多面体模型中分裂分块算法的设计与实现

李颖颖 赵 捷 庞建民
(解放军战略支援部队信息工程大学 郑州 450001)
(数学工程与先进计算国家重点实验室 郑州 450001)

摘 要 循环分块是一种提升程序局部性的循环变换技术.多面体模型中实现了简单的平行四边形分块,但这种分块形状无法有效进行分块之间的并行.为了解决循环分块的块间并行问题,研究人员提出了分裂分块、钻石分块等各种复杂的分块形状.其中,钻石分块已经在多面体模型编译器中得到了实现,但分裂分块由于设计复杂,目前还没有一个有效的实现算法.本文设计了一种分裂分块算法,基于平行四边形分块实现分裂,避免了传统分裂分块依赖于非仿射表达式的问题,并在多面体模型编译器 PPCG 中对该算法进行了实现.实验对涵盖各种情况的 stencil 计算进行了测试,并分别在 CPU 和 GPU 架构上生成分裂分块代码.结果表明,本文提出的算法能在 CPU 架构上与当前最先进的钻石分块性能相当;同时,分裂分块将 PPCG 在 GPU 上生成的代码性能提高 2.7 倍~5.6 倍.

关键词 多面体模型;循环分块;分裂分块;stencil 计算;并行计算
中图法分类号 TP312 **DOI 号** 10.11897/SP.J.1016.2020.01038

Split Tiling Design and Implementation in the Polyhedral Model

LI Ying-Ying ZHAO Jie PANG Jian-Min
(*PLA Strategic Support Force Information Engineering University, Zhengzhou* 450001)
(*State key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou* 450001)

Abstract Loop tiling is an essential transformation for improving locality in programs. The polyhedral model implements parallelogram tiling, but it fails to exploit full tile-level parallelism with such tile shape. The research community designed some complex tiles shapes, including split tile shape, diamond shape, etc., to enable full tile-level parallelism. Unfortunately, only part of the proposed tile shapes, e.g., diamond tiling, were implemented in existing polyhedral compilers. Split tiling is still considered as foreign to the polyhedral model, since its implementation has to resort to non-affine expressions which are out of the scope of the polyhedral model. In this paper, we design a split tiling algorithm and implement the algorithm in a polyhedral compiler, PPCG, by splitting a parallelogram tile which has already been integrated in most existing polyhedral compilers, avoiding the difficulty of implementation in the polyhedral model due to non-affine expressions. We conduct experiments on a suite of iterated stencil computations, targeting on both CPU and GPU platforms, to validate the performance of the algorithm. The experimental results show that the generated code of split tiling performs similarly with that of diamond tiling, the state-of-the-art tiling technique, on CPU. More importantly, split tiling may speed up the generated code of PPCG by 2.7x to 5.6x on GPU.

Keywords polyhedral model; loop tiling; split tiling; stencil computation; parallel computing

收稿日期:2019-09-27;在线出版日期:2020-02-16.本课题得到国家自然科学基金(61702546)资助.**李颖颖**,博士研究生,副教授,中国计算机学会(CCF)会员,主要研究方向为先进编译技术.E-mail:liyingying1005@163.com.**赵 捷**(通信作者),博士,讲师,中国计算机学会(CCF)会员,主要研究方向为先进编译技术.**庞建民**,博士,教授,博士生导师,中国计算机学会(CCF)会员,主要研究领域为高性能计算、信息安全.
* 本文实现的分裂分块算法开源代码仓地址为 https://github.com/yaozhujia/ppcg(测试版本).

1 引　言

多面体模型[1-2]是程序自动并行化领域的一个重要分支，通过发掘程序并行性和数据局部性，将满足仿射约束的串行程序自动转换成能够在不同架构上运行的并行代码. 一般而言，多面体模型通过依赖关系分析[3]、调度变换[4-7]和代码生成[8-10]三个部分实现面向不同架构的程序变换. 其中，调度变换是多面体模型的关键步骤，主要思想是根据依赖关系分析的结果计算出满足某种(如通信量最小)或多种优化目标的调度，以期充分利用目标平台的并行硬件资源. 同时，调度变换通过实现循环合并[11-12]等循环变换技术来缩短活跃变量的生命周期，从而实现“生产-消费”关系的周期最小化，以此来达到提升数据局部性的目的.

调度变换并不能解决异构架构上的多级缓存数据局部性的问题. 循环分块[13-15]是解决多级缓存数据局部性的一种循环变换技术，多面体模型通过对计算出的调度进行仿射变换，将原有的循环计算空间按照特定的方式进行切分，并将切分后的分块映射到不同层次的缓存结构来提升数据局部性.

然而，这种数据局部性的提升在一定程度上影响了程序的并行性. 以图 1[2](a)所示的 stencil 计算为例，多面体模型将该计算抽象成如图 1(b)所示的二维空间上的多面体形式. 循环分块后，该循环计算空间被划分成如图 1(c)所示形式. stencil 计算是科学工程计算中最常见的计算模式之一，在动态规划、图像处理等应用中发挥着重要作用. 针对 stencil 计算模式研究循环分块优化是多面体模型和并行计算领域的一个研究热点[16-18].

不难发现，在图 1(b)所示的空间上，沿水平方向上的所有计算都可以并行执行，而图 1(c)中由于分块之间的依赖关系，相邻两个分块之间，右边分块的所有计算必须在左边分块计算完成后才能启动，从而影响了原有的水平方向之间的并行性.

导致这一现象的原因在于多面体模型在实施仿射变换时，循环分块的方向只能沿变换后坐标轴的方向对循环计算空间进行划分，使得多面体模型得到的分块形状为平行四边形. 为了提升循环分块之间的并行性，一种方式是在循环分块之后对分块再进行调度，实现分块之间的并行. 如图 1(c)所示，经过对循环分块的调度，相同颜色的分块可以并行执行，但这种方式实现的是一种流水并行，必须经过流水装载和排空(起始和结束)阶段.

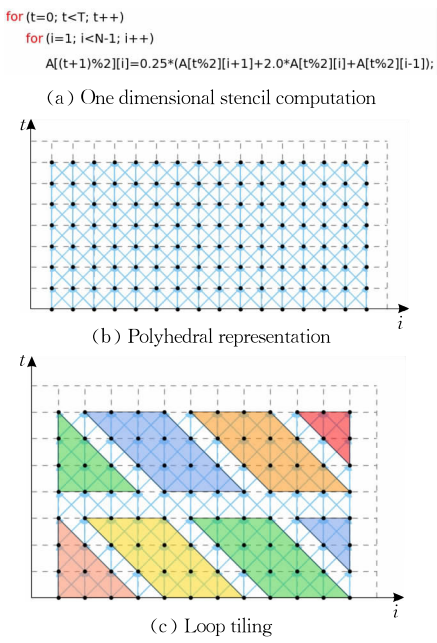


图 1　一维 stencil 计算及其分块示意图

另一种实现分块之间并行性的手段是改变循环分块的形状. 在多面体模型中，能够实现分块之间并行性的形状包括梯形分块(也称为交叉分块)[19-22]、分裂分块[19,23]、钻石分块[24]和六角形分块[25]等. 其中，梯形分块是通过扩展平行四边形的左边，将该平行四边形在水平方向上所需的所有计算进行冗余计算来实现分块水平方向上的并行性，这种冗余计算不仅对程序的性能提升有一定的影响，而且对程序在多面体模型中的表示带来了一定的困难，因为这些冗余计算部分需要被相邻的分块多次执行，与仿射变换的唯一性相冲突.

分裂分块的思想是将梯形分块的冗余计算从分块中分裂出来，将梯形分块划分成两个阶段来消除冗余计算带来的影响，这导致分裂分块的实现也成了一个难以解决的问题. 因此，多面体模型编译器中很少有实现梯形分块和分裂分块的算法.

