软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2013,24(8):1775−1785 [doi: 10.3724/SP.J.1001.2013.04343]

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

基于依赖分析的 SPMD 程序隐式同步检测及处理算法*

岳 峰, 庞建民, 赵荣彩

(解放军信息工程大学,河南 郑州 450002)

通讯作者: 岳峰, E-mail: firstchoiceyf@163.com

摘 要: SPMD翻译是指将一种特定类型的SPMD程序编译到多种设备上,当前的细粒度SPMD翻译研究建立在线程之间相互独立的假定上,线程之间只通过显式同步进行通信.但线程之间还隐含存在着各种数据依赖,如隐式同步,这导致了 SPMD 翻译在处理隐式同步时的正确性缺陷.为了对隐式同步进行处理,对细粒度 SPMD 模型 CUDA 中的隐式同步进行了系统的分析,指出了当前翻译 CUDA 程序到多核平台的相关研究在处理隐式同步上的不足,提出了基于依赖分析的隐式同步检测方法.在检测出隐式同步的基础上,设计了循环重排序的优化处理算法,对显式同步和隐式同步进行了统一处理.实验结果表明,与现有的SPMD翻译方法相比,该检测及处理算法能够正确而快速地检测并翻译 CUDA 中的各种隐式同步,代价较小,有助于编译器产生正确而有效的翻译结果.

关键词: SPMD翻译;显式同步;隐式同步;依赖分析;线程循环;循环重排序

中图法分类号: TP311 **文献标识码**: A

中文引用格式: 岳峰,庞建民,赵荣彩.基于依赖分析的SPMD程序隐式同步检测及处理算法.软件学报,2013,24(8):1775−1785. http://www.jos.org.cn/1000-9825/4343.htm

英文引用格式: Yue F, Pang JM, Zhao RC. Detecting and treatment algorithm of implicit synchronization based on dependence analysis in SPMD program. Ruan Jian Xue Bao/Journal of Software, 2013,24(8):1775−1785 (in Chinese). http://www.jos.org.cn/1000-9825/4343.htm

Detecting and Treatment Algorithm of Implicit Synchronization Based on Dependence Analysis in SPMD Program

YUE Feng, PANG Jian-Min, ZHAO Rong-Cai

(PLA Information Engineering University, Zhengzhou 450002, China)

Corresponding author: YUE Feng, E-mail: firstchoiceyf@163.com

Abstract: SPMD translation compiles programs of one SPMD-threaded programming model to multi devices. The current researches base on the supposition that different threads are independent except in communication with explicit synchronizations. However, the data dependence relation between threads such as implicit synchronizations results in the correctness pitfalls in SPMD translation. In order to deal with implicit synchronizations, the implicit synchronizations in fine-grained SPMD programming model CUDA are analyzed systematically. The correctness pitfalls in existing SPMD translation from CUDA to Multi-core are revealed in which this paper proposes a method of detecting implicit synchronizations based on dependence analysis. On the basis of implicit synchronizations detecting, an optimized treatment algorithm is designed to treat explicit and implicit synchronizations synthetically by the loop reorder. The experimental results show that compared with existing SPMD translation, the detecting and optimized algorithm could treat kinds of implicit synchronizations in fine grained SPMD translation correctly and quickly by small expense, which helps compiler produces correct and efficient result.

Key words: SPMD translation; explicit synchronization; implicit synchronization; dependence analysis; thread loop; loop reorder

* 基金项目: 国家高技术研究发展计划(863)(2009AA012201); 国家科技重大专项(核高基)(2009ZX01036-001-001); 河南省重大科技攻关专项(092101210501)

收稿时间: 2011-07-03; 定稿时间: 2012-10-19

异构与多核设备在计算领域得到了快速发展,各种各样的异构多核系统纷纷出现.然而,不同类型的异构多核系统有其独特的编程模型和特性,为特定的系统开发程序必须严格遵守它们的编程规范.这造成了代码移植的障碍,严重影响了编程效率,同时对并行编程和编译带来了挑战.

为了应对挑战,近年来,并行编程者们展开了许多面向跨设备的通用编程模型研究,包括开发通用的并行编程语言、库和编译器,如支持多种设备的 Lime 语言[1]、开放的并行编程框架 OpenCL(open compute language,开放计算语言)[2]、并行程序移植框架 MCUDA[3]和 Ocelot[4].在对不同设备进行编译时,一个重要的挑战是处理程序中的同步,不同的设备对同步有不同的限制,如典型的图形加速设备要求程序提供显式的数据同步,并以 SIMT (single instruction multi-thread,单指令多线程)的方式保证线程间指令同步执行[5].而传统的多核系统仅提供有限的存储器栏栅指令和互锁原语来保证存储空间或临界资源的一致性[6].这两个平台在同步方面的差异,导致了编写通用程序或跨平台代码移植的困难.当前的通用编程模型缺乏对设备相关的同步机制的系统研究,跨设备代码移植工作在平衡不同系统间同步的差异方面还存在着不足,尤其是在细粒度 SPMD(single program multi-data,单程序多数据)翻译方面.

