部分对全文进行总结。

1 数据流执行模型

本研究采用的数据流执行模型面向科学计算类的应用,其结构如图 1 所示。该数据流执行模型由主处理器核(host core)、片外存储(memory)、直接内存存取部件(direct memory access, DMA)和数据流加速阵列(accelerator)组成。操作系统和主程序都在主处理器核上运行,数据流处理部件是该系统的加速器,用于加速计算密集型的并行程序段。该数据流加速器包括由 16 个处理单元(processing element,PE)组成的阵列、片上暂存存储部件(scratch PAD memory, SPM)、片上网络(network-on-chip, NoC)以及处理阵列控制部件(micro control)。每个处理单元中包括指令存储部件(instruction slots)、指令发射部件(fire selection)、指令执行部件(execution pipeline)和数据打包部件。指令存储部件用于保存在处理单元当中执行的指令译码信息。指令执行部件包括定点执行部件和浮点执行部件,支持流水执行方式。数据打包部件把流水线的执行结果打包为网络传输数据包的格式,并且把数据包发送到与处理单元相邻的路由装置当中。PE 阵列在执行过程中产生的操作数、控制信号和访存消息通过 NoC 传递。SPM 保存指令和数据 2 种信息,可以同时支持来自 PE 阵列和外部 DMA 的访存请求。处理阵列控制部件负责控制数据流加速器的整体运行,包括指令的分发和执行过程中的循环控制。

该执行模型对应的数据流指令格式如图 2 所示。数据流指令包括如下字段:操作类型、操作数个数、目标处理单元的地址。每条指令最多指定 4 个目标地址。每个目标地址包含 3 类信息,分别是目标指令所在 PE 的坐标、指令编号和操作数编号。

通过这 3 类信息可以准确定位到某个 PE 单元的某条指令的某个操作数。

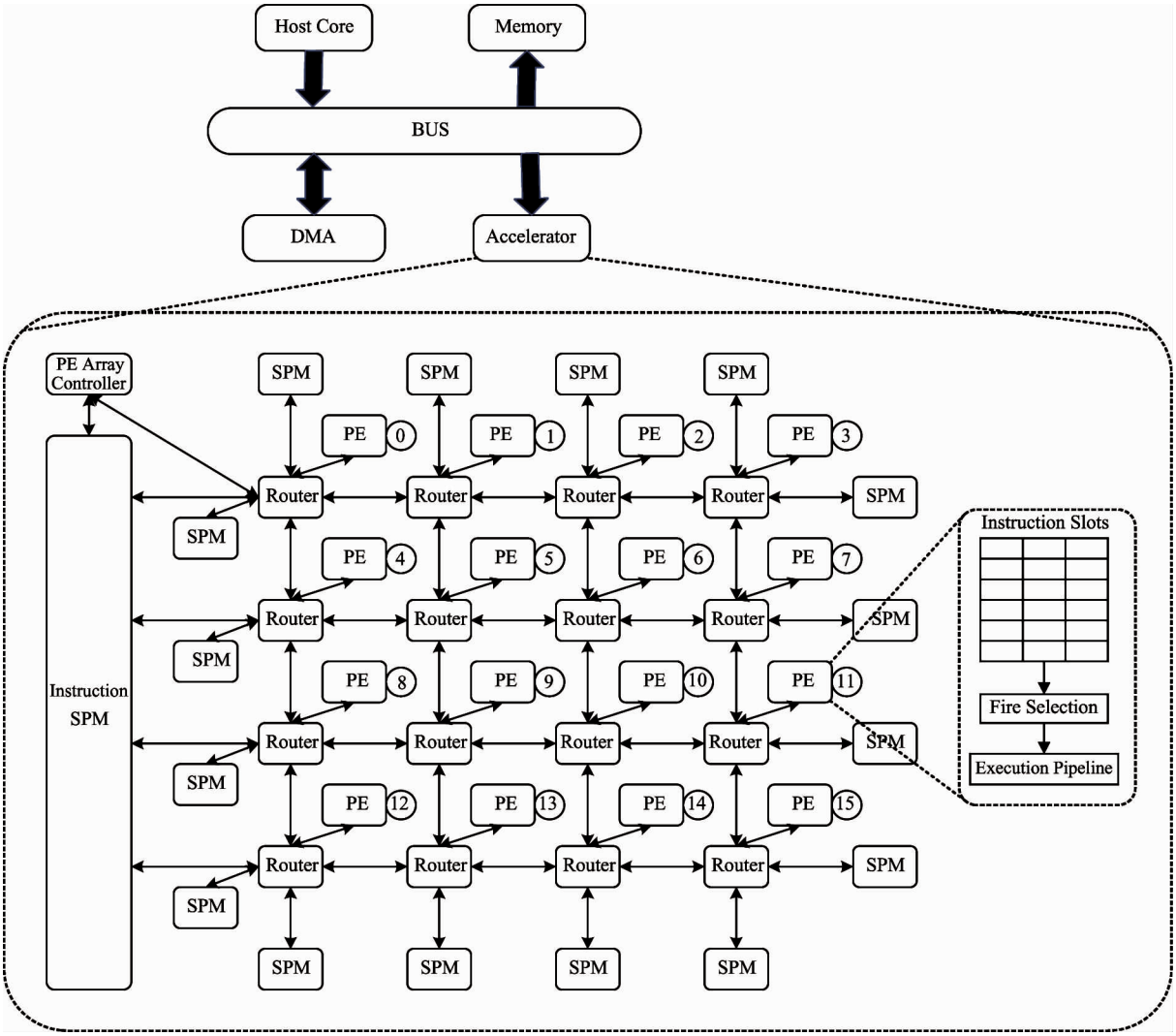

图 1　数据流系统示意

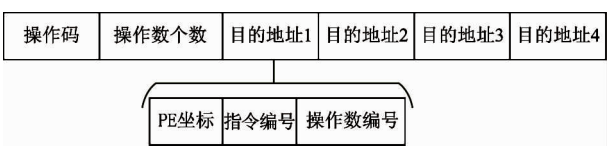

图 2　数据流指令格式

数据流结构的执行特点是指令与指令之间直接通信,不需要通过共享的命名空间(如寄存器堆)交换数据。指令按照数据流图的顺序执行,每条指令的操作数在指令码中直接采用物理位置表示,一旦指令的所有源操作数都准备完毕,该指令就可以发射执行。

本研究采用的数据流加速器将分别采用 3 种不同的调度算法将数据流程序静态调度到各个处理单元上。处理单元阵列在 1 次执行过程中只能够存放 1 个数据流图,每次循环可以针对不同的数据块进行相同的操作,每一次针对 1 个数据块的数据流图操作叫做 1 个上下文。图 3 展示了 1 个指令映射的示例,图中左边展示了 1 个程序中指令的依赖关系,右边是各个指令映射到 PE 阵列上一种可能的形式。