为了解决这个问题，本文提出了一种有效实现分裂分块的算法. 与分裂分块提出的思想不同，本文提出的分裂分块算法不以梯形分块为基础进行分裂，而是直接在平行四边形分块的基础上对分块进行分裂，这样既有效利用了现有多面体模型编译器的循环分块算法，又避免了梯形分块无法用仿射变换直接表示的困难，为分裂分块在多

面体模型中的实现提供了一种行之有效的算法.

为了验证本文算法的有效性,我们在多面体模型编译器 PPCG[26] 中对分裂分块进行了实现. PPCG 是一种面向不同异构架构的多面体模型编译器,能够自动生成面向 CPU 架构的 OpenMP 代码和面向 GPU 的 CUDA/OpenCL 代码. 但 PPCG 对 CPU 架构的支持并不完善,我们选择在 PPCG 中实现分裂分块的另一目的也是为了提升 PPCG 编译器生成的 OpenMP 代码性能,并与另一个多面体编译器 Pluto[4] 生成的 OpenMP 代码性能进行比较.

实验结果表明,本文提出的算法能够有效提升 PPCG 编译器生成的 OpenMP 代码性能,与 Pluto 算法实现的钻石分块性能相当. 与此同时,利用 PPCG 面向 GPU 代码生成的功能,我们将分裂分块应用于面向 GPU 的代码生成,使 PPCG 生成的 CUDA 代码性能提高了 2.7 倍~5.6 倍.

2 背景知识

2.1 多面体模型

多面体模型是通过将程序中的循环嵌套抽象成空间多面体,并利用多面体上的几何操作来实现程序变换的编译优化模型. 一般而言,多面体模型利用迭代空间(用于表示循环嵌套中语句实例的集合)、访存映射(用于表示语句实例与访存数据之间的仿射关系)、依赖关系(用于表示语句实例之间的依赖关系)和调度(用于表示语句实例的执行顺序)这几种映射关系来表示程序语义. 通过对输入的程序计算新的调度,发掘适合目标体系结构的并行性和数据局部性.

仍以图 1(a)所示的 stencil 计算为例,在对该循环进行程序变换之前,循环嵌套内的语句调度可以表示为

$$S(t,i)\rightarrow(t,i) \tag{1}$$

其中:$S(t,i)$是一个名为 S 的二元组,表示所有语句 S 的执行实例;(t,i)为无名二元组,表示语句的执行顺序;$\rightarrow$代表从 $S(t,i)$到(t,i)的映射,表示语句 $S(t,i)$的所有实例按照(t,i)的方式执行,即先沿 t 轴再沿 i 轴的顺序执行语句 S.

通过计算该循环嵌套的依赖关系,多面体模型能够计算出一个新的调度,其结果为

$$S(t,i)\rightarrow(t,t+i) \tag{2}$$

即多面体模型对该循环嵌套实施了循环倾斜变换.

2.2 调度树

多面体模型用于表示程序的几种集合和映射关系,在多面体模型相关的库[27] 中都可以找到相应的实现. 然而,在实现一些复杂的程序变换时,在集合和映射的基础上进行操作显得十分繁琐,一些情况下这种繁杂的操作使工程实现变得很困难. 因此,本文基于调度树[10,28] 来实现相应的算法.

多面体模型中的调度树是一种无向无环图,其主要节点包括 *domain*、*band*、*filter*、*sequence* 和 *set* 等①. 其中,*domain* 节点表示程序的迭代空间,只能作为一棵调度树的根节点出现,并且调度树的根节点也只能是 *domain* 节点. *band* 节点用于表示其父节点的全序/偏序执行关系,直观上可以想象成程序的循环嵌套. 此外,*band* 节点中可以包含多个成员,每个成员用一对大括号表示,每个成员对应一层循环. 当一个 *band* 节点包含多个成员时,表示这些成员构成的循环嵌套是一个完美循环嵌套.

当调度树需要分叉时,*filter* 节点可以作为每一棵子树的根节点出现,包含被当前子树调度的所有计算实例的集合. 调度树中除了 *sequence* 和 *set* 之外的所有节点都只有一个子节点.

sequence 和 *set* 节点是调度树中引入的特殊节点类型,可以有多个子节点. *sequence*/*set* 节点的意义表示其子节点可以按照顺序/乱序方式执行,并且 *sequence*/*set* 的子节点只能是 *filter* 节点.

如图 2(a)是图 1(a)所示 stencil 计算在多面体模型变换前的调度树表示,经过多面体模型变换之后,其调度树如图 2(b)所示. 其中 *domain* 节点为迭代空间(具体内容可通过图 1(a)中的循环边界计算得出).

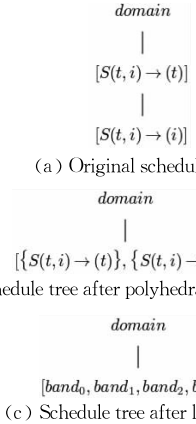


图 2 图 1 所示代码的调度树

① 其它节点类型请参考文献[10].

调度树由多面体编译器的前端解析器根据程序的语义构建. 首先，解析器分析出程序的迭代空间并将其作为 *domain* 节点创建调度树的根节点. 然后，当遇见一个仿射循环时，解析器就向调度树中插入一个 *band* 节点，该 *band* 节点将 *domain* 节点中的语句根据循环迭代顺序构建一个映射关系. 类似地，解析器还可以向调度树中插入其它类型的节点. 当遇见两个并列语句时，可以借助 *sequence* 节点显式地表示语句之间的并列关系. 由此可以得到程序的原始调度树表示.

在得到程序的原始调度树之后，多面体模型通过调度算法计算出一个新的调度树. 在面向 GPU 架构生成代码时，可以通过向调度树中添加不同类型的节点来实现非多面体语义的生成与实现.

2.3　循环分块

在计算出新的调度之后，多面体模型会在新的调度树的 *band* 节点上实现循环分块. 与其它循环变换不同，循环分块是通过增加多面体所在的几何空间维度来实现仿射变换的. 多面体模型能够实现循环分块的充分必要条件是实施循环分块后，分块之间不存在块间的依赖环[16,24].

对于新生成的调度，多面体模型可以通过形如

$$S(t,i)\rightarrow(t/T_t,(t+i)/T_s) \tag{3}$$

的(近似)仿射变换来实现循环分块，其中 T_t 和 T_s 表示对应循环层上的分块大小.

从调度(3)可以看出，多面体模型需要沿新坐标系的两个坐标轴方向对多面体进行切分，即只能沿 t 轴和 $t+i$ 轴实现循环分块. 如果我们在原坐标系 (t,i) 中表示分块后的形状，就可以得到如图 2(c)所示的平行四边形分块后的调度树. 为了便于理解，我们将 *band* 节点中的成员分别用 $band_0$、$band_1$、$band_2$ 和 $band_3$ 来表示，其中，$band_0=\{S(t,i)\rightarrow(t/T_t)\}$ 代表分块后 t 轴方向上的块间迭代的循环，$band_1=\{S(t,i)\rightarrow(t+i)/T_s\}$ 代表分块后 $t+i$ 轴方向上的块间迭代的循环；$band_2=\{S(t,i)\rightarrow(t)\}$ 代表分块后 t 轴方向上的块内迭代的循环，$band_3=\{S(t,i)\rightarrow(t+i)\}$ 代表分块后 $t+i$ 轴方向上的块内迭代的循环.

为了获得更好的程序性能，多面体模型可以继续对循环分块进行调度，从而实现分块之间的并行. 通过利用多面体模型变换，程序可以获得形如

$$S(t,i)\rightarrow(t/T_t,t/T_t+(t+i)/T_s) \tag{4}$$

的调度. 这样可以实现如图 1(c)所示的块间流水并行方式[29]. 在面向 CPU 架构生成 OpenMP 并行代码时，Pluto 编译器中实现了上述块间流水并行，但 PPCG 中并没有实现该并行方式.