细粒度 SPMD 程序的跨平台编译是指以一种 SPMD 编程模型(如 CUDA(compute unified device architecture,统一设备计算架构))为基准,为多核或其他设备编译出不同的代码[7].在细粒度的 SPMD 程序里,大量的线程在相同的内核函数执行不同的数据集,单线程的粒度较小,因此,任务之间的并行性可以最大程度地开发.同时,通过任务扩大可以产生粗粒度任务的代码以移植到其他设备.SPMD 的翻译简化了异构系统的编程,并且能够促进不同设备的集成,使任务光滑地分配或移植.在假设线程之间相互独立的情况下,源代码层的 MCUDA[3]和二进制层的 Ocelot[4]等研究实现了翻译 CUDA 程序到 x86 多核平台,本文中称它们的工作为基础 SPMD 翻译.

当前的通用编程模型只处理了显示同步,然而线程之间除了显式同步之外,还存在一些控制与数据依赖关系,这些关系使线程间不再独立.Guo 等人正式提出了隐式同步[8]的概念,相对于由显式语句表示的显式同步,隐式同步是指在没有同步语句的情况下,由 GPU 的硬件特性保证的线程间同步执行指令.隐式同步给基础 SPMD 翻译带来了问题,它们不能正确地检测并处理隐式同步,对包含隐式同步的程序无法翻译,影响了进一步的应用.Guo 等人随后给出了一种单索引依赖分析[8]以及构造线程级依赖图的隐式同步检测方法[9],但是单索引依赖分析不具备通用性,另外,线程级依赖图是程序运行时动态构造的,需要复杂的代码变换,不利于代码的维护和移植,判断过程也增加了检测及处理时间.

此外,一些研究为改进 GPU 的编程能力而开发了 OpenMP 到 CUDA 的编译器,或直接扩展 CUDA 或 OpenCL.比如,文献[10]利用 GPU 在运行时阶段优化 CPU 程序,文献[11]优化 CUDA 程序中的分歧控制流,文献[12]研究编译和运行时的 CPU 和 GPU 任务划分以及其他多种通过软件或者硬件优化 GPU 性能的技术.这些研究出于性能优化的目的,涉及到研究同步的位置与数量对性能的影响,但大多考虑的是显式同步,隐式同步作为循环展开的一种扩展优化技术,其重要性往往被忽视.

本文关注细粒度 SPMD 程序翻译中同步的处理,以源代码层的 MCUDA 为实验平台.第 1 节阐述显式同步与隐式同步的关系,揭示了基础 SPMD 翻译方法的不足.基于第 1 节的发现.第 2 节提出可保留依赖判定定理.通过构造多索引下标串行循环,对线程间只可能发生通信的共享变量进行分析,实现对可保留隐式同步和关键隐式同步进行判定.第 3 节基于依赖分析结果设计统一的同步处理方法及优化的隐式同步处理算法,将通过重排序可以保留的隐式同步保留,加快处理效率.第 4 节对本文的检测及优化处理算法进行测试,以基础 SPMD 翻译及其修正版为比较对象,评估本文算法的正确性及效率.第 5 节给出总结.

本文的主要贡献在于:

- 对细粒度 SPMD 翻译中的显式同步和隐式同步进行了系统的分析,揭示了基础 SPMD 翻译在处理隐式同步时存在的问题;
- 提出了通用的基于依赖分析的隐式同步检测及分类方法,适用于各种情况的隐式同步检测;
- 给出了统一的同步处理以及优化方法,以解决 SPMD 翻译中的隐式同步处理问题.

1 隐式同步及其带来的问题

1.1 隐式同步描述

CUDA 的设备函数遵循并扩展了 BSP(bulk synchronous parallel,大规模同步并行)模型[13].根据这一模型,CUDA 设备函数支持发布数量众多的线程并分层组织,一定数量的线程组成线程束,线程束组成线程块,线程块进而组成线程格.线程块之间不需要通信,线程块内部的通信通过显式的同步函数调用来实现.这种同步方式为显式同步.而线程束内的线程由硬件以 SIMD(single program multi-data,单指令多数据)方式执行来保证的同步执行指令为隐式同步.

显式同步有这样的特点:如果 CUDA 线程在 if-else 结构中执行不同的分支,就禁止在两个分支中都调用栅栏同步[14].因此,线程块中的所有线程必须全部执行或全部不执行分支上的同步.这个特性对在多核架构上映射 CUDA 线程有重要的影响.这是因为显式同步是线程无关的,即当前线程的同步不受其他线程控制流及数据流的影响.基础 SPMD 翻译利用该特性通过分割显式同步之间的代码并构造串行线程循环,使 CUDA 设备函数能够在多核平台上串行执行.

相对于显式同步,隐式同步呈现出不同的特性.它是线程相关的,不同的线程之间存在着数据依赖,隐式同步仅发生在引用共享变量的语句,其他访问局部变量等的语句不需考虑.这是因为隐式同步是由于线程间发生了数据依赖而产生,而线程间的通信只能通过共享变量发生.

显式同步和隐式同步的主要区别在于:显式同步要求同步范围(线程块)内所有线程都进行同步;隐式同步由硬件使用掩码使同步范围(线程束)内部分线程同步的执行结果消除,表现出剩余的部分线程同步.这种区别使得显式同步的处理方法不再适用于隐式同步.当前,基础 SPMD 均没有对隐式同步进行处理,下面说明隐式同步带来的翻译正确性缺陷.

