使用 GPU 加速计算矩阵的 Cholesky 分解

沈 聪 高火涛
(武汉大学电子信息学院 湖北 武汉 430072)

摘 要 针对大型实对称正定矩阵的 Cholesky 分解问题,给出其在图形处理器(GPU)上的具体实现。详细分析了 Volkov 计算 Cholesky 分解的混合并行算法,并在此基础上依据自身计算机的 CPU 以及 GPU 的计算性能,给出一种更为合理的三阶段混合调度方案,进一步减少 CPU 的空闲时间以及避免 GPU 空闲情况的出现。数值实验表明,当矩阵阶数超过 7000 时,新的混合调度算法相比标准的 MKL 算法获得了超过 5 倍的加速比,同时对比原 Volkov 混合算法获得了显著的性能提升。

关键词 图形处理器 乔里斯基分解 加速比 混合算法

中图分类号 TP361 **文献标识码** A DOI:10.3969/j.issn.1000-386x.2016.09.066

ACCELERATING CALCULATION OF CHOLESKY FACTORISATION OF MATRIX WITH GPU

Shen Cong Gao Huotao
(*School of Electronic Information, Wuhan University, Wuhan 430072, Hubei, China*)

Abstract A concrete implementation of Cholesky factorisation on graphic processing unit (GPU) for large real symmetric positive definite matrix is described in this article. We analyse the hybrid parallel algorithm presented by Volkov for computing the Cholesky factorisation in detail. On that basis, and according to the computational performances of CPU and GPU on our own computers, we present a more reasonable hybrid three-phase scheduling strategy, which further reduces the idle time of CPU and avoids the occurrence of GPU in idle status. Numerical experiment shows that the new hybrid scheduling algorithm achieves a speedup of more than 5 times compared with the standard MKL algorithm when the order of a matrix is larger than 7000, and it also observably outperforms the performance of original Volkov's hybrid algorithm.

Keywords GPU Cholesky factorisation Speedup Hybrid algorithm

0 引 言

近年来,随着计算机技术的发展,图形处理器(GPU)越来越强大,并且在计算能力上已经超过了通用 CPU。使用 GPU 计算可以以低廉的价格获得巨大的计算性能,因此成为了科学计算领域的一个应用热点。自 2007 年 NVIDIA 公司推出了 CUDA 运算平台,并使用 C 语言为 CUDA 构架编写程序以来,GPU 计算技术已经广泛应用于诸如信号处理、图像处理、信息安全等热门领域中。在数值线性代数中,GPU 可以用于加速大规模的矩阵计算问题,包括矩阵的分解和求特征值以及特征向量等。许多组织和机构在 GPU 上实现了 LAPACK 库中的函数,并发布了相关软件包以供科研人员使用。比较著名的软件包有 CULA 和 MAGMA 等。

Cholesky 分解是矩阵计算中的一个基本分解问题,常常用于求解系数矩阵为对称正定矩阵的线性方程组,由于其计算量比使用 LU 分解求解一般线性方程组的算法约减少一半。因此在科学计算中也有广泛的应用。此外,Cholesky 分解也应用于 Kalman 滤波[1]以及 Monte Carlo 仿真[2]等算法中。随着基本矩阵算法在科学计算中广泛使用以及其处理的数据矩阵越来越大,研究使用 GPU 加速矩阵计算问题是十分必要的。

CUBLAS 是在 GPU 上实现的 BLAS。利用 CUBLAS 库可以很方便地移植 CPU 代码到 GPU 上进行加速计算。自 2008 年 Volkov 等人提出 CPU - GPU 混合计算矩阵的 LU、Cholesky 和 QR 分解算法[5]以来,混合算法思想被广泛应用于各个计算领域,如流体仿真计算[6]、光线跟踪算法[7]等。混合算法思想也因为 Volkov 等人的工作正逐步应用于各基本矩阵算法中,如矩阵的 Hessenberg 约化、二对角化以及矩阵的特征值求解中。使用 CPU - GPU 混合算法可以通过减少 CPU 的空闲时间,进一步加速矩阵计算。然而混合算法的性能很大程度上取决于调度算法的优劣。好的调度方案可以最小化 CPU 与 GPU 的空闲时间,达到混合最优的目的;不适当的调度策略可能导致计算错误或者计算时间的延长。基于此,本文将在最新的 Kerpler GPU 构架上重新考察 Volkov 的混合算法,并给出一种新的调度方案,以达到混合更优的目的。数值实验表明基于新的三阶段混合调

收稿日期:2015 - 04 - 10。湖北省自然科学基金重点项目(ZRZ2014000286)。**沈聪**,硕士生,主研领域:并行计算。**高火涛**,教授。

度算法的函数 New_Sche 相比 MKL 中的 dpotrf 函数最高获得了 5.2 倍的加速比,而且其计算性能显著优于原 Volkovde 混合算法。文中给出的针对 Cholesky 分解的调度策略同样适用于矩阵的其他分解及约化算法。