2.4　其它分块形状的块间并行

虽然流水并行方式在一定程度上解决了平行四边形分块块间并行的问题，但程序性能还有进一步提升的空间. Pluto 编译器中实现了一种钻石分块，该算法的思想是通过修改 Pluto 调度算法[4,30]，使其寻找到一种能够获得钻石分块形状的调度. 对于图 1(a)所示的循环嵌套，Pluto 计算出的调度为

$$S(t,i)\rightarrow(t-i,t+i) \tag{5}$$

在此基础上再沿坐标系 $(t-i,t+i)$ 的两根轴对空间进行切分，可以得到钻石分块的调度：

$$S(t,i)\rightarrow((t-i)/T,(t+i)/T) \tag{6}$$

其中，T 为分块大小. 在该调度下获得的分块形状如图 3 所示.

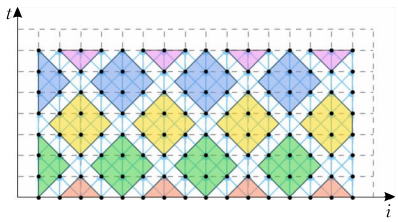


图 3　钻石分块示意图

钻石分块要求分块大小在各个维度上保持一致，而且钻石分块获得的调度(5)在数据局部性上不如调度(2)，因此在实现块间并行后需要对块内再进行调度来提升块内的数据局部性.

另一种通过改变分块形状来实现块间并行的方式是梯形分块和分裂分块. 其中，分裂分块是通过将梯形分块的冗余计算和剩余部分进行分裂而得到的分块形状. 由于梯形分块引入了冗余计算，对冗余计算部分中的某一个语句实例 $S(t,i)$，多面体模型需要计算出一个“一对多”的映射关系，与多面体模型中的仿射约束相冲突，超出了多面体模型的能力范围. 因此，在现有的多面体模型编译器中，梯形分块和分裂分块的实现存在一定的困难.

2.5　研究动机

与梯形分块[19-22]相比，分裂分块[19,23]不引入冗余计算，但是由于分裂分块的设计依赖于梯形分块的实现，因此分裂分块也面临实现困难的问题.

为了能够在多面体模型中实现分裂分块，本文设计了一种新的分裂分块算法. 与传统分裂分块算法实现的方式不同，本文提出的算法不依赖于梯形分块的实现，而是基于平行四边形分块进行分裂，从而避免了梯形分块非仿射表达式的映射问题.

与钻石分块[24]相比，本文设计的分裂分块算法不改变多面体模型的调度算法，因此也避免了钻石分块在实施块间分块后对块内进行重新调度的需求，简化了多面体模型的变换过程.

与六角形分块[25]相比，本文的方法考虑了多条语句等复杂的情况，因此具有更一般的适用性.

3 分裂分块

为了简化说明，我们先描述如何在平行四边形分块的基础上实现分裂分块，然后描述如何在多面体模型中进行实现.

3.1 基于平行四边形分块的分裂

如前文所述，大多数多面体模型编译器中已经实现了平行四边形分块. 因此，我们可以假设已经有形如图 1(c)所示的平行四边形分块.

在整个迭代空间上，所有的平行四边形分块是同构的. 任取其中一个分块，如图 4 所示，我们选取黄色分块作为参考. 在该分块中，通过多面体模型计算，我们可以获得该分块中按字典序最小的点，即图中所示点 A.

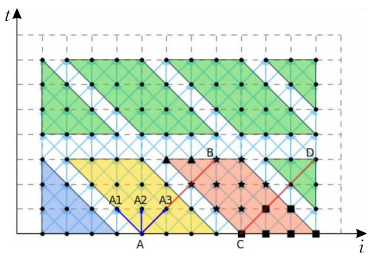


图 4 基于平行四边形分块的分裂

从点 A 出发有三条依赖关系，即 $A\rightarrow A_1$，$A\rightarrow A_2$ 和 $A\rightarrow A_3$. 由于我们考虑的是水平方向上的并行性，$A\rightarrow A_2$ 这条依赖关系在水平方向上的分量为 0，我们不用考虑该依赖关系. $A\rightarrow A_1$ 这个依赖关系由平行四边形的斜边满足.

在实现了平行四边形分块之后，分块之间的依赖关系也是通过原来的依赖关系计算得出. 假设在图 4 中沿 t 轴方向进行平行四边形分块时选取的分块大小为 T_t，那么我们需要对一个分块内按字典序最小的点计算依赖的第 T_t-1 次幂. 对于某一点 P 的一条依赖，其第 n 次幂表示由该依赖关系传递 n 次所导致的依赖. 通过计算点 $A\rightarrow A_3$ 的第 T_t-1 次幂，可以得到点 A 的依赖汇点为 B，如图 4 所示，该点所在的分块用红色表示.

由于 A 是黄色分块中字典序最小的点，说明红色分块内所有在 $A\rightarrow B$ 依赖左边的点不仅依赖于黄色分块内的点，也依赖于蓝色分块内的点. 也就是说，这部分计算产生两个块间依赖，一个是由蓝色分块到红色分块的块间依赖，一个是由黄色分块到红色分块的块间依赖.

现在考虑红色分块内按字典序最小的点 C. 同样地，我们可以计算 C 与 $A\rightarrow A_3$ 同向依赖的第 T_t-1 次幂，这时可以得到如图 4 中所示的依赖 $C\rightarrow D$. 考虑红色分块中 $C\rightarrow D$ 依赖边右边部分，该部分计算的依赖源点都在红色分块内，也就是说，这部分的计算不产生水平块间依赖.

剩余部分由所有在 $A\rightarrow B$ 依赖右边且在 $C\rightarrow D$ 依赖左边的计算组成. 不难看出，这部分的计算产生水平块间依赖，而该水平块间依赖的源点块为黄色分块，汇点块为红色分块.

通过上面的分析，我们可以将红色分块分裂为三个阶段来执行. 第一部分为包括 $C\rightarrow D$ 依赖边右边的所有计算(方块)，该部分计算由于不产生水平块间依赖，因此可以在水平块之间并行执行. 由于平行四边形分块在整个迭代空间上是同构的，因此所有分块对应的部分都可以被分裂出去.

接下来，我们将所有在 $A\rightarrow B$ 依赖右边且在 $C\rightarrow D$ 依赖左边的计算作为第二部分(五角星)，由于黄色分块中第一部分对应的计算已经分裂并计算完成，也就是红色分块第二部分所依赖的计算已全部完成，此时同一水平方向上所有分块对应第二部分的计算也可以并行执行，我们将这部分计算也从红色分块中分裂出去.

最后，考虑 $A\rightarrow B$ 依赖左边的计算(三角形)，这部分计算同时依赖于蓝色分块和黄色分块. 其中，蓝色分块的依赖部分已经在第一阶段计算完成，黄色分块内的依赖部分已经在第二阶段完成. 因此，同一水平方向上所有分块对应部分的计算也可以并行执行. 至此，红色分块内的计算已经全部完成. 由于水平方向上分块的同构性，所有同一水平方向上的分块也全部执行完成.

从上述描述过程可以看出，平行四边形由依赖 $A\rightarrow B$ 和 $C\rightarrow D$ 分成三个部分，而依赖 $C\rightarrow D$ 可以由依赖 $A\rightarrow B$ 向右平移 T_s 个点获得. 对于不同的平行四边形分块，我们可能需要对依赖 $A\rightarrow B$ 进行多次平移然后对分块进行切分. 由此，我们可以断定，从任一分块最小字典序的点 A 计算依赖的第 T_t-1 次幂，获得其对应的依赖汇点 B 及其所在的分块，假设依赖 $A\rightarrow B$ “穿过”m 个分块，说明平行四边形

应分成 $m+1$ 个阶段来实现分裂. 图 5 是 $T_t=T_s$ 时的分裂分块示意图，通过图 4 计算得出；图 6 是 $T_t<T_s$ 时的分裂分块示意图，$T_t>T_s$ 时，分裂分块的示意图可以通过将图 5 中所有的红色三角形与下面的黄色三角形合并之后获得. 与钻石分块不同，我们的方法允许不同循环维度上的分块大小不同.