1.2 基础SPMD翻译的缺陷

SPMD 翻译是指将 SPMD 程序编译为非 GPU 设备(如多核平台)可接受的代码.本文以 MCUDA 作为说明平台,其翻译原理可以通过图 1(a)和图 1(b)所示的翻译过程来展示.通过识别显式同步调用及应用在控制结构上的代码分裂,MCUDA 构造串行线程循环以实现线程的串行执行,图 1(a)中的 for 循环经翻译后产生了图 1(b)中的 3 个串行循环,依次执行.

```
_global_void
matriMul(...)
{
...
for (a=astart;
      a<aEnd;
      a+=aStep) {
      ...
      _syncthreads();
      ...
      }
...
}
```

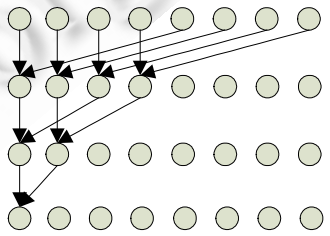
(a) 最初的 CUDA 代码

```
_global_void
matriMul(...)
{
...
thread_loop{
   ...
   a=astart;
   }

While (a<aEnd) {
      thread_loop{
         ...
         }
      //_syncthreads();
      thread_loop{
         ...
         a+=aStep;
         }
      }
...
}
```

(b) 对(a)中代码的翻译

```
Reduce4(...)
{
...
//sdata[] is shared array;
//warpSize =32;
1  if (tid<32)
2  {
3    sdata[tid]+=sdata[tid+16];
4    sdata[tid]+=sdata[tid+8];
5    sdata[tid]+=sdata[tid+4];
6    sdata[tid]+=sdata[tid+2];
7    sdata[tid]+=sdata[tid+1];
8  }
}
```



(c) CUDA SDK 中 reduce 程序的部分代码及算法

Fig.1 Illustration of MCUDA's translation principle and it's pitfall

图 1 MCUDA 翻译原理及缺陷说明

MCUDA 的翻译方法在没有隐式同步的情况下是正确的,但在出现隐式同步的代码中会出现问题.考虑 CUDA SDK[15]中存在隐式同步的程序 Reduce,部分代码如图 1(c)左半部分所示.语句 3~语句 7 存在着隐式同步的关系,对于线程 0~线程 31 组成的线程束,会执行 if 的分支,即在 GPU 上线程束内的所有线程将并行执行语句 3~语句 7,最后规约的结果集中到线程 0.算法示意如图 1(c)右半部分所示.经 MCUDA 翻译后,每一个线程将依次串行地执行语句 3~语句 7.这与程序的原算法是明显不同的,将产生错误的执行结果.

导致翻译隐式同步语句出错的原因就是线程之间的数据依赖.线程束内的线程以 SIMD 的方式并行执行,这导致线程对共享数据引用的值有可能是其他线程在之前同步执行语句产生的结果.而翻译后指令在线程间依次串行执行,它们的数据依赖关系不能被正确保留,因而会产生错误.

2 基于依赖分析的隐式同步检测

从以上的分析可以得知,隐式同步是由于线程间发生了数据依赖而产生的,那么对其检测主要集中于数据依赖分析[16].为不失一般性,本文分析隐式同步的各种情况,这些分析可以很容易地应用到其他模型,如 OpenCL 的同步检测中.首先对检测方法中使用的几个概念进行定义.

定义 1. 对于 CUDA 设备函数代码 *C*(为简化分析,假设其不包含显式同步,并默认在一个线程束范围内),经现有基础 SPMD 翻译后得到的串行化的循环代码称为线程循环,以 *L* 表示.

L 以 *C* 的内容为循环体,以三维线程索引 *tid.z*,*tid.y*,*tid.x* 作为 3 个循环索引,*C* 中所有对 *tid.z*,*tid.y*,*tid.x* 引用的变量均使用归纳变量替换的方法替换.

定义 2. 线程间共享的数据必须按正确的顺序由不同的线程生产和消费,这些约束称为线程间数据依赖.可能发生的 3 种依赖方式为:

- 真依赖:线程 *A* 在语句 *S* 真依赖于线程 *B*,是指 *A* 对共享数据的读依赖于 *B* 之前对其的写;
- 反依赖:与真依赖相反,线程 *B* 首先对共享数据读,然后由进程 *A* 对其写;
- 输出依赖:线程 *A* 和进程 *B* 均对共享数据进行了写.

定义 3. 采用基础 SPMD 方法转换 *C* 到 *L* 时,不能被保留的线程间数据依赖称为关键线程间依赖.

在上述定义的基础上,隐式同步的检测可以描述为判断 *C* 中是否存在 *L* 不能保留的关键线程间依赖,因为不能保留的线程间依赖不能被基础 SPMD 正确翻译.为方便分析,我们可以把检测方向逆转,转变为判断 *L* 向 *C* 转换的等价性,这样,串行循环的形式可更方便地被用来进行数据依赖分析,即检测 *L* 中是否存在数据依赖使其在 *C* 中不被保留.*C* 和 *L* 的形式及等价性判断如图 2 所示.

