异构平台上稀疏矩阵计算问题的计算结构分析

一、异构平台上稀疏矩阵计算问题简介

稀疏矩阵与向量乘（SpMV）是迭代法求解稀疏线性方程组的核心运算，且在现在流行的大数据计算、深度学习训练模型等各个工程大规模运算领域，所用到的数据普遍都是稀疏矩阵。因此，稀疏矩阵与向量相乘的高效率计算方法的研究一直都是计算科学方面的热点问题[1]。因为矩阵的计算性能和矩阵的存储结构密切相关，为了充分利用处理器的处理性能，我们需要结合系数矩阵中的非零元素分布特性，来选择合适的存储格式，在降低空间开销的同时还能降低问题的计算规模。

另外，为了实现稀疏矩阵的高性能计算，研究者常常利用不同平台来进行 SpMV 的计算研究，如向量处理机、多核 CPU 和 GPUs[2,3]，以获得更高的计算加速比。目前并行计算 SpMV 方面的研究还主要集中在把现有的 SpMV 实现分别调度到 CPU 和 GPU 上。并行编程框架一直是高性能计算的研究热点，针对多核和多 CPU 的 OpenMP 并行编程技术[4,5,6]得到了广泛的使用，而对于典型应用领域产生的稀疏矩阵存储格式的重新设计，和计算任务分配调度的研究尚少。近年来，一些不同于 CPU 的异构处理器作为加速器提供了强大的并行计算能力，例如 IBM 的 Cell 处理器，FPGA，GPU 等[7,8]。在这些加速器当中，GPU 也因其拥有高能效和经济性得到了广泛使用，并且基于 GPU 的编程模型也较为成熟，再加上 CPU 的高频率且逐步多核化的特点，SpMV 能够通过 CPU+GPU 这样的异构计算平台提升计算性能，且在平台上也有着较高的能效比。

本文将介绍一种在混合平台上的稀疏矩阵运算方法，它提出了针对于 GPU 多核特性的稀疏矩阵分割策略[8]，之后根据 GPU 中不同计算核心 Knernel 的计算能力来进行合适的任务划分。

二、稀疏矩阵计算问题的分块求解算法

2.1 稀疏矩阵的几种存储格式

我们统一使用矩阵 A 来作为例子，说明不同存储格式的存储方法。A 为

4×4的稀疏矩阵[9, 10]，如下所示：

$$\mathrm{A} = \begin{pmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 4 & 0 \end{pmatrix}$$

COO 存储格式：采用三元组来存储稀疏矩阵中的非零元素，其中 row 为元素所在的行，column 为元素所在的列，value 为元素的值。例 A 的 COO 存储格式如下所示：

$$row = (0 \quad 0 \quad 1 \quad 1 \quad 2 \quad 2 \ \ldots)$$
$$column = (0 \quad 1 \quad 1 \quad 2 \quad 0 \quad 2 \ \ldots)$$
$$value = (1 \quad 7 \quad 2 \quad 8 \quad 5 \quad 3 \ \ldots)$$

CSR 存储格式：是一种行压缩存储方法。对于一个长度为 k 的数组A_v按行顺序存储 A 中的所有非零元素值，另外一个长度也为 k 的数组A_j对应存储每一个非零元素在行中的列号，还有一个长度为 n+1 的数组A_p，其中元素 i 存放 A 中第 i 行之前的非零元素的数目。 A 的 CSR 存储格式如下：

$$A_p = (0 \quad 2 \quad 4 \quad 7 \quad 9)$$
$$A_j = (0 \quad 1 \quad 1 \quad 2 \quad 0 \quad 2 \ \ldots)$$
$$A_v = (1 \quad 7 \quad 2 \quad 8 \quad 5 \quad 3 \ \ldots)$$

ELL 存储格式：利用了一个$n \times k$的矩阵来存储数据，k 为包含非零元素最多行的非零元素数目。ELL 采用一个$n \times k$的数据矩阵 EData 来存储每一行非零元素的值，和一个$n \times k$的列偏移指针矩阵 Offset 来存储对应位置元素的列号。而对于矩阵中的零元素，Offset 采用-1 来填充。这一格式不适用于每行非零元素个数相差太大的稀疏矩阵。A 的 ELL 存储格式如下：

$$\mathrm{Offset} = \begin{pmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 1 & 2 \\ 1 & 3 & * \end{pmatrix}, \mathrm{EData} = \begin{pmatrix} 1 & 7 & * \\ 2 & 8 & * \\ 5 & 3 & 9 \\ 6 & 4 & * \end{pmatrix}$$

