·先进计算与数据处理· 文章编号：1000-3428(2021)08-0131-09 文献标志码：A 中图分类号：TP311

面向异构架构的传递闭包并行算法

肖 汉[1,3]，郭宝云[2]，李彩林[2]，周清雷[3]

（1.郑州师范学院 信息科学与技术学院，郑州 450044；2.山东理工大学 建筑工程学院，山东 淄博 255000；
3.郑州大学 信息工程学院，郑州 450001）

摘 要：传统求图传递闭包的方法存在计算量大与计算时间长的问题。为加快处理大数据量的传递闭包算法的计算速度，结合算法密集计算和开放式计算语言(OpenCL)框架的特征，采用本地存储器优化的并行子矩阵乘和分块的矩阵乘并行计算，提出一种基于OpenCL的传递闭包并行算法。利用本地存储器优化的并行子矩阵乘算法来优化计算步骤，提高图形处理器(GPU)的存储器利用率，降低数据获取延迟。通过分块矩阵乘并行计算算法实现大数据量的矩阵乘，提高GPU计算核心的利用率。数据结果表明，与CPU串行算法、基于开放多处理的并行算法和基于统一设备计算架构的并行算法相比，传递闭包并行算法在OpenCL架构下NVIDIA GeForce GTX 1070计算平台上分别获得了593.14倍、208.62倍和1.05倍的加速比。

关键词：矩阵乘；传递闭包；图形处理器；开放式计算语言；并行算法

开放科学(资源服务)标志码(OSID)：

中文引用格式：肖汉，郭宝云，李彩林，等. 面向异构架构的传递闭包并行算法[J]. 计算机工程，2021，47(8)：131-139.

英文引用格式：XIAO H，GUO B Y，LI C L，et al. Parallel transitive closure algorithm for heterogeneous architecture[J]. Computer Engineering，2021，47(8)：131-139.

Parallel Transitive Closure Algorithm for Heterogeneous Architecture

XIAO Han[1,3]，GUO Baoyun[2]，LI Cailin[2]，ZHOU Qinglei[3]

（1.School of Information Science and Technology，Zhengzhou Normal University，Zhengzhou 450044，China；
2.School of Civil and Architectural Engineering，Shandong University of Technology，Zibo，Shandong 255000，China；
3.School of Information Engineering，Zhengzhou University，Zhengzhou 450001，China）

【Abstract】The traditional method for obtaining the transitive closure of the graphs faces the large amount of calculation and long calculation time. In order to improve the computing speed of the transitive closure algorithm for dealing with large amounts of data，an Open Computing Language（OpenCL）-based parallel algorithm for transitive closure is proposed. The algorithm combines the characteristics of algorithm-intensive computation and OpenCL architecture，and uses the parallel submatrix multiplication and block matrix multiplication optimized by local memory for parallel computing. The parallel submatrix multiplication algorithm is used to optimize the computational steps，improves the memory utilization of the Graphic Processing Unit（GPU），and reduces the data acquisition delay. The parallel block matrix multiplication algorithm is used to implement matrix multiplication involving large amounts of data，and improve the utilization of the GPU computing cores. The experimental results show that compared with the sequential CPU-based algorithm，parallel algorithm based on Open Multi-Processing，and parallel algorithm based on Compute Unified Device Architecture（CUDA），the proposed parallel transitive closure algorithm provides a 593.14 times，208.62 times and 1.05 times speedup respectively on the NVIDIA GeForce GTX 1070 computing platform with OpenCL architecture.

【Key words】matrix multiplication；transitive closure；Graphic Processing Unit（GPU）；Open Computing Language（OpenCL）；parallel algorithm

DOI：10. 19678/j. issn. 1000-3428. 0058071

基金项目：国家自然科学基金(41601496，41701525，61572444)；山东省自然科学基金(ZR2017LD002)；山东省重点研发计划项目(2018GGX106002)。

作者简介：肖 汉(1970—)，男，教授、博士后，主研方向为大规模并行算法、遥感大数据并行处理；郭宝云(通信作者)，讲师、博士；李彩林，副教授、博士；周清雷，教授、博士、博士生导师。

收稿日期：2020-04-15 **修回日期：**2020-07-16 **E-mail：**xiaohan70@163.com

0 概述

传递闭包运算在图论、网络、计算机形式语言、语法分析以及开关电路中的故障检测和诊断领域都有着广泛的应用价值[1-2]。根据定义,关系传递闭包的计算是通过多次进行集合复合运算完成,运算量很大。同时,假如二元关系在某种情况下发生了改变,其中的某些序偶增加或减少,需要按照原方法将变化的二元关系重新计算来得到新关系的传递闭包,运算量则进一步增大[3-5]。这样容易造成大量数据无法实时处理,最终使整个应用系统处理的时间增加,因此如何快速有效地处理传递闭包问题成为了一个急需解决的问题[6-7]。

开展传统的利用CPU集群的高性能计算是解决大规模科学计算问题的常用方法,然而集群的并行计算性能对于CPU的更新换代的依赖性很大。由于CPU芯片单位面积内的晶体管集成度越来越高,散热和能耗问题凸显,致使提升CPU的速度放缓,发展陷入瓶颈[8-10]。为了更快地增强计算能力,计算机硬件设计的异构化的趋势越发明显[11-12]。由若干不同架构的CPU处理器和协处理器共同工作,通用计算处理器与多个加速器设备互连构成的异构计算系统逐渐成为主流[13-14]。

开放式计算语言(Open Computing Language,OpenCL)是一个面向异构硬件平台的、免费的、开放的行业标准。遵循OpenCL规范的不同架构的硬件,提供需要的编译和运行平台,就能够在OpenCL平台上开发普适的应用系统,为多核CPU、CPU+GPU、DSP和多GPU等异构计算提供良好的研发平台[15-16]。

本文基于开放式计算语言平台,提出一种基于CPU+GPU的高效传递闭包并行算法,并采用具有可移植性的OpenCL架构来实现该算法。对在不同数据集下和不同体系结构下的算法和加速比进行分析。

