·先进计算与数据处理· 文章编号:1000-3428(2019)09-0023-09 文献标志码:A 中图分类号:TP312

基于 GPU 的稀疏矩阵存储格式优化研究

杨世伟[a],蒋国平[b],宋玉蓉[b],涂 潇[a]

(南京邮电大学 a. 计算机学院; b. 自动化学院,南京 210023)

摘 要: 稀疏矩阵存储格式中的稀疏矩阵向量乘(SpMV)计算效率低下,且分块行列(BRC)存储格式的计算结果缺少再现性和确定性。为此,提出一种改进的 BRCP 存储格式。采用不同的二维分块策略,根据矩阵各行非零元素分布的统计特性自适应调节分块参数,提高 SpMV 在 GPU 平台上的并行性,并设计基于快速分段求和算法的 GPU 内核函数,保证计算结果的确定性及其在不同 GPU 平台上的再现性。实验结果表明,BRCP 存储格式具有较高的计算效率,相比 BRC 存储格式可减少并行环境中的 SpMV 计算误差,并提高 PageRank 排序的准确率。

关键词: 稀疏矩阵向量乘; 计算统一设备架构; 图形处理器; 存储格式; 浮点运算

开放科学(资源服务)标志码(OSID):

中文引用格式: 杨世伟,蒋国平,宋玉蓉,等. 基于 GPU 的稀疏矩阵存储格式优化研究[J]. 计算机工程,2019,45(9):23-31,39.

英文引用格式: YANG Shiwei,JIANG Guoping,SONG Yurong,et al. Research on storage format optimization of sparse matrix based on GPU[J]. Computer Engineering,2019,45(9):23-31,39.

Research on Storage Format Optimization of Sparse Matrix Based on GPU

YANG Shiwei[a],JIANG Guoping[b],SONG Yurong[b],TU Xiao[a]

(a. School of Computer Science; b. School of Automation,Nanjing Utniversity of Posts and Telecommunications,Nanjing 210023,China)

【Abstract】 Sparse Matrix-Vector Multiplication(SpMV) calculation in sparse matrix storage format is inefficient,and the computing results of Blocked Row-Column(BRC) storage format lack reproducibility and certainty. To solve the problem,this paper proposes an improved Blocked Row-Column Plus(BRCP) storage format. The BRCP storage format adopts different two-dimensional blocking strategies,adaptively adjusts the blocking parameters according to the statistical characteristics of the distribution of non-zero elements of each row in the matrix,and improves the parallelism of SpMV on the GPU platform. A GPU kernel fuction based on fast segmented summation algorithm is designed to ensure the certainty of calculation results and their reproducibility on different GPU platforms. Experimental results show that the BRCP storage format has high computational efficiency,which reduces the SpMV calculation error in the parallel environment and improves the accuracy of PageRank sorting compared to the BRC storage format.

【Key words】 Sparse Matrix-Vector Multiplication(SpMV); Compute Unified Device Architecture(CUDA); Graphic Processing Unit(GPU); storage format; floating-point operation

DOI: 10.19678/j.issn.1000-3428.0053513

0 概述

稀疏矩阵向量乘(Sparse Matrix-Vector Multiplication,SpMV)是科学计算中的核心算法之一[1],广泛应用于信息检索[2]、数据挖掘[3]等领域。随着互联网时代数据量的快速增长,如何在大规模数据集上有效实现 SpMV,已成为高性能计算领域的研究热点。稀疏矩阵具有多种存储格式,不同存储格式的 SpMV 存在较大性能差异[4]。文献[5]提出一个 SpMV 自动调优器,结合矩阵特征选择并返回最优的存储格式,从而得到性能提升。文献[6]针对申威处理器提出基于压缩稀疏行(Compressed Sparse Row,CSR)格式的通用异构众核并行 SpMV 算法,提升向量访存命中率,实现负载均衡。

CPU 由于体系结构的限制,性能很难进一步提升。NVIDIA 公司推出的计算统一设备架构

基金项目: 国家自然科学基金(61672298,61873326,61373136)。

作者简介: 杨世伟(1994—),男,硕士研究生,主研方向为复杂网络、GPU 并行计算;蒋国平、宋玉蓉,教授、博士生导师;涂 潇,博士。

收稿日期: 2018-12-28 **修回日期:** 2019-02-19 **E-mail:** songyr@njupt.edu.cn

(Compute Unified Device Architecture, CUDA)[7] 为 GPU 编程提供了无需借助图形学 API 的统一操作接口,高效利用 GPU 提供的强大浮点运算能力和高速数据传输能力,已成为高性能 SpMV 的主要研究方向[8]。文献[9]采用对图切分的压缩数据形式,设计基于多 GPU 平台的支持高效可扩展的大规模图数据处理系统 GFlow。文献[10]结合 ELL 和 CSR 格式,提出 HEC 格式,提高了稀疏矩阵的存储效率。文献[11]设计的 LightSpMV 通过使用原子操作和 warp shuffle 指令,以及动态分配矩阵不同行的元素来提高效率。文献[12]扩展了 CSR 格式,提出转换开销较小的 CSR5 格式,并设计一种快速分段求和算法,提高 SpMV 的性能。

文献[13]提出一种自适应分块行列(Blocked Row-Column, BRC)存储格式,利用矩阵的稀疏特性进行行列分块,减轻线程发散,实现负载均衡。然而,该分块策略无法保证结果的确定性和再现性。当使用基于 BRC 的 SpMV 时,编程人员无法控制 GPU 中线程束(*warp*)的调度顺序,每个 *warp* 可能在不同的系统上以不同的顺序对结果向量执行累加[14]。浮点数的舍入误差导致浮点加法不严格满足结合律,编译器优化也会导致浮点加法的重排序[15],因此无法保证结果的复现。这种误差对于大多数应用来说是可接受的,但在某些情况下,例如用于评估节点重要性的 PageRank 计算中,保证结果向量的准确性十分必要。

在此基础上,本文改进现有的 BRC 数据结构,提出 BRCP(Blocked Row-Column Plus)存储格式,采用二维分块策略,根据稀疏矩阵的非零元素的分布特性自适应调节分块参数,并且设计基于 BRCP 格式的 SpMV GPU 内核,在维持 BRC 格式高效并行性的同时,优化计算结果的确定性,并保证在不同 GPU 平台上的再现性。