2.2 稀疏矩阵分割策略

要实现 SpMV 的并行计算，需要把稀疏矩阵分割为若干块，构建多个计算子任务，然后把这些子任务分配到不同的计算核心上实现并行计算。要想实现较好的并行效率，不但需要针对并行计算体系结构特征构建与之相适应 SpMV 的并行算法框架，还需要进行合理的任务分割和映射，以提高计算资源利用率。

总体来看算法有以下几步：1、构建稀疏矩阵的元素分布函数。2、利用构造的分布函数对矩阵进行分块。3、对分块后的矩阵进行重排序，使得非零元素个数相近的块排列在一起。4、针对不同处理核心的计算能力，给他们分配对应权重的块进行运算。

2.2.1 分布函数 DF（Distribution Function）

根据稀疏矩阵的分布模式，采用合适的存储格式，有助于提高 SPMV 的性能。我们可以用 DF 精确描述稀疏矩阵的分布模式，并从 DF 中得到稀疏分布的数值特征[11]。通过稀疏分布的数值特征，可以将合适的块体从稀疏矩阵中分割出来。假设我们有矩阵 A，可以做如下划分：

$$A_{N\times M}=\begin{bmatrix} a_{11} & \cdots & a_{1M} \\ \vdots & \ddots & \vdots \\ a_{N1} & \cdots & a_{NM} \end{bmatrix}=\begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_N \end{bmatrix}$$

如上图所示，$r_1, r_2, \ldots, r_N$为 N 个 M 维行向量，他们一同组成了矩阵 A。

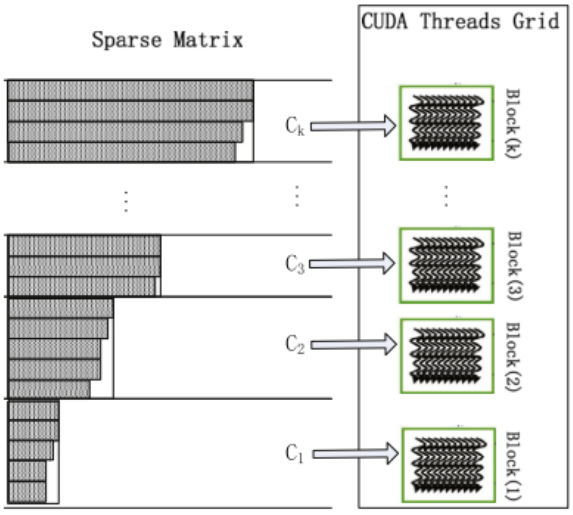
定义R_m为一个集合，该集合用于存储矩阵 A 的行向量，其中R_m存放的向量具有 m 个非零元素。所以，$A = R_1 \cup R_2 \cup \ldots \cup R_M$。将这些集合$R_1, R_2, \ldots, R_M$称为行向量集（RVS）。我们可以基于此，于是把 DF 定义为：

$$f_A: \{1,2,\ldots,M\} \rightarrow \{0,1,2,\ldots,N\}$$

映射方式是：$f_A(m) = |R_m| = b_m$，即R_m集合中的向量个数定义为b_m。所以显然有：$b_1 + b_2 + \cdots + b_M = N$。举例来看，假设我们有一个矩阵 A 如下：

$$A=\begin{pmatrix} 3 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 9 & -1 & 0 & 0 & 7 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 12 & 3 & 0 & 0 & 0 & 0 & 0 \\ -1 & 8 & 0 & 2 & 0 & 5 & 0 & 2 & 7 & 9 \\ 0 & 0 & 0 & 0 & 0 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 6 & 4 & 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 2 & 0 & 5 & 0 & 0 & 8 & 0 & 1 & 0 \\ 2 & 1 & 0 & 5 & 0 & 3 & 7 & 0 & 0 & 4 \\ 0 & 0 & 3 & 0 & 0 & 7 & 0 & 0 & 0 & 0 \end{pmatrix},$$

则我们的划分可以写成：$A=\{r_1,r_2,r_3,\ldots,r_{10}\}$;我们新添加一个矩阵 B，用于存放$R_1,R_2,\ldots,R_M$的数目，则$B=\{b_1,b_2,b_3,\ldots,b_{10}\}=\{2,3,2,\ldots,0\}$.因为$\mathrm{R}_1=\{r_3,r_6\}$,$\mathrm{R}_2=\{r_1,r_4,r_{10}\}$,$\mathrm{R}_3=\{r_2,r_7\}$……我们把矩阵按分布函数 DF 划分后，每一行的非零元素个数一目了然，如下图所示：



