基于CUDA的稀疏矩阵-向量乘法的实现

计算机学院

吴振伟（14060050）

摘 要

稀疏矩阵-向量乘法（SpMV）是科学计算领域的一个基础问题。 因此，高效实现SpVM，对于提升科学计算问题求解性能至关重要。以GPU作为协处理单元提升计算机系统 并行数据处理能力的技术，今年来收到广泛关注。

本文对使用GPU加速SpVM求解的技术进行了系统学习和总结，并分别基于CPU和GPU对COO、CSR、ELL三种格式的计算性能进行了对别分析实验，得出ELL格式更适用于GPU计算的初步结论。

关键词: SpMV, GPU, CUDA

1 引言

众多科学计算问题，如线性方程组求解、偏微分方程求解等，均以稀疏矩阵-向量乘法（SpMV)为基础。高效实现SpMV，长期以来备受关注。

受限于工艺水平、材料物理特性、功耗等因素，单处理机性能提升达到极限。Intel, AMD等主流处理器厂商，均采用多核并行处理的技术，来提高处理器的整体运算性能。相应的，通过并行数据处理对大规模科学运算问题的优化，成为主流技术。

近年来，以通用图形处理单元(GPGPU)作为协处理部件，协同中央处理器(CPU)加速计算机系统对计算密集型程序的执行速度，引起了人们的广泛关注。 在市场需求的驱使下，传统的可编程图形处理单元（GPU）逐步发展成为具有强大运算能力和高度并行性的处理单元。GPGPU是指对图形处理器强大的并行处理能力和可编程流水线加以利用，协助中央处理器对非图形数据进行处理，即利用GPU对非图形相关的通用计算任务进行加速。

与CPU有所不同，GPU所采用的设计专门面向于具有高度并行性的数据密集型运算。GPU支持的硬件线程数，要远远大于多核CPU支持的硬件线程数。同一段程序在多组数据上并行执行，大大降低了运算过程的控制复杂性，并能够通过数据运算来隐藏访存延迟。这使得在GPU设计中，更多的硬件资源被专用于进行数据处理，而非用于数据缓存和程序流控制；使得GPU拥有相比CPU更加大的浮点运算能力和更高的存储器带宽。

NVIDIA公司于2007年提出了统一计算架构(Compute Unified Device Architecture, CUDA) [1]，该架构提供了一个清晰的GPU编程模型。 开发者可以基于CUDA，更加高效、充分地利用GPU的运算能力，对通用计算任务进行加速 [2]。

本文接下来将分别对GPU和CUDA的架构进行简要介绍； 并对基于CUDA编程模型，实现稀疏矩阵-向量乘法的流程进行介绍。

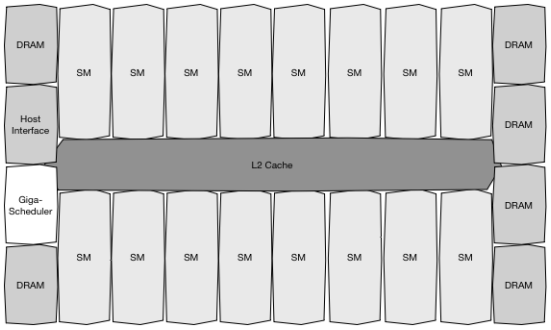


图 1: Fermi微体系结构示意图

2 GPU硬件逻辑结构

常见的多核心CPU的核数从几个到几十个不等，所支持的并发硬件线程数，在几个到几百个这样一个数量级。而常见的GPU器件，核心数达到几千个，并发硬件线程数远远超过CPU。GPU采用了一些特殊的设计和技术，来调度和管理如此大规模的并发线程。为了加深对GPU硬件结构的认识，这里不失一般性，以Nvidia公司开发的Fermi GPU微体系结构，如图 1 ，对GPU的硬件逻辑结构进行简要的介绍 [3]。

GPU构建在一个多线程流式多核处理器（Streaming Multi-processors, SM）阵列之上。如图 2所示，每个SM又由32个处理器核心组成，即最大可以支持32个线程同时运行。在运行时，由GigaSchedular模块负责对任务进行粗粒度的调度——面向SM进行线程调度；SM中的WrapSchedular则负责更细粒度的调度——面向SM中的处理器核心进行线程调度。在下一小节中，本文将结合CUDA编程模型，分别对这两个粗粒度、和细粒度的调度过程，进行更加具体的描述。

3 CUDA编程模型

CUDA将线程抽象分组抽象为3个级别：由32个thread组成一个wrap;若干个wrap组成一个block;若干个block进而组成一个grid.