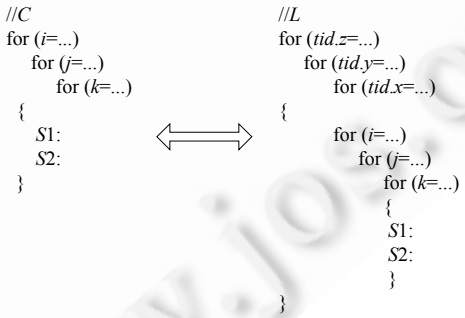


Fig.2 Illustration of data dependence detecting

图 2 数据依赖检测示意

定理 1. 假设线程循环 *L* 中两条语句 *S*1 和 *S*2,*S*1 的序号小于 *S*2,*S*1 到 *S*2 的依赖方向向量为 *V*.*S*1 到 *S*2 的数据依赖在 *C* 中可以保留,当且仅当下述条件中有 1 个满足:

(1) *V*(1)=*V*(2)=*V*(3)=0;

(2) *V*(1:3)最左非 0 元素符号是“<”,且 *V*(4:|*V*|)是 0;

(3) $V(1:3)$中第1个非0元素的符号等于$V(4:|V|)$中的第1个非0元素的符号且不是“*”.

证明:对第1个条件,线程循环L中在$S1$和$S2$之间不存在对$tid.z$,$tid.y$,$tid.x$这3个迭代变量的数据依赖,那么$S1$和$S2$的依赖是线程内部的依赖,C在并行执行时就不存在线程之间的数据依赖.两条语句间的隐式同步的忽略不影响C的正确执行.

第2个条件,$V(4:|V|)$是0,即C中不存在循环携带依赖.同时,$V(1:3)$中最左非0元素符号是“<”,表明不存在反向依赖,C在执行时不会出现这样的情况:依赖的源点在$S2$而汇点在$S1$.因为C在GPU上以SIMD方式执行时,依赖的源点必定在汇点之前,所以$V(1:3)$中的依赖可以保留.

第3个条件,$V(4:|V|)$存在非0的元素,表明C中的一些循环在$S1$和$S2$之间携带数据依赖,此时,C在GPU上执行时依赖的方向取决于$V(4:|V|)$第1个非0的元素.而在L中,$V(1:3)$中第1个非0元素决定$S1$和$S2$之间的依赖方向.因此,第3个条件确保了L和C中相同的依赖方向,因此可以被保留.

以上证明了该定理的充分条件,必要条件的证明可以通过列举不属于这3种条件的各种情况来证明.

第1种情况,$V(1:3)$最左非0元素符号是“>”或“*”,且$V(4:|V|)$是0.此时,C中不存在循环携带依赖,但$V(1:3)$中存在反向依赖,即依赖的源点在$S2$而汇点在$S1$.当C在GPU上执行时,依赖的源点必定在汇点之前,因此该依赖不能被保留.

第2种情况,$V(1:3)$中第1个非0元素的符号不等于$V(4:|V|)$中的第1个非0元素的符号.此时,L和C中存在相反的依赖方向,相应的数据依赖不能被保留.证毕. □

该定理不区分依赖的种类以及语句的位置.无论是真依赖、反依赖还是输出依赖,均适用于该定理;同时,不区分发生语句的位置,即使依赖的语句发生在与线程相关的分歧分支中.

另外,线程束的结构也可以帮助降低L的迭代层数.线程束内的线程是连续的,线程ID依次递增,第1个束包含线程0.线程索引与线程ID直接相关:对于一维的块,它们相同;对于二维长度为(Dx,Dy)的块,线程索引为(x,y)的线程ID是$(x+yDx)$;对于三维长度为(Dx,Dy,Dz)的块,索引为(x,y,z)的线程ID为$(x+yDx+zDxDy)$.因此,进行依赖符号向量判定时可针对线程索引和线程ID的关系做一些调整,在一维的情况下,令L中$V(1)=V(2)=0$;二维的情况下$V(1)=0$.这个假定使L迭代层数降低,使判定过程变得更为快速、精确.

值得注意的是,线程循环L的构造方式是升序迭代三维线程索引$tid.z$,$tid.y$,$tid.x$,但由于C的并行执行的特性,L的构造可以在保留依赖的情况下采用其他的排序如逆序构造,这种循环重排序的方法提供了处理关键线程间依赖的优化手段.对于一些类型的关键线程间依赖,典型的如$V(1:3)$中最左非0符号是“>”而$V(4:|n|)$最左非0符号是“<”的情况,逆序循环索引中$V(1:3)$中“>”对应的索引,可使该关键线程间依赖变成可保留依赖.因此,对检测出的数据依赖进行分类,以便随后的处理中针对不同的依赖情况采用不同的处理算法.