1 相关理论及技术

1.1 PageRank 算法

PageRank 算法定义了网络中节点重要性的度量指标。该算法的基本思想是:一个网页的重要性取决于链接指向它的其他网页的数量和质量[2]。基于该假设,PageRank 的计算公式为:

$$r_{k+1}(P_i)=\alpha\sum_{P_j\in B_{P_i}}\frac{r_k(P_j)}{|P_j|}+(1-\alpha) \tag{1}$$

其中,$r_k(P_i)$ 表示页面 P_i 在第 k 次迭代时的 PageRank 值,B_{P_i} 是链接指向 P_i 的页面集合,$|P_j|$ 是页面 P_j 的出链数,$\alpha\in(0,1)$ 是模拟用户浏览网页时点击超链接的阻尼因子。

终止条件为:

$|r_{k+1}(P_i)-r_k(P_i)|<\varepsilon$

基于式(1)的原理,可将页面视为马尔可夫链的状态并将页面排名建模为马尔可夫过程,其类似于一个“随机冲浪者”以特定的状态转移概率从一个页面随机移动到另一个页面。给定 n 个页面,定义 $\boldsymbol{G}$ 为 $n\times n$ 大小的邻接矩阵,矩阵中的每个元素 G_{ij} 为从页面 P_i 到页面 P_j 的转移概率,其中 $0\leqslant i,j<n$。$\boldsymbol{R}$ 表示长度为 n 的分值排名向量,计算公式为:

$$\boldsymbol{R}=\alpha\boldsymbol{G}^{\mathrm{T}}\boldsymbol{R}+(1-\alpha)\boldsymbol{E} \tag{2}$$

其中,$\boldsymbol{E}$ 是均匀分配给每个页面的初始分值向量,PageRank 算法依据式(2)不断迭代更新各页面的分值直到收敛。

PageRank 算法的计算核心为 SpMV,SpMV 是一个内存带宽受限的算法,从内存中读取的每个矩阵元素在计算中仅使用一次。当矩阵是稀疏矩阵时,由于非直接和不规则的内存访问引发的内存开销,使问题更复杂,因此其性能很大程度上取决于稀疏矩阵的稀疏度以及每行非零值的分布。

1.2 数据并行原语

1.2.1 归约

归约[16]是一类基本的数据并行原语,对 n 个元素的输入序列 $[a_0,a_1,\cdots,a_{n-1}]$ 进行二元可结合运算 $\oplus$,生成一个归约结果 $a_0\oplus a_1\oplus\cdots\oplus a_{n-1}$。

1.2.2 扫描

扫描[17-18],又称前缀扫描、前缀求和,是许多并行算法的基础,可分为闭扫描和开扫描。对于 n 个元素的输入序列 $[a_0,a_1,\cdots,a_{n-1}]$,使用二元可结合运算 $\oplus$,返回序列 $[a_0,(a_0\oplus a_1),\cdots,(a_0\oplus a_1\oplus\cdots\oplus a_{n-1})]$,即为闭扫描。使用标识值 I,使得 $I\oplus a=a$,返回序列 $[I,a_0,(a_0\oplus a_1),\cdots,(a_0\oplus a_1\oplus\cdots\oplus a_{n-2})]$,即为开扫描。

1.3 稀疏矩阵压缩存储格式

对于大规模稀疏矩阵,由于内存限制,SpMV 无法使用通用存储格式在 GPU 上进行处理。文献[19]总结了多种常用存储格式,如 COO(Coordinate)格式、CSR 格式、ELL 格式优化用于特定体系结构的稀疏矩阵的计算。稀疏矩阵 $\boldsymbol{A}$ 作为本文中的示例,其大小为 4×4,非零元素数量(Number of Non-Zero, NNZ)为 6。

$$\boldsymbol{A}=\begin{pmatrix}1&0&2&0\\0&0&0&0\\3&4&0&5\\0&0&6&0\end{pmatrix}$$

1.3.1 COO 存储格式

COO 格式是三元组(行、列、值)的简单存储方案。数组 *row*、*col*、*value* 分别存储矩阵中非零元素的行索引、列索引和非零值,数组长度均为 *nnz*,其中 $nnz=6$。

row = (0 0 2 2 2 3)

col = (0 2 0 1 3 2)

value = (1 2 3 4 5 6)

从示例稀疏矩阵 ***A*** 中可看出行索引为 0、列索引为 0 的位置存放元素值 1,行索引为 0、列索引为 2 的位置存放元素值 2,以此类推。

1.3.2　CSR 存储格式

CSR 格式是一种主流的通用稀疏矩阵表示格式,将列索引和非零值存储在数组 *col* 和 *value* 中,这 2 个数组长度为 *nnz*。第 3 个数组 *rowPtr* 表示 *value* 数组中每行的起始位置。

value = (1　2　3　4　5　6)

col = (0　2　0　1　3　2)

rowPtr = (0　2　2　5　6)

对于 $n \times m$ 矩阵($n = m = 4$),*rowPtr* 具有 $n + 1$ 个元素,并且在 *rowPtr*[*i*]中存储第 *i* 行的起始位置,最后一个元素的值是 *nnz*。对于示例稀疏矩阵 ***A***,*rowPtr*[0] = 0 表示第 0 行的起始位置从 *value*[ ]的第 0 号开始,*rowPtr*[1] = 2 表示第 1 行的起始位置从 *value*[ ]的第 2 号开始,因此 *value*[ ]的第 0 号 ~ 第 2 号之前存放了矩阵 ***A*** 第 0 行的元素值,以此类推。

1.3.3　ELL 存储格式

对于每行最多 *k* 个非零元素的 $n \times m$ 矩阵(k = 3,因为矩阵 ***A*** 第 2 行有 3 个元素),ELL 格式将非零值存储在稠密的 $n \times k$ *value* 数组中,具有少于 *k* 个非零值的行用 0 填充,例如矩阵 ***A*** 第 0 行只有 2 个元素,因此第 3 个位置 *value*[0][2]用 0 填充。与 *value* 对应的列索引存储在 *offset* 数组中,数组大小同样是 $n \times k$,并有用于填充的标记值(填充值通常用 –1 或 *k* 表示,本文中用 * 表示),例如 *offset*[2][2] = 3,表示其相应元素值 *value*[2][2] = 5 在原矩阵 ***A*** 中第 2 行的列偏移是 3,即 *A*[2][3] = 5。

$$\boldsymbol{value} = \begin{pmatrix} 1 & 2 & 0 \\ 0 & 0 & 0 \\ 3 & 4 & 5 \\ 6 & 0 & 0 \end{pmatrix}, \boldsymbol{offset} = \begin{pmatrix} 0 & 2 & * \\ * & * & * \\ 0 & 1 & 3 \\ 2 & * & * \end{pmatrix}$$

