文章编号:1007-5321(2019)03-0106-08 **DOI**:10.13190/j.jbupt.2018-252

高性能行任务散列法 GPU 一般稀疏矩阵–矩阵乘法

汤 洋[1], 赵达非[2,3], 黄智濒[2,3], 戴志涛[2,3]

(1. 北京邮电大学 理学院, 北京 100876; 2. 北京邮电大学 智能通信软件与多媒体北京市重点实验室, 北京 100876;
3. 北京邮电大学 计算机学院, 北京 100876)

摘要: 针对一般稀疏矩阵–矩阵乘法(SpGEMM)的性能问题,提出了一种基于任务分类和低延迟散列表的图形处理器上的加速 SpGEMM 算法 RBSPARSE. 该算法由一种低成本子任务复杂度预分析方法和一种低延迟共享内存上的散列表的方法组成,以达到最大效率. 通过解决负载均衡和内存延迟问题,RBSPARSE 可以显著减少计算的总时间. 比较了 RBSparse 和 BHSparse,前者是最快的 SpGEMM 算法,结果表明 RBSparse 的性能是 BHSparse 的平均 3.1 倍,在最佳情况下可达到 14.49 倍.

关 键 词: 稀疏矩阵–矩阵乘法; 图形处理器; 性能优化; 散列表; 共享内存

中图分类号: TP391 **文献标志码**: A

High Performance Row-Based Hashing GPU SpGEMM

TANG Yang[1], ZHAO Da-fei[2,3], HUANG Zhi-bin[2,3], DAI Zhi-tao[2,3]

(1. School of Science, Beijing University of Posts and Telecommunications, Beijing 100876, China;
2. Beijing Key Laboratory of Intelligent Telecommunication Software and Multimedia, Beijing University of Posts and Telecommunications, Beijing 100876, China;
3. School of Computer Science, Beijing University of Posts and Telecommunication, Beijing 100876, China)

Abstract: Aiming at the performance problem of general sparse matrix-matrix multiplication (SpGEMM), a graphics processing unit (GPU)-accelerate SpGEMM algorithm based on task classification and low-latency Hashing table, RBSPARSE, was presented in the paper. RBSPARSE consists of a low-cost pre-analysis method to identify the complexity of sub-tasks, and a Hashing table-based algorithm which could utilize low-latency shared memory to achieve max efficiency. By taking the load balancing issue and the memory latency issue into consideration, RBSPARSE could significantly reduce the overall time in computation. RBSparse and BHSparse are compared. BHSparse is the previous state-of-the-art algorithm for SpGEMM. The result shows that our algorithm is 3.1 times faster than BHSparse on average, and could achieve a maximum 14.49 times faster speed in the best scenario.

Key words: general sparse matrix-matrix multiplication; graphics processing unit; performance optimization; Hash table; shared memory

一般稀疏矩阵–矩阵乘法(SpGEMM, general sparse matrix-matrix multiplication)是给出稀疏矩阵 $m \times k$ 的 $\boldsymbol{A}$ 和 $k \times n$ 的 $\boldsymbol{B}$,求 $m \times n$ 的 $\boldsymbol{C}$ 的过程. 在许多领域,例如代数多重网格法[1]、广度优先搜索、最

收稿日期: 2018-10-09

基金项目: 中央高校基本科研业务费专项资金项目(2017RC42); IBM SUR 项目(IA2016010); 提升政府治理能力大数据应用技术国家工程实验室重点支持项目; 中国博士后科学基金面上项目(2014M550662)

作者简介: 汤 洋(1997—), 男, 硕士生; 黄智濒(1978—), 男, 讲师, 硕士生导师, E-mail:huangzb@bupt.edu.cn.

短路径问题和某些图问题(如图聚类[2]等),SpGEMM 都是算法的重要构成部分. 因此 SpGEMM 效率的提升,对于许多领域都有重要意义.

与传统的中央处理器(CPU, central processing unit)相比,图形处理器(GPU, graphic processing unit)可以提供更高的运算性能和内存带宽[3-5]. 近年来,一些作者利用 GPU 在浮点运算和内存带宽上的优势对 SpGEMM 算法进行加速. 目前已有的加速算法已取得了良好的效果,但是仍然有限,因为算法中频繁的全局内存(global memory)访问造成的内存延迟和粗放的任务分配所导致的负载极度不均衡[6-7]. 在 SpGEMM 中,某些情况下计算所产生的中间产品会达到最终实际结果数量的 10 倍甚至更多. 已有算法中,合并中间产品往往采用排序的方法,在全局内存上进行,使得计算效率低. 最近,Liu 等[8]采用寄存器和共享内存(shared memory)来进行 SpGEMM 的加速工作,效果仍不够理想.