1 相关研究

近年来,很多学者对传递闭包运算进行了研究。文献[17]用一阶有界传递闭包模糊逻辑来刻画模糊有穷自动机。文献[18]研究了稠密图条件下采用X-Hop方法,对传递闭包进行高压缩比存储和有效查询的算法。文献[19]提出改进的传递闭包求解方法,并在传递闭包改进的求解方式基础上,设计了传递闭包的增量式更新方法。文献[20]证明只要函数在Jensen的J层次结构的某个多项式级别中是统一可行的,则相对于函数参数的传递闭包,其在任意集上是安全递归的。文献[21]提出改进的Floyd-Warshall算法,其中最耗时的部分(描述程序循环中自相关的传递闭包)是通过依赖距离向量计算,减少了传递闭包计算时间。文献[22]基于程序依赖图的传递闭包,提出一种在瓦片内生成具有任意顺序循环的并行代码的方法。文献[23]使用循环嵌套依赖图的传递闭包来执行原始矩形瓦片的校正,生成并行无同步代码。文献[24]通过应用依赖图的传递闭包,提出生成Nussinov RNA折叠算法的并行代码的加速因子。文献[25]通过MPI并行化实现了Warshall方法,进而快速求取了关系R的传递闭包R^+。

文献[26]通过向量化方法将循环结构并行化,实现了传递闭包并行算法。文献[27]在二叉树并行计算模型上,实现了一种基于MPI的传递闭包并行算法。文献[28]通过实现传递闭包并行算法,提高了在图形和高维数据中挖掘中心对象算法的收敛性。文献[29]利用MPI提出了基于VLSI的传递闭包并行算法。文献[30]通过合并Dijkstra单源最短路径方法中的贪婪技术的特征和传递闭包属性来找到所有点对最短路径,并在MapReduce平台上实现了ex-FTCD算法。

综上所述,目前大部分研究工作是通过优化算法本身从而实现对传递闭包算法的快速计算,有些则利用传统的向量化和CPU集群的MPI并行计算方式设计传递闭包算法。但是,性能加速效果在这些相关研究中表现的均不明显。同时,算法研究和平台设计局限于单一类型,对于多算法和多平台的系统性能评估不多。本文将根据传递闭包算法特性和OpenCL架构的特征,研究异构协同计算下的传递闭包并行算法,以及在多种计算平台上算法的性能移植。

2 传递闭包算法

2.1 OpenCL异构编程模型

OpenCL是一种面向开放的、通用并行编程的、跨平台的行业标准,软件开发人员可以方便地将CPU、GPU和其他各类计算设备接入系统计算[31-32]。OpenCL标准是编程语言与编程框架的集合体,人们可以基于硬件抽象层API和面向数据的异构编程环境进行OpenCL系统的开发和优化应用。OpenCL框架主要由OpenCL平台层、OpenCL运行时环境和OpenCL编译器3个部分组成[33-35]。平台层允许用户收集可用的OpenCL设备信息。开发者可以查询特定设备的详细资料,比如缓存大小、存储器结构、核心数量等。OpenCL Runtime提供了管理设备存储器、运行kernel、在设备与主机之间传输数据等一系列API[36-39]。OpenCL编译器创建包含OpenCL kernel的可执行程序,把kernel编译成设备能够识别的代码。

2.2 算法定义

图的传递闭包可以采用布尔矩阵的平方法计算。首先假定$\boldsymbol{A}$是一个m点有向图的$m\times m$的布尔邻接矩阵,当且仅当有向图中从顶点i到顶点j之间

有一条边时，矩阵元素 a_{ij} 为1。然后利用矩阵乘法对布尔矩阵的传递闭包 $\boldsymbol{A}^{+}$ 求解[40-41]。设 $\boldsymbol{I}$ 是单位矩阵，大小为 $m\times m$ 的关系矩阵为 $\boldsymbol{B}=\boldsymbol{A}\cup\boldsymbol{I}$。使矩阵 $\boldsymbol{B}$ 的第 i 行上的元素与矩阵 $\boldsymbol{B}$ 的第 j 列上的元素按顺序分别相乘再相加，得到新的关系矩阵 $\boldsymbol{B}$ 的第 i 行第 j 列的元素（关系矩阵 $\boldsymbol{B}$ 在不断更新），即 $\boldsymbol{B}$ 的定义如下：

$$\boldsymbol{B}_{ij}=\sum_{k=0}^{m-1}\boldsymbol{B}_{ik}\,\boldsymbol{B}_{kj},\ 0\leqslant i,j\leqslant m-1 \tag{1}$$

得到新的关系矩阵 $\boldsymbol{B}$ 重复上一步进行循环，即依次计算 $\boldsymbol{B}^{2^1},\boldsymbol{B}^{2^2},\cdots$，直到 $\boldsymbol{B}^{2^p}=\boldsymbol{B}^{2^{p+1}}$，即执行 $p<\log_a m$ 次，得到布尔矩阵的传递闭包 $\boldsymbol{A}^{+}$[42-44]。由此可知，算法的时间复杂性在最坏的情况下为 $O\left(m^3\log_a m\right)$，当 m 非常大时，该算法运算将非常耗时[45]。

2.3　算法的并行特征分析

算法的可并行性高低与算法自身存在的数据依赖性有关。如果算法运算前后依赖性越强，则算法的可并行性就越低，反之，如果算法运算前后依赖性越弱，则算法的可并行性就越高，算法并行化后进行并行计算的性能就会越好。图1所示是一个有向图的传递闭包算法的可并行性分析。

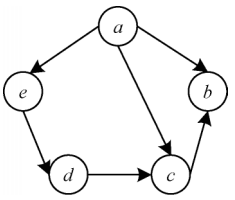

图1　有向图

Fig.1　Directed graph

根据图1的5个顶点的有向图表示出布尔矩阵 $\boldsymbol{A}_{ij}$，计算布尔矩阵 $\boldsymbol{A}_{ij}$ 的闭包 $\left(\boldsymbol{A}_{ij}\right)^{\div}$ 过程如下：

$$\boldsymbol{B}_{ij}=\boldsymbol{A}_{ij}\cup\boldsymbol{I}_{ij}=\begin{pmatrix}0&1&1&0&1\\0&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\end{pmatrix}\cup\begin{pmatrix}1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\end{pmatrix}=\begin{pmatrix}1&1&1&0&1\\0&1&0&0&0\\0&1&1&0&0\\0&0&1&1&0\\0&0&0&1&1\end{pmatrix}\Rightarrow$$

$$(\boldsymbol{B}_{ij})^{2^1}=\begin{pmatrix}1&1&1&1&1\\0&1&0&0&0\\0&1&1&0&0\\0&1&1&1&0\\0&0&1&1&1\end{pmatrix}\Rightarrow(\boldsymbol{B}_{ij})^{2^2}=\begin{pmatrix}1&1&1&1&1\\0&1&0&0&0\\0&1&1&0&0\\0&1&1&1&0\\0&1&1&1&1\end{pmatrix}=(\boldsymbol{A}_{ij})^{\div}$$

