基于 SDFG 的控制流模型到数据流模型的程序转换方法

付佳伟，陈俊仕，安虹

（中国科学技术大学计算机科学与技术学院，安徽 合肥 230026）

摘　要： 数据流编程是一种新型的并行编程范式，其通过细粒度的任务拆分实现高性能的并行计算，但也带来了额外的编程复杂性。本文提出一种新型的程序转换方法与编译流程，借助两层程序中间表示实现了将控制流结构的计算程序全自动无缝转换为数据流结构。由于两种编程模型在程序的表达方式上具有诸多差异，自动化转换过程中面临许多挑战，本文将其细化为 4 个具体的类别，并逐一提出应对方案。在 PolyBench/C 数据集上的实验结果表明，通过本文设计的转换方案产生的数据流程序，其整体并行计算性能接近基于有状态数据流多重图（SDFG）的 OpenMP 并行化方案，并在近一半的样例中超越 OpenMP 方案。本文还选择了多个具体的程序进行深入分析，讨论了不同编译方案产生的结果存在性能差异的原因，展现了 Codelet 模型在大量线程下面对不规则并行模式的优越性。

关键词： 数据流编程；Codelet 模型；有状态数据流多重图；程序转换；源到源翻译器

中图分类号： TP312　　**文献标志码：** A　　**DOI：** 10.19678/j.issn.1000-3428.0252315

Program Transpilation Method from Control Flow Model to Dataflow Model Based on SDFG

FU Jiawei, CHEN Junshi, AN Hong

(School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, Anhui, China)

【Abstract】 Dataflow programming is an emerging parallel-programming paradigm. By splitting tasks into fine granularities, high-performance parallel computing can be achieved at the expense of increased programming complexity. This study proposes a new program transpilation method and compilation procedure that leverages two levels of intermediate representation and converts control flow computational programs into a dataflow architecture in a fully automatic manner. Owing to the substantial disparities between these programming models, many challenges exist in automatic conversion schemes, which are split into four categories and addressed separately. Experimental results on the PolyBench/C dataset indicate that the proposed conversion scheme can produce dataflow parallel programs on par with those produced by the Stateful DataFlow multiGraph (SDFG)-based OpenMP parallelizing compiler while surpassing it in approximately half of the cases. This study also conducts case studies on multiple program samples and discusses the causes of performance distinctions in different compilation schemes, revealing the superiority of the Codelet model with a massive number of threads on irregular parallelism.

【Key words】 dataflow programming; Codelet model; Stateful DataFlow multiGraph (SDFG); program transpilation; source-to-source compiler

0　引言

过去的几十年间，串行计算机体系结构的性能发展已经显著放缓，并行体系结构成为提升计算机整体性能的主要途径[1]，而并行计算技术的发展也驱动了并行程序设计的发展。最初，开发者直接使用操作系统提供的线程接口来构造并行程序，这种低级编程方式不仅代码复杂，而且容易出现竞争、死锁等并发问题[2]。为了解决这些问题，并行编程模型应运而生。例如，消息传递接口（MPI）为分布式内存系统提供了标准化的通信机制，使得多个节点间的协同工作变得更加可靠[3]；而 OpenMP 则利用编译制导语句简化了共享内存系统下的多线程编程，使得开发者可以更专注于并行逻辑本身，而不必纠结于底层细节[4]。随着并行计算需求的不断多样化，近年来 CUDA[5]、OpenCL[6] 等框架更是成功将图形处理器（GPU）这一专用并行处理平台引入主流计算领域，大幅提升了在图形、物理模拟及深度学

基金项目： 中国科学院战略性先导科技专项（XDB0500102）。

作者简介： 付佳伟（CCF 学生会员），男，硕士研究生，主研方向为并行编程模型、编译器设计；陈俊仕（CCF 会员），研究员、博士；安虹（CCF 会员），教授、博士。

收稿日期： 2025-04-11　　**修回日期：** 2025-06-05　**E-mail：** ibug@mail.ustc.edu.cn

习等方面的计算效率。这些编程模型不仅对底层硬件资源进行了高效抽象，而且通过提供简洁的接口大大减轻了开发者的负担，从而使得大规模并行应用的开发变得更加高效和可靠。

数据流编程模型作为一种新兴技术，与传统的控制流编程模型相比，能够更加细粒度地表达出程序的并行潜力[7]。传统并行模型受限于集中式内存访问和指令调度的连续性，难以实现大规模并行计算[8]。数据流模型是一种以数据为核心驱动计算过程的编程范式，其基本思想是将程序看成一个有向无环图，图中的每个节点代表一段操作，边则表示数据流动的方向。任务之间不再具有层次关系，只要所需的输入数据都准备就绪，一个任务就可以开始执行，消除了传统并行编程模型中由于代码的控制结构导致的同步和等待。这种数据到达即触发操作的方式，可以充分发挥并行处理和异步执行的优势，同时解决了传统编程模型中并行访问数据带来的竞争问题。从本质上讲，数据流模型将计算过程从严格的控制流转化为以数据依赖为驱动的事件流，从而大幅提高系统的吞吐能力和响应速度[9]。

尽管数据流并行模型具有优秀的并行潜力，然而其目前仍未在高性能科学计算领域获得广泛的推广和应用，其中一个显著原因便是编程的复杂性[10]。Legion[11]编程模型设计了新的关键字和语法，使得开发者可以将任务和数据划分为逻辑区域，并标记每个区域所管理的数据及其所需的与其他区域的通信和数据交换，通过一个软件定义的乱序执行处理器对各个区域进行调度和并行执行。Intel TBB[12]是一套针对并行计算环境设计的 C++模板库，其提供了一套数据流应用程序编程接口(API)，允许用户创建各种计算节点，定义节点之间的数据传输关系，构造数据流图，通过显式地发送输入数据来触发图中各节点的计算任务，然后等待数据流图完成运算。CUDA Graph[13]采用与 Intel TBB 类似的做法，提供构建 CUDA 计算图的 API，但图中的计算节点则由 GPU kernel 实现，并由 CUDA 运行时系统负责执行程序提交的 CUDA 计算图及调度和运行图中所包含的 GPU kernel。由以上例子可见，数据流模型消除了传统编程模型中的栈和函数调用层次结构，不再通过直接访问全局数组和函数传参等传统方式在计算代码之间共享数据，而是要求开发者将计算代码包装进数据流图的节点中，并定义不同节点之间的数据传输方向和依赖关系。这样的编程方式虽然能够更好地揭示任务之间的依赖关系和发掘程序的并行性，但同时也使得程序设计和调试过程变得繁琐复杂。为了充分利用数据流模型所具有的并行潜力，开发者需要对并行程序设计乃至并行计算机体系结构具备深入理解，这对于大多数科学计算应用开发者而言，门槛过高且实现难度较大[14]。因此，尽管数据流编程方式能够提升程序的并行能力，但也使得现有的应用程序难以移植到数据流模型上，而重新以数据流方式改写大型程序又是一项艰巨的任务。