提出了低开销的预分析方法,将矩阵的每行按照标准进行分组,以达到良好的负载平衡,并在不同情况下充分利用宝贵的共享内存. 对不同的组按行来组织计算,用共享内存上的散列表处理中间产品,以减少内存周期延迟和合并中间产品所带来的大量时间开销.

1　背景知识

1.1　GPU 编程

GPU 诞生的动机是图形处理,并在随后的发展中获得了更为一般的设计目的,即高度并行化的计算密集型场景. 笔者以 NVIDIA 公司生产的 GPU 为例进行说明.

GPU 执行任务分配的基本处理单元是多核处理器(SM, stream multiprocessor),其中包含许多流处理器(SP, stream processor)用于单一线程的执行. SM 采用单指令多线程(SIMT, single instruction multiple threads)的方式同时运行大量线程.

为了有效利用 GPU 的计算资源,同时提供一个基于硬件实现又相对统一的模型,NVIDIA 公司提出了统一计算设备架构(CUDA, compute unified device architecture)的概念. CUDA 的设计参考底层硬件设计的同时又为应用程序提供了在不同硬件上使用同一模型的便利. RBSparse 的代码开发均基于 CUDA 实现.

1.2　内存延迟

内存延迟是指等待内存数据访问完成时引起的延迟. 主要因为处理器与内存的频率差异,若某处理器主频近 4 GHz,而内存频率仅为 400 MHz,时钟速度之比为 10∶1,当处理器访问内部高速缓存之外的数据项时,每个周期必须等待 10 个时钟周期才能使内存芯片完成数据的提取和发送. 通常,这些提取需要多个内存周期,然后需要更长时间到达处理器. 这就意味着提取数据会占数百个处理器时钟周期,在此期间应用不能处理其他任何任务.

NVIDIA 公司的 GPU 上,可被操作的内存分为全局内存和共享内存. 全局内存大小一般在 GB 数量级,GPU 上任何流处理器的任何线程都可访问到它. 全局内存一般有 300 ~ 600 个流处理器时钟周期的访问延迟. 共享内存只能被同 SM 的线程访问到,即一个线程块(block)内的线程. CUDA 允许的每个线程块的最大共享内存大小为 48KB. 共享内存的访问延迟一般为 20 个左右的流处理器时钟周期.

2　相关工作及问题

2.1　BHSparse

BHSparse[6]是一种快速的 GPU 上 SpGEMM 算法,其提出了一种估计未合并的 $\boldsymbol{C}$ 矩阵(称为 $\boldsymbol{C}_{\mathrm{t}}$ 矩阵)的每行非零元素个数上限的方法. $\boldsymbol{C}_{\mathrm{t}}$ 矩阵的每行非零元素个数上限进行估计,按照估值对 $\boldsymbol{C}$ 矩阵各行进行分组申请内存,用不同排序方法对其进行处理.

某些情况下,估值与实际内存需求相差很大,某些行压缩率可达 10%,在申请内存时造成了极度浪费,从而性能下降;另一方面,按照此分组方式,某些组的排序方法无效,比如堆排序会造成线程歧义,大于 512 上限的行被粗略地统一处理,逐步申请内存,造成分歧.

2.2　BalancedHash

BalancedHash[9]利用散列表来尽量消除随机的全局内存访问,并提出了一种优化散列表性能的方法来估计输出矩阵中非零值数量,达到了平均 1.5 倍 BHSparse 性能的加速效果.

然而,BalancedHash 生成工作表占总体时间开销 1/4 左右. 另外,跨行的分块方法并不有效,某些情况下,跨行造成内存访问性能严重下降.

2.3 CUSP

CUSP[10]库使用扩展、排序和压缩(ESC, expansion, sorting and compression)的方法. 首先,算法生成或展开所有中间产品,并存储在一个数组 C^* 中. 然后,使用基数排序按行和列对 C^* 中的条目进行排序. 最后,将具有相同行和列索引条目相加以计算矩阵 $\boldsymbol{C}$ 的非零值. 该方法思路清晰,但粗放的中间产品处理办法使负载不均衡,时间成本高. Dalton 等[10]在新版本中对其做出了改进,但是不能有效地改进工作效率.