在 $(\boldsymbol{A}_{ij})^{\div}$ 的计算过程中可以发现，每一计算步骤中的任意一个元素的计算过程与其他元素计算互不影响，相互之间并没有依赖性。因此，可以在计算某一个元素值时，同时对其他元素值进行运算。结合OpenCL的计算模型，将每个元素的计算过程放入工作项中，每个工作项计算得出相应元素的结果。若矩阵中每个元素计算结束，则本次计算结束，如果需要继续迭代，则再次重复以上过程。

3　传递闭包算法并行映射模型

3.1　并行算法

有向图布尔矩阵 A 的传递闭包可以利用 $\boldsymbol{B}=(\boldsymbol{A}+\boldsymbol{I})$ 的自乘 $\log_a m$ 次得到。设定工作空间中的工作组和工作项排成 $m\times m$ 的二维阵列，即其坐标为 (tx,ty)。每个工作组用数组 as 和数组 bs 存储矩阵 $\boldsymbol{B}$ 中相应子矩阵，Pvalue保存的是每次子矩阵计算之后得到的值，数组 $\boldsymbol{C}$ 为每次计算完成之后最终数据。传递闭包并行算法描述如算法1所示。

算法　SIMT模型上的传递闭包并行算法

输入　有向图的布尔矩阵 $\boldsymbol{A}_{m\times m}$

输出　$\boldsymbol{A}_{m\times m}$ 的传递闭包 $C_{m\times m}$

```
begin
1./*形成单位矩阵 */
2.for j = 0 to m–1 par-do A(0, j, j) ← 1 end for
3./*将A的内容复制到B*/
4.for j = 0 to m–1 par-do
5.  for k = 0 to m–1 par-do
6.   B(0, j, k) ← A(0, j, k)
7.  end for
8.end for
9./*利用更新后的B矩阵相乘求A的传递闭包*/
10.for i = 1 to⌈log_a(n–1)⌉do
11./*将B矩阵中的相应子矩阵预取至快速存储
   器as和bs中*/
12. for j = 0 to BLOCK_SIZE–1 par-do
13.   for k = 0 to BLOCK_SIZE–1 par-do
14.    as(0, j, k) ← B(0, j, k), bs(0, j, k) ← B(0, k, j)
15.   end for
16. end for
17./*矩阵乘法*/
18. for a = aBegin to aEnd, b = bBegin to bEnd do
19.   Pvalue = 0
20.   for k = 0 to BLOCK_SIZE par-do
21.    Pvalue = Pvalue + as(ty, k) × bs(k, tx)
22.   end for
23.   a = a + BLOCK_SIZE, b = b + BLOCK_SIZE
24. end for
25.C[c + wB × ty + tx] = Pvalue
26.for j = 0 to m–1 par-do
27.    for k = 0 to m–1 par-do
```

```
28.      B(0, j, k) ← C(0, j, k)
29.    end for
30.  end for
31.end for
end
```

3.2 并行算法整体并行化思路

基于OpenCL的传递闭包并行执行流程如图2所示。

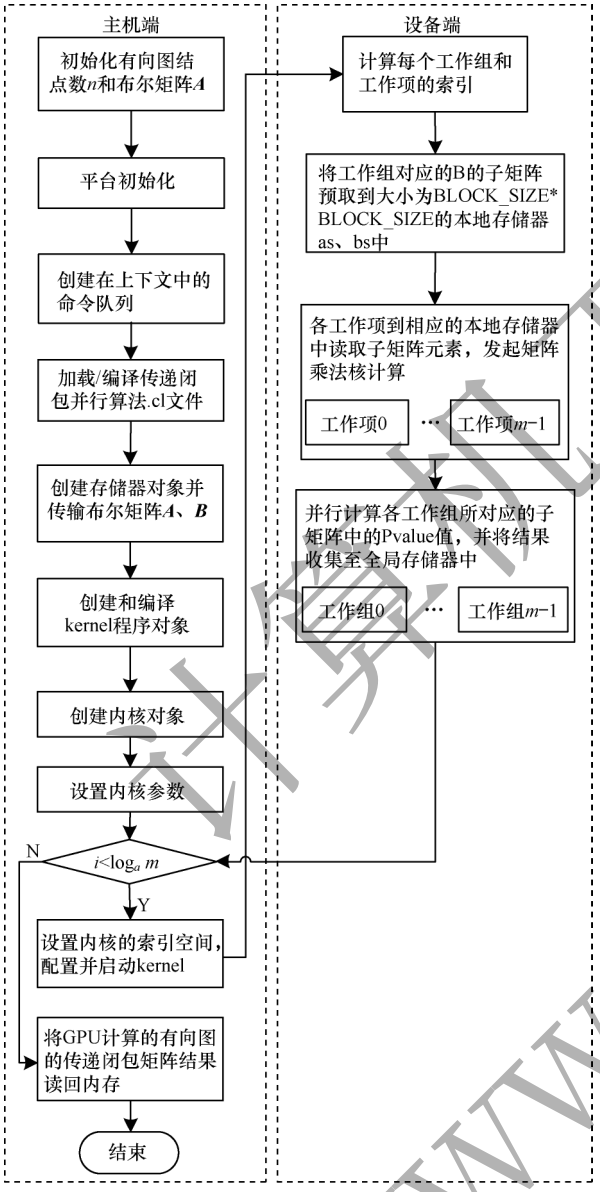

图2 传递闭包并行算法实现流程

Fig.2 Implementation procedure of transitive closure parallel algorithm

传递闭包并行算法执行过程如下：

1)在主机端根据对应的顶点数,初始化布尔矩阵$\boldsymbol{A}$,并保存初始化后的布尔矩阵。

2)初始化OpenCL平台。

3)创建上下文,并在目标设备上创建命令对象。为了协调内核计算,在上下文和计算设备之间利用clCreateCommandQueue命令建立一个逻辑链接。

4)读入源程序文件,并创建和编译程序对象。根据上下文中的设备特性,利用运行时编译系统构建程序对象。

5)设置存储器对象和数据传输。在全局内存中创建buffer存储器对象,然后将存储器访问任务加入到命令队列,最后通过clCreateBuffer将布尔矩阵$\boldsymbol{A}$、$\boldsymbol{B}$从CPU端隐式地传输到设备端的全局内存中。

