UPPA:面向异构众核系统的统一并行编程架构

吴树森 董小社 王宇菲 王龙翔 朱正东

(西安交通大学计算机科学与技术学院 西安 710049)

摘 要 主流异构并行编程方法如 CUDA 和 OpenCL,其编程抽象层次低,编程接口靠近底层,无法为用户屏蔽底层硬件和运行时细节,导致编程逻辑复杂,编程困难易错.同时应用性能绑定于底层运行时环境,在硬件架构变化时需要根据硬件特征进行针对性改动和优化,无法保证上层应用的统一.为了简化异构并行编程,提高编程效率,实现上层应用的统一和跨平台,本文提出了一种面向异构众核系统的高层统一并行编程架构 UPPA(Unified Parallel Programming Architecture).架构中首先提出了数据关联计算编程模型,实现了不同层级不同模式并行性的统一描述,简化了异构并行编程逻辑,提供了高层统一的并行编程抽象;继而设计了数据关联计算描述语言为用户提供简便易用的统一编程接口,通过高层语义结构保留了应用的并行特征,可以指导编译和运行时系统实现向不同硬件架构的自动映射,保证了上层应用的统一,并采用 C 语言兼容的语法提供针对高层语义结构的语言扩展,保证编程接口的易学易用;最后提供了基于 OpenCL 的编译和运行时原型系统,以 OpenCL 为中间语言实现了高层应用在不同异构系统上的执行,提供了良好的跨平台特性.我们使用数据关联计算描述语言对 Parboil 和 Rodinia 测试集中的多个测试用例进行了重构,并在 NVIDIA GPU 和 Intel MIC 两种异构平台上进行了验证测试.每个测试用例重构的代码量与测试集提供的串行代码相当,仅为测试集 OpenCL 代码的 13%~64%,有效地降低了异构编程的工作量.在编译和运行时系统的支持下,重构代码无需改动就可以在两种平台上执行.相比于人工编写且经过优化的测试集 OpenCL 代码,重构代码在 GPU 和 MIC 两种平台下分别能够达到其性能的 91%~100%和 76%~98%,这表明了本文方法的有效性和编译与运行时系统的高效.

关键词 异构并行编程;数据关联计算;并行编程模型;统一编程架构;OpenCL

中图法分类号 TP312 **DOI 号** 10.11897/SP.J.1016.2020.00990

UPPA: Unified Parallel Programming Architecture for Heterogeneous Systems

WU Shu-Sen DONG Xiao-She WANG Yu-Fei WANG Long-Xiang ZHU Zheng-Dong

(*School of Computer Science and Engineering, Xi'an Jiaotong University, Xi'an* 710049)

Abstract Mainstream heterogeneous parallel programming methods such as CUDA and OpenCL provide close-to-mental programming interface and present low-level programming abstraction. They merely shield the underlying hardware and runtime details for heterogeneous developers, resulting in complicated programming logic and making heterogeneous programming difficult and error-prone. Meanwhile, the performance of an application developed with low level programming methods is bound to specific runtime environment. As a result, high-level applications cannot execute on different hardware or perform poorly after been ported to other heterogeneous systems. Specific and manual modification and optimization to the application according to the hardware features are essential when the hardware architecture changes. High-level applications cannot maintain unified and lack cross-platform feature. In order to simplify heterogeneous parallel programming, improve programming productivity, and achieve the goal of producing unified and cross-platform high-level

收稿日期:2019-04-29;最终修改稿收到日期:2019-10-21. 本课题得到国家自然科学基金(61572394)、国家重点研发计划(2017YFB0202002)资助.**吴树森**,博士研究生,主要研究方向为高性能计算、异构并行编程模型与方法. E-mail: wuss153@stu.xjtu.edu.cn. **董小社**(通信作者),博士,教授,博士生导师,主要研究领域为高性能计算、高效能存储系统、云计算. E-mail: xsdong@xjtu.edu.cn. **王宇菲**,博士研究生,主要研究方向为高性能计算、高效能存储系统. **王龙翔**,博士,工程师,主要研究方向为高性能计算、高效能存储系统. **朱正东**,博士,教授级高工,主要研究领域为高性能计算、云计算与大数据.

applications, this paper presents UPPA, a Unified Parallel Programming Architecture for heterogeneous many-core systems. Firstly, the UPPA proposes data associated computation (DAC) programming model, which realizes unified description of parallelism of different levels and different patterns. The DAC model provides a high-level unified parallel programming abstraction and simplifies the heterogeneous parallel programming logic. Secondly, the UPPA provides a unified programming interface for the developers with the DAC description language. The DAC description language implements the DAC model with language extensions. High-level semantic structures are designed to preserve the parallel features of the application and guide the compilation and runtime system to conduct automatic mapping of high-level applications onto different hardware architectures, saving programming effort while keeping high-level applications unified. What is more, the DAC description language adopts C-like syntax for the language extensions that implements these high-level semantic structures, ensuring the easy-to-learn and easy-to-use features of the programming interface. Finally, a prototype system which is consisting of a source-to-source compiler and runtime support is implemented on the top of OpenCL. The runtime system encapsulates OpenCL runtime APIs with runtime library functions. Based on these library functions, the source-to-source compiler generates standard OpenCL code from the application developed with the DAC description language. Using OpenCL as an intermediate language, the compiler and runtime system achieves efficient execution of high-level applications on different heterogeneous systems, providing a fine cross-platform feature. We rebuilt multiple benchmarks which are selected from the Parboil benchmark suite and the Rodinia benchmark suite with the DAC description language and conducted experimental tests on both a NVIDIA GPU and an Intel MIC platforms. The code size of each rebuilt benchmark is roughly equivalent to that of the serial code provided by the corresponding benchmark suite, which is only 13% to 64% of the original benchmark OpenCL code, reducing the workload of heterogeneous programming significantly. With the support of the compile and runtime systems, the rebuilt benchmarks can execute on both platforms smoothly without modification. The rebuilt benchmarks yield 91% to 100% and 76% to 98% of the performance of the handcrafted and optimized benchmark OpenCL code on the GPU platform and the MIC platform, respectively. That demonstrates the effectiveness of the UPPA method and the efficiency of the compiler and runtime system.

Keywords　heterogeneous parallel programming; data associated computation; parallel programming model; unified programming architecture; OpenCL

1　引　言

由于功耗和散热问题导致摩尔定律的失效，处理器向着多核和众核的方向发展，集成有众多简单核心的大规模并行异构处理器如GPU、MIC、FPGA等，已经成为构建高性能计算机系统的重要选择. 不同架构、不同计算特征且在不断发展的异构众核系统对并行编程模型与方法提出了更高的要求[1].

编程效率、可移植性和可扩展性以及性能是评价并行编程模型与方法的重要指标[2-3]. 主流的异构并行编程方法CUDA①、OpenCL② 解决了异构系统的可用性问题并能够提供良好的性能，构成了异构并行编程的基础[4]. 然而靠近底层的编程接口和编程抽象暴露了底层运行时细节，这一方面导致用户编程复杂易错，使得异构编程效率的问题日益突出；另一方面由于应用中包含底层运行时细节，限制了应用的可移植性和可扩展性. 因此，虽然OpenCL提供了统一的编程接口和标准，但要发挥不同架构处理器的计算效能仍需要结合硬件特征进行人工移植和优化[5]，不能保证上层应用的跨平台可移植性. 面

① CUDA toolkit. https://developer.nvidia.com/cuda-toolkit 2019,4,23

② OpenCL overview. https://www.khronos.org/opencl/2019,4,23

对不断发展的异构系统，现有的编程方法无法满足用户对高层统一编程的需求.

为了实现高层统一编程，首先需要提高并行编程抽象层次，隔离底层硬件和上层编程，同时为了简化异构编程、提高编程效率，需要为用户提供简便易用的高层统一编程接口，使得用户编程时无需管理底层运行时细节，就能实现运行时无关的并行编程.上层应用到不同硬件平台的映射与执行等工作可以交由编译器或运行时系统自动完成.

本文提出了一种面向异构众核系统的高层统一并行编程架构 UPPA，包括数据关联计算编程模型、数据关联计算描述语言以及源到源编译器和运行时原型系统.其中，数据关联计算模型提供了高层统一的并行编程抽象，数据关联计算描述语言提供了统一的编程接口，二者结合实现运行时无关的高层统一编程.源到源编译器和运行时原型系统负责高层应用到具体运行时环境的映射执行，保证上层应用的统一和跨平台.

数据关联计算编程模型以数据、关联结构和计算三大模块组织计算任务，简化了编程逻辑，通过计算任务的数据标签表达任务并行性，通过计算任务中的关联结构表达不同模式的数据并行性，实现了统一的并行表达，同时计算模块的编程得以串行化，降低了编程难度.

数据关联计算描述语言在 C 语言的基础上提供了附加语法结构和标识符的形式实现了对数据关联计算模型的支持，语法简单，易学易用，且需要额外编程的关联结构模块独立于计算部分，最大程度兼容已有代码，减少代码重构和移植的工作量.高层语法结构和标识符保留了上层应用的并行度和并行模式等信息，能够为编译和运行时系统进行自动的映射提供指导，保证在不同系统上的映射效率和应用执行性能，实现高层统一编程.

本文实现了以 OpenCL 为中间语言的编译和运行时原型系统以进行验证测试.源到源编译器根据高层应用自动生成 OpenCL 代码，运行时系统封装了 OpenCL 的编程接口，实现了自动的线程映射和执行管理.使用数据关联计算描述语言重构的测试用例经过编译后，在 GPU 和 MIC 两种平台下进行了测试，与测试用例中人工编写且经过优化的 OpenCL 代码相比，生成代码的性能与之相当，验证了方法的有效性和编译与运行时系统的高效.本文方法的实现并不仅限于 OpenCL，在以后的研究中可以通过不同的源到源编译器实现向其他底层编程方法的映射.

本文第 2 节介绍相关工作；第 3 节阐述基于数据关联计算编程模型的高层统一并行编程抽象；第 4 节中展示统一并行编程接口即数据关联计算描述语言的设计；第 5 节为编译和运行时原型系统的实现；第 6 节为实验测试；第 7 节总结全文.

2 相关工作

并行编程领域事实上的业界标准 MPI 和 OpenMP 以及主流的异构并行编程方法 CUDA 和 OpenCL 均为典型的多进程/线程编程方法，并行组织和执行管理由用户负责，虽然可以获得较高的性能，但牺牲了用户编程效率和应用的可移植性和可扩展性.Pocl[6] 提供了一种可移植的 OpenCL 实现，基于内核编译技术部分解决了 OpenCL 应用的可移植性问题，但使用 OpenCL 的编程效率问题没有改观.这些底层方法可以作为其他高层方法实现的基础，本文方法和 Lime[7]、Lift[8] 等均选取了 OpenCL 作为中间语言以实现在不同平台上的执行，其他方法如 PARRAY[9]、Copperhead[10] 等则基于 CUDA 实现了对 GPU 平台的支持.

围绕着如何提高并行编程效率和编程抽象层次，对于实现高层统一编程等问题有着许多相关研究.表1

表 1 UPPA 架构与其他高层编程模型及方法的对比

编程方法	实现特征	基础语法	代码重构	细粒度数据并行	粗粒度数据并行	任务并行	数据管理	同步	异构支持	模块化
UPPA	语法结构和标识符	C	轻量	√	√	√	自动	自动	跨平台	√
OpenACC	指导语句	C/C++	轻量	√	√		人工	人工	跨平台	
Chapel	语言	类 C	中等	√	√	√	人工	人工	否	
Lime	语言	Java	重度	√	√	√	人工	自动	跨平台	
Copperhead	并行语句	Python	中等	√			自动	自动	GPU	
SkePU	并行语句	C++	中等	√			自动	自动	跨平台	
Lift	并行语句	函数式	重度	√	√		自动	自动	跨平台	
PARRAY	语言扩展	C	中等	√	√	√	人工	自动	GPU	
Kokkos	运行时库	C++	轻量	√			自动	人工	跨平台	

总结并对比了 UPPA 架构和其他高层编程模型及方法的特点. UPPA 架构基于数据关联计算模型提供了高层编程抽象，实现了不同模式并行性的统一表达，通过数据、关联结构和计算三大模块组织计算任务，并定义不同模块之间的连接规则，提供了模块化特性，简化了并行编程逻辑，提高编程效率的同时计算模块可以较好地兼容已有代码，降低应用开发和重构的工作量. UPPA 架构设计了类 C 的语法结构和标识符，为用户提供简便易用的统一编程接口，用户编程时无需考虑底层运行时细节，并行组织、数据管理和同步等工作由源到源编译器和运行时系统自动完成.