3 优化动机

3.1 SpGEMM 主要开销

首先介绍 SpGEMM 算法. 假设有 2 个稀疏矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$,须计算 $\boldsymbol{C}=\boldsymbol{A}\times\boldsymbol{B}$. $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 通常以只记录非零元素(COO, coordinate)或按行压缩(CSR, compressed sparse row)格式存储. m_{ij} 表示矩阵 $\boldsymbol{M}$ 第 i 行第 j 列位置的元素,m_{i*} 表示矩阵 $\boldsymbol{M}$ 第 i 行的所有元素,m_{*j} 表示矩阵 $\boldsymbol{M}$ 第 j 列的所有元素.

首先,将矩阵 $\boldsymbol{C}$ 中的所有值初始化为空. 其次,对于 $\boldsymbol{A}$ 的每一行,遍历行中的非零元素. 如果元素位于第 j 列,继续遍历矩阵 $\boldsymbol{B}$ 的第 j 行. 给定 b_{j*} 中的非零元素,比如在第 k 列,计算一个中间乘积 value $=a_{ij}b_{jk}$. 然后,检查 c_{ik} 是否仍然为空,若是,设置 $c_{ik}\leftarrow$value;否则,设置为 $c_{ik}\leftarrow c_{ik}+$value. 最后,遍历一行 a_{i*} 后,生成输出行 c_{i*} 中的所有值. 算法 1 中展示了该过程.

算法 1 SpGEMM 的伪代码

```
1  for each a_{i*} in the matrix A do
2    set c_{i*} to empty
3    for each nonzero entry a_{ij} in a_{i*} do
4      load b_{j*}
5      for each nonzero entry b_{jk} in b_{j*} do
6        value ←a_{ij}b_{jk}
7        if c_{ik} is not belong to c_{i*} then
8          insert c_{ik} to c_{i*}
9          c_{ik}←value
10       else
11         c_{ik}←c_{ik} + value
12       end if
13     end for
14   end for
15 end for
```

算法 1 中,在 SpGEMM 的一行的运算中,$\boldsymbol{A}$ 中的每个非零元素与 $\boldsymbol{B}$ 中对应列的非零元素做乘法后,需要进行插入操作,放置在临时的 $\boldsymbol{C}_t$ 矩阵的对应行的相应位置,若该位置存在元素,则与之合并. 总之,SpGEMM 算法的主要时间开销集中在算法 1 的第 7 ~ 11 行,即合并和排序的过程. 现有 GPU 上 SpGEMM 算法大都在内存延迟更高的全局内存上组织计算.

3.2 按行组织计算

在 SpGEMM 的运算过程中(不考虑运算过程中中间产品的暂存方法),跨行组织运算——每一块任务的分割边界在一行内部,会造成内存访问无法尽可能高效地使用共享内存.

式(1)给出了较小矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 便于说明原理,a_{xy} 表示矩阵 $\boldsymbol{A}$ x 行 y 列位置的元素. 如果在运算时,a_{23} 至 a_{31} 的元素 c, 0, d 被分配到一个 block[1]进行运算,a_{32} 至 a_{34} 的元素 0, e, 0 被分配到下一个 block[2]进行运算.

$$\boldsymbol{A}=\begin{bmatrix}0&0&a&0\\0&b&c&0\\d&0&e&0\\0&0&0&f\end{bmatrix},\boldsymbol{B}=\begin{bmatrix}0&h&0&0\\i&j&0&0\\0&k&0&l\\m&0&0&n\end{bmatrix}\tag{1}$$

对于 block[1],首先,矩阵 $\boldsymbol{A}$ 中第 2 行元素 c 和矩阵 $\boldsymbol{B}$ 中第 3 行非零元素 k, l 做乘法,分别得到 $c_{22}=ck$, $c_{24}=cl$. 然后,矩阵 $\boldsymbol{A}$ 中第 3 行元素 d 和矩阵 $\boldsymbol{B}$ 中第 1 行元素 h 做乘法,得到 $c_{32}=dh$. 此时,block[1]所分配的共享内存被写入 $c_{22}=ck$, $c_{24}=cl$, $c_{32}=dh$.