6)建立内核对象。在指定的一个内核对象中将内核参数和内核函数通过clCreateKernel封装进来。

7)设置需要传递的内核对象参数。

8)创建kernel函数,调度kernel执行。

9)循环调用kernel函数对数据进行相应的处理,计算矩阵乘积大小。

10)将显存端完成运算任务后的结果复制到主机端内存,并且释放设备端显存空间,将最终的计算结果保存到对应的文件中。

3.3 算法的并行方案设计

在设计传递闭包并行算法时,矩阵乘法的并行计算采用了工作项分块的方法实现,计算原理如图3所示。工作组中的每个工作项读取矩阵$\boldsymbol{B}$中的一行和矩阵$\boldsymbol{B}$中的一列,将行、列中对应元素相乘之后再相加,得到新矩阵$\boldsymbol{B}$的对应位置元素值,即每个工作项对应计算新矩阵$\boldsymbol{B}$中的一个元素。以上操作循环经过$p<\log_a m$次后得到有向图的传递闭包$\boldsymbol{A}^+$的形式矩阵。

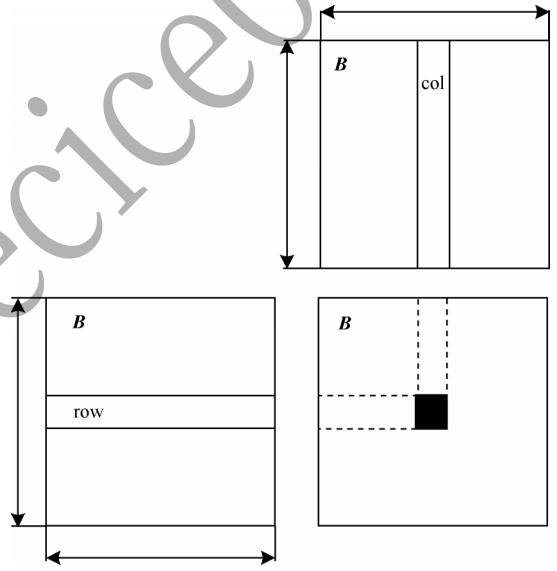

图3 传递闭包算法中的矩阵相乘

Fig.3 Matrix multiplication in the transitive closure algorithm

按照相互之间无重叠的划分原则,整个矩阵$\boldsymbol{B}$将被划分成若干个计算区域。计算区域可作为一个基本处理单位,由工作组处理。文中采用二维工作空间进行设计,从数据层面上看,每个工作组在x,y方向上的维度均为BLOCK_SIZE。工作空间

在 x、y 方向上共有 $\left(\left(m+B_{\text{BLOCK\_SIZE}-1}\right)/B_{\text{BLOCK\_SIZE}}\right)\times\left(\left(m+B_{\text{BLOCK\_SIZE}-1}\right)/B_{\text{BLOCK\_SIZE}}\right)$ 个工作组，每个工作组中执行了 BLOCK_SIZE×BLOCK_SIZE 个工作项。

矩阵乘法中每一对元素间的乘-加计算由一个工作项负责。在内核函数中循环完成矩阵 $\boldsymbol{B}$ 第 i 行元素与矩阵 $\boldsymbol{B}$ 第 j 列元素的乘-加运算，并将乘-加的结果赋给 Pvalue。在该 kernel 函数中，矩阵 $\boldsymbol{B}$ 中的每一个元素从全局存储器中读取了 m 次，造成了时间上的大量延迟。

3.4　优化设计

GPU 全局存储器属于片下存储器，存储空间较大，但具有较高的访存延迟。而本地存储器是 GPU 片上的高速存储器，它的缓冲区驻留在物理 GPU 上。因此，本地存储器的访存延迟要远远低于全局存储器，大量工作项的并行执行能够在一定程度上掩盖全局存储器操作的延迟。

将矩阵相乘后得到的新矩阵 $\boldsymbol{B}$ 分解成小矩阵块，每一个工作组负责计算一个小矩阵块。若矩阵 $\boldsymbol{B}$ 的大小是 $m\times m$，则新矩阵 $\boldsymbol{B}=\boldsymbol{B}\times\boldsymbol{B}$。假设 $m=b\times b$，将新矩阵 $\boldsymbol{B}$ 分为 $b\times b$ 个小的子矩阵 $\boldsymbol{b}_{ij}$，则每一个子矩阵 $\boldsymbol{b}_{ij}$ 的大小为 $b\times b$。2 个相乘的矩阵 $\boldsymbol{B}$ 同新矩阵 $\boldsymbol{B}$ 一样，划分为 $b\times b$ 个小的子矩阵 $\boldsymbol{b}_{ij}$，且每一个子矩阵 $\boldsymbol{b}_{ij}$ 的大小同为 $b\times b$，则传递闭包并行算法中采用分块矩阵乘法的定义为 $\boldsymbol{b}_{ij}=\sum_{k=0}^{b-1}b_{ik}b_{kj}$，计算原理如图 4 所示。

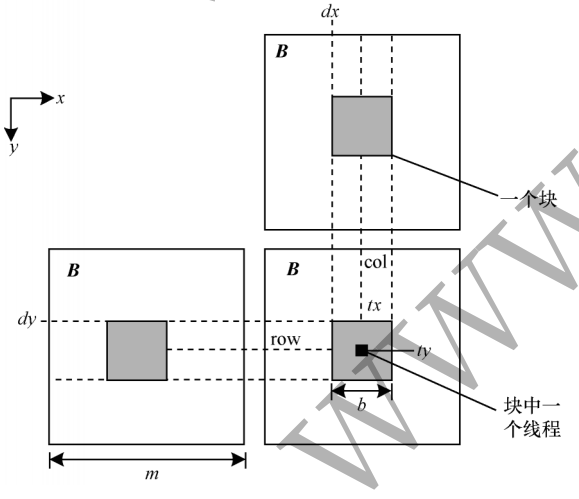

图 4　传递闭包算法中的分块矩阵相乘

Fig.4　Multiplication of block matrix in transitive closure algorithm

在传递闭包并行算法的分块矩阵乘法中，采用静态方式定义大小为 BLOCK_SIZE×BLOCK_SIZE 的本地存储器数组，用于存储矩阵 $\boldsymbol{B}$ 子块数据。

```
__local float as[BLOCK_SIZE][BLOCK_SIZE]
__local float bs[BLOCK_SIZE][BLOCK_SIZE]
```