对检测出来的数据依赖分为4类:

- 线程内部依赖I(intro-thread),是指检测定理中符合第1个条件的依赖.
- 可保留依赖H(hold),是指检测定理中符合条件(2)、条件(3)的数据依赖.
- 逆转可保留依赖R(reverse),是指逆转迭代方向后可保留的数据依赖.符合这种类型的依赖主要有两类:$V(1:3)$中最左非0符号是“>”而$V(4:|n|)$最左非0符号是“<”,或$V(1:3)$中最左非0符号是“<”而$V(4:|n|)$最左非0符号是“>”.在这两种情况下,C中的循环携带依赖由$V(4:|n|)$最左非0符号控制,它与$V(1:3)$中第1个非0元素的方向相反决定了$S1$和$S2$之间的依赖方向是反向的,因此,逆序构造相应循环索引的L后,使L和C中的依赖方向相同,该依赖可被保留.
- 关键依赖K(key),是指除上述3种类型外剩余的数据依赖.

3 隐式同步的处理

3.1 简单扩展的不足

对于隐式同步,最简单的解决方案是直接扩展基础SPMD翻译中显式同步的处理方法,即相当于在连续的指令间设置一个显式的同步,为CUDA程序的每条语句都构造循环.然后,利用显式同步的方法进行处理.该方

案对于图 1(c)中的代码可以正确处理.但是对处于复杂控制结构中的隐式同步,这种方法仍然会引起错误.

如图 3(a)中所展示的代码,若有 2 个线程并行执行,那么在 *sdata*1 数组发生了数据依赖,两个线程在第 5 条语句中对 *sdata*1 的读分别是另外的线程在第 1 条语句对 *sdata*1 的写,GPU 上执行完后的结果是 *sdata*1[2]=[1,1], *sdata*2[2]=[0,−1].若直接采用显式同步的处理方法,在第 1 条、第 2 条和第 3 条语句之间插入显式同步,利用显式同步的处理方法得到的串行代码如图 3(b)所示,其执行结果是 *sdata*1[2]=[2,1],*sdata*2[2]=[0,−2],结果显然是错误的.这是因为隐式同步是线程束内的部分线程参与的同步,简单扩展的方法会使所有线程束的线程都进行同步,因此不正确.所以,针对控制流和数据流混合的情况,需要为线程添加掩码来标识线程是否活跃.

```
//shared array sdata1[2]=[0,−1],
  sdata2[2]=[2,0];
   L1:
1     sdata1[|tid−1|]++;
2     sdata2[tid]−−;
3     if (sdata1[tid]>0 &&
4         sdata2[tid]>0 )
5         goto L1;
```

(a)

```
L1:
  thread_loop{
     sdata1[|tid−1|]++;
  }
  thread_loop{
     sdata2[tid]−−;
  }
  thread_loop{
     if (sdata1[tid]>0 &&
        sdata2[tid]>0)
        goto L1;
  }
```

(b)

Fig.3 Defect of simple extension

图 3 简单扩展的不足

3.2 基础SPMD翻译的修正

由于直接扩展基础 SPMD 翻译的方法存在不足,所以需要对简单扩展后的程序进行修正.修正的方法主要是在简单扩展方法应用后,在分歧[11]的分支中应用掩码来标识活跃的线程.

在分歧分支的源点设置掩码标志 mask[WARPSIZE],每一位标志着线程是否活跃.

在分歧分支中,分支代码仅对活跃的线程执行;在分歧分支嵌套的情况下,需要设置多个掩码标志,条件分支的执行依赖于多个掩码标志的结果.

对应图 3(a)中的例子,掩码处理结果如图 4(a)所示,掩码处理的方法同样应用在随后的优化算法中.

```
   mask[2]=[1,1];
L1:
   thread_loop{
      if (mask[tid]==1)   sdata1[|tid−1|]++;
      }
   thread_loop{
      if (mask[tid]==1)   sdata2[tid]−−;
      }
   thread_loop{
      if (mask[tid]==1)   mask[tid]=0;
      }
   thread_loop{
      if (sdata1[tid]>0 &&
         sdata2[tid]>0)
         mask[tid]=1;
   }
   if (mask[0]||mask[1])
         goto L1;
```

(a)

```
   mask[2]=[1,1];
L1:
   thread_loop{
      if (mask[tid]==1)
      {
      sdata1[|tid−1|]++;
      sdata2[tid]−−;
      mask[tid]=0;
      }
   }
   thread_loop{
      if (sdata1[tid]>0 &&
         sdata2[tid]>0)
         mask[tid]=1;
         }
   }
   if (mask[0]||mask[1])
         goto L1;
```

(b)

Fig.4 Comparison between modified SPMD translation algorithm and optimized algorithm

图 4 修正的 SPMD 翻译算法与优化算法的对比

但是,修正的方法产生了众多小规模的循环,虽然编译器的优化能够合并一些循环,但编译器优化毕竟不能处理好所有的循环,效率仍受影响.

3.3 优化的处理算法

经过以上分析可知,在基础SPMD翻译的基础上,隐式同步的优化处理面临两个主要问题:一是分歧分支的影响.分歧分支是与线程索引变量相关的条件分支,导致不同的线程执行不同的代码路径.这虽然不影响 L 中关键隐式同步的判断,但在对其处理时需要判断线程迭代是否活跃;二是一些关键隐式同步可以通过重排序的方法转换为可保留的数据依赖,这些依赖就不需要更多额外的处理,因此需要尽可能地识别可逆转保留依赖.