1.3.4　BRC 存储格式

对于非零元素分布极不规则的稀疏矩阵,若采用一般的存储格式,并行效果较差。文献[13-14]提出基于二维分块机制的 BRC 格式。首先对行分块,并结合对行重新排序和零填充的思想,避免 *warp* 内的线程发散、冗余计算,并控制数据传输的成本。在对行分块的基础上,对列分块,通过将各行划分为更小的块实现负载均衡,使不同的 *warp* 执行相同的工作量。此外,设计一种自调整参数策略,根据矩阵稀疏特性合理选择块大小。

图 1 展示了将示例矩阵 ***A*** 转换为 BRC 格式矩阵后的内存布局。*value* 数组包含非零元素,*col* 包含相应的列索引,*rowPerm* 保留了行重新排序后的原始行下标,其中填充元素标为灰色。*H* 代表一个块中的行数,设为所使用的处理器的 SIMD 执行单元(CUDA 中即 *warp*)的大小。*V* 代表一个块中的列数,取决于原矩阵每行非零元素的统计特性。在构建所有块后,将 *value* 和 *col* 分别放入列优先的一维数组中。因此,使用 *blockBegin* 数组表示每个块的偏移量,*nzPerBlock* 数组表示每个块中的列数。

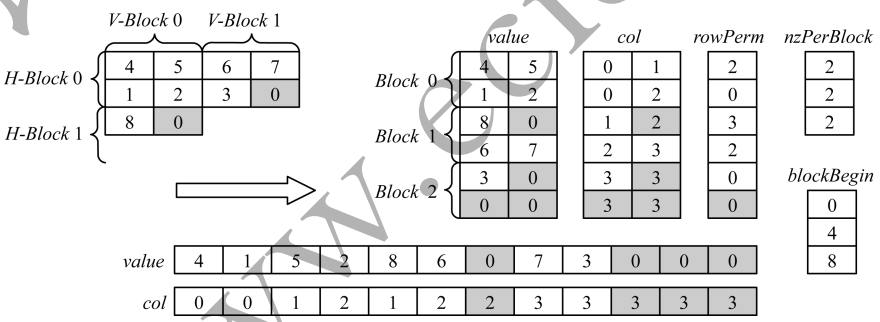

图 1　BRC 格式的内存布局

2　基于 BRCP 的 SpMV 计算

如图 2 所示,块 0 中的线程 1、块 2 中的线程 4 和线程 5 分别进行其所属块中某一行的计算,并且它们所处理的行都是原矩阵中的第 0 行,在 SpMV kernel 的最后它们将各自的结果做原子加法(CUDA 中的 atomicAdd)到结果向量中下标 0 的位置。但是,在不同的系统上,各线程块的调度顺序不能确定,线程块中的各 *warp*(分块行数设为 32,每个块由 1 个 *warp* 处理)的调度顺序同样不能确定,因此此处无法知道线程 1、线程 4、线程 5 的 3 个部分结果以何种顺序添加到结果向量中。由于浮点数在进行每一次加法时,除了系统精度产生的表示误差外,还会产生一个由于舍入(对阶和向右规划操作)而产生的计算误差,因此浮点数不能严格满足加法结合律,浮点数累加过程中不同的结合顺序会使计算结果产生误差[15]。随着数据的增多,尤其对于原稀疏矩阵中非零元素数量较大的行,这类误差在并行计算中叠加、放大,该非确定性不仅产生了程序验证和调试的问题,而且最终会导致计算结果不准确甚至没有意

义。对于一些对计算精确度有高要求的应用,该缺陷更加会被放大,例如在重要节点排序算法 PageRank 中,不同系统上排序结果的不确定性会导致算法失去其价值,那么使用 GPU 并行加速将没有意义。

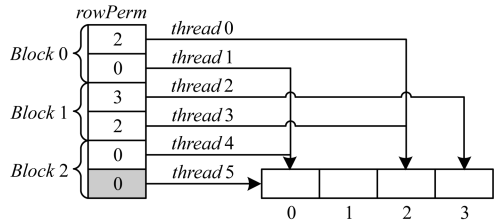

图 2 基于 BRC 格式的 SpMV 内核线程执行结果

为达到计算结果的可再现性和确定性,本文改进 BRC 格式并提出 BRCP 格式的矩阵表示,结合原始二维分块机制和一些额外的步骤,并基于低开销的分段求和算法尽可能减少计算误差。

2.1 BRCP 分块策略

本文设计的 BRCP 数据结构继承了 BRC 格式的设计范式,沿用其在 2 个维度上分块的策略。对于原始稀疏矩阵,按照每行非零元素个数排序后,首先对行分块,一个块中的行数 $H=32$,*warp* 是 GPU 中流多处理器(Stream Multiprocessor,SM)的基本调度单位,目前 CUDA 的 *warp* 大小是 32,这样保证了一个 *warp* 处理一个块中的数据,其中的 32 个线程分别进行每一行的计算,避免了线程发散。在具有幂律特征的稀疏矩阵中,排序后位于前面的行中的非零元素个数远多于后面的行,由于分配一个 *warp* 处理一个块,那么会造成各个 *warp* 之间负载不均衡。为解决该问题,对列进行分块,一个块的列数 V 取决于原矩阵每行非零元素的统计特性以及当前行剩余元素的个数。图 3 表示例矩阵 $\boldsymbol{A}$ 的 BRCP 表示方法的内存布局,为便于说明,设置分块大小为 2×2。