1　使用 CPU 计算 Cholesky 分解

给定一个对称正定矩阵 A,则存在主对角元素全为正数的下三角矩阵 L 满足:

$$A = LL^{\mathrm{T}} \tag{1}$$

通常称下三角矩阵 L 为矩阵 A 的 Cholesky 因子。下面给出 Cholesky 分解的直接算法[3]及基于块的下递推更新算法[4]。

1.1　直接 Cholesky 分解

对于对称正定矩阵 A 直接由式(1)得到如下等式:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} r_{11} & 0 & \cdots & 0 \\ r_{21} & r_{22} & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ r_{n1} & r_{n2} & \cdots & r_{nn} \end{bmatrix} \begin{bmatrix} r_{11} & r_{21} & \cdots & r_{n1} \\ 0 & r_{22} & \cdots & r_{n2} \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & r_{nn} \end{bmatrix} \tag{2}$$

则对于第 j 个对角元素 a_{jj},有:

$$a_{jj} = r_{j1}^2 + r_{j2}^2 + \cdots + r_{jj}^2 \tag{3}$$

对于位置为$(i,j)(i>j)$的元素,我们可得如下等式:

$$a_{ij} = r_{i1}r_{j1} + r_{i2}r_{j2} + \cdots + r_{ij}r_{jj} \tag{4}$$

由此可得到求解 Cholesky 分解的公式如下:

$$r_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} r_{jk}^2} \tag{5}$$

$$r_{ij} = (a_{ij} - \sum_{k=1}^{j-1} r_{ik}r_{jk})/r_{jj} \quad i = j+1, \cdots, n \tag{6}$$

若令 $j=1,2,\cdots,n$,循环使用式(5)、式(6)计算 L 的下三角元素,这样便得到计算 Cholesky 分解的直接算法。由于该直接算法主要是 Level 2 BLAS 操作,不适合移植到 GPU 上计算。

1.2　基于块的 Cholesky 分解

BLAS 库内部实现了各种线性代数的基本运算。由于其代码的高效性和完整性,因此一直为科研人员广泛使用。其中 Level 3 BLAS 是依据现代计算机的内存等级进行了大量的优化处理,使得其计算性能远高于 Level 2 BLAS。类似的结论也在 GPU 上成立。

为了能利用矩阵的子块进行计算,下面使用 Cholesky 分解的块递推更新算法。对矩阵 A 进行分块,如下:

$$A = \left[\begin{array}{c|c} A_{11} & B^{\mathrm{T}} \\ \hline B & \hat{A} \end{array}\right] \quad A_{11} \in \Re^{r\times r}$$

则由式(1)有:

$$A = \left[\begin{array}{c|c} A_{11} & B^{\mathrm{T}} \\ \hline B & \hat{A} \end{array}\right] = \left[\begin{array}{c|c} L_{11} & 0 \\ \hline S & \hat{L} \end{array}\right] \left[\begin{array}{c|c} L_{11}^{\mathrm{T}} & S^{\mathrm{T}} \\ \hline 0 & \hat{L}^{\mathrm{T}} \end{array}\right]$$

于是得到如下递推公式:

$$L_{11} = cholesky(A_{11}) \tag{7}$$

$$S = B \cdot L_{11}^{-\mathrm{T}} \tag{8}$$

$$\hat{L} \cdot \hat{L}^{\mathrm{T}} = \hat{A} - S \cdot S^{\mathrm{T}} \tag{9}$$

对矩阵按式(7) – 式(9)依次递推更新下去,便可得到 Cholesky 分解。

具体的递推过程如图 1 所示。

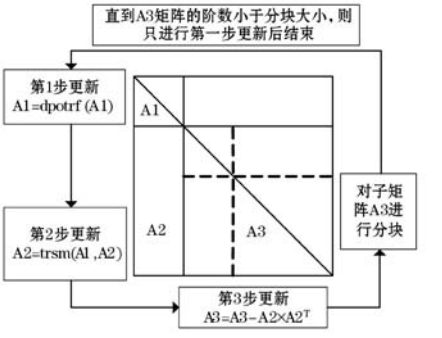

图 1　基于块的 Cholesky 分解

1.3　算法及分析

基于块的 Cholesky 分解算法如下:

Step1　使用 Intel MKL 库中的 dpotrf 函数计算第一个子块 $A1$ 的 Cholesky 分解;

Step2　使用 BLAS 库中的 Level 3 函数 *trsm* 求解矩阵方程 $X \times A1^{\mathrm{T}} = A2$;

Step3　使用 BLAS 库中的 Level 3 函数 *syrk* 更新尾部矩阵 $A3 = A3 - A2 \times A2^{\mathrm{T}}$;

Step4　对尾部矩阵 $A3$ 重新分块并针对 $A3$ 重复执行以上三步,直到计算完最后一个子块的 Cholesky 分解则结束。

在余部矩阵 $A3$ 的阶数较大时,Step 3 占据一次循环约 