针对上述两个方面的问题,本文设计了显式与隐式同步统一处理及优化算法.首先,利用显式同步进行代码分割以构造线程循环;然后,在分割后的代码中检测隐式同步并分类,进而调用隐式同步处理算法处理;最后,给分歧分支中的代码加入掩码.优化后的线程循环构造算法如图5所示.

```
//功能:线程循环构造算法
//输入:kernel函数C.
//输出:转换后的线程循环代码集Ls.
procedure Thread_Loop_Construction(C)
//处理显式同步,利用显式同步语句对C进行代码分割并构造线程循环
线性扫描kernel函数,标记控制结构范围和显式同步语句ES(函数头和尾是默认的两个显式同步点);
连续两条显式同步语句ES之间构成一个线程循环L;
for each ES do begin
    if ES被包含在控制结构中, then 将它对控制结构进行深度分裂;   //控制结构处于ES前后的多个线程循环L中
end
//对隐式同步处理,进一步代码分裂
设隐式同步语句集Is为空;
for each L do begin
    for each statement S do begin
        if S包含共享变量的引用, then begin
            计算它的序号;
            计算当前代码片段Is中每条语句S'与S之间的依赖符号向量V;
            对V进行分析,如果不属于I,则将S归入H,R和K等3类;
        end
        当前语句入Is;
    end
按照依赖分类,属于H的是可保留的隐式依赖,不用额外处理.如果R或K不为空,则调用隐式同步处理算法;
对处于分歧分支中的隐式同步,加入掩码进行处理;
end
end Thread_Loop_Construction
```

Fig.5 Algorithm of thread loop construction

图5 线程循环构造算法

对产生的隐式同步语句集 Is 调用隐式同步处理算法进行处理,对连续的可保留依赖构造升序线程循环,连续的逆转可保留依赖构造降序线程循环.关键线程间依赖涉及到的语句逐条构造线程循环.隐式同步处理算法如图6所示.

对处于分歧分支中的关键数据依赖,采用增加线程掩码的方式进行处理.处理方法和基础SPMD修正时的相同.

相对于修正的基础SPMD翻译,优化算法仅对隐式同步语句进行分析,尽可能地保留连续的语句在一个线程循环中,虽然增加了分析时间,但产生的线程循环数量小了很多,可以有效地减轻翻译后编译器进一步优化的压力,提高程序运行速度.

应用算法的一个实例如图 4(b)所示.该例子中由于不存在逆转可保留依赖,两个线程循环均是升序循环.与图 4(a)中修正的基础 SPMD 翻译方法相比,优化算法通过识别隐式同步语句之间的依赖关系,把不存在依赖的连续语句放在一个线程循环中,降低了线程循环的数量.

```
//功能:隐式同步处理算法
//输入:隐式同步语句集Is;
//输出:处理隐式同步后的线程循环代码.
//Sa是升序索引构造线程循环集
//Sd是降序索引构造线程循环集
procedure Implicit_synchronization_Treatment(Is)
Sa=Sd=NULL;
for each S in Is do begin
    if S仅属于R,then Sd加入S;
    if S仅属于H,then Sa加入S;
    else S属于K,或同时属于B和R, then begin
        为Sa的语句建立升序循环;
        为Sb的语句建立降序循环;
        为当前语句建立升序循环;
    end
end
end Implicit_synchronization_Treatment
```

Fig.6 Algorithm of implicit synchronization treatment

图 6 隐式同步处理算法

4 实验测试

实验的目的在于验证隐式同步的检测及处理算法的正确性及效率.正确性测试主要测试算法应用在包含隐式同步的测试用例的正确性,效率测试主要比较优化的处理方法与修正的基础 SPMD 翻译的效率.

本文所用的测试用例选取 CUDA SDK2.3,包括 *particles*,*radixSort*,*reduction*,*smokeParticles* 以及 *threadFenceReduction*,见表 1.所有测试用例均包含隐式同步.它们可以验证隐式同步检测算法的正确性及效率.

Table 1 Test cases

表 1 测试用例

测试用例	应用程序	依赖种类
particles	Simulating physical systems implementation on GPU	*K*
radixSort	Parallel radix sort implemented in C for CUDA	*R*,*K*
reduction	Parallel reduction in a single kernel	*K*
smokeParticles	Adding volumetric shadowing to particle systems	*K*
threadFenceReduction	Parallel reduction in two or more kernels	*K*

测试环境选取 Intel 处理器以及 ICC 编译器,见表 2.所有的编译都采用最高的优化级别,以充分利用处理器的向量化部件执行编译器产生的向量化指令.

Table 2 Test circumstances

表 2 测试环境

CPU	Intel Pentium Dual CPU E2200 @2.2.0Ghz
Memory	1GB DDR-1333 DRAM
CPU compiler	ICC-11.0
GPU compiler	NVCC-2.3
OS	32-bit Fedora 10

每个测试用例都分为 3 个版本,依次是:

- 基础 SPMD 翻译版本.采用源代码级的基础 SPMD 翻译平台翻译产生的程序,这个版本的程序用来检验基础SPMD的正确性缺陷,其执行时间也作为基准时间来比较修正的SPMD翻译和本文的优化算法的执行时间.
- 修正的基础 SPMD 翻译版本.对基础 SPMD 进行修正,在每两条连续的语句间插入隐含的同步语句,然后用显式同步的方法处理,对处于分歧分支的语句插入掩码标识线程是否活跃.
- 采用本文优化算法的翻译版本.使用本文依赖检测及优化处理算法翻译产生的程序.