大量的科学计算应用都可以将一个整体问题拆解成若干个子问题,子问题之间相互独立,没有依赖相关或者依赖距离可控。目前普遍采用中央处理器(central processing unit, CPU)或者图形处理器

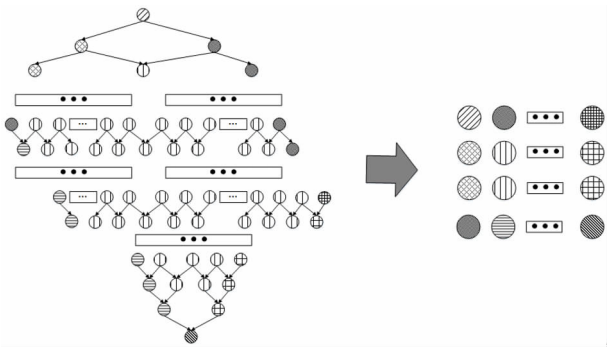

图 3 指令映射示例

(graphics processing unit, GPU)对科学计算类的应用进行加速。CPU 采用通用的冯诺伊曼结构,通过程序计数器确定当前执行的指令以达到控制指令执行顺序的目的。该执行方式无法掩盖计算单元中等待操作数的延迟,导致计算部件的利用率低,造成计算资源的浪费。GPU 在执行科学计算应用时,将一个整体的大任务分成若干个子任务,相比于 CPU 大大提高了并行执行效率[18]。但是通常 GPU 即使采用先进的生产工艺,其功耗和面积的开销都很大,无法达到较好的能效比。

在本研究采用的数据流执行模型中,将整个操作数空间拆解为多个相互独立的数据块,多个数据块可以被计算节点阵列并行连续地处理,形成流水迭代的执行模式。在每个计算节点内部,一条指令完成操作数空间的某一次迭代的执行之后,立刻清空等待下一次迭代操作数的到来。单次迭代的代码称为计算核心(kernel),不同子空间的求解对应相同的计算核心,向同一计算核心依次输入整体问题空间的索引变量,即可依次完成各子空间的计算,从而解决原问题。数据直接在指令之间传递,能够减少共享存储的访问次数[19],有效地掩盖数据传输延迟。该执行模式能够提高计算部件的利用率和系统的能效比。

本研究实现了 6 种科学计算场景的内核,分别在 CPU、GPU 以及数据流加速器模型进行测试。6 种内核分别是:FFT(快速傅里叶变换),2D Stencil,3D Stencil,GEMM(稠密矩阵乘法),LBM(格子玻尔兹曼方法)和 SAD(绝对误差和)。其中,FFT 的 CUDA(compute unified device architecture,统一计算设备架构)代码采用英伟达公司提供的 cuFFT 标准库实现,cuFFT 是经过高度优化的 CUDA 实现。其他几种内核的 CUDA 代码和 C ++ 代码都来源于 Parboil 标准测试集当中的优化版本[19]。实验使用的 GPU 的配置参数为 Nvidia Titian XP 2017, 12 GB GDDR5X,单精度浮点峰值性能为 12. 1TFLOPS (1/32 FP64),16 nm FinFET 工艺,热设计功耗为 250 W。实验使用的 CPU 的配置参数为 Intel Xeon E5-4627V2,主频 3.6 GHz, 22 nm 工艺,热设计功耗为 130 W。

此外,本研究还使用硬件编程语言 Verilog 实现了数据流加速器,并采用 TSMC 的 40 nm 工艺库和 Synopsys 公司的 EDA 软件评估其功耗和面积的开销。经过测算,数据流加速器的峰值功耗为2.8 W。实验结果如图 4 所示,针对科学计算类的应用,本研究所采用的数据流加速器的能效高于 CPU 和 GPU,比 CPU 高 3 699.99 倍,比 GPU 高 10.0 倍。本研究采用的数据流加速器相比于高性能 CPU 和 GPU 而言,消耗更低的能量达到同样的运算性能。

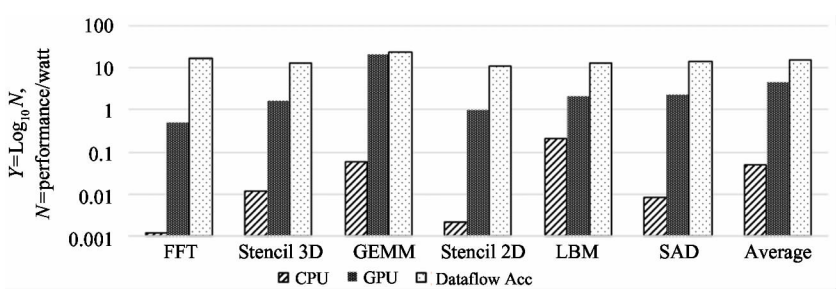

图 4 CPU、GPU 和数据流加速器的性能功耗比对比示意图

2 相关工作与研究动机

由于数据流架构的指令通信方式对软件可见,传统的数据流架构一般采用静态调度的方式。该方式在把程序编译为数据流图的阶段,综合考虑计算负载的均衡、指令间的通信距离和运算单元之间的冲突等因素,以得到优化的静态调度结果。Ramadass 等[16]提出了针对数据流架构的静态调度动态发射的执行模式(static placement dynamic issue,SPDI)。SPDI 依赖编译器为每条指令选择其对应的执行单元,在动态执行过程中如果 1 条指令对应的源操作数都准备好之后,由硬件完成指令的动态发射。SPDI 的静态调度器需要在多种优化目标当中寻找最佳的调度策略,使得指令之间的通信延迟最小化、运算单元的冲突最小化、指令密度尽量大以及使得发射窗口内的指令最大化。SPDI 虽然能够很好地提高单个处理单元的执行效率,但是缺乏对计算阵列全局因素的考虑,比如片上网络的拥塞和网络负载分布,造成局部计算效率高但是整体协同较差的情况。Katherine 等[20]针对数据流架构提出一种基于空间路径的指令调度方法(spatial path scheduling, SPS)。SPS 算法采用执行延迟加上路由距离共同表示指令调度的开销。开销越大,表示对应的调度状态对性能的负面影响越大。在起始阶段,每条指令对应若干潜在的调度状态,SPS 分别针对每种调度状态计算对应的调度开销。调度算法根据指令对应的开销值进行决策。SPS 调度算法相比于 SPDI 调度算法的优点是并不要求指令的父节点全部被调度完成之后才可以对其进行调度,而是根据事先确定的锚指令对指令的路由延迟和开销进行计算。SPS 虽然增加了路由距离的因素,但是该信息仍然是静态的,不能够很好地适应数据流的动态执行场景。申小伟等[21]提出面向科学计算数据流结构的 LBC(load balance centric)指令调度算法,以运算单元的负载均衡为核心,将每个节点与其所有父节点的距离之和作为传输代价,达到减少片上网络消息传递的跳数的目的。LBC 算法虽然有效降低了数据包在片上网络的传输距离,但是缺乏对动态执行环境的感知,仍然无法避免造成局部网络负载过重的问题。

