软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2023,34(4):1570−1593 [doi: 10.13328/j.cnki.jos.006688]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

面向申威异构架构的并行代码自动生成*

陶小涵[1,2], 朱 雨[1,2], 庞建民[1,2], 赵 捷[1,2], 徐金龙[1,2]

[1](信息工程大学, 河南 郑州 450001)

[2](数学工程与先进计算国家重点实验室, 河南 郑州 450001)

通信作者: 庞建民, E-mail: jianmin_pang@126.com

摘 要: 异构架构逐渐成为高性能计算领域的主流架构, 但相较于同构多核架构, 其硬件结构及存储层次更为复杂, 程序编写更为困难. 先进的优化编译器可以协助程序开发人员实现更为高效的代码, 降低程序开发复杂度. 多面体编译模型通过抽象分析将程序抽象成空间多面体表示形式, 能够将多种循环变换与硬件映射相结合, 并面向特定体系结构生成相应的代码. 设计实现了一个面向国产申威异构架构的并行代码自动生成系统, 采用"源-源"编译模式, 基于多面体编译模型实现. 系统针对申威异构架构特点将程序计算过程进行硬件部署, 同时实现数据传输与内存空间的自动管理. 实验基于 Polybench 测试集中线性代数相关用例进行测试. 结果表明, 利用代码自动生成系统生成的异构并行代码能够在申威异构平台上正确运行, 并能够有效发挥申威异构平台的性能, 基于申威异构平台利用 64 线程加速计算的平均加速比达到了 539.16 倍.

关键词: 申威异构架构; 多面体模型; 并行计算; 代码生成

中图法分类号: TP311

中文引用格式: 陶小涵, 朱雨, 庞建民, 赵捷, 徐金龙. 面向申威异构架构的并行代码自动生成. 软件学报, 2023, 34(4): 1570−1593. http://www.jos.org.cn/1000-9825/6688.htm

英文引用格式: Tao XH, Zhu Y, Pang JM, Zhao-J, Xu JL. Parallel Code Generation for Sunway Heterogeneous Architecture. Ruan Jian Xue Bao/Journal of Software, 2023, 34(4): 1570−1593 (in Chinese). http://www.jos.org.cn/1000-9825/6688.htm

Parallel Code Generation for Sunway Heterogeneous Architecture

TAO Xiao-Han[1,2], ZHU Yu[1,2], PANG Jian-Min[1,2], ZHAO-Jie[1,2], XU Jin-Long[1,2]

[1](Information Engineering University, Zhengzhou 450001, China)

[2](State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China)

Abstract: Heterogeneous architectures are dominating the realm of high-performance computing. However, these architectures also complicate the programming issue due to its increasingly complex hardware and memory hierarchy compared to homogeneous architectures. One of the most promising solutions to this issue is making use of optimizing compilers which can help programmers develop high-performance code executable on target machines, thereby simplifying the difficulty of programming. The polyhedral model is widely studied due to its ability to generate effective code and portability to various targets, which is realized by first converting a program into its intermediate representation and then combining the compositions of loop transformations and hardware binding strategies. This paper presents a source-to-source parallel code generator targeting the domestic, heterogeneous architecture of the Sunway machine using the polyhedral model. In particular, the computation is deployed automatedly onto the Sunway architecture and memory management, minimizing the amount of data movements between the management processing element and computing processing elements of the target. The experiments are conducted on 13 linear algebra applications extracted from the Polybench Benchmarks. The experimental results show that the proposed approach can generate effective code executable on the Sunway heterogeneous architecture, providing a mean speedup of 539.16 × on 64 threads over the sequential implementation executed on a management processing element.

Key words: Sunway heterogeneous architecture; polyhedral model; parallel computing; code generation

* 基金项目: 国家自然科学基金(61702546)

收稿时间: 2021-11-25; 修改时间: 2022-02-02; 采用时间: 2022-03-17

在高性能计算领域, 计算机系统不仅需要提供更强的计算能力, 其在功耗及散热等方面也面临着诸多挑战. 目前, 主流的高性能计算机大多采用了“主处理器+加速部件”的异构架构, 如Summit、Sierra[1]超级计算机, 均使用NVIDIA公司的Tesla V100[2]作为加速部件, 旨在实现更高的性能功耗比和计算密度. 国产超级计算机“神威·太湖之光”采用的异构众核处理器SW26010则在处理器内部采用了主从核异构架构[3], 将具有控制管理功能的通用处理器核心和大量用于加速计算的精简计算核心集成在同一芯片上, 使得“神威·太湖之光”的性能功耗比提升到了6.05 GFlops/W.

异构架构为高性能计算提供了诸多优势的同时, 也由于其更为复杂的体系结构及内存层次, 给程序设计及编译优化带来了更大的挑战. 从程序设计及编译优化的角度看, 异构架构主要有以下特点: 一是加速部件采用的精简计算核心结构与主处理器的通用核心结构不一致, 需要利用各自的结构特点, 进一步挖掘程序的并行性; 二是主流异构架构处理器中大量使用了通过软件管理的SPM (scratch-padmemory, 便签式存储器), 需通过应用程序管理SPM空间, 并显式地实现数据在主处理器与加速部件之间的传输; 三是多数异构架构中加速部件所采用的SPM空间十分有限, 如NVIDIA的TeslaV100中采用SPM结构的共享内存可用空间最大为128 KB, 而SW26010中采用SPM结构的LDM (local data memory, 局部数据存储)可用空间为64 KB, 这使得程序设计过程中需要在保证程序并行性的同时, 尽可能地提升数据局部性, 以获得更好的程序执行效率.

随着异构架构的发展, 其硬件特征及存储层次越来越多样化, 适用于异构架构的并行编程模型同样日趋复杂. 程序设计人员为编写出执行效率很高的程序, 需熟悉应用程序相关领域, 了解硬件架构特征, 精通编程模型, 使得手工编写异构并行程序变得尤为困难. 利用异构架构进行并行程序开发无疑对程序设计人员提出了更高的要求, 也使得手工优化程序的效率降低, 因此面向异构架构的并行代码自动生成及编译优化技术的重要性越来越突出. 但在目前, 在面向异构架构的并行代码自动生成技术中, 大多数的研究工作仅仅是面向CPU-GPU平台的. 例如, 多面体编译工具PPCG[4]可在GPU平台上实现由串行程序向CUDA及OpenCL异构并行程序的自动变换. 虽然PPCG针对异构平台仅支持GPU平台的自动代码生成, 但其利用的多面体编译模型[5]可以将循环变换和硬件映射结合, 这为在异构众核平台上实现代码自动生成提供了新的思路.

为解决国产申威异构架构上的编程难问题, 从而为国产芯片及其软件生态的发展做出贡献, 本文基于多面体编译模型设计实现了一个面向申威异构架构的并行代码自动生成系统. 该系统通过“源-源”编译模式, 可以将C语言程序中用户指定的并行区串行代码转换为能够在申威异构架构上运行的主从核并行代码. 系统主要由4个模块组成.

(1) 预处理模块通过模型提取工具提取出程序中的多面体模型, 并进行依赖关系分析, 根据依赖关系分析结果进行循环变换;

(2) 硬件映射模块将并行程序进行任务划分以部署到运算核心;

(3) 内存管理模块构建通用管理核心与加速部件之间的数据传输集合, 实现精确且高效的数据传输过程;

(4) 代码生成模块依据硬件映射且优化后的多面体模型中间表示生成面向申威异构架构的异构并行程序.

为了验证系统的有效性, 我们利用SW26010异构众核处理器以及Polybench测试集评估所生成的申威异构并行代码正确性及运行效率. 实验结果表明, 本文所实现的异构并行代码自动生成系统能够面向申威异构架构生成高效的并行程序, 在64线程规模下运行时, 相较于串行程序能够获得539.16倍的平均加速比, 同时大大降低了并行程序开发成本.

1 背景知识

1.1 申威异构架构

申威架构在芯片内实现了“主处理器+加速部件”的主从核异构架构[6], 最具代表性的是“神威·太湖之光”超级计算机, 其核心为国产SW26010异构众核处理器. SW26010是面向高性能计算领域开发的处理器, 采用片上计算阵列集群和分布式共享存储结构相结合的异构架构, 使用定制的申威指令集, 是“神威·太湖之光”得以实现高计算速度和低功耗的核心部件.

如图 1 所示, SW26010 处理器芯片主要由 4 个核组、片上互联网络和系统接口组成. 核组是系统进行计算资源管理的基础单位, 包括一个通用管理核心、一个运算核心阵列和存储控制器等. 每个运算核心阵列包含 64 个精简运算核心, 采用 8×8 的阵列通信网络进行连接.

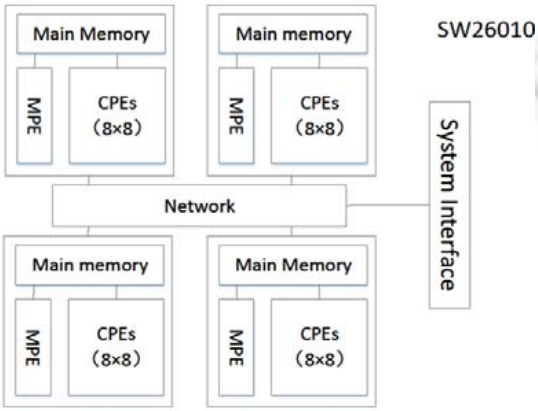


图 1 SW26010 芯片结构图

申威架构在核组间利用 MPI (message processing interface)实现进程级并行, 在核组内采用 Athread 并行编程模型[7]实现线程级并行, 利用运算核心阵列实现主从加速并行方式, 如图 2 所示. 程序的核心段通过 Athread 模型开启线程组并被加载到运算核心阵列上进行加速计算, 在运算核心阵列进行核心段运算过程中, 通用管理核心处于等待状态, 直至运算核心阵列完成该核心段的计算任务. 由于申威平台在线程组开启时开销较大, 因此我们将尽可能保证并行粒度最大化, 即尽可能实现外层循环的并行执行.

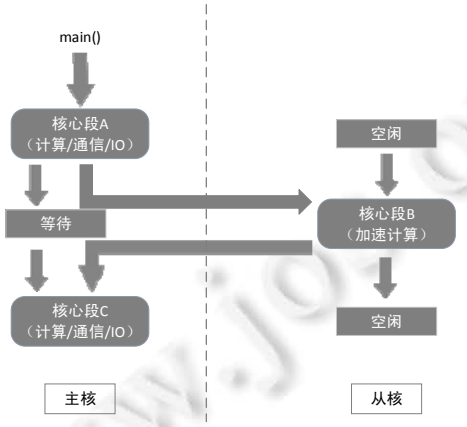


图 2 主从加速并行

存储结构方面, SW26010 处理器单个芯片上 4 个核组的物理空间统一编址, 通用管理核心和运算核心均可以访问芯片上的所有主存空间. 单个核组的存储系统结构如图 3 所示, 通用管理核心拥有 *L*1 和 *L*2 两级硬件 Cache, 运算核心存储为采用 SPM 结构的 LDM, LDM 空间大小为 64 KB. 运算核心可以通过 Athread 模型加载 DMA 命令实现 LDM 和主存之间的批量数据传输. DMA 传输的效率与传输的数据量、DMA 命令数量、数据在内存中的连续性以及 DMA 传输方式等密切相关, 硬件同时支持阻塞 DMA 传输与非阻塞 DMA 传输. 虽然通用管理核心与运算核心都可以访问主存和 LDM, 但处理器核心访问不同存储器的延迟有很大差别, 运算核心访问 LDM 的延迟远低于访问主存的延迟, 而芯片面积及功耗的限制导致 LDM 的空间有限. 因此在实

际应用中, 运用运算核心进行加速计算时, 如何充分利用访问时延短的 LDM 是发挥 SW26010 处理器性能优势的关键.

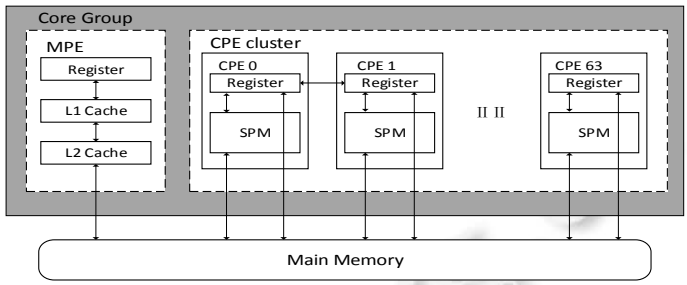


图 3 申威架构存储结构图

1.2 多面体模型

多面体模型是将语句实例抽象成空间多面体, 并通过这些多面体上的集合操作来分析和指导循环变换的编译优化模型[5,8]. 多面体模型通常使用迭代空间(语句实例集合)、访存关系(语句实例与数据访问的映射关系)、依赖关系(语句实例之间的依赖关系)和调度(语句执行顺序)表示程序及其语义.