针对数据流编程模型在实际应用中所面临的编程复杂性问题，本文提出了一种自动化的程序编译与转换方案。首先，本文设计了一套编译与转换流程，借助多级中间表示(MLIR)[15]和数据中心中间表示(DCIR)[16]相关的工具链，将原始程序翻译至采用数据流思想的有状态数据流多重图(SDFG)中间表示层[14]。其次，本文设计了一个转换方案，将 SDFG 中间表示转换至 Codelet 数据流模型[17-18]，并针对控制流结构映射、可并行区域的代码结构映射、上下文与局部变量的维护、同步语义转换等多个关键问题提出了解决方案。最后，本文以 PolyBench/C[19]为数据集，验证了所提方案的有效性。本文的主要贡献总结如下：

1)提出了一种完全自动化的编译与转换流程，将采用传统并行编程模型编写的科学计算应用程序自动转换为基于数据流编程模型的实现形式，进而实现从源代码到高效数据流执行的无缝迁移。

2)针对 SDFG 模型和 Codelet 数据流模型之间的差异，提出了解决方案和优化措施，使得采用 SDFG 中间表示的程序可以正确地转换至 Codelet 模型。

3)通过实验验证了转换方案的有效性，并分析了转换后程序的性能表现。

1 研究背景

不同的数据流编程模型在计算结构的抽象、任务划分与局部性处理等方面各有区别。本章分别介绍与本文工作直接相关的 Codelet 模型与 SDFG 模型。

1.1 Codelet 数据流模型

Codelet[17]模型是一种细粒度数据流运行模型，其核心理念在于将复杂且大规模的计算任务分解为称为 codelet 的微型代码片段，并通过构建由 codelets 构成的数据流图(Codelet Graph)来挖掘算法的并行潜力。codelet 之间通过显式信号传递来管理其间的依赖关系，这一机制不仅减少了对全局调度的需求，同时简化了运行时系统的设计，使其仅需维护 codelet 的任务依赖及资源依赖，便可有效地进行调度与优化，如图 1 所示。

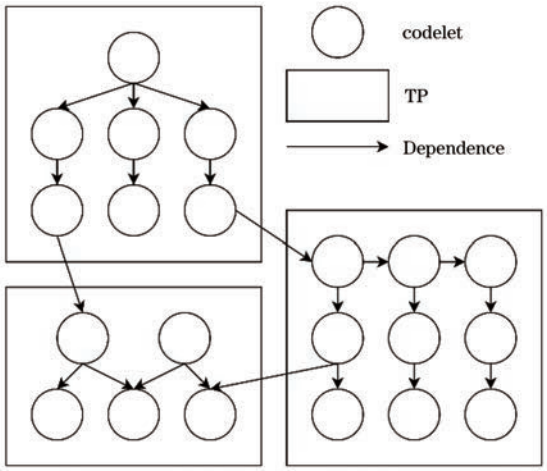

图 1 Codelet 执行模型

Fig. 1 The Codelet execution model

Codelet 模型首先提出了一种结合轻核计算单元(CU)与重核调度单元(SU),具备多层次内存与通信结构的抽象计算机模型。该模型为 codelet 计算提供了多层次、高度并行化的硬件架构基础,同时具备非统一内存访问(NUMA)及异构计算等特性,使其能够高效地映射到各种实际的大规模计算系统之上,并在复杂处理环境中维持优良的运行性能,极大地简化了 Codelet 执行模型的设计与实现。

codelet 作为 Codelet 执行模型中最基本的代码单元,具备一系列依赖关系,这些依赖可以源自其他 codelet,也可能涉及特定的计算资源。当某个 codelet 的所有依赖均被满足后,其进入就绪状态,并可由调度器即时调度至计算单元上执行。codelet 的执行过程具有不可抢占性与原子性,即其一旦开始执行,便不会被中断,且整个执行过程作为一个不可分割的操作完成。在执行过程中,codelet 还可向其他 codelet 发送信号,以动态控制数据流图的运行。codelet 执行的不可抢占性与原子化特性有效避免了隐藏或不可预测的依赖关系引发计算核心等待资源的情况,从而确保了数据流图能够以有序且高效的方式运行。

由于大量 codelet 承担了数据流图中的主要计算任务,Codelet 模型引入了一种更高层次的抽象单元,即线程过程(TP)。TP 本质上是一种异步函数,其封装了一组 codelet 及其运行所需的上下文信息,如输入数据、输出数据及存储环境等,起到组织与管理的作用。在执行过程中,每个 codelet 仅能访问其所属 TP 所提供的上下文,而无法访问该 TP 之外的数据或资源,这种限制有效防止了程序运行中的不可控延迟与依赖关系。TP 通过将一组高度相关的 codelet 及其上下文共同调度到计算单元及其本地高速存储中,维持并增强程序的时间局部性与空间局部性。此外,TP 采用延迟实例化机制,即在创建时不会立即分配计算资源,而是首先进入 TP 池,待适当时机再实例化到具体的计算单元上。此机制使得 Codelet 模型能够有效适配复杂的异构计算环境,充分利用底层计算与通信资源,从而提升程序的并行性与可扩展性。

DARTS[18] 是 Codelet 模型的一种编程框架及运行时系统,其核心特点在于尽可能遵循 Codelet 模型的设计原则,是当前最符合 Codelet 模型的数据流框架。该系统采用 C++语言实现,利用现代 C++语言特性,如面向对象编程与模板化编程等,以在保持系统与硬件高效执行能力的同时,尽量减少额外的依赖与运行开销。此外,DARTS 提供了一种模块化且可移植的编程接口,使其能够适应多种计算环境。因此,本文工作选择 DARTS 作为数据流程序的运行时系统,以充分利用其高效的数据流调度能力和良好的可扩展性。