3 种调度机制的共同特点是,都在静态编译时确定了指令调度的信息,无法很好地适应动态执行场景的实时需求。更进一步地解释,3 种方式都没有考虑数据流网络动态负载的实时变化状况,而只是考虑了指令的传输距离最短化和计算负载的均匀分配与否,静态调度模型本质上无法覆盖动态场景的网络变化情况,从而导致指令调度的效率变差。本研究在数据流模拟器上分别实现了 SPDI、SPS 和 LBC 3 种调度算法。通过对实验结果的分析发现,3 种方式都会引起数据流结构的片上网络局部负载分布不均衡的情况,即每个 PE 节点向东南西北 4 个方向发送的数据包的流量不均衡。实验结果如图 5 所示,选取了位于数据流阵列中央位置的 6 号 PE 和 9 号 PE,位于顶角位置的 0 号 PE 和位于侧边位置的 7 号 PE,分别代表了不同位置的 PE 所观察到的网络负载不平衡的状况。图中的每个条柱的不同图例分别表示每个方向的数据包占该节点总数据包的百分比,根据图中条柱的长度可以直观地观测到每个样例节点在不同方向的数据包分布状况。在由 16 个处理单元组成的数据流架构中,每个处理单元通过东南西北 4 个方向和相邻的处理单元通过网络传递操作数和结果数据。比如采用 LBC 算法,即使是位于阵列中间位置的 PE7,在执行 FFT 算法的时候,北向的负载明显大于其他 3 个方向的负载,而在执行 Stencil3D 算法的时候,南向负载明显大于其他 3 个方向的负载,在执行 GEMM 算法的时候,东向的负载又明显小于其他 3 个方向的负载。由此可以合理推断,采用不同的指令调度模式叠加不同的算法,每个 PE 的网络负载都呈现出不同的分布特征。本研究的实验数据表明采用 SPDI、SPS 和 LBC 调度算法,分别平均至少 90% 的 PE 存在周边网络负载分别不平衡的情况。

数据流网络负责在处理单元之间传递数据,网络的拥塞越少,数据的传输效率越高,处理单元的利用率也越高。反之,网络负载如果不均衡,局部负载过重或者过轻,都会导致网络拥塞状况的增多,从而使得数据的平均传输效率降低。传统的片上网络研究领域偏向于采用自适应路由策略来解决因网络负

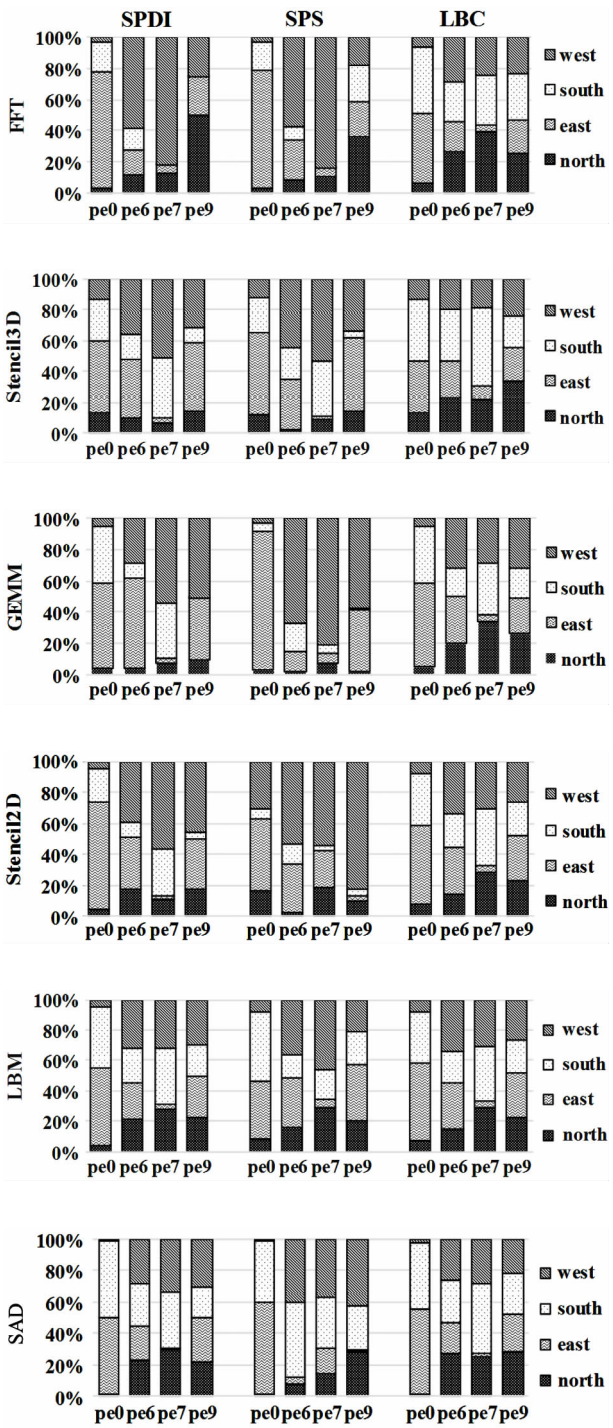

图 5　数据流网络负载分布图

载不均衡而发生拥塞的问题，通过让数据包绕过网络的拥塞点使得负载在网络上更加均匀地分布[22-24]。但是自适应路由网络的实现开销过大，内部结构复杂导致网络延迟大，以及面积和功耗的开销增大。数据流架构的所有数据传输都是通过网络传输，性能对网络延迟十分敏感，所以复杂的自适应路由网络设计不适用于数据流网络。

本研究针对目前数据流架构普遍采用的静态指令调度算法缺乏考虑网络均衡的问题，从指令调度的角度优化网络负载不均衡的问题，提出了一种基于网络负载特征进行动态指令调度的机制，该方法有助于提高数据流网络的传输效率，从而提高处理单元的利用率和数据流架构的性能。

3　基于网络负载特征感知的调度机制

在大部分数据流架构中，处理单元通常采用流水级数较少、面积和功耗开销较小的顺序流水线实现。所有指令都被保存在指令队列当中，那些源操作数已经准备好的指令等待被选择。指令选择/发射逻辑的功能是针对所有已经准备好的指令进行选择，然后发射到执行流水线。指令在执行单元中完成计算，定点指令进入定点执行单元，浮点指令进入浮点执行单元。指令在流水线中执行完成之后，计算结果被打包，然后经由相邻的片上网络路由装置发送给下游的消费者指令。在数据流架构当中，指令的并行执行导致大量数据包消息同时通过片上网络传输，片上网络中处于不同位置的路由装置的实时负载并不均衡。然而，通常片上网络每个节点都采用同构设计，负载重的节点方向更容易发生拥塞，而负载轻的节点方向则带宽利用率不足。当与处理单元相连接的路由装置发生拥塞之后，处理单元的执行流水线应对网络拥塞通常有两种方式，但是两种方式都会对性能产生负面影响，如图 6 所示，下面逐一进行分析。