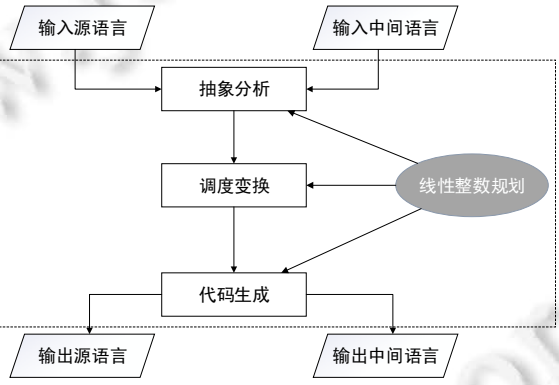


图 4 多面体模型一般编译流程

如图 4 所示, 一般而言, 多面体模型编译工具的编译流程分为抽象分析[9]、调度变换[10−13]及代码生成[14−16] 3 个部分, 均以线性整数规划工具为基础. 其中, 抽象分析过程提取出程序的多面体模型并进行依赖分析; 调度变换依据依赖分析结果实现合法的循环变换组合; 代码生成则将多面体模型中间表示依据目标体系结构的编程模型生成对应的并行程序.

```
#pragma scop
  for (i=0;i<M;i++)
    for (j=0;j<N;j++)
      C[i][j]=0;
  for (i=0;i<M;i++)
    for (j=0;j<N;j++)
      for (k=0;k<K;k++)
        C[i][j]+=A[i][k]*B[k][j];
#pragma scop
```

图 5 矩阵乘计算 C 语言代码

以图 5 所示的矩阵乘计算为例, 多面体模型将该计算抽象成三维空间上的多面体形式. 其调度可以被表示为

$$\{S_0(i,j)\rightarrow(0,i,j);S_1(i,j,k)\rightarrow(1,i,j,k)\} \tag{1}$$

其中, $S_0(i,j)$和 $S_1(i,j,k)$表示语句的实例, 分别对应图 5 中数组 C 的初始化语句以及矩阵乘计算规约语句; $(0,i,j)$

和(1,*i*,*j*,*k*)表示语句的执行顺序, “→”表示从语句实例到语句执行顺序的映射关系. 以语句 S_0 为例, 其所有实例将按(0,*i*,*j*)的方式执行, 即按照先沿 *i* 轴再沿 *j* 轴的顺序执行. 值得注意的是, 上述调度中语句执行顺序中存在的常量 0 和 1 为逻辑时间上的偏序关系, 即在常量所在维度上, 常量 0 所对应的语句须在常量 1 所对应语句被执行前执行. 根据计算出的该循环嵌套依赖关系, 多面体模型能够计算出一个新的调度, 其结果为

$$\{S_0(i,j)\to(i,j,0);S_1(i,j,k)\to(i,j,1,k)\} \tag{2}$$

由式(1)、式(2)的变换过程即多面体模型对该矩阵乘计算实施了循环合并优化, 优化后的代码如图 6 所示.

```
for (i=0;i<M;i++)
  for (j=0;j<N;j++){
      C[i][j]=0;
      for (k=0;k<K;k++)
          C[i][j]+=A[i][k]*B[k][j];
  }
```

图 6　循环合并后矩阵乘计算 C 语言代码

另外, 以此程序为例, 多面体模型不仅能够实现多种循环变换, 还能够分析出 *i*、*j* 两层循环是否可以并行执行以及循环分块在此程序中的合法性, 有效降低了程序开发人员在并行优化中程序分析的复杂度.

1.3 调度树

由于多面体模型中的调度被用来表示程序中语句实例的执行顺序, 所以调度在本质上可以表示为树的形式. 除传统的集合与映射外, 多面体模型中有多种调度的中间表示形式, 如 Kelly 等人提出的中间表示[17]、“2*d*+1”形式的中间表示[18]、Presburger 算数表达式[19]以及调度树[16,20]等. 本文采用调度树作为多面体模型的中间表示形式.

基于调度树, 我们可以很方便地实现多种编译优化操作, 如仿射变换、循环交换、循环合并与分布、循环分块等, 并且, 多面体模型中代码生成模块可以根据调度树生成对应的抽象语法树(AST), 进而生成面向特定体系结构的并行代码. 然而, 相较于其他多面体模型中间表示, 调度树最大的优势在于其可以很方便地在对应位置插入数据传输语句、同步语句等操作, 从而实现代码的硬件映射, 这也是 PPCG 等多面体编译器实现面向 GPU 等异构架构生成代码的最重要功能之一.

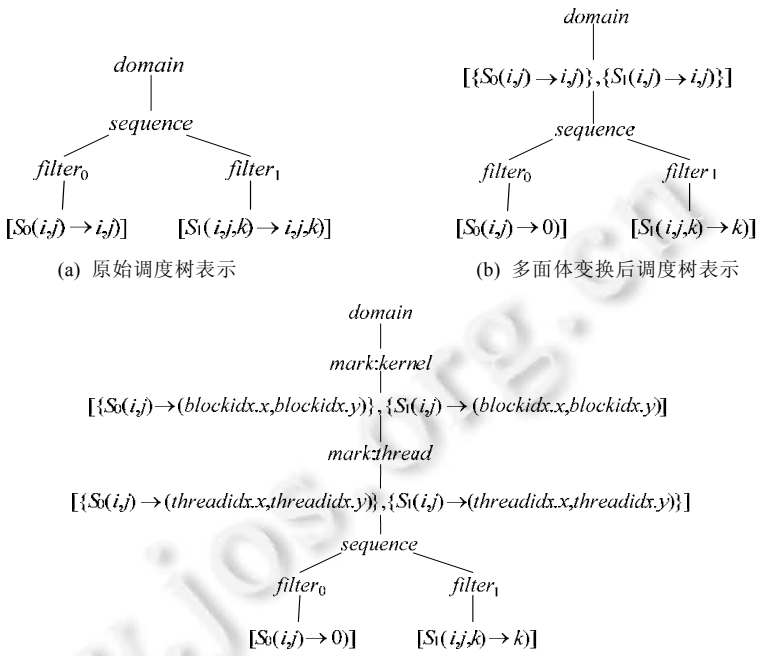
图 7(a)是图 5 中矩阵乘计算在多面体模型变换前的调度树表示, 经多面体变换实现循环合并后的调度树如图 7(b)所示, 而图 7(c)则为该矩阵乘计算映射到 GPU 平台后的调度树表示. 图中 $filter_0$ 和 $filter_1$ 集合表示为

$$filter_0=\{S_0(i,j):0\leqslant i<M\wedge 0\leqslant j<N\} \tag{3}$$

$$filter_1=\{S_1(i,j,k):0\leqslant i<M\wedge 0\leqslant j<N\wedge 0\leqslant k<K\} \tag{4}$$

图 7 所示调度树中, *domain* 节点表示程序的迭代空间, 并且必须作为调度树的根节点出现. *sequence* 节点表示其子节点之间必须顺序执行, 其子节点须为 *filter* 节点. *filter* 节点可以作为调度树中子树的根节点出现, 其表示被当前子树调度的所有语句实例的集合. *band* 节点包含了语句实例的部分调度, 可以直观想象为循环嵌套. 在图 7 中, 由(a)到(b)的变换过程可以看出, 语句 S_0 和 S_1 所对应 *band* 节点的 *i* 和 *j* 两层循环被合并到一起, 并作为 *sequence* 节点的父节点, 该过程即实现了循环合并优化. *mark* 节点可被用于标记调度树中子树的任何信息, 如标记子树在 CUDA 编程模型中对应 *grid* 或 *block*. 如图 7(c)中, *mark*:*kernel* 节点的子树将映射到 GPU 的一个 *grid* 上执行, *mark*:*thread* 节点的子树将被映射到一个 *block* 上执行.

除上述节点外, 我们还将在多面体变换过程中使用 *extension* 节点. *extension* 节点表示在程序中新添加需要被调度的迭代空间元素, 一个典型的应用即为在程序中添加数据传输语句以实现对异构架构存储层次的有效利用.



(a) 原始调度树表示 (b) 多面体变换后调度树表示

(c) 映射到 GPU 平台后调度树表示

图 7 矩阵乘计算调度树表示

1.4 研究现状及动机

目前, 基于申威异构架构进行并行程序设计及优化的工作大多由程序开发人员手工实现. 例如, 刘芳芳等人[21]针对稀疏矩阵乘法操作面向申威处理器提出了一种通用异构众核并行算法, 从任务划分、LDM 空间划分等方面进行了精细设计, 并利用动静结合的任务调度方法以实现负载均衡. 许志耿[22]结合申威架构中寄存器通信机制, 针对稠密矩阵乘法重新设计了上层并行计算算法, 并设计了基于寄存器通信的核间数据共享方案以优化 stencil 计算在申威架构上的执行效率.

通过程序员手工优化可以很大程度地提升程序执行效率, 甚至接近硬件处理器峰值性能, 但其编程门槛较高, 需程序员对申威异构架构及程序算法有很深入的了解. 同时, 手工实现的方法只是针对某个课题或是某个核心计算代码段进行优化, 不具有通用性. 而面向申威异构架构实现并行代码自动生成的工作较少, Zhu 等人[23]针对 2D-stencil 计算实现了面向 SW26010 异构众核处理器的并行代码自动生成器, 将海洋模型系统中的 stencil 核心计算转换为基于 Athread 编程模型的并行代码, 相较于串行版本能够获得约 3.8 倍加速. 在本课题组前期研究工作中, 李雁冰等人[24]基于 Open64 开源编译器实现了面向 SW26010 处理器的并行编译框架, 可以将程序自动转换为 OpenACC 并行程序, 但 OpenACC 编程模型相较于 Athread 代码执行效率较低, 且该工作仅能处理规则的数据及计算, 相较于基于多面体模型的并行代码自动生成有很大的局限性.

基于多面体模型面向异构架构并行代码自动生成的工作主要集中在 CPU-GPU 及分布式存储平台等. 例如前文中提到的面向 CPU-GPU 异构架构的并行代码自动生成工具 PPCG[4]. Shirako 等人[25]依据 CUDA 编程模型中两级并行的特点, 对 PPCG 中调度变换算法进行了进一步优化. Grosser 等人[26]基于 LLVM 编译器中 Polly 模块实现了一种从编译器中间表示到中间表示的代码生成工具 Polly-ACC, 其最终能够面向 CPU-GPU 架构生成 CUDA 或 OpenCL 代码. 而多面体模型编译器 Tiramisu[27], 则可以根据用户提供的计算表达式面向多核 CPU、CPU-GPU、FPGA 以及分布式结构生成并行代码, 并且实现数据传输控制、同步以及数据向不同存储结构的映射. 除此之外, 基于多面体模型实现的优化编译器同样可以面向人工智能领域 NPU 完成并行代码的自动生成及深度优化工作, 这其中, AKG[28]作为代表, 可以面向华为公司推出的人工智能芯片昇腾

910[29]生成异构并行代码, 并且基于多面体模型提出了一种新的循环分布与循环合并的组合方式, 有效提升了并行程序的访存效率[30].

可以看出, 当前申威异构架构上缺乏通用的并行代码自动生成工具, 且基于多面体模型面向其他异构架构自动生成并行代码的前人经验积累丰富. 因此, 本文将基于多面体模型开发面向申威异构架构的并行代码自动生成系统.

2 系统框架

面向申威异构架构的并行代码自动生成系统采取“源-源”编译模式, 将 C 语言程序作为输入, 并利用#pragmascop 和#pragmaendscop 编译指示指定程序并行区, 如图 5 所示. 系统针对用户指定并行区代码进行分析、变换, 并参照申威 Athread 编程模型转换为异构并行代码, 最终得到可以在申威异构架构上以主从加速并行方式执行的并行程序.

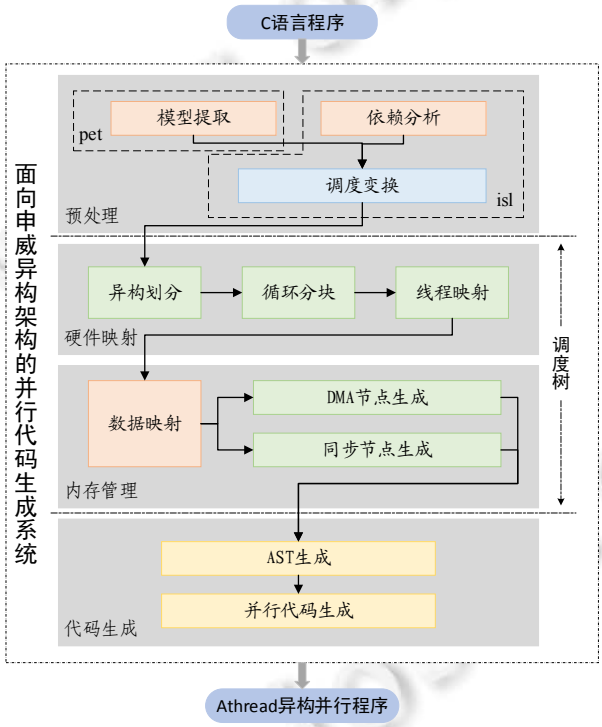


图 8 并行代码自动生成系统框架示意图

并行代码自动生成系统的编译优化流程如图 8 所示, 系统由预处理、硬件映射、内存管理以及代码生成 4 个模块组成, 其中, 硬件映射、内存管理以及代码生成 3 个模块(图 8 中绿色及黄色步骤)将是本文工作的核心.