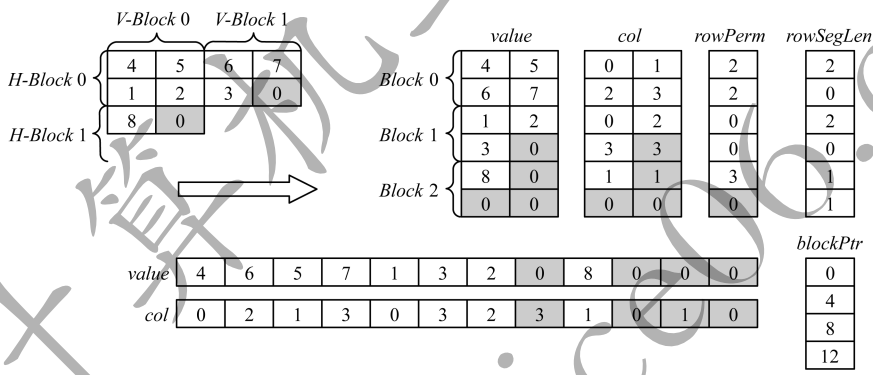

图 3 BRCP 格式的内存布局

本文提出的 BRCP 格式与 BRC 格式不同,从第一行开始扫描稀疏矩阵(已按行进行排序),将矩阵元素按行优先的方式放入块中,取完一行的所有元素后才继续遍历下一行,这样改进的目的是避免 BRC 格式导致的 SpMV 计算结果的不确定性,为此引入另一个辅助数组 *rowSegLen*。

由于与 BRC 格式的分块方案不同,因此本文选择每个块的列数 V 的方法由式(3)决定。对于非零元素数量较多的行,这样做能够使这些行的计算结果更加准确(通常度较大的节点更加关键)。由于已对行进行排序,因此对于剩下的那些非零元素数目较少的行,需要关注扫描的当前行的剩余元素个数,如果不足 T,则继续考虑下一行的非零元素个数。

$$V=\begin{cases} ceil(nnz/64), & \text{当前行的 } nnz \text{ 大于 } 64T \\ \min(T,\max(\text{当前行剩余 } nnz,\text{下一行 } nnz)), & \text{其他} \end{cases} \tag{3}$$

其中,$T=round(\mu+\sigma)$,μ 和 σ 分别是该行非零元素数目的平均值和标准差,在此借鉴 BRC 格式的思想,使用矩阵稀疏性作为统一线程工作量的标准,达到根据稀疏矩阵特性调节参数的目的。

对于大小为 $H\times V$ 的分块,该块后续每一行如果非零元素不足 V,则按照 ELL 格式的机制填充 0,如图 3 的左上部分所示。该方法能较好地适应各种类型的稀疏矩阵,本文的排序和分块策略基于稀疏矩阵中非零元素的分布特性,因此适用于具有幂律分布的真实世界网络。

将原始稀疏矩阵分块后,剩下部分与 BRC 格式的处理类似,将分好块的矩阵元素及其下标按列优先的顺序存入 2 个一维数组中,*value* 数组存放矩阵实际的非零元素和填充的 0,*col* 数组包含 *value* 相应的列下标,列优先的存放方式能使线程合并访问,充分利用 GPU 性能。*rowPerm* 数组用于存储重新排序后该行在原始稀疏矩阵的行下标,*rowSegLen* 数组通过 *rowPerm* 数组产生,表示行下标的分段长度,是实现本文 SpMV 计算确定性的核心。*blockWidth* 数组用于存储每个块的列数,*blockPtr* 数组采用 CSR 格式中 *rowPtr* 数组的思想,表示每个块在 *value* 数组中的起始下标位置,因此 *blockPtr* 的元素个数为

分块个数加 1,并且最后一个元素值与 *value* 和 *col* 数组的元素个数相同。在实际存储过程中,通过 *blockPtr* 数组可以得到每个块的列数,第 i 个分块的列数 $blockWidth[i]$ 与 *blockPtr* 数组的关系为:$blockWidth[i] \times H = blockPtr[i+1] - blockPtr[i]$,从而节省存储空间,无需与 BRC 格式一样使用 *blockBegin* 数组。算法 1 给出了 CSR 格式的矩阵转换到 BRCP 格式的伪代码。

算法 1　BRCP 格式转换算法

输入　CSR 矩阵(*rowPtr*,*col*,*value*)

输出　BRCP 矩阵(*rowPerm*,*rowSegLen*,*col*,*value*,*blockPtr*)

```
1. begin
2. extract μ, σ and rowVec (a vector of struct containing index and nnz per row) from rowPtr;
3. sort rowVec based on nnz per row;
4. H = 32, T = round(μ + σ);
5. numBlocks = 0, blockPtr[0] = 0, idx = 0;
6. while idx < rows and rowVec[idx].nnz do
7. extract a block and add it to the list of blocks;
8. determine blockWidth;
9. for i = 1:H do
10. if idx = rows then
11. add dummy rows;
12. else
13. add current row index to rowPerm;
14. store value and col data of length of blockWidth to the next row of current block;
15. update idx and currentPtr of rowVec;
16. store data of current block to BRCP format in column order;
17. blockPtr. push_back(blockPtr[numBlocks] + H * block_width[numBlocks]);
18. set rowSegLen of current block according rowPerm;
```

2.2　基于 BRCP 的 SpMV 内核线程执行

本文设计的 BRCP 格式采用分块策略的优势为:原始稀疏矩阵相同行的元素尽可能处于同一个分块,在每个块中可对块中所有行的部分结果先进行分段求和,将处于原矩阵同一行的中间值进行求和,再原子加到结果向量中,而不是像基于 BRC 格式的 SpMV 方案那样直接将块中每一行的计算结果原子加到结果向量中。并行的分段求和算法[20]已有成熟的算法思路和各种优化方法,其求和数值结合顺序能够确定,因此求和结果明确。

图 4 展示了本文 SpMV 内核的线程执行过程。将一个块中的元素按其所在行分段,先在每个块中执行分段求和,例如块 0 中的线程 0 和线程 1 负责的元素都位于原始矩阵的第 2 行,先在块内将它们的计算结果相加,再由块中的第 1 个线程执行原子加法累加到结果向量中,那么块 0 中的线程 0、线程 1就不会因调度顺序不同产生误差。但是也存在此类情况,例如线程 2、线程 3 在块 1 中,线程 5 在块 2 中,但它们都负责第 0 行元素的计算,最后线程 2 和线程 5 都会将其计算结果原子加到下标为 0 的位置,无法有效统一这一操作的执行顺序。不过浮点加法具有交换律,如果只有 2 个块中的线程进行累加,那么结果不会受到影响,如果存在 3 个或超过 3 个块的线程都负责同一行的计算,那么该行的累加结果无法保证确定性。因此,对于那些含有非零元素较多的行,在分块时应尽可能地保证负责该行的线程数不超过 64 个,虽然数量超过 33 个时线程就可能分布在 3 个块中,但出现这种情况的概率极小。为避免产生浮点数的舍入误差,对于非零元素较大的行,使负责该行的线程数小于等于 64,即线程块的列数大于等于该行 *nnz*/64,于是将当前块的实际列数 V 设计为 $V = ceil(nnz/64)$。