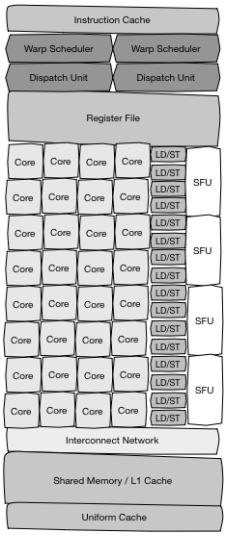


图 2: 多线程流处理器结构示意

CUDA采用这样一种多层级、嵌套式的方式组织线程，获得了一种对软件开发过程透明的可扩展性。GPU以block为单位，对线程进行粗粒度调度：任意一个block可以被调度到任意一个可用的GPU内部的SM上运行。在SM内部，WrapSchedular又针对被调度的block中的若干个wrap，进行细粒度的调度。 在软件开发过程，编程人员通过上述线程组织结构对计算任务拆分进行描述。这样一来，在CUDA框架下编写的程序，可以动态适应拥有任意处理器核心数的GPU。

GPU内部的SM，被设计成为SIMT（Single Instruction Multi-Thread）结构。在SIMT结构下，处理器以wrap为最小单位对线程进行调度和管理。当一个wrap被调度时，它所包含的32个thread从同一条指令开始执行；而每个thread拥有其独立的程序计数器和寄存器，所以同一个wrap中的多个thread可以自由进行分支跳转，相互独立地运行。在无跳转发生时，同一wrap中的所有thread共享执行路径；在有跳转发生时，处理器先后串行执行处于不同执行路径上的thread，直至

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

$$data = \begin{bmatrix} * & 1 & 7 \\ * & 2 & 8 \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \quad offsets = \begin{bmatrix} -2 & 0 & 1 \end{bmatrix}$$

图 3: 对角矩阵A的DIA表示

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

$$data = \begin{bmatrix} 1 & 7 & * \\ 2 & 8 & * \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \quad indices = \begin{bmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 1 & 3 & * \end{bmatrix}$$

图 4: 对角矩阵A的ELL表示

所有分支路径最后又收敛到同一执行路径。忽略SIMT的这些处理细节，是可以保证程序的正确性的。而基于SIMT结构的这些特点，在编写thread代码时，有意识地规避掉执行路径出现分歧的情况，是提高程序性能的一个很好的切入点。

4 几种典型的稀疏矩阵存储格式

针对稀疏矩阵中多数元素为0的这一特点，往往采用压缩存储的方式，实现更高效的存储。本节后续部分，将对几种常见的压缩方式如CSR、ELL等进行简要介绍[4]。

4.1 几种典型的稀疏矩阵存储格式

4.1.1 DIA

针对对角矩阵、分块对角矩阵等非零元素主要集中在主对角线附近区域的稀疏矩阵，可以使用DIA格式实现高效存储。 如图 3所示：

DIA格式由data和offsets两部分组成。data矩阵中的一列，对应矩阵A中的一条非全零对角线，向量offsets一次存放了相应列编号相对主对角列编号的偏移量。 由于超对角线和次对角线上的元素少于主对角线元素，data矩阵中的相应列会出现空缺，应对的方案是分别在次对角线列和超对角线列的头部和尾部填充任意值。 这样一种处理方式，使得A中非零元的位置信息在data和offsets中得到充分描述——非零元在A矩阵中的行索引，等于其在data矩阵中的行索引；非零元在A矩阵的列索引，等于其行索引加上相应的偏移量。其总体效果是，实现了对数据的高效存储。

4.1.2 ELLPACK

ELLPACK(ELL)是一种更通用的存储方式，可应用于具有任何非零元分布形式的稀疏矩阵。如图 4所示：

ELL格式由data和indices两个矩阵组成。data矩阵中依次存放矩阵A中每一行的非零元，indices矩阵中依次存放相应的非零元在矩阵A中的列索引。 这种存储方式的空间开销，取决于矩阵A单行非零元个数的最大值。一个潜在的问题是，对于单行非零元最大值远远大于各行非零元平均值的矩阵，ELL存储方式将造成大量的空间浪费。 比如一个N节点星型网络的邻接矩阵，除中心点外，各个点的度为1，而中心点的度为N-1，这种情形就非常不适合使用ELL。