(1) 预处理模块将首先根据程序特征构建多面体模型, 并进行依赖关系分析, 进而利用 isl (integer set library)调度算法[31]挖掘程序并行性以及完成合法的调度变换, 生成调度树中间表示. 在该过程中, 由于模型提取、依赖分析以及调度变换过程均为标准化过程, 因此我们采用 pet (polyhedral extraction tool)[32]实现模型提取功能, 并采用 isl 库[31]作为依赖分析和调度变换的工具;

(2) 硬件映射模块首先将并行区代码进行主从核异构划分, 再通过循环分块变换出与硬件特征相对应的循环层, 将计算过程及数据根据硬件特征进行划分, 并决定计算过程与硬件之间的映射关系;

(3) 内存管理模块将对语句实例和访存操作之间的映射关系进行分析, 并对数据访问进行分组, 通过在

调度树中添加 DMA 节点及同步节点实现数据在主从核间显式的非阻塞传输;

(4) 代码生成模块根据上述过程得到的调度树中间表示首先生成 AST, 再由 AST 生成对应的申威主从核并行代码.

值得注意的是, 如图 8 所示, 系统中硬件映射及内存管理模块将基于调度树中间表示实现, 通过对调度树的一系列操作(图中绿色步骤)实现程序的变换.

3 预处理

预处理模块将 C 语言程序的用户指定并行区抽象成多面体模型的调度树中间表示形式, 同时挖掘程序并行性, 用以进一步分析及优化. 该模块由模型提取、依赖关系分析及调度变换 3 个子模块组成. 由于上述 3 个子模块均为标准化过程且采用 pet 及 isl 库实现, 因此本文中仅利用图 5 所示矩阵乘计算为例解释其在并行代码自动生成系统中的基本工作流程, 而 pet 和 isl 库的实现原理及算法可见文献[31,32].

3.1 模型提取

系统通过调用 pet 构建程序并行区的多面体模型, 其中包括迭代空间、访存关系以及调度, 均以仿射约束形式表示. 迭代空间表示输入程序中语句实例的集合, 每一个语句实例均可以由迭代变量的值唯一确定. 以矩阵乘计算为例, 其迭代空间为

$$\left\{S_0(i,j):0\leqslant i<M\wedge 0\leqslant j<N;S_1(i,j,k):0\leqslant i<M\wedge 0\leqslant j<N\wedge 0\leqslant k<K\right\} \tag{5}$$

访存关系描述了语句实例与其访问的数据元素之间的映射关系, 包括了写访存关系与读访存关系. 示例矩阵乘计算在满足公式(5)中迭代空间约束的前提下, 其写访存关系为

$$\left\{S_0(i,j)\to C(i,j);S_1(i,j,k)\to C(i,j)\right\} \tag{6}$$

读访存关系为

$$\left\{S_1(i,j,k)\to A(i,k);S_1(i,j,k)\to B(k,j);S_1(i,j,k)\to C(i,j)\right\} \tag{7}$$

调度确定了语句实例的执行顺序, 每一个语句实例与一个整数元组一一对应, 将按照整数元组的字典序进行执行. 示例矩阵乘计算的原始调度可以由公式(1)表示.

3.2 依赖关系分析

依赖关系是保证程序变换合法性的前提, 包括循环变换在内的各种重排序变换只有在满足依赖基本定理的基础上才能保证程序的正确执行[33]. 因此, 在利用调度变换发掘程序并行性之前, 依赖关系分析过程是必不可少的.

仍以矩阵乘计算为例, 根据计算出的访存关系, 我们可以获得其语句实例间的依赖关系:

$$\left\{S_0(i,j)\to S_1(i,j,0);S_1(i,j,k)\to S_1(i,j,k+1)\right\} \tag{8}$$

其中, 变量 i、j、k 均须满足公式(5)中迭代空间约束条件. 我们将依赖关系结合迭代空间以及公式(1)所示的原始调度进行分析, 可以得出原始程序两个循环中 i 和 j 两个维度可以并行执行, 所以我们可以将循环的 i 层和 j 层合并, 同时进行循环分块操作, 保证后续硬件映射过程的合法性.

3.3 调度变换

调度变换是多面体模型中的核心, 其需要在依赖关系分析的基础上通过循环仿射变换挖掘程序并行性. 同依赖关系分析一样, 面向申威异构架构的并行代码自动生成系统基于 isl 库实现调度变换过程.

循环中不存在循环携带依赖是循环能够被并行执行的充分条件. 因此, 我们需要先利用多面体模型调度算法, 通过循环倾斜、循环偏移等仿射变换消除循环携带依赖, 挖掘程序并行性. 另外, 在保证程序并行性的同时, 可以利用循环合并、循环交换等仿射变换尽可能地提升数据局部性, 进一步提高并行程序的执行效率.

isl 库中所采用的是基于标准 Pluto 调度算法[10]改进的 isl 调度算法[31]. 其主要区别在于 isl 调度算法允许计算中负系数的存在, 并且 isl 调度算法里的循环合并算法在保证程序并行性方面做了更好的改进.

调度变换过程可以理解为利用线性整数规划实现多维空间几何的变基过程. 我们令 m_s 表示语句 S 所在的循环层数, I_s 为 S 的迭代空间, m_s 维列向量 $\vec{i_S}$ 表示语句实例 S_i 的字典序, 其中 $\vec{i_S} \in I_S$. 语句的仿射变换过程可以用仿射超平面表示, 一个仿射超平面是 n 维空间映射到 n–1 维仿射子空间的一维仿射变换函数[34]. 语句实例 S_i 的一维仿射变换函数为

$$\phi_S\left(\vec{i_S}\right)=\left(c_1^S,c_2^S,...,c_{m_S}^S\right)\cdot\vec{i_S}+c_0^S=\vec{h_S}\cdot\vec{i_S}+c_0^S,\ c_1^S,c_2^S,...,c_{m_S}^S\in\mathbb{Z} \tag{9}$$

其中, $\vec{h_S}=\left(c_1^S,c_2^S,...,c_{m_S}^S\right)$ 是一个仿射超平面的法向量. c_0^S 是一个一维整数空间上的常数, 用来表示仿射变换过程中的偏移过程. 将上述一维仿射变换函数扩展成 n 维形式为

$$\Phi_S\left(\vec{i_S}\right)=\left(\phi_S^1\left(\vec{i_S}\right),\phi_S^2\left(\vec{i_S}\right),...,\phi_S^n\left(\vec{i_S}\right)\right)^T=\begin{pmatrix}c_{11}^S & c_{12}^S & \cdots & c_{1,m_S}^S\\ c_{21}^S & c_{22}^S & \cdots & c_{2,m_S}^S\\ \vdots & \vdots & \ddots & \vdots\\ c_{n1}^S & c_{n2}^S & \cdots & c_{n,m_S}^S\end{pmatrix}\cdot\vec{i_S}+\begin{pmatrix}c_{10}^S\\ c_{20}^S\\ \vdots\\ c_{n0}^S\end{pmatrix}=M\cdot\vec{i_S}+C \tag{10}$$

利用 isl 调度算法进行调度变换即为求解公式(10)中系数矩阵 M 以及向量 C 的过程.

为便于理解, 我们仍以矩阵乘计算为例, 公式(1)表示的是图 5 中程序的原始调度. 经 isl 调度算法计算后, 系数矩阵 M 以及常数向量 C 的结果为

$$M=\begin{pmatrix}0&1&0&0\\0&0&1&0\\1&0&0&0\\0&0&0&1\end{pmatrix},\ C=\begin{pmatrix}0&0&0&0\end{pmatrix}^{\mathrm{T}} \tag{11}$$

将 M 和 C 带入公式(10)中:

$$\Phi_{S_0}\begin{pmatrix}0\\i\\j\\0\end{pmatrix}=\begin{pmatrix}0&1&0&0\\0&0&1&0\\1&0&0&0\\0&0&0&1\end{pmatrix}\cdot\begin{pmatrix}0\\i\\j\\0\end{pmatrix}+\begin{pmatrix}0\\0\\0\\0\end{pmatrix}=\begin{pmatrix}i\\j\\0\\0\end{pmatrix} \tag{12}$$

以及

$$\Phi_{S_1}\begin{pmatrix}1\\i\\j\\k\end{pmatrix}=\begin{pmatrix}0&1&0&0\\0&0&1&0\\1&0&0&0\\0&0&0&1\end{pmatrix}\cdot\begin{pmatrix}1\\i\\j\\k\end{pmatrix}+\begin{pmatrix}0\\0\\0\\0\end{pmatrix}=\begin{pmatrix}i\\j\\1\\k\end{pmatrix} \tag{13}$$

由此可以得到调度变换后的调度如公式(2)所示, 即对原始矩阵乘计算核心段实现了循环合并优化.

4 硬件映射

在利用多面体模型调度算法充分挖掘程序并行性的基础上, 为使程序在执行时能够充分发挥硬件架构特点, 并行代码自动生成系统需要将程序根据申威异构架构进行映射. 前文中提到, 由于申威异构架构在核组间利用 MPI 机制实现通信, 与其他分布式存储结构无异, 在申威异构并行程序编写中并非难点, 因此在这里我们将针对申威异构架构特有的核组内主从核异构并行进行讨论, 将关注点集中在 Athread 编程模型中的线程级并行.

4.1 异构划分

经过预处理阶段完成对程序的抽象分析以及并行性发掘后, 我们将得到一个包含若干个 *band* 节点的调度树, 其中的 *band* 节点可以理解为循环嵌套.

由于申威异构架构程序执行时从核线程组开启的开销较大, 在映射过程中应尽可能地保证程序并行粒度最大化, 以减少从核线程组开启次数. 因此, 我们将寻找调度树中指向可并行执行循环的最外层 *band* 节点,

并在该 *band* 节点前添加一个 *mark* 节点, 标记该 *band* 节点对应为一个从核 *kernel* 函数, 如图 9 中的 *mark*:*kernel* 节点. 调度树中该 *mark* 节点前的部分保持在主核上串行执行, 同时, 将以该 *mark* 节点为根节点的子树映射到从核阵列上执行, 即每一个上述符合条件的 *band* 节点映射成一个从核 *kernel* 函数, 并部署到从核阵列上执行.

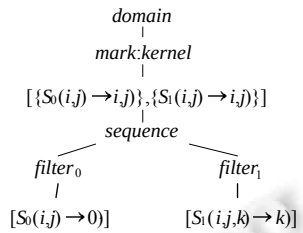


图 9 异构划分后矩阵乘计算调度树表示

图 9 所示的经变换后的矩阵乘计算所对应的调度树中, 符合条件的 *band* 节点其位置在调度树的起始位置, 因此该矩阵乘计算程序中所有计算过程都将部署到从核上运行, 且仅需创建一个从核 *kernel* 函数.

4.2 循环分块

正如第 1.1 节中所介绍的, 申威异构架构的一个从核阵列包含 64 个从核, 以 8×8 结构排列. 在一个核组内部利用从核阵列进行加速计算时, 需通过 Athread 编程模型开启从核线程组将计算核心部署到从核上运行. 由于从核阵列的特殊结构, 在进行从核加速计算时, 可以由用户通过编译选项指定一维或二维并行. 在第 4.1 节中提到, 我们将可并行执行的最外层 *band* 节点映射成一个从核 *kernel* 函数, 该过程可视为把一个可并行执行的循环嵌套部署到从核阵列上执行. 此时, 我们需要判断程序是否满足实现二维并行的条件: 当该 *band* 节点存在最外层的两个以上维度可以并行执行且循环交换操作合法时, 可根据用户需求实现一维或二维并行, 否则该程序仅可以实现一维并行.

接下来, 需要根据并行维度对满足上述条件的 *band* 节点进行循环分块/分段操作, 分别对应二维/一维并行. 众所周知, 由于循环分段可以看作循环分块在一个维度上的特殊情况, 且循环分段操作始终合法, 所以在这里我们将只针对循环分块(二维并行)进行讨论.

在面向二维并行进行循环分块操作时, 首先要确定每个维度上的分块大小 *size*[*i*], 由对应维度上的循环迭代次数 *iters*[*i*]以及线程组在该维度上开启线程数 *num_thread*[*i*]所决定:

$$size[i]=\left\lceil \frac{iters[i]}{num\_thread[i]} \right\rceil,\ i\in\{0,1\} \tag{14}$$

在编译过程中, 参数 *size*[*i*]、*iters*[*i*]以及 *num_thread*[*i*]均为常量.