而采用 ELL 存储格式的存储密度与非零元素息息相关，这样的划分有助于我们对矩阵的行进行重排序，即把非零元素个数相近的行排在一起，有利于我们对矩阵的行分块进行存储，在 GPU 上能有效节省存储的空间。

2.2.2 行向量分割算法

假设有 K 个处理器，每一个处理器的计算能力分别为CP_1、$CP_2\ldots CP_K$。这个系统的总计算能力就等于$CP=\sum_{i=1}^{K}CP_i$。而如何把 SpMV 的计算任务分割成为 K 个子任务分别分配给这些运算核心上呢？主要有三种划分方法，分别是：基于行的数目划分[12]，基于非零元素个数划分和基于分布密度[11]来进行划分。我们主要介绍基于分布密度模型的分割方法。

对稀疏矩阵进行压缩，ELL 压缩格式数据结构较为规则，比较适合在 GPU 上进行并行计算，在较多情况下性能较之 CSR 和 COO 要好[13]。虽然根据非零元素数目进行分割，但是分割出来的行向量块的非零元素数目偏差可能较大。

基于行分布密度是对基于非零元素个数划分的一个改进。设 A 可以分割为 q 个行向量块$B_1,B_2,\ldots,B_q$, 其中每一个行向量块拥有相同的非零元素数目。我们假设$W(B_1)<W(B_2)<\cdots<W(B_q)$。A 在分割成 q 个行向量块的时候，行的顺

序就已经重排序了。这样行向量块有较高的密度，从而降低了 ELL 存储格式中的零元素的填充。具体算法如下：

算法 4.3 基于分布密度模型的分割算法

Input: 稀疏矩阵行的数目, N; 稀疏矩阵A的行向量, $Av[1..N]$; 稀疏矩阵的非零元素数目, NNZ; 分布密度函数对应的行向量块, $B_1, B_2, ..., B_q$; $B_1, B_2, ..., B_q$的密度, $p_1, p_2, ..., p_q$; K个处理器对应的计算能力$CP_1, CP_2, ..., CP_K$.

Output: 分割得到的K个行向量块, $A_1, A_2, ..., A_K$.

```
1: for each i ∈ [1..K] do
2:   A_i ← ∅;
3: i ← 1;
4: j ← 1;
5: while (i ≤ K) do
6:   NNZ_C ← (CP_i/CP) × NNZ;
7:   NNZ_T ← 0;
8:   while (NNZ_T < NNZ_C) do
9:     if j × p_j × N ≤ (NNZ_C − NNZ_T) then
10:      A_i ← A_i ∪ B_j;
11:      NNZ_T ← NNZ_T + j × p_j × N;
12:      j ← j + 1;
13:    else
14:      row ← ⌈(NNZ_C − NNZ_T)/j⌉;
15:      B'_j ← the first row vectors from B_j;
16:      A_i ← A_i ∪ B'_j;
17:      B_j ← B_j − B'_j;
18:      p_j ← p_j − row/N;
19:      NNZ_T ← NNZ_T + row × j;
20:  i ← i + 1;
21: return A_1, A_2, ..., A_K.
```

这一种分割算法能够找到一种稀疏矩阵的分割，使得该分割得到的行向量块采用 ELL 格式存储需要的存储空间最少，也就是平均密度最高。对于一个大的计算任务分成几个子任务分别分配到不同的处理上进行同步计算，如果每个子任务在不同的处理器上计算时间相同，这样每个子任务将会同时完成，这种情况将会有最大的并行效率。因此一个稀疏矩阵按照这一方法分割为不同的行向量块分配到不同的处理器上进行运算 SpMV，因为行向量块的规模与处理器的计算能力是完全匹配的，因此每一个行向量块的计算时间将会是很一致，因此也有非常高的并行效率。

三、 稀疏矩阵计算密度分析

3.1 压缩格式的空间复杂度分析

我们定义下述变量：S_i为存储一个整数需要的存储空间，$S_s and S_d$分别表示存储一个单精度和双精度所需要的存储空间。则三类存储格式的空间复杂度分析[14]如下：