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

$$row = \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 2 & 3 & 3 \end{bmatrix}$$

$$col = \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix}$$

$$data = \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix}$$

图 5: 对角矩阵A的COO表示

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

$$ptr = \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix}$$

$$indices = \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix}$$

$$data = \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix}$$

图 6: 对角矩阵A的CSR表示

4.1.3 COO

Coordinate(COO)格式是一种非常简单、直观的稀疏矩阵存储格式。如图 5所示：

由row、col、data三个向量，存放矩阵A中非零元的位置索引及其相应的值。

4.1.4 CSR

与COO格式类似，Compressed sparse row(CSR)格式采用三个向量，显示地对非零元信息进行存储，如图 6。 区别在于对行索引的描述。对于一个行数为M的矩阵，CSR存放M+1个指针至向量ptr。前M个指针，分别记录每一行元素在indices和data向量中的起始偏移量；第M+1个指针存放的偏移量，等于第M行的偏移量与第M行非零元个数的和。

CSR存储方式在获得了一定的存储效率优化的同时，可以通过一个简单的减法运算(ptr[i+1]-ptr[i])来获得第i行非零元的总数，这一点带来了更多计算性能上的优化。

5 基于CUDA实现SpMV

基于CUDA编写的GPU并行程序，可以分为运行在CPU上的串行部分，和运行在GPU上的并行部分；以host代指CPU，以device代指GPU.运行在GPU上的线程，被称为kernel.将一个计算任务划分成若干个可以并行执行的子任务的过程，主要体现在对kernel的编写上。

图 7展示了一个基于ELL格式的SpMV kernel的实现。其中，参数data和indices是两个一维数组的指针，分别指向ELL中的矩阵data和矩阵indices按照列优先次序组织成的一维数组。

kernel函数是对SIMT结构中，每个线程的行为描述。如图 7中的kernel所示，在计算y=Ax时，每个线程计算y的一个分量，即计算矩阵A的某一行和向量x的内积。

由于矩阵data和矩阵indices按照列优先的方式组织为一维数组，同一个wrap内的多个线程对数组data和数组indices的访存地址是连续的。这一点有利于充分利用底层硬件的访存特性，实现联合访存，以提高访存效率。

6 实验验证

这一部分对分别在CPU和NVIDIA GPU上基于COO、CSR、ELL这三种压缩格式实现SpMV的性能进行了对比分析实验。 实验矩阵测例来自Matrix Market [5],表 1给出了这些矩阵测例的几个重要属性。 实验测试脚本见 [6]。实验基于2.3 GHz Intel Core i7 CPU和NVIDIA GeForce GT 650M 512 MB GPU,

```
__global__ void spmv_ell_kernel(const int num_rows, const
    int num_cols, const int num_cols_per_row, const int
    * indices, const float * data, const float * x,
    const float * y)
{
  int row = blockDim * blockIdx.x + threadIdx.x;
  if(row < num_rows)
  {
    float dot = 0;
    for (int i = 0; i < num_cols_per_row; i++)
    {
      int col = indices[num_rows * i + row];
      float val = data[num_rows * i + row];
      dot += val * x[col];
    }
    y[row] += dot;
  }
}
```

图 7: SpMV ELL kernel

CUDA版本为V6.5.12. 实验过程中，仅针对计算过程对各测例进行时间测量 [7]，未包括建立压缩格式的过程和CPU主存和GPU主存之间的数据拷贝过程。

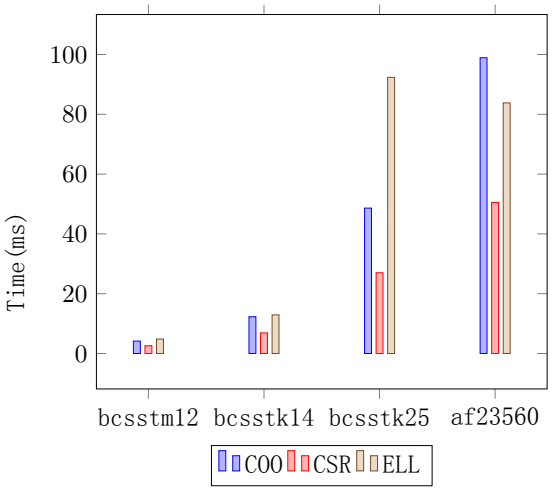
图 8(a)显示了分别基于COO、CSR、ELL三种存储格式，在CPU上实现SpMV的计算性能。 可以看出，CSR格式在CPU上相比其他两种格式，有较高的执行效率。 图 8(b)则显示了三种格式在GPU上的性能表现。可以看出，在GPU上，ELL格式的性能表现更加突出。 基于上述实验结果计算得到的GPU相对CPU的加速比，在图 8(c)中显示。