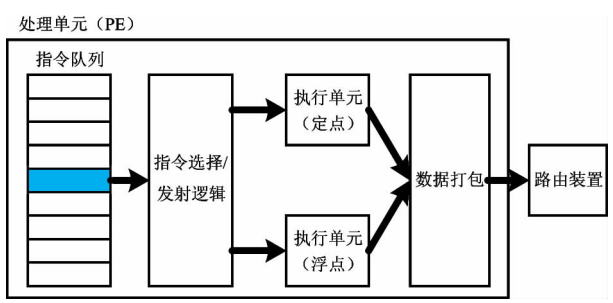

图 6　处理单元内部的指令调度示意图（传统方式）

方式 1　一条指令从指令队列通过发射逻辑被

送入到执行单元之后,该指令对应的上下文数据从指令队列当中删除,该指令所占用的指令槽变为空闲,可以接收新的上下文数据。指令在执行单元当中计算完成之后,结果数据要被发送到路由装置的时候,如果数据包的目标方向在路由装置当中发生拥塞,数据包必须在处理单元 PE 的出口进行等待。因为该数据包对应的源操作数信息已经从指令队列当中删除,不具备重新发射执行的条件,所以无法重新进入流水线。在这种情况下,由于处理单元的出口堵塞,即使指令队列当中存在可以被发送到非拥塞方向的指令,也无法从处理单元发送到路由装置。在拥塞数据包发送到路由装置之前,执行单元都处于空闲状态,无法被有效指令充分利用。

方式 2 一条指令从指令队列通过发射逻辑被送入到指令单元之后,该指令对应的上下文数据仍然保存在指令队列当中,直到确定该指令的计算结果数据包已经被发送到路由装置,对应在指令队列当中的数据才被删除,接收来自下一个上下文的数据。和方式 1 相比,在方式 2 当中,1 个上下文数据在处理单元当中停留的时间周期最少增加 3 个时钟周期(对应于指令发射,指令执行和结果数据打包 3 个流水级),即下一轮上下文数据必须晚 3 个时钟周期才能发送。如果一条指令对应的结果数据包在进入路由装置的时候,路由装置发生拥塞,那么该数据包可以被丢弃,该指令可以重新发射进入到流水线执行。后面的其他指令不会因此而被堵塞在处理单元内部。该方式的好处是在指令通过路由网络发送的时候,不会由于网络路由的拥塞而阻塞后面的指令发射。但是缺点在于上下文数据的流动速度变慢,因为新的一组上下文数据必须要等待前一组执行完之后才能进入。上下文流动速度的变慢,会影响到数据流架构的执行效率。所以,为了保证不影响上下文数据的流动速度,本研究中使用的基础执行模式是方式 1。

无论是方式 1 还是方式 2,都是普通的硬件指令调度方式,不利于在数据流执行模式的场景下充分利用网络资源快速传输数据的要求。本研究的设计目标是针对处理单元的指令调度机制进行改进和调整,从而缓解网络数据包拥塞对执行流水线利用率降低以及数据流执行效率降低的影响。要达到这样的设计目标,设计方案需要满足如下条件。

(1)不影响上下文数据在数据流架构中的流动速度。

(2)面积和功耗开销小,对能效指标的影响小。

(3)处理单元内部的指令执行不受到网络路由装置在某个方向上发生拥塞的影响。

(4)提高数据流网络的传输效率和处理单元的利用率。

基于以上设计目标,本研究提出基于网络负载特征感知的指令调度机制。该机制能够感知网络负载的变化,并根据网络负载的动态状况调整指令调度策略。

指令在流水线终点处的执行状态直接反映当时的网络路由状态,如果指令被打包完成之后无法进入路由装置,表明该指令的目标方向在对应的路由装置中负载较重,发生拥塞。反之,表明该指令的目标方向在对应的路由装置负载正常。如果指令调度结合实时的网络负载状况,可以有效地平衡路由装置在各个方向的负载,提高路由装置各方向的资源利用率。

本研究提出的改进结构如图 7 所示。在原始结构的基础上新增了 4 个部件,分别是发送检测部件、发送选择部件、数据包暂存队列和优先级调整部件。

在本研究所采用的数据流执行模型当中,为了保证上下文数据流动的速度不受影响,指令一进入到流水线执行,相应的数据信息就从指令所占用的指令槽当中删除。为了保证那些由于网络拥塞不能被正常发送的数据包不影响其他数据包的执行和发送,在每个处理单元的流水线尾部增加数据包暂存队列,用于暂时保存那些未能发送到路由装置当中的指令结果数据包。

发送检测部件用于检测和判断一个数据包在流水线执行完成并且打包之后,对应的目标方向在相邻的路由装置中是否发生拥塞。如果检测到拥塞,数据包不能进入路由装置,而是被暂存在数据包暂存队列的指令槽当中。等到路由装置的局部拥塞解除之后,该数据包再被发送到路由装置。

由于增加了数据包暂存队列,从处理单元发送

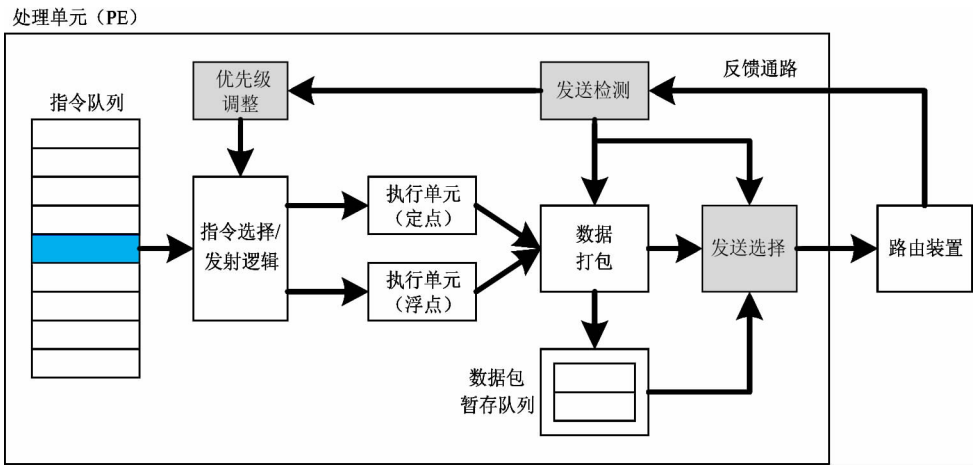

图 7　基于网络特征感知的指令调度结构

到路由装置的数据包来源由原来的 1 个变成了 2 个，但是路由装置在同一个时刻只能接收来自其中 1 个来源的数据包，所以发送选择部件负责在 2 个数据包来源之间进行选择。