为了提高并行编程效率，基于编译器实现的自动并行化研究试图最大程度地降低用户并行开发的工作量. 但基于编译器实现自动的串行代码并行化面临着并行解析困难、编译实现复杂且性能不高的问题. 另一类基于编译器的方法则提供了编译指导语句，指导编译器完成并行映射和执行，如 OpenACC①. 但基于编译器的方法往往缺乏高层并行抽象，如 OpenACC 的编程模型和 CUDA 与 OpenCL 一脉相承，并行组织、数据管理和同步等依然需要用户显式进行，其代码在不同平台上的性能优化也需要人为干预[11]，难以实现高层统一编程. UPPA 架构通过数据关联计算模型提供了统一的并行抽象，便于并行表达，同时用户无需管理运行时并行细节.

为了提高并行编程抽象层次，新型并行语言如 Chapel[12] 和 Lime[7] 提供了详尽的高层并行语法结构. Chapel 语言坚持显式并行编程，由用户负责并行开发、数据分布和同步，并通过域的概念提高了 Forall 语句的灵活性，但其细粒度并行表达依然主要针对 for 循环展开，同时尚未实现对异构系统的支持. Lime 语言结合了流编程模型和面向对象特性以表达应用中的并行性. 虽然编程抽象层次得到提高，但编程逻辑没有简化，复杂的语法也增加了学习和使用难度，不利于编程效率的提高，同时应用需要重新开发或移植，使得新语言难以推广应用. UPPA 架构通过数据、关联结构和计算三大模块组织计算任务，在提供高层并行抽象的同时简化了并行编程逻辑. 围绕着关联结构进行描述的语法结构和标识符更方便于学习和使用，计算模块能够较好地兼容已有代码，降低应用开发成本.

基于并行语句的方法针对特定并行模式进行了抽象. Fortran 90② 中添加的 Forall 语句和 C++ AMP③ 提供的 parallel_for_each 结构等主要用于程序中的 for 循环展开. 面向大规模数据处理的 MapReduce 模型[13] 将应用抽象为 Map 和 Reduce 两种并行过程，简化了并行编程逻辑. 多种方法吸收了 Map 结构用于异构并行编程，如 merge[14]、Copperhead[10] 等，Lime 语言中也通过 map 语句提供了对数据并行的支持. SkePU[15] 提供了 Map、Reduce、Scan 和 MapArray 等语句，Lift[7] 在函数编程的基础上加入了 map、zip、reduce 等语句并进一步通过 split、join 语句提供了对粗粒度数据并行的支持. 由于单一语句只能描述特定的并行模式，并行语句只能提供有限的并行抽象，且目前的主要研究还是针对于细粒度数据并行，与 UPPA 架构相比缺乏对统一并行表达的支持. Map 等并行语句影响了计算任务的编程，增加了代码移植的负担. 同时 Lift 等基于函数式编程的方法有着较高的学习和使用门槛，不便于编程效率的提高.

PARRAY[9] 使用了数组结构来组织数据和线程，通过维度树来表示不同数据的存储层次，通过线程数组指示 Pthread、MPI 和 CUDA 线程，简化了异构集群上需要不同方法混合编程的难题，支持不同层级的并行表达. 但其并行表达是显式进行的，还需要用户使用语言扩展管理数据移动，同时 PARRAY 主要针对的是异构集群上线程组织和通信问题，并没有简化具体计算任务的编程.

Kokkos[16] 提供了基于 C++ 的运行时库以减少用户异构编程的工作量，主要实现以细粒度并行模式执行并管理数据的存储层次. 使用 Kokkos 需要对数据结构和函数实现进行重构，并显式指定计算核心和选择执行设备，其跨平台特性需要不同的后端实现支持.

数据流编程模型[17-19] 通过数据流图和数据依赖关系表达应用中的并行性，适合于组织并行的计算任务，对于数据并行的支持则相对缺乏. 流编程模型[20-22] 将数据视作流并提供了相应的流操作，可以充分利用应用和硬件中的数据并行性，但基于流的数据抽象限制了模型的应用场景.

3　高层统一的并行编程抽象

本节展示了数据关联计算编程模型的设计，并

① OpenACC homepage. https://www.openacc.org/2019, 4, 23
② Fortran 90/95. http://www.fortran.com/the-fortran-company-homepage/fortran-training/fortran-9095/2019, 4, 23
③ C++ Accelerated Massive Parallelism. https://msdn.microsoft.com/en-us/library/hh265137.aspx 2019, 4, 23

阐述了数据关联计算模型如何提供高层的并行编程抽象,实现统一的描述式并行表达.在本节最后通过矩阵相乘的示例介绍数据关联计算模型的应用.

3.1 数据关联计算编程模型

程序=数据结构+算法,根据数据驱动模型和数据流编程模型,计算任务由数据和独立于具体计算任务的计算方法组成,数据驱动了计算任务,应用的并行性取决于计算任务之间的数据依赖关系.不同层级的并行性导致数据依赖关系的解析异常复杂,同时由于上层应用中缺乏对并行特征的描述,编译器实现困难,仅依靠数据和算法两个模块不足以描述应用中的并行性.

数据关联计算模型在数据和计算两大结构的基础上提出了关联结构以描述应用中的并行性,数据、关联结构和计算成为构成并行程序的基本模块,计算任务表示为数据、关联结构和计算三大模块的组合,并行应用则为计算任务的集合.

计算、数据、关联结构的定义如定义 1、定义 2 和定义 6 所示.数据包括计算任务的输入和输出,是计算所操作的对象.计算是独立于具体计算任务能够完成特定功能的抽象计算过程.关联结构则连接数据和计算构成具体的计算任务.关联结构实现了以下两种功能:

(1)将模块化的计算关联于特定数据,使计算任务通过数据实现标签化,由计算数据标记不同的计算任务;

(2)构建数据到计算过程之间的管道,描述数据如何分解映射到计算过程的数据接口形成并行计算实例.

定义 1. 计算.计算为拥有良好定义数据接口的算法过程.计算 C 可以通过映射 f 来表示:

$$f: D_1\times\cdots\times D_n\to R_1\times\cdots\times R_m;\ n,m\in\mathbf{N}^*.$$

定义 2. 数据.根据定义 1,定义计算 C 相关的数据的可能组织结构为 DT:

$$DT\subsetneq\bigcup_{\gamma=0}^{\infty}\Gamma_\gamma,$$

其中,$\Gamma_0=D_i$ 或 R_j,$i\in[1,n]$,$j\in[1,m]$,$\Gamma_1=2^\Gamma$,$\Gamma_{i+1}=2^{\Gamma_i}$.

DT 即表示一个数据.当 DT 为 Γ 上的幂集或高阶幂集的某个真子集时,DT 表示了集合的复合结构,即 DT 为集合组成的集合,其中的每个元素都是计算 C 的一个可能的输入集合或集合的集合.

定义 3. 降维算子.降维算子 $\overline{D}$ 表示将集合展开为各个元素以分别进行处理.其性质为:

(1)若有穷集合 $A=\{a_1,\cdots,a_n\}$,则有 $\overline{D}(A)\to(a_1,\cdots,a_n)$;$n=|A|=|\overline{D}|$.$n$ 为有穷集合 A 的基数,同时代表了此降维算子所提供的数据并行度.

(2)当降维算子作用于多个集合 $A,B,\cdots,M$ 且 $n=|A|=|B|=\cdots=|M|$,则有 $\overline{D}(A,B,\cdots,M)\to(a_i,b_j,\cdots,m_k)_n$.不同的集合展开为 n 个元组,每个元组中包含每个集合中的一个元素,同一集合中的不同元素包含在不同的元组中.

(3)当集合 A 为高阶幂集上的真子集时,集合 A 包含多重集合的嵌套,可以通过不同的降维算子递归展开:

$$\overline{D}_2(\overline{D}_1(A))\to(a_{11},a_{12},\cdots,a_{1k_1},\cdots,a_{i1},a_{ik_i},\cdots,a_{nk_n});$$
$$n=|A|,\ k_i=|a_i|.$$

(4)独立算子之间的乘法原则:

$$\overline{D}_1(A)\circ\overline{D}_2(B)\to(a_1,\cdots,a_m)\times(b_1,\cdots,b_n)=$$
$$((a_1,b_1),\cdots,(a_1,b_n),\cdots,(a_m,b_1),\cdots,(a_m,b_n));$$
$$m=|A|,\ n=|B|.$$

定义 4. 分区算子.分区算子 $\overline{F}$ 表示对集合的划分,集合分解为多个不相交真子集以分别进行处理.其性质为:

(1)$\overline{F}(A)\to(A_1,\cdots,A_m)$;$m=|\overline{F}|$.$m$ 为分区数量,同时表示此分区算子所提供的并行度,分区特征为:

① 各子集之间均匀划分,即

$$\lfloor|A|/m\rfloor\le|A_i|\le\lfloor|A|/m\rfloor+1.$$

② 各子集之间不相交,即

$$A_i\cap A_j=\varnothing;\ i,j\in[1,m],\ i\ne j.$$

③ 各子集的合集为 A,即

$$A=\bigcup_{i=1}^{m}A_i.$$

(2)当分区算子作用于多个集合 $A,B,\cdots,L$,则有 $\overline{F}(A,B,\cdots,L)\to(A_i,B_j,\cdots,L_k)_m$.各集合按照相同的分区数量 m 进行划分,构成 m 个元组,每个元组包含每个集合的一个分区,每个集合的不同分区包含在不同的元组中.

(3)独立算子之间的乘法原则:

$$\overline{F}_1(A)\circ\overline{F}_2(B)\to(A_1,\cdots,A_m)\times(B_1,\cdots,B_n)=$$
$$((A_1,B_1),\cdots,(A_1,B_n),\cdots,(A_m,B_1),\cdots,(A_m,B_n));$$
$$m=|\overline{F}_1|,\ n=|\overline{F}_2|.$$

定义 5. 保形算子.保形算子 $\overline{E}$ 表示集合不可划分,即 $\overline{E}(A)\to A$.

定义 6. 关联结构.关联结构连接数据和计算,通过降维算子、分区算子和保形算子三类算子描述数据如何映射到计算的数据接口,使数据符合计

算的数据接口特征. 关联结构由三元组〈DS,C,R〉定义,其中：

(1) DS 为数据集的集合：

$$DS=\{ds_1,\cdots,ds_m\},\ m\geq 1,\ DS\neq\varnothing.$$

(2) C 为算子集合,即降维算子集合 D、分区算子集合 F 和保形算子集合 E 的并集,$\boldsymbol{N}$ 表示自然数集合,其中：

① $C=\{c_1,\cdots,c_n\},\ n\geq 1, C\neq\varnothing$；

② $C=D\cup F\cup E$；

③ $D=\{d_i\mid 1\leq i\leq k, k\in\boldsymbol{N}\mid\}$, d_i 为降维算子；

④ $F=\{f_i\mid 1\leq i\leq l, l\in\boldsymbol{N}\}$, f_i 为分区算子；

⑤ $E=\{e_i\mid 1\leq i\leq h,\ h\in N\}$, e_i 为保形算子；

⑥ $k+l+h=n$.