综上，可以得出一个初步的结论，CSR格式更适用于面向CPU的计算，而ELL格式则更适用于面向GPU计算。

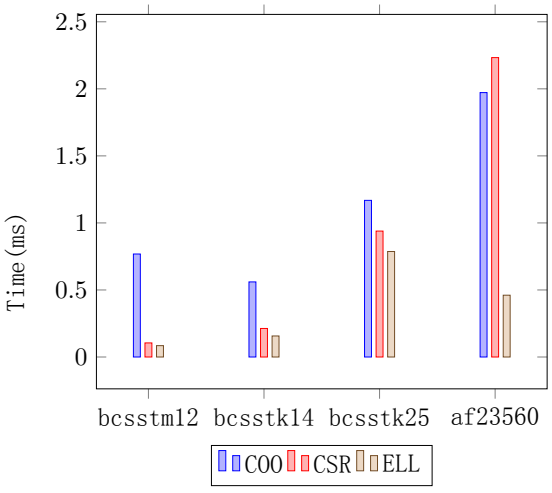
7 总结

本学期，我选修了《高级计算机体系结构》这门课程，并在这门课上，第一次学习GPU体系结构。 出于对GPU加速的好奇，我选择了基于CUDA实验稀疏矩阵-向量乘法这样一个题目。 通过做这次课程报告，我对GPU的体系结构有了更加深入的认识。并掌握了基于CUDA面向GPU编程的基本技能。

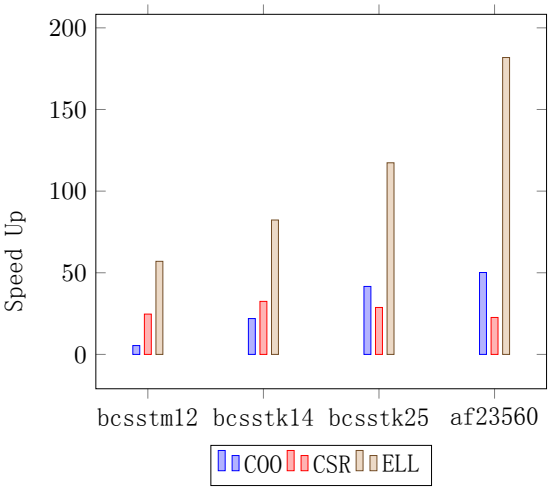
当然，学无止境。基于CUDA实现GPU加速还远不止这么简单，还有很多更加深刻的内



(a) CPU



(b) GPU



(c) Speed Up

图 8: result

容值得去学习和探索，如更好的利用GPU存储层次结构 来对程序进行优化、通过改良kernel设计实现负载均衡等。

8 致谢

感谢成礼智教授的授课以及对科研经历的分享。他的敬业精神，为我树立了良好的榜样。

引用

[1] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," Queue, vol. 6, pp. 40 – 53, Mar. 2008.

[2] "Gpu-accelerated applications." http://www.nvidia.com/content/tesla/pdf/gpu-apps-catalog-mar14-digital-fnl-hr.pdf. Accessed: 2015-1-8.

[3] "Introduction to cuda." http://www.karimson.com/posts/introduction-to-cuda/. Accessed: 2015-1-8.

[4] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[5] "The matrix market." http://math.nist.gov/MatrixMarket/. Accessed: 2015-1-17.

[6] "Evaluation scripts." https://github.com/wuzw10/cuda_evaluation. Accessed: 2015-1-18.

[7] "Implement performance metrics in cuda c/c++." http://devblogs.nvidia.com/parallelforall/how-implement-performance-metrics-cuda-cc/. Accessed: 2015-1-17.

Table 1: 实验矩阵属性 [5]

Matrix name	size	nonzero	nonzero of longest row	nonzero of shortest row
bcsstm12	1473 x 1473	19659	22	2
bcsstk14	1806 x 1806	63454	48	1
bcsstk25	15439 x 15439	252241	59	2
af23560	23560 x 23560	460598	21	9