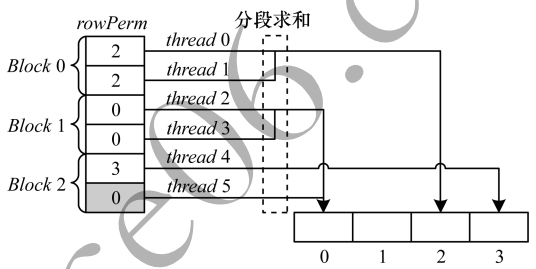

图 4　基于 BRCP 格式的 SpMV 内核线程执行结果

算法 2 给出了 BRCP 格式对应的 CUDA kernel 函数。在此 kernel 中,i 循环的每次迭代都会处理大小为 $H \times V$ 的分块,其中调用了 *warp* 级的快速分段求和算法。在 j 循环中,每个线程处理块中的一行并执行 T 次计算,最后累加出部分结果,$T = blockWidth[cb] = (blockPtr[cb+1] - blockPtr[cb])/H$。

算法 2　基于 BRCP 格式的 SpMV 内核函数求解算法

```
1. begin
2. id = blockIdx.x * blockDim.x + threadIdx.x;
3. b_row = id % H;
4. for i = 1:rep do
5. cb = id / H;
6. tmp = 0;
7. for j = 0:T - 1 do
8. index = blockPtr[cb] + j * H + b_row;
9. tmp += value[index] * x[col[index]];
10. CALL FUNCTION <Fast Segmented Sum>;
11. id += gridDim.x * blockDim.x;
```

2.3 快速分段求和算法

算法 2 中调用的快速分段求和算法是使 BRCP 格式稀疏矩阵实现精确计算 SpMV 的关键,核心思想是将块中的元素按其所在行分段,同一个行的计算结果先进行求和,这时的求和结果能够确定,而不是将每一行的结果分别单独加到结果向量的一个元素中,避免受到 GPU 线程调度顺序的不确定而发生变化。

在预处理过程中产生的数组 *rowSegLen* 用于执行该分段求和算法。由于分段求和算法[20]比较复杂且效率不佳,因此文献[12]设计一种快速分段求和算法,使用辅助数组 *seg_offset* 表示段偏移,并结合前缀扫描[18]的思想实现分段求和。本文采用类似的策略,设计一种 *warp* 级的快速分段求和算法。与文献[12]算法不同,本文没有使用段偏移数组,因为如果 *seg_offset* 的一个元素是 0,既有可能表示该元素未处于段首,又有可能是一个段长为 1 的分段段首,采用 *seg_offset* 不能较好地区分这 2 种情况,具有二义性。因此本文采用表示段长的 *rowSegLen* 作为辅助数组。图 5 给出了使用 *rowSegLen* 和前缀闭扫描相结合的快速分段求和算法。

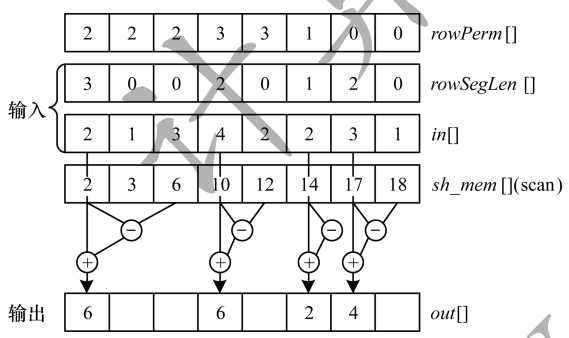

图 5 快速分段求和算法

在图 5 中,*rowSegLen* 数组表示各段的长度,要生成 *rowSegLen*,只需遍历 *rowPerm*,向后搜寻连续相同值的个数。例如,图 5 中 *rowPerm* 数组的前 3 个元素均为 2,表示分块后该行元素位于原矩阵的第 2 行,则 *rowSegLen* 的第 1 个元素为 3,标识当前位置是一个分段的段首,且段长为 3。*in*[ ]为每行元素计算后的结果,对 *in*[ ]并行前缀闭扫描后的结果存于共享内存进行后续操作,以实现快速的线程通信。为进一步提高效率,本文通过使用 warp shuffle 指令[21]实现 *warp* 级的前缀闭扫描。CUDA 9.0 引入一组内建函数[22](例如 *shfl_up* 和 *shfl_down*),能够在一次操作中完成 *warp* 级下标的广播,而不需要通过共享内存执行数据交换,这种更加轻量级的操作使得同一 *warp* 内的线程可实现更快通信。最后每个线程查看其 *rowSegLen*,*rowSegLen* 非零则标识了段首,其值标识了段尾位置,例如图 5 中的第 1 个线程 0,其 *rowSegLen*[0] = 3,表明线程 0 负责的元素为段首元素,则 *out*[0] = *sh_mem*[0 + 3 −1] − *sh_mem*[0] + *in*[0] = 6,即为该段的和。算法 3 给出了快速分段求和算法的伪代码。

算法 3 快速分段求和算法

```
1. begin
2. id = blockIdx.x * blockDim.x + threadIdx.x;
3. tid = threadIdx.x;
4. laneId = threadIdx.x % 32;
5. __share__ space[1 024];
6. tmp = in;
7. for offset = 1;offset < 32;offset * = 2 do
8. t = in;
9. t + = warp.shfl_up(t,offset);
10. if laneId > = offset then
11. in = t;
12. space[tid] = in;
13. if rowSegLen[id] > 0 then
14. in = tmp + space[tid + rowSegLen[id] - 1] - in;
15. atomicAdd(out + rowPerm[id],in);
```

3 实验结果与分析

实验选取 2 种测试环境:1)Intel(R) Core(TM) i5-3210M CPU @ 2.50 GHz 4 核,8.00 GB 内存,Windows 10 操作系统,NVIDIA GeForce GT 650M 显卡;2)Intel(R) Xeon(R) Gold 5122 CPU @ 3.60 GHz 8 核,Ubuntu 16.04 操作系统,GeForce GTX 1080Ti 显卡。CUDA 版本均为 9.0。GPU 各项性能指标如表 1 所示。