为从全局存储器预取计算子矩阵到本地存储器，根据工作组的 ID 和工作项的 ID 确定 $\boldsymbol{B}$ 的计算子矩阵的位置，并将 $\boldsymbol{B}$ 中用于计算的 2 个计算子矩阵分别预取至本地数组 as 和 bs 中。每个工作项负责计算一对元素的乘积和 $P_{\text{Pvalue}+}=as[ty][k]\times bs[k][tx]$。原来矩阵的一行或一列数据需要从全局存储器读取 m 次，现在只需要读取 $m/B_{\text{BLOCK\_SIZE}}$ 次，这样在新矩阵 $\boldsymbol{B}$ 的计算过程中矩阵数据需要从全局存储器读取 $m\times m$ 次，优化后只需要读取 $m^2/B_{\text{BLOCK\_SIZE}}$ 次。因此，通过对 GPU 的存储带宽进行充分的利用，减少从全局存储器中重复读取数据。使用本地存储器不仅可以降低访问延迟以此提高访问速率，同时节约了对全局存储器的访问带宽。

4　实验测试与结果分析

本节将给出所描述的传递闭包方法的测试结果。由于单精度浮点运算针对现代计算机，特别是在 GPU 上进行了高度优化，因此本文选择单精度数据类型实现算法。

4.1　测试环境和实验结果

实验软硬件平台如下：

1）硬件平台

平台 1：CPU 为 AMD Ryzen5 1600X 3.6 GHz（六核心），24.0 GB 的系统内存。GPU 型号是 NVIDIA GeForce GTX 1070，CUDA 核心 1 920 颗，1 506 MHz 的核心频率，1 683 MHz 的流处理器频率，8 GB GDDR5 的显存，256 bit 的显存位宽，256 Gb/s 的显存带宽，显存存取速率为 8 Gb/s。

平台 2：CPU 为 AMD Ryzen5 1600X 3.6 GHz（六核心），24.0 GB 的系统内存。GPU 型号是 AMD Radeon RX 570，其中，计算单元 32 组，每组计算单元具有 64 个处理单元，总计 2 048 颗流处理单元，1 168 MHz 的核心频率，256 bit 的显存位宽，8 GB GDDR5 显存。

2）软件平台：操作系统采用微软 Windows 8.1 64 位；集成开发环境为微软 Visual Studio 2017；系统编译环境为 CUDA Toolkit 8.0，OpenCL 1.2 标准被支持。

有向图的顶点集合大小 n 分别取为 20、40、50、70、200、300、500、1 024，作为 rand() 随机数函数的随机数种子分别生成一组随机数，构成布尔矩阵 $\boldsymbol{A}$。根据本文的传递闭包算法的描述，基于 OpenMP 平台和基于 CUDA 平台的传递闭包并行算法均在文中实现。

传递闭包算法运行在基于 OpenMP 系统、基于 CUDA 系统、基于 AMD GPU 的 OpenCL 系统和基于 NVIDIA GPU 的 OpenCL 系统的上处理时间，如表 1 所示。处理时间包括传递闭包算法的所有处理步骤。在 OpenCL 中实现 GPU 并行算法时，必须执行

额外的步骤，如内核创建（读取、创建和构建最终内核对象）、主机内存和GPU全局存储器之间的数据传输以及数据结构初始化。

表1 传递闭包算法执行时间

Table 1 Execution time of transitive closure algorithm

有向图顶点数	串行处理时间/ms	并行处理时间/ms			
		OpenMP	CUDA	OpenCL（AMD）	OpenCL（NVIDIA）
20	28.06	22.45	0.32	0.32	0.32
40	75.48	48.08	0.42	0.41	0.41
50	202.65	85.15	0.51	0.51	0.50
70	377.19	131.43	0.64	0.64	0.63
200	3 406.81	1 048.25	7.02	6.99	6.94
300	7 519.75	1 825.18	22.25	22.17	21.89
500	16 446.00	3 282.63	92.70	92.27	90.05
1 024	89 055.10	16 676.99	804.33	791.81	765.41

用加速比作为加速效果的衡量标准，可以直观地验证各种架构下并行算法的效率，其定义如下：

CPU串行算法执行时间与并行算法执行时间的比值即为加速比：

$$S_{\text{Speedup}}=\frac{T_{\text{serial}}}{T_{\text{parallel}}} \tag{2}$$

其中：T_{serial}是在CPU上单个线程的顺序运算时间；T_{parallel}是在多核CPU或CPU+GPU上多线程实现的并行运算时间。

相对加速比1 基于OpenMP的并行算法运算时间与基于NVIDIA GPU的OpenCL并行算法运算时间的比值：

$$R_{\text{Relative-Speedup1}}=\frac{T_{\text{parallel-OpenMP}}}{T_{\text{parallel-NOpenCL}}} \tag{3}$$

其中：$T_{\text{parallel-OpenMP}}$是在多核CPU上多线程的并行运算时间；$T_{\text{parallel-NOpenCL}}$是在NVIDIA GPU上OpenCL的并行运算时间。

相对加速比2 基于NVIDIA GPU平台的CUDA并行算法运算时间与基于NVIDIA GPU平台的OpenCL并行算法运算时间的比值：

$$R_{\text{Relative-Speedup2}}=\frac{T_{\text{parallel-CUDA}}}{T_{\text{parallel-NOpenCL}}} \tag{4}$$

其中：$T_{\text{parallel-CUDA}}$是在CUDA上的并行执行时间；$T_{\text{parallel-NOpenCL}}$是在NVIDIA GPU上OpenCL并行实现的并行执行时间。$T_{\text{parallel-CUDA}}$和$T_{\text{parallel-NOpenCL}}$定义如下：

$$T_p=T_{\text{kernel}}+T_{\text{ovehead}}+T_{\text{other}} \tag{5}$$

其中：T_{kernel}为OpenCL内核在CPU和GPU上总的执行时间；T_{ovehead}为在CPU和GPU上数据传输时间开销的总和；T_{other}为数据结构初始化等操作总的运行时间。