1.2 SDFG 模型

DaCe[14] 是一种创新性并行编程框架,旨在将使用 Python/NumPy 及其他高级语言编写的代码高效映射至中央处理器(CPU)、GPU 及现场可编程门阵列(FPGA)等异构计算平台。该框架以 SDFG 作为中间表示层,如图 2 所示,从而实现代码开发与优化过程的解耦,使得领域科学家能够继续使用其熟悉的编程语言和工具进行算法实现,而无须关注底层优化策略及跨平台迁移的复杂性;与此同时,性能工程师则可在不修改源代码的前提下,通过对 SDFG 进行变换与调整,针对不同计算后端进行优化、编译与性能分析,以最大程度地提升代码执行效率。

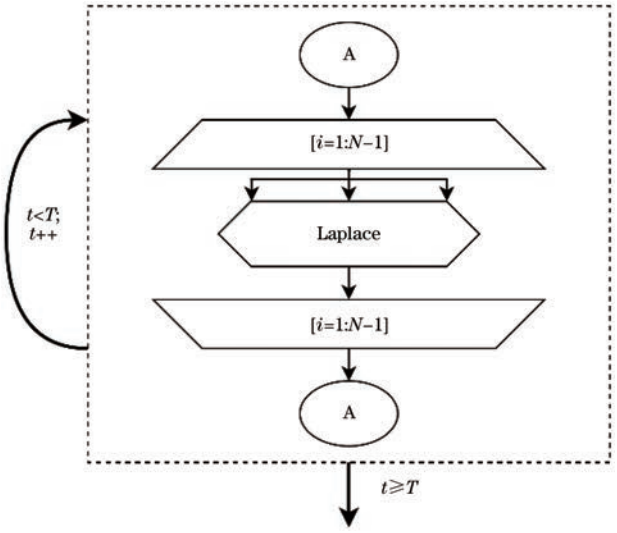

图 2 SDFG 模型中的 Laplace 迭代算法

Fig. 2 Iterative Laplace algorithm in SDFG model

作为面向通用计算程序的并行编程与优化框架，DaCe 及其设计的 SDFG 模型在挖掘程序并行性及实施优化的方式上与 Codelet 数据流模型有所不同：DaCe 将通用计算过程抽象为数据在数据流图节点内部的变换以及节点之间的传输；SDFG 模型则将程序视作一张计算图，程序所使用的数据位于图中的数据容器节点中，并且具有生命周期，随着作用域的进入和退出进行分配、复制和释放。计算节点（Tasklet）包含了无状态的计算代码，而 SDFG 图中的边连接着计算节点与数据容器，负责向计算节点提供输入/输出数据。程序的并行性在 SDFG 中通过 Map 作用域展现，即将一个数组中的每个元素分别传入一个计算节点进行运算。对于不规则的循环模式，也可以在 Map 作用域内使用嵌套的 SDFG（NestedSDFG）元素，使得循环的每次迭代可以具有不同的执行方式。SDFG 模型还基于常见的程序设计模式引入“状态”机制，以更精准地表达诸如不固定迭代次数的循环和分阶段算法等典型程序结构，从而提升对复杂计算模式的刻画能力。

相较于以 codelet 为核心的 Codelet 模型，SDFG 模型更侧重于数据在计算图中的传输与变换，并基于数据流思想构建了一套以图变换为核心的方法论来实现程序优化。在该模型下，计算程序的优化过程被表述为一系列对数据流图的变换操作，例如代码合并、循环展开，或将 SDFG 中的特定子图映射到特定计算后端，如 GPU、CUDA 和 FPGA 等，从而为优化决策提供系统化支持。这种设计使得领域科学家能够在不破坏或影响程序已有优化成果的前提下，自由调整计算逻辑，从而实现科学计算程序的开发与高性能优化的解耦，使程序能够在不同计算平台上高效执行，同时保持其可读性和可维护性。

综上，SDFG 通过将计算程序表示为可变换的数据流图，不仅能够精准建模通用计算中的数据流模式，还能通过系统化的图变换机制实现高效的跨平台优化，从而有效解耦程序开发与性能调优。基于这些优势，本文将采用 SDFG 作为中间表示层，以实现并行计算代码的自动数据流转化，充分挖掘程序的并行性并优化其在异构计算平台上的执行效率。

2 数据流模型翻译器的设计与实现

为了综合利用 SDFG 模型在数据流程序优化方面的优势以及 Codelet 模型在数据流程序执行方面的优越性，本文设计并实现了一种方法，将使用 C 和 C++语言编写的原始计算程序转换为 SDFG 数据流图，并借助 DaCe 框架对其进行数据流优化，随后将优化后的表示转换为基于 Codelet 数据流模型的代码。该方法不仅充分发挥了 SDFG 在优化过程中对计算结构的灵活重构能力，同时也利用了 Codelet 模型在高效执行数据流任务方面的特性，实现了从程序表示到运行时调度的高效协同。