如果发送检测部件检测到一条指令对应的结果数据包无法进入路由装置而是被保存在暂存队列的指令槽当中，优先级调整部件将把处理单元的指令调度逻辑切换到非拥塞方向优先的调度策略，即暂时性地降低拥塞方向的指令被发射到流水线当中执行的优先级，提高非拥塞方向指令被发射到流水线执行的优先级。正常情况下，指令调度逻辑在选择指令的时候采用的是公平轮询策略。为了保证执行调度的公平性，所有可执行的指令的优先级都一样。但是如果路由装置当中拥塞方向的负载持续过重，再继续调度指令到该方向，一方面不会对网络路由的处理速度有任何提升，另一方面导致网络路由各方向的带宽利用率不均衡。此时如果能够增加非拥塞方向的负载以平衡路由装置内部的负载分布的话，可以提高路由装置在各个方向的资源利用率，同时也有助于提高处理单元内部的执行单元的利用率。

发送检测部件

发送检测部件主要完成如下功能。

- 感知与处理单元相邻的路由装置在东南西北 4 个方向的网络拥塞状态。
- 通过反馈通路向优先级调整部件发送调整调度优先级的请求。
- 通过协调流水线空拍与发送选择部件之间的配合，在网络拥塞解除之后发送那些保存在暂存队列当中的数据包。

发送检测部件内部针对东南西北 4 个方向各自设置状态位表示其网络状态。每当路由装置内部检测到东南西北 4 个方向的拥塞状态发生变化的时候，通过反馈通路设置发送检测部件内部对应的状态位。

如果路由装置在某个方向上的负载过重，无法接收新的数据包，对应的拥塞状态位从无效变成置位。任一拥塞状态位的变化都能够触发发送检测部件。发送检测部件一方面通知优先级调整部件，降低拥塞方向上指令的调度优先级；另一方面通知数据打包部件把拥塞方向的数据包发送到暂存队列当中保存，而不是发送到路由装置，直至拥塞解除为止。反之，如果检测到路由装置在该方向上的拥塞解除，对应的拥塞状态从置位变成无效，之前被保存在暂存队列当中的数据包可以重新发送。此时，发送检测部件一方面通知优先级调整部件预留流水线空拍，另一方面通知发送选择部件在流水线空拍时选择来自暂存队列的数据包进行发送。同时，也恢复对应调度优先级的状态为原始状态。

优先级调整部件

指令调度策略的调整机制如图 8 所示。未引入网络状态信息之前的指令调度选择，是针对处理单元内部所有源操作数已经准备好的指令进行公平地轮询选择。引入网络状态信息之后，参与选择的源

操作数都准备好的指令被分为 2 组,根据网络状态位区分为高优先级组和休眠组(即低优先级组)。目标方向和拥塞方向不同的指令划分为高优先级组,目标方向和拥塞方向相同的指令划分为休眠组(图 8 所示的分组为示例)。每个分组当中的指令根据目标方向的网络状态动态调整分组。

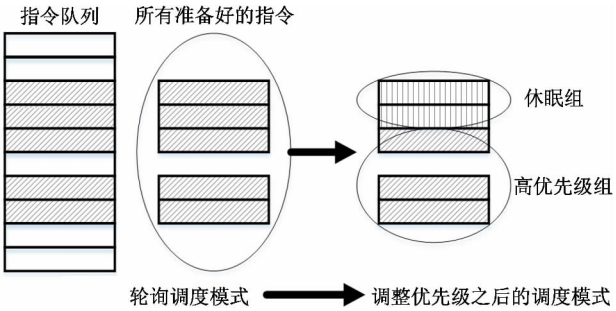

图 8　调整指令调度策略示意图

针对指令分组之后,优先对高优先级组当中的指令进行调度,组内仍然采用公平的轮询调度策略。高优先级组内的指令执行完成之后预期可以被顺利发送到路由装置,不会引起流水线的堵塞或者暂存队列的占用。当路由装置中的拥塞方向释放后,对应发送往该方向的指令也相应从休眠组重新划分到高优先级组当中。路由装置当中的网络变化状态对应于指令分组的动态变化。指令调度策略本身在任何级别的分组上都采用公平的轮询调度策略,只是参与调度的指令子集发生了变化。

执行调度过程

本研究提出的指令调度机制的执行过程如下所示。

(1) 路由装置检测到内部方向 A 发生拥塞,通过反馈通路把发送检测部件当中方向 A 的状态位设置为置位状态。

(2) 任何拥塞状态位被置位之后,数据打包部件在发送数据包之前,判断每个数据包方向是否和 A 相同。如果相同的话,数据包被存入到数据包暂存队列当中;反之,数据包通过发送选择部件发送到路由装置。

(3) 优先级调整部件针对所有准备好要执行的指令进行动态分组,将那些目标方向与 A 相同的指令划分到休眠组,其他的指令划分到高优先级组。指令调度逻辑从高优先级组当中选择指令进入执行单元的流水线。

(4) 路由装置检测到内部方向 A 的拥塞状态解除,通过反馈通路把发送检测部件中方向 A 的状态位设置为复位状态。

(5) 优先级调整部件将目标方向与 A 相同的指令从休眠组移动到高优先组,同时产生指令调度的空拍。指令空拍经过执行单元和数据打包部件之后被发送选择部件捕捉,发送选择部件将选择暂存队列当中目标方向为 A 的数据包发送到路由装置。

其中,流水线空拍的插入和消耗如图 9 所示,表示 4 个周期空拍传递的完整过程。当优先级调整部件收到来自发送检测部件的空拍插入请求之后,在指令发射周期加入空拍,表示此时钟周期不处理指令队列当中的有效指令。该空拍依次通过指令执行、结果打包之后到达发送选择部件。在发送选择部件当中,由于执行部件的流水线传递的空拍不存在有效指令,所以选择来自指令暂存队列的指令发送到路由装置当中。

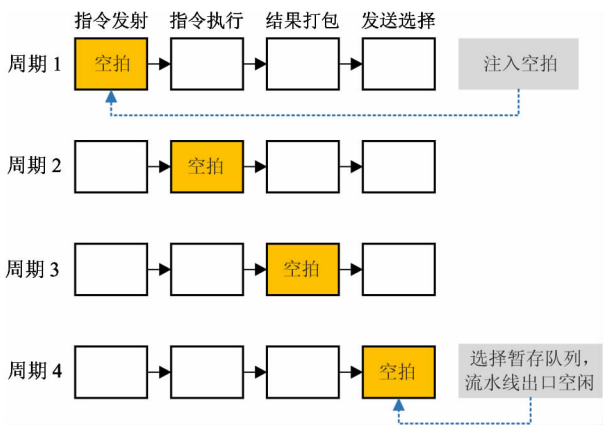

图 9　流水线空拍传播示意图

如上所述的过程表示路由装置当中方向 A 从拥塞状态置位到释放的完整过程。该机制在指令级的粒度上有效填补了网络中未被充分利用的空拍,通过改变指令执行序列的顺序,提高了数据包在网络中的传输效率。