COO 压缩格式的空间复杂度：COO 格式是用三个等长的数组来存储稀疏矩阵的非零元素，其中两个是整数数组，另一个是浮点数数组。数组的长度为 NNZ。如果浮点数采用单精度存储，则对于稀疏矩阵采用 COO 格存储需要的存储空间为：

$$S_{COO} = S_s \times NNZ + 2 \times S_i \times NNZ$$

CSR 压缩格式的空间复杂度：CSR 格式也是用三个数组来存储稀疏矩阵。其中序列序号数组和值数组与 COO 格式一样，另外一个存储行指针的数组长度为 N+1。因此 CSR 压缩格式的存储空间为：

$$S_{csr} = S_s \times NNZ + S_i \times NNZ + S_i \times (N+1)$$

ELL 压缩格式的空间复杂度：ELL 格式包含两个同等规模的二维数组构成，其中一个二维数组表示列序号为整数数组，另一个为表示值，为浮点数数组。二维数组的行数与稀疏矩阵 A 行数一样，列数为$K = \max\{i|P(X=i) > 0\}$。因此 ELL 压缩格式的存储空间为：

$$S_{ell} = S_s \times N \times K + S_i \times N \times K$$

如果稀疏矩阵存在较多的全是零的行，那么将会造成严重的存储空间浪费。我们采用先分割再存储的方式来存储，将能提高存储密度。

3.2 行向量块分布密度与 SpMV 计算密度分析

我们定义A_i为从矩阵 A 分割出来的 RVS，其中 r 为A_i中的一个行向量。定义 NNZ 为 r 中的非零元素数目，那么A_i中的非零元素数目为：$\mathrm{NNZ}(A_i) = \sum_{r\epsilon A_i} NNZ(r)$[11, 12]。定义$A_i$的宽度为：$W(A_i) = \max\{NNZ(r)|r\epsilon A_i\}$。定义$A_i$的行向量数目为$N(A_i) = |A_i|$。我们采用 ELL 来存储的话，需要存储的元素数目为：

$$E(A_i) = N(A_i) \times W(A_i)$$

所以可以定义A_i的密度为A_i中非零元素数目与作为稠密矩阵需要存储的元素数目之比，用$D(A_i)$表示，则$D(A_i)$可以利用以下公式进行计算：

$$D(A_i) = \frac{NNZ(A_i)}{E(A_i)}$$

定义零元素的填充比例$F(A_i)$为A_i中需要填充的零元素数目与整体存储的元素数目之比，可以得到$F(A_i) = 1 - D(A_i)$。假设稀疏矩阵 A 已经被分割为 K 个 RVS$(A_1, A_2, \dots, A_K)$，其中每一个行向量块A_i都可以看做为$N(A_i) \times W(A_i)$的一个子矩阵。则这种分割的平均密度为：

$$D(A_1, A_2, \dots, A_K) = \frac{NNZ(A)}{E(A_1) + E(A_2) + \cdots + E(A_K)}$$

我们以矩阵$A_{10\times10}$为例子，将含有 q 个非零元素的行在整个矩阵中出现的

概率记作p_q, 那么$p_1 = \frac{2}{10}$, $p_2 = \frac{3}{10}$, $p_3 = \frac{2}{10}$, ……, 如果整体按照 ELL 格式来存储，则存储的密度为$\frac{31}{7\times10} = 0.443$。如果按照上述 6 个 RVS 来存储，则平均密度为$\frac{31}{[E(B_1)+E(B_2)+\cdots E(B_{10})]} = 1$, 无需进行零元素填充。而且后续根据分割的密度来分配计算任务，也将提高计算性能。

通过这一种存储格式，进行稀疏矩阵向量乘，我们可以得出它的计算密度：首先，一个稀疏矩阵中的待计算元素由两个矩阵存储，共需要读取的双精度浮点数共有$2 \times E(A_i)$个，而向量的维度为 n，则共需读取 n 个双精度浮点数。计算结果写回以向量的格式，也需要 n 个双精度浮点数。则一共的访存次数为：

$$(2 \times E(A_i) + 2\mathrm{n}) \times 8 \quad (\text{字节})$$

浮点操作数目共有$NNZ(A_i)$次的乘法，则计算密度为：

$$\frac{NNZ(A_i)}{(2 \times E(A_i) + 2\mathrm{n}) \times 8}$$

四、现有实现的性能概述与理想的计算结构

4.1 执行时间分析

计算时间包括两个部分：数据从主机传输到 GPU 的时间T_t，和 GPU 上运算 SpMV 的时间T_t。根据 CUDA 线程网格的并行计算模式，拥有每行等长的 ELL 存储格式比不等长的 COO 和 CSR 格式更适合 GPU 上进行并行计算[8,14]。