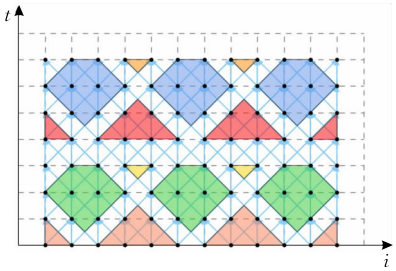


图 5　$T_t=T_s$ 时的分裂分块示意图

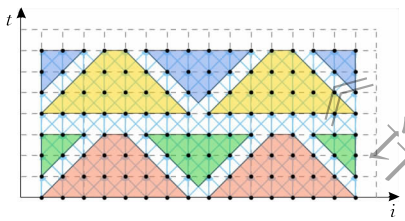
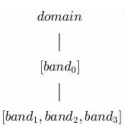


图 6　$T_t<T_s$ 时的分裂分块示意图

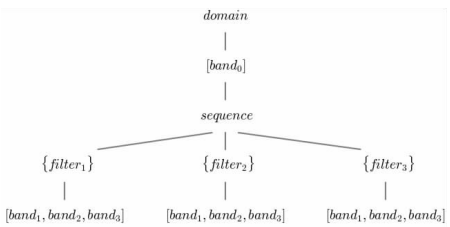
3.2　多面体模型表示

由于多面体模型编译器中已经实现了平行四边形分块，我们可以很容易地获取到平行四边形分块后的调度树如图 2(c)所示. 为了能够在调度树中对上述分裂分块进行表示，我们需要在平行四边形的调度树上进行操作.

首先，分裂分块考虑的是水平方向上的块间并行，需要将平行四边形在表示水平方向的块间循环层上进行分裂. 第一步将表示水平方向的块间循环层成员($band_1$)从 *band* 节点中分裂出来，即我们可以将图 2(c)变换成图 7(a)所示的调度树.



(a) Split $band_0$ from original band node



(b) Introduce sequence node and filters

图 7　分裂分块调度树变换示意图

接下来，我们需要将原有的平行四边形分裂成多个部分. 为了便于说明，我们仍以图 5 所示 $T_t=T_s$ 的情况进行描述. 我们需要将原有的平行四边形分成三个阶段，假设这三个阶段分别为 $phase_1$、$phase_2$ 和 $phase_3$，对应 3.1 节中介绍的第一阶段、第二阶段和第三阶段，其对应迭代空间分别可以用 $filter_1$、$filter_2$ 和 $filter_3$ 表示，并先假设所有 *filter* 中的内容已经明确. 由于这三个阶段是按序执行的，我们需要在这三个 *filter* 节点上引入一个 *sequence* 节点.

插入 *sequence* 节点和 *filter* 节点之后的调度树如图 7(b)所示. 不难发现，每一个 *filter* 节点的子树是在图 7(a)中分裂出来的子树，在调度树上插入多个 *filter* 节点时该子树需要被拷贝到每个 *filter* 节点下面，以保证 *filter* 节点之后的调度关系仍然按照原来的偏序关系执行.

这样，我们就完成了基于平行四边形的分裂分块的多面体模型表示，说明该方法在多面体模型中是可以实现的. 但是，上面的表示基于一种假设，即所有 *filter* 节点中的迭代空间已经明确，因此，如何确定 *filter* 节点的内容是实现分裂分块的关键.

3.3　*filter* 节点的表达式

多面体模型对图 1(a)所示的代码实施了循环倾斜，即现有的调度是按照调度(2)的形式执行计算的. 对于任意一个平行四边形分块，每一个计算点沿斜边的坐标分量可以表示为

$$S_s:=(t+i)-T_s\times floor((t+i)/T_s) \tag{7}$$

其中：$(t+i)$ 为块内点在整个迭代空间上的坐标分量；$T_s\times floor((t+i)/T_s)$ 为该点所在平行四边形分块按字典序最小的点的坐标分量；*floor* 表示对括号内表达式进行向下取整.

类似地，每一个平行四边形分块内点沿垂直方向边的坐标分量可以表示为

$$S_t:=t-T_t\times floor(t/T_t) \tag{8}$$

由于我们只考虑水平方向的块间并行，需要将垂直方向上的因素从调度中消除. 同时，按照 3.1 节中描述的内容，我们需要判定依赖 $A\rightarrow B$ 的表达式.

对于图 1(a)所示的程序，迭代空间上的全局依赖关系为 $A\rightarrow A_1$，$A\rightarrow A_2$ 和 $A\rightarrow A_3$，对应的依赖距离分别是(1，−1)，(1，0)和(1，1). 我们需要找到依赖 $A\rightarrow B$ 的表达式，而该依赖是通过计算 $A\rightarrow A_3$ 的幂获得的，该依赖关系对应的划分平面的法向量为(−1，1)，同时 $A\rightarrow B$ 依赖边也在平行四边形块内，

因此依赖 $A\rightarrow B$ 的表达式可以表示为

$$D:=(-t+i)-T_s\times floor((-t+i)/T_s) \quad (9)$$

以 3.1 节中图 4 所示情况为例，其它形如依赖 $C\rightarrow D$ 的表达式可以通过依赖 $A\rightarrow B$ 向右平移一个或多个平行四边形的水平距离进行计算. 不失一般性，划分平行四边形的依赖边表达式可以表示为

$$D_k:=D-k\times T_s \quad (10)$$

因此，一个 *filter* 节点的表达式可以表示为

$$D_k\leq S_s-S_t<D_{k+1}(-1\leq k\leq m) \quad (11)$$

其中：$k=-1$ 是 3.1 节中第三阶段左边的依赖边(未标出，可通过将 $A\rightarrow B$ 向右平移 $-T_s$ 位，即向左平移 T_s 位获得)；$k=m$ 表示第一阶段右边的依赖边(未标出，可通过将 $A\rightarrow B$ 向右平移 $2T_s$ 位获得).

4 扩展实现

到目前为止，我们已经可以实现分裂分块. 但目前只考虑了包含一个语句的一维 stencil 计算，并且也没有考虑如何将分裂分块映射到 GPU 硬件架构上.

4.1 多维 stencil 的分裂分块

在实现多维 stencil 计算的分裂分块时，不失一般性，假设语句抽象为 $S(t,i_0,i_1,\cdots,i_{n-1})$，我们可以沿时间轴和空间的第一维，即 (t,i_0) 的维度上实现分裂分块，剩余的 $n-1$ 个空间维度 $(i_1,\cdots,i_{n-1})$ 上仍采用平行四边形分块，这样将多维 stencil 计算的分裂分块转换为一维的情况处理.

我们只在多维 stencil 的第一维上实现分裂分块的原因是基于以下几点考虑. 首先，分裂分块实现分块之间并行性的代价是引入了分块之间的同步，在多个维度上都实现分裂分块可能导致同步开销增大，无法达到提升程序性能的目的. 其次，即便在多个维度上实现了分裂，可并行的循环并不连续，导致这些并行循环无法映射到目标架构的不同并行硬件上. CPU 上只提供线程级一层并行，而 GPU 上则要求这些并行循环必须连续才能映射到线程块的不同维度上.

4.2 多个语句的分裂分块

当 stencil 计算中包含多个语句时，依赖关系的情况会变得比较复杂. 不失一般性，假设有两个语句 S_1 和 S_2，如果经过多面体模型变换后所有语句的调度相同，即：

$$S_1(t,i)\rightarrow(t,t+i);\ S_2(t,i)\rightarrow(t,t+i) \quad (12)$$

那么可以按照一条语句的情况处理，所有语句的表达式都可以参照表达式(10)来构造.

但如果多面体模型计算出的调度结果不完全相同，即如果计算出的调度为

$$S_1(t,i)\rightarrow(t,t+i);\ S_2(t,i)\rightarrow(t,t+i+s) \quad (13)$$

其中，s 为调度偏移，那么我们需要对 *filter* 节点的表达式进行进一步处理.