2.1 翻译器整体框架

本文构建了从科学计算程序的原始代码到高性能程序的完整优化流程，该流程实现了代码向 Codelet 模型的高效转换，其整体架构如图 3 所示。

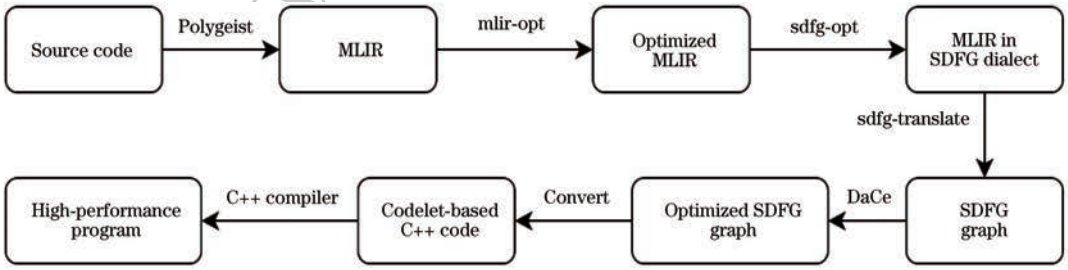

图 3 完整的编译与转换流程

Fig. 3 Complete compilation and transpilation procedure

首先，本文借助 Polygeist[20] 编译器将科学计算程序的 C/C++源代码编译为 MLIR 形式，以利用 MLIR 现有的基础设施对程序进行初步优化（对应图 3 中的 mlir-opt[15] 步骤）。随后，采用 DCIR[16] 提供的相关工具，将优化后的通用 MLIR 代码转换为 SDFG 方言（dialect）中的 MLIR 代码，并进一步转换为序列化的 SDFG 文件（对应图 3 中的 sdfg-opt 与 sdfg-translate 两个步骤）。在此基础上，本文应用第 2.2 节所提出的核心方法，从数据流视角对 SDFG 进行优化，并将其转换为 Codelet 数据流图，最终生成采用 DARTS[18] 框架实现的 C++项目工程。最后，利用标准编译器（如 GCC 或 Clang 等）对上述 C++工程进行编译，即可获得针对目标硬件优化的高性能科学计算程序。

本文所设计的编译与转换流程实现了全自动化处理，无需对原始代码进行任何修改或额外标注，即

可完成从源代码到高性能可执行程序的转换与优化。用户仅需执行 make 命令，即可触发图 3 所示的完整编译与转换流程，并生成最终程序。在此过程中，本文的构建系统会保留所有中间文件，如优化前后的 MLIR 代码和 SDFG 文件等。这些中间文件使用户能够根据自己的需求对中间结果进行观察分析，或者进行更加细粒度、针对性的调优，如手工应用额外的 MLIR 优化或 SDFG 图变换等。本文设计的构建系统能够自动检测手工修改过的中间文件，并从修改位置开始执行后续步骤，而不会无条件地从头执行完整的编译与转换流程，从而能够保留这些手动优化的结果，并缩短编译与转换流程的执行时间，提高编程开发效率。

2.2 SDFG 模型到 Codelet 模型的转换

本文工作的核心在于搭建了 SDFG 数据流模型与 Codelet 数据流模型之间的桥梁。尽管两者均为以数据为中心、将程序视作数据在不同的节点之间的流动与变换的编程模型，但是由于两个数据流模型采用的设计思想、预定的功能目标及面向的应用场景不同，两个模型之间的许多概念并非简单的一一对应关系，并不存在一种明显而直接的方式将一个模型中的数据流图映射到另一个模型的数据流图中。为了能够顺利地将 SDFG 数据流图以一种合理的方式转换为 Codelet 数据流图，本文深入分析了 SDFG 数据流图中的各种元素在程序的执行和发掘程序并行性中的作用，并对两种数据流模型中的元素进行比较，以确保最终的转换结果具有合适的图结构。

本文对 SDFG 与 Codelet 数据流模型中各元素的层次结构进行了系统比较，如表 1 所示。可以看出，SDFG 模型由数据流图、状态、节点三个层次组成，而 Codelet 模型相对更为扁平，仅由 TP 和 codelet 两个层次构成，这反映了两者在程序表示结构上的不同抽象方式。在 SDFG 中，数据流图作为整体的计算表示，包含多个状态(states)，这些状态进一步包含具体的计算和数据操作节点，从而形成对程序上层控制流的建模。然而，在 Codelet 模型中并不存在与状态直接对应的概念，因为 codelet 主要基于数据驱动执行，而非显式的状态转换控制。而在图节点层次上，SDFG 进一步区分了数据容器节点和计算节点，并通过数据容器节点维护程序的访存局部性；Codelet 模型则只有计算节点，而访存局部性则交由计算节点的上层封装单元 TP 负责维护。因此，在将 SDFG 数据流图转换为 Codelet 数据流图的过程中，需要解决控制流结构映射、可并行区域的代码结构映射、上下文与局部变量的维护、同步语义转换等多个关键问题。

表 1 SDFG 与 Codelet 数据流图的元素层次结构比较

Table 1 Comparison of the hierarchical structure of elements between SDFG and Codelet dataflow graphs

SDFG model	Codelet model	Description
SDFG graph	TP	Represents complete programs, but may also appear nested inside another, like NestedSDFG nodes in SDFG and TPs created through invoke
SDFG states	—	Represents the control flow in programs that may only be executed sequentially; a corresponding concept does not exist in the Codelet model
Data container nodes	(Data stored in TP)	Represents data storage structure and manages data access for compute code, maintaining data locality
Non-data container nodes	codelets	Represents compute code responsible for the actual computation

2.2.1 控制流结构映射

SDFG 和 Codelet 模型在层次结构上的差异可以分别处理。顶层结构是两个模型中唯一一个可以直接映射的结构，因此本文的转换方案将每个 SDFG 数据流图映射为一个对应的 TP。为确保程序能够正确进入 Codelet 运行环境，本文将 SDFG 程序的入口函数替换为初始化 DARTS 运行时系统的代码，使其能够适配 DARTS 框架。类似地，对于“调用嵌套的 SDFG(invoke NestedSDFG)”类型的节点，其转换目标为“invoke 一个新 TP 并等待其执行完毕”，从而保持程序执行语义的一致性。以上即为所有能够以直观的方式进行直接映射的数据流元素。

在 SDFG 模型中，“状态”概念用于表示程序中最上层的、不可并行的控制流结构。为了在 Codelet 模型中合理映射这一概念，本文将不同的 SDFG 状态转换为 Codelet 计算图中的一系列仅可顺序执行的不重叠子图，并通过这些子图及其间的依赖关系表示程序的控制流结构。具体而言，本文将每个状态中的计算节点转换为一个或多个 codelet，这些 codelet 构成整个 Codelet 数据流图中的子图，且不同状态所对应的 codelet 子图之间没有重叠。

SDFG 状态之间的迁移可转化为在 Codelet 模型中向目标状态对应的 codelet 发送信号，并立即终止当前 codelet 的执行，以此模拟 SDFG 状态转换的语义，确保执行逻辑的正确性和完整性。