测试结果表明,5 个测试用例若使用基础 SPMD 翻译方法,产生的结果是不正确的,如 *reduction*,使用基础SPMD翻译产生的结果是210 864,而GPU上直接运行的结果以及采用修正的SPMD翻译方法和本文的优化算法的结果是 133 784 454.其他例子的结果见表 3.这验证了基础 SPMD 翻译的缺陷以及本文处理算法的正确性.

Table 3 Test results comparison

表 3 测试结果对比

测试用例	GPU 执行结果	基础 SPMD 翻译执行	修正的 SPMD 翻译执行结果	本文优化算法执行结果
particles	PASSED!	FAILED!	PASSED!	PASSED!
radixSort	正确排序	排序不正确	正确排序	正确排序
reduction	133 784 454	210 864	133 784 454	133 784 454
smokeParticles	PASSED!	PASSED!	PASSED!	PASSED!
threadFenceReduction	0.062 298 238 277	0.068 984 265 032 2 (误差过大,不正确)	0.062 298 242 003	0.062 298 242 003

同时,3 个版本翻译效率的测试结果如图 7 所示,翻译效率与翻译时间成反比.以基础 SPMD 翻译方法为基准,由图中可以看出,修正的 SPMD 翻译方法在翻译时间上增幅是最大的,显然,这种方法对所有语句进行的处理,包括插入同步语句、构造线程循环是很费时的.本文的优化算法在翻译时间上没有较大的增加,因为仅对引用共享变量的语句进行检测,其他访问局部变量等语句无须考虑.修正的 SPMD 翻译方法和本文的优化算法在分歧分支的处理方面采用相同的方法,处理时间是相同的.因此,优化算法的翻译效率优势主要体现在只对包含隐式同步的语句进行处理.

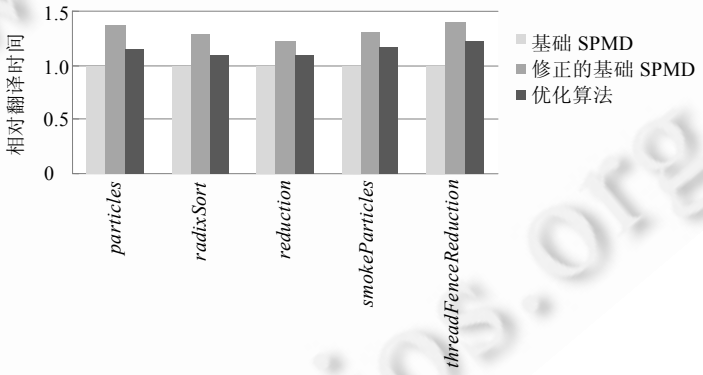


Fig.7 Comparison of translation time

图 7 翻译时间比较

程序执行时间的测试结果如图 8 所示.从图中可以分析出,以基础 SPMD 翻译方法为基准,修正的 SPMD 翻译方法在执行时间上有了显著的增加.这是因为修正的方法产生了数量众多的小规模线程循环,它们全部依赖编译器的循环融合优化技术,而编译器优化的程度是有限的,因此执行时间会有大幅度的增加.相对于基础SPMD 翻译方法,本文的优化算法在执行时间上略有增加,主要原因在于:算法中仅对引用共享数组变量的语句进行分析,这样不会产生很大规模的线程循环;同时,经过重排序所做的优化最大程度地降低了线程循环的数量.因此,5 个测试用例的平均执行时间即使与基础翻译相比时间有少量的增加,但是与修正翻译相比效率却较

大的提升.另外,优化的算法对可以经过重排序变换保留的数据依赖进行了优化,可使部分测试用例的执行时间得到进一步的提升.如 *radixSort*,对其优化后,时间的增加是很少的.

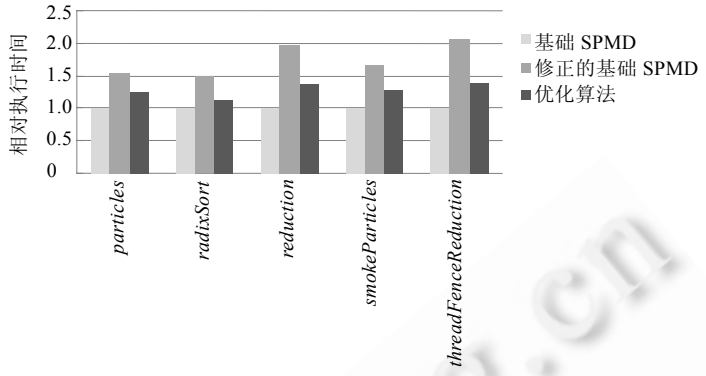


Fig.8 Comparison of execution time

图 8 执行时间比较

5 结束语

本文对细粒度 SPMD 程序中的显式同步和隐式同步进行了分析,针对当前基础 SPMD 翻译方法在翻译隐式同步时的正确性缺陷,提出了基于依赖分析的隐式同步检测方法,给出了可保留依赖判定定理.在依赖分析的基础上,对隐式同步设计了相应的处理算法,并利用重排序的手段对线程循环进行了优化.实验测试结果表明,本文的检测定理及处理算法可以正确而有效地应对细粒度 SPMD 程序翻译时的同步处理.