表 1 GPU 各项性能指标

性能指标	GeForce GT 650M	GeForce GTX 1080Ti
计算架构	Kepler	Pascal
计算能力	3.0	6.1
流多处理器数	2(384 Cores)	28(3 584 Cores)
线程块/流多处理器的最大线程数	1 024/2 048	1 024/2 048
线程块的共享存储器/KB	48	48
L2 缓存/KB	48	2 816
全局存储器/MB	2 048	11 168
峰值带宽/(GB·s^{-1})	64	484.44

为评估基于 BRCP 格式的 SpMV 性能,本文从以下 4 个方面进行实验:1)与 BRC 格式比较 SpMV 操作的计算耗时;2)与 BRC 格式比较

SpMV 得出的结果向量与准确值的相对误差;3)与 BRC 格式以及 LightSpMV 格式比较 PageRank 的加速性能;4)与 BRC 格式比较 PageRank 排序结果的准确率。

3.1　SpMV 计算耗时

本文选择 10 个稀疏矩阵评估 SpMV 的性能,前 4 个来自 NVIDIA Research,剩下的来自 UF Sparse Matrix Collection[23],即佛罗里达大学稀疏矩阵集。这 10 个矩阵均是带权稀疏矩阵,本文避免设置矩阵数值,使实验结论更具信服力。表 2 给出矩阵特征,包括矩阵行列数、非零元素总数、各行非零元素的最小值/最大值、平均值/标准差(μ/σ)。

实验统计 BRC 和 BRCP 进行一次 SpMV 操作的计算耗时,排除主机和设备之间数据传输耗时以及在主机端执行的一次数据结构转换的开销,避免受到 CPU 性能和内存带宽的影响,只统计在 GPU 中的计算时间。每次实验进行 20 次 SpMV 计算,并记录耗时的算术平均值。

表 2　SpMV 稀疏矩阵集

稀疏矩阵	行列数	非零元素总数	各行非零元素的最小值/最大值	各行非零元素的平均值/标准差
cant	62 451	2 034 917	1/40	7/33
consph	83 334	3 046 907	1/66	37/11
rma10	46 835	2 374 001	4/145	51/28
webbase-1M	1 000 005	3 105 536	1/4 700	3/25
Belcastro/human_gene1	22 283	24 691 926	2/7 940	1 108/1 409
Belcastro/mouse_gene	45 101	29 012 392	2/8 033	643/857
DIMACS10/fl2010	484 481	2 346 294	1/177	5/3
DIMACS10/il2010	451 554	2 164 464	1/90	5/2
vanHeukelum/cage13	445 315	7 479 343	3/39	17/5
vanHeukelum/cage14	1 505 785	27 130 349	5/41	18/5

BRC 和 BRCP 格式在 2 种平台下进行一次 SpMV 计算的执行时间如图 6、图 7 所示。

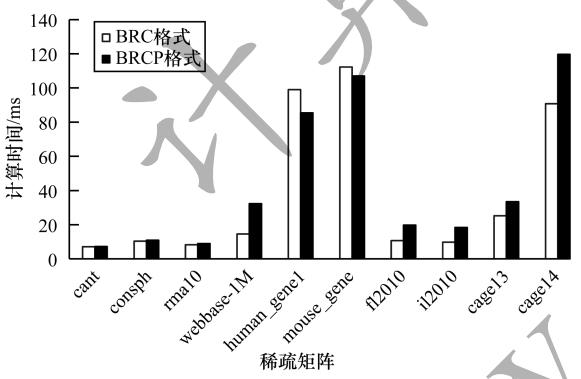

图 6　GeForce GT 650M 平台上 BRC 与 BRCP 的 SpMV 耗时比较

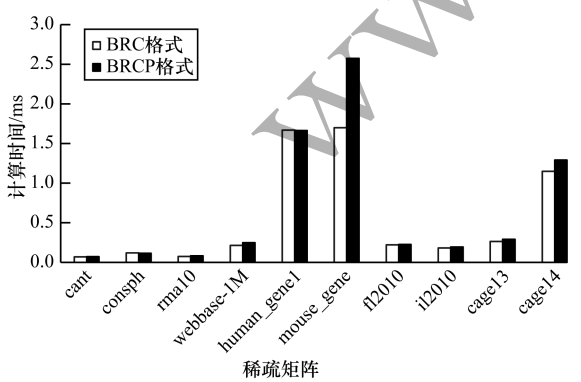

图 7　GeForce GTX 1080Ti 平台上 BRC 与 BRCP 的 SpMV 耗时比较

BRCP 格式相比 BRC 的 GPU kernel 多出一个分段求和的步骤,因此大部分情况下基于 BRCP 的 SpMV 耗时更多,BRC 格式在 GT 650M GPU 平台上耗时平均为 BRC 的 1.25 倍,在 GTX 1080Ti GPU 平台上耗时平均为 BRC 的 1.1 倍,可见 BRCP 格式在大部分 SpMV 计算中牺牲了一定的计算效率。

3.2　SpMV 结果向量的相对误差

将表 2 中的带权稀疏矩阵与一个全 1 向量相乘,分别用 BRC 格式和 BRCP 格式进行 SpMV 计算,并将得到的结果向量($x_1, x_2, \cdots, x_n$)与真值向量($\mu_1, \mu_2, \cdots, \mu_n$)按式(4)计算出相对误差。

$$RE = \sum_i \frac{x_i - \mu_i}{\mu_i} \tag{4}$$

BRC 与 BRCP 的 SpMV 相对误差比较如表 3 所示。对于矩阵 cant 的 SpMV,BRC 格式在 2 种 GPU 上表现均优于 BRCP。对于矩阵 consph,BRCP 在 GeForce GT 650M 上的误差比 BRC 小,而在 GeForce GTX1080 Ti 上误差较大。在其余大部分情况下,本文提出的 BRCP 格式相对于 BRC 格式,与准确值之间的误差较小。BRC 格式在矩阵 consph 和 webbase-1M 的计算中,受到不同平台线程调度不确定性的影响,在 2 种 GPU 平台下的计算结果误差差距较大,而 BRCP 格式在不同平台下的结果表现出更好的再现性。