2.2.2 可并行区域的代码结构映射

由于 codelet 本质上是小规模的可执行代码段，由单个处理单元顺序执行，其自身不具备并行执行的能力，因此在将 SDFG 模型映射至 Codelet 模型的过程中，所有能够或需要并行执行的程序结构必须依据并行度分别分解并映射到多个 codelet 或相互无依赖的 codelet 子图上，以确保并行性的有效利用。具体而言，在 SDFG 模型的某一状态内，若存在多个可并行执行的子图，则这些子图应当映射至同一 TP 内部的不同 codelet 上（或 codelet 子图中），以提升程序的并行性并缩短整体运行时间。同理，SDFG 模型中的 Map 和 Consume 两类节点所定义的作用域（scope）内的子图亦需分别映射至不同的 codelet，并且这类 codelet 的数量由对应的 Map 或 Consume 节点的并行度参数决定，从而确保 Codelet 模型能够充分利用其所定义的数据驱动并行执行机制，实现高效并行计算。

2.2.3 上下文与局部变量

尽管 SDFG 数据流图和 Codelet 数据流图最终均以 C++语言实现，但两者在数据管理方式和作用域控制上存在显著差异。其中，DaCe 框架将优化后的 SDFG 数据流图转换为 C++函数，并通过全局的“状态结构体”统一存储和维护需要在不同状态间共享和传递的数据（如全局数组等），从而使 SDFG 中的计算代码能够根据 C++作用域规则直接访问所有在当前层级可见的标识符，包括局部变量、函数参数以及已声明的全局变量等。相较之下，DARTS 框架作为 Codelet 模型的具体实现，则不使用全局变量，也不鼓励维护全局状态（尽管 C++语言本身支持这种做法），而是仅允许 codelet 计算代码通过指向其所属 TP 的指针显式访问 TP 中存储的数据。

为了使转换后的计算代码能够正确访问到其所需的上下文环境，如 SDFG 中的状态变量及栈帧中的局部变量等，同时尽可能贴近 Codelet 模型定义的访存模式，本文采用基于 TP 结构的数据封装方式，将 SDFG 中需要共享和传递的数据统一存储在 TP 内。具体而言，本文将 SDFG 全局状态结构体中的所有变量定义移动至 TP 中，并通过 C++的 auto& 语法为 TP 所包含的全部 codelet 添加引用声明，使计算代码能够在不更改变量名或访问方式的前提下，直接引用其对应的上下文环境。此外，为了进一步优化数据管理，局部数组的分配和释放也可以移动至 TP 的构造函数与析构函数中，使其随 TP 的生命周期自动创建、分配和释放。通过此方法，各个 codelet 可以通过其所属的 TP 引用这些数据容器并访问其中的数据，从而在维护访存局部性的同时，使数据访问模式与 Codelet 模型的内存结构保持一致，减少数据传输开销。

2.2.4 同步语义转换

SDFG 与 Codelet 模型对计算任务的组织方式和同步机制有着显著差异。SDFG 模型的设计模式和程序结构保留了许多传统控制流驱动的并行编程思想与元素，例如并行区域、并行屏障（也称同步点）以及归约操作等。这些机制通常是基于显式同步和控制流管理的，并在程序执行过程中动态协调不同计算任务的运行。Codelet 模型则采用依赖计数和原子操作来管理待执行的 codelet，其调度策略确保进入就绪状态的 codelet 一旦被发射执行，便以不可抢占的方式运行于其分配到的计算单元上，从而保证整个数据流图能够高效、无阻塞地执行。因此，在将 SDFG 模型转换为 Codelet 模型时，嵌入于计算代码片段中的同步语义不能直接映射至单个 codelet 之中，否则将违反 codelet 的原子性的定义，进而破坏其对程序执行性能的优化保证。因此在转换过程中，本文重新设计了同步机制，以符合 Codelet 模型的运行约束，同时保持程序的正确性与运行效率。

在 Codelet 模型的运行过程中，实现“使待执行的计算代码等待特定资源或信号”的唯一机制是 codelet 本身的依赖计数器。具体而言，codelet 之间通过显式传递信号的方式建立依赖关系，使得数据流图中处于前驱位置的 codelet 能够控制其后继 codelet 的就绪时机，确保后继 codelet 仅在接收到所有必要的信号后才进入执行状态。因此，为了在 Codelet 模型中正确重现 SDFG 模型中的同步语义，需要以 SDFG 程序中的同步点为分割边界，将同步点前后的计算逻辑拆分至不同的 codelet 内，并基于同步关系正确设置这些 codelet 的依赖计数及其在数据流图中的依赖连接。这样，原本依赖同步机制的代码块将在 Codelet 模型中转换为尚未就绪（依赖计数未归零）的 codelet，从而确保同步点之后的计算仅在满足所有同步条件后才开始执行，最终实现与 SDFG 模型等价的同步行为，同时保持 Codelet 模型无阻塞、高并行度的执行特性。

2.3 并行粒度的优化

SDFG 模型在设计之初即旨在从数据流的视角对科学计算程序和算法进行优化，而非直接以数据流方式运行或调度程序中的代码片段或模块。实际上，SDFG 数据流图在经过变换与优化后，会生成 C++代码作为后端输出，并进一步通过 C++编译器编译为最终的二进制可执行程序，从而利用 C++编译器提供的底层分析与优化能力，例如基于静态单赋值形式(SSA)进行的可达性分析、常量传播和无用代码消除等技术。这一特性使得 SDFG 模型在表达可执行代码的粒度时具有极大的自由度，特别是无需担心 Tasklet 过度拆分而引入额外的运行时开销。这是因为多个连续的、顺序执行的 Tasklet 经过编译后将被整合为一个 C++代码块，不存在显式或隐式的同步机制，并在相同作用域内共享上下文环境，直接访问全局和局部变量进行数据读写与传输。借助现代的 C++语言的编译流程，SDFG 模型在程序优化过程中能够将重心放在数据流分析与优化中，而无需关注底层实现细节，从而实现更高效的程序表示与执行优化。与之相反的是，如果对 codelet 进行过于细粒度的拆分，则可能在 codelet 之间引入大量的同步开销，甚至导致逻辑上连续且相邻的 codelet 被调度至不同的执行单元，从而破坏程序的连续性，导致整体性能严重下降。因此，在将 SDFG 模型转换至 Codelet 模型的过程中，必须谨慎考虑任务拆分的粒度，以在并行度与调度开销之间取得平衡。