对于 block[2],矩阵 $\boldsymbol{A}$ 中第 3 行元素 e 与矩阵 $\boldsymbol{B}$ 中第 3 行元素 k, l 做乘法,分别得到 $c_{32}=ek$, $c_{34}=el$. 可以看到,该块运算存储 c_{32} 需要与 block[1]中的数据求和,造成了 2 个方面的问题:共享内存并不能跨线程块访问,为了解决该问题,必须在每个线程块运算完成后立即将数据拷贝到全局内存中再进行操作,这将付出高额的时间代价;不同线程块的数据拷贝到共享内存中后,需要进行下一步的合并、排序操作. 因此,在 RBSparse 中按照行来组织运算.

4 主要方法

文中所指的稀疏矩阵乘法问题为 $\boldsymbol{C}=\boldsymbol{A}\times\boldsymbol{B}$ 在 $\boldsymbol{A}=\boldsymbol{B}$ 时的情况.

基本思路是:首先,对矩阵每行的计算情况和整体特性进行分析;其次,按照划分标准将计算量相近

的行的行号记录至散列表大小不同的组中；然后，同时发射几个内核，按行组织计算，每个线程块计算一行，充分使用共享内存，发生溢出的行统一进行常规计算；最后，将计算结果进行回测，测试通过后复制回主机内存.

4.1　行计算密度

使用单个 GPU 线程来计算数组 nnz_RowCt 中的每个元素. 算法 2 描述了该过程.

算法 2　计算 C 矩阵中间产品行上限的伪代码

```
1  for each entry nnz_RowCt_i in nnz_RowCt in parallel do
2     nnz_RowCt[i] ← 0
3     for each nonzero entry a_ij in a_i* do
4        nnz_RowCt[i] ← nnz_RowCt[i] + nnz(b_j*)
5     end for
6  end for
```

创建一个大小为 m 的数组 nnz_RowCt，其中 $m=n$ 是矩阵 $\boldsymbol{C}$ 的行数，nnz_RowCt[i] 存储了矩阵 $\boldsymbol{C}$ 第 i 行的中间产品数量上限，称之为行计算密度，nnz(b_{j*}) 为矩阵 $\boldsymbol{B}$ 第 j 行的非零元素个数(下同).

4.2　散列表

如图 1 所示，算法定义每个散列表最多可以存储多少个中间变量为散列表的大小. 在数组 idx 中写入结果的列标，在数组 val 中写入结果的值.

idx[0]	idx[1]	idx[2]	idx[3]	…	idx[127]
val[0]	val[1]	val[2]	val[3]	…	val[127]

图 1　散列表在共享内存上的示意图

每个 idx[i] 为一个整型，占 4 Byte，每 val[i] 为一个双精度浮点型，占 8 Byte，即每个位置需要 12 Byte. 当散列表大小为 128 时，占用 1.5 KB 的共享内存. 在实验中使用的 NVIDIA GTX 1070 上，每个线程块上支持的最大共享内存大小为 48 KB. 当散列表大小为 4 096 时，占用共享内存为 48 KB，是最大情况.

4.3　计算的均衡特征

根据矩阵 $\boldsymbol{C}$ 中间产品每行非零元素个数上限的方差来确定该矩阵计算的均衡特性. 定义当方差除以 $\boldsymbol{C}_t$ 每行非零元素平均值小于等于 0.5 时(即 $\frac{\text{variance\_nnzCt}}{\text{avg\_nnzCt}} \leqslant 0.5$)，为均衡计算型矩阵；当方差除以 $\boldsymbol{C}_t$ 矩阵每行非零元素平均值大于 0.5 时(即 $\frac{\text{variance\_nnzCt}}{\text{avg\_nnzCt}} > 0.5$)，为非均衡计算型矩阵.

对于均衡计算型矩阵，因为矩阵 $\boldsymbol{C}$ 中间产品每行非零元素个数上限(下文称为行计算密度)的方差不大，即在整体计算过程中每行的计算量波动并不大，所以在计算过程中可以使用一个统一的散列表大小进行计算，而不会造成共享内存的浪费.