假设对于采用 ELL 格式存储的稀疏矩阵，一个行向量块A_i传输到 GPU 的数据大小为$\sum_{i=1}^{K} E(A_i)$。假设存储向量块的大小为$\mathrm{DS}(A_i)$，则 DS 的计算公式为：

$$DS(A_i) = E(A_i) \times (S_s + S_i) + N(A_i) \times S_i$$

而传输时间T_t可以表示为：

$$T_t = \frac{\sum_{i=1}^{K} DS(A_i)}{PCI_e} = \frac{(S_s + S_i) \times \sum_{i=1}^{K} E(A_i) + S_i \times N}{PCI_e}$$

其中的PCI_e为连接 CPU 和 GPU 的PCI_e总线速率。针对 ELL 格式，采用密度分割算法得到的行向量块所占存储空间是最小的，因此传输时间也是最小的。对于一个稀疏矩阵, 平均零元素填充率就是一种存储空间浪费率，存储空间

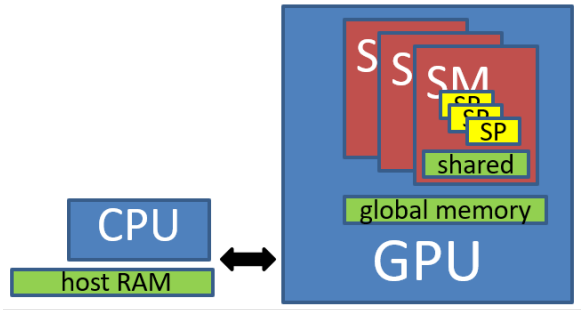
浪费率的减少可以降低数据的计算规模，减少数据访问次数，从而可以提高计算效率。

4.2 理想的计算结构与现有性能概述

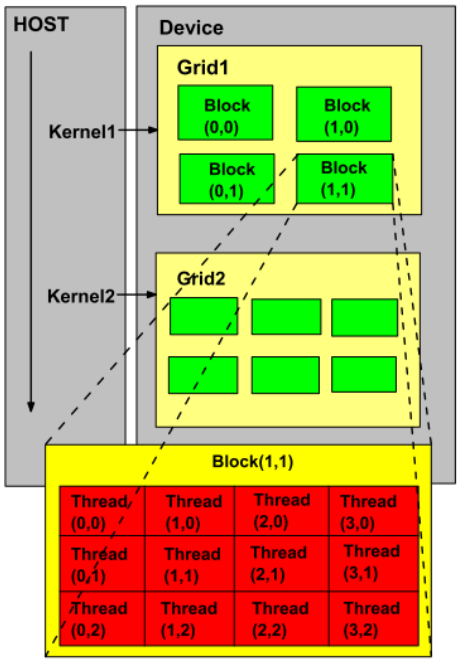
稀疏矩阵采用何种压缩格式较为合适与其稀疏模式相关，其主要是与稀疏矩阵中非零元素的分布密切相关。对于 SpMV 而言，其基本运算是稀疏矩阵中的一行与向量的积。对于一行与向量的乘，其计算性能与该行非零元素数目和排列相关，非零元素数目决定运算次数，非零元素排列影响数据访问。但是对于多行进行并行计算时，不同行的非零元素数目的差异会造成不同执行线程的计算任务的差异，对于每个线程可以独立调度的系统（多核 CPU 系统），而线程之间计算任务的不平衡一般不会因为同步而造成资源浪费，但是频繁的调度也会影响性能[16]。

而对于线程不能独立调度必须成束调度的 GPU 而言，同一束的线程之间计算任务的不均衡就会因为同步造成线程的等待而使计算资源闲置，从而使并行计算效率降低。另外分配给同一线程束的数据集的数据分布也会影响线程束的数据访问效率，比如数据块的对齐访问和连续访问等。因此针对 GPU 而言，SpMV 的并行计算性能与行的非零元素分布关联较大[11, 14]。

我们所需求的理想计算结构是 CPU+GPU 的异构平台，简单图示如下：



其中 CPU 起到任务分配调度的功能，而的 GPU 最好具有多个计算能力强大且相近的核心的 GPU 结构。CUDA 将计算任务映射为大量的可以并行执行的线程，并且硬件动态调度和执行这些线程，GPU 上的核心以线程网格（Grid）的形式组织，每个线程网格由若干个线程块组成。