(3) R 为数据、算子与集合{0,1}上的关系：

$$R: DS\times C\rightarrow DS\times C\times\{0,1\}.$$

关系 R 可以表示为 $m\times n$ 阶的 0-1 矩阵 $\boldsymbol{P}=[p_{ij}]$：

$$p_{ij}=\begin{cases}1, & R:(ds_i,c_j)\rightarrow(ds_i,c_j,1)\\ 0, & R:(ds_i,c_j)\rightarrow(ds_i,c_j,0)\end{cases}$$

关系 R 的性质可以通过矩阵 $\boldsymbol{P}$ 表示：

① 对每个行向量 $\boldsymbol{p}_i$,有：

$$\bigvee_{j=1}^{n} p_{ij}=1.$$

即关联结构中的每个数据集都有对应的算子进行标记；

② 若 $p_{ij}=p_{ik}=1$ 且 $j\neq k$,则 $c_j, c_k\in D$,即每个数据集只能被多个降维算子标记,不同类型算子之间互斥.

3.2　运行时无关的统一并行表达

数据关联计算模型通过模块化的数据、关联结构和计算提供了“并行程序＝数据＋关联结构＋计算”的编程抽象,实现不同层级不同模式并行性的统一表达,如图 1 所示.

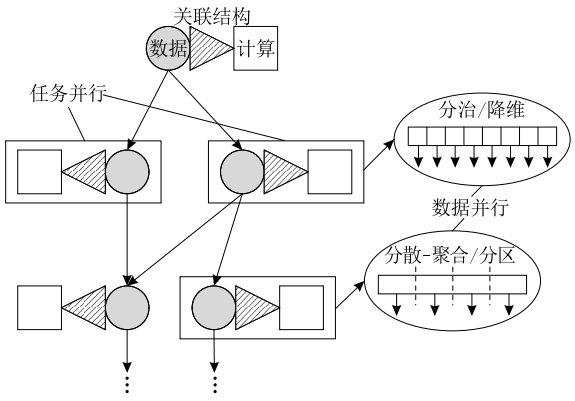


图 1　不同层级不同模式并行性的统一表达

关联结构将模块化的计算绑定于特定的计算数据,实现了计算任务的数据标签化. 数据标签化后的计算任务可以结合数据流模型组织调度计算任务,通过数据流图和任务间的数据依赖关系表达应用中的任务并行性,提供对任务并行性的支持. 相比于数据流模型,标签化的计算任务将数据流图的应用定位在任务并行级,简化了数据流图,便于编译器的实现和编译与运行时的并行解析,有助于提高编译映射效率.

关联结构中通过不同的算子标识了数据在运行时不同模式的划分方法,为数据并行的实现提供了依据. 数据并行的实现策略主要为两类,一类是将问题分解为可并行处理的子问题,将数据分解为子数据,分而治之；另一类是分散-聚合策略,将数据划分为多个小数据并行处理再收集各个中间结果合并为最终结果. 在关联结构中,降维算子描述了分治策略的数据划分特征,分区算子则描述了分散-聚合策略的划分特征,通过定义不同的算子和独立算子之间的复合关系描述数据的划分特征,实现了对不同模式的数据并行性的表达.

关联计算模型实现了运行时无关的并行表达. 应用中的任务并行性取决于计算任务之间的数据流关系,而数据并行性则由数据本身的划分特征决定. 用户只需要通过模型充分表达应用中的并行性,而无需关注具体的运行时实现细节. 通过对高层应用的解析,编译和运行时系统可以获取上层的并行特征,自动实现向底层硬件的映射.

3.3　编程模型应用示例

图 2 展示了如何通过数据关联计算模型将矩阵相乘表示为并行的向量相乘计算,以表达矩阵相乘计算过程中的并行性. 矩阵相乘是数值线性代数中重要且基本的计算过程,具有良好的数据并行性.

矩阵相乘的计算方法为

$$C_{ij}=\sum_{l=1}^{k} A_{il}\times B_{lj}.$$

其计算核心为向量相乘,计算的数据接口的输入元素为两个维度值相等表示为 i 的向量,输出元素为标量. 矩阵为向量的集合,向量为标量的集合,矩阵相乘计算中的两个输入矩阵则为计算数据接口输入元素的集合,输出矩阵则为输出元素集合的集合.

根据算法和模块化要求,关联结构的形式数据接口为三个矩阵,各矩阵的维度值表示为 $m\times k$、

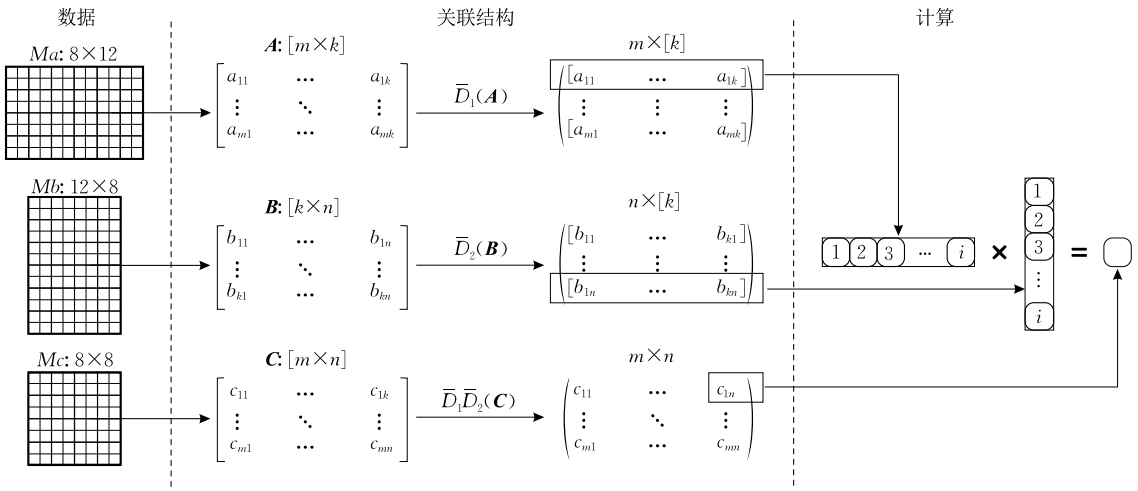


图 2 矩阵相乘的数据关联计算表示

$k\times n$、$m\times n$. 输入矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 通过两个降维算子 $\overline{D}_1$ 和 $\overline{D}_2$ 分别降维为 m 和 n 个长度相等的行向量与列向量，$\overline{D}_1$ 和 $\overline{D}_2$ 同时标记矩阵 $\boldsymbol{C}$，根据定义 3-2 和 3-3，将其对应分解为 $m\times n$ 个标量. 最终根据独立算子间的乘法原则，将三个矩阵分解为 $m\times n$ 个由两个长度相等的向量和一个标量组成的元组，每个元组满足计算模块的输入数据接口的要求.

图中的计算数据为 8×12、12×8、8×8 的三个矩阵，符合关联结构数据接口的要求，根据数据特征和关联结构可以得到此任务的最大并行度为 8×8=64. 编译和运行时系统解析上层应用的并行度和数据划分特征自动决定任务的划分方式、生成相应平台上的执行代码、组织具体的执行线程并管理数据，实现在不同运行时环境上的执行.

4 统一的并行编程接口

我们设计了数据关联计算描述语言实现了对数据关联计算模型的支持，为用户编程提供了统一的编程接口. 数据关联计算描述语言为数据、关联结构和计算定义了 C 语言兼容的语法结构，使得编程接口简便易用. 数据表示的语法结构将数据类型等属性参数化，提高了编程的灵活性，有利于上层应用的统一. 关联结构中为不同的算子定义了正交化的关联属性标识以充分表达计算任务中的并行性. 数据、关联和计算三种结构采用模块化设计，数据关联计算表达式定义了各个模块的组合规则，通过数据关联计算表达式组合三种模块结构，表示具体的计算任务. 本节最后展示了使用数据关联计算描述语言实现的矩阵相乘示例，并通过与 OpenCL 代码的对比说明了描述语言在并行编程中的特点.

4.1 数据

数据关联计算描述语言采用了高维向量作为计算数据的统一表示，通过离散化的数据表示为并行性的表达提供便利. 数据的声明格式如下：

DAC_data data_name([(*dim*0)]…[(*dim*N)]);

合法的数据声明包括 *DAC_data* 标识符和数据名，数据的维度和维度值为可选项. 其中 DAC 为数据(Data)、关联(Association)和计算(Computation/Calculation)的首字母缩写，代表本文提出的数据关联计算方法. 数据声明以 *DAC_data* 标识符开始，以区分数据和普通参数，只有数据可以作为计算核心操作的对象，普通参数作为计算核心的控制变量，通过 C 语言变量的声明格式进行定义.

数据类型不出现在数据声明中，而与数据的维度和各维度值一起作为数据的内建属性. 运行时系统负责为数据的各个属性创建内建变量，使得数据属性参数化. 数据类型参数化可以提高应用对数据类型的兼容性，避免数据类型变化时对应用进行改动，而参数化的数据维度和维度值可以作为控制变量同时方便并行度的计算，共同提高了编程的灵活性. 参数化的数据属性可以在程序其他部分进行调用，调用格式如表 2 所示.

表 2 数据属性调用格式

数据属性	*main* 函数中	计算模块中
数据类型	*.type*	*.type*
维度	*.dim*	不能使用
第 *i* 维维度值	*.range*[*i*]	*.ri*

由于计算模块处理的是其数据列表中声明的数据接口，因此数据维度属性的调用格式与在 *main*

函数中的调用有所不同以进行区别.

表 3 展示了数据关联计算描述语言中提供的三种附加数据操作结构. 在计算模块以外的程序中对数据的每次读和写操作都应当被 *DAC_fill* 和 *DAC_get* 结构所标记，根据对数据读写操作的标记，编译器和运行时系统就可以结合具体的运行时环境根据需要插入数据一致性操作并自动管理数据通信和同步. *DAC_shape* 用于对指定数据的属性进行补充或更新，其参数列表依次为数据名、数据类型以及数据的各维度值. 数据类型支持 C 语言标准类型以及用户自定义数据类型. 维度值的数量隐含了数据的维度，维度值可以为任意整型常量或变量，但必须在运行时可知.

表 3　附加数据操作结构

附加结构语法	功能
DAC_fill〈*data list*〉{…}	标记对计算数据进行写入的代码段
DAC_get〈*data list*〉{…}	标记对计算数据进行读取的代码段
DAC_shape(*name*,*type*,[*range*0,…]);	完善计算数据的数据类型和维度值属性

在 *main* 函数中通过 *DAC_shape* 接口可以更新数据的一个或多个维度值，或直接通过数据属性调用 *.range*[*i*]来修改数据的某个特定维度值，用户编程时可以根据具体的算法需求进行调用，提高编程的灵活性. 直接调用 *.range*[*i*]进行修改存在一定的风险，当修改值大于属性初始值时，会导致数据访问越界. 调用 *DAC_shape* 接口对维度值属性进行更新时，编译器生成相应的代码对修改值进行检查，若修改值超出属性初始值，程序会在执行时报错.

4.2　计算

数据关联计算描述语言将计算定义为一种没有返回值的函数，通过数据接口与外界进行数据交互，其内部的计算过程在运行时串行执行，在数据接口可见的数据范围上封闭，计算结果体现为输出数据. 通过在函数外部划分不同的数据范围，可以生成多个计算实例，每个计算实例独立对数据进行操作，实现并行执行.

为了与普通函数相互区别，自定义的计算函数均通过关键字 *DAC_calc* 进行标识，定义格式如下：

```
DAC_calc  calc_name(arguments)〈data list〉
{
    //Computing process;
}
```

计算定义中通过参数列表和数据列表来区分普通参数和计算数据. 数据列表确定了计算过程的数据接口，需要指定每个计算数据的维度. 只要数据符合计算核心接口的维度要求就可以使用计算核心进行处理而不受数据类型的限制. 编译器根据计算任务中的数据类型生成具体的执行代码，保证上层应用的统一.