图 7(b)所示为经过循环合并优化后矩阵乘计算的调度树表示, 显然, 其中符合条件的 *band* 节点为

$$\left[\left\{S_0(i,j)\to(i,j)\right\},\left\{S_1(i,j)\to(i,j)\right\}\right] \tag{15}$$

我们对该 *band* 节点所对应循环嵌套最外层的两个维度循环(即 *i* 和 *j* 层循环)进行循环分块, 为便于表示, 将其 *band* 节点分裂为

$$\left.\begin{array}{l}\left[\left\{S_0(i,j)\to\left(\left\lfloor\frac{i}{size[0]}\right\rfloor,\left\lfloor\frac{j}{size[1]}\right\rfloor\right)\right\},\left\{S_1(i,j)\to\left(\left\lfloor\frac{i}{size[0]}\right\rfloor,\left\lfloor\frac{j}{size[1]}\right\rfloor\right)\right\}\right],\\ \left[\left\{S_0(i,j)\to(i\bmod size[0],j\bmod size[1])\right\},\left\{S_1(i,j)\to(i\bmod size[0],j\bmod size[1])\right\}\right]\end{array}\right\} \tag{16}$$

为充分发挥申威异构架构的硬件结构性能, 在并行计算时, 应尽可能开启从核阵列中全部 64 个从核进行加速计算. 因此, 并行代码自动生成系统默认通过 Athread 编程模型开启 64 个线程并将线程一一部署到从核上运行. 所以, 在一维并行时, *num_thread*[0]默认设定为 64; 二维并行时, 默认设定 *num_thread*[0]=*num_thread*[1]=8. 当然, 这里也可以根据程序的不同需求修改为其他正整数, 限制条件为

$$num\_thread[0]=1\wedge num\_thread[1]\leqslant 8\cup num\_thread[0]\leqslant 8\wedge num\_thread[1]=8 \tag{17}$$

矩阵乘计算为进行二维并行而实现循环分块后的调度树表示如图 10 所示.

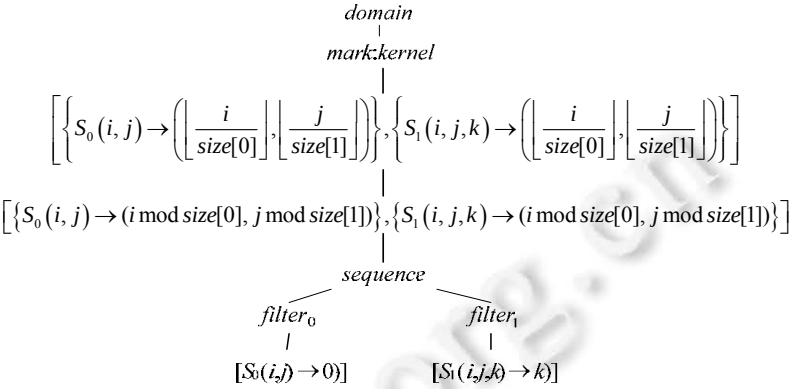


图 10 循环分块后调度树表示

4.3 线程映射

在一维并行条件下, 循环分块后, *kernel* 函数中可并行执行的最外层循环迭代次数为 *num_thread*[0], 与从核线程组开启的线程数量相对应. 此时, 我们将最外层循环的每一次循环迭代映射到Athread编程模型的一个线程上执行, 并使用变量 *tid*(0≤*tid*≤63)作为线程标识. 以面向一维并行进行循环分段后的矩阵乘计算为例, *tid* 与循环层(*ii*,*i*,*j*,*k*)的对应关系为

$$(tid)\to\{(ii,i,j,k)|ii=tid\} \tag{18}$$

其中, 循环层 *ii* 为通过循环分段得到的新循环层.

在二维并行条件下, 我们将最外层两层循环的每一次循环迭代映射到 Athread 编程模型两个维度的线程上一一执行, 并使用变量 *rid* 和 *cid* 分别作为每个线程的行标识和列标识. 值得注意的是, 在利用申威处理器从核阵列加速计算时, 必须按照从 0 号至 63 号的顺序使用从核, 例如开启 16 线程时, 只能使用序号为 0–15 的从核. 显然, 在二维并行时变量 *rid* 和 *cid* 须满足条件: 1≤*rid*≤7∧*cid*=8. 以面向二维并行进行循环分块后的矩阵乘计算为例, *rid* 和 *cid* 与循环层(*ii*,*jj*,*i*,*j*,*k*)的对应关系为

$$(rid,cid)\to\{(ii,jj,i,j,k)|ii=rid,jj=cid\} \tag{19}$$

其中, 循环层 *ii*, *jj* 为通过循环分块得到的新循环层.

此时, 公式(16)将变换为

$$\left.\begin{array}{l}\left[\{S_0(i,j)\to(rid,cid)\},\{S_1(i,j)\to(rid,cid)\}\right],\\ \left[\{S_0(i,j)\to(i \bmod size[0], j \bmod size[1])\},\{S_1(i,j)\to(i \bmod size[0], j \bmod size[1])\}\right]\end{array}\right\} \tag{20}$$

矩阵乘计算在进行线程映射后的调度树表示如图 11 所示.

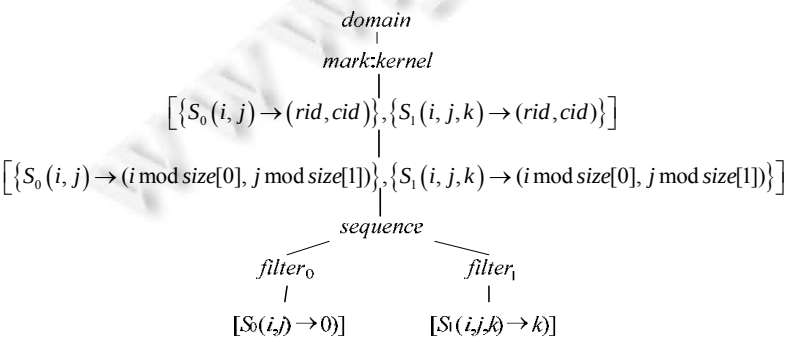


图 11 线程映射后调度树表示

5 内存管理

通过硬件映射模块, 我们将程序并行区中能够并行执行的计算核心部署到申威异构架构的从核阵列上执行, 从核的计算部件得以充分利用. 接下来, 如何充分利用申威异构架构的存储结构, 使得程序执行时拥有更高的访存效率, 是提升并行程序执行效率的关键.

第 1.1 节中介绍到, 申威异构架构有着较为复杂的存储结构, 其中从核阵列在进行加速计算时可以访问从核上独有的 LDM, 同时也可以通过 *globalld/st* 指令直接访问主存. 但对于科学计算程序中大部分情况来说, 从核在读取主存数据时先通过 DMA 命令将数据传输至从核 LDM, 然后访问 LDM 进行数据读取的效率要远高于从核直接从主存读取数据, 存储过程亦然.

因此, 我们在内存管理模块的实现中, 采用了较为简单但高效的从核数据访问模式, 即通过 DMA 命令实现从核 LDM 与主存之间的数据传输, 从核在计算过程中直接访问 LDM 即可. 系统自动生成的从核 *kernel* 程序一般按照"*copyin*→核心计算→*copyout*"的流程执行.

5.1 DMA传输命令

Athread 编程模型中 DMA 读取命令的语法如下.

athread_get(*dma_mode mode*,*void *src*,*void *dst*,*int len*,*void *reply*,*char mask*,*int stride*,*int bsize*)

其中, *dma_mode* 指定 DMA 的传输命令模式, 一般采用 PE_MODE (即点对点传输方式), *src* 为传输主存源地址, *dst* 为从核 LDM 目标地址, *len* 表示以字节为单位的传输数据量, *mask* 为广播模式掩码, *stride* 和 *bsize* 分别表示跨步访存模式下跨幅与跨步块大小. 反之, 存储命令 *athread_put* 的语法与读取命令基本一致, 区别仅为 *src* 用于表示从核 LDM 源地址, *dst* 表示主存目标地址. 因此, 在 DMA 命令自动生成的过程中, 需要通过对语句实例访存关系的分析, 确定传输过程的源地址、目标地址、传输数据量以及跨步信息.

值得注意的是, 在 DMA 命令的参数中存在回答字变量 *reply*, 这是因为申威异构架构的 DMA 传输支持阻塞通信与非阻塞通信. 相较于阻塞通信, 非阻塞通信需要通过对于回答字变量 *reply* 的判断实现对传输过程已完成的确认, 即当非阻塞 DMA 通信的传输过程完成时, *reply* 的值自增 1. 非阻塞通信功能的支持使得申威异构架构的编程难问题显得更为突出, 但该功能的支持为异构并行程序后续的进一步优化, 例如通信与计算过程的隐藏等提供了基础, 因此, 在并行代码自动生成系统中, 我们将非阻塞 DMA 通信作为默认通信方式予以支持.

5.2 数据映射

上文中提到, 在利用申威异构架构的从核阵列进行加速计算时, 应尽可能地将数据加载到从核 LDM 空间上以提升访存效率. 因此, 我们需要首先对哪些数据需要映射到从核 LDM 空间进行分析. 数据的选择需遵循以下几个原则.

(1) 数组中所有被从核计算过程使用且不在从核程序中作为输入的元素, 需要映射到从核 LDM 空间, 并在计算过程开始前由 *copyin* 过程从主存传输至 LDM;

(2) 数组中所有在从核计算过程被更新的元素, 需要映射到从核 LDM 空间, 并在计算过程完成后由 *copyout* 过程从 LDM 传输至主存;

(3) 标量按照零维数组处理;

(4) 只读标量作为参数传入从核 *kernel* 函数中.

按照上述原则, 我们以矩阵乘计算为例, 其 *copyin* 和 *copyout* 过程所传输的数据为

$$copyin := \{A(i,k): 0 \leqslant i < M \wedge 0 \leqslant k < K; B(k,j): 0 \leqslant k < K \wedge 0 \leqslant j < N\} \tag{21}$$

$$copyout := \{C(i,j): 0 \leqslant i < M \wedge 0 \leqslant j < N\} \tag{22}$$

这里 *copyin* 和 *copyout* 集合中数组 *A*、*B*、*C* 均位于主存, 其约束所表示的范围也是整个 *kernel* 函数所访问的. 而根据申威异构架构的存储结构特点, 每个从核可以分别开启 DMA 通道以实现 LDM 与主存之间的数

据传输. 因此, 接下来需要按照计算过程划分以及线程映射将 *copyin* 和 *copyout* 集合中的数据映射到 LDM 空间.

第 1.1 节中介绍到, 从核上 LDM 空间极其有限, 以 SW26010 处理器为例, 其每个从核可使用的 LDM 空间仅为 64 KB. 为同时满足大数据量计算程序的需求以及申威架构存储结构的限制, 我们在进行数据映射前需先将数据进行分块, 划分方式如图 12 所示. 在矩阵乘计算中, 数组 *A*、*B*、*C* 均需同时存储在从核 LDM 上, 我们假设每个从核上的 3 个计算维度分别为 *x*、*y*、*z*, 以前缀"*slave_*"表示主存数组在 LDM 上所对应的从核数组, 则在每个从核上 *slave_A* 大小为 $x\times z$, *slave_B* 大小为 $y\times z$, *slave_C* 大小为 $x\times y$, 如图中蓝色部分. 从核数组大小须满足条件:

$$\left(x\times z+y\times z+x\times y\right)\times sizeof\left(type\right)<mem_{LDM} \tag{23}$$

此时, 可以根据各个并行维度上的线程数计算出整个从核阵列可同时存储的数据量 $\hat{X}\times\hat{Y}+\hat{X}\times\hat{Z}+\hat{Y}\times\hat{Z}$, 如图中灰色部分. 在整个矩阵乘计算过程中, 需先将数据按照 3 个维度大小分别为 $\hat{X}$、$\hat{Y}$、$\hat{Z}$ 进行划分, 以满足从核 LDM 空间限制.

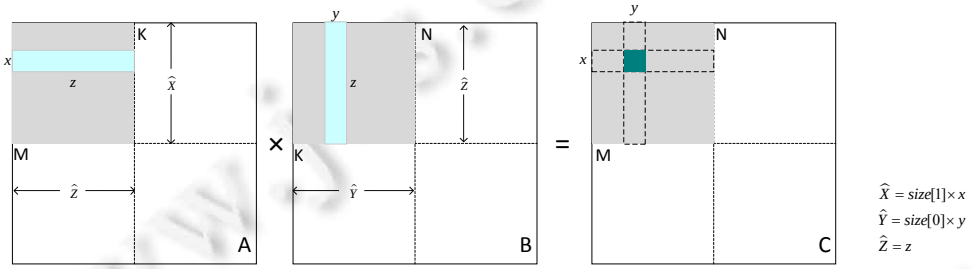


图 12 矩阵乘计算数据划分示意图

为确定生成 DMA 命令所需传输过程的源地址、目标地址和传输数据量等信息, 我们将从核 LDM 数组地址映射至主存数组地址. 以数组 *C* 为例, 其从核 LDM 数组集合表示为

$$\left\{slave\_C\left(c_0,c_1\right)\middle|0\leqslant c_0<x,0\leqslant c_1<y\right\} \tag{24}$$

根据线程映射, 在一个数据分块内部(图 12 中灰色部分), 从核数组到主存的映射关系为

$$\left(rid,cid,i,j,k\right)\rightarrow\left\{slave\_C\left(c_0,c_1\right)\rightarrow C\left(c_2,c_3\right)\middle|c_2=c_0+rid\times x,c_3=c_1+cid\times y\right\} \tag{25}$$

每个从核计算结束时需要通过 DMA 写回主存的数组 *C* 范围可表示为

$$\left(rid,cid,i,j,k\right)\rightarrow\left\{C\left(c_2,c_3\right)\middle|rid\times x\leqslant c_2<x+rid\times x,cid\times y\leqslant c_3<y+cid\times y\right\} \tag{26}$$

在此基础上, 考虑数据分块步骤的影响, 每个从核 LDM 上数组 *slave_C* 对应主存的地址范围为

$$\begin{aligned}&\left(X,Y,Z,rid,cid,i,j,k\right)\rightarrow\\&\left\{C\left(c_2,c_3\right)\middle|\hat{X}\times X+rid\times x\leqslant c_2<\hat{X}\times X+x+rid\times x,\hat{Y}\times Y+cid\times y\leqslant c_3<\hat{Y}\times Y+y+cid\times y\right\}\end{aligned} \tag{27}$$

其中, *X*、*Y*、*Z* 分别表示为满足 LDM 空间限制所做循环分块而产生的循环层.