4　实验方法

本研究使用中科院计算所自主研发的大规模并

行模拟框架 SIMICT 作为平台,运行上文提出的数据流加速器模型[25]。数据流加速器的结构如图 1 所示,表 1 列出了实验环境的各项具体配置参数。由于仿真速度的原因,该模拟器平台主要用于评估数据流模型的性能。本实验采用 Verilog 实现数据流处理器,用于评估面积和功耗。

表 1 实验环境配置

模拟器配置	
主处理器核	1 GHz,顺序多发射,ARM 指令集
片外存储	16 GB,DDR4
PE 处理单元	1 GHz,共 16 个,SIMD-4, 128 条指令, 4FMACs, 8ALUs
片上网络	XY 路由,64 比特数据包,4 套独立网络,相邻节点 1 拍延迟
指令缓存	8 Bank,4 MB
数据缓存	16 Bank,4 MB
双精度峰值性能	128 GFLOPS

在模拟平台中,通用主处理器用于执行控制程序,对性能的要求不高,计算密集型的程序段都在加速器上执行。因此,采用简单的 ARM 顺序多发射处理器作为主处理器核。数据流加速阵列的每个 PE 都设置有 4 个浮点乘加单元和 8 个定点运算单元。PE 之间通过 2D MESH 网络连接,网络路由采用 4 套独立的网络完成 PE 之间的数据交互。片上网络采用的路由策略是 *X-Y* 路由,*X* 方向具有高优先级。加速器共设置 4 MB 的缓存,分别用于保存指令和数据。

在启动阶段,主处理器将数据流加速器所需的指令和数据通过 DMA 从内存拷贝到加速器的缓存当中,然后通过配置加速器的处理阵列控制部件来启动加速器开始执行。数据流加速器完成计算任务之后,结果数据通过 DMA 从加速器的缓存再拷贝回内存当中。

本实验采用 6 种科学计算场景的应用作为测试程序,分别是 FFT(快速傅里叶变换),2D Stencil,3D Stencil,GEMM(稠密矩阵乘法),LBM(格子玻尔兹曼方法)和 SAD(绝对误差和)。各测试程序的配置详情如表 2 所示。表中的第 2 列表示每个测试程序的计算规模。FFT 表示针对 32 个点运行 16 384 个时间步;Stencil3D 表示对 1 个大小为 512 × 512 × 64 的三维矩阵计算 100 个时间步;矩阵乘法表示 2 个大小为 1 024 × 1 024 的矩阵相乘;Stencil 2D 表示对 1 个大小为 512 × 512 的二维矩阵计算 100 个时间步;LBM 表示针对大小为 120 × 120 × 150 的三维矩阵表示的格子计算 3 000 个时间步;SAD 表示输入的 2 个二维矩阵的大小都是1 920 × 1 072。表中的第 3 列表示每种测试程序对应的双精度浮点操作的总个数(其中 SAD 为定点操作的总个数)。以上选取的测试程序在科学计算领域都十分具有代表性,FFT 在计算物理和信号处理领域有十分广泛的应用。Stencil 在结构化网格和偏微分方程求解的应用涉及到气象模拟的热力/流体力学领域到电磁学领域。GEMM 是数值计算函式集 LAPACK 和高性能数值计算函式集 ScaLAPACK 的基础算法,对函数测试集的性能影响十分重要。LBM 和 SAD 分别是流体力学和数字图像处理领域的代表应用。本研究所选取的测试程序既包括通用库的算法,也包括特定领域的专用算法[26-28]。不同的算法覆盖了从访存带宽、运算通量到网络延迟容忍方面的不同特征。

表 2 测试程序配置

程序名称	计算规模	操作个数
FFT	32 points × 16384	10485760
Stencil 3D	512 × 512 × 64 100 个时间步	13421772800
GEMM	1024 × 1024 1024 × 1024	2147483648
Stencil 2D	512 × 512 100 个时间步	157286400
LBM	120 × 120 × 150 3000 个时间步	1010880000000
SAD	1920 × 1072 1920 × 1072	4487059968

实验将分析本研究提出的调度机制对于网络延迟、计算单元利用率以及性能的影响,并且给出 3 种软调度算法(即指令静态调度算法)的横向对比结果。网络延迟的降低表明数据的传输效率有所提

升,单位时间内更多的源操作数可以到达消费者指令。计算部件利用率和性能(GFLOPS/GOPS)的提升,表明数据在加速器阵列的传输效率和执行速度都有所提高。实验还进一步分析了应用本研究提出的机制之后,对于面积和功耗开销的影响。

5 结果分析

图 10 表示针对前面提到的 SPDI、SPS 和 LBC 3 种指令软调度策略,分别应用本研究提出的硬件调度机制之后,在片上网络的各个路由点观测到的数据包延迟的平均优化百分比分别是 13.5%,12.1% 和 12.9%。3 种软调度策略对应的优化百分比有所不同,但是都呈现出延迟缩小的趋势。单位时间内通过片上网络路由节点的数据包数量由于资源被更加充分地利用而有所增加,但是数据包的总数量并没有发生改变,所以传输同样数量的数据包的延迟有所降低。

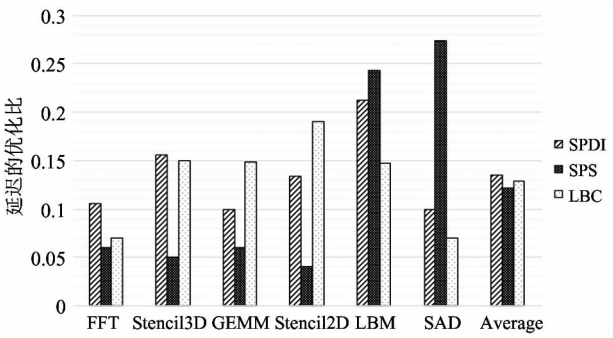

图 10 路由网络的平均传输延迟的优化比

在数据流架构当中,跨 PE 节点的指令数据以及访存数都是通过片上网络的路由节点进行传输,数据包的网络传输延迟直接影响到上游指令与下游指令之间传输上下文数据的速度。数据传输越快,指令等待源操作数的时间越短,指令从等待状态变成可发射状态的时间也越短,那么在单位时间内可以被发射到执行单元的指令数据也越多,意味着执行单元的利用率将得到提升。图 11、图 12 和图 13 分别表示了 SPDI、SPS 和 LBC 3 种指令调度策略在应用本研究提出的硬件调度优化策略前后,执行单元的部件利用率的对比情况。图中其中带有 OPT 标识的对应表示应用了硬件优化机制后的结果,未带有 OPT 标识的对应表示没有应用硬件优化机制的参考设计的结果。部件利用率表示一个运算部件实际上处理有效操作的时间占整个运行时间的比例。在总的操作数一定的前提下,部件利用率越高,总的执行时间越短,单位时间内处理的有效操作越多。应用了本研究提出的硬件优化策略之后,SPDI、SPS 和 LBC 的部件利用率分别得到了 13.2%、19.8% 和10.2% 的提升。