计算过程主体使用 C 语言编程，同时可以通过 *.type* 调用数据的数据类型来定义中间变量，数据的各维度值也可以通过 *.ri* 调用获得. 数据类型和各维度值的参数化可以提高计算核心代码的通用性和模块化特性，在上层数据属性变化时可以由编译器根据具体的数据属性生成相应的执行代码，避免由用户手动修改，减轻用户编程负担，提高编程的灵活性和应用的稳定性.

4.3　关联结构

在数据表示和计算函数数据接口定义的基础上，关联结构描述了计算数据如何划分以符合计算数据接口的要求. 关联结构通过 *DAC_shell* 关键字标识，定义格式如下：

```
DAC_shell  shell_name()〈data list〉
{
    //Data association
}
```

关联结构的数据列表确定了关联结构的数据接口，同样需要指定计算数据的维度. 关联结构的主体则指定了数据之间的输入输出关系和各个数据的划分方法.

数据之间的输入输出关系通过标识符“⇔”标记，在标识符同一行的左侧为输入数据，在标识符同一行的右侧为输出数据，并且允许某一侧为空. 输出和输入数据根据计算过程中是否对数据有写入操作进行区分，只要有对数据的写入操作，即为输出数据，否则为输入数据. 输入输出关系为运行时自动的数据拷贝提供了依据.

表 4 为根据第 2 节中对关联结构的定义为输入输出数据设计的与三种算子相对应的关联属性标识，关联属性标识只出现在关联结构中.

表 4　关联属性标识

	输入	输出
关联属性标识	*IVs*、*sp*、*bg*	*IVs*、*sp*、*dup*、*atomic*

索引变量 *IVs*(Index Variables)与降维算子对应，是一种特殊的标识符，其遵循 C 语言变量的命名规则，但不需要声明和初始化，只用于标记数据的

某个维度.计算数据在运行时按照索引变量所在的维度进行降维展开,划分为低维度的子数据集合.同一索引变量可以多次使用以标记不同数据的不同维度;在运行时的同一计算实例中,被相同索引变量标记的不同维度的索引值相同;当计算数据为高维向量时,其不同的维度可以被不同的索引变量标记,数据将在多个维度上同时降维展开.

与分区算子对应的是 *sp* 标识符,用来标记数据可以划分为多份保持数据维度特征的小规模数据,其使用格式如下:

sp⟨data list⟩(post_process);

每个 *sp* 标识符可以标记多个数据,其后紧邻的尖括号之内的数据列表即其作用范围.后处理过程为可选项,只针对其作用范围内的输出数据.处于同一 *sp* 标识符作用范围内的数据在运行时按照相同的数量进行切分,不同数据的数据子集应按序对应;处于不同的 *sp* 标识符作用范围的数据,其切分方法相互独立,不同数据的数据子集之间可以自由组合.标识符 *sp* 的使用示例如下:

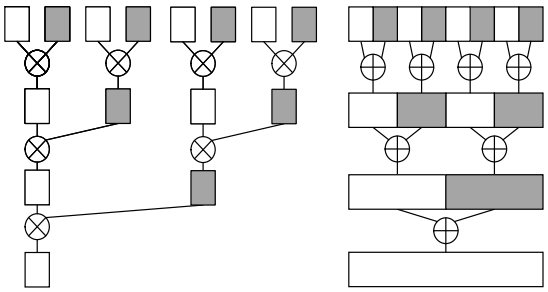
(1) *sp*⟨A,B⇔C⟩; (2) *sp*⟨A,B⟩⇔C;

如示例(1)所示,*sp* 标识符的作用范围可以跨越输入输出标识符.示例(1)和示例(2)是不等价的,示例(1)中,所有数据处于同一 *sp* 标识符作用范围内,因此须遵循相同的划分方法,保证划分后的子数据按序对应;示例(2)中,数据 A、B 和数据 C 处于不同 *sp* 标识符作用范围内,因此数据 C 的划分方法独立于数据 A 和 B.

分区算子将问题划分为多个小问题进行处理,而各个小问题的计算结果仅为中间结果,根据问题特征可以分为三种情况:第一种中间结果就是最终结果,第二种中间结果可以就地处理,第三种中间结果需要额外存储.对于第一种和第二种情况,输出数据可以通过 *sp* 标识符进行标记.针对第三种情况,我们设计了 *dup* 标识符,表示在运行时申请额外的空间为每个小问题的计算生成一份被标识的输出数据的备份以保存中间结果,*dup* 标识符使用格式如下:

dup⟨data list⟩(post_process);

对于中间结果的第二种和第三种情况,需要对中间结果进行后处理以生成最终的输出结果.后处理过程是一种特殊的计算函数,其数据列表中仅包含两个输入数据和一个输出数据,表示中间结果的一次处理过程.运行时系统根据标识符调用相应的后处理函数,按照图 3 的方式实现自动的并行后处理.



(a) *dup*模式 (b) *sp*模式

图 3 并行后处理模式

标识符 *sp* 默认对数据的最高维进行切分,现有的实现尚未支持对高维数据的不同维度同时切分.这一方面带来了算法实现上的简便,降低了编程难度,另一方面避免了数据多维划分导致的复杂数据聚合问题,降低了后处理的实现难度和性能开销.标识符 *sp* 主要标记粗粒度并行,可以通过适当增大切分粒度来减小对性能的影响.

对应于保形算子,我们根据数据的输入输出属性设计了不同的标识符 *bg* 和 *atomic* 以分别标记输入和输出数据,使用格式如下:

bg⟨data list⟩; *atomic*⟨data list⟩;

标识符 *bg* 和 *atomic* 自带输入输出属性,因此只能出现在输入输出标识符对应的一侧或在编程时单独占据一行.当输入数据不可切分时,多个并行实例共享数据,可能存在数据读取冲突,主要影响访存效率;但对于输出数据,多个并行实例同时进行写入操作会导致写冲突,影响数据一致性和结果的正确性.不同的标识符标记了不同的访存特性,为运行时相应的处理和优化提供了依据.

4.4 数据关联计算表达式

有了数据、关联结构和计算模块的语法定义,就需要把它们组合形成具体的并行计算任务,这就是数据关联计算表达式.其定义格式如下:

⟨data list⟩⇒shell_name(calc_name(arguments));

连接符"⇒"将数据、关联结构和计算核心这三大模块组合在一起.不同模块间的组合规则在于数据接口的匹配.首先定义:

(1) $data\_list$、$data\_list_{\text{shell}}$、$data\_list_{\text{calc}}$ 分别为数据关联计算表达式、关联结构和计算核心的数据列表,$num(data\_list_x)$ 为某个数据列表中数据参数的数量;

(2) δ 表示 $data\_list$ 中的某个数据,δ_{shell} 和 δ_{calc} 为该数据在 $data\_list_{\text{shell}}$ 和 $data\_list_{\text{calc}}$ 对应的形参,$dim(\delta_x)$ 表示某个数据或形参的维度;

(3) δ'_{shell} 为 δ_{shell} 在运行时根据关联属性标识处理后的数据，其维度的计算为

$$dim(\delta'_{\text{shell}})=\begin{cases}dim(\delta_{\text{shell}})-x, & \text{被索引变量标识}\\ dim(\delta_{\text{shell}}), & \text{被其他属性标识}\end{cases},$$

其中，x 为 δ_{shell} 中被索引变量标识所有维度的数量.

根据以上定义，数据关联计算表达式的组合规则定义如下：

(1) $num(data\_list)=num(data\_list_{\text{shell}})=num(data\_list_{\text{calc}})$；

(2) $\forall\delta\in data\_list$，有 $dim(\delta)=dim(\delta_{\text{shell}})$，$dim(\delta'_{\text{shell}})=dim(\delta_{\text{calc}})$.

其中，规则(1)表示数据关联计算表达式、关联结构与计算模块的数据列表中参数数量必须相等. 规则(2)表示数据关联计算表达式数据列表中的每个数据参数的维度必须与关联结构的数据列表中对应参数的维度相等；经过关联结构标记的每个数据参数在运行时的维度必须与计算核心数据列表中对应数据参数的维度相等.

4.5　矩阵相乘实现示例

图 4 示例为使用数据关联计算描述语言实现的矩阵相乘代码. 我们在示例中尽可能多地展示了不同语法扩展和标识符的使用. 与串行代码相比，额外的代码开销主要在于添加了关联结构，而关联结构语法简单易用，开销很小，同时计算模块的编程得以串行化，可以有效利用已有代码，提高编程效率.

示例的第 4 行展示了常见的数据声明的不同方式，其中数据 a 声明为 10×100 的二维向量，数据 b 只声明了维度未指定具体的维度值，数据 c 未指定任何数据属性，第 5～7 行使用 *DAC_shape* 接口对各数据的属性进行了补充. 示例的第 8～16 行和 18～21 行分别使用了 *DAC_fill* 和 *DAC_get* 对数据读写操作进行标记. 同时 *DAC_fill* 所标记的内容里，通过 *.range* 对数据的维度值进行了调用.

图 4 示例的 30～40 行是矩阵相乘的计算核心：向量相乘. 如示例的 33 行和 35 行所示，其计算过程中可以通过 *.type* 调用数据的数据类型来定义中间变量，数据的维度值也可以通过 *.ri* 调用获得并用于循环控制.

图 4 示例的 25～28 行为关联结构的实现. 通过两个索引变量 i 和 j，标记矩阵 ***A*** 按行展开，矩阵 ***B*** 按列展开，矩阵 ***C*** 对应展开为标量，展开后的标量的索引取决于矩阵 ***A***、***B*** 展开后的行列向量索引的组合. 索引变量 i 所在的维度值为 10，索引变量 j 所在的维度值为 10，标记应用最大数据并行度为 10×10 即 100，具体的映射执行则由运行时系统自动完成.

```
1.  int main(void)
2.  {
3.      int i,j;
4.      DAC_data a[10][100],b[][],c;
5.      DAC_shape(a,int);
6.      DAC_shape(b,int,100,10);
7.      DAC_shape(c,int,10,10);
8.      DAC_fill⟨a,b⟩
9.      {
10.       for(i=0;i<a.range[0];i++)
11.           for(j=0;j<a.range[1];j++)
12.           {
13.               a[i][j]=1;
14.               b[j][i]=2;
15.           }
16.     }
17.     ⟨a,b,c⟩⇒mtov(vm);
18.     DAC_get⟨c⟩
19.     {
20.         printf("%d\n",c[5][5]);
21.     }
22.     return 0;
23.  }//end main
24.  //DAC_shell example
25.  DAC_shell mtov()⟨a[][],b[][],c[][]⟩
26.  {
27.     ⟨a[i][],b[][j]⟩⇔⟨c[i][j]⟩;
28.  }
29.  //DAC_calc example
30.  DAC_calc vm()⟨a[],b[],c⟩
31.  {
32.     int i;
33.     a.type num;
34.     num=0;
35.     for(i=0;i<a.r0;i++)
36.     {
37.       num+=a[i]*b[i];
38.     }
39.     c=num;
40.  }
```

图 4　矩阵相乘的数据关联计算描述语言实现示例

图 4 示例的第 17 行为数据关联计算编程的核心：数据关联计算表达式. 表达式通过关联结构 *mtov* 将数据 a、b、c 关联于计算核心 *vm*. 数据 a、b、c 满足关联结构的数据接口要求，经过索引变量的降维展开，和计算核心 *vm* 的数据接口一一对应.