为了更好地对应用系统速度进行客观评价，采用加速比指标来反映在一定的计算架构下的并行算法相较串行算法的效率提升幅度。使用相对加速比1指标来反映基于NVIDIA GPU的OpenCL并行算法相比基于多核CPU的OpenMP并行算法的效率提升情况，相对加速比2指标则反映出基于NVIDIA GPU的OpenCL并行算法相比基于GPU的CUDA并行算法的效率提升情况，如表2所示。

表2 传递闭包并行算法性能对比

Table 2 Performance comparison of transitive closure parallel algorithm

序号	有向图顶点数	加速比				相对加速比1	相对加速比2
		OpenMP	CUDA	OpenCL（AMD）	OpenCL（NVIDIA）		
1	20	1.25	87.69	87.86	88.02	70.16	1.00
2	40	1.57	179.71	181.75	183.21	117.27	1.02
3	50	2.38	397.35	398.12	401.23	170.30	1.02
4	70	2.87	589.36	591.57	593.14	208.62	1.02
5	200	3.25	485.30	487.54	490.67	151.04	1.01
6	300	4.12	337.97	339.21	343.51	83.38	1.02
7	500	5.01	177.41	178.24	182.64	36.45	1.03
8	1 024	5.34	110.72	112.47	116.35	21.79	1.05

4.2 实验数据分析

4.2.1 系统性能瓶颈分析

在存储器读写操作时，需要邻接矩阵数据的$m\times m\times m$次存储器读取，有向图的传递闭包矩阵数据的$m\times m$次存储器写入操作。设一个$m=200$点的有向图，每个像素值分配存储空间大小是4 Byte，所以，存储器存取数据总量约为0.032 GB，除以kernel实际执行的时间0.000 257 s，得到的带宽数值是约124.51 GB/s，这已经接近GeForce Tesla C2075显示存储器的150.34 GB/s带宽。因此，可以很明显地看出，基于OpenCL架构的传递闭包并行算法的效率受限于全局存储器带宽。

从表2可以看出，基于CPU+GPU的算法加速效果明显，但GPU并行算法的加速比随着有向图顶点数的增加呈现缓慢下降的趋势。主要原因是在OpenCL并行算法操作中，CPU负责读取和输出图的

邻接矩阵数据，而这一过程并没有加速。随着被处理邻接矩阵规模的增加，读取和输出邻接矩阵数据所花费的时间也在增加。因此，OpenCL架构下的传递闭包并行算法的性能瓶颈是显存带宽和主存与显存之间数据传输的带宽。

4.2.2　传递闭包并行算法性能分析

不同并行计算平台下的传递闭包并行算法加速比对比曲线如图5所示。在多核CPU平台上，传递闭包算法的运算速度得到加速。然而，限于核心数，系统的加速比相对较小且变化不大，但由于GPU具有较丰富的计算资源，在CUDA架构和OpenCL架构下的传递闭包算法就可以拥有足够的工作项来进行大量数据的并行处理。1 920个处理单元通过时间分割机制分配到一定数量的工作项，加速比得到较大提高且增幅明显。通过表2分析，在对计算密集型特征明显的大规模数据集计算时，GPU系统运算时间有小量增幅，体现了GPU用于计算密集型的任务运算不如CPU敏感，显现出GPU强大的运算能力。

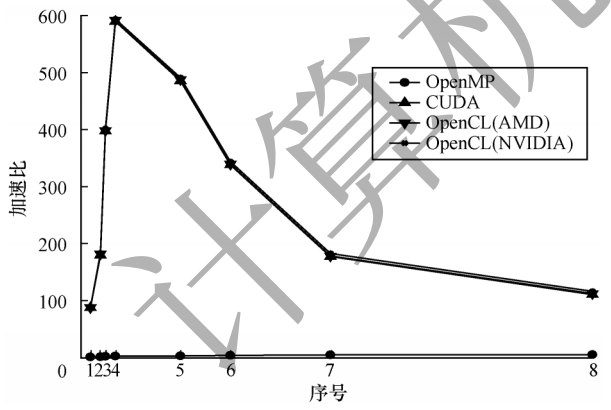

图5　传递闭包并行算法的加速比对比

Fig.5　Comparison of acceleration ratios of transitive closure parallel algorithm

由图5可知，随着布尔矩阵规模的增加，GPU加速下的加速比曲线斜率急剧变大，曲线变得十分陡峭。加速比呈现出快速增加的趋势，比较明显地体现出并行处理的性能提升效果。然而当布尔矩阵大小超过70×70继续增大时，曲线呈现出一种下降趋势。虽然随着布尔矩阵规模的增大，工作空间中包含的工作组数也随之增多，系统中可同时执行更多的子矩阵，对于提高访问全局存储器和本地存储器的效率有益，也越容易隐藏存储器延时，但是布尔矩阵规模的增大，主机端和设备端存储器之间交互数据的时间成本变大，较大程度地抵消了GPU并行计算的优势，导致GPU系统加速性能下降，整体系统性能受到制约。

4.2.3　传递闭包并行算法跨平台性分析

可移植性不但要求源码能够在不同的平台上成功地编译、运行，而且还需要算法应当有相当的性能。运算结果表明，在CUDA架构下的传递闭包并行算法受到单一硬件平台的限制，而基于OpenCL的传递闭包并行算法则在多种硬件平台上获得了较好的可移植性和兼容性，其最大加速比为593.14倍，如图6所示。