由于 S_1 的调度结果与一个语句时情况相同，我们可以直接使用 S_1 调度的分裂分块结果，其中每个 *filter* 节点中 S_1 的表达式都依据表达式(11)进行构造. 但对于 S_2，计算出的调度存在一个偏移量 s，需要将这种偏移量从左边的分块中包含到当前 *filter* 节点中.

这种包含关系可以表现在表达式(9)中 *floor* 算式的变量中，即对于如式(13)的形式，我们可以计算出 S_2 的依赖边表达式为

$$D:=(-t+i+s)-T_s\times floor((-t+i-s)/T_s) \quad (14)$$

其中，*floor* 括号内的 $-s$ 是由 S_2 的偏移量导致的，而前面部分的常数项 s 是由依赖关系对应的划分平面的法向量决定的.

值得注意的是，除了本文提出的算法，多面体模型中各种分块形状中只有钻石分块可以处理多条语句的情况.

4.3 GPU 硬件映射

GPU 上的分裂分块仍可以采用 CPU 上的分裂分块算法，即只在 (t,i_0) 维度上实现分裂分块，剩余的 $n-1$ 个空间维度 $(i_1,\cdots,i_{n-1})$ 上仍采用平行四边形分块，原因已在 4.1 节中描述.

不同的是，当面向 CPU 架构时，实现分块算法之后就可以生成代码，不需要做硬件的映射，在 CPU 架构上的分块是为了提升 Cache 上的数据局部性. 当面向 GPU 架构时，GPU 硬件本身提供了线程块和线程两级并行硬件，在分块后还需要将对应的循环层映射到 GPU 硬件层上.

对于如图 7(b)所示的调度树，我们需要实现将分裂后的 *band* 节点映射到不同的 GPU 的两级硬件层上. 在从图 7(a)到图 7(b)转换的过程中，我们将水平方向的块间循环层与垂直方向的块间循环层进行分裂，因为垂直方向的块间循环层是无法并行的. 插入 *sequence* 节点后，*sequence* 节点下每个 *filter* 节点后面的 *band* 节点都可以并行执行，但需要将该 *band* 节点映射到两级硬件层上，因此对 *sequence* 下的每个 *band* 节点再次进行分裂，得到如

图 8 所示的调度树. 对应水平方向块间循环层的 $band_1$ 节点被映射到线程块上，分裂后 $band_2$ 节点，即块内循环层对应的 $band$ 节点被映射到线程上.

如果输入程序是一个多维 stencil 计算，那么图 8 中的 $band_1$ 节点包含所有的空间维度. 由于在多维 stencil 计算时我们只在空间第一维上进行分裂分块，因此 $band_1$ 节点的第一个成员将被映射到线程块上，而后面所有的成员必须和该成员分裂，这些成员对应的循环无法并行，也无法直接映射到线程块上. 如何实现这些成员的流水并行并在 GPU 硬件上实现映射是我们未来研究的目标.

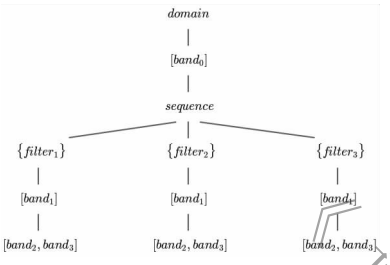


图 8　GPU 硬件映射的调度树变换

4.4　同步最小化

基于平行四边形分块实现的分裂分块将平行四边形划分成多个阶段，然后将这些阶段按序执行，从而在每个阶段完成之后都要进行同步操作.

为了将分裂分块引入的同步最小化，在实现分裂分块时我们提供了一种最小化同步的过程. 如图 5 所示，将黄色小三角形和上面的红色大三角形（组合成一个菱形）进行合并，这样就可以减少因分裂分块引入的同步操作. 此时，我们可以获得和图 3 中所示分块形状相近、同步数量相同的结果. 虽然图 3 中的菱形与图 5 组合而成的菱形不完全一样，但是这无关紧要，我们的目的是通过合并达到同步的最小化. 这种同步操作在沿垂直方向上的分块大小大于等于垂直方向的迭代次数时达到最少. 我们在实现过程中提供了一种同步最小化的编译选项，当该选项被触发时，分裂分块将自动把垂直方向上的分块大小设置为垂直方向上的迭代次数，以达到同步最小化.

4.5　适用范围

首先，本文实现的分裂分块算法是以平行四边形为基础进行分裂的，因此不同的 stencil 计算模式是否能够实现平行四边形分块是本文算法能够适用的前提.

其次，本文实现的算法对平行四边形进行分裂是从计算分块内按字典序最小的点开始的，这要求语句实例之间的依赖距离分量必须为常数，才能在线性整数规划工具 isl[27] 中计算出分块内按字典序最小的点.

最后，由于我们目前还没有完成 GPU 上多维 stencil 计算到 GPU 线程级并行抽象的映射，当前的算法在面向 GPU 架构时处理能力还有待提升，但这并不影响本文算法的适用范围. 目前，我们正在实现这部分功能.

5　实验结果

为了验证本文算法的有效性，我们在多面体模型编译器 PPCG 中对分裂分块进行了实现①. 由于 PPCG 中已经实现了平行四边形分块（用 PPCG Standard 表示），我们提供了一个选项--split-tile 来实现分裂分块（用 PPCG Split/Our Work 表示）. --min-sync 选项用于实现同步最小化；分块大小的选项通过 PPCG 提供的选项支持.

5.1　环境配置和测试用例

我们在 PPCG 中实现了分裂分块，并分别面向 CPU 架构和 GPU 架构，利用 PPCG 生成带有分裂分块的 OpenMP/CUDA 代码. 实验采用的 CPU 架构相关信息见表 1，GPU 架构相关信息见表 2.

表 1　CPU 测试平台信息

Intel(R) Xeon(R) Silver 4110 CPU 2-socket NUMA	
Architecture	x86_64
Clock	2.10 GHz
Cores	8 cores/socket, 2 socket (total: 16 cores)
Hyperthreading	enabled
Cache sizes	64 KB L1, 1024 KB L2, 11 264 KB L3
Compiler	gcc 8.3.0
Compiler flags	-O3 -march=native -mtune=native -ftree-vectorize -DTIME -DVERIFY -fopenmp -lm
ppcg	0.08.1
ppcg flags	--target=c --min-sync --split-tile/--tile --tile-size/--sizes=…
pluto	0.11.4-252
pluto flags	--parallel --partlbtile/--tile --pet
OS	Linux 3.10.0-957.el7.x86_64 (64-bit)

我们采用与钻石分块中使用的相同测试集来验证本文的算法，该测试集包含 8 个测试用例，涵盖了 stencil 计算的各种情况，测试用例的信息如表 3 所示，其中我们也给出 CPU 平台下使用的分块大小，以及各测试用例在单核上的运行时间.

① 本文实现的分裂分块的开源代码仓地址为 https://github.com/yaozhujia/ppcg(测试版本).

表 2 GPU 测试平台信息

NVIDIA Tesla V100 PCIe 32GB	
Architecture	Volta
SMs	84
FP32 Cores	5376
INT32 Cores	5376
FP64 Cores	2688
Tensor Cores	672
Peak(FP64)	7.8 TFLOP/s
Peak(FP32)	15.7 TFLOP/s
Compiler	nvcc: NVIDIA (R) Cuda (V10.1.105)
Compiler flags	-DTIME -DVERIFY -O3
ppcg	0.08.1
ppcg flags	--sizes="…" --split-tile --min-sync

在该测试集中,heat-1d/2d/3d 用例分别代表一维、二维、三维的 heat 计算,其中 heat-1d 的核心循环如图 1(a)所示,对这三个用例的分析可以说明分裂分块在不同维度上的可扩展性. game-of-life[31] 和 3d27pt[32] 代表了大数据规模的 stencil 计算,对这些用例的测试可以验证分裂分块在大型应用中的有效性. apop[33] 用例中循环边界为符号常数,fdtd-1d/2d[34] 用例中包含多条语句,分别用于测试分裂分块在特殊情况下的性能.