图 5 展示了使用 OpenCL 实现的矩阵相乘代码. 如图 5 所示，OpenCL 编程中需要用户逐步进行包括平台、上下文、设备和命令队列在内的执行环境配置，通过内存对象进行数据拷贝，设置内核执行参数，设置内核执行线程空间以及数据拷回等一系列工作，这些工作均需要通过 OpenCL 提供的编程接口显式完成. 此外，线程索引也需要出现在内核中，因此内核编程与内核执行线程空间的设置息息相关. 其编程逻辑复杂，编程接口学习使用成本较高，应用开发工作量大，且不能隔离底层硬件细节，无法保证上层应用的统一. 与之相比，UPPA 架构简化了

```
1.  int main(void)
2.  {
3.    int i,j;
4.    int *a,*b,*c;
5.    cl_platform_id platform;
6.    cl_context context;
7.    cl_device_id device;
8.    cl_command_queue queue;
9.    cl_program program;
10.   cl_kernel kernel;
11.   cl_mem dA,dB,dC;
12.   cl_int clStatus;
13.   clGetPlatformIDs(1,&platform,NULL);
14.   cl_context_properties clCps[3]={CL_CONTEXT_PLATFORM,
                    (cl_context_properties)platform,0};
15.   context=clCreateContextFromType(clCps,CL_DEVICE_
                    TYPE_GPU,NULL,NULL,&clStatus);
16.   clGetDeviceIDs(platform,CL_DEVICE_TYPE_GPU,1,&device,
                    NULL);
17.   queue=clCreateCommandQueue(context,device,
                CL_QUEUE_PROFILING_ENABLE,&clStatus);
18.   const char* source[]={readFile("kernel.cl")};
19.   program=clCreateProgramWithSource(context,1,source,
                                    NULL,&clStatus);
20.   clBuildProgram(program,1,&device,NULL,NULL,NULL);
21.   kernel=clCreateKernel(program,"mm",&clStatus);
22.   a=(int*)malloc(sizeof(int)*10*100);
23.   b=(int*)malloc(sizeof(int)*100*10);
24.   c=(int*)malloc(sizeof(int)*10*10);
25.   for(i=0;i<10;i++)
26.     for(j=0;j<100;j++)
27.     {
28.       a[i*100+j]=1;
29.       b[j*10+i]=2;
30.     }
31.   dA=clCreateBuffer(context,CL_MEM_READ_ONLY,
                    sizeof(int)*1000,a,&clStatus);
32.   dB=clCreateBuffer(context,CL_MEM_READ_ONLY,
                    sizeof(int)*1000,b,&clStatus);
33.   dC=clCreateBuffer(context,CL_MEM_WRITE_ONLY,
                    sizeof(int)*100,c,&clStatus);
34.   clEnqueueWriteBuffer(queue,dA,CL_FALSE,0,
                    sizeof(int)*1000,a,0,NULL,NULL);
35.   clEnqueueWriteBuffer(queue,dB,CL_FALSE,0,
                    sizeof(int)*1000,b,0,NULL,NULL);
36.   int lda=100;
37.   int ldb=10;
38.   clSetKernelArg(kernel,0,sizeof(cl_mem),(void*)&dA);
39.   clSetKernelArg(kernel,1,sizeof(int),(void*)&lda);
40.   clSetKernelArg(kernel,2,sizeof(cl_mem),(void*)&dB);
41.   clSetKernelArg(kernel,3,sizeof(int),(void*)&ldb);
42.   clSetKernelArg(kernel,4,sizeof(cl_mem),(void*)&dC);
43.   size_t globalSize[2]={10,10};
44.   clEnqueueNDRangeKernel(queue,kernel,2,NULL,globalSize,
                    NULL,0,NULL,NULL);
45.   clFinish(queue);
46.   clEnqueueReadBuffer(queue,dC,CL_TRUE,0,
                    sizeof(int)*100,c,0,NULL,NULL);
47.   printf("%d\n",c[5][5]);
48.   clReleaseKernel(kernel);
49.   clReleaseProgram(program);
50.   clReleaseMemObject(dA);
51.   clReleaseMemObject(dB);
52.   clReleaseMemObject(dC);
53.   clReleaseCommandQueue(queue);
54.   clReleaseContext(context);
55.   return 0;
56. }//end main
//kernel.cl
57. __kernel void mm(__global int *A, int lda,
                    __global int *B, int ldb,
                    __global int *C)
58. {
59.   int c=0;
60.   int m=get_global_id(0);
61.   int n=get_global_id(1);
62.   for (int i=0; i<lda; ++i) {
63.     c+=A[m*lda+i]*B[i*ldb+n];
64.   }
65.   C[m*ldb+n]=c;
66. }
```

图 5 矩阵相乘的 OpenCL 实现示例

编程逻辑,编程接口简单易用,用户编程时无需关注底层实现细节,具体的映射执行由编译和运行时系统自动完成,有效降低了异构编程的难度和工作量,实现高层统一编程.

5 跨平台的编译与运行时原型系统

为了验证数据关联计算编程模型和描述语言的有效性,本文实现了基于 OpenCL 的编译和运行时原型系统,以 OpenCL 为中间语言实现了高层应用向不同异构系统的跨平台映射执行,如图 6 中深色部分所示.虽然目前的工作以 OpenCL 为中间语言,但编译与运行时系统的实现并不仅限于 OpenCL.如图 6 所示,高层数据关联计算应用在进行编译前端并行解析后,可以通过不同的编译后端映射到不同

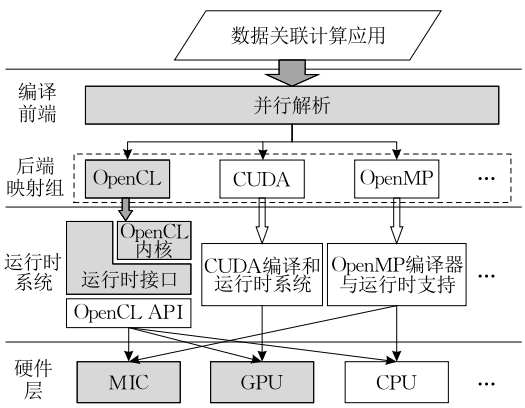


图 6 编译和运行时系统架构设计与实现

的底层编程方法,如 CUDA、OpenMP 等,以实现在更多平台上的优化执行.OpenCL 和 CUDA 有着一致的编程模型且编程接口更加复杂,以 OpenCL 为中间语言的实现也验证了高层应用向 CUDA 映射

的潜力. 以 CUDA 和 OpenMP 为后端的源到源编译器将在进一步工作中实现，本节则介绍以 OpenCL 为中间语言的源到源编译器和运行时原型系统实现细节.

在实现中，运行时系统通过精简的运行时接口封装了 OpenCL 的运行时 API，简化了 OpenCL 的编程逻辑，降低源 3 到源编译器的实现难度. 源到源编译器负责解析数据关联计算应用中的相关语法结构和标记，生成符合 OpenCL 标准的内核(kernel)代码并结合运行时系统提供的接口生成主机端(host)代码，由运行时系统负责具体计算任务的执行. 运行时系统根据编译器生成的运行时函数调用结合具体的硬件特征实现自动的线程映射和数据管理.

5.1　运行时接口

运行时原型系统通过精简的运行时接口函数对 OpenCL 的运行时 API 进行了封装，简化了 OpenCL 的编程逻辑. 运行时系统的接口函数分为主接口和次级接口，主接口对源到源编译器可见，次级接口由主接口函数进行调用.

主接口函数如表 5 所示. 接口 *chk_Env* 检查系统内的平台和设备信息，创建 OpenCL 执行环境，包括创建内核执行的上下文、命令队列并编译内核程序. 接口 *dealcomptRT* 则负责内核的具体执行，包括内核创建、参数设置、内核执行线程配置和数据管理. 接口 *chkts* 负责检查并更新相应数据对象的时间戳. 接口 *chkhostdata* 一般和 *chkts* 配合使用，负责检查主机端数据并将计算结果从设备端拷回.

表 5　运行时主接口函数

主接口	功能	封装 OpenCL APIs
chk_Env	检查系统并创建 OpenCL 执行环境	clGetPlatformIDs clGetDeviceIDs clGetDeviceInfo clCreateContext clCreateProgramWithSource clBuildProgram clGetProgramBuildInfo clCreateCommandQueueWithProperties
dealcomptRT	创建并管理 OpenCL kernel 执行	clCreateKernel clSetKernelArg clEnqueueNDRangeKernel clFinish
chkts	检查并更新时间戳	未包含 OpenCL API
chkhostdata	检查主机端数据并完成数据拷回	clEnqueueReadBuffer

次级接口如表 6 所示. 接口 *chk_Env* 调用接口 *prog_Src* 将内核文件内容读入内存以编译内核程序. 接口 *chkdata* 负责设备端的数据管理，接口 *dealcomptRT* 调用 *chkdata* 以准备内核执行的数据对象.

表 6　运行时次级接口

次级接口	功能	封装 OpenCL API
prog_Src	将内核文件读入内存，由 *chk_Env* 进行调用	未包含 OpenCL API
chkdata	检查并创建设备端数据对象，完成数据拷入，由 *dealcomptRT* 进行调用	clCreateBuffer clEnqueueWriteBuffer clEnqueueCopyBuffer

运行时接口将 OpenCL 的主要编程逻辑简化为 *chk_Env* 和 *dealcomptRT* 两个接口函数，大大简化了编译器的工作. 编译器只需要生成对 *chk_Env* 的一次调用即可完成 OpenCL 执行环境的设置. 数据关联计算表达式中将计算数据和计算核心关联在一起，在编译生成时，只需要将数据和相应的内核函数作为参数传递给 *dealcomptRT* 接口函数，通过生成 *dealcomptRT* 调用完成计算任务.

5.2　编译过程

数据关联计算描述语言到 OpenCL 的源到源编译器使用 C 语言编写，根据高层应用代码完成并行解析并生成相应的主机端文件和内核文件，主机端文件包含符合 OpenCL 标准的主机端程序，内核文件由各个内核函数代码组成. 高层应用中的数据关联计算表达式包含了与计算任务相关的计算过程和所需的数据，源到源编译器的工作即围绕其展开.

5.2.1　内核生成

编译器根据数据关联计算表达式可以解析出对应的计算核心，这是生成内核函数的基础. 根据计算核心与 OpenCL 内核的区别，内核函数的生成主要包括三方面的工作：内核函数参数生成、数据属性的解引用和数据索引的下标转换.

在内核函数生成时，除了计算核心的普通参数外，参与计算的每个数据及其各个维度值都将作为内核函数的参数. 此外，关联属性标识也会影响内核函数的参数. 由于 OpenCL 标准中内核执行的线程索引空间 NDRange 的最大维度为 3，根据关联属性标识的数量最多选取前三个索引变量或 *sp* 标识进行线程映射，在内核函数的计算过程中最多也只能插入三个线程索引函数调用. 由于索引变量改变了数据的维度，通过对关联结构的解析找到所有独立的索引变量和 *sp* 标识后，超过三个的索引变量将转化为内核函数的参数；而 *sp* 标识不改变数据的维度，因此超过范围的 *sp* 标识不影响内核生成，可以直接略过. 线程索引空间 NDRange 及根据关联属

性标识进行的线程映射将在 5.3 节中详细阐述.

在对计算过程的处理中,每个.*type* 引用需要使用具体的数据类型进行替换,对数据维度值的.*ri* 引用也替换为内核函数参数中对应的数据维度值.由于 OpenCL 的标准中内核函数不支持任意维度的高维向量,因此需要根据数据的高维索引和各维度值参数完成高维向量到一维向量的下标转换.

在编译过程中,我们为每个计算核心设置了一个内核函数链表,记录已经生成的内核函数名及其对应的数据类型属性和关联结构.内核函数名由计算核心名和序列号组成,序列号对应内核函数的生成次序.对于每个数据关联计算表达式都要搜索对应计算核心的内核链,根据数据类型属性和关联结构进行检查,若符合要求的内核函数已生成则返回对应内核函数名,否则需要生成新的内核函数.图 7 展示了由图 4 示例生成的内核函数代码.

```
__kernel void vm1(__global int *a, int a_r0, int a_r1,
                  __global int *b, int b_r0, int b_r1,
                  __global int *c, int c_r0, int c_r1)
{
size_t tID0,tID1;
tID0=get_global_id(0);
tID1=get_global_id(1);
    int i;
    int num;
    num=0;
    for(i=0;i<a_r1;i++)
    {
        num+=a[tID0*a_r1+(i)]*b[(i)*b_r1+tID1];
    }
    c[tID0*c_r1+tID1]=num;
}
```

图 7 向量相乘计算编译生成的内核函数代码

5.2.2 主机端文件生成

主机端文件的生成主要围绕着如何组织内核执行而展开.编译器的主要工作是在主机端程序开始处生成 *chk_Env* 调用,创建 OpenCL 内核执行环境,同时根据数据关联计算表达式解析参与计算的数据和内核函数名,设置 *dealcomptRT* 调用的参数,生成相应的 *dealcomptRT* 调用.