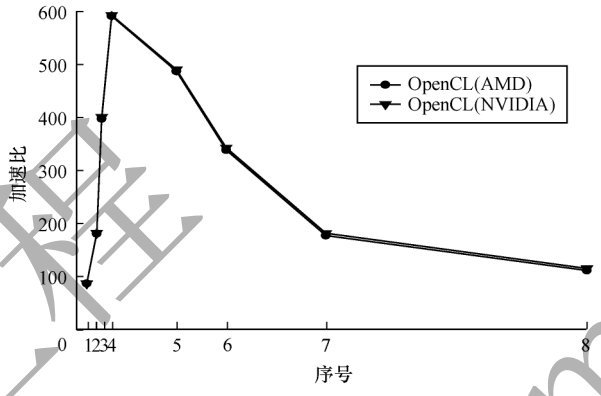

图6　OpenCL加速比趋势

Fig.6　OpenCL acceleration ratio trend

由于采用离线编译内核读写数据文件的OpenCL加速的传递闭包并行算法，相比在线编译内核读写数据文件的CUDA加速的传递闭包并行算法减少了应用初始化时间。在同等数据集规模下，基于OpenCL的传递闭包并行算法的运算耗时更少，与CUDA计算平台上的算法性能相比略有提升，最大获得了1.05倍加速比。而OpenCL加速的传递闭包并行算法性能较之OpenMP计算平台下的算法性能则有很大的提高，加速比最大获得了208.62倍，如图7所示。

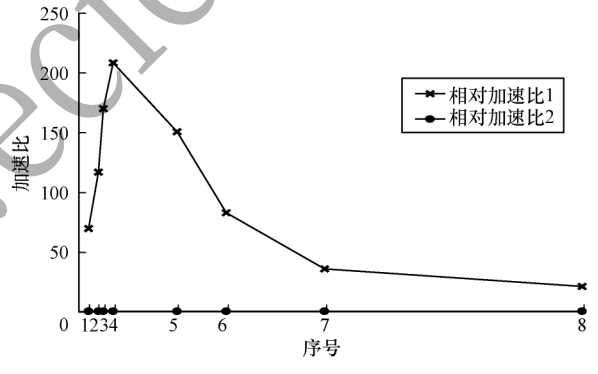

图7　相对加速比趋势

Fig.7　Relative acceleration ratio trend

5　结束语

在许多应用系统中传递闭包是必要的基本部件，且为系统中较为耗时的部分，而矩阵乘对整个系统实时性能则有较大影响。本文针对传递闭包算法串行性能低下的不足，提出适合于OpenCL架构的计算模式，并设计实现了传递闭包GPU并行算法。实验结果表明，基于OpenCL架构的传递闭包并行算法

的性能相比CPU串行算法、基于CPU的OpenMP并行算法和基于GPU的CUDA并行算法，分别取得了593.14倍、208.62倍和1.05倍的加速比。在算法的GPU实现过程中配置适当的内核参数和合理的分块参数，能有效提高处理效率，且实现同等计算量的GPU相比CPU，性价比更高。因此，采用本文GPU异构计算模式对大规模数据运算且系统实时性要求较高的应用，将是一条新的思路。

参考文献

[1] PALKOWSKI M, BIELECKI W. Parallel tiled codes implementing the Smith-waterman alignment algorithm for two and three sequences[J]. Journal of Computational Biology, 2018, 25(10): 1106-1119.

[2] DE B B, DE M H. Cutting levels of the winning probability relation of random variables pairwisely coupled by a same Frank copula[J]. International Journal of Approximate Reasoning, 2019, 112(3): 22-36.

[3] COHEN L, CONSTABLE R L. Intuitionistic ancestral logic[J]. Journal of Logic and Computation, 2019, 29(4): 469-486.

[4] BIELECKI W, SKOTNICKI P. Insight into tiles generated by means of a correction technique[J]. The Journal of Supercomputing, 2019, 75(5): 2665-2690.

[5] COOLEY O, KANG M, PERSON Y. Largest components in random hypergraphs[J]. Combinatorics Probability and Computing, 2018, 27(5): 741-762.

[6] ROTHSTEIN A, DREYFUSS M. Ranking of elements in system reliability modeling: the least influence method[J]. Systems Engineering, 2018, 21(5): 501-508.

[7] QI X F, JIANG Z L. Precise slicing of interprocedural concurrent programs[J]. Frontiers of Computer Science, 2017, 11(6): 971-986.

[8] WEN J H, MA H M. Real-time smoke simulation based on vorticity preserving lattice Boltzmann method[J]. The Visual Computer, 2019, 35(9): 1279-1292.

[9] WU J H, YANG X, ZHANG Z R, et al. A performance model for GPU architectures that considers on-chip resources: application to medical image registration[J]. IEEE Transactions on Parallel and Distributed Systems, 2019, 30(9): 1947-1961.

[10] NIKITIN V V, CARLSSON M, ANDERSSON F, et al. Four-dimensional tomographic reconstruction by time domain decomposition[J]. IEEE Transactions on Computational Imaging, 2019, 5(3): 409-419.

[11] BAHIG H M, ABDELBARI K A. A fast GPU-based hybrid algorithm for addition chains[J]. Cluster Computing, 2018, 21(4): 2001-2011.

[12] GRECO S, MOLINARO C, PULICE C. Efficient maintenance of shortest distances in dynamic graphs[J]. IEEE Transactions on Knowledge and Data Engineering, 2018, 30(3): 474-487.

[13] MARTIN E, STEPHAN F. Implementing fragments of ZFC within an r. e. universe[J]. Journal of Logic and Computation, 2018, 28(1): 1-32.

[14] SHARAN S, TIWARI S P, KUMARI N. On relationship between generalized rough multisets and multiset topologies[J]. International Journal of Machine Learning and Cybernetics, 2017, 8(6): 2017-2024.

[15] BALLESTER-BOLINCHES A, HEINEKEN H, SPAGNUOLO F. On sylow permutable subgroups of finite groups[J]. Forum Mathematicum, 2017, 29(6): 1307-1310.

[16] KAPOUTSIS C A, MULAFFER L. A logical characterization of small 2NFAs[J]. International Journal of Foundations of Computer Science, 2017, 28(5): 445-464.

[17] 范艳焕，耿生玲，李永明. Pebble模糊有穷自动机和传递闭包逻辑[J]. 模糊系统与数学，2015，29(4)：38-44.
FAN Y H, GENG S L, LI Y M. Finite automata with membership values in lattices and monadic second-order lattice-valued logic[J]. Fuzzy Systems and Mathematics, 2015, 29(4): 38-44. (in Chinese)

[18] 舒虎，崇志宏，倪巍伟，等. X-Hop：传递闭包的多跳数压缩存储和快速可达性查询[J]. 计算机科学，2012，39(3)：144-148.
SHU H, CHONG Z H, NI W W. X-Hop: storage of transitive closure and efficient query process[J]. Computer Science, 2012, 39(3): 144-148. (in Chinese)

[19] 汪小燕，杨思春，叶红，等. 传递闭包的增量式更新研究[J]. 苏州科技学院学报(自然科学版)，2015，32(1)：45-48.
WANG X Y, YANG S C, YE H, et al. Research on the incremental updating of the transitive closure[J]. Journal of Suzhou University of Science and Technology (Natural Science), 2015, 32(1): 45-48. (in Chinese)

[20] ARNOLD B, SAUEL R B, SY-DAVID F. Safe recursive set functions[J]. Journal of Symbolic Logic, 2015, 26(3): 1-26.