表 3 测试集信息

Benchmarks	Problem Size (time×data)	Pluto tile sizes		PPCG tile sizes		1 core Execution time /s			
		standard	diamond	standard	split	pluto-stan.	pluto-diam.	ppcg-stan.	ppcg-split.
heat-1d	$1000\times1\,600\,000$	1024^2	2048^2	4096^2	64×2048	0.58	0.58	0.99	0.56
heat-2d	1000×4000^2	16×64^2	128^3	256^3	128^3	11.59	8.77	14.20	8.54
heat-3d	100×150^3	$16^3\times256$	$16^3\times256$	64^4	$8^3\times256$	0.35	0.34	0.64	0.36
game-of-life	$500\times16\,000^2$	64^3	128^3	256^3	128^3	95.08	84.16	78.86	81.88
apop	$10\,000\times2\,000\,000$	512^2	512^2	2048^2	512^2	10.23	13.70	22.25	13.28
3d27pt	200×256^3	$8^3\times256$	$8^3\times256$	32^4	$8^3\times256$	12.22	12.06	15.60	11.94
fdtd-2d	128×2048^2	64^3	64^3	16^3	16^3	6.05	7.00	7.34	7.87
fdtd-1d	$10\,000\times1\,000\,000$	256^2	256^2	2048^2	128^2	24.23	23.98	11.29	14.69

5.2 CPU 上的性能测试

由于 PPCG 编译器主要面向 GPU 架构生成代码,其面向 CPU 架构的优化能力相对较弱. 为了提升 PPCG 面向 CPU 架构的优化能力,我们首先用 PPCG 生成带有分裂分块的 OpenMP 代码,并将生成的代码与当前最先进的 Pluto 编译器进行比较. 与 PPCG 不同,Pluto 编译器实现了平行四边形分块(用 Pluto Standard 表示)的流水并行及钻石分块(用 Pluto Diamond 表示).

图 9～图 16 列出了实验使用的 8 个测试用例的性能对比. 由于 PPCG 没有实现流水并行,因此在大部分情况下其平行四边形分块的性能弱于 Pluto.

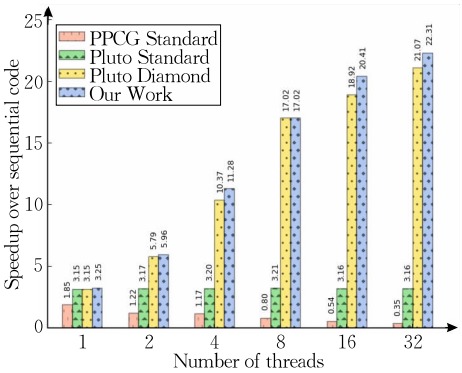


图 9 heat-1d 在 CPU 上的测试性能对比

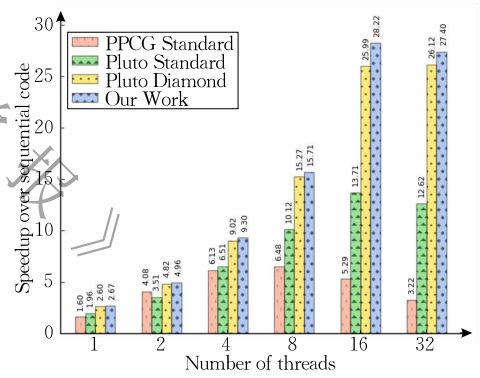


图 10 heat-2d 在 CPU 上的测试性能对比

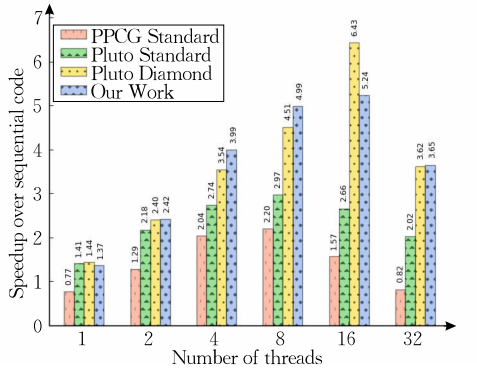


图 11 heat-3d 在 CPU 上的测试性能对比

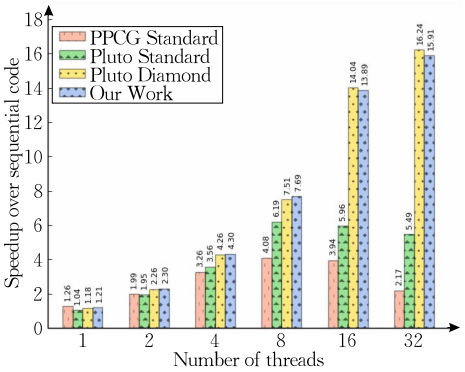


图 12　game-of-life 在 CPU 上的测试性能对比

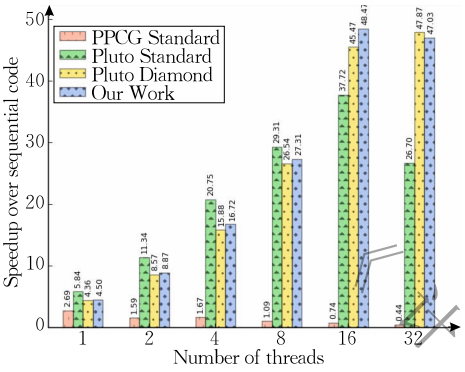


图 13　apop 在 CPU 上的测试性能对比

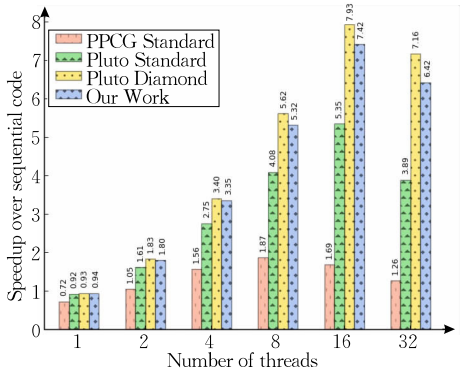


图 14　3d27pt 在 CPU 上的测试性能对比

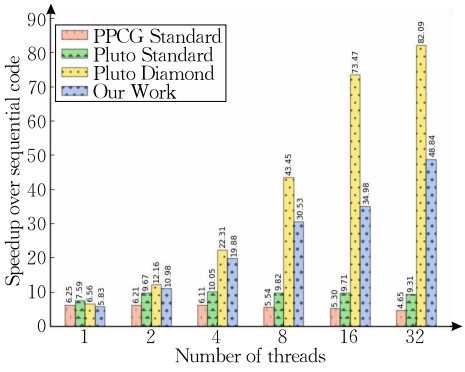


图 15　fdtd-2d 在 CPU 上的测试性能对比

从整体上讲，钻石分块的性能优于平行四边形分块，分裂分块在包括 heat-1d/2d/3d、apop 和 fdtd-1d 等五个用例上的性能优于钻石分块，game-of-life

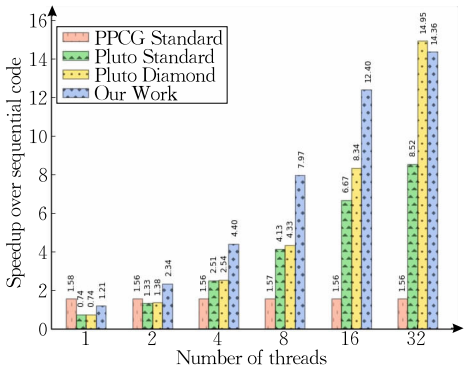


图 16　fdtd-1d 在 CPU 上的测试性能对比

用例上的性能与钻石分块相当. 其中，由于分裂分块算法的设计，PPCG 在生成 apop 的并行代码时实现了 if 控制流外提，导致分裂分块性能优于钻石分块；而分裂分块相当于将钻石分块的三个阶段进行了展开，虽然生成的代码有所膨胀，但循环边界表达式比钻石分块简单，使得我们的算法生成的代码在上述五个用例上性能优于钻石分块.