在生成 *dealcomptRT* 调用之外,编译器需要生成相应的数据管理代码来管理主机端和设备端的数据移动.编译器通过解析数据声明和相关的 *DAC_shape* 结构,按照解析的数据类型和维度值信息生成符合 OpenCL 标准的主机端数据空间的分配代码、数据各项属性的初始化代码并完成数据时间戳的设置.*DAC_fill* 结构标记了主机端数据写入操作的位置,在写入操作结束的位置,编译器需要为每个被写入的数据添加 *chkts* 调用,更新数据的时间戳.编译器通过 *DAC_get* 结构标记找到主机端数据读取操作的位置,在数据读取操作之前添加 *chkhostdata* 调用,检查每个需要进行读取的数据,自动管理从设备端到主机端的数据拷回.

生成后的运行时代码与运行时接口库经过平台上的 OpenCL 宿主编译器的编译后可以直接执行.虽然源到源编译器在代码生成时没有对源代码中的数据类型进行检查,但运行时代码中如数据类型等基础语法的正确性可以由 OpenCL 的宿主编译器保证.

5.3 执行管理

内核执行由运行时接口函数 *dealcomptRT* 自动完成.在内核的执行过程中,需要组织内核执行的线程完成线程映射并管理主机端和设备端的数据移动.由于本文运行时系统的首要目标是验证数据关联计算编程方法的可行性,为了降低实现的难度,在接口 *dealcomptRT* 的实现中,默认选取系统中第一个 OpenCL 平台下的第一个计算设备,在执行时独占整个设备.对多平台多设备的支持将在进一步的工作中进行研究实现.

5.3.1 线程映射

异构众核处理器高并发和高吞吐量的特点要求应用执行时能够生成尽量多的线程以隐藏访存延迟,充分利用处理器中的大规模计算单元.在 OpenCL 中,线程组织由线程索引空间 NDRange 的设置决定.线程映射的任务就是根据应用最大并行度设置处理器支持范围内的 NDRange.

根据 OpenCL 标准,线程索引空间 NDRange 最高为三维,各维度值的乘积即为运行时的线程数量,因此最多可选取三个关联属性标识进行线程映射.独立的关联属性标识数量小于等于 3 时,标识的数量决定了线程索引空间的维度,每个标识对应的数据并行度决定了 NDRange 每个维度的维度值.当属性标识的数量大于 3 时,需要进行选取.运行时系统优先选择索引变量进行线程映射,因为索引变量通常对应较高的数据并行度,可以生成更多的线程进行计算,有利于大规模并行处理器计算效能的发挥,提高应用的执行性能.

对于选取结束后冗余的关联属性标识,需要分情况进行处理.对于索引变量,由于其标记了数据的某个维度,影响了计算过程和数据接口,因此将超出的索引变量变为内核函数的参数,通过设置外围循

环，进行多次内核调用并依次传入不同的值，保证内核执行的正确性. 对于 *sp* 标识，由于其不改变数据的维度，对计算过程的编程无影响，被其标记的数据也不再进行切分.

对于每个 *sp* 标识，运行时默认设置的切分参数为 64，切分参数决定了 *sp* 标识所提供的数据并行度. 根据切分参数设置 NDRange 的对应维度值，影响了运行时的线程数量. 运行时系统根据切分参数对数据的最高维度进行切分，划分每个线程处理的数据范围，当切分参数超过数据最高维的维度值时，需要进行检查并进行二分退让，保证切分参数为 2 的幂值.

5.3.2　数据管理

由于运行时系统仅为验证方法的有效性，仅支持单设备执行，对超出单设备内存容量的大规模计算数据的自动划分机制将在进一步工作中实现.

OpenCL 的内存模型中，设备端需要通过特殊的内存对象访问计算数据. 当存储架构分离时，在内核执行前需要将最新的计算数据从主存中拷入设备内存中，计算完成后需要将数据从设备端拷回主存. 为了减少数据拷贝的次数，最大程度地复用数据，提高数据本地性，我们设计了基于时间戳的数据管理机制，保证只在必要的时候进行数据移动，其规则为：

(1) 在计算数据初始化和设备端内存对象创建时，初始化其时间戳；

(2) 在主机端对计算数据进行写操作后，检查计算数据和相应内存对象的时间戳，设置计算数据的时间戳为最新且大于原有的最大时间戳；

(3) 在内核执行之前，检查计算数据及其对应的内存对象，若内存对象不存在则创建相应内存对象，检查输入数据和对应的内存对象的时间戳，若内存对象的时间戳不是最新的，则进行数据拷贝，并更新内存对象时间戳与相应计算数据相同；

(4) 内核执行后，检查输出数据和相应的内存对象的时间戳，将内存对象的时间戳设置为最新且大于原有最大时间戳；

(5) 在主机端对计算数据进行读操作前，检查计算数据和相应内存对象的时间戳，如果计算数据的时间戳不是最新，则需要进行数据拷贝，并更新计算数据的时间戳与相应内存对象相同.

5.3.3　同步操作

OpenCL 的执行模型中提供了两级同步机制：工作组同步和命令队列同步. 工作组同步实现了同一工作组内工作项之间的同步，命令队列同步则通过事件对象、发送阻塞命令、设置栅栏或调用 *clFinish* 函数等手段设置同步点，实现命令队列中不同命令之间的同步.

数据关联计算模型保证了生成的并行计算实例之间是独立的，因此在实现中避免了工作组内部的工作项同步. 在 *dealcomptRT* 接口的实现中封装了 *clFinish* 函数以进行命令队列同步，在内核执行前后和 *dealcomptRT* 接口退出之前均插入了对 *clFinish* 函数的调用，设置了同步点，在内核执行前后均对命令队列中的命令进行同步，保证内核执行的正确性，在 *dealcomptRT* 执行结束后进行同步，以确保后续执行代码的正确执行.

6　实验测试

为了验证 UPPA 架构的有效性，本文从异构并行编程测试用例集 Parboil① 和 Rodinia[23] 中选取了典型应用，展示了如何使用数据关联计算方法对应进行重构. 通过对比重构代码、串行代码和测试集提供的 OpenCL 代码的代码量直观展示了数据关联计算方法在降低异构编程工作量方面的效果. 重构代码经过源到源编译就可以在 GPU 和 MIC 两种异构平台执行，不需要进行任何代码移植或改动. 通过在不同异构平台上的测试验证了 UPPA 架构的跨平台可移植性，并通过与测试集提供的 OpenCL 代码的性能对比检验了编译与运行时系统的效率. 实验表明，UPPA 架构能够有效地简化异构编程、提高编程效率，实现高层统一编程.

6.1　基于数据关联计算方法的测试用例重构

测试用例 SGEMM、SpMV、Stencil、Histo 和 BFS 从 Parboil 测试集中选取，测试用例 NN 和 Kmeans 选取自 Rodinia 测试集. 重构工作均从测试集提供的串行代码开始，根据数据关联计算描述语言要求修改数据表示，添加数据操作接口调用并设计相应的关联结构实现并行表达，无需对主体算法实现进行改动，重构工作量小且编程简单.

(1) SGEMM

稠密矩阵相乘 SGEMM 常作为基本测试用例

① The IMPACT Research Group Parboil Benchmarks, http://impact.crhc.illinois.edu/Parboil/parboil.aspx 2019.4.23

以测试系统性能或者优化方法的效果.本文 3.3 节和 4.5 节的示例展示了使用数据关联计算方法和描述语言实现的矩阵相乘,通过两个索引变量将矩阵相乘的计算降维成并行向量相乘的计算.

(2) SpMV

稀疏矩阵向量相乘 SpMV 采用了 JDS 格式存储稀疏矩阵,计算过程包括两层循环:外围循环为每个输出向量元素查找对应的矩阵行向量在数组中的位置,内层循环中完成向量相乘的计算.实现中通过索引变量将外围循环展开,内层向量相乘的计算过程关联于输出向量元素和存储稀疏矩阵每行元素数量的数组,其他数组通过 *bg* 属性标记,由各个计算过程共享.

(3) Stencil

测试用例 Stencil 实现了 3D 结构化网格热方程的迭代式雅可比求解器.每个节点的迭代计算都依赖其坐标相邻的节点,并行度由需要进行迭代的节点数量决定.为了保证坐标信息的传入,重构代码中设置了三个坐标数组,一组坐标和一个输出网格节点唯一确定一次计算过程,通过索引变量将 3D 网格降维到一个节点.根据关联结构的定义,同一数据不能同时作为输入和输出,因此重构代码中使用了两个网格来进行交替迭代.输入网格的关联属性为 bg,对所有的计算过程可见.

(4) Histo

测试用例 Histo 为图像直方图统计.由于每个输入值可能和任意的输出值相关,只能通过划分输入数据将问题拆分成多个小问题来并行处理,因此输入数据的关联属性为 *sp*.输出矩阵的关联属性可以设置为 *dup* 以指示编译和运行时系统申请额外的空间存放各子集的中间结果,也可以设置为 *atomic*.

(5) BFS

广度优先搜索 BFS 是图计算中的常用算法,通常设置有两个搜索队列,从当前搜索队列中依次取出节点进行搜索,将所有找到但未完成搜索的节点放到下一搜索队列中,迭代这个过程直到搜索队列为空.其并行度取决于当前搜索队列的长度,处于动态变化中,通过索引变量使得队列中的每个节点可以并行地进行搜索.由于并行找到的节点要写入同一队列中,写入位置未知,因此需要为输出队列添加 *atomic* 标识,每次迭代完成需要通过 *DAC_shape* 接口或 *.range* 调用重置当前队列长度.

(6) NN

最近邻 NN 实现的应用场景是计算一系列台风到特定经纬度的最近距离.输入集合由各个台风的经纬度组成,输出集合是每个台风的经纬度与特定经纬度之间的欧氏距离.在重构实现中,通过一个索引变量标记输入和输出数据,将计算展开为对两个点之间欧式距离的计算,并行模式明确.

(7) Kmeans

Kmeans 是一种在数据挖掘中广泛使用的聚类算法,通过将每个数据点与其最近的聚类相关联,计算新的聚类质心并迭代直到收敛来识别相关的数据点.每次迭代的计算核心为计算每个数据点的特性与当前各个聚类中心点特性的相关性,记录相关性最大的聚类中心点.实现中通过一个索引变量标记输入数据点集和记录每个数据点聚类中心的输出集合,将数据集降维到每个数据点,同时使用 *bg* 标记当前聚类中心集合.

表 7 展示了各测试用例的应用领域与实现模式.UPPA 架构面向异构众核系统,主要适用于可以大规模并行加速的科学计算应用,通过对不同领域典型应用的重构表明,数据关联计算模型和描述语言能够有效的表达应用中不同模式的并行性,可以用于实现计算密集型应用,有一定的通用性.

表 7 测试用例应用领域和实现模式

测试用例	应用领域	实现模式
SGEMM	数值线性代数	*IVs-IVs*
SpMV	科学计算	*IVs*,*bg-IVs*
Stencil	结构化网格	*bg-IVs*
Histo	统计学、图像处理	*sp-dup/atomic*
BFS	图计算	*IVs-atomic*
NN	统计学、机器学习	*IVs-IVs*
Kmeans	数据挖掘	*IVs*,*bg-IVs*

6.2 代码量对比

得益于编程抽象的提高、编程逻辑的简化和简便易用的语法设计,UPPA 架构可以有效地提高异构系统上的并行编程效率.