本文将连续且顺序执行的 Tasklet 映射至一个 codelet，仅在遇到控制流分支(如 Map 和 Consume 节点)或状态转移(即进入 SDFG 中的另一个状态)时，才将分支点或状态转移点前后的代码拆分至不同的 codelet 中进行处理。这种映射方式能够在保持程序逻辑连贯性和完整性的同时，避免过于细粒度的拆分，从而减少不必要的调度与同步开销。此外，该方法在尽可能保留程序内部的并行性结构及其他并行语义的同时，确保代码能够尽可能高效地执行，使得在映射过程中既能维持 SDFG 模型原有的并行计算特性，又能提升整体调度与执行效率。

3 实验设计与分析

3.1 实验环境与数据集

本实验采用 PolyBench/C[19] 作为源程序集进行优化测试。PolyBench/C 是一套为并行计算代码优化研究和性能评测设计的基准测试集，广泛用于编译器优化、计算机体系结构研究以及高性能计算领域。PolyBench 提供了一套典型的科学计算程序及算法核(Kernel)，如线性代数运算、图像处理、物理模拟和数据挖掘算法等。为了充分测试程序的并行能力，减少因并行带来的额外开销在程序运行时间中的占比，本文对 PolyBench/C 中的所有样例均采用特大规模的数据集大小(具体做法是在第一个编译步骤中添加编译参数-DEXTRALARGE_DATASET)，并以经过编译优化后的二进制程序的执行时间作为主要的性能评价指标。本文以具有 NUMA 结构的华为海思鲲鹏众核处理器为实验平台，具体配置如表 2 所示。

表 2 实验平台信息

Table 2 Experiment platform information

Item	Description
CPU	Dual-socket Huawei HiSilicon Kunpeng 920 Processor
Core count	96 cores (2 sockets×48 cores each)
Memory	192 GB
OS	Ubuntu 20.04.6 LTS
Kernel	Linux 5.4.0-196-generic
C++ compiler	GCC 9.4.0
DaCe version	0.15.1 (git commit 7056675)
Polygeist	(git commit e703db1)
PolyBench	Ver. 4.2.1 beta

3.2 程序性能评测

本文对 PolyBench 提供的每个科学计算程序分别使用多种全自动的方式编译，从而得到多份优化方式和优化等级各有差异的二进制文件作为不同的比较对象，如图 4 所示。其中：Plain 方案使用 GCC 直接编译 PolyBench 提供的源代码，不添加任何优化参数；O3 方案与 Plain 方案类似，但开启-O3 优化等级；SDFG 方案使用本文第 2 章介绍的构建系统，将 PolyBench 的源代码经过编译与转换流程，但在倒数第二步使用原始的 DaCe 框架直接将 SDFG 输出至 C++代码，并使用 GCC-O3 优化等级编译输出产物；DARTS 方案与 SDFG 方案类似，但经过完整的编译与转换流程，并将 SDFG 输出至 Codelet 模型的 C++代码，使用 GCC-O3 优化等级编译输出产物。

本文将 PolyBench 提供的源程序采用 4 种编译方案分别编译运行，结果如表 3 所示。其中：Plain 和 O3 方案的程序为单线程程序；SDFG 方案的程序采用 32 线程 OpenMP 运行；DARTS 方案的程序采用 4 个调度单元和 32 个计算单元运行。在优化 durbin 的过程中，DaCe 错误处理了一个局部变量的作用域，使得转换出来的DARTS方案的代码无

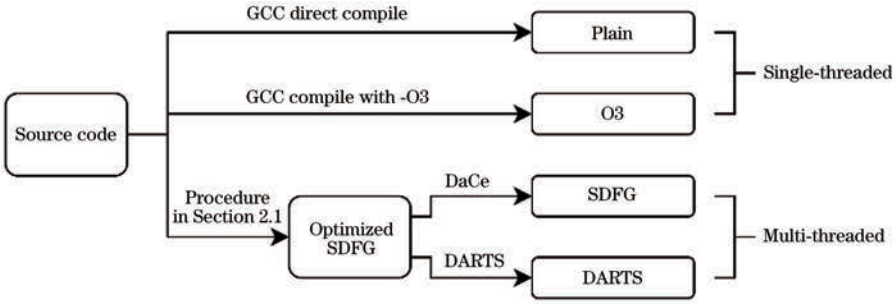

图 4 参与评测的编译方案

Fig. 4 Compilation schemes compared

法编译，表 3 中对应项目标记为“*”。另有 7 个样例的运行时间不足 0.1 s，考虑到测量精度为 10 ms，这些结果无法准确反映出不同编译方案的性能差异。在剔除以上 8 个样例后，本节对剩余的 21 个测试样例进行分析。

表 3 各编译模式下的程序运行时间对比

Table 3 Comparison of execution time of programs compiled under different compilation modes 单位：s

Test case	Plain	O3	SDFG	DARTS
2mm	325.34	222.27	14.49	17.03
3mm	452.06	371.25	23.53	36.86
adi	452.97	263.70	17.93	19.07
atax	0.09	0.06	0.03	0.07
bicg	0.08	0.07	0.03	0.03
cholesky	2 787.90	2 236.38	222.54	218.22
correlation	449.26	437.52	28.03	27.67
covariance	436.66	409.69	447.69	458.84
deriche	7.68	6.31	9.21	3.06
doitgen	29.91	7.98	4.37	6.13
durbin	0.11	0.02	0.08	*
fdtd-2d	178.49	45.31	6.03	12.89
floyd-warshall	961.26	285.61	540.86	243.40
gemm	92.14	14.61	1.48	0.90
gemver	0.90	0.70	0.06	0.12
gesummv	0.26	0.24	0.03	0.05
gramschmidt	691.65	690.02	45.93	185.51
heat-3d	536.46	93.02	9.24	21.64
jacobi-1d	0.04	<0.01	<0.01	0.01
jacobi-2d	198.87	59.86	5.65	13.42
lu	3 225.86	2 537.50	624.49	778.28
ludcmp	2 880.65	2 627.10	627.30	769.03
mvt	0.76	0.68	0.06	0.11
seidel-2d	444.86	266.97	267.63	267.26
symm	218.43	206.83	205.69	226.16
syr2k	301.12	260.84	18.50	10.05
syrk	130.22	88.79	7.82	3.39
trisolv	0.14	0.11	0.04	0.05
trmm	198.03	189.84	11.82	11.66