详细地讲，对于非均衡计算型行矩阵，计算过程中每行计算量的波动相对较大，例如极端情况下，某 $\boldsymbol{C}_t$ 矩阵的上限个数在少的行为 32，在多的行为 4 096，而且这些行还都在矩阵计算中占据了不可忽视的比例.

如果统一设定散列表大小为 4 096，对于分到行计算密度为 1 的行的线程块，分配的 num_threads 个线程在单位时间内最多只有一个在运行状态，这样就会造成极大的资源浪费.

如果统一设定散列表大小为 32，对于分到行计算密度为 4 096 的行任务的线程块，散列表就会出现严重的冲突和溢出，这样也会使得整体性能发生显著下降. 因此，对于非均衡计算型矩阵应根据行计算密度的大小进行分区，将行计算密度接近的行划分为一组，以一个统一的散列表大小进行运算，以达到尽可能好的资源利用，提升计算性能.

4.4　散列表大小的分组

将散列表按大小分为 6 组，即 Group[0] ~ Group[5]，大小分别为 32、64、128、256、512 和 2 048，用 $\frac{\text{nnz\_RowCt}}{\pi}$ 的值来区分. 因为 nnz_RowCt 是某行中间产品的计算上限，即当该行产生的所有元素都不发生合并时，将占用最大情况的内存.

但是，实际大多数情况下，中间产品都将发生合并，所以使用 nnz_RowCt 除以 π，这样每个块内散列表既不会发生严重的冲突，又可以尽可能多地节省计算资源，使一个 SM 尽可能多地同时进行计算.

1) 预处理

读取 COO 格式的矩阵 $\boldsymbol{A}$，并以 CSR 格式写入 GPU 全局内存中，以便使用.

2) 预分析

① 计算行计算密度

进行计算密度 nnz_RowCt 数组的计算，以供使用和参考.

② 计算方差

计算 nnz_RowCt 的方差，用于区分均衡计算型和不均衡计算型 2 种矩阵，并以不同的处理方法进

行计算.

3) 分组

① 均衡计算型\非均衡计算型矩阵

根据$\frac{\text{variance_nnzCt}}{\text{avg_nnzCt}}$的值判断当次计算的均衡特征. 均衡计算型矩阵采用统一的散列表大小进行计算,非均衡计算型矩阵采用分组散列表大小进行计算.

② 非均衡计算型矩阵不同的散列表大小

根据$\alpha=\frac{\text{nnz_RowCt}[i]}{\pi}$的大小来确定第$i$行分至哪个组进行计算.

当$0\leqslant\alpha<32$时,该行被分配至Group[0];

当$32\leqslant\alpha<64$时,该行被分配至Group[1];

当$64\leqslant\alpha<128$时,该行被分配至Group[2];

当$128\leqslant\alpha<256$时,该行被分配至Group[3];

当$256\leqslant\alpha<512$时,该行被分配至Group[4];

当$512\leqslant\alpha$时,该行被分配至Group[5].

4) 计算

以散列表大小为128为例. RBSparse在主机上发射内核(在GPU上执行的代码),并将指定的参数(线程块数目和每块线程数目)和数组信息一并传递.

① 计算与合并

利用共享内存上的散列表来实现计算与合并的过程,最初散列表的idx数组被全部初始化为⊥,val数组被全部初始化为0.

首先,将矩阵$\boldsymbol{A}$的一行的非零元素分给每个线程,每个线程处理一个非零元素. 每个线程遍历与该元素a_{ij}需要做乘法的b_{jk},做乘积之后得到value,对下标k做散列,得到h_idx=hash(k),散列函数采用简单的取模运算,将得到的value写入已申请好的散列表的h_idx位置. 如此处没有结果,则直接写入k至idx[k],value至val[k];如果此处已有结果,若k和idx[h_idx]相等,则将value累加至val[k],若k和idx[h_idx]不相等,则对散列表中的下一位置进行同样的判断,直到寻找到可以直接写入或合并的位置. 如果遍历完整个散列表还未找到可存储的位置,则将当前行标记为无效行,稍后以传统方法统一处理无效行. 算法3中的伪代码展示了该过程.