对于 fdtd-2d 用例，钻石分块的性能比分裂分块的性能更好，这种性能差异是由 PPCG 和 Pluto 的代码生成策略导致的，Pluto 对生成的代码进行了深度展开，降低了控制流开销，但 Pluto 在该用例上的编译时间也被大大延长.

5.3　GPU 上的性能测试

与钻石分块不同，本文的算法在 PPCG 中实现，因此也可以完成 GPU 上的代码生成. 由于在 GPU 平台上，除了实现分裂分块，还要设计相应的硬件映射策略，目前我们暂时只实现了一维 stencil 计算的映射策略，在 GPU 平台上我们采用表 3 中的部分一维测试用例进行实验.

表 4 列出了我们在 GPU 平台上使用的测试用例以及不同分块的分块大小. 与此同时，我们还列出了这些测试用例在 GPU 上的运行时间.

表 4　GPU 上测试用例

Benchmarks	PPCG tile sizes			Execution time		
	stan.	hybr.	split	ppcg-stan.	ppcg-hybr.	ppcg-split.
heat-1d	128	32×16	256^2	36.37 ms	13.34 ms	13.08 ms
apop	128	32×16	256^2	1.06 s	217.07 ms	187.84 ms
fdtd-1d	128		128^2	643.01 ms		146.36 ms

我们利用 PPCG 生成 CUDA 代码来测试 GPU 上的性能. 在生成 CUDA 代码时，PPCG 会计算一个最外层可以并行的调度，但是调度(2)会导致外层循环不可以并行，因此在默认选项下 PPCG 生成的 CUDA 代码(用 PPCG Standard 表示)不会实现循环倾斜，而是选择只对内层循环进行分块. 六角形分

块(用 PPCG Hybrid 表示)和分裂分块(用 PPCG Split/Our Work 表示)弥补了其不足,通过对内外层循环进行分块,并对内层循环之间的分块间实现并行,达到提升数据局部性的目的.由于六角形分块的形状复杂,其生成的 CUDA 代码中循环边界更复杂,也包含更多的 if 控制流语句,导致其 CUDA 代码线程间的负载不均衡;同时,六角形分块比分裂分块线程块之间的同步次数更多.本文提出的算法仍采用 CPU 平台上相同的分裂分块,相对于 PPCG 默认情况下的分块策略,性能可提升 2.7 倍~5.6 倍;相对于六角形分块,可支持多条循环语句的处理,且性能更优,如图 17 所示.

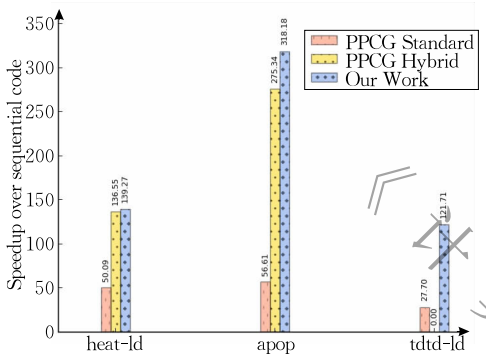


图 17 GPU 平台上的程序性能对比图

5.4 分裂分块与六角形分块的性能对比

在 GPU 上进行性能测试时,我们用本文设计的分裂分块算法与六角形分块算法进行了对比,本文算法的性能优于六角形分块,其主要原因有两点:

首先,分裂分块生成的代码比六角形分块生成代码含有更少的 if 控制流分支语句.利用 PPCG 生成 CUDA 代码时,用户可以调整的参数是 CUDA 程序中逻辑上的并行抽象,即线程块和线程.其中,线程块对应 GPU 上的流式多处理器(Streaming Multiprocessor),线程对应 GPU 上的流处理器(Streaming processor).从硬件角度来讲,在流式多处理器和流处理器之间还有一层线程束(wrap)的抽象.线程束是流式多处理器中的基本执行单元,线程束内的流处理器无法同时执行不同的指令.GPU 处理控制流语句的策略是让线程束内的所有流处理器计算所有的分支,这导致不同的控制流分支在线程束内是串行执行的,而且在执行每个分支时会有部分流处理器闲置的情况,造成 GPU 上的程序性能下降.这种情况在嵌套分支语句的情况下更加明显.六角形分块比分裂分块生成的代码中拥有更多的嵌套分支语句,这是其生成的 CUDA 代码比分裂分块代码性能较低的主要原因.

其次,通过对比不同分块形状生成的 CUDA 代码,我们发现六角形分块会导致更多的同步操作.我们在设计分裂分块时通过--min-sync 编译选项的支持,实现了同步的最小化,降低了 CUDA 代码 kernel 函数之间的同步开销.

5.5 编译时长测试

实验也对不同循环分块形状的编译时长进行了对比.CPU 平台生成不同形状的循环分块编译时间如表 5 所示,GPU 平台上不同形状的循环分块编译时间如表 6 所示.其中,ppcg-split 表示未实现同步最小化功能的分裂分块算法,ppcg-split-sync 表示已实现同步最小化功能的分裂分块算法.

表 5 CPU 平台编译时间

Benchmarks	PPCG /ms			Pluto /ms	
	standard	split	split-sync	standard	diamond
heat-1d	0.104	8.123	7.144	1.680	7.158
heat-2d	0.176	13.732	162.861	4.764	11.287
heat-3d	0.210	18.946	373.988	10.761	25.995
life	0.166	15.243	76.065	7.244	18.121
apop	0.123	12.499	12.553	1.810	4.076
3d27pt	0.203	33.222	338.278	34.477	89.260
fdtd-2d	0.876	59.311	330.421	55.337	85.788
fdtd-1d	0.197	21.412	2351.017	7.489	14.461

表 6 GPU 平台编译时间

Benchmarks	PPCG /ms		
	standard	hybrid	split
heat-1d	6.3360	106.6260	116.5090
apop	13.9590	136.9400	423.0600
fdtd-1d	10.9730		295.4210

与 PPCG 实现的默认分块形状相比,无论是在 CPU 架构还是 GPU 架构上,本文提出的分裂分块算法在未实现同步最小化功能时会导致一定的编译时间开销,但这与 Pluto 上实现钻石分块的编译时间基本相当或更优.

当利用本文提出的算法面向 CPU 架构实现同步最小化功能时,会有较大的时间开销,这是因为实现同步最小化后调度树 band 节点下面会有多个 filter 节点,增加了代码生成的时间.另外,实现同步最小化功能后,生成的代码规模也会膨胀.

当利用本文提出的算法面向 GPU 架构实现同步最小化功能时,也会有较大的时间开销,原因与 CPU 架构的情况相同.但在大部分情况下,都与六角形分块所需的编译时间基本相当.

6 结 论

循环分块是提升程序局部性的一种有效变换技

术，在多级缓存等复杂的体系结构中显得更加重要.多面体模型是实现程序自动并行化的一种有效方法，在多面体模型中实现循环分块具有十分重要的意义.然而，受限于仿射变换和仿射表达式的约束，多面体模型中只能实现简单的平行四边形分块，使程序性能无法达到最优.

本文提出了一种在多面体模型中易于实现的分裂分块算法，并在多面体编译器 PPCG 中进行了实现.首先，本文提出的算法不依赖于梯形分块，因此解决了分裂分块在多面体模型中实现复杂的问题.其次，本文实现的分裂分块算法允许不同循环维度上的分块大小不同，也能够处理多条语句、符号常量循环边界等复杂情况，比当前最先进的钻石分块、六角形分块具有更一般的通用性.最后，本文提出的算法在 PPCG 中实现，并初步实现了 GPU 硬件映射策略，与当前最先进的分块技术相比，能够支持更多的目标体系架构.