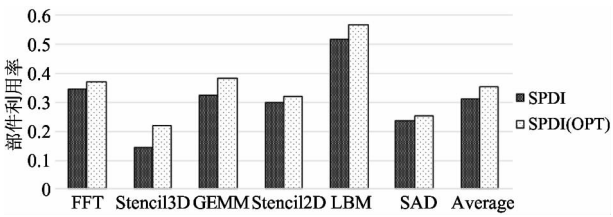

图 11 部件利用率对比(SPDI)

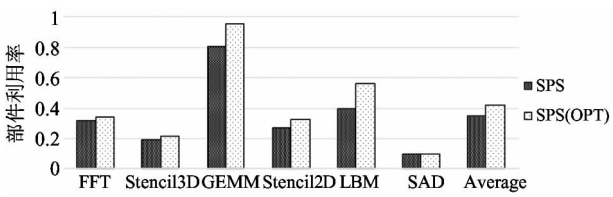

图 12 部件利用率对比(SPS)

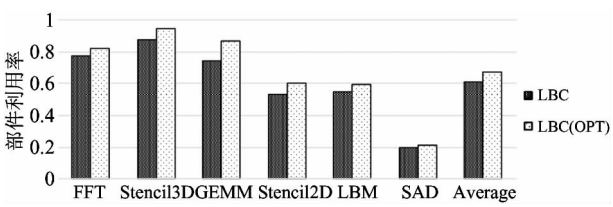

图 13 部件利用率对比(LBC)

本研究使用硬件描述语言 Verilog 实现了硬件调度优化策略,并且采用 TSMC 的 40 nm 工艺和 Synopsys 公司的 EDA 工具进行功耗和面积的评估。评估结果分别如图 14 和图 15 所示,面积和功耗的增加都控制在 1% 以内。

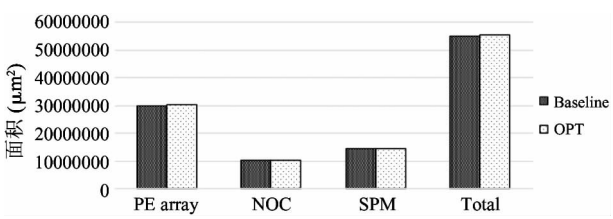

图 14 面积对比图

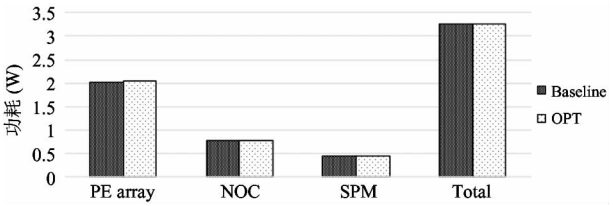

图 15 功耗对比图

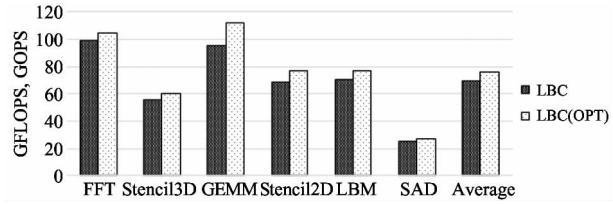

图 18 性能对比(LBC)

数据流架构的性能通常采用 GFLOPS/GOPS 来衡量,分别表示每秒所执行的浮点运算次数/每秒执行的定点运算次数。在本实验中,由于 SAD 的实现没有浮点操作,所以它的性能采用 GOPS 表示,其余 5 个测试程序的性能均采用 GFLOPS 表示。GFLOPS/GOPS 是衡量数据流架构的计算能力的指标,尤其是在科学计算领域。图 16、图 17 和图 18 分别表示了 SPDI、SPS 和 LBC 在应用本研究提出的硬件优化机制前后的性能对比图,性能提升的百分比分别是 13.3%、20.3% 和 10.4%。数据流架构性能的提升主要源于两个方面,一方面是路由网络传输速度的提升,使得指令之间的数据传输更快;另一方面来自于部件利用率的提升,在每个处理单元内部,浮点和定点运算可以并行开展。

实验结果表明,本研究针对数据流架构所提出的硬件调度优化机制针对不同的软调度策略都具有进一步优化的效果,有效地提升数据流网络传输效率的同时,也提高了数据流架构的性能。

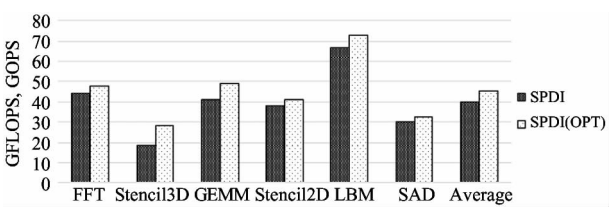

图 16 性能对比(SPDI)

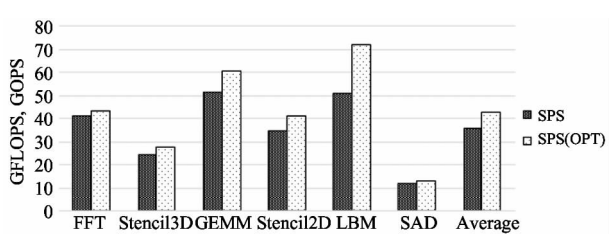

图 17 性能对比(SPS)

6 结 论

本文研究了 3 种软指令调度方式在数据流架构上执行的网络负载分布特点,分析了网络负载分布不均衡对于性能的影响。基于目前常用的软指令调度机制没有考虑网络负载分布不均对于数据流性能的影响,本文提出并实现了一种基于网络负载特征感知的硬件指令调度机制。该机制根据网络路由节点反馈的实时负载信息,针对处理单元内部的指令调度机制进行动态调整,从而达到提高网络数据传输效率的目的,进而提高数据的处理性能。

实验结果表明,本文提出的优化机制可以有效地提高数据流网络的传输效率、处理单元的利用率以及数据流加速器的性能。同时,实现该机制所需的硬件面积和功耗的开销增加都控制在 1% 以内。

后续研究将针对软指令调度机制进行进一步地优化,结合硬件优化的机制,有助于进一步提高数据流加速器的性能。

参考文献

[1] Frederico P, Diego O, Oliver P, et al. Accelerating the computation of induced dipoles for molecular mechanics with dataflow engines[C]. In: Proceedings of IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines, Seattle, USA, 2013. 177-180