算法3 每个线程向散列表插入中间值的伪代码

```
1  value ← a_ij b_jk
2  h_idx ← hash(k)
3  for step in 1 to hash_size do
4    h←(h_idx + step) mod hash_size
5    if atomicCAS(idx[h], ⊥, k) = k then
6      atomicAdd(val[h], value)
7      break
8    end if
9  end for
10  if step = hash_size then
11    mark row i as invalid
12  end if
```

② 排序

对于已经在共享内存中完成计算和合并过程的行,对其按照列标升序进行排序. 这里笔者对基础的双调排序(bitonic sort)[11]算法进行改进,以使其在RBSparse中得以高效运行,满足算法的排序需求. 在此,不再做详细叙述.

排序完成后,算法将结果拷贝至全局内存中,清空共享内存,继续其他行的运算.

5) 归集

RBSparse对每行使用一个线程,将GPU全局内存中的不同行的数据拷贝到主机(host)内存的连续空间中.

6) 测试

使用CUSP库进行相同矩阵的乘法运算,并将RBSparse的计算结果与其进行比对,比对成功则完成测试.

5 实验方法

5.1 实验平台

实验过程中所用详细配置如表1所示.

表1 用于实验的计算机配置详情

配置类别	配置详情
中央处理器	One Intel i7-7700k (four Sandy Bridge cores, 4.2 GHz, boost up to 4.5 GHz, Hyper-Treading on, 8MB L3 cache)
系统内存	8GB DDR4-2400
图形处理器	NVIDIA GeForce GTX 1070 (15 SMs, 1920 CUDA cores, 1 506 MHz, 6.5 TFlops in single precision, 203 GFlops in double precision)
图形处理器显存	8 GB GDDR5 (256 GB/s bandwidth)
操作系统与库	Ubuntu Linux 16.04, NVIDIA CUDA SDK 9.1, CUSP 0.5.1 and GPU driver version 390.48

5.2　所用矩阵

实验中使用的矩阵如表 2 所示，都来自于佛罗里达大学矩阵集合[12]，涵盖了理论化学、量子化学、计算流体动力学、2D/3D 问题等各个领域.

表 2　用于实验的矩阵

输出图	矩阵名	行数/10^3	A 非零元素数 每行平均方差	C_t 非零元素数 每行平均方差	C 非零元素数 每行平均方差	压缩率方差
	2cubes_sphere	101	1 647 264 16. 23 13. 968 620	6 157 486 63. 63 0. 099 269	3 128 782 30. 83 0. 017 523	0. 523 481 0. 012 310
	accelerator	121	2 624 331 21. 65 45. 6244 93	14 129 784 116. 59 0. 184 198	4 316 033 35. 61 0. 004 135	0. 287 550 0. 028 592
	boneS01	127	5 516 602 43. 36 129. 558 166	95 449 761 750. 25 1. 068 799	11 161 908 87. 73 0. 009 775	0. 152 355 0. 013 035
	boydl	93	1 211 231 12. 98 179. 685 653	652 246 6. 99 179. 685 653	652 246 6. 99 179. 685 653	1. 000 000 0. 000 000
	bratu3d	28	173 796 6. 25 11. 406 695	6 632 0. 24 0. 000 274	6 632 0. 24 0. 000 274	0. 228 987 0. 176 552
	bundlell	11	770 811 72. 84 11. 406 695	6. 632 0. 24 0. 000 274	6. 632 0. 24 0. 000 274	0. 228 987 0. 176 552
	cant	63	4 007 383 64. 16 53. 815 692	69 049 153 1 105. 65 19. 539 461	7 844 845 125. 62 0. 248 661	0. 118 912 0. 003 197
	consph	83	6 010 480 72. 15 118. 749 310	115 397 187 1 384. 76 1. 631 751	11 632 918 139. 59 0. 004 607	0. 144 055 0. 034 023
	denormal	90	1 156 224 12. 93 0. 092 776	4 340 610 48. 55 0. 001 021	1 682 781 18. 82 0. 000 163	0. 388 336 0. 000 148
	economics	207	1 273 389 6. 17 19. 676 901	7 556 897 36. 60 0. 005 404	6 704 899 32. 47 0. 006 113	0. 888 727 0. 008 474
	hood	221	5 494 489 24. 91 133. 085 478	93 825 449 425. 43 1. 315 198	10 289 595 46. 66 0. 006 323	0. 169 108 0. 024 875
	k3plates	11	378 927 34. 12 30. 701 306	13 268 493 1 194. 61 3. 409 709	1 040 143 93. 65 0. 016 769	0. 078 974 0. 000 029
	mac_econ_ fwvd500	206	1 273 389 6. 17 19. 676 901	7 556 897 36. 60 0. 005 404	6 704 899 32. 47 0. 006 113	0. 888 727 0. 008 474
	majorbasis	160	1 750 416 10. 94 0. 177 112	19 178 064 119. 86 0. 035 028	8 243 392 51. 52 0. 005 822	0. 429 995 0. 000 012
	offshore	260	2 251 231 8. 67 9. 759 520	18 197 612 70. 05 0. 001 254	8 920 914 34. 34 0. 000 336	0. 509 730 0. 007 273