虽然编程效率本身难以直接量化对比,但在实验中可以通过直观的编程代码量对比侧面反映.如图 8 所示,使用数据关联计算描述语言重构的代码量与测试集提供的 C/C++串行版本实现相当,而远远小于测试集 OpenCL 代码,有效降低了异构并行编程的工作量.

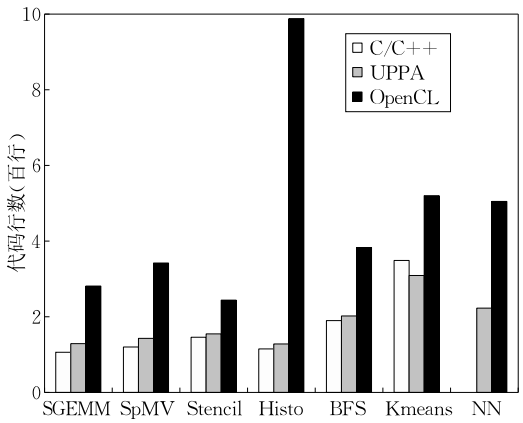


图 8　测试用例各版本实现代码量对比

表 8 展示了使用不同方法实现的计算核心的代码量. 数据关联计算模型使得计算模块的编程串行化，简化了编程逻辑的同时还能兼容已有串行代码. 因此使用本文方法不仅降低了整体编程的代码量，同时也简化了计算核心的实现. 相比于基于函数式编程的 Lift 方法，本文方法在减少编程工作量方面也有一定的优势.

表 8　不同方法计算核心实现代码量(单位：行)

测试用例	DAC_calc	OCL kernel	Lift
SGEMM	9	12	
SpMV	14	20	
Stencil	12	17	
Histo	9	450	
BFS	25	78	
NN	3	17	7
Kmeans	23	48	25

6.3　跨平台可移植性分析

本节通过在 GPU 和 MIC 两种平台上与 Parboil 和 Rodinia 测试集提供的 OpenCL 代码的对比测试来验证分析 UPPA 架构的跨平台可移植性. 由于 UPPA 架构主要解决异构系统上的高层统一编程问题，性能不是首要目标. 测试平台环境配置如表 9 所示.

表 9　测试平台环境配置

	GPU 平台	MIC 平台
操作系统及内核	CentOS 6.9 2.6.32-696.el6.x86_64	RHEL Server 6.3 2.6.32-279.el6.x86_64
处理器	Intel Xeon E5620 NVIDIA Tesla C2050	Intel Xeon E5-2670 Intel Xeon Phi 7110P
安装包	CUDA 6.5	Intel OpenCL runtime 14.2
版本号	OpenCL 1.1	OpenCL 1.2
编译器	GCC 4.4.7	GCC 4.4.6

在 Parboil 测试集提供的数据集的基础上，我们对测试用例 SGEMM、Stencil、Histo 的测试数据进行了补充，测试用例 NN、Kmeans 的测试数据由 Rodinia 测试集提供的脚本自动生成，最终每个测试用例均在小、中、大三种典型规模的数据上进行了对比测试，以检验 UPPA 架构在不同数据规模下的性能和可扩展性. 数据集特征如表 10 所示.

表 10　各测试数据集特征

测试用例	数据集 small	数据集 medium	数据集 large
SGEMM	M,N,K：128×160×96	M,N,K：1024×1056×992	M,N,K：2048×2048×1024
SpMV	稀疏矩阵(JDS 格式)1138×19	稀疏矩阵(JDS 格式)11948×49	稀疏矩阵(JDS 格式)146689×49
Stencil	网格规模 128×128×32	网格规模 512×512×64	网格规模：512×512×512
Histo	图像高×宽：996×1040	图像高×宽：1992×2080	图像高×宽：4096×4096
BFS	纽约市路图	节点数 1.25e6 的近规则图	节点数 1e6 的非规则图
NN	节点数 8e6	节点数 1.6e7	节点数 3.2e7
Kmeans	聚类数 5，节点数 2e6	聚类数 5，节点数 4e6	聚类数 5，节点数 8e6

虽然 OpenCL 提供了编程接口的统一标准，但在编程时需要用户根据底层硬件显式配置内核执行环境，在底层硬件变化时首先需要对这一部分进行改动. 测试集 OpenCL 代码均是针对 GPU 平台或兼容 CPU 平台的实现，因此必须人工修改以移植到 MIC 平台. OpenCL 在不同硬件上的执行依赖于硬件厂商提供的运行时支持，不同运行时实现细节上的不同也会影响 OpenCL 应用的跨平台执行. 如测试用例 Histo 的 OpenCL 代码移植到 MIC 平台后无法正确执行，需要调试并修改编程接口的调用. 此外，由于靠近底层的编程接口向用户暴露了硬件细节，内核的编写和执行参数设置都会受到底层硬件的影响，当底层硬件变化时会导致应用执行性能的损失，影响上层应用的统一. 使用本文的方法则不需要对重构后的代码进行改动，只需要在相应平台上进行编译即可，编译和运行时系统隐藏了底层平台细节，保证了上层应用的统一和跨平台.

通过在 GPU 平台上的测试发现，Parboil 测试集提供的 OpenCL_nvidia 版本和 OpenCL_base 版本代码的性能没有明显差异，因此为了测试对比的统一以及应用跨平台特性分析的便利，测试中均采用了

OpenCL_base 版本的代码. Rodinia 测试集则只提供了一种 OpenCL 实现. 两个测试集的 OpenCL 代码都主要针对 GPU 平台进行了优化. 其中内核执行的线程块大小优化在 SGEMM、Stencil、BFS、NN 和 Kmeans 中均有应用. SpMV 使用本地存储以提高访存效率,并采用了预取技术来隐藏访存延迟. Stencil 结合了 2D 分块技术和寄存器拼接技术进行优化. 测试用例 Histo 的 OpenCL 实现根据数据集大致遵循高斯分布的特点改进了算法,设计了四个内核函数来提高分布中心附近数据的吞吐量. BFS 在采用本地存储优化之外,在搜索队列较小时,只创建一个线程块执行,以提高执行效率. 重构代码则全部基于测试用例给出的串行代码进行实现,源到源编译器和运行时系统自动完成 OpenCL 代码的生成和优化执行.

图 9 展示了生成代码和测试集代码的测试对比结果. 图例中的 OCL 为 OpenCL 的简写. 如图 9 所示,自动生成的代码在两种平台上的性能分别能够达到测试集 OpenCL 代码的 91%～100%和 76%～98%,性能接近或相当.

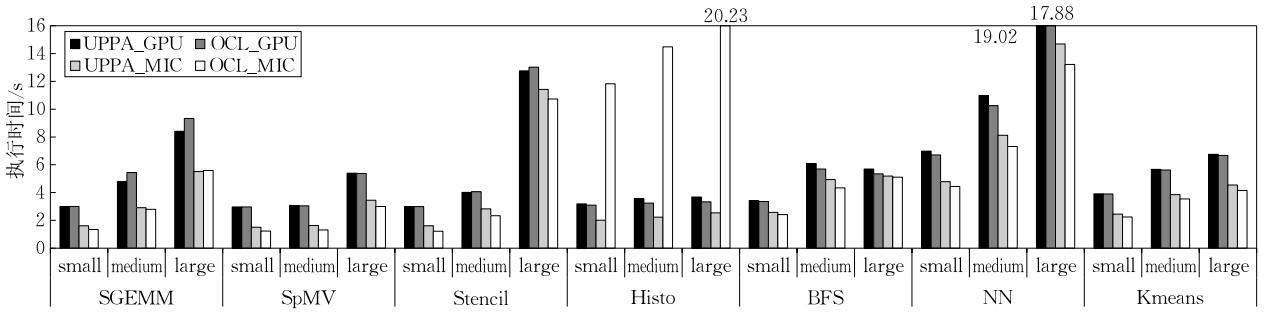


图 9 GPU 和 MIC 平台上生成代码和测试集 OpenCL 代码整体执行时间对比

图 9 中测试用例 Histo 展示的是采用 *sp-atomic* 模式重构后的生成代码性能. 由于输出数据规模较大,采用 *sp-dup* 模式需要将多个输出数据集进行合并,会引入较大开销. 采用 *sp-dup* 模式重构的代码 small 数据集下的执行时间在 GPU 和 MIC 平台上的分别达到了 13.08 s 和 8.62 s,相较之下 *sp-atomic* 模式更适合测试用例的实现.

图 10 和图 11 分别展示了生成代码在 GPU 平台和 MIC 平台上的对测试集代码的分项加速比. 其中 IO 项为应用 IO 性能;Copy 项对应主机端和设备端之间的数据拷贝;Kernel 项代表了内核执行性能,Ocl 项统计了各种 OpenCL API 调用的时间.

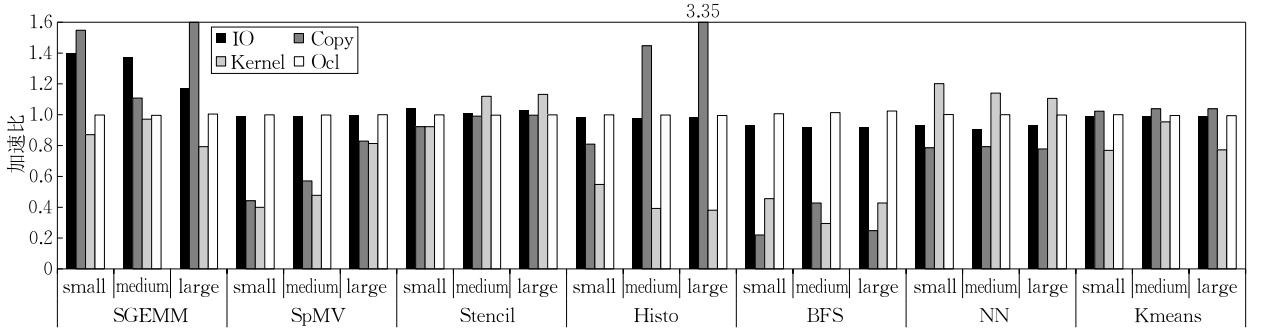


图 10 GPU 平台上各测试用例生成代码的分项加速比

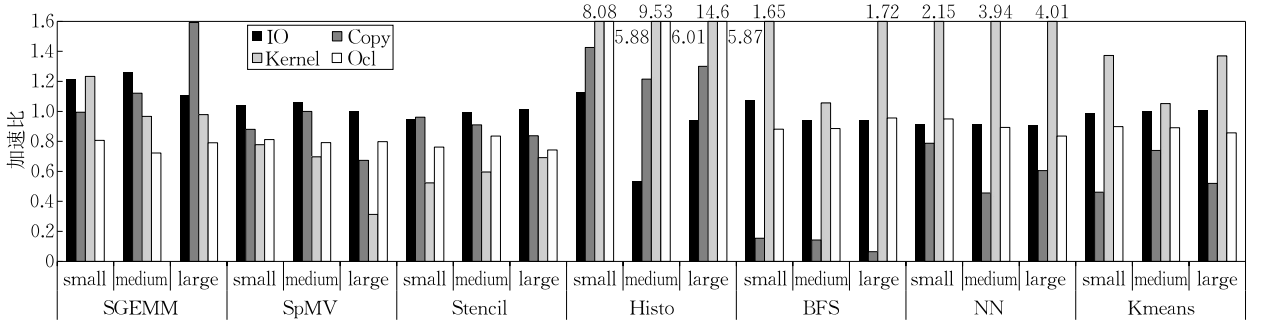


图 11 MIC 平台上各测试用例生成代码的分项加速比

SGEMM 在重构时由于数据表示方法的不同没有使用测试集代码中基于 C++的 vector 类，其他测试用例的重构代码 IO 部分则与测试集代码保持了一致. 因此在图 10 和图 11 中，除 SGEMM 外，生成代码的 IO 性能在两种平台下和测试集代码持平.

BFS 的生成代码 Copy 项性能在两种平台上均出现了较为明显的开销. 检查代码后发现，测试集 OpenCL 代码中的大部分数据拷贝采用了 *no_blocking* 模式，主机端不必等待拷贝结果可以继续执行，隐藏了数据拷贝的延迟.