由图 12 可以看出, 在进行矩阵乘计算时, 每个线程划分到的数组 *A*、*B*、*C* 中数据利用图形展示时均为规则的矩形. 而当遇到传输的数据离散或为不规则多面体图形时, 考虑到 DMA 传输连续数据效率更高这一特性, 我们采取向上近似的策略, 将不规则或离散的数据在每一维度上均向上近似至一个规则的多面体图形, 如二维的矩形, 由此保证数据传输效率. 另外, 在系统实现过程中, 为便于实现进一步优化, 我们将多维数组进行了线性化处理.

5.3 DMA节点生成

根据从核数组与主存之间的映射关系, 我们能够确定哪些数据需要通过 DMA 实现数据传输. 接下来, 就需要通过调度树中的 *extension* 节点实现 DMA 数据传输语句的添加, 并利用 *sequence* 节点确保程序按照

“*copyin*→核心计算→*copyout*”的顺序执行, 以保证程序执行的正确性.

除数据传输的范围与映射关系外, DMA 数据传输语句在 *kernel* 函数中的位置亟需确定. 在并行代码自动生成系统中, 我们采用的原则是首先在数据映射关系中找到影响数据范围的最内层循环, 将 DMA 节点添加在表示该循环的 *band* 节点的子树中, 并尽可能靠近该 *band* 节点. 将 DMA 节点置于符合上述条件的循环内首先保证了循环索引变量发生改变时, DMA 数据传输范围的正确性, 而让 DMA 节点靠近该循环是为了提升数据局部性, 尽可能增大数据的复用率, 减少 DMA 传输次数, 从而提升程序执行效率.

在矩阵乘计算中, 由公式(27)可以分析出影响数组 *C* 范围的循环维度为(*X*,*Y*,*Z*,*rid*,*cid*), 其最内层循环为映射到 *cid* 的循环层, 即在 *i* 层循环外部. 数组 *C* 对应的 DMA 节点生成位置如图 13 所示. 数组 *A*、*B* 的分析过程与 *C* 相同, 不再赘述.

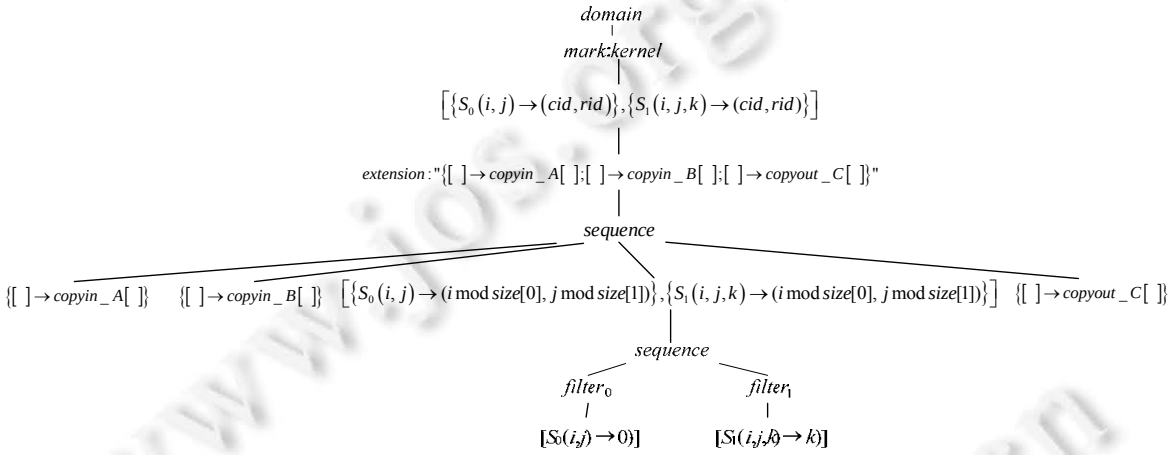


图 13 DMA 节点添加后调度树表示

5.4 同步节点生成

第 5.1 节中介绍到, 申威异构架构的 DMA 传输支持阻塞通信与非阻塞通信, 我们在并行代码自动生成系统中选取非阻塞通信作为默认通信方式. 根据 Athread 编程模型, 非阻塞通信除需要 athread_get/athread_put 传输语句开启 DMA 传输过程外, 还需要通过对 DMA 中 *reply* 回答字参数的判断确定通信过程是否完成. 因此, 我们在完成调度树中添加 DMA 节点的同时, 还需要添加对应 *reply* 回答字参数判断功能的同步节点.

为了给后续实现计算与通信隐藏及双缓冲等优化方法提供基础, 我们要构建同步节点与对应 DMA 节点之间的映射关系, 使同步节点与 DMA 节点一一对应, 即针对每一个需要传输的数据集合生成其独有的 *reply* 回答字参数. 该映射关系可表示为

$$\left\{reply\_A(\,)\to read\_A\left(slave\_A\left(a_0,a_1\right)\to A\left(a_2,a_3\right)\right)\right\} \tag{28}$$

确定映射关系后, 在调度树中对应 DMA 节点的同一位置, 利用 *extension* 节点添加同步语句节点.

6 代码生成

代码生成模块将经过硬件映射及内存管理后的调度树作为输入, 利用多面体模型中的多面体扫描技术生成与调度树相对应的 AST, 最后, 以 AST 为基础, 参照 Athread 编程模型分别生成对应的主核代码及从核代码. 多面体扫描技术, 其任务就是扫描调度树中迭代空间的每一个点, 并按照调度所表示的信息生成对应的 AST.

在该模块中, 我们选择“调度树→AST→异构并行代码”的代码生成流程, 而非由调度树直接生成异构并行代码, 其原因在于希望在调度树阶段尽可能实现与硬件体系结构无关的通用性, 在调度树中保留优化阶段的完整信息, 将与硬件体系结构强相关的信息或语句节点类型放置在 AST 阶段处理. 例如, 在调度树阶段,

DMA 节点中存储数据传输的范围和数据类型等完整信息, 其信息存储的方式及完整度与面向 CUDA 等其他结构的代码生成过程无异, 但关于面向申威异构架构特有的 DMA 语句生成方式等信息将在 AST 中体现, 具体实现过程将在第 6.1 节中具体阐述.

6.1 AST生成

由调度树生成 AST 的过程可利用 isl 库实现. 在 isl 库中, AST 节点的类型可分为: ast_node_for 类型节点表示循环嵌套, ast_node_if 类型节点表示条件判断语句, ast_node_block 类型节点表示基本块, ast_node_mark 节点与调度树中的 mark 节点相对应, ast_node_user 节点可表示计算或函数调用等语句. 在面向申威异构架构的并行代码自动生成系统中, 上述语句所对应 AST 节点利用 isl 库实现的生成过程可见文献[16].

相较于其他平台的编程模型, 申威架构 Athread 编程模型中 DMA 传输语句更为复杂, 上述 AST 的节点类型并不能对应 DMA 传输语句. 因此, 在 AST 生成过程中, 我们添加 ast_node_dma 节点类型与调度树中的 DMA 节点和 Athread 编程模型中的 DMA 传输语句相对应. 第 5.1 节中介绍到, 在 DMA 语句生成过程中, 需要确定传输过程的源地址、目标地址、传输数据量以及跨步等信息. 此时, 我们将对调度树中 DMA 节点所存储的数据取值范围等信息进行分析, 并将上述信息确定后存储在 AST 的 ast_node_dma 类型节点中.

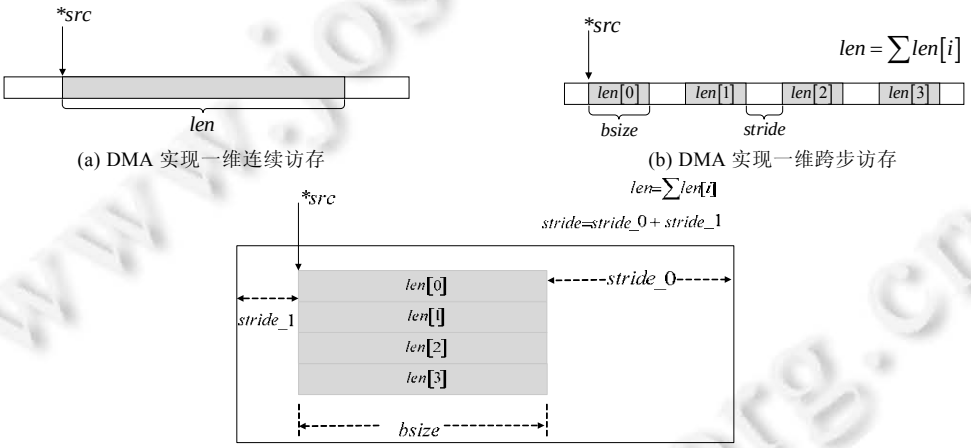


图 14 DMA 传输过程示意图

如公式(27)所示, 调度树中 DMA 节点内所表示的是 DMA 传输过程的地址范围及其映射关系, 此时, 一次 DMA 传输过程就存在多维数据传输和跨步传输等复杂情况. 通过对 Athread 编程模型的研究, 我们总结出可以利用一条 DMA 传输语句实现以下 3 种情况: (1) 一维数组连续访存; (2) 一维数组跨步访存; (3) 二维数组连续访存, 如图 14 所示.

图 14 中所示内存空间均为主存空间, 我们针对每一个从核数组均声明了其独有的 LDM 空间, 因此 DMA 数据传输过程的从核 LDM 内存地址均为数组起始地址. 一维连续访存仅需确定数组的主存访问起始地址和传输数据量, 二维连续访存可以将二维数组看作一维数组, 即为一维跨步访存. 针对一维跨步访存情况, 除需要确定其主存访问起始地址, 还需要确定其跨步信息. *bsize* 表示一个跨步块的数据量大小, *stride* 表示跨步块之间的距离(跨幅), *len* 表示 DMA 传输总数据量. 跨步访存的过程可以理解为每隔 *stride* 距离传输 *bsize* 大小的数据块, 当传输总数据量达到 *len* 时, 当前 DMA 传输过程完成.

除上述 3 种情况外, 我们需要借助 for 循环来实现更为复杂的数据传输过程. 以三维数组连续访存为例, 其优先存储的两个维度数据传输可以利用一条 DMA 传输语句实现, 第 3 个维度也就是最外层维度的实现则需要利用 for 循环多次开启 DMA 传输过程. 因此, 在 DMA 节点对应的 AST 生成过程中, 我们首先要通过 DMA 传输的地址范围以及约束条件确定 DMA 传输的类型, 从而生成对应的 ast_node_for 类型节点与 ast_node_dma 类型节点.

在内存管理模块中添加的与 DMA 节点成对的同步节点对应 Athread 编程模型中非阻塞通信的回答字判断语句, 在 AST 阶段可以通过 ast_node_user 类型节点表示. 在 AST 生成时, 每一条回答字判断语句中 *reply* 回答字参数的值需要和与其对应的 DMA 节点所表示的 DMA 传输过程开启次数(即外层 for 循环迭代次数)一致. 此时, 就需要根据 DMA 传输的类型计算出 *reply* 回答字参数的值, 并将其存储在其对应的 ast_node_user 类型节点中.

6.2 并行代码生成

在完成AST生成后, 系统将参照Athread编程模型, 进行申威异构并行代码的生成. 此过程将异构划分所得的主核代码与从核 *kernel* 函数代码分别输出至主核代码文件和从核代码文件, 再由基础编译器进行编译链接从而生成可执行文件. 由 AST 生成 Athread 并行代码的过程较为简单, 只需将 AST 中存储的程序信息按照 Athread 编程模型一一对应即可.

```
void kernel0( unsigned long* arg0)
{
  int rid=athread_get_id(−1)/8;
  int cid=athread_get_id(−1)%8;
  double slave_A[64][32], slave_B[32][64], slave_C[64][64];
  volatile unsigned long reply_A, reply_B, reply_C;
  double * A=arg0[0];
  double * B=arg0[1];
  double * C=arg0[2];
  for(int c0=0;c0<K;c0 +=32)
  {
    athread_get(PE_MODE,&A[rid*64*64+c0],&slave_A[0][0],64*32*sizeof(double),&reply_A,0,32*sizeof(double),(M*K-32)*sizeof(double));
    while (reply_A !=1);
    athread_get(PE_MODE,&B[c0*64+cid*64],&slave_B[0][0],32*64*sizeof(double),&reply_B,0,64*sizeof(double),(K*N-64)*sizeof(double));
    while (reply_B !=1);
    for(int c1=0;c1<=63;c1 +=1)
      for(int c2=0;c2<=63;c2 +=1) {
        slave_C[c1][c2]=0;
          for(int c3=0;c3<=31;c3 +=1)
            slave_C[c1][c2] +=slave_A[c1][c3]*slave_B[c3][c2];
      }
    athread_put(PE_MODE,&C[rid*64*64+cid*64],&slave_C[0][0],64*64*sizeof(double),&reply_C,64*sizeof(double),(M*N-64)*sizeof(double)
    while (reply_C !=1);
  }
}
```

图 15 矩阵乘计算从核代码核心段

由并行代码自动生成系统生成的矩阵乘计算从核代码核心段如图 15 所示. 其中, athread_get_id 函数获取当前线程的线程标识 *tid* 用于一维并行, 在二维并行时则利用 athread_get_id 函数与 8 的除法和取模操作分别获得线程的行列标识 *rid* 和 *cid*. 另外, 主核数组在主存的起始地址将统一存储在指针数组中并将该数组的指针作为函数参数传入从核 *kernel* 函数中.