目前我们在 PPCG 中实现了一种简单的 GPU 硬件映射策略，为了更好地验证本文算法的有效性，我们的下一步工作将完善 GPU 硬件映射策略的实现.另外，在 GPU 平台上实现块内重新调度也是我们下一步的计划.当前，我们对所有的 stencil 计算采用的都是一维 stencil 的映射策略，即线程块和线程上采用相同的映射，我们的初步想法是针对多维 stencil 计算的分裂分块，在线程块和线程两级硬件架构上采用不同的映射策略，在线程块上调度分块之间的流水并行，在线程上实现块内的完全并行，充分利用和发挥 GPU 架构的并行特征.

致　谢　感谢审稿人对本文提出的意见！

参　考　文　献

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

[2] Zhao Jie, Li Ying-Ying, Zhao Rong-Cai. "Black magic" of polyhedral compilation. Journal of Software, 2018, 29(8): 2371-2396(in Chinese)
(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)

[3] Feautrier P. Dataflow analysis of array and scalar references. International Journal of Parallel Programming, 1991, 20(1): 23-53

[4] Bondhugula U, Hartono A, Ramanujam J, Sadayappan P. A practical automatic polyhedral parallelizer and locality optimizer// Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, USA, 2008: 101-113

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

[6] Acharya A, Bondhugula U, Cohen A. Polyhedral auto-transformation with no integer linear programming// Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018). New York, USA, 2018: 529-542

[7] Kong M, Pouchet L N. Model-driven transformations for multi-and many-core CPUs//Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019). New York, USA, 2019: 469-484

[8] Bastoul C. Code generation in the polyhedral model is easier than you think//Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT'04). Washington, USA, 2004: 7-16

[9] Chen C. Polyhedra scanning revisited//Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12). New York, USA, 2012: 499-508

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

[11] Bondhugula U, Gunluk O, Dash S, et al. A model for fusion and code motion in an automatic parallelizing compiler// Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10). New York, USA, 2010: 343-352

[12] Jangda A, Bondhugula U. An effective fusion and tile size model for optimizing image processing pipelines//Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'18). New York, USA, 2018: 261-275

[13] Wolfe M. More iteration space tiling//Proceedings of the 1989 ACM/IEEE Conference on Supercomputing. New York, USA, 1989: 655-664

[14] Wolf M, Lam M. S. A data locality optimizing algorithm// Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, USA, 1991: 30-44

[15] Irigoin F, Triolet R. Supernode partitioning//Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language. New York, USA, 1988: 319-329

[16] Rawat P S, Vaidya M, Sukumaran-Rajam A, et al. On optimizing complex stencils on GPUs//Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS). Washington, USA, 2019: 641-652

[17] Zhao T, Basu P, Williams S, et al. Exploiting reuse and vectorization in blocked stencil computations on CPUs and GPUs//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19). New York, USA, 2019: 1-44

[18] Korch M, Werner T. Improving locality of explicit one-step methods on GPUs by tiling across stages and time steps. Future Generation Computer Systems, 2020, 102(1): 889-901

[19] Krishnamoorthy S, Baskaran M, Bondhugula U, et al. Effective automatic parallelization of stencil computations//Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, USA, 2007: 235-244

[20] Holewinski J, Pouchet L N, Sadayappan P. High-performance code generation for stencil computations on GPU architectures//Proceedings of the 26th ACM International Conference on Supercomputing. New York, USA, 2012: 311-320

[21] Davis E C, Strout M M, Olschanowsky C. Transforming loop chains via macro dataflow graphs//Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO'18). New York, USA, 2018: 265-277

[22] Zhao J, Cohen A. Flextended tiles: A flexible extension of overlapped tiles for polyhedral compilation. ACM Transactions on Architecture and Code Optimization, 2019, 16(4): 47:1-47:25

[23] Grosser T, Cohen A, Kelly P H J, et al. Split tiling for GPUs: Automatic parallelization using trapezoidal tiles//Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units. New York, USA, 2013: 24-31

[24] Bondhugula U, Bandishti V, Pananilath I. Diamond tiling: Tiling techniques to maximize parallelism for stencil computations. IEEE Transactions on Parallel and Distributed Systems, 2017, 28(5): 1285-1298

[25] Grosser T, Cohen A, Holewinski J, et al. Hybrid hexagonal/classical tiling for GPUs//Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization. New York, USA, 2014: 66-75

[26] Verdoolaege S, Carlos Juega J, Cohen A, et al. Polyhedral parallel code generation for CUDA. ACM Transactions on Architecture and Code Optimization, 2013, 9(4): 54: 1-54:24

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

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

[29] Hogstedt K, Carter L, Ferrante J. Determining the idle time of a tiling//Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, USA, 1997: 160-173

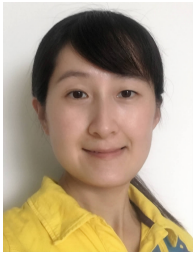
[30] Bondhugula U, Baskaran M, Krishnamoorthy S, et al. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model//Proceedings of the International Conference on Compiler Construction. Berlin, Heidelberg: Springer-Verlag, 2008: 132-146

[31] Gardner M. Mathematical Games. New York: Scientific American, 1970: 120-123

[32] Datta K, Murphy M, Volkov V, et al. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures//Proceedings of the 2008 ACM/IEEE Conference on Supercomputing. Piscataway, 2008: Article 4

[33] Hull J. Options, Futures, and Other Derivatives. 6th Edition. Upper Saddle River: Pearson Prentice Hall, 2006

[34] Polybench [EB/OL]. Available: http://polybench.sourceforge.net, 2019



LI Ying-Ying, Ph. D. candidate, associate professor. Her research interest is advanced compilation technology.

ZHAO Jie, Ph. D., lecturer. His research interest is advanced compilation technology.

PANG Jian-Min, Ph. D., professor, Ph. D. supervisor. His research interests include high performance computing and information security.

Background

This paper presented an implementation of split tiling in the polyhedral model. The polyhedral model is recognized as its power to composite various affine loop transformations, and is therefore considered as one of the natural candidates for optimizing deep neural networks. However, classical polyhedral compilation frameworks are usually impotent when modeling non-affine transformations.

Derived from overlapped tiling, split tiling is an effective tiling technique for enabling tile-wise concurrent start that classical rectangle/parallelogram tiling cannot find. However,

the implementation of split tiling was also constrained by that of overlapped tiling, which requires the modeling of non-affine expressions in polyhedral compilation frameworks. This makes the implementation of split tiling in the polyhedral model a difficult task. Unlike existing work that extends the polyhedral model to handle non-affine expressions using auxiliary strategies like pre-processing, programming language specifications, runtime assistance, etc., the algorithm designed in this paper takes another way by implementing split tiling that is derived from classical rectangular/parallelogram tile shapes, which can be implemented in the polyhedral model using quasi-affine relations. The algorithm presented in this paper improves the performance of the PPCG compiler when generating OpenMP code for CPU architectures, and obtains competitive performance with that of another polyhedral compiler, i.e., Pluto. Moreover, the CUDA code generated by the PPCG compiler also benefits from the split tiling technique introduced in this paper. The performance of the generated CUDA code using our technique is better than the state-of-the-art hexagonal tiling for GPUs.

This work is supported by the National Natural Science Foundation of China under Grant No. 61702546. The purpose of this project is to solve the programming issue on diverse heterogeneous architectures using the polyhedral model. The members of this project have made progress on extending polyhedral compilation for handling non-affine applications and modeling non-affine loop transformations. For example, we extended the polyhedral model to handle dynamic counted loops that presents frequently in dynamic programming, sparse matrix computation, finite element method, etc. We also implemented overlapped tiling to improve the performance of image processing pipelines on both CPUs and GPUs. These contributions have brought the polyhedral model to the front scene of many application domains like deep learning and image processing. Recently, the polyhedral model has been integrated into the MLIR (Multi-level Intermediate Representation) project. The members of MLIR are designing a so-called "graybox" polyhedral optimizations, which is similar to our implementation for handling non-affine applications. Overlapped tiling becomes one of the most suitable tile shapes for deep neural networks due to the nature of convolution operations that involves a lot of redundant computation. Our implementation of overlapped tiling in general-purpose compilers will benefit automatic transformations of deep neural networks and we are now working along this direction.