表 3 结果表明，以 SDFG 编译方案产生的程序的性能为参考基准，采用 DARTS 编译方案产生的程序的平均性能可以达到 SDFG 方案的 94%（几何平均值），并在其中 9 个样例中领先于 SDFG 方案，且最高加速比达到 3，如图 5 所示。整体来看，DARTS 方案应用于结构和算法更不规则的程序时的性能表现更好。此外，在这 21 个样例中，如果将 SDFG 方案相较于 O3 方案的加速比低于 2 定义为“难以并行”的，则有 6 个样例符合此标准。在这 6 个难以并行的样例中，DARTS 方案的程序性能达到了 SDFG 方案的 97%（几何平均值），说明 DARTS 方案在面对并行性较差的程序时，其产生的额外开销也较小。

由以上实验结果可见，尽管 DARTS 运行时框架采用依赖计数与延迟实例化等技术产生了较大的额外开销，但通过本文设计的流程全自动化构建及优化得到的 Codelet 数据流程序仍然能够使其执行性能与基于传统并行模式的程序基本持平。

3.3 案例分析

基于表 3 所呈现的实验结果，本节选取具有代表性的样例进行深入分析，讨论 DARTS 编译方案相比于 SDFG 方案的优缺点。借助 Linux 系统提供的 perf 采样能力与 flamegraph-rs[21] 工具，本研究能够对程序运行过程中的热点函数及其调用栈进行精细化分析，从而揭示不同方案在性能表现上的差异，并提供相应的解释与讨论。

首先，以 covariance 为例，其通过 SDFG 方案和 DARTS 方案产生的程序均表现出极差的并行性。从输出的 C++代码中观察到，其主要的计算核具有三层嵌套循环，且由于其中具有重叠的访存模式，SDFG 数据流模型的优化能力在这里难以发挥作用，将这三层嵌套循环结构原样保留了下来。对应地，SDFG 方案和 DARTS 方案产生的程序均花费了大量的时间在单线程的运算上，从而使得并行机制的差异带来的额外开销并不显著，此时 SDFG 版本和 DARTS 版本的程序没有明显的性能差异。

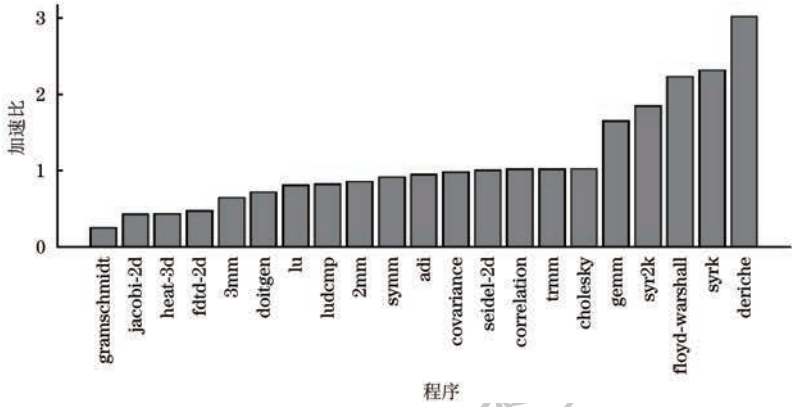

图 5 DARTS 方案相对于 SDFG 方案在 21 个程序中的加速比

Fig. 5 Acceleration ratio of DARTS over SDFG in 21 programs

其次，在并行度较好的程序中，本文选取 3mm 为例，其算法为 3 个矩阵的乘法，具有十分规律的并行模式。在该样例中，通过 DARTS 方案产生的程序相比于 SDFG 方案仅获得了 64%的加速比。从程序的运行情况来看，SDFG 版本和 DARTS 版本的程序的全程平均 CPU 占用率分别为 3 087%和 3 139%，且两个版本的程序的 perf 结果也并未展现出异常。然而，本文注意到，SDFG 版本的程序全程仅有 1.08 万次主动上下文切换，而 DARTS 版本的程序有高达 805 万次主动上下文切换，是 SDFG 版本的约 745 倍。同时，SDFG 版本的程序发生了 32 万次软缺页，而 DARTS 版本的程序则发生了 93 万次软缺页，是 SDFG 版本的约 2.9 倍。根据输出的代码分析，DaCe 将三个双层循环的循环体分别划分至 NestedSDFG 节点中，从而使 DARTS 版本的程序为每次迭代都创建了一个 TP，在此过程中发生了大量的内存分配与释放动作；而在 SDFG 版本的程序中，NestedSDFG 节点仅仅翻译为一个简单函数调用，因而避免了大量的内存管理操作，减少了上下文切换的开销，从而获得了较好的运行性能。

最后，在并行度较好的程序中，本文选取 syrk 为例，其通过 DARTS 方案产生的程序相比于 SDFG 方案获得了 231%的加速。本文注意到在 SDFG 版本程序的运行时 perf 采样结果中，主线程仅出现在不足 1%的采样点中，据此推测由 OpenMP 运行时创建的工作线程承担了主要的计算任务。同时，按照 Linux 对 CPU 占用率的统计方式，以单个 CPU 线程保持满载所产生的 CPU 占用率作为 100%，则 SDFG 版本的程序全程平均 CPU 占用率仅为 561%，而 DARTS 版本的程序维持了 3 098%的平均 CPU 占用率。在此样例中，由于该算法的并行计算模式较为不规则，Codelet 模型的灵活性使得其能够更充分地利用 CPU 的计算能力，从而获得更高的计算性能。

4 结束语

针对 Codelet 细粒度数据流模型编程困难的问题，本文提出了一种新型的程序转换方式。首先，以 MLIR[15] 和 SDFG 数据流模型[14] 两级中间表示层为桥梁，本文设计了一套全自动的程序编译、优化、转换与构建流程，使得现有的基于传统并行编程模式的程序能够无缝迁移至 Codelet 数据流模型；其次，本文讨论了两种数据流模型在设计思想与概念上的诸多差异，以及由此带来的不一致性；接着，本文将数据流模型转换时遇到的困难与挑战细分为层次结构、可并行区域的代码结构、上下文与局部变量、同步语义等问题，对这些问题逐一提出应对措施和解决方案，并对代码的拆分粒度提出了优化方法；最后，本文阐述了前述转换方案的一种实现方式，并设计实验分析了本文方案的实现成果。