本文所做的工作系统地分析了细粒度 SPMD 程序中的同步情况,因此可以扩展到 CUDA 程序向其他设备,如异构多核、众核的移植研究中去,进而为并行系统间的程序移植时同步的处理提供思路.

致谢 在此,向对本文研究工作提供基金支持的单位和评阅本文的审稿专家表示衷心的感谢,向为本文研究工作提供基础和研究平台的前辈致敬.

References:

[1] Auerbach J, Bacon DF, Cheng P, Rabbah R. Lime: A Javacompatible and synthesizable language for heterogeneous architectures. In: Proc. of the ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages and Applications. ACM Press, 2010. 89−108. [doi: 10.1145/1932682.1869469]

[2] OpenCL. http://www.khronos.org/opencl/

[3] Stratton JA, Stone SS, Hwu WMW. MCUDA: An effective implementation of CUDA kernels for multi-core CPUs. In: Proc. of the 21st Int'l Workshop on Languages and Compilers for Parallel Computing. Springer-Verlag, 2008. 16−30. [doi: 10.1007/978-3-540-89740-8_2]

[4] Diamos G, Kerr A, Kesavan M. Ocelot, a dynamic optimization framework for bulk-synchronous applications in heterogeneous system. In: Proc. of the 19th Int'l Conf. on Parallel Architectures and Compilation Techniques. ACM Press, 2010. 353−364. [doi: 10.1145/1854273.1854318]

[5] NVIDIA Corporation. NVIDIA CUDA Programming Guide. Version 2.3, NVIDIA Corporation, 2009. 71−75.

[6] Wang PY, Chen YJ, Shen HH, Chen TS, Zhang H. Memory consistency verification of chip multi-processor. Ruan Jian Xue Bao/ Journal of Software, 2010,21(4):863−874 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3705.htm [doi: 10.3724/SP.J.1001.2010.03705]

[7] Stratton J, Grover V, Marathe J, Aarts B, Murphy M, Hu Z, Hwu WMW. Efficient compilation of fine-grained spmd-threaded programs for multicore CPUs. In: Proc. of the 2010 Int'l Symp. on Code Generation and Optimization. ACM Press, 2010. 111−119. [doi: 10.1145/1772954.1772971]

[8] Guo ZY, Zhang EZ, Shen XP. Correctly treating synchronizations in compiling fine-grained spmd-threaded programs for CPU. In: Proc. of the 20th Int'l Conf. on Parallel Architectures and Compilation Techniques. IEEE Computer Society, 2011. 310−319. [doi: 10.1109/PACT.2011.62]

[9] Guo ZY, Shen XP. Fine-Grained treatment to synchronizations in GPU-to-CPU translation. Technical Report, WM-CS-2011-02, 2011.

[10] Wu B, Zhang EZ, Shen XP. Enhancing data locality for dynamic simulations through asynchronous data transformations and adaptive control. In: Proc. of the 20th Int'l Conf. on Parallel Architecture and Compilation Techniques. IEEE Computer Society, 2011. 243−252. [doi: 10.1109/PACT.2011.56]

[11] Zhang EZ, Jiang YL, Guo ZY, Tian K, Shen XP. On-the-Fly elimination of dynamic irregularities for GPU computing. In: Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. ACM Press, 2011. 369−380. [doi: 10.1145/1950365.1950408]

[12] Luk CK, Hong S, Kim H. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: Proc. of the Int'l Symp. on Microarchitecture. ACM Press, 2009. 45−55. [doi: 10.1145/1669112.1669121]

[13] Kirk DB, Hwu WMW. Programming Massively Parallel Processors. Elsevier Inc., 2010. 39−42.

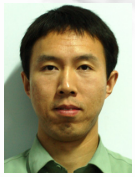
[14] Aiken A, Gay D. Barrier inference. In: Proc. of the 25th ACM Symp. on Principles of Programming Languages. IEEE Press, 1998. 342−354. [doi: 10.1145/268946.268974]

[15] NVIDIA Corporation. Getting Started With CUDA SDK Samples. NVIDIA Corporation, 2012. 2−5.

[16] Allen R, Kennedy K. Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Elsevier Science, 2001. 34−39.

附中文参考文献:

[6] 王朋宇,陈云霁,沈海华,陈天石,张珩.片上多核处理器存储一致性验证.软件学报,2010,21(4):863−874. http://www.jos.org.cn/1000-9825/3705.htm [doi: 10.3724/SP.J.1001.2010.03705]



岳峰(1985－),男,山西长治人,博士生,CCF 学生会员,主要研究领域为动态编译,系统虚拟化.
E-mail: firstchoiceyf@163.com

赵荣彩(1957－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为并行与高性能计算.
E-mail: rczhao126@126.com

庞建民(1964－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为高性能计算,信息安全.
E-mail: jianmin_pang@hotmail.com