续表 2

输出图	矩阵名	行数/10^3	A 非零元素数 每行平均方差	C_1 非零元素数 每行平均方差	C 非零元素数 每行平均方差	压缩率方差
	poisson3Da	14	352 762 26. 10 189. 460 809	11 768 678 870. 85 6. 173 953	2 957 530 218. 85 0. 045 693	0. 267 852 0. 001 975
	protein	36	2 190 591 60. 15 753. 805 496	120 201 916 3 300. 71 1 809. 331 214	7 867 946 216. 05 3. 324 496	0. 071 277 0. 000 528
	pwtk	218	11 524 432 52. 88 31. 571 147	155 917 778 715. 49 24. 391 671	14 585 029 66. 93 0. 143 880	0. 098 535 0. 001 169
	scircuit	171	958 936 5. 61 19. 291 088	8 676 313 50. 74 0. 012 775	5 222 525 30. 54 0. 004 120	0. 548 646 0. 022 945
	specular	1. 6	7 647 040 15. 99 0. 000 000	122 361 856 256. 00 0. 000 000	19 646 572 41. 10 0. 000 100	0. 160 561 0. 000 550
	spheres	83	3 046 907 36. 56 118. 749 310	115 397 187 1 384. 76 1. 631 751	11 632 918 139. 59 0. 004 607	0. 144 055 0. 034 023
	thread	30	4 444 880 149. 47 2 596. 651 850	127 177 905 4 276. 90 738. 798 235	8 179 128 275. 06 0. 242 639	0. 099 723 0. 004 154
	trdheim	22	1 935 324 87. 57 314. 562 586	47 937 280 2 169. 30 13. 180 911	3 180 579 143. 93 0. 001 667	0. 073 440 0. 000 684
	TSOPF_RS_ b300_c1	14. 5	1 474 325 101. 41 10 455. 580 372	11 526 689 792. 87 14. 061 423	6 665 142 458. 46 3. 655 736	0. 603 461 0. 004 312
	vibrobox	12	301 700 24. 47 68. 343 887	2 561 491 207. 78 0. 334 558	807 378 65. 49 0. 000 510	0. 362 671 0. 008 947

6 实验结果

6.1 RBSparse 与 BHSparse 性能对比

对比了 RBSparse 算法和现有最快的 BHSparse 算法的性能,如图 2 和图 3 所示,以 BHSparse 的性能为 1,RBSparse 的速度为其倍数来展示.

提出的 RBSparse 在实验 1 和 2 中达到了平均 3. 1 倍的性能表现. 在 TSOPF_RS_b300_c1 矩阵上的表现达到了最佳的 14. 49 倍的性能表现.

6.2 RBSparse 与 BalancedHash 性能对比

由于 Pham 等[9]并没有公开 BalancedHash 的代码,所以无法直接比较. 但是在与其相同的测试矩阵上,RBSparse 达到了平均 3. 32 倍 BHSparse 的性能,优于 BalancedHash 的平均 1. 5 倍性能.

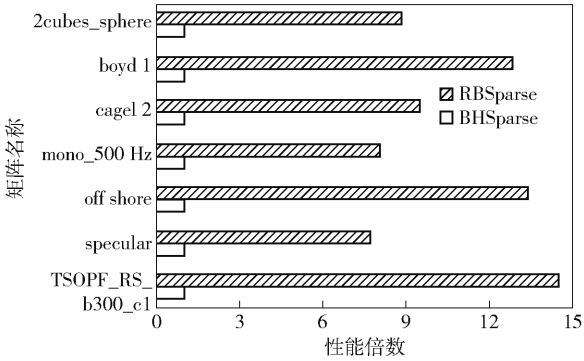

图 2 RBSparse 与 BHSparse 性能对比(实验 1)