UPPA 的运行时系统封装了 OpenCL API，因此 Ocl 项对应 UPPA 运行时系统的执行性能，反映了封装后运行时接口的性能开销. 为了实现自动的 OpenCL 执行环境配置和数据管理并保证执行的正确性，封装后的接口中有一定的冗余 API 调用，如为了保证同步添加的 *cl_finish* 接口调用等. 实验结果表明运行时系统的性能在 GPU 平台上和测试集代码基本相同，开销仅 1%～2%，但在 MIC 平台上却有了 10%～20%的性能损失. GPU 平台和 MIC 平台上 OpenCL 运行时支持由硬件厂家各自实现，两种平台上 OpenCL 实现的不同是导致这种差异的重要原因，这在测试用例 Histo 的实验结果中也得到了证实. 测试用例 Histo 代码由于算法设计原因包含了大量 OpenCL API 调用，其执行性能在两种平台上也出现了较大差异，图 11 和表 11 说明了 OpenCL API 调用的开销是图 9 中 Histo 测试用例的测试集 OpenCL 代码在 MIC 平台上性能反常的原因，体现出不同平台提供的 OpenCL 实现的不同.

表 11　Histo OCL_MIC 执行时间详情（单位：s）

数据集	IO	Copy	Kernel	Ocl	Total
small	0.0253	0.0749	1.3620	10.3606	11.8229
medium	0.0330	0.1930	3.8658	10.3951	14.4869
large	0.0502	0.2338	9.4876	10.4571	20.2287

Kernel 项执行性能在不同平台和不同测试用例间有着较大的差异. 内核执行线程空间设置是影响内核执行性能的重要因素，UPPA 的运行时系统根据上层应用的并行度，以硬件支持范围内的最大并行规模组织线程. 理论上较大规模的数据可以提升应用的并行度，进而增大在运行时生成的线程数量，可以更有效发挥众核处理器的大规模并行计算能力，如测试用例 Stencil，随着数据规模的增大，生成代码的内核执行性能逐步提升. 但实际的内核执行性能则受到具体的应用特征及硬件特征的影响. 测试用例 SpMV 中需要共享向量数据，在 GPU 平台上可以通过增大线程数量获得加速效果，但在 MIC 平台上则成为访存瓶颈，线程数量的增大反而导致执行效率的下降. 测试用例 NN 的并行模式较简单，线程间访存冲突概率较小，因此以较大规模线程执行的生成代码在两种平台上的性能均超过了测试集 OpenCL 代码. 但随着数据规模的增大，线程数量逐渐饱和，GPU 平台上的内核执行效率随之下降，MIC 平台上的加速效果也趋于平缓. 测试用例 SGEMM、BFS、Kmeans 的内核执行性能不随数据规模单调变化，这说明对于这些应用，通过提高线程数量来提高性能只在某些数据规模区间内有效，同时不同平台上的变化规律又显示出不同硬件架构和并行规模对性能的影响. 线程空间的设置需要适配具体应用和硬件的特征，尚没有方法能够针对不同应用和硬件实现自动的最优化设置，需要进一步研究.

应用中采用的各类优化措施也影响着内核执行性能. 测试集代码的优化措施主要针对 GPU 平台，以利用 GPU 中流处理器 SM 到流处理器 SP 的多级计算架构和全局存储到本地存储的分层缓存架构. 这也是图 10 中 Histo 和 BFS 的生成代码内核执行性能不及测试集 OpenCL 代码的原因. 但这些基于 GPU 架构特点的优化措施并不具有通用性. 图 11 显示 MIC 平台上多个测试用例 Kernel 项性能有着与 GPU 平台相反的规律. MIC 处理器其实质是共享内存的众核处理器，单核计算能力较强，但每个计算核心私有的 L2 缓存容量有限，针对 GPU 平台的线程块优化、本地存储优化等措施并不适合 MIC 架构. 图 11 显示 MIC 平台上生成代码的内核执行性能整体较好，说明 UPPA 架构具有较强的通用性.

Lift 方法也实现了 Rodinia 测试集中的 NN 和 Kmeans 测试用例. 基于 Lift 方法生成和优化后的内核代码在 AMD 和 NVIDIA 两种 GPU 平台上进行了对比测试，NN 测试用例中的加速比为 1.0～1.05，Kmeans 测试用例中的加速比为 0.9～1.0. UPPA 方法在 GPU 平台下的内核执行加速比与之相当，在 MIC 平台下则略有超过，从另一方面说明了本文方法的有效性和编译实现的高效.

7　结论与未来工作

为了简化异构并行编程，提高用户编程效率，实现高层统一并行编程，本文提出了一种面向异构众

核系统的高层统一并行编程架构 UPPA，通过数据关联计算模型简化了并行编程逻辑，实现了运行时无关的统一并行表达；通过数据关联计算描述语言提供了简单易用的统一并行编程接口，为用户隐藏了底层的运行时细节，实现高层统一编程，同时为编译和运行时的映射执行提供了指导；通过源到源编译器和运行时系统实现了上层应用向底层异构平台的自动映射，保证了上层应用的跨平台执行. 通过对多个测试用例的重构和在 GPU 和 MIC 两种平台上的实验测试表明，本文的方法可以有效地表达应用中的并行性，提高编程效率，同时提供了良好的跨平台可移植性.

测试结果验证了 UPPA 架构的有效性，同时也为编译和运行时系统的优化提供了依据，为未来大规模工程应用打下基础. 在数据拷贝中，可以通过 no_blocking 标志等措施重叠通信和计算. 在 MIC 平台上，OpenCL API 调用的次数会明显影响应用性能，因此需要对运行时系统中 OpenCL API 的调用进行优化. 内核优化方法和底层硬件架构的关系也需要进一步研究. 在优化工作的基础上，大规模工程应用和测试将是我们未来工作的重点.

参 考 文 献

[1] Liu Ying, Lü Fang, Wang Lei, et al. Research on heterogeneous parallel programming model. Journal of Software, 2014, 25(7): 1459-1475(in Chinese)
(刘颖，吕方，王蕾等. 异构并行编程模型研究与进展. 软件学报，2014，25(7)：1459-1475)

[2] Cheng Hua, Wang Li-Sheng, Xia Zhi-Yu, Jia Jia-Tao. High-productive parallel programming: Challenges and evaluation//Proceedings of the National Annual Conference on High Performance Computing 2013. Guilin, China, 2013: 839-846 (in Chinese)
(程华，王礼生，夏志禹，贾家涛. 高效能并行编程的挑战及评价研究//2013 全国高性能计算学术年会论文集. 桂林，中国，2013：839-846)

[3] Amarasinghe S, Hall M, Lethin R, et al. Exascale programming challenges//Proceedings of the Workshop on Exascale Programming Challenges. Marina del Rey, USA, 2011: 1-65

[4] Diaz J, Muñoz-Caro C, Niño A. A survey of parallel programming models and tools in the multi and many-core era. IEEE Transactions on Parallel & Distributed Systems, 2012, 23(8): 1369-1386

[5] Martineau M, Mcintosh-Smith S, Gaudin W, et al. An evaluation of emerging many-core parallel programming models//Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'16). Barcelona, Spain, 2016: 1-10

[6] Schnetter E, Raiskila K, Takala J, et al. Pocl: A performance-portable OpenCL implementation. International Journal of Parallel Programming, 2015, 43(5): 752-785

[7] Auerbach J, Bacon D F, Cheng P, et al. Lime: A Java-compatible and synthesizable language for heterogeneous architectures//Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10). Reno/Tahoe, USA, 2010: 89-108

[8] Steuwer M, Remmelg T, Dubach C. Lift: A functional data-parallel IR for high-performance GPU code generation//Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO'17). Austin, USA, 2017: 74-85

[9] Chen Y, Cui X, Mei H. PARRAY: A unifying array representation for heterogeneous parallelism//Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12). New Orleans, USA, 2012: 171-180

[10] Catanzaro B, Garland M, Keutzer K. Copperhead: Compiling an embedded data parallel language//Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11). San Antonio, USA, 2011: 47-56

[11] Taylor B, Marco V S, Wang Z. Adaptive optimization for OpenCL programs on embedded heterogeneous systems//Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2017). Barcelona, Spain, 2017: 11-20

[12] Chamberlain B L, Callahan D, Zima H P. Parallel programmability and the chapel language. International Journal of High Performance Computing Applications, 2007, 21(3): 291-312

[13] Dean Jeffrey, Ghemawat Sanjay. MapReduce: Simplified data processing on large clusters. Communications of the ACM, 2008, 51(1): 107-113

[14] Linderman M D, Collins J D, Wang H, et al. Merge: A programming model for heterogeneous multi-core systems//Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII). Seattle, USA, 2008: 287-296

[15] Ernstsson A, Li L, Kessler C. SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. International Journal of Parallel Programming, 2018, 46(1): 62-80

[16] Carter E H, Trott C R, Sunderland D. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. Journal of Parallel and Distributed Computing, 2014, 74(12): 3202-3216

[17] Lee E A, Messerschmitt D G. Synchronous data flow. Proceedings of the IEEE, 1987, 75(1): 1235-1245

[18] Johnston W M, Hanna J R P, Millar R J. Advances in dataflow programming languages. ACM Computing Surveys, 2004, 36(1): 1-34

[19] Nowatzki T, Gangadhar V, Sankaralingam K. Exploring the potential of heterogeneous von neumann/dataflow execution models//Proceedings of the 42nd Annual International Symposium on Computer Architecture. Portland, USA, 2015: 298-310

[20] Thies W, Karczmarek M, Amarasinghe S. StreamIt: A language for streaming applications//Proceedings of the 11th International Conference on Compiler Construction. Grenoble, France, 2002: 179-196

[21] Hormati A H, Samadi M, Woh M, et al. Sponge: Portable stream programming on graphics engines//Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI). Newport Beach, USA, 2011: 381-392

[22] Hong J, Hong K, Burgstaller B, et al. StreamPI: A stream-parallel programming extension for object-oriented programming languages. Journal of Supercomputing, 2012, 61(1): 118-140

[23] Che S, Boyer M, Meng J, et al. Rodinia: A benchmark suite for heterogeneous computing//Proceedings of the IEEE International Symposium on Workload Characterization. Austin, USA, 2009: 44-54

WU Shu-Sen, Ph. D. candidate. His main research interests include high performance computing, heterogeneous parallel programming models and methods.

DONG Xiao-She, Ph. D., professor, Ph. D. supervisor. His main research interests include high performance computing, high effectiveness storage system and cloud computing.

WANG Yu-Fei, Ph. D. candidate. His main research interests include high performance computing and high effectiveness storage system.

WANG Long-Xiang, Ph. D., engineer. His main research interests include high performance computing and high effectiveness storage system.

ZHU Zheng-Dong, Ph. D., professorate senior engineer. His main research interests include high performance computing, cloud computing and big data.

Background

Mainstream heterogeneous parallel programming methods provide low-level programing abstraction and close-to-metal programming interface, making heterogeneous programing difficult and error-prone. The developers cannot exploit different heterogeneous processors with a unified programming method and the applications lack of portability and scalability. Unified High-level parallel programming method becomes the key to exert the massively parallel computing capability of heterogeneous many-core processors.

This work presents UPPA, a unified parallel programming architecture for heterogeneous systems. It includes the data associated computation (DAC) model, a corresponding descriptive language and a prototype compiler and runtime system based on OpenCL. The DAC model and corresponding descriptive language provide unified parallel abstraction and high-level programming interface for developers. The compiler and runtime system realize cross-platform execution of high-level applications. The UPPA improves heterogeneous programming productivity and achieves the goal of exploiting different heterogeneous systems with unified high-level applications while maintain the performance.

This work is partially supported by the National Natural Science Foundation of China (No. 61572394) and the National Key Research and Development Program of China (No. 2017YFB0202002). These projects aim to provide unified parallel programming framework for heterogeneous systems and exascale supercomputers.