实验结果表明，在具有众多硬件线程的平台上，本文为 Codelet 数据流编程模型设计的自动化程序转换方案所产生的程序性能可以与已知最佳的 OpenMP 并行化实现相当，并在近一半的样例中获得更好的性能。基于实验结果，本文选取了 3 个典型的测试样例，分别讨论了本文方案处理具有不同属性的程序的性能表现，并深入分析了程序性能优劣的原因。特别地，对于规则的计算模式，传统的基于 OpenMP 的并行方式通常性能更好，而 Codelet 数据流模型则在计算和并行结构不规则的程序上更有优势。

本文提出的方案仍有一定的局限性，后续将从多个角度开展工作，进一步改进输出的程序性能。例如参考 AceMesh[22] 引入编译制导语句或 PaRSEC[23] 附加标注文件，借助由用户提供的额外信息，在转换数据流图时进行更加精细的控制；也

可与数据流运行时系统联动，在输出的程序中插入提示信息，帮助运行时系统进行更好的调度决策，以期降低运行时的额外开销。另外，文献[24-25]设计了直接基于 Codelet 模型的 MLIR 方言，也可将其作为后续工作的程序转换目标或中间状态。

参考文献

[1] LUNDSTROM M S, ALAM M A. Moore's law: the journey ahead [J]. Science, 2022, 378(6621): 722-723.

[2] LEE E A. The problem with threads[J]. Computer, 2006, 39(5): 33-42.

[3] GROPP W, LUSK E, DOSS N, et al. A high-performance, portable implementation of the MPI message passing interface standard[J]. Parallel Computing, 1996, 22(6): 789-828.

[4] DAGUM L, MENON R. OpenMP: an industry standard API for shared-memory programming [J]. IEEE Computational Science and Engineering, 1998, 5(1): 46-55.

[5] SANDERS J, KANDROT E, SANDERS J, et al. CUDA by example: an introduction to general purpose GPU programming [M]. Boston, USA: Addison-Wesley Professional, 2010.

[6] STONE J E, GOHARA D, SHI G C. OpenCL: a parallel programming standard for heterogeneous computing systems [J]. Computing in Science & Engineering, 2010, 12(3): 66-73.

[7] DENNIS J B. Data flow supercomputers[J]. Computer, 1980, 13(11): 48-56.

[8] FLYNN M J. Some computer organizations and their effectiveness[J]. IEEE Transactions on Computers, 1972, 100(9): 948-960.

[9] JOHNSTON W M, PAUL H J R, MILLAR R J. Advances in dataflow programming languages[J]. ACM Computing Surveys, 2004, 36(1): 1-34.

[10] OVIEDO E I. Control flow, data flow and program complexity[D]. Buffalo, USA: State University of New York at Buffalo, 1984.

[11] BAUER M, TREICHLER S, SLAUGHTER E, et al. Legion: expressing locality and independence with logical regions[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. New York, USA: ACM Press, 2012: 1-11.

[12] PHEATT C. Intel threading building blocks[J]. Journal of Computing Sciences in Colleges, 2008, 23(4):298.

[13] GRAY A L. CUDA graphs: flexible dependency representation for efficient GPU work submission[EB/OL]. [2025-01-10]. https://developer.nvidia.com/blog/cuda-graphs.

[14] BEN-NUN T, FINE L J, ZIOGAS A N, et al. Stateful dataflow multigraphs: a data-centric model for performance portability on heterogeneous architectures[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. New York, USA: ACM Press, 2019: 1-14.

[15] LATTNER C, AMINI M, BONDHUGULA U, et al. MLIR: scaling compiler infrastructure for domain specific computation[C]//Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Seoul, Korea: IEEE Press, 2021: 2-14.

[16] BEN-NUN T, ATES B, CALOTOIU A, et al. Bridging control-centric and data-centric optimization[C]//Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization. New York, USA: ACM Press, 2023: 173-185.

[17] SUETTLERLEIN J, ZUCKERMAN S, GAO G R. An implementation of the Codelet model[C]//Proceedings of Euro-Par 2013. Berlin, Germany: Springer, 2013: 633-644.

[18] SUETTERLEIN J. DARTS: a runtime based on the Codelet execution model [D]. Newark, USA: University of Delaware, 2014.

[19] YUKI T, POUCHET L N. PolyBench 4.2.1 (pre-release) [EB/OL]. [2025-01-10]. https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1/blob/master/polybench.pdf.

[20] MOSES W S, CHELINI L, ZHAO R Z, et al. Polygeist: raising C to polyhedral MLIR[C]//Proceedings of the 30th International Conference on Parallel Architectures and Compilation Techniques (PACT). Atlanta, USA: IEEE Press, 2021: 45-59.

[21] flamegraphs for Rust[EB/OL]. [2025-01-10]. https://github.com/flamegraph-rs/flamegraph.

[22] CHEN L, TANG S L, FU Y, et al. AceMesh: a structured data driven programming language for high performance computing[J]. CCF Transactions on High Performance Computing, 2020, 2(4): 309-322.

[23] BOSILCA G, BOUTEILLER A, DANALIS A, et al. PaRSEC: exploiting heterogeneity to enhance scalability[J]. Computing in Science & Engineering, 2013, 15(6): 36-45.

[24] KABRICK R, PERDOMO D A R, RASKAR S, et al. CODIR: towards an MLIR codelet model dialect[C]//Proceedings of the 4th Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM). Washington D. C., USA: IEEE Press, 2020: 33-40.

[25] 李金熹, 尹首一, 魏少军, 等. 基于 MLIR 的数据流模型[J]. 计算机工程与科学, 2024, 46(7): 1151-1157.
LI J X, YIN S Y, WEI S J, et al. A codelet model based on MLIR[J]. Computer Engineering and Science, 2024, 46(7): 1151-1157. (in Chinese)

文字编辑 金胡考
栏目编辑 宋 圆