7 结束语

提出了利用散列表和高速的共享内存并按行组

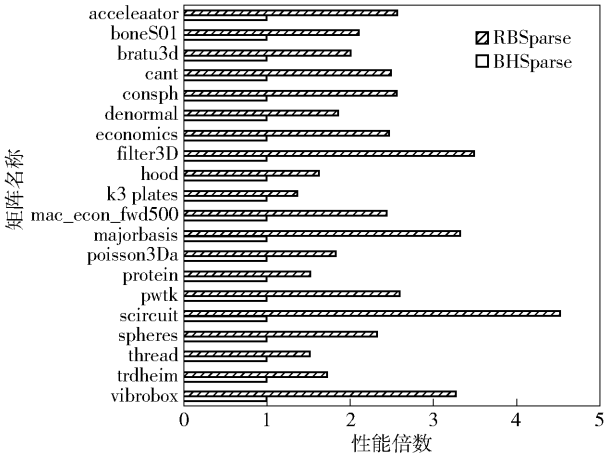

图 3　RBSparse 与 BHSparse 性能对比(实验 2)

织计算来提高 GPU 上的 SpGEMM 的性能. RBSparse 算法比目前最快的 GPU SpGEMM 算法 BHSparse 执行速度平均快 3.1 倍. 在以后的研究中,将进一步研究稀疏矩阵的结构和 SpGEMM 的复杂性之间的关系,以提高算法的性能,并使其性能在不同的矩阵上得到较为均衡的提升,以及将 RBSparse 集成到一个通用的 GPU 稀疏矩阵库中,研究其在实际应用中的使用.

参考文献:

[1] Bell N, Dalton S, Olson L N. Exposing fine-grained parallelism in algebraic multigrid methods[J]. SIAM Journal on Scientific Computing, 2012, 34(4): 123-152.

[2] Buluç A, Gilbert J R. The combinatorial BLAS: design, implementation, and applications[J]. International Journal of High Performance Computing Applications, 2011, 25(4): 496-509.

[3] Yuan Tao, Huang Zhibin. Shuffle reduction based sparse matrix-vector multiplication on Kepler GPU [J]. International Journal of Grid and Distributed Computing, 2016, 9(10): 99-106.

[4] Greathouse J L, Daga M. Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format[C]// Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. New York: IEEE Press, 2014: 769-780.

[5] 菅立恒, 易卫东. 使用 GPU 加速无线传感器网络信道仿真[J]. 北京邮电大学学报, 2013, 36(2): 24-27.
Jian Liheng, Yi Weidong. Acceleration of simulation of radio channel in wireless sensor networks using GPU [J]. Journal of Beijing University of Posts and Telecommunications, 2013, 36(2): 24-27.

[6] Liu Weifeng, Vinter B. An efficient GPU general sparse matrix-matrix multiplication for irregular data[C]//2014 IEEE 28th International Parallel and Distributed Processing Symposium. New York: IEEE Press, 2014: 370-381.

[7] 黄智濒, 周锋, 马华东. 自适应访问模式的缓存替换策略[J]. 北京邮电大学学报, 2016, 39(3): 44-48.
Huang Zhibin, Zhou Feng, Ma Huadong. A cache replacement policy adapting to the request access pattern [J]. Journal of Beijing University of Posts and Telecommunications, 2016, 39(3): 44-48.

[8] Liu Junhong, He Xin, Liu Weifeng, et al. Register-based implementation of the sparse general matrix-matrix multiplication on GPUs [J]. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York: ACM, 2018: 407-408.

[9] Anh P N Q, Fan Rui, Wen Yonggang. Balanced Hashing and efficient GPU sparse general matrix-matrix multiplication[C]// Proceedings of the 2016 International Conference on Supercomputing. New York: ACM, 2016: 36.

[10] Dalton S, Bell N, Olson L, et al. CUSP: generic parallel algorithms for sparse matrix and graph computations: Version 0.5 [EB/OL]. (2015-03-13) [2018-05-30]. https://cusplibrary.github.io.

[11] Batcher K E. Sorting networks and their applications [C]// Spring Joint Computer Conference. New York: ACM, 1968: 307-314.

[12] Davis T A, Hu Yifan. The University of Florida sparse matrix collection[J]. ACM Transactions on Mathematical Software, 2011, 38(1): 1.