因为流多处理器的计算效率依赖于分配到其上面运行的行向量块的密度，这样当我们提高稀疏矩阵的存储密度的时候，矩阵运算的计算性能就会得到更为显著的提高。

现有的实验中，在 GPU 和多核 CPU 的异构平台上，因为测试平台有两块 GPU 和两块 CPU，不过 CPU 需要分配两个核心来分别控制两块 GPU，另外 6 个核心可以参与计算。所以分割后，实验选用了 10 个测试矩阵，发现前两种分割算法得到的行向量块的非零元素数目与其分配的 GPU 和 CPU 的计算能力具有较好的匹配，且三类分割矩阵后的平均密度分别为 11%,40%,61%[8, 11, 14]。在计算效率上，三类分割法单精度平均执行时间分别减少 13.95%,37.94%和 34.31%，双精度分别为 13.10%,49.79%和 50.03%。所以本文中基于分布密度的矩阵分割算法较现有的方法有较好的普适性，能够适应各种特征不同的稀疏矩阵。而且对于 SpMV 的计算加速效果显著，能够适应 CPU、GPU 以及其他加速器组成的异构计算平台。

参考文献

[1] Chen X, Ren L, Wang Y, et al. GPU-Accelerated Sparse LU Factorization for Circuit Simulation with Performance Modeling. IEEE Transactions on Parallel and Distributed Systems, 2015. 26(3):786-795

[2] 徐敬华,高铭宇,苟华伟,张树有,谭建荣.基于非规则分块压缩的 3D 打印稀疏矩阵存储与重构方法[J/OL].计算机学报,2019:1-12[2019-12-23].

[3] High Performance Computing; Reports Outline High Performance Computing Study Findings from School of Computer Science and Technology (Sparse Matrix Partitioning for Optimizing SpMV On CPU-GPU Heterogeneous Platforms)[J]. Computers, Networks & Communications,2019.

[4] 阳王东,李肯立.基于 HYB 格式稀疏矩阵与向量乘在 CPU+GPU 异构系统中的实现与优化[J].计算机工程与科学,2016,38(02):202-209.

[5] 刘芳芳,杨超,袁欣辉,吴长茂,敖玉龙.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.

[6] 杨世伟,蒋国平,宋玉蓉,涂潇.GPU 上稀疏矩阵向量乘法优化策略[J/OL].计算机工程:1-16[2019-12-23].

[7] 尹孟嘉,许先斌,何水兵,胡婧,叶从欢,张涛.GPU 稀疏矩阵向量乘的性能模型构造[J].计算机科学,2017,44(04):182-187+206.

[8] 阳王东. CPU+GPU 异构平台上稀疏线性系统快速并行求解算法研究[D].湖南大学,2017.

[9] Shengen Yan, Chao Li, Yunquan Zhang,等. yaSpMV: yet another SpMV framework on GPUs[C]// Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming. ACM, 2014.

[10] Jeljeli H. Accelerating Iterative SpMV for the Discrete Logarithm Problem Using GPUs[J]. 2014, 9061:25-44.

[11] K. Li, W. Yang, K. Li, Performance analysis and optimization for SpMV on GPU using probabilistic modeling, IEEE Trans. Parallel Distrib. Syst. 26 (1) (2015) 196–205.

[12] P. Guo, L. Wang, P. Chen, A performance modeling and optimization analysis tool for sparse matrix vector multiplication on GPUs, IEEE Trans. Parallel Distrib. Syst. 25 (5) (2014) 1112–1123.

[13] NVIDIA, cuSPARSE library, Tech. rep., March 2015.

[14] W. Yang, K. Li, Z. Mo, K. Li, Performance optimization using partitioned SPMV on GPUs and multicore CPUs, IEEE Trans. Comput. 64 (9) (2015) 2623–2636.

[15] A.N. Yzelman, D. Roose, High-level strategies for parallel shared-memory sparse matrix vector multiplication, IEEE Trans. Parallel Distrib. Syst. 25 (1) (2014) 116–125.

[16] Wangdong Yang, Kenli Li, Zeyao Mo,等. Performance optimization using partitioned SpMV on GPUs and multicore CPUs[J]. IEEE Transactions on Computers, 2015, 64(9):2623-2636.