表 3 BRC 与 BRCP 的 SpMV 相对误差比较

稀疏矩阵	GeForce GT 650M		GeForce GTX 1080Ti	
	BRC	BRCP	BRC	BRCP
cant	0.006 353	0.071 345	0.005 989	0.064 358
consph	6.898 257	0.201 417	0.022 777	0.091 067
rma10	5.461 187	0.000 000	5.461 187	0.000 000
webbase-1M	1.726 860	0.446 535	4.442 622	0.362 778
Belcastro/human_gene1	7.471 732	0.011 107	7.471 729	0.011 107
Belcastro/mouse_gene	9.278 843	0.020 003	9.278 842	0.020 003
DIMACS10/fl2010	12.187 372	0.001 693	12.187 372	0.001 693
DIMACS10/il2010	9.107 744	0.001 344	9.107 744	0.001 344
vanHeukelum/cage13	7.196 009	0.169 302	7.196 009	0.169 302
vanHeukelum/cage14	1.528 160	0.569 980	1.528 160	0.569 980

3.3 PageRank 排序耗时

为衡量基于 BRCP SpMV 的 PageRank 性能，本文选取表 4 给出的 10 个来自不同领域的真实网络数据集，其中 real1 ~ real4 分别表示自治系统网络、Internet 网络、道路网络和社交网络（http://www.dcjingsai.com/），其余 6 个网络检索自 UF Sparse Matrix Collection[23]。

表 4 PageRank 稀疏矩阵集

稀疏矩阵	行列数	非零元素总数
real1	1 694 616	22 188 418
real2	1 957 027	5 520 776
real3	426 485	17 086 642
real4	855 802	8 582 704
LAW/eu-2005	862 664	19 235 140
LAW/in-2004	1 382 908	16 917 053
SNAP/as-skitter	1 696 415	22 190 596
SNAP/higgs-twitter	456 626	14 855 842
SNAP/sx-stackoverflow	2 601 977	36 233 450
SNAP/wiki-topcats	1 791 489	28 511 807

实验将 PageRank 算法中的阻尼系数 α 设为 0.85，收敛约束 ε 设为 10^{-5}。根据式(2)，GPU 中 PageRank 的每轮迭代包含一次并行的 SpMV 和一次并行向量加法，将计算结果与上一轮迭代结果的误差向量做规约得到误差值，若小于 ε 则终止迭代。不同矩阵格式的 PageRank 只在 SpMV 操作中有所不同，其余操作均不受矩阵格式影响。本文统计在 GPU 中 PageRank 所有迭代过程的时间，每个实验重复 20 次并记录平均值。本文引入基于 CSR 格式的 SpMV 方法 LightSpMV[11] 作为性能基准，该算法的执行效率优于 CUSP、ViennaCL 和 cuSPARSE 等，同时使用基于 BRC 格式的 SpMV 方法，并与本文提出的 BRCP 算法做比较，执行时间如图 8、图 9 所示。

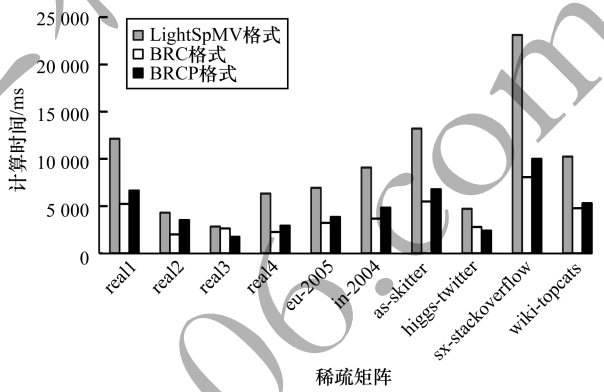

图 8 GeForce GT 650M 平台上基于 LightSpMV、BRC 和 BRCP 格式的 PageRank 耗时比较

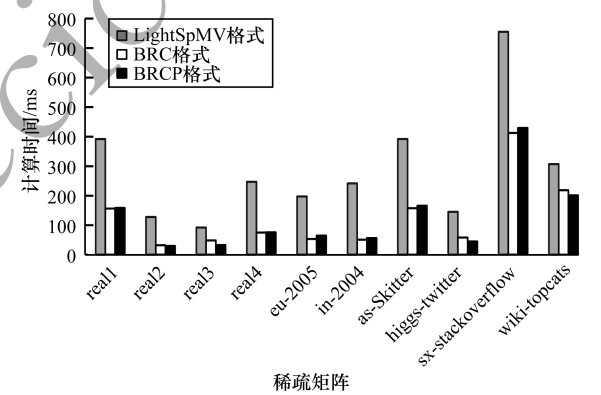

图 9 GeForce GTX 1080Ti 平台上基于 LightSpMV、BRC 和 BRCP 格式的 PageRank 耗时比较

可以看出，基于 BRCP 的 PageRank 执行时间在每种情况下都优于 LightSpMV，在 GT 650M GPU 上平均加速比为 1.87，最高加速比达到 2.31；在 GTX 1080Ti GPU 上平均加速比为 2.91，最高加速比为 4.29。在 GT 650M GPU 上，基于 BRCP 的 PageRank 比 BRC 格式总体稍慢一些，速度仅为 BRC 的89%；而在 GTX 1080Ti GPU 上，BRCP 格式与 BRC 格式的执行效率大体一致，平均获得了 1.05 倍

的加速。可见,基于 BRCP 格式的 SpMV 在实际应用于 PageRank 算法时,不仅充分利用了 GPU 的并行计算能力,而且在修复 BRC 格式漏洞的同时,也保持了原有计算效率。

3.4　PageRank 排序结果准确率

实验分别基于 BRC 格式和 BRCP 格式的 SpMV 进行 PageRank 计算,得到每个节点的分值,并根据分值按照重要性从大到小的顺序对节点排序,最后将 2 种排序结果与准确的排序结果相比较,得到2 种方法执行 PageRank 的准确率。例如有 3 个节点 A、B、C,节点重要性的正确顺序是 $A>B>C$,若得出的顺序是 $A>C>B$,则排序正确的节点数为 1;若得出的顺序是 $B>C>A$,则排序正确的节点数为 0。在节点重要性排序中,重要程度越高的节点是否准确排序更为关键,因此本文统计了结果向量的前 100 个,比较其与正确结果的相同排名数量,如图 10、图 11 所示。