7 实验结果

为验证并行代码自动生成系统的有效性, 我们利用 SW26010 处理器以及 Polybench 测试集[35]对生成异构并行代码的正确性及执行效率进行评估.

7.1 实验设置

在第 4 节中提到, 并行代码自动生成系统主要面向申威异构架构的线程级并行进行代码生成, 因此在实验中我们利用 SW26010 处理器的一个核组(一个主核及一个从核阵列)评估核组内主从核异构代码的执行效率. 其中, 主核拥有两级 Cache 结构, 分别为 32 KB 的 *L*1Cache 和 512 KB 的 *L*2Cache, 从核采用 SPM 结构的 LDM, 其存储空间大小为 64 KB. 在实验过程中, 我们首先将测试用例通过申威架构上的基础编译器 sw5cc 进行编译, 编译选项为-O3-msimd, 其中-msimd 为向量指令集的开关, 将此串行可执行程序提交到 SW26010 处理器运行得到其串行程序执行时间, 以此作为实验的参照. sw5cc 编译器版本号为 5.421-sw-500.

接下来, 将测试用例通过并行代码自动生成系统分别生成其对应的主从核文件, 同样利用 sw5cc 基础编译器进行编译, 在编译时利用编译选项-mhost 和-mslave 区分主从核文件, 利用选项-mhybrid 进行链接得到可执行程序, 进而提交到 SW26010 处理器上运行并记录时间, 求得并行加速比. 为保证测试结果不受基础编译器其他优化方法的影响, 并行版本的编译选项同样为-O3-msimd.

7.2 测试用例

我们采用 Polybench 测试集中线性代数相关的 13 个测试用例来评估系统所生成异构并行代码的性能, 测试用例信息见表 1.

表 1 测试用例介绍

用例	介绍
gemm	通用矩阵乘法($\alpha \times AB+\beta \times C$)
gemver	矩阵-向量乘法及矩阵加法
gesummv	矩阵-标量及向量乘法
symm	对称矩阵乘法
syrk	矩阵秩 2k 更新
syr2k	矩阵秩 k 更新
trmm	三角矩阵乘法
2mm	2 次矩阵乘法($E=AB$; $F=EC$; $G=F+D$)
3mm	3 次矩阵乘法($E=AB$; $F=CD$; $G=EF$)
atax	矩阵转置及矩阵-向量乘法
bicg	稳定双共轭梯度法求解线性方程程序核心
doitgen	多分辨率分析程序核心
mvt	矩阵-向量乘积转置

由表 1 可以看出, 测试用例主要核心为矩阵乘以及矩阵向量乘计算. 这些计算过程作为热点在科学计算以及深度学习等诸多领域的实际应用程序中被频繁调用, 因此, 将这 13 个测试用例进行并行加速对于实际应用程序的执行效率提升有着重要作用. 另外, 在 Polybench 测试集中还有 data mining、solvers、medley 以及 stencils 类别共 17 个测试用例, 其中, datamining、solvers 以及 medley 中测试用例在利用 isl 调度算法进行并行性挖掘时, 只能挖掘出最内层循环的并行性, 并不适合在申威架构上进行并行执行, 因此我们不将其选为测试用例进行实验. 而 stencils 类别的用例在进行并行时, 若要提升并行粒度, 则需要对时间轴进行循环分块, 并进行块间流水并行, 这种分块及并行方法在文献[36,37]中均有介绍. 但文献中的方法仅面向多核 CPU 架构和 GPU 异构架构, 并不适合申威异构架构. 我们利用 stencils 类别中最为典型的 jacobi-2d 用例进行测试, 利用 64 线程仅能获得 9 倍左右的加速比, 这其中影响性能提升的主要原因在于调度算法没有对时间轴进行分块, 导致时间轴只能在主核上执行, 而时间轴的每一次循环迭代都将开启和同步一次从核线程组, 这一过程意味着巨大的时间开销. 因此, 面向申威异构架构针对于 stencil 计算的循环分块以及流水并行加速的相关研究将是我们下一步研究的方向.

在 Polybench 测试集中, 测试数据规模被分为 Mini、Small、Medium、Large 和 Extralarge 这 5 类, 其中, Mini、Small 以及 Medium 数据规模过小, 利用申威架构进行并行计算的实际意义不大, 因此我们选用 Large 及 Extralarge 规模作为本文实验中的测试规模, 见表 2.

在第 5.2 节中提到, 当数据规模大于从核阵列 LDM 空间时, 并行代码自动生成系统首先要对数据进行分块, 此时我们针对大多数情况默认采用每个维度均为 32 的分块大小. 从表 2 中可以看出, 并非所有的测试用例各维度规模均为 32 的整倍数, 因此在进行数据分块时需利用 min 或 max 操作以保证该维度边界的正确划分. 而在 DMA 传输命令生成过程中, 当 DMA 数据传输范围存在 min 或 max 操作时, 系统无法在编译时计算出准确的传输量, 此时, 只能利用“循环+DMA 命令”的方式实现该类型的 DMA 跨步传输. 但该方式会增加 DMA 通道的开启次数, 相较于仅开启一次 DMA 通道的跨步传输, 其代码执行效率将大幅下降. 因此, 如表 2 所示, 我们在进行测试时, 分别增加了 Large 规模和 Extralarge 规模中各维度向上近似为 32 的整倍数两个数据规模, 利用表中四种数据规模分别验证在数据分块规整及一般情况下的并行程序执行效率.

表 2 测试用例规模

用例	LARGE 规模	LARGE 规模向上近似	EXTRALARGE 规模	EXTRALARGE 规模向上近似
gemm	1000×1100×1200	1024×1120×1216	2000×2300×2600	2048×2304×2624
gemver	2 000	2 048	4 000	4 096
gesummv	1 300	1 312	2 800	2 816
symm	1000×1200	1024×1216	2000×2600	2048×2624
syrk	1000×1200	1024×1216	2000×2600	2048×2624
syr2k	1000×1200	1024×1216	2000×2600	2048×2624
trmm	1000×1200	1024×1216	2000×2600	2048×2624
2mm	800×900×1100×1200	800×928×1120×1216	1600×1800×2200×2400	1600×1856×2240×2400
3mm	800×900×1000×1100×1200	800×928×1024×1120×1216	–	
atax	1900×2100	1920×2112	1800×2200	1856×2240
bicg	1900×2100	1920×2112	1800×2200	1856×2240
doitgen	140×150×160	160×160×160	220×250×280	224×256×288
mvt	2 000	2 048	4 000	4 096

在测试中, 由于我们能够利用的计算节点将任务执行时间限制在 1 h 以内, 导致 Extralarge 规模及其向上近似规模中的 3 mm 用例串行时间无法测出, 因此在上述两个规模测试中不再将 3 mm 作为测试结果展示.

7.3 并行程序性能测试

我们在第 4.3 节介绍线程映射过程时提到, 并行代码自动生成系统支持一维和二维并行代码的生成. 同时, 在利用申威架构的一个核组进行加速计算时, 可以由用户指定所需开启的从核线程数. 因此, 我们在实验中对申威异构架构上常见的几种情况展开测试, 即分别利用 16 线程、32 线程以及 64 线程对程序进行并行加速, 同时在每一组线程数中再次进行一维并行和二维并行代码的测试. 我们在实验中将 6 组并行方式分别表示为: [1,16]、[2,8]、[1,32]、[4,8]、[1,64]、[8,8], 其中, [1,16]表示以一维并行方式实现 16 线程并行代码生成, [2,8]表示以二维并行方式实现 16 线程并行代码生成.

第 7.2 节中介绍的 4 种规模测试结果分别如图 16–图 19 所示, 图中纵坐标表示并行程序的加速比, 由于不同测试用例间并行加速比差距很大, 为便于展示, 我们在图中采用以 2 为底的对数坐标轴作为纵坐标. 针对测试用例的四种不同规模, 并行代码自动生成系统所生成的 Athread 并行代码相较于串行程序均能够获得可观的加速比, 在 64 线程规模下, 最高可获得单个测试用例 3 093.09 倍的加速比以及 539.16 倍的平均加速比. 从图中可以看出, 在利用不同线程规模进行加速计算时, 大部分测试用例都能够达到线程数以上的超线性加速比, 这是因为在前文中提到, 为满足 LDM 空间限制, 我们在进行数据映射前, 需先进行数据分块(循环分块), 该过程能够有效提升程序的数据局部性, 从而大幅提升并行程序执行效率, 将并行程序加速比提升至线性加速比之上.

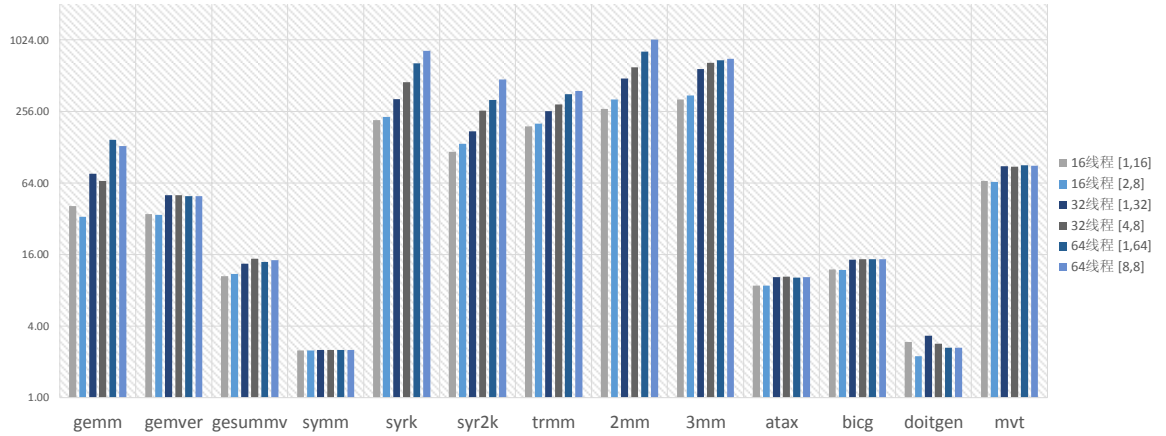


图 16 Large 规模性能测试结果

同时, 在线程映射过程中, 二维并行需基于循环分块实现, 而一维并行需基于循环分段方式实现. 循环分段并不会改变代码的执行顺序, 因此也不能提升程序的数据局部性, 所以从图中可以看出, 在不同线程规模下, 除 *gemm* 用例外, 二维并行普遍能够获得比一维并行更高的加速比. 通过对程序进行进一步分析, 我们发现在 *gemm* 测试用例中, 当采用二维并行方式时, 次外层循环(映射到 *cid*)的循环步进值为 32×8=256, 而在 4 种测试规模中, 该层循环的上界分别为 1 100、1 120、2 300、2 304, 可以看出, 该层循环在最后一次迭代时将存在大量的冗余计算, 但程序必须等待这一次迭代执行结束才能继续执行, 而在该层循环外部依然有最外层循环(映射到 *rid*), 且由线程映射规则可知, 二维并行最外层循环迭代次数为一维并行最外层循环迭代次数的 8 倍, 进一步放大了冗余计算的负加速效果, 因此, 在本次测试中的 4 种规模下, *gemm* 测试用例利用二维并行相较于一维并行将拥有更多的冗余计算, 影响了其加速效果.