[21] BIELECKI W, KRASKA K, KLIMEK T. Using basis dependence distance vectors in the modified Floyd-Warshall algorithm[J]. Journal of Combinatorial Optimization, 2015, 30(2): 253-275.

[22] MAREK P, WLODZIMIERZ B. Parallel tiled code generation with loop permutation within tiles[J]. Computing and Informatics, 2017, 36(6): 1261-1282.

[23] BIELECKI W, PALKOWSKI M, SKOTNICKI P. Generation of parallel synchronization-free tiled code[J]. Computing, 2018, 100(3): 277-302.

[24] PALKOWSKI M, BIELECKI W. Parallel tiled Nussinov RNA folding loop nest generated using both dependence graph transitive closure and loop skewing[J]. BMC Bioinformatics, 2017, 18(1): 290.

[25] 朱尚勇. 求传递闭包R+的Warshall法的一种并行计算[J]. 上海交通大学学报，1985，19(5)：101-108.
ZHU S Y. A parallel solution of Warshall algorithm for computing transitive closure R^+[J]. Journal of Shanghai Jiaotong University, 1985, 19(5): 101-108. (in Chinese)

[26] 徐祖渊，张志德，李森. 关于串行循环的并行执行[J]. 计算机工程与设计，1983，8(3)：1-17.
XU Z Y, ZHANG Z D, LI S. About parallel execution of serial loops[J]. Computer Engineering and Design, 1983, 8(3): 1-17. (in Chinese)

[27] 马军，高冈忠雄. 图的最短路径和传递闭包的并行算法[J]. 计算机学报，1990，11(9)：706-708.
MA J, TADAO T. A parallel algorithm for computing the shortest paths and the transitive closures[J]. Chinese Journal of Computers. 1990, 11(9): 706-708. (in Chinese)

[28] ANDRAS K, AGNES V F, JANOS A. Geodesic distance based fuzzy c-medoid clustering-searching forcentral points in graphs and high dimensional data[J]. Fuzzy Sets and Systems, 2016, 286(C): 157-172.

[29] 陈绪斌,曹嘉麟,陈赟,等. 高性能并行Turbo译码器的VLSI设计[J]. 计算机工程,2012,38(23):255-258.
CHEN X B, CAO J L, CHEN B, et al. VLSI design of high performance parallel turbo Decoder [J]. Computer Engineering, 2012, 38(23): 255-258. (in Chinese)

[30] GAYATHRI R G, JYOTHISHA J N. Ex-FTCD: a novel mapreduce model for distributed multi source shortest path problem[J]. Journal of Intelligent and Fuzzy Systems, 2018, 34(6): 1643-1652.

[31] WU S S, DONG X S, ZHANG X J, et al. NoT: a high-level no-threading parallel programming method for heterogeneous systems[J]. The Journal of Supercomputing, 2019, 75(7): 3810-3841.

[32] LIU S S, TANG J, WANG C, et al. A unified cloud platform for autonomous driving[J]. Computer, 2017, 50(12): 42-49.

[33] CHEN C, LI K L, OUYANG A J, et al. FlinkCL: an OpenCL-based in-memory computing architecture on heterogeneous CPU-GPU clusters for big data[J]. IEEE Transactions on Computers, 2018, 67(12): 1765-1779.

[34] MOREN K, GÖHRINGER D. A framework for accelerating local feature extraction with OpenCL on multi-core CPUs and co-processors [J]. Journal of Real-Time Image Processing, 2019, 16(4): 901-918.

[35] SCHMIDTKE R, ERLEBEN K. Chunked bounding volume hierarchies for fast digital prototyping using volumetric meshes [J]. IEEE Transactions on Visualization and Computer Graphics, 2018, 24(12): 3044-3057.

[36] CARPENO A, RUIZ M, GONZALEZ C, et al. OpenCL implementation of an adaptive disruption predictor based on a probabilistic Venn classifier[J]. IEEE Transactions on Nuclear Science, 2019, 66(7): 1007-1013.

[37] LIANG Y, TANG W T, ZHAO R Z, et al. Scale-free sparse matrix-vector multiplication on many-core architectures[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2017, 36(12): 2106-2119.

[38] CIGLARIČ T, ČEŠNOVAR R, ŠTRUMBELJ E. An OpenCL library for parallel random number generators[J]. The Journal of Supercomputing, 2019, 75(7): 3866-3881.

[39] SINGH D P, JOSHI I, CHOUDHARY J. Survey of GPU based sorting algorithms [J]. International Journal of Parallel Programming, 2018, 46(6): 1017-1034.

[40] BESSOUF O, KHELLADI A, ZASLAVSKY T. Transitive closure and transitive reduction in bidirected graphs[J]. Czechoslovak Mathematical Journal, 2019, 69(2): 295-315.

[41] GARHWAL S, JIWARI R. Conversion of fuzzy automata into fuzzy regular expressions using transitive closure[J]. Journal of Intelligent and Fuzzy Systems, 2016, 30(6): 3123-3129.

[42] ANDERSSON N, BEAUCHAMP M, NAVA-AGUILERA E, et al. The women made it work: fuzzy transitive closure of the results chain in a dengue prevention trial in Mexico[J]. BMC Public Health, 2017, 17(supl): 408.

[43] VASILEY A V, CHURIKOV D V. The 2-closure of a 3/2-transitive group in polynomial time [J]. Siberian Mathematical Journal, 2019, 60(2): 279-290.

[44] VAGVOLGYI S. Intersection of the reflexive transitive closures of two rewrite relations induced by term rewriting systems[J]. Information Processing Letters, 2018, 134(5): 47-51.

[45] KAHN C E. Transitive closure of subsumption and causal relations in a large ontology of radiological diagnosis[J]. Journal of Biomedical Informatics, 2016, 61: 27-33.

编辑　索书志