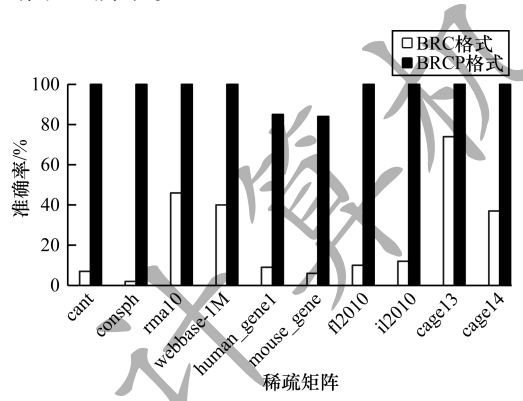

图 10　GeForce GT 650M 平台上 BRC 与 BRCP 前 100 个结果向量的准确率比较

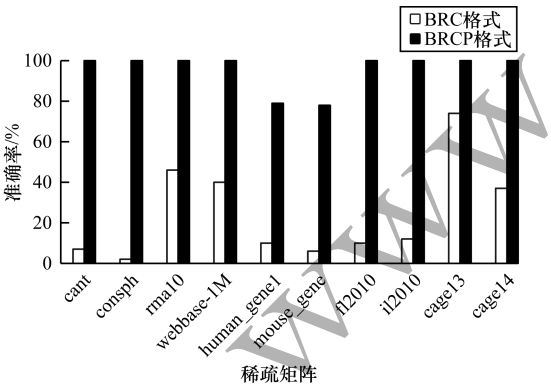

图 11　GeForce GTX 1080Ti 平台上 BRC 与 BRCP 前100 个结果向量的准确率比较

虽然实验中 BRC 格式 SpMV 的误差在数值上较小,但从中可以发现,当其应用于 PageRank 算法时,微小的误差也会导致无法准确地对网络节点进行排名,而修复了 BRC 缺陷的 BRCP 几乎能够得到与正确答案相同的结果。

4　结束语

本文在支持 CUDA 的 GPU 上提出可用于 PageRank 等数据挖掘算法的 SpMV 方法。该方法基于 BRCP 数据结构,使用分块策略表示稀疏矩阵,适用于各种稀疏矩阵,如各行非零元素分布较平均或者是各行非零元素标准差远高于均值、波动较大的矩阵,尤其是具有幂律特征的稀疏矩阵。BRCP 存储格式在并行环境中具有较高的计算准确率,并保证了不同系统中计算结果的再现性,最终通过实验验证了该方法的有效性和可行性。随着网络规模的扩大,单 GPU 已经难以处理超大规模的数据,因此将基于单 GPU 的算法扩展到单机多 GPU 以及分布式环境的多机多 GPU 中将是下一步的研究方向。

参考文献

[1] 尹孟嘉,许先斌,何水兵,等. GPU 稀疏矩阵向量乘的性能模型构造[J]. 计算机科学,2017,44(4):182-187.
[2] BRIN S, PAGE L. The anatomy of a large-scale hypertextual Web search engine[J]. Computer Networks and ISDN Systems,1998,30(1):107-117.
[3] TONG Hanghang, FALOUTSOS C, PAN Jiayu. Random walk with restart: fast solutions and applications[J]. Knowledge and Information Systems, 2008, 14(3): 327-346.
[4] LANGR D, TVRDÍK P. Evaluation criteria for sparse matrix storage formats[J]. IEEE Transactions on Parallel and Distributed Systems,2016,27(2):428-440.
[5] 李佳佳,张秀霞,谭光明,等. 选择稀疏矩阵乘法最优存储格式的研究[J]. 计算机研究与发展,2014,51(4):882-894.
[6] 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[J]. 软件学报,2018,29(12):3921-3932.
[7] LINDHOLM E, NICKOLLS J, OBERMAN S, et al. NVIDIA Tesla: a unified graphics and computing architecture[J]. IEEE Micro,2008,28(2):39-55.
[8] MAGGIONI M, BERGER-WOLF T. Optimization techniques for sparse matrix-vector multiplication on GPUs[J]. Journal of Parallel and Distributed Computing,2016,93(C):66-86.
[9] 张珩,张立波,武延军. 基于 Multi-GPU 平台的大规模图数据处理[J]. 计算机研究与发展,2018,55(2):273-288.
[10] 程凯,田瑾,马瑞琳. 基于 GPU 的高效稀疏矩阵存储格式研究[J]. 计算机工程,2018,44(8):54-60.
[11] LIU Yongchao, SCHMIDT B. LightSpMV: faster CUDA-compatible sparse matrix-vector multiplication using compressed sparse rows[J]. Journal of Signal Processing Systems,2018,90(1):69-86.
[12] LIU Weifeng, VINTER B. CSR5: an efficient storage format for cross-platform sparse matrix-vector multiplication[C]//Proceedings of ACM International Conference on Supercomputing. New York, USA: ACM Press,2015:339-350.

(下转第 39 页)

（上接第 31 页）

[13] ASHARI A, SEDAGHATI N, EISENLOHR J, et al. An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on GPUs [M]. Berlin, Germany: Springer, 2014.

[14] ASHARI A, SEDAGHATI N, EISENLOHR J, et al. A model-driven blocking strategy for load balanced sparse matrix-vector multiplication on GPUs [J]. Journal of Parallel and Distributed Computing, 2015, 76: 3-15.

[15] WILKINSON J H. Error analysis of floating-point computation[J]. Numerische Mathematik, 1960, 2(1): 319-340.

[16] HILLIS W D, STEELE G L. Data parallel algorithms[J]. Communications of the ACM, 1986, 29(12): 1170-1183.

[17] HARRIS M, SENGUPTA S, OWENS J D. Parallel prefix sum (scan) with CUDA[M]//NGUYEN H. GPU Gems 3. New Jersey, USA: Addison Wesley, 2007: 851-876.

[18] SENGUPTA S, HARRIS M, ZHANG Yao, et al. Scan primitives for GPU computing [C]//Proceedings of ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware. New York, USA: ACM Press, 2007: 97-106.

[19] FILIPPONE S, CARDELLINI V, BARBIERI D, et al. Sparse matrix-vector multiplication on GPGPUs [J]. ACM Transactions on Mathematical Software, 2017, 43(4): 1-49.

[20] BLELLOCH G E, HEROUX M A, ZAGHA M. Segmented operations for sparse matrix computation on vector multiprocessors: CMU-CS-93-173 [R]. Pittsburgh, USA: Carnegie Mellon University, 1993: 1-7.

[21] CHENG J, GROSSMAN M, MCKERCHER T. Professional CUDA C Programming[M]. [S.l.]: Wrox, 2014.

[22] NVIDIA. CUDA C programming guide [EB/OL]. [2018-11-27]. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[23] DAVIS T A, HU Yifan. The university of florida sparse matrix collection [J]. ACM Transactions on Mathematical Software, 2011, 38(1): 1-25.

编辑　陆燕菲