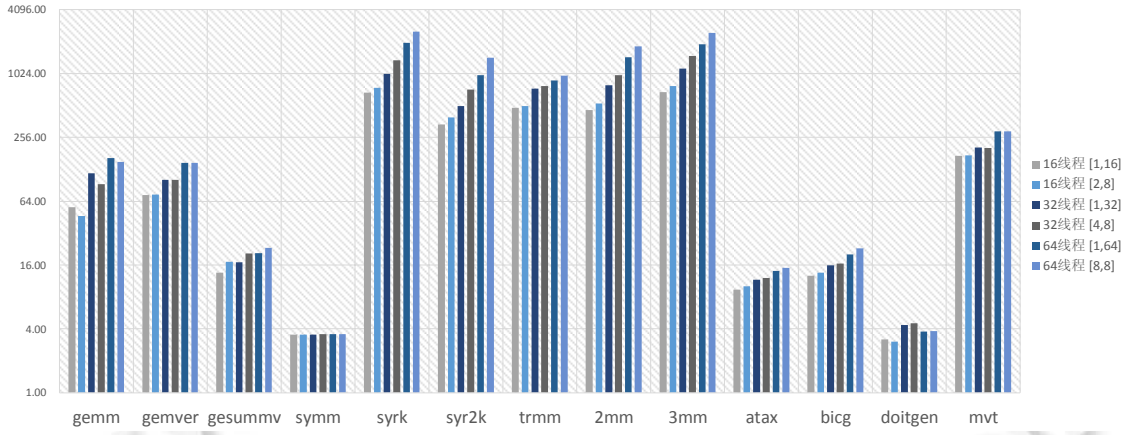


图 17 Large 向上近似规模性能测试结果

从 4 种规模的测试结果中我们可以发现 syrk、syr2k、trmm、2 mm 以及 3 mm 等测试用例相较于其他用例拥有更高的加速比, 这是因为通过循环分块的方式提升了数据局部性, 使得读取到从核 LDM 空间上的数据拥有更高的复用率. 除循环分块对程序性能的影响外, 通过对程序特征与测试结果的分析可以看出, 从核函数中计算过程与数据传输过程的比值(计算访存比)越大, 例如 2 mm、3 mm 用例, 其并行程序从核 LDM 空间的数据利用率越高, 程序利用申威异构架构进行并行加速的效果越好.

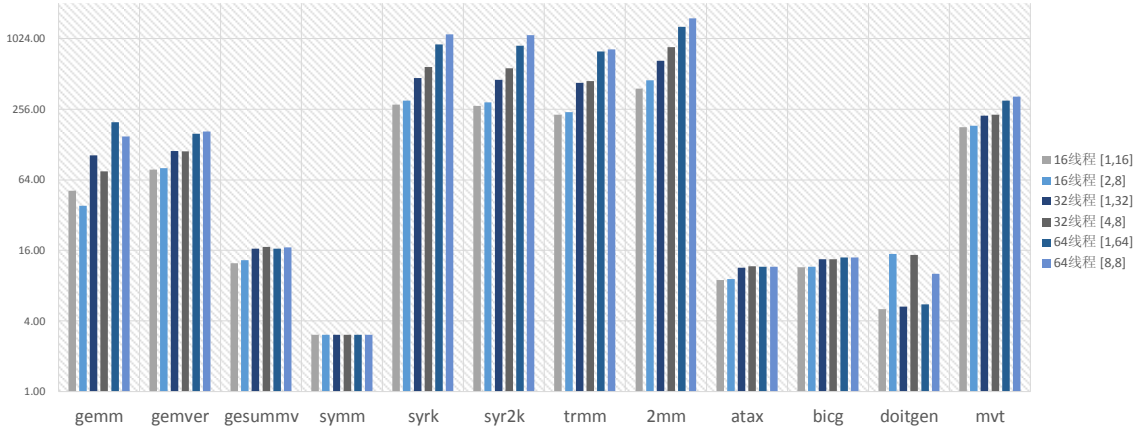


图 18 Extralarge 规模性能测试结果

另外, 当测试用例的串行版本在主核上执行时, 由于主核存储层次中采用了两级 Cache 结构, 其程序的执行效率将很大程度受访存过程 Cache 命中率的限制, 而从核在访问 LDM 上任何地址时的时延均一致, 因此

对于诸如 trmm 这种三角矩阵运算程序, 其 Cache 命中率越低, 主核串行程序执行效率越低, 利用从核并行计算得到的加速效果也就越好.

在测试过程中, 我们求出了 13 个测试用例的 4 种规模在不同线程规模下的平均加速比, 如图 20 所示. 从图中可以看出, 在各个线程条件下, Large 规模和 Extralarge 规模其对应的向上近似版本都能够获得相较于原始版本两倍以上的加速比. 这印证了第 7.1 节中提到的, 由于编译过程中可以确定 DMA 传输量, 从而生成更少的 DMA 通道开启次数, 计算规模中每一个维度均为 32 的整倍数将会为程序带来更高的加速比. 这一点可以为用户程序的编写提供指导, 即在程序编写过程中, 可以通过边缘填充方式(0-padding)添加冗余计算, 从而将规模调整为 32 的整倍数尤其是 256 的整倍数, 以提高程序的执行效率.

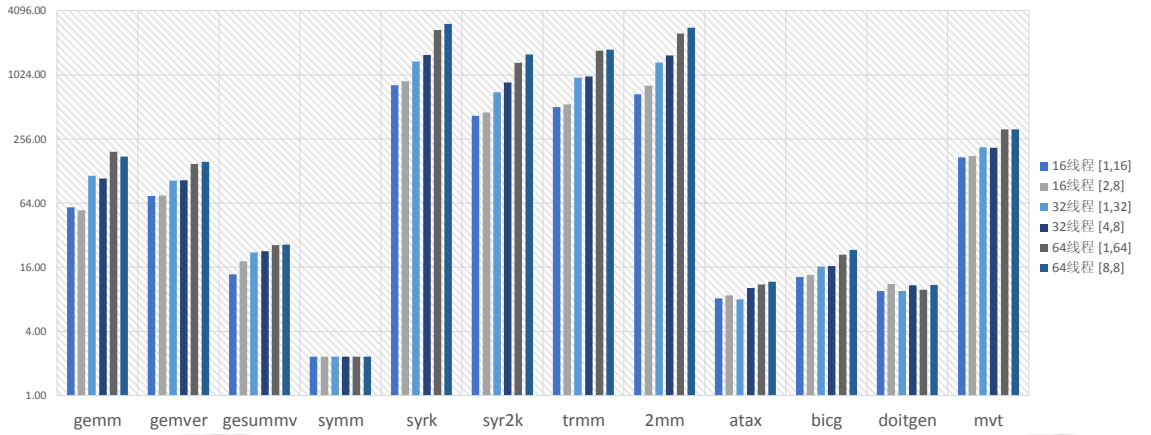


图 19 Extralarge 向上近似规模性能测试结果

在图 20 中通过横向对比可以观察出程序执行效率随着线程数发生改变的变化趋势. 上文中提到, 在同一线程数时, 二维并行普遍能够获得比一维并行更高的加速比. 除此之外, 线程规模增大时, 平均加速比也随之提升. 我们以 Extralarge 向上近似规模为例(图中黄色折线), 分别对比其一维并行和二维并行的 3 种线程规模数据, 当线程数增大一倍时, 可以获得接近两倍的平均加速比, 这表明并行代码自动生成系统可以在不同线程规模下有效利用申威异构架构硬件资源, 从而达到加速计算的目的.

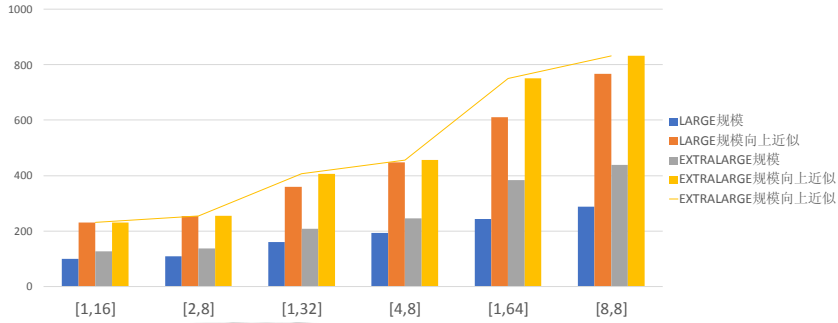


图 20 不同线程规模平均加速比对比

7.4 编译时长测试

除对并行代码自动生成系统所生成的并行代码性能展开测试外, 我们还对代码生成过程中的编译时间进行了测试, 结果见表 3.

表 3 编译时长测试结果(单位: s)

用例	gemm	gemver	gesummv	symm	syrk	syr2k	trmm	2 mm	3 mm	atax	bicg	doitgen	mvt	平均
编译时间	0.73	0.72	0.48	0.51	0.60	0.75	0.53	0.95	1.40	0.49	0.47	0.52	0.40	0.66

测试中 13 个测试用例的平均编译时长为 0.66 s, 其中耗时最长的 3 mm 用例编译时长为 1.40 s, 而用户手工编写申威异构并行程序至少需要以小时为单位计算, 甚至面对复杂程序编写时需要以天为单位计算. 相比之下, 利用本文提出的异构并行代码自动生成系统能够大大降低并行程序开发的成本, 从而有效解决申威异构架构的编程难问题.

8 讨 论

8.1 系统通用性讨论

本文提出的面向申威异构架构的并行代码自动生成系统可以认为是一种面向异构系统通用的代码生成方法, 过程中首先使用 isl 调度算法来尽量增大循环合并的机会, 使程序中更多的中间变量能够在申威架构从核 LDM 空间声明和使用, 从核 LDM 也可以对应为其他异构系统的加速部件局部存储, 如 GPU 的 SharedMemory. 在通过 isl 调度算法确定调度后, 再进行硬件映射可以达到尽量减少主从核之间数据通信的目的, 这也是本文主要完成的研究内容, 即对合并后的循环进行分块然后再进行硬件部署, 循环分块后的外层循环被映射到从核阵列上. 在这个过程中, 调度和硬件映射可认为是完全解耦的.

当然, 在硬件映射之后, 循环分块后内层循环还可以进行进一步地调度. 我们可以利用 Feautrier 调度算法来开发内层循环的并行性[38,39], 以期提高从核上向量化的机会, 此时, 内层循环将会被重新调度, 并实现面向硬件向量指令的映射. 在这一过程中, 内层循环的调度和硬件映射依然可以认为是完全解耦的. 这一部分工作集中在课题组前期研究工作中, 因此没有在本文中体现. 综合来看, 代码生成及编译优化过程中调度和硬件映射交替出现, 但每个阶段内调度和硬件映射均为完全解耦的. 因此, 可以认为本文提出的代码自动生成方法在异构系统内具有一定的通用性.

由于本文从通用性角度出发设计实现面向申威异构架构的并行代码自动生成方法, 因此对于相关领域特定程序的优化效果与平台手工库实现的尚有一定差距, 主要在于程序手工实现可以针对特定程序实现寄存器通信、计算与通信隐藏、双缓冲优化以及指令重排等. 以通用矩阵乘计算程序(GEMM)为例, 本文实验中 Polybench 测试集 *gemm* 测试用例能够达到峰值性能的 3%左右, 而手工 BLAS 库中实现的 GEMM 可以达到峰值性能的 95%[40]. 当然, 这种性能上巨大的差异在通用工具与专用优化间存在, 尤其是在通信开销更大的异构平台上是可以接受的. 例如在利用 MLIR 面向多核 CPU 进行 GEMM 优化时能够获得峰值性能的 25%[41], 可以看出, 利用多面体模型通用编译器(如 Pluto)进行矩阵乘优化虽然比商用编译器(如 GCC、LLVM)可以得到更好的效果, 但是和手工库的性能比较相差深远, 这是因为这些专用优化库可以多利用程序结构特征的优化, 但要做到通用就无法实现这样的效果.

不过, 将通用工具针对 GEMM 优化进行定制化是可行的, 本文的工作正是为此提供了基础. 在作者的同期工作中, 完成了面向申威异构架构的 GEMM 并行程序自动优化, 过程中自动实现了 DMA 优化、片上通信优化、双缓冲优化以及手工 kernel 核心调度等优化手段, 其性能最终达到了峰值性能的 90.14%[42]. 然而本文要解决的是面向申威异构架构上高性能计算程序通用编程性问题, 是针对特定领域应用程序实现自动优化的基础, 因此没有将针对 GEMM 专用的自动优化一并讨论. 此外, 我们还计划专门针对申威异构架构实现 Stencil 计算的领域特定优化.

8.2 与其他异构平台比较

面向申威异构架构进行并行代码自动生成工作与 GPU 和昇腾等架构相比, 其需求都在于尽可能增大并行粒度, 即在硬件映射前尽量对循环进行合并. 但与此同时, 面向申威异构架构的并行代码自动生成与 GPU、昇腾等架构存在不同的特点. 首先, 与 GPU 相比, 申威异构架构在核组内仅支持一级并行(线程级并行), 而 GPU 可以支持两级并行(block,thread), 与此同时, 申威异构架构在加速部件端的存储结构仅有从核的 LDM, 而 GPU 则包含 Global Memory、Shared Memory 和 Private Memory 这 3 种存储结构, 这些都将导致在面向两种异构架构进行硬件映射时的循环分块策略有所不同, 例如循环分块的层数不同.

昇腾架构主要面向深度学习应用, 与其相比, 申威异构架构主要面向高性能计算, 应用领域的不同将直接导致其存储结构和通信方式的不同. 昇腾的存储结构具有多级、多向的复杂特征, 数据流管理更复杂, 需要在数据搬移到芯片后将相应的计算部署到特定的处理功能部件上; 而申威结构上更类似传统的金字塔型结构, 所有从核地位相同, 数据流管理比昇腾更简单. 昇腾芯片在外层进行映射时, 由于数据通信开销较大, 可以为了充分利用存储结构牺牲程序并行性, 即循环可以尽量合并, 外层只需要一层并行, 在内层可以通过重新调度恢复更多层的并行性. 但申威上采用 8×8 的从核排列, 只有在尽量保证外层两层循环的并行才能充分利用计算资源, 虽然在外层上循环合并的机会比昇腾小, 但内层循环恢复并行性的开销也就相对较小. 另外, 昇腾的 Cube 计算功能部件需要对矩阵乘进行特定的分块大小设计和匹配, 在计算部署到芯片上后, 需要进行多级循环分块, 但申威只需要依据 LDM 空间的大小对矩阵分块大小进行选择, 在计算部署到芯片上后只需要进行一次循环分块即可.