[2] Jack D, David M. A preliminary architecture for a basic data-flow processor[C]. In: Proceedings of 25 Years of the International Symposia on Computer Architecture, New York, USA, 1998. 125-131

[3] Diego O, Simon T, Marino M, et al. Acceleration of a meteorological limited area model with dataflow engines [C]. In: Proceedings of the 2012 Symposium on Application Accelerators in High Performance Computing, Chi-

cago, USA, 2012. 129-132

[4] Haohuan F, Lin G, Robert C, et al. Scaling reverse time migration performance through reconfigurable dataflow engines[J]. *IEEE Micro*, 2014, 34(1): 30-40

[5] Tony N, Vinay G, Karthikeyan S. Exploring the potential of heterogeneous von Neumann/dataflow execution models [C]. In: Proceedings of the 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture, Portland, USA, 2015. 298-310

[6] Milutinovic V, Salom J, Trifunovic N, et al. Guide to DataFlow Supercomputing[M]. Switzerland: Springer International Publishing, 2015. 1-129

[7] Michael T, Jason K, Jason M, et al. The raw microprocessor: a computational fabric for software circuits and general-purpose programs[J]. *IEEE Micro*, 2002, 22(2): 25-35

[8] Gajinder P, Daniel T, Andrew D, et al. Deterministic parallel processing[J]. *International Journal of Parallel Programming*, 2006, 34(4): 323-341

[9] Doug B, Stephen K, Kathryn M, et al. Scaling to the end of silicon with EDGE architectures[J]. *Computer*, 2004, 37(7): 44-55

[10] Steven S, Andrew S, Martha M, et al. The WaveScalar architecture[J]. *ACM Transactions on Computer Systems*, 2007, 25(2): 1-54

[11] Masasuke K, Hiroshi Y, Yasusuke K. DDDP-a distributed data driven processor[C]. In: Proceedings of the 10th Annual International Symposium on Computer Architecture, New York, USA, 1983. 236-242

[12] Angshuman P, Michael P, Michael A, et al. Triggered instructions: a control paradigm for spatially-programmed architectures[C]. In: Proceedings of the 40th Annual International Symposium on Computer Architecture, New York, USA, 2013. 142-153

[13] Angshuman P, Michael P, Michael A, et al. Efficient spatial processing element control via triggered instructions[J]. *IEEE Micro*, 2014, 34(3): 120-137

[14] Jan H, Henk C. Transport-triggering vs. operation-triggering[C]. In: Proceedings of the Fritzson P. A. (eds) Compiler Construction, Berlin, Germany, 2005. 435-449

[15] Philip T, David B, Richard H. Data-driven and demand-driven computer architecture[J]. *ACM Computing Surveys*, 1982, 14(1): 93-143

[16] Ramadass N, Sundeep K, Doug B, et al. Static placement, dynamic issue (SPDI) scheduling for EDGE architectures[C]. In: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, Washington, USA, 2004. 74-84

[17] Shen X W, Ye X C, Tan X, et al. An efficient network-on-chip router for dataflow architecture[J]. *Journal of Computer Science and Technology*, 2017, 32(1): 11-25

[18] Robert I. Toward a dataflow/von Neumann hybrid architecture[C]. In: Proceedings of the 15th Annual International Symposium on Computer architecture, Los Alamitos, USA, 1988. 131-140

[19] John S, Christopher R, I-Jui S, et al. Parboil: a revised benchmark suite for scientific and commercial throughput computing, IMPACT-12-01[R]. Illinois: University of Illinois at Urbana-Champaign, Center for Reliable and High-Performance Computing, 2012

[20] Katherine C, Xia C, Doug B, et al. A spatial path scheduling algorithm for EDGE architectures[C]. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, New York, USA, 2006. 129-140

[21] 申小伟,叶笑春,王达,等. 一种面向科学计算的数据流优化方法. 计算机学报, 2017, 40(9): 2182-2196

[22] Paul G, Boris G, Stephen K. Regional congestion awareness for load balance in networks-on-chip[C]. In: Proceedings of 2008 IEEE 14th International Symposium on High Performance Computer Architecture, Salt Lake City, USA, 2008. 203-214

[23] Wang C F, Hu W H, Bagherzadeh N. Scalable load balancing congestion-aware network-on-chip router architecture[J]. *Journal of Computer and System Sciences*, 2013, 79(4): 421-439

[24] Marcelo B. Network-on-chip with load balancing base on interleave of flits technique[J]. *International Journal of advanced studies in Computer Science and Engineering*, 2015, 4(10): 16-25

[25] Ye X C, Fan D R, Sun N H, et al. SimICT: a fast and flexible framework for performance and power evaluation of large-scale architecture[C]. In: Proceedings of the International Symposium on Low Power Electronics and Design, Beijing, China, 2013. 273-278

[26] Anthony N, Nadathur S, Jatin C, et al. 3.5-D blocking

optimization for stencil computations on modern CPUs and GPUs[C]. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, Washington, USA, 2010. 1-13

[27] Jakub K, Stanimire T, Jack D. 2012. Autotuning GEMM Kernels for the Fermi GPU[J]. *IEEE Transactions on Parallel and Distributed Systems*, 2012, 23(11): 2045-2057

[28] del Mundo C, Feng W C. Towards a performance-portable FFT library for heterogeneous computing[C]. In: Proceedings of the 11th ACM Conference on Computing Frontiers, New York, USA, 2014. 1-10

Study of network loading based instruction scheduling mechanism in dataflow architecture

Feng Yujing[*,**], Ou Yan[*,**], Ye Xiaochun[*], Fan Dongrui[*,**], Tan Xu[*,**], Tang Zhimin[*,**]

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

([**]School of Computer and Control Engineering, University of Chinese Academy of Sciences, Beijing 100049)

Abstract

Instruction scheduling mechanism of dataflow architectures is analyzed. Based on the commonly-used instruction scheduling method, a dynamic hardware instruction scheduling mechanism with mode switching is proposed to optimize the efficiency of data transfer in dataflow architectures. Since the dataflow architectures are characterized by paralleled execution, large amounts of data are transferred and computed concurrently. However, the network loading is not evenly distributed all over the processing array. If a part of the network is congested due to unbalanced loading, the pipeline of the neighboring processing element will be stalled and transfer efficiency of the dataflow network will be greatly reduced. As a result, the latency of data transfer, utilization of computational units and execution efficiency of the dataflow architecture will be affected. This study proposes a hardware instruction scheduling mechanism to optimize the efficiency of data transfer in dataflow network under the cases of inevitable network imbalance. The proposed mechanism is simulated by using a dataflow simulator system, and the results show that by adopting the proposed mechanism, the average transfer latency through dataflow network is reduced by 12.8%, average utilization of computational units is improved by 14.4%, and average performance of dataflow architecture is improved by 14.7%.

Key words: dataflow architecture, dynamic instruction scheduling, network-on-chip (NoC), network loading, utilization of computational unit