稀疏矩阵向量乘法在申威众核架构上的性能优化

李亿渊[1),2)] 薛 巍[1),2)] 陈德训[1),2)] 王欣亮[1)] 许 平[1)] 张武生[1),2)] 杨广文[1),2)]

[1)](清华大学计算机科学与技术系 北京 100084)

[2)](国家超级计算无锡中心 江苏 无锡 214072)

摘 要 计算机数值模拟是现代科学和技术发展的重要触发力量.在数值模拟中,求解大规模稀疏线性方程组是非常重要的一个环节.迭代求解过程中稀疏矩阵向量乘法是耗时最长的计算核心之一,存在严重的数据局部性差、写冲突、负载不均衡等问题.因此,稀疏矩阵向量乘法已经成为了当前性能优化的难点和研究热点.本文面向国产众核处理器架构,以申威 26010 国产众核处理器为平台,针对稀疏矩阵向量乘法,在线程级和指令级并行层面上进行细粒度的并行算法设计和优化实现.其核心思想是,将众核架构设计精巧的矩阵分层分块技术用于矩阵存储、访问和任务调度,在保证右端向量数据复用的同时有效实现了负载均衡,避免了申威 26010 上因频繁缓存判断和细粒度访问导致的潜在性能问题.通过对 SuiteSparse 矩阵集合中的 2710 个算例的测试,该算法可以获得与主核上的串行算法相比 11.7 倍的平均加速和 55 倍的最高加速.

关键词 申威众核处理器;并行计算;矩阵向量乘法;矩阵格式;稀疏矩阵计算

中图法分类号 TP391 **DOI 号** 10.11897/SP.J.1016.2020.01010

Performance Optimization for Sparse Matrix-Vector Multiplication on Sunway Architecture

LI Yi-Yuan[1)] XUE Wei[1),2)] CHEN De-Xun[1),2)] WANG Xin-Liang[1)]
XU Ping[1)] ZHANG Wu-Sheng[1),2)] YANG Guang-Wen[1),2)]

[1)](*Department of Computer Science and Technology, Tsinghua University, Beijing* 100084)

[2)](*National Supercomputing Center in Wuxi, Wuxi, Jiangsu* 214072)

Abstract Numerical simulation is an important trigger for the development of modern science and technology. Meanwhile, solving large-scale linear systems with the iterative methods is widely used in nowadays numerical simulations and is regarded as one of the most time-consuming components. During the solving process, the sparse matrix-vector multiplication ($\boldsymbol{Ax}=\boldsymbol{b}$, where $\boldsymbol{A}$ is a sparse matrix and both $\boldsymbol{x}$ and $\boldsymbol{b}$ are vectors; abbreviated as SpMV) is one of the most critical computing kernels. The inherent issues of SpMV, such as the poor data-locality, write-conflict, and load-imbalance, hinder the high performance achieved. Moreover, the design features of recent Chinese home-grown many-core processor Shenwei 26010, including cache-less and inefficient fine-grained memory access, make the implementation optimization of SpMV even more difficult. This work focuses on developing high-performance sparse matrix-vector multiplication algorithm, well exploiting the parallelism of thread-level and effectively resolving the issues mentioned on Shenwei 26010.

收稿日期:2018-11-13;在线出版日期:2019-09-07.本课题得到国家电网公司科技项目"适应于电力系统应用的高性能计算技术研究与开发"(合同号:XT71-19-022)资助.**李亿渊**,博士研究生,中国计算机学会(CCF)会员,主要研究方向为高性能计算. E-mail: liyiyuan18@mails.tsinghua.edu.cn. **薛 巍**,博士,副教授,中国计算机学会(CCF)高级会员,主要研究方向为大规模科学计算、量化不确定分析. **陈德训**,博士研究生,高级工程师,主要研究方向为计算机体系结构. **王欣亮**,博士,主要研究方向为高性能计算、异构计算、并行计算. **许 平**,硕士,主要研究方向为高性能计算. **张武生**,博士,高级工程师,主要研究方向为集群计算、并行计算、分布式计算. **杨广文**,博士,教授,中国计算机学会(CCF)高级会员,主要研究方向为计算机体系结构.

In order to achieve high performance of SpMV on Shenwei 26010, this paper designed a many-core SpMV algorithm based on hierarchical blocking. First, the matrix is divided into several matrix bands by rows. Every matrix band is assigned to at most one row of CPE cluster of SW26010 for calculation according to greedy strategy. This task allocation scheme can ensure load-balance across different rows of CPE cluster as much as possible. Second, the CPEs in the same row share the value of the vector $\boldsymbol{b}$ through register-level communication, and only the selected leading CPE writes the result back to memory. This method solves the problem of write-conflict. Third, each matrix band is equally divided into several sub-matrices according to the number of non-zero elements. And the sub-matrices are assigned to CPEs in the same row of CPE cluster for calculation. In this way, the load-balance across CPEs is also guaranteed. Finally, the sub-matrices are further divided into several small matrix blocks due to the limited size of the SPM. The small matrix blocks are reordered in memory according to the calculation order of each CPE. And the CPE loads both the non-zero elements in the matrix block and the associated elements in vector $\boldsymbol{x}$ together so that the cache checking before calculation can be avoided and also the data locality is well improved. Since the storage format of the matrix blocks in the memory is all reordered according to the calculation order, for each CPE, both the access to the non-zero elements of matrix blocks and the access to the vector $\boldsymbol{x}$ can be continuous and coarse-grained. With the above algorithm, both the inherent issues of SpMV and the difficulties of implementation optimization over Shenwei 26010 are all effectively resolved.

By evaluation with all the 2710 benchmarks from Suite Sparse Collection, the proposed algorithm can achieve an average speedup of 11.7 and the best speedup of 55.0, compared with the sequential method on the MPE of Shenwei 26010. Given that the total memory bandwidth of CPEs is only five times that of management processing element, it is quite good to achieve such speedups.

This work is supported by Science and Technology Fund of State Grid of China (Research and Development of High-Performance Computing Technology for Power System Applications).

Keywords　Sunway many-core architecture; sparse-matrix computation; Sparse Matrix-Vector Multiplication; Matrix format; parallel computing

1　引　言

稀疏矩阵向量乘法(Sparse Matrix-Vector Multiplication,SpMV)是现代科学计算中一个广泛使用的计算核心.例如,在数值模拟计算中,通常会使用迭代法去求解大规模稀疏线性方程组,而稀疏矩阵向量乘法的效率直接影响了整体的求解效率,故提高 SpMV 算法的性能至关重要.

虽然稀疏矩阵向量乘法是一个非常传统的科学计算问题,但由于矩阵中非零元的排布情况复杂且无规律可循,计算硬件架构不断升级,故时至今日,其并行算法的设计与优化实现依然是研究的热点.随着体系结构的变迁,在不同处理器架构(例如 x86、GPU、申威)上设计更高效的 SpMV 依然是富有挑战的研究课题.

虽然在通用处理器和 GPGPU 上 SpMV 的并行实现已有不少研究成果,但国产众核处理器申威 26010 的异构和无缓存架构,相对更低的每核访存带宽和高的细粒度访存延迟,使得在其上实现高效 SpMV 仍值得进一步深入探究.故本文面向国产申威众核架构,在线程级和指令级并行的层面上,设计了新的并行稀疏矩阵向量乘法算法(SW Sparse Matrix-Vector Multiplication,swSpMV).

该算法基于 CSR 格式,绑定相邻行形成矩阵带,并按从核上便笺式存储器的大小将矩阵带分成若干小矩阵块,以矩阵块为单位进行 SpMV 的计算任务调度,解决了数据局部性差和频繁手动缓存判断引发的性能问题;借助申威 26010 特有的寄存器通信机制,最大可能减少写入内存的次数,解决写冲突的问题;根据小矩阵块中的非零元数量来进行任务分配,提高负载均衡.

本文测试了来自 SuiteSparse[1] 矩阵集合中的2710个算例以进行算法性能的评估.测试结果显示,swSpMV可以获得与主核上的串行算法相比11.7倍的平均加速和55.0倍的最高加速,平均有效内存带宽利用率达到80%.

2 背 景

2.1 申威众核处理器26010

SW26010处理器,是当今世界领先的超级计算机"神威·太湖之光"(目前在TOP500中排名第三)上所使用的处理器,图1展示了其硬件架构.处理器计算能力可达3TFlops,理论带宽是134GB/s.每个SW26010处理器都有4个核心组(Core Group,CG),每个CG包括一个主核(Management Processing Element,MPE),以及一个由64个从核组成的从核阵列(Computing Processing Element,CPE).

每个主核包含一个大小为32KB的一级数据缓存、一个大小为32KB的一级指令缓存和一个大小为256KB的二级缓存,这3类缓存都是由硬件控制的.每个从核包含一个大小为64KB的便笺式存储器(Scratch Pad Memory,SPM),该存储器也被称为手动缓存.从核上手动缓存的延迟与带宽和主核上的一级缓存相似:在流水意义下,每个时钟周期可以读写一个向量化寄存器的数据(32字节).从核上的向量化长度对于单双精度的并不一致,单精度浮点运算的长度是128bits(4个单精度浮点数),而双精度浮点运算的长度是256bits(4个双精度浮点数).

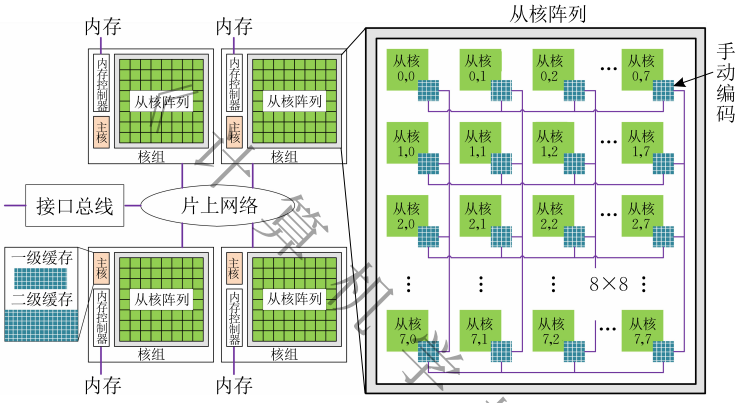


图1 SW26010硬件架构

处理器上的内存控制器支持直接内存访问(Dire-ct Memory Access,DMA)和全局读入/写出(Global load/store,gload/gstore)这两种模式的内存访问.其中DMA适合内存与从核缓存之间大量连续的粗粒度数据访问,而gload/gstore适合于内存和从核寄存器间零散的整型或浮点数的细粒度数据访问.若使用DMA访问零散的数据,则会使带宽利用率非常低.对一个核组内的64个从核进行Stream Triad测试[2],测得DMA的带宽可以高达22.6GB/s,而gload/gstore的带宽只有不到1.5GB/s.显然,若条件允许(对内存的访问应尽可能连续,每次至少访问连续的256B内存空间),应尽量使用DMA.

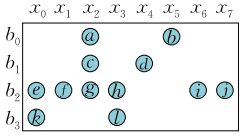
申威处理器的另一个特性是从核阵列上的低延迟(小于11个指令周期)高带宽(集合带宽达600+GB/s)的寄存器通信技术[2].从核阵列上的64个从核是按照8×8的网格排列在一起的,如图1右侧.在同行或同列的从核,可以高速互传数据.每个从核上有一个寄存器发送缓冲区,一个寄存器列接收缓冲区,和一个寄存器行接收缓冲区.硬件会将一个从核的寄存器发送缓冲区内的数据发送给另一个从核的寄存器列接收缓冲区(同列从核)或者寄存器行接收缓冲区(同行从核).这个发送过程是自动且不间断的,直到发送方的寄存器发送缓冲区已空,或者接收方的寄存器接收缓冲区已满.

1.2 稀疏矩阵向量乘法的相关工作

稀疏矩阵向量乘法是用于计算形如等式 $\boldsymbol{Ax}=\boldsymbol{b}$ 的方法,其中等式左侧的 $\boldsymbol{A}$ 表示稀疏矩阵,$\boldsymbol{x}$ 是参加计算的已知向量,右侧的 $\boldsymbol{b}$ 代表经计算后得到的结果向量.

图2左上方展现了一个简单的稀疏矩阵向量乘法的例子,其中,矩形方框代表一个4×8规模的矩阵,矩形中的字母代表矩阵中的非零元,共12个,并由字母 a 到 l 表示,矩阵上方对应位置的字母表示参与运算的向量 $\boldsymbol{x}$,矩阵左侧表示运算后的结果向量 $\boldsymbol{b}$,而向量 $\boldsymbol{b}$ 中各元素的计算公式见图2左下部分.

因在稀疏矩阵中非零元的数量非常稀少(一般非零元数量少于同规模稠密矩阵元素数的5%,甚



COO(坐标列表, Coordinate list):
row_index=[0,0,1,1,2,2,2,2,2,2,3,3]
row_index=[0,0,1,1,2,2,2,2,2,2,3,3]
value =[a,b,c,d,e,f,g,h,i,j,k,l]

CSR(稀疏行压缩, Compress Sparse Row):
row_pointer=[0,2,4,10,12]
row_index =[2,5,2,4,0,1,2,3,6,7,0,3]
value =[a,b,c,d,e,f,g,h,i,j,k,l]

CSC(稀疏列压缩, Compress Sparse Column):
row_pointer=[0,2,4,10,12]
row_index =[2,5,2,4,0,1,2,3,6,7,0,3]
value =[e,k,f,a,c,g,h,l,d,b,i,k]

$$\begin{cases} b_0=a\cdot x_2+b\cdot x_2 \\ b_1=c\cdot x_2+d\cdot x_4 \\ b_2=e\cdot x_0+f\cdot x_1+g\cdot x_2+ \\ \quad h\cdot x_3+i\cdot x_6+j\cdot x_7 \\ b_3=k\cdot x_0+l\cdot x_3 \end{cases}$$

图 2　稀疏矩阵向量乘法样例及三种基础矩阵存储格式

至低于 1%),为了避免大量零元造成的冗余计算和存储,不能使用和普通稠密矩阵一样的二维数组方式来保存.选择和使用合适的存储结构来保存稀疏矩阵中的非零元会直接影响 SpMV 的算法性能.

在一般对稀疏矩阵的处理中,通常有 4 种比较基础的稀疏矩阵存储方式,即坐标列表(Coordinate List,COO)、稀疏列压缩(Compress Sparse Column,CSC)、稀疏行压缩(Compress Sparse Row,CSR)以及 ELLPACK[3],分别如图 2 右半部分和图 3 所示.其中 COO 格式使用三元组的形式来保存每个非零元;而 CSR 与 CSC 则是在 COO 的基础上,对行坐标和列坐标进行压缩所得;ELLPACK 格式则与保存稠密阵的方式较为相似,只是矩阵的列数上进行了压缩.

ELLPACK:

$$col\_index=\begin{bmatrix} 2 & 5 & \times & \times & \times & \times \\ 2 & 4 & \times & \times & \times & \times \\ 0 & 1 & 2 & 3 & 6 & 7 \\ 0 & 3 & \times & \times & \times & \times \end{bmatrix} \quad value=\begin{bmatrix} a & b & \times & \times & \times & \times \\ c & d & \times & \times & \times & \times \\ e & f & g & h & i & j \\ k & l & \times & \times & \times & \times \end{bmatrix}$$

图 3　ELLPACK 存储格式

算法 1 是使用 CSR 格式存储稀疏矩阵的串行稀疏矩阵向量乘法,该片段也是此方向许多相关工作中的核心计算过程.程序片段中,只有第 4 行是唯一真正涉及计算的代码,而正是这行代码体现了稀疏矩阵向量乘法中固有的几大问题.

算法 1.　稀疏矩阵向量乘法 $\boldsymbol{Ax}=\boldsymbol{b}$ 基于硬件自动缓存.

输入. N, row_pointer, col_index, value, $\boldsymbol{x}$
输出. $\boldsymbol{b}$
1. for i=0 to N−1 do
2.　$\boldsymbol{b}[i]$=0
3.　for j=row_pointer[i] to row_pointer[i+1]−1 do
4.　　$\boldsymbol{b}[i]$=$\boldsymbol{b}[i]$+value[j]×$\boldsymbol{x}$[col_index[j]]
5.　end for
6. end for

数据局部性差:无论使用上述的哪一种格式来存储稀疏矩阵,在乘法计算过程中,对于向量 $\boldsymbol{b}$ 和向量 $\boldsymbol{x}$ 的访问,至少存在一个是十分随机且不连续的,故算法的数据局部性很差.

写冲突:在计算过程中,如果对一个向量 $\boldsymbol{b}$ 中的元素求值涉及多个元素的多次运算被分配在不同的线程上完成,就会导致多个线程写同一位置数据时的写冲突问题.

负载不均衡:稀疏矩阵中每行包含的非零元数量可能不同且差异很大,在由串行算法转为并行算法时,如果仅仅以行为单位来划分任务给不同的线程/进程,就会导致负载不均衡的问题,并且这一问题会随着处理器核数、线程数的增多愈发严重.

针对以上 3 个固有问题,已有很多研究提出了相应的改进方案.

基于上述 4 种基本的存储结构,有诸如 Kourtis 等人[4]提出的 CSX(拓展压缩格式),该方法针对共享内存的架构,利用矩阵内的子结构来压缩元数据;以及 Ashari 等人[5]提出的 BRC 格式,这是一种二维分块结构,以减少线程发散的问题.

同时,针对 GPU 上的工作也在不断展开.Bolz 等人[6]第 1 次在 GPU 上实现了高效的 SpMV 算法.而针对 ELLPACK 的缺点,Bell 等人[7]提出了一种名为 HYB 的混合存储格式,该格式用传统的 ELLPACK 格式来存储原稀疏矩阵中负载均衡的部分,而使用 COO 格式去存储矩阵中剩余的负载不均衡部分,这样就可以避免由于稀疏矩阵中每行非零元数量不均衡而导致的存储空间过于冗余的问题.之后也有一些研究工作尝试解决减少 ELLPACK 过量填充非零元导致的冗余问题,如 Monakov 等人[8]提出的 SELL 方法.但这些方法并没有很好地考虑到负载均衡的问题,也缺乏提高数据局部性的设计.Yang 等人[9]针对准对角矩阵提出了新的混合压缩算法以提高并行性.

基于 CSR 的存储格式,Koza 等人[10]提出了 CSMR 存储格式,该方法可以使用同一个线程来处理相邻行的数据,以此来提高数据复用.Sun 等人[11]面向稀疏对角矩阵,设计了 CRSD 格式.

为解决 CSR 存储格式的负载均衡问题,Greathouse 等人[12]提出了 CSR-adaptive 方法,Ashari 等人[13]提出了 ACSR 方法.前者将稀疏矩阵分为两部分,用不同的方式分别存储非零元多的行和非零元少的行.后者针对非零元数量非常多的行,采用了 Nvidia Kepler 框架并采用在 GPU 线程上再开启线程的机制,去进行负载均衡的处理.以上两种方法,包括 Merrill 等人[14]提出的 MCSR 方法,都是基于

原始的 CSR 格式进行的，并没有做过多的预处理. 不过从性能上看，基于 CSR 格式在 GPU 上性能最好的方法是 Liu 等人[15]提出的 CSR5 格式，如图 4 所示.

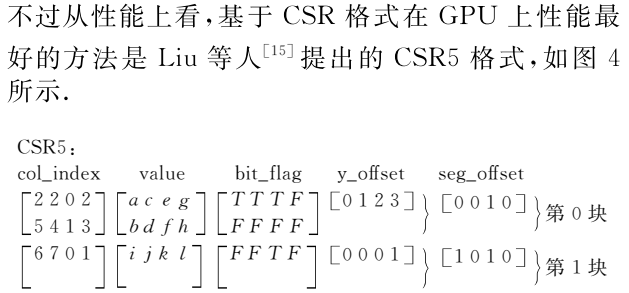


图 4 CSR5 存储格式

该方法以向量化长度为块宽度的单位，将非零元分成若干块，尽可能使每块内非零元的数量相同. 将不同的块交給 GPU 的不同线程束来完成，并且使用共享内存来解决同一个线程块内的写冲突，使用原子操作来保证跨线程块的写冲突.

另外，还有着许多工作在讨论如何使用分块的方式对稀疏矩阵进行保存，比如 Buluc 等人[16]提出的 CSB 格式，Choi 等人[17]提出的 BCSR 和 BELL 格式，以及 Liang 等人[18]提出的 HCC 格式(COO 格式+CSR 格式).

为解决 SpMV 算法在 Intel 架构上的性能瓶颈问题，Liu 等人[19]基于 ELLPACK 提出了名为 ESB 的新存储格式，该格式通过对非零元的位置进行编码，以减小带宽需求，并使用列分块来改善存储器访问的局部性. 除此之外，分块也被广泛用于优化 CPU 上 SpMV 性能. 通过在寄存器上[20-21]、高速缓存[20]中设置分块，利用数据复用来改善 SpMV 的空间和时间局部性问题.

算法 2. 稀疏矩阵向量乘法 $\boldsymbol{Ax}=\boldsymbol{b}$ 基于软件手动缓存系统的串行算法.

输入. N, row_pointer, col_index, value, $\boldsymbol{x}$
输出. $\boldsymbol{b}$
1. for i=0 to N−1 do
2. $\boldsymbol{b}[i]$=0
3. for j=row_pointer[i] to row_pointer[i+1]−1 do
4. if $\boldsymbol{x}$[col_index[j]] is not in SPM then
5. Swap $\boldsymbol{x}$[col_index[j]] into SPM
6. end if
7. $\boldsymbol{b}[i]=\boldsymbol{b}[i]+$value[$j$]$\times\boldsymbol{x}$[col_index[$j$]]
8. end for
9. end for

以上的研究都是针对硬件自动缓存的架构，这就导致大多研究工作都不再适合申威众核架构. 算法 2 展示了基于手动缓存的串行算法. 该算法中的矩阵同样采用了 CSR 的存储格式. 对该算法的并行优化工作，除了需解决稀疏矩阵向量乘法固有的几个问题外，还需降低显式缓存判断的巨大开销以及解决如何充分利用申威处理器上 DMA 访问的问题.

Sun 等人[22]在申威众核架构上设计了 SWC-SR-SpMV 算法. 该算法基于 CSR 存储格式，对矩阵行切片，以保证每个行片都可以保存在手动缓存中，以便对向量 $\boldsymbol{x}$ 进行数据复用；并将核组分割成更小的通信范围，以共享工作线程的公共数据. Liu 等人[23]在申威架构上也用了类似的对矩阵分块的方法. 这些分块策略更多针对发掘并发性、提升局部性和增大数据访问粒度提出，对负载均衡等问题缺乏全面考虑.

通过相关工作分析得出，需要针对申威众核处理器，设计一种全新的并行稀疏矩阵向量乘法. 该算法需要在解决手动缓存判断和内存细粒度访问问题的同时，还需要分析并改善稀疏矩阵向量乘法中固有的数据局部性差、写冲突和负载不均衡的三大问题.

在第 3 节中，本文依次提出 3 种 swSpMV 的方法，分别是固定划分法、一维划分法和二维划分法，并循序渐进地解决之前提到的这些问题.

3 swSpMV 算法设计

3.1 固定划分方法

针对申威众核处理器架构，频繁的缓存判断会对性能产生很大的影响，这一问题需要优先解决.

一个非常直观的想法就是，在进行每次计算时，需要参与到该次计算中的数据如果都能保证已经在手动缓存中，那么在计算之前就无需再进行额外的手动缓存判断了. 比如当前步计算 $\boldsymbol{A}_{i,j}\times\boldsymbol{x}_i=\boldsymbol{b}'_j$，若此时能确定 $\boldsymbol{A}_{i,j}$ 和 $\boldsymbol{x}_i$ 这两个数据都已经在手动缓存 SPM 中，那就可以不必进行缓存判断，直接进行该次乘法计算.

进而，将单次计算推广到连续多次计算. 如，假设矩阵 $\boldsymbol{A}_{i-j,k-l}$ 是原稀疏矩阵 $\boldsymbol{A}$ 的一个子矩阵小块，表示由矩阵 $\boldsymbol{A}$ 的第 i 至 j 行、第 k 至 l 列所共同包含的矩阵空间. 那么为了计算与该子矩阵小块相对应的子结果量 $\boldsymbol{b}_{i-j}$，即 $\boldsymbol{b}_{i-j}=\boldsymbol{b}_{i-j}+\boldsymbol{A}_{i-j,k-l}\times\boldsymbol{x}_{k-l}$，需要用到的数据为子矩阵小块 $\boldsymbol{A}_{i-j,k-l}$ 中包含的非零元和 $\boldsymbol{x}$ 向量的子向量 $\boldsymbol{x}_{k-l}$ 中的元素. 如果 $\boldsymbol{A}_{i-j,k-l}$ 和 $\boldsymbol{x}_{k-l}$ 中的元素已经提前被载入到缓存中，在计算时就不用再进行任何额外的缓存判断，直接使用算法 1 中的计算过程即可. 同时，在计算过程中可以发现，子向量 $\boldsymbol{b}_{i-j}$ 和 $\boldsymbol{x}_{k-l}$ 都是可以被复用的，这也提高

了数据局部性. 另外，可以通过将子矩阵 $\boldsymbol{A}_{i-j,k-l}$ 中的非零元存储在一起，并一起载入至缓存中，以达到对内存中数据粗粒度访问的目的，利用 DMA 提高在申威众核处理器上的访存效率，同时，可以延迟到子矩阵小块 $\boldsymbol{A}_{i-j,k-l}$ 中非零元都完成计算后，再将子结果量 $\boldsymbol{b}_{i-j}$ 写出至内存中，以解决将计算结果频繁写出至内存时造成的写冲突问题.

基于此，本文首先设计了固定划分方法. 该方法包括了预处理阶段和计算阶段. 前者分析 $\boldsymbol{A}$ 中的非零元排布，找到较合适的分块方式、存储结构，并按计算顺序将非零元都存储在内存中；而后者则按照新的存储顺序，依次完成每个子矩阵小块的矩阵向量乘法，从而完成整个 $\boldsymbol{Ax}=\boldsymbol{b}$ 的计算.

下面先介绍算法的预处理部分——构造最合适的固定划分存储结构，该存储结构包含了原矩阵-子矩阵-矩阵小块 3 个层次.

预处理第 1 步：假设处理器包含了 ROWS×COLS 个从核，则将原稀疏矩阵均分成 ROWS×COLS 个子矩阵，并分配给所有从核.

预处理第 2 步：设某从核负责处理的子矩阵是 $\boldsymbol{A}_{i-j,k-l}$，且该从核上的缓存大小只能够同时存储 LX 个 $\boldsymbol{x}$ 向量的元素和 LB 个 $\boldsymbol{b}$ 向量的元素，则将子矩阵 $\boldsymbol{A}_{i-j,k-l}$ 进一步划分成若干个 $LB\times LX$ 的子矩阵小块，子矩阵 $\boldsymbol{A}_{i-j,k-l}$ 中的矩阵小块的数量和排布由非零元的数量和排布所决定.

预处理第 3 步：将每个子矩阵小块中的非零元存储在一起，以适应申威处理器上的 DMA 操作. 每个从核负责的所有小块也存储在一起，方便进行预取操作. 小块的存储顺序和计算的顺序一致. 即在原矩阵 $\boldsymbol{A}$ 中同行的小块连续排布存储，不同行的小块，则根据行坐标的顺序依次排布.

预处理第 4 步：需要记录每个小块负责计算的 b 元素的流向. 假设 $\boldsymbol{A}_{i'-j',k'-l'}$ 是子矩阵 $\boldsymbol{A}_{i-j,k-l}$ 中第 i' 至第 j' 行的最后一个矩阵小块，且在矩阵空间上，在矩阵 $\boldsymbol{A}_{i'-j',k'-l'}$ 前，还有属于第 id 号从核的其他小块 $\boldsymbol{A}_{i'-j',k''-l''}$，则在计算完小块 $\boldsymbol{A}_{i'-j',k'-l'}$ 后，需要将从核 id 上由 $\boldsymbol{A}_{i'-j',k''-l''}$ 计算得到 $b_{i'-j'}$ 累加到本从核上计算完成的 $b_{i'-j'}$ 上；同样的，假设 $\boldsymbol{A}_{i'-j',k'-l'}$ 是子矩阵 $\boldsymbol{A}_{i-j,k-l}$ 中，第 i' 至第 j' 行的最后一个小块，且在矩阵空间上，在矩阵 $\boldsymbol{A}_{i'-j',k'-l'}$ 后，还有属于第 id 号从核的其他小块 $\boldsymbol{A}_{i'-j',k''-l''}$，则在计算完 $\boldsymbol{A}_{i'-j',k'-l'}$ 后，需要通过寄存器通信将矩阵 $\boldsymbol{A}_{i'-j',k'-l'}$ 的计算结果 $b_{i'-j'}$ 发送给从核 id，由从核 id 进行下一步处理，以解决写冲突问题；假设 $\boldsymbol{A}_{i'-j',k'-l'}$ 是原稀疏矩阵 $\boldsymbol{A}$ 中，第 i' 至第 j' 行的最后一个小块，则在计算完 $\boldsymbol{A}_{i'-j',k'-l'}$，同累加了其他从核计算得到的 $b_{i'-j'}$ 后，需要将 $b_{i'-j'}$ 写回内存，并初始化手动缓存.

根据上述初始化流程，需要使用 8 个不同的标志信号用于不同小块的保存：b_{offset} 和 b_{length} 记录小块累加的向量 b 元素的值；x_{offset} 和 x_{length} 记录小块计算所需的向量 x 的元素值；id_{getput} 记录处理完当前小块后，是否有来自于其他从核的 b 元素，如果有，则保存该 b 元素的来源，若没有，则 id_{getput} 就为当前从核的 id；id_{put} 记录处理完当前小块后，是否需要将 b 元素发送至某个从核，如果需要，则保存发送至的从核 id，若将其直接写入内存，则 id_{put} 值为 −1，其他情况下，id_{put} 为当前从核的 id. 为方便使用同一块内存区域来保存所有的非零元，也考虑到每个小块中的非零元分布可能非常的稀疏，故本文选取 COO 格式来保存所有的小块，并用 nnz_{offset} 和 nnz_{length} 来确定每个矩阵小块中非零元的位置和数量.

图 5 以一个 16×16 的稀疏矩阵 $\boldsymbol{A}$ 为例，展示了针对拥有 2×2 从核阵列处理器的固定划分存储结构，并假设其中每个从核上的缓存大小只能存储 2 个向量 $\boldsymbol{b}$ 元素（$LB=2$）和 4 个向量 $\boldsymbol{x}$ 元素（$LX=4$）.

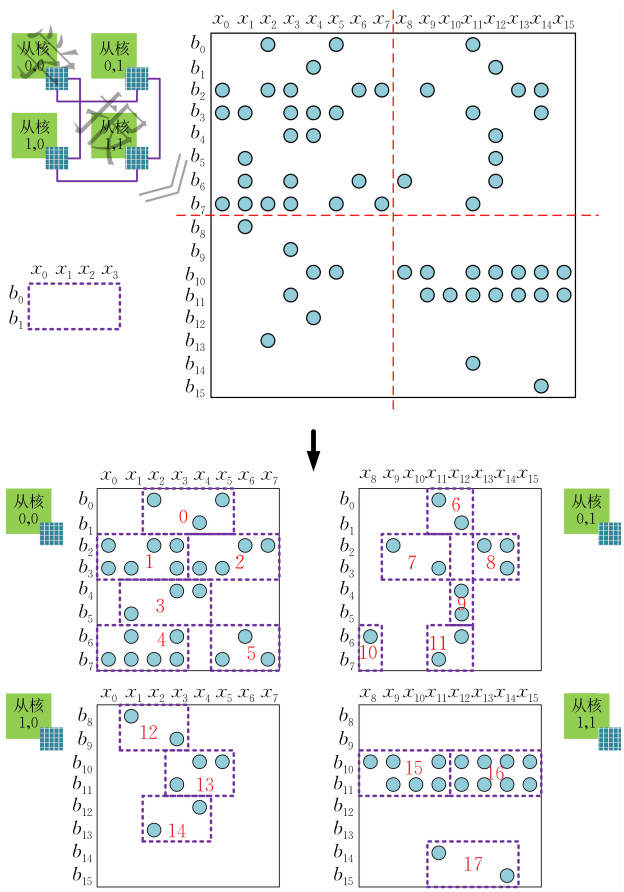


图 5　固定划分方法的预处理阶段

预处理第 1 步：如图 5 中上半部分所示，将原稀疏矩阵 $\boldsymbol{A}$ 根据 2×2 个从核，等分成了 4 个子矩阵并分配给所有从核进行.

预处理第 2 步：如图 5 中下半部分所示. 每个从核根据手动缓存可以容纳数据的多少以及非零元的排布位置，将子矩阵继续分为若干矩阵小块，此处以最大为 2×4 的小块为例. 划分结果如紫色方块所示. 块中数字从小到大的顺序代表每个从核计算小块的顺序. 注意到，其中有些小块的列数大小要小于 4，这是因为用较小的矩阵已经可以覆盖所有的非零元，同时也提高有效带宽，在非零元排布允许的情况下，尽可能少的读入与计算无关的 x 元素，如小块 5 中的 x_5 一定不会被用到，就无需读入，相同情况的还有如小块 8，11，13 等. 但有些情况下读入与计算无关的 x 元素是无法避免的，如小块 7，读入了与计算无关的 x_{10}，这是因为若将这些矩阵拆成 2 个小块，则每个矩阵块中的非零元数量及连续的 $\boldsymbol{x}$ 向量元素将过少，将无法满足 DMA 操作对最少连续地址读写的要求，进而只能采用 gload 的方式从内存中读取非零元，反而会使得整体的读取数据效率降低. 相同情况的还有如小块 3，14，17 等.

预处理第 3 步：将属于同一个小块的非零元储存在一起，每个从核各自负责的小块也存储在一起，不同小块的存储顺序与其计算顺序相同，如图 5 上编号所示.

预处理第 4 步：记录每个小块负责计算的 b 元素的流向. 例如第 0 号小块的 id_{getput} 为 0，而 id_{put} 为 1，因为空间上第 0 小块之前没有其他的小块了，但之后还有 6 号小块，故 0 号从核在计算完 b_0，b_1 的值后需要将它们通过寄存器通信传给 1 号从核进行结果的累加. 对应的，第 6 号小块的 id_{getput} 为 0，而 id_{put} 为 −1，因为第 6 号小块之前有第 0 小块，并且第 6 号小块是原稀疏矩阵 $\boldsymbol{A}$ 的第 0 行至第 1 行中的最后一个小块，故在类加完 b_0，b_1 后需要将结果写回至内存中.

根据上述的预处理过程，以及用于记录每个小块信息的 8 个标志信号，可以得到如算法 3 所示的基于固定划分方法的稀疏矩阵向量乘法.

算法 3. 处理基于固定划分的每个小块的算法流程.

输入：x_{offset}，x_{length}，b_{offset}，b_{length}，nnz_{offset}，nnz_{length}，id_{getput}，id_{put}，row_index，col_index，value，$\boldsymbol{x}$

输出：$\boldsymbol{b}$

```
1. CACHE c_x, c_row, c_row_index, c_col_index, c_value
2. load x[x_offset : (x_offset + x_length − 1)] to c_x
3. load row_index[nnz_offset : (nnz_offset + nnz_length − 1)] to c_row_index
4. load col_index[nnz_offset : (nnz_offset + nnz_length − 1)] to c_col_index
5. load value [nnz_offset : (nnz_offset + nnz_length − 1)] to c_value
6. for i=0 to nnz_length − 1 do
7.  c_b[c_row_index[i]] − b_offset] += c_value[i] × c_x[c_col_index[i] − x_offset]
8. end for
9. if id_getput is not my_id then
10.  for i=0 to b_length − 1 do
11.   c_b[i] += get (id_getput)
12.  end for
13. end if
14. if id_put is not my_id then
15.  if id_put is −1 then
16.   store c_b to b[b_offset : (b_offset + b_length − 1)]
17.   for i=0 to b_length − 1 do c_b[i]=0
18.  else
19.   for i=0 to b_length − 1 do
20.    put (c_b[i], id_put)
21.   end for
22.  end if
23. end if
```

第 1 行：表示在从核上的手动缓存的空间，用于存储 row_index，col_index，$\boldsymbol{x}$，$\boldsymbol{b}$ 和 value.

第 2 行：从 $\boldsymbol{x}$ 向量的 x_{offset} 位置开始，载入 x_{length} 个元素到缓存 c_x 中，即载入与每个小块计算相关的 $\boldsymbol{x}$ 向量中的元素.

第 3～5 行：分别从 row_index，col_index 和 value 的 nnz_{offset} 位置开始，载入 nnz_{length} 个元素至缓存空间 c_row_index，c_col_index 和 c_value 中，即依次载入各个小块中的非零元值.

第 6～8 行：根据缓存中非零元的位置，用相应的 x 元素进行计算后累加至 b 元素中.

第 11 行：获取从核 id_{getput} 中传来的 b 元素，并累加至缓存中的 b 元素上

第 16～17 行：将缓存中的 c_b 中的 b 元素值写回至内存中，并将 c_b 初始化.

第 20 行：将缓存 c_b 中的 b 元素，通过寄存器通信传给从核 id_{put} 中.

可以注意到，算法 3 的第 7 行之前，并没有如算法 2 的第 4～6 行用于判断手动缓存是否命中的代码. 这是由于通过限制每个矩阵小块的大小可以保证所有会被用到的 x 元素和 b 元素一定都在手动缓

存中，以此来节省频繁的缓存判断所需要付出的高昂代价. 同时将所有的非零元按小块存储，也可以做到粗粒度的内存访问来适应申威处理器上的 DMA.

图 6 展示了基于图 5 划分方式的算法执行流程. 其中红色箭头表示通过寄存器通信传递向量 $\boldsymbol{b}$ 元素；蓝色虚线表示处理完不同的小块后在缓存内对向量 $\boldsymbol{b}$ 的累加；紫色箭头表示将向量 $\boldsymbol{b}$ 的值写出至内存. 设处理每个非零元的时间为单位时间，则黑色数字表示程序在不同阶段需要花费的时间. 而最下方的数字代表每个从核处理的非零元的数量，红色表示远超平均值，黄色代表与平均值接近，绿色表示远低于平均值.

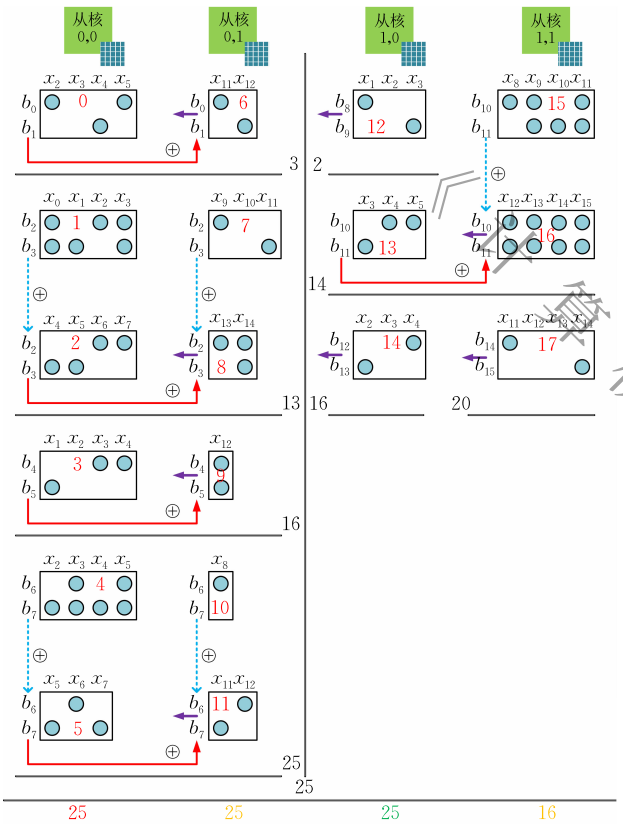


图 6　基于图 5 的计算过程图

以下是对每个从核处理其前 2 个小块的具体流程描述：

在从核(0,0)开始计算第 0 号小块之前，需要从内存中载入第 0 号小块内的 3 个非零元和 x_2 至 x_5. 同时，从核(0,1)载入第 6 小块和 x_{11} 至 x_{12}，从核(1,0)载入第 12 小块和 x_1 至 x_3，从核(1,1)载入第 15 小块和 x_8 至 x_{11}.

一旦小块中非零元和向量 $\boldsymbol{x}$ 元素载入完成，即可开始计算. 以从核(0,0)为例，从核(0,0)需要计算 $b_0=A_{0,2}\times x_2+A_{0,5}\times x_5$ 和 $b_1=A_{1,4}\times x_4$. 由于在计算前，已经将 x_2 至 x_5 载入至缓存中，所以这里无需进行缓存判断语句. 另，注意到从核(0,0)载入至缓存中的 x_3 并没有被使用，不过仍需要载入. 就像上文中分析过的，若在此处改为分别载入 x_2 和 x_4 至 x_5，则会导致 2 个隔断的 DMA 操作，对于 DMA 的带宽反而更加不利，所以这里会选择载入一下冗余的 x 元素，而不是为了避免载入冗余数据而进行多次细粒度的隔断 DMA 操作.

从核(0,0)完成计算后，将 b_0 至 b_1 通过寄存器通信传递给从核(0,1). 对于从核(0,0)来说，如果传递的数据量很小，不足以填满从核(0,0)的寄存器发送缓冲区和从核(0,1)的寄存器行接收缓冲区，那么从核(0,0)会马上开始处理第 1 号小块而不需要等待从核(0,1)接收数据，否则从核(0,0)会阻塞在消息传递环节，等待从核(0,1)接收数据. 从核(0,1)完成对于第 6 号小块的计算之后，将接收来自于从核(0,0)的 b_0 至 b_1 并累加至自己缓存中的 b_0 至 b_1 上，然后将 b_0 至 b_1 写回至内存中.

从核(1,1)完成对于第 15 小块的计算之后，由于第 15 小块之后还有第 16 小块同样需要累加至 b_{10} 至 b_{11}，所以不会等待从其他从核传递过来的 b_{10} 至 b_{11}，也不会将其写回内存，而是直接开始载入第 16 小块和 x_{12} 至 x_{15}，并开始完成对于第 16 小块的计算. 只有当完成对于第 16 小块的计算之后，从核(1,1)才会接收来自与从核(1,0)的 b_{10} 至 b_{11}，并在完成累加之后将 b_{10} 至 b_{11} 写回内存.

在此总结一下固定划分法的特点：

缓存判断：因将原稀疏矩阵 $\boldsymbol{A}$ 分成若干个矩阵小块，每个小块的大小受限，故保证每个小块所需的向量 $\boldsymbol{b}$ 和向量 $\boldsymbol{x}$ 都一定在从核的缓存中，无需进行额外的缓存判断.

细粒度访问：所有的非零元都按照小块为单位存储在一起，使得数据可以批量载入. 而且，非零元的存储数据和计算顺序一致，可以做预取操作. 同样，对于 $\boldsymbol{x}$ 向量，在计算前也会载入连续的 x 元素，以保证 DMA 的效率.

数据局部性：每个小块都是矩阵 $\boldsymbol{A}$ 的一个二维子矩阵，在计算过程中向量 $\boldsymbol{x}$ 和向量 $\boldsymbol{b}$ 中的元素都可以被多次复用.

写冲突：通过寄存器通信，传递向量 $\boldsymbol{b}$ 元素的值给同行中其他从核以解决写冲突问题. 因寄存器通信的带宽极高，延迟又极低，故在申威架构上解决写冲突需花费的代价要比其他平台低廉的多.

可见，固定划分法的 swSpMV 算法已经可以解

决绝大多数算法固有的问题，且算法也可以与申威架构很好的结合.

2.2 一维负载均衡划分方法

虽然固定划分方法可以解决数据局部性差、写冲突、频繁缓存判断和细粒度访问的问题，可却没有考虑到负载均衡的问题，所以算法的并发效率并不高.

如图 6 所示，假设处理每个非零元计算为单位时间，那么原稀疏矩阵 $\boldsymbol{A}$ 中有 60 个非零元，理论上均分给从核的话，只需要 15 个单位时间即可完成. 但如图 6 下方的数字所示，程序总体上会在 25 个单位时间后才完成整个稀疏矩阵向量乘法的计算，比理论值多出了 10 个单位时间. 这正是因为每个从核处理的非零元数量差异过大，从核(0,0)被分配了多达 25 个非零元，远远超出了平均值 15 个. 相对应从核(1,0)只被分配了 7 个非零元，明显过少.

可能更严重的是，在算法中存在非常多的寄存器通信，而每一次寄存器通信都是一次隐式同步. 如从核(1,0)在完成对第 13 号小块的计算之后，需要将 $\boldsymbol{b}$ 向量传递给刚完成对第 16 号小块计算的从核(1,1)，这里就存在一次隐式同步. 可以发现，在消息传递之前，从核(1,0)只计算了 2+3=5 个非零元，而从核(1,1)却计算了 6+8=14 个非零元. 同步之后，从核(1,0)和从核(1,1)都需要从第 14 个单位时间开始继续计算. 故在完整的计算过程中，就算总体上看每个从核处理的非零元数量相当，若存在频繁的局部非零元不一致的情况，总时间也会远超出平均非零元数量所对应的计算时间.

至此，本文以行并行划分方法为基础，设计了一维负载均衡的方法. 与固定划分法相比，一维负载均衡只升级了预处理部分，即用于存储稀疏矩阵 $\boldsymbol{A}$ 的存储结构. 一维负载均衡方法的存储结构包括了 4 个层次：原矩阵-矩阵带-子矩阵-小块. 由于小块的保存方式和固定划分法是一致的，所以计算部分只要延用算法 3 即可. 根据缓存可以存储的向量 $\boldsymbol{b}$ 元素上限 LB，一维负载均衡法将矩阵 $\boldsymbol{A}$ 分割成宽度为 LB 的若干个矩阵带，并将矩阵带交替地分给不同行的从核负责，以保证不同行从核间的负载均衡. 此外，进一步均分矩阵带内非零元数量至不同的子矩阵，分给同行不同的从核负责，以此保证同行内从核的负载均衡.

图 7 展示了一维负载均衡划分法的预处理过程，具体如下：

预处理第 1 步：如图所示矩阵高度为 16，$LB=2$，则分为 8 个矩阵带，每个矩阵带的非零元数量为 5,15,5,12,2,17,2 和 2 个.

预处理第 2 步：将矩阵带交替分给不同行的从核负责.

预处理第 3 步：假设每行包含了 COLS 个从核，则依次将矩阵带分为 COLS 个子矩阵，并尽可能使各个划分出的子矩阵包含的非零元数量接近. 子矩阵的划分如图 7 中红线所示，因图中每行包含 2 个从核，则每个矩阵带被分成 2 个子矩阵，红线左侧的红矩阵归该行从核中的 0 号从核计算，右侧归 1 号从核计算. 注意到，这里矩阵中的红色数字的大小顺序，依然是每个从核对这些矩阵小块的计算顺序，如(0,0)从核负责计算的小块依次为 0,1,2,3，(0,1)从核负责计算的小块依次为 4,5,6,7.

预处理第 4 步：与固定划分法中预处理的第 2～4 步一致，根据从核缓存大小，将子矩阵划分成若干小块，存储在一起，并记录各个小块的 id_{put} 与 id_{getput}.

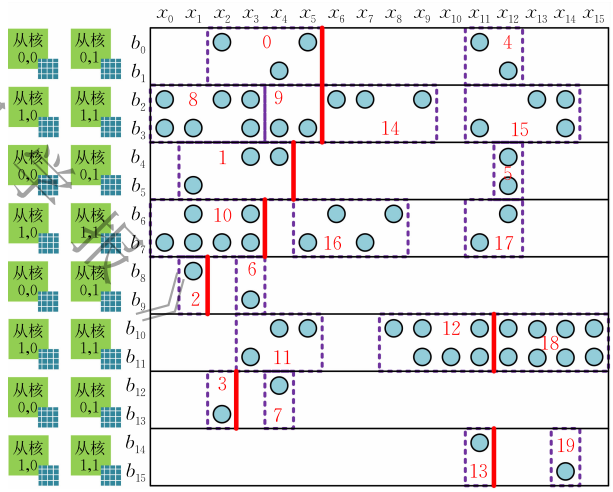


图 7 一维负载均衡划分方法的预处理阶段

一维负载均衡解决了同行上不同从核的负载均衡问题，这就使得每次由寄存器通信所导致的隐式同步不会由于某个从核的计算量过大，而拖慢同行上所有从核的执行效率. 另外，交替地把矩阵带交给不同的从核完成，也可以大概率保证不同行的从核负载均衡.

2.3 二维负载均衡划分方法

虽然一维负载均衡方法可以在大概率使得各从核负载均衡，不过在交替分配矩阵带前，并没有对矩阵中的非零元排布进行分析. 而基于静态分析的任务分配总会遇到不同行从核间负载不均衡的情况. 如图 7 中，第 0 行从核处理的非零元数量为 5+5+

2＋2＝14 个，第 1 行从核处理的非零元数量为 15＋12＋17＋2＝46 个，第 1 行的从核明显处理了更多的非零元.

另外，对于最下方的矩阵带，虽然只有 2 个非零元，但依旧被分成了 2 个子矩阵，交由 2 个从核处理，还需要引入通信进行右端向量加和处理，这是得不偿失的. 故本文还需要解决由一维负载均衡方法带来的细粒度同步问题.

针对以上的两个问题，本节在一维负载均衡的基础上，针对同一矩阵带中的负载均衡，设计了二维负载均衡划分方法. 该方法也只对存储方式进行了优化，即只升级了预处理部分，而计算部分，依旧直接延用算法 3 即可. 二维负载均衡划分法的存储方式和一维的保持一致，依然包含了原矩阵-矩阵带-子矩阵-小块这 4 部分. 而与一维划分不同的是，不再交替地将矩阵带分配给不同行的从核，而是使用贪心的分配方式，尽可能保证不同行之间的负载均衡. 另一方面，也会将非零元数量较少的矩阵带，直接交给一个从核去完成，而不再均分给同行上的几个从核.

图 8 展示了二维负载均衡划分法的预处理过程：

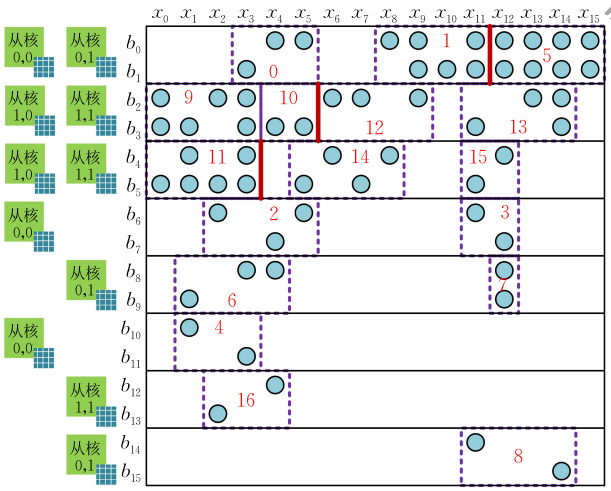


图 8　二维负载均衡划分方法的预处理阶段

预处理第 1 步：和一维划分时一致，根据 LB 的大小，将稀疏矩阵 $\boldsymbol{A}$ 划分成宽度为 LB 的若干矩阵条.

预处理第 2 步：统计每个矩阵带中包含的非零元数量，并据此进行从大到小的排序. 如图所示，矩阵带 $\boldsymbol{A}_{10-11,0-15}$ 包含最多的 17 个非零元，故被排在了第一的位置. 需要注意的是，此处的重排并不是物理上的对矩阵 $\boldsymbol{A}$ 中的非零元和向量 $\boldsymbol{b}$ 进行重排，而只是逻辑上的顺序调整，即只是对计算顺序的调整，额外的开销只是排序所需的时间. 因不同矩阵带的计算是相互独立的，故此调整不会对计算的结果产生影响.

预处理第 3 步：用贪心的方法，将不同的矩阵带交给不同的从核处理，如算法 4 所示. 算法输出的 set 表示不同行的从核需要负责的矩阵带的编号. 算法 4 中第 3 行，Select 函数会在所有 ROWS 行的从核中，选择出目前负责非零元数量最少的一行，并将当前的矩阵带 i 分配给它.

算法 4.　用贪心算法分配矩阵带给不同的从核处理.

输入：$tiles$，nnz_{tile}，ROWS

输出：set

1. $nnz_{set} \leftarrow 0$
2. for $i=0$ to tiles -1 do
3. 　Select id if $nnz_{set}[id]$ is minimal
4. 　$nnz_{set}[id] += nnz_{tiles}[i]$
5. 　$set[id] = set[id] \cup \{i\}$
6. end for

预处理第 4 步：将非零元较多的矩阵带，划分成若干个子矩阵. 为了保证算法有较好的并发度，先定义每个从核需要最少处理的子矩阵数量为 $submatrix_{\min}$，再根据每行从核负责的非零元数量 nnz，可以算得每个子矩阵包含的非零元上限 $nnz_{\max} = \lceil nnz/submatrix_{\min} \rceil$，且如果某矩阵带的非零元数量 nnz_{tile} 超过 $nnz_{\max}$，就认为该矩阵带包含的非零元较多，需要被进一步划分. 进而，将该矩阵带切割成 MAX(COLS, $2^{\left\lceil \log_2 \left(\frac{nnz_{\text{tile}}}{nnz_{\max}} \right) \right\rceil}$) 个子矩阵，注意到 $2^{\left\lceil \log_2 \left(\frac{nnz_{\text{tile}}}{nnz_{\max}} \right) \right\rceil}$ 是 2 的幂次，以保证每个矩阵带只能被切成 1，2，4，8 个子矩阵，保证可以较为灵活的给从核进行任务分配. 以图 8 为例，只有前 3 个矩阵带进行了进一步分割.

预处理第 5 步：将切割好的子矩阵分给相应的从核处理. 因为矩阵带已排序，且子矩阵数量都为 1，2，4，8 个，故只要依序分配，就可做到每个从核所负责的非零元数量几乎相等.

预处理第 6 步：和固定划分法的第 2～4 步一致，将各子矩阵进行分块、存储并记录 id_{put} 与 id_{get}.

注意到，在预处理过程中基于贪心的任务划分，都只是对不同行从核间进行的任务划分，以尽量做到负载均衡. 而对于每个从核而言，其处理对象依然是子矩阵(对矩阵带进一步划分而成)，最小的计算对象依然是矩阵小块，故不会影响到数据局部性差等之前已经解决的问题.

二维划分方法的计算过程和固定划分法的一致，故计算过程只需延用算法 3 即可. 图 9 展示了基于图 8 划分方法后的执行流程. 其中的红色箭头表

示通过寄存器通信传递向量 $\boldsymbol{b}$，紫色箭头表示将向量 $\boldsymbol{b}$ 写出至内存，黑色表示在不同阶段所需要花费的时间，最下方的数字表示每个从核处理的非零元数量，可见，是非常均衡的. 另一方面，程序只有在处理排在前面的、非零元数量较多的矩阵带时，才会需要使用寄存器通信来解决写冲突. 随着程序的运行，越到后期，每个矩阵带包含的非零元数量会越来越少，被分成的子矩阵也会越来越少，即参加该矩阵带计算的从核也会越来越少，也就越不会因为寄存器通信带来的隐式同步承受额外的代价.

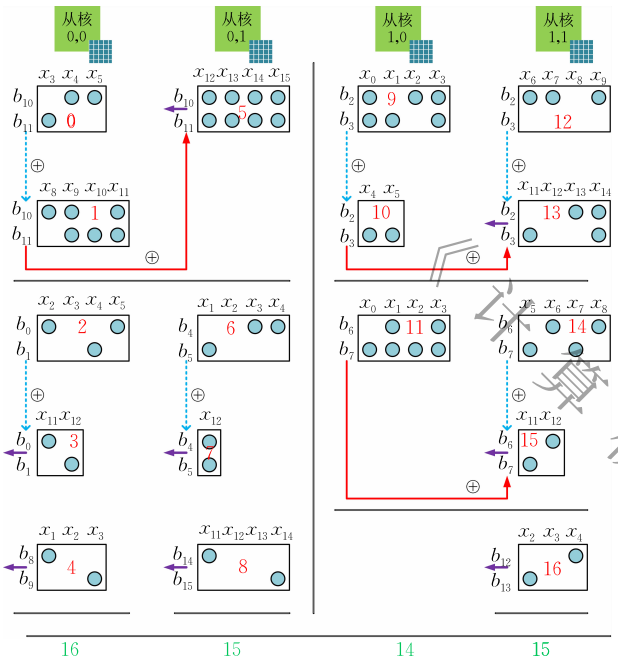


图 9 基于图 8 的计算过程图

在此总结一下二维负载均衡划分法的优势：该算法在保持了固定划分解决的稀疏矩阵向量乘法固有的数据局部性差和写冲突的问题，以及与申威架构结合后带来的缓存判断问题和细粒度访问的问题，同时做到了不同行之间和同行之内从核之间的负载均衡，最小化了同步开销，从而真正意义上做到了所有从核的负载均衡.

4 swSpMV 算法性能测试

4.1 实验设计与测试环境

为了覆盖尽可能多的矩阵特性，本节使用了来自于 SuiteSparse 的稀疏矩阵集合中的 2710 个矩阵作为测试样例，矩阵集中包含了方阵和非方阵，而非零元数量也从 1 个到 1.1 亿个不等，其中非零元大于 1000 个的矩阵占了 90%.

测试平台使用的是申威 26010 处理器的单个核组，其理论访存带宽是 34 GB/s. 本节将测试 4 种方法，包括主核上的串行 SpMV 算法的标准实现，基于固定划分的 SpMV 算法实现、基于一维负载均衡划分法的 SpMV 算法实现和基于二维均衡划分法的 SpMV 算法实现，3 种算法都使用 C 语言和 Athread 线程库实现. 所有的测试都采用双精度浮点计算，量测的计算时间中不包含预处理的时间.

在实现中，3 个参数分别设置为 $LB=2048$，$LX=2048$，$submatrix_{\min}=4$. LB，LX 的大小是根据计算精度和 SPM 大小决定的，在 SPM 存储空间的划分上，我们将 64 KB 分 16 KB 存储右端向量 $\boldsymbol{x}$，分 16 KB 存储结果向量 $\boldsymbol{Y}$ 以及 32 KB 存储矩阵中的非零元素.

4.2 算法性能测试结果及分析

图 10 展示了在申威处理器上不同算法计算处理不同矩阵算例时的性能，评价标准是单位时间内浮点数的运算次数(GFlops)，数值越高，性能越好.

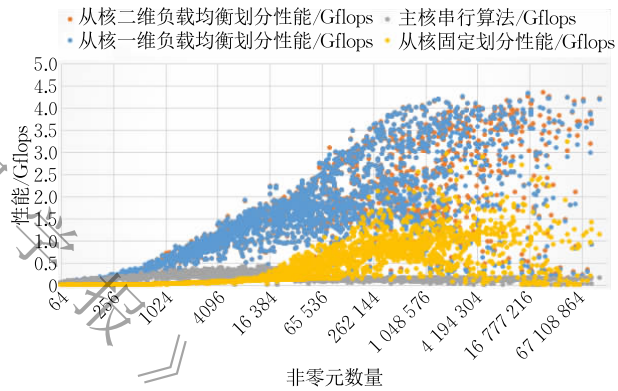


图 10 申威 26010 上 4 种算法性能(GFlops)

在图 10 中可见，在主核上串行算法的性能(灰色点)在非零元数量不断增加的情况下，出现了明显下降. 这是因为在小规模矩阵下，大部分的向量 $\boldsymbol{x}$ 和向量 $\boldsymbol{b}$ 元素都能保留在主核的自动缓存中. 但矩阵规模不断增加，向量 $\boldsymbol{x}$ 和向量 $\boldsymbol{b}$ 的元素无法再全部存储在主核的缓存中，导致性能下降. 受制于单个主核的低带宽，其串行算法的最高性能仅为 0.428 G-Flops.

而采用了固定划分法(黄色点)后，性能明显提升. 与主核串行算法相比，性能平均提高了 3.75 倍，而最高加速为 70.5 倍，出现在算例“fron_g500_logn20”下(该算例矩阵为行列均为 1M 的方阵，非零元个数略低于 90M). 而 70.5 倍已经超过了从核的数量(64 个)，这是因为与主核串行算法相比，固定划分法的存储相当于将所有从核的高速缓存“连接”在了一起形成了一个大缓存，更大的快速缓存，更好的数据复用性，从而提高了性能. 但固定划分法

很难做到真正的负载均衡，故绝对性能并不算高，其中最高性能为 3.23 GFlops 出现在算例“spal_004”下（该算例矩阵为10K 行，322K 列，有 46M 的非零元）.

采用一维划分方法（蓝色点）和二维划分方法（橙色点）后，使得各从核间负载均衡，算法性能会得到显著提升. 前者与串行算法相比，可以获得平均 11.4 倍的加速，最高加速 54.8 倍，最高性能为 4.33 GFlops；后者与串行算法相比平均加速 11.7 倍，最高加速为 55 倍，最高性能为 4.35 GFlops.

图 11 更直观的展示了包括串行算法在内 4 种不同算法在所有测试集上的平均性能和最高性能情况. 与固定划分法相比，一维划分和二维划分，分别能带来平均 11.4 和 11.7 倍的性能提升.

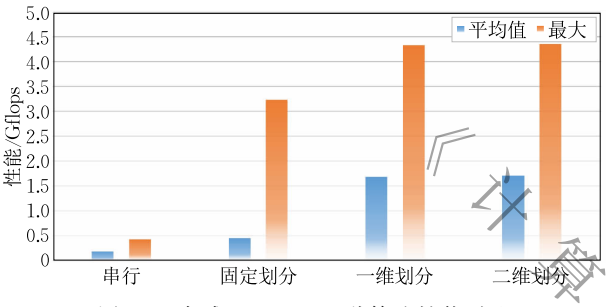


图 11　申威 26010 上 4 种算法性能对比

具体对比一维划分法和二维划分法可以得出，在大多数情况下，两者无论在最高性能，还是在平均加速比和最高加速比下，差异不大. 这是因为在测试矩阵集中，绝大多数的矩阵都可以通过一维划分法的交替分配方案达到负载均衡. 但也有一维负载划分法无法达到负载均衡的算例，如算例“sls”（该算例矩阵为 1.7M 行，62K 列，有 6.8M 非零元），只有通过二维划分法才能达到更好的负载均衡，而在该算例下，二维划分法在性能上可以比一维划分法高 1.76 倍，这也是两种算法间的最高加速效果比.

不过在图 10 中也可以看到，对于一些非零元非常多的算例，二维负载均衡划分法的性能表现并不佳. 如算例“NLR”（该算例矩阵是行列都为 4.2M 的方阵，有 25M 非零元），实际性能仅为 0.1 GFlops. 类似的问题主要是因为算法的带宽效率较低. 图 12 展示了 3 种划分方法 SpMV 实现的实际带宽，随着非零元的增加，算法可以发挥的实际带宽迅速增加，以算例“NLR”为例，在二维划分法上带宽高达 24 GB/s；最高带宽甚至可以达到 28 GB/s，出现在“web-Google”算例. 如此高的性能，已经高于了 Stream Triad 的测试结果. 这是由于 Stream Traid 对于内存的读写比例是 2∶1，频繁的读写交替会限制申威处理器上 DDR3 内存带宽的发挥；而 SpMV 的数据读入量远远大于数据写出量，所以更有益于 DDR3 内存带宽的发挥. 但高的实际带宽并不意味着高的计算性能.

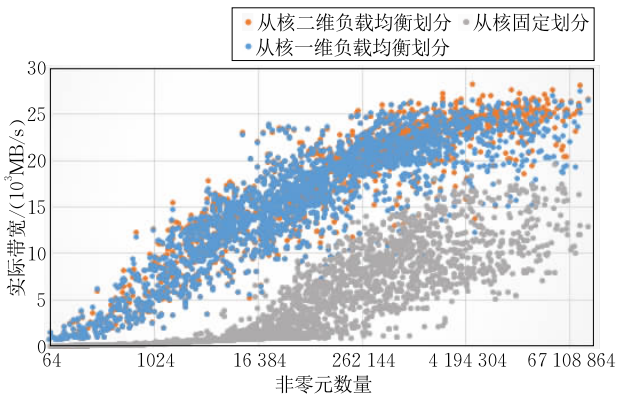


图 12　3 种算法的实际带宽对比

图 13 展示了从核上两种划分方法 SpMV 实现的带宽效率，即带宽效率＝有效带宽/实际带宽＝有效访问量/实际访问量. 此处对有效访问量的定义为计算一次稀疏矩阵向量乘法所需的最少访问量，即稀疏矩阵 $\boldsymbol{A}$ 的大小、向量 $\boldsymbol{x}$ 的大小和向量 $\boldsymbol{b}$ 的大小之和. 从图 13 中可以看出，在二维负载均衡划分下，大多数算例的带宽效率已经接近 100%，即已经最大限度地利用了带宽. 而固定划分看起来带宽效率也很高，这是因为其真实带宽过低而造成的. 从图中也可以发现，不同算法的带宽效率并没有什么特定的规律，这是因为访问效率只和稀疏矩阵的特性有关，不同矩阵间的差异会非常大，而任何算法的局部性策略也只能尽可能提高带宽效率，但都不能做到完美无缺. 以算例“NLR”为例，在二维划分算法下，有效带宽仅为 0.85 GB/s，带宽效率仅有 3.6%.

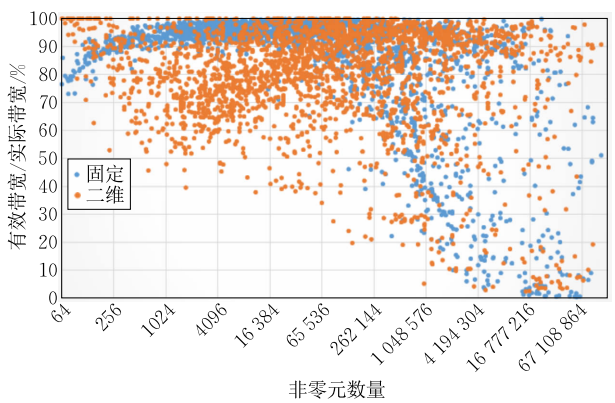


图 13　两种算法的有效带宽/实际带宽

如此低的带宽效率主要有两个原因：一方面是由于非零元在原矩阵中的分布较为平均，导致每个从核在处理矩阵带时都需要读入大量的 x 元素，另

一方面是由于对于 x 的需求跨度很大，这样使得为了提高内存访问效率而选择的连续访问会读入大量冗余的 x 元素，进一步降低了有效带宽. 以上两点解释了为何即使在并发度非常高的算例上，算法的性能依旧不甚理想.

最后，将基于二维划分法的 swSpMV 实现，与 GPU 上的 CSR5 格式的 SpMV 实现进行性能对比. 测试用的 GPU 为 Nvidia GPU Kepler K40m. 因申威 26010 的内存带宽上限为 22.6 GB/s，远低于 GPU 内存带宽，故计算性能无法直接进行比较. 图 14 中，将各自算法的有效带宽除以内存带宽上限进行比较，此处的有效带宽也是只计算初始矩阵按 CSR 格式的一次读入，$\boldsymbol{x}$ 向量的一次读入与 $\boldsymbol{b}$ 向量的一次输出. 这样可以认为算法是在基本相似的带宽条件下进行性能的比较. 因 SpMV 是典型的带宽密集型问题，故有限带宽除以带宽上限得到的数字大小，可以侧面体现出算法的性能. 我们的算法平均能达到 53% 的带宽利用率，而 GPU 上的 CSR5 只能有 29% 的利用率. 从图 14 中也可以看出，总体而言，我们算法能发挥的带宽能力更高，更有效. 但在某些非零元较多的点上，如图上右下角部分，两种算法的带宽利用率都很低，这是因为这些矩阵都是极其稀疏的，平均每行非零元只有十几个甚至几个. 而低稀疏度对申威架构更不友好，这也是在这些算例上，我们的算法带宽利用率不佳的原因. 这也正如之前分析到的，不同矩阵稀释特性的不同，对申威手动缓存架构的挑战是巨大的，也无法通过一种固定的算法，使其对任何矩阵都能做到尽善尽美.

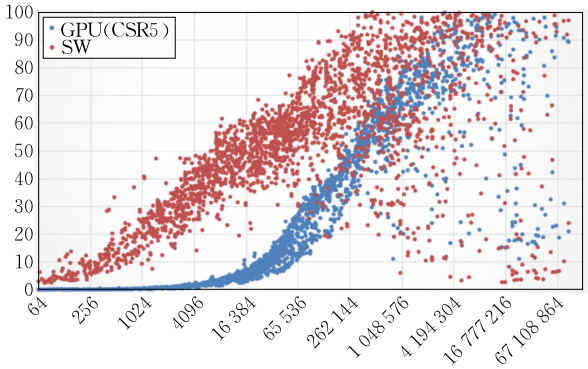


图 14 申威 26010 上二维负载均衡划分 SpMV 实现与 GPU 上 CSR5 SpMV 性能对比

5 总 结

稀疏矩阵向量乘法无论在数值计算中，还是在大数据分析领域，都是非常重要的计算核心. 本文设计并提出了二维负载均衡划分法，通过按照申威 26010 高速内存大小，将矩阵以行为基础分成矩阵带，再以分块的方式计算每个矩阵带内的非零元，从而解决稀疏矩阵向量乘法固有的数据局部性差、写冲突和负载不均衡的问题，且充分利用了粗粒度访问并降低了手动缓存判断次数. 使用来自 SuiteSparse 稀疏矩阵集合中全部 2710 个算例，在申威 26010 处理器单个核组上对算法进行测试. 本文最终提出的二维负载均衡划分法可以最高获得 4.35 GFlops 的性能，以及与主核上串行算法相比平均 11.7 倍加速和最高 55.0 倍加速.

参 考 文 献

[1] Davis T A, Hu Y. The university of florida sparse matrix collection. ACM Transactions on Mathematical Software, 2011, 38(1): 1-25

[2] Xu Z, Lin J, Matsuoka S. Benchmarking sw26010 many-core processor//Proceedings of the Parallel and Distributed Processing Symposium Workshops (IPDPSW). Orlando/Buena Vista, USA, 20172017: 743-752

[3] Rice J R. Ellpack: A research tool for elliptic partial differential equations software. Mathe-Matical Software. Amsterdam: Elsevier, 1977: 319-341

[4] Kourtis K, Karakasis V, Goumas G, et al. Csx: An extended compression format for spmv on shared memory systems. ACM SIGPLAN Notices, 2011, 46(2): 247-256

[5] Ashari A, Sedaghati N, Eisenlohr J, et al. An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on gpus//Proceedings of the 28th ACM International Conference on Supercomputing. Muenchen, Germany, 2014: 273-282

[6] Bolz J, Farmer I, Grinspun E, et al. Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. ACM Transactions on Graphics (TOG), 2003, 22(3): 917-924

[7] Bell N, Garland M. Implementing sparse matrix-vector multiplication on throughput-oriented processors//Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. Portland, USA, 2009: 18

[8] Monakov A, Lokhmotov A, Avetisyan A. Automatically tuning sparse matrix-vector mul-tiplication for gpu architectures. International Conference on High-Performance Embedded Architectures and Compilers. Pisa, Italy, 2010: 111-125

[9] Yang Wang-Dong, Li Ken-Li, Shi Lin. Quasi-diagonal matrix hybrid compression algorithm and implementation for SpMW on GPU. Computer Science, 2014, 41(7): 290-296 (in Chinese)

（阳王东，李肯立，石林. 一种准对角矩阵的混合压缩算法及其与向量相乘在GPU上的实现. 计算机科学，2014，41(7)：290-296）

[10] Koza Z，Matyka M，Szkoda S，et al. Compressed multirow storage format for sparse matrices on graphics processing units. SIAM Journal on Scientific Computing，2014，36(2)：C219-C239

[11] Sun X，Zhang Y，Wang T，et al. Optimizing spmv for diagonal sparse matrices on gpu//Proceedings of the 2011 International Conference on Parallel Processing（ICPP）. Taipei，China，2011：492-501

[12] Greathouse J L，Daga M. Efficient sparse matrix-vector multiplication on gpus using the csr storage format//Proceedings of the International Conference for High Performance Computing，Networking，Storage and Analysis. New Orleans，USA，2014：769-780

[13] Ashari A，Sedaghati N，Eisenlohr J，et al. Fast sparse matrix-vector multiplication on gpusfor graph applications//Proceedings of the International Conference for High Performance Computing，Networking，Storage and Analysis. New Orleans，USA，2014：781-792

[14] Merrill D，Garland M. Merge-based parallel sparse matrix-vector multiplication//Proceedings of the International Conference for High Performance Computing，Networking，Storage and Analysis. Salt Lake City，UT，USA，2016：58

[15] Liu W，Vinter B. Csr5：An efficient storage format for cross-platform sparse matrix-vector multiplication//Proceedings of the 29th ACM International Conference on Supercomputing. Newport Beach/Irvine，USA，2015：339-350

[16] Buluc A，Fineman J T，Frigo M，et al. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks//Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures. Calgary，Alberta，Canada，2009：233-244

[17] Choi J W，Singh A，Vuduc R W. Model-driven autotuning of sparse matrix-vector multiply on gpus. ACM Sigplan Notices，2010，45：115-126

[18] Liang Y，Tang W T，Zhao R，et al. Scale-free sparse matrix-vector multiplication on many-core architectures. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems，2017，36(12)：2106-2119

[19] Liu X，Smelyanskiy M，Chow E，Dubey P. Efficient sparse matrix-vector multiplication on x86-based many-core processors//Proceedings of the International ACM Conference on Supercomputing. Eugene，OR，USA，2013：273-282

[20] Im E，Yelick K，Vuduc R. SPARSITY：Optimization framework for sparse matrix kernels. International Journal High Perference Computer Applied，2014，18：135-158

[21] Mellor-Crummey J，Garvin J. Optimizing sparse matrix-vector product computations using unroll and jam. International Journal of High Perference Computer Appiled，18：225-236，200

[22] Sun Q，Zhang C-Y. Bandwidth reduced parallel SpMV on the SW26010 many-core platform//Proceedings of the 47th International Conference on Parallel Processing. Eugene，USA，2018：54

[23] Liu C X，Xie Biwei，Liu X，et al. Towards efficient SpMV on sunway many-core architectures//Proceedings of the 2018 International Conference on Supercomputing. Portland，USA，2018：363-373

LI Yi-Yuan，Ph. D. candidate. His research interests include high performance computing.

XUE Wei，Ph. D.，associate professor. His research interests include scientific computing and uncertainty quantification.

CHEN De-Xun，Ph. D. candidate，senior engineer. His research interests include computer architecture.

WANG Xin-Liang，Ph. D. His research interests include high performance computing，heterogeneous computing，parallel computing.

XU Ping，M. S. His research interests include high performance computing.

ZHANG Wu-Sheng，Ph. D.，senior engineer. His research interests include cluster computing，parallel computing，distributed computing.

YANG Guang-Wen，Ph. D.，professor. His research interests include computer architecture.

Background

Sparse matrix vector multiplication（abbreviated as SpMV）is widely used in iterative methods for solving large-scale linear equations，which calculates the form of equation $\boldsymbol{Ax}=\boldsymbol{b}$，where $\boldsymbol{A}$ on the left side of the equation represents the sparse matrix，$\boldsymbol{x}$ is the known vector，and $\boldsymbol{b}$ on the right side represents the result vector obtained after calculation. The top left part of Fig. 2 shows a simple example of SpMV. Because the number of non-zero elements in the sparse matrix

is very rare, specific storage formats need to be designed for skipping out zero elements. Selecting and using an appropriate storage structure to store the non-zero elements in sparse matrix will directly affect the performance of SpMV as well as other computing kernels of sparse matrix.

Currently, there are four basic matrix storage formats, named Coordinate list (COO), Compressed Sparse Column (CSC), Compressed Sparse Row (CSR) and ELLPACK[1], respectively shown in Figs. 2 and 3. The COO format uses triples to store each non-zero element; Both CSR and CSC compress the row or column coordinates on the basis of COO. The ELLPACK format is similar to the way that dense matrices are stored, except that the number of columns of the matrix is compressed.

Algorithm 1 is a sequential SpMV algorithm using CSR format, which is also the core calculation process in many related works. In Algorithm1, only line 4 involves computation, which introduces three inherent issues for implementing high performance SpMV:

Poor data-locality: No matter which format is used to store the sparse matrix, at least one of the access to vector $\boldsymbol{b}$ or vector $\boldsymbol{x}$ is random and discontinuous, so the data locality is poor.

Write-conflict: In parallel implementation, if the calculations of an element in vector $\boldsymbol{b}$ are assigned to different threads, it will introduce write conflicts when multiple threads write the same element.

Load-imbalance: The number of non-zero elements contained in each row may be quite different. When the sequential algorithm is refactored to the parallel one, the task is not easy to be divided into balanced workloads. And this problem will become more and more serious with the increase of the number of cores and threads.

In recent years, there have been many research works trying to resolve the issues mentioned above over different kinds of architectures. Especially, for Chinese home-grown many-core processor Shenwei 26010, Refs. [19-20] presented the specific tiling or blocking schemes to improve the reuse of vector $\boldsymbol{x}$ as well as improve performance of SpMV. However, these blocking strategies are more aimed at exploring concurrency, improving locality and increasing the granularity of data access, but lack of comprehensive consideration for the issue of load-imbalance.

Our contribution is to implement an efficient SpMV algorithm on Shenwei26010. By hierarchically dividing the original sparse matrix into matrix blocks and well reorganizing the blocks in memory with calculation order, the proposed algorithm can well resolve the inherent issues of SpMV at the same time and also can easily present high performance implementation on Shenwei26010. By evaluation with all the 2710 benchmarks from SuiteSparse Collection, the proposed method can achieve an average speedup of 11.7 and the best speedup of 55.0, compared with the sequential method on the management processing element of Shenwei 26010.