9 结 论

异构架构相较于同构多核架构能够在带来更多计算资源的同时, 拥有更低的功耗. 但异构架构硬件结构更加复杂, 存储层次更加多样, 从而导致程序员在进行程序优化时需要对硬件结构有更清晰的了解, 降低了程序编写效率. 先进编译器的编译优化及自动代码生成可以帮助程序员降低程序分析难度, 提升程序优化效率. 多面体编译模型是实现程序在特定体系结构上自动并行化的一种有效方法, 其可以将循环变换与硬件映射过程相结合, 同时可以实现数据传输过程优化和对内存空间的有效管理.

为解决申威异构架构的编程门槛高问题, 本文设计实现了一个面向申威异构架构的并行代码自动生成系统. 系统采用“源-源”编译模式, 基于多面体编译模型实现, 并在系统中利用 isl 库实现依赖分析、调度变换以及代码生成模块. 当然系统并不完全依赖于 isl 库, 相应功能在其他多面体模型的库中也可以实现, 所以系统中的实现方法具有一般性. 系统将标记并行区的串行 C 语言程序作为输入, 抽象成多面体模型中调度树中间表示形式, 并依次经过预处理、硬件映射、内存管理以及代码生成模块, 最终生成面向申威异构架构的 Athread 并行程序. 最后, 利用 Polybench 测试集中线性代数相关测试用例进行测试, 结果显示系统生成的异构并行代码在 SW26010 异构众核处理器上利用 64 线程加速时能够取得 539.16 倍的平均加速比, 且代码生成过程中编译平均时长为 0.66 s, 大大降低了异构并行程序开发成本.

目前, 我们在代码自动生成系统中采用的是主从核加速并行方式, 在线程内部可视为数据传输过程与计算过程串行执行, 为取得更高的加速效果, 下一步工作将在异构并行程序中实现异步 DMA 传输过程, 实现计算过程与数据传输过程的相互隐藏. 另外, 系统生成的异构并行程序为多线程程序, 属于粗粒度并行, 下一步我们将基于本文工作实现向量化等细粒度并行, 以进一步提升程序的执行效率. 同时, 在第 7.1 节中提到的面向申威异构架构并针对 stencil 计算的循环分块以及流水并行的相关研究也将作为我们下一步的研究方向.

References:

[1] Vazhkudai SS, de Supinski BR, Bland AS, *et al*. The design, deployment, and evaluation of the CORAL pre-Exascale systems. In: Proc. of the Int'l Conf. for High Performance Computing, Networking, Storage, and Analysis (SC 2018). 2018, Article 52, 1–12.

[2] NVIDIA. NVIDIA Tesla V100 GPU architecture, 2017.

[3] Fu HH, Liao JF, Yang JZ, *et al*. The Sunway Taihu Light supercomputer: System and applications. Science China (Information Sciences), 2016, 59(7): 113−128 (in Chinese with English abstract).

[4] Verdoolaege S, Juega JC, Cohen A, *et al*. Polyheral parallel code generation for CUDA. ACM Trans. on Architecture and Code Optimization, 2013, 9(4): 54:1−54:24.

[5] Feautrier P, Lengauer C. Polyhedron model. Encyclopedia of Parallel Computing. Berlin, Heidelberg: Springer-Verlag, 2011. 1581−1592.

[6] Zheng F, Xu Y, Li HL, *et al*. A homegrown many-core processor architecture for high-performance computing. Science China (Information Sciences), 2015, 45: 523−534 (in Chinese with English abstract).

[7] NSCCWX. Sunway TaihuLight Compiler user guide. 2016. http://www.nsccwx.cn/

[8] Zhao J, Li YY, Zhao RC. “Black magic” of polyhedral compilation. Ruan Jian Xue Bao/Journal of Software, 2018, 29(8): 2371−2396 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5563.htm [doi: 10.13328/j.cnki.jos.005563]

[9] Feautrier P. Dataflow analysis of array and scalar references. Int’l Journal of Parallel Programming, 1991, 20(1): 23−53.

[10] Bondhugula U, Hartono A, Ramanujam J, Sadayappan P. A practical automatic polyhedral parallelizer and locality optimizer. In: Proc. of the 29th ACM SIGPLAN Conf. on Programming Language Design and Implementation. 2008. 101−113.

[11] Bondhugula U, Acharya A, Cohen A. The Pluto+ algorithm: A practical approach for parallelization and locality optimization of affine loop nests. ACM Trans. on Programming Languages and Systems, 2016, 38(3): 12:1−12:32.

[12] Acharya A, Bondhugula U, Cohen A. Polyhedral autotransformation with no integer linear programming. In: Proc. of the 39th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2018). 2018. 529−542.

[13] Kong M, Pouchet LN. Model-driven transformations for multi- and many-core CPUs. In: Proc. of the 40th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2019). 2019. 469−484.

[14] Bastoul C. Code generation in the polyhedral model is easier than you think. In: Proc. of the 13th Int’l Conf. on Parallel Architectures and Compilation Techniques (PACT 2004). 2004. 7−16.

[15] Chen C. Polyhedra scanning revisited. In: Proc. of the 33rd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2012). 2012. 499−508.

[16] Grosser T, Verdoolaege S, Cohen A. Polyhedral AST generation is more than scanning polyhedral. ACM Trans. on Programming Languages and Systems, 2015, 37(4): 12:1−12:50.

[17] Kelly W, Pugh W. A unifying framework for iteration reordering transformations. In: Proc. of the 1st IEEE Int’l Conf. on Algorithms and Architectures for Parallel Processing (ICAPP 1995), 1995.

[18] Girbal S, Vasilache N, Bastoul C, *et al*. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. Int’l Journal of Parallel Programming, 2006, 34(3): 261−317.

[19] Verdoolaege S. Counting affine calculator and applications. In: Proc. of the 1st Int’l Workshop on Polyhedral Compilation Techniques (IMPACT 2011). 2011.

[20] Chelini L, Zinenko O, Grosser T, Corporaal H. Declarative loop tactics for domain-specific optimization. ACM Trans. on Architecture and Code Optimization, 2019, 16(4): 55:1−55:25.

[21] Liu FF, Yang C, Yuan XH, *et al*. General SpMV implementation in many-core domestic Sunway 26010 processor. Ruan Jian Xue Bao/Journal of Software, 2018, 29(12): 3921−3932 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5309.htm [doi: 10.13328/j.cnki.jos.005309]

[22] Xu ZG. Optimizations of scientific kernels on SW26010 many-core processor [MS. Thesis]. Shanghai: Shanghai Jiaotong University, 2018 (in Chinese with English abstract).

[23] Zhu X, Zeng Y, Wei Y, *et al*. An auto code generator for stencil on SW26010. In: Proc. of the 21st IEEE Int’l Conf. on High Performance Computing and Communications; the 17th IEEE Int’l Conf. on Smart City; the 5th IEEE Int’l Conf. on Data Science and Systems (HPCC/SmartCity/DSS). IEEE, 2019.

[24] Li YB, Zhao RC, Han L, *et al*. Parallelizing compilation framework for heterogeneous manycore processors. Ruan Jian Xue Bao/Journal of Software, 2019, 30(4): 981−1001 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5370.htm [doi: 10.13328/j.cnki.jos.005370]

[25] Shirako J, Hayashi A, Sarkar V. Optimized two-level parallelization for GPU accelerators using the polyhedral model. In: Proc. of the 26th Int’l Conf. on Compiler Construction (CC). 2017. 22−33.

[26] Grosser T, Hoefler T. Polly-ACC transparent compilation to heterogeneous hardware. In: Proc. of the 2016 Int’l Conf. on Supercomputing (ICS 2016). 2016. 1−13.

[27] Baghdadi R, Ray J, Romdhane MB, *et al*. Tiramisu: A polyhedral compiler for expressing fast and portable code. In: Proc. of the 2019 IEEE/ACM Int’l Symp. on Code Generation and Optimization (CGO 2019). 2019. 193−205.

[28] Zhao J, Li BJ, Nie W, *et al*. AKG: Automatic kernel generation for neural processing units using polyhedral transformations. In: Proc. of the 42nd ACM SIGPLAN Int’l Conf. on Programming Language Design and Implementation (PLDI 2021). New York: Association for Computing Machinery, 2021. 1233−1248.

[29] Liao H, Tu JJ, Xia J, Zhou XP. DaVinci: A scalable architecture for neural network computing. In: Proc. of the 2019 IEEE Hot Chips 31 Symp. (HCS), 2019. 1−44.

[30] Zhao J, Di P. Optimizing the memory hierarchy by compositing automatic transformations on computations and data. In: Proc. of the 53rd Annual IEEE/ACM Int’l Symp. on Microarchitecture (MICRO). 2020. 427−441.

[31] Verdoolaege S. isl: An integer set library for the polyhedral model. In: Proc. of the ICMS 2010. LNCS 6327, Berlin, Heidelberg: Springer-Verlag, 2010. 299−302.

[32] Verdoolaege S, Grosser T. Polyhedral extraction tool. In: Proc. of the 2nd Int’l Workshop on Polyhedral Compilation Techniques (IMPACT 2012). 2012.

[33] Kennedy K, Allen R. Optimizing Compilers for Modern Architectures: A Dependence-based Approach. San Francisco: Morgan Kaufmann Publishers Inc., 2001.

[34] Bondhugula U. Effective automatic parallelization and locality optimization using the polyhedral model [Ph.D. Thesis]. Ohio State University, 2008.
[35] Polybench [EB/OL]. 2019. http://polybench.source-forge.net
[36] Li YY, Zhao J, Pang JM. Split tiling design and implementation in the polyhedral model. Chinese Journal of Computers, 2020, 43(6): 1038−1051 (in Chinese with English abstract).
[37] Zhao J, Cohen A. Flextended tiles: A flexible extension of overlapped tiles for polyhedral compilation. ACM Trans. on Architecture and Code Optimization, 2019, 16(4): 47:1−47:25.
[38] Feautrier P. Some efficient solutions to the affine scheduling problem. Part I: One-dimensional time. Int'l Journal of Parallel Programming (IJPP), 1992, 21(5): 313−347.
[39] Feautrier P. Some efficient solutions to the affine scheduling problem. Part II: Multidimensional time. Int'l Journal of Parallel Programming (IJPP), 1992, 21(6): 389−420.
[40] Jiang L, Chao Y , Ao Y , *et al*. Towards highly efficient DGEMM on the emerging SW26010 many-core processor. In: Proc. of the Int'l Conf. on Parallel Processing. IEEE, 2017.
[41] Bondhugula U. High performance code generation in MLIR: An early case study with gemm. arXiv:2003.00532, 2020.
[42] Tao XH, Zhu Y, Wang BY, *et al*. Automatically generating high-performance matrix multiplication kernels on the latest sunway processor. In: Proc. of the 51st Int'l Conf. on Parallel Processing (ICPP 2022). New York: Association for Computing Machinery, 2022. Article 52. https://doi.org/10.1145/3545008.3545031

附中文参考文献:

[6] 郑方, 许勇, 李宏亮, 等. 一种面向高性能计算的自主众核处理器结构. 中国科学: 信息科学, 2015, 45: 523−534.
[8] 赵捷, 李颖颖, 赵荣彩. 基于多面体模型的编译"黑魔法". 软件学报, 2018, 29(8): 2371−2396. http://www.jos.org.cn/1000-9825/5563.htm [doi: 10.13328/j.cnki.jos.005563]
[21] 刘芳芳, 杨超, 袁欣辉, 等. 面向国产申威 26010 众核处理器的 SpMV 实现与优化. 软件学报, 2018, 29(12): 3921−3932. http://www.jos.org.cn/1000-9825/5309.htm [doi: 10.13328/j.cnki.jos.005309]
[22] 许志耿. 面向国产 SW26010 众核处理器的科学计算核心深度优化研究 [硕士学位论文]. 上海: 上海交通大学, 2018.
[24] 李雁冰, 赵荣彩, 韩林, 等. 一种面向异构众核处理器的并行编译框架. 软件学报, 2019, 30(4): 981−1001. http://www.jos.org.cn/1000-9825/5370.htm [doi: 10.13328/j.cnki.jos.005370]
[36] 李颖颖, 赵捷, 庞建民. 多面体模型中分裂分块算法的设计与实现. 计算机学报, 2020, 43(6): 1038−1051.

陶小涵(1995－), 男, 博士生, 主要研究领域为先进编译技术.

朱雨(1998－), 女, 博士生, 主要研究领域为先进编译技术.

庞建民(1964－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为先进计算.

赵捷(1987－), 男, 博士, 讲师, CCF 专业会员, 主要研究领域为先进编译技术.

徐金龙(1985－), 男, 博士, 讲师, 主要研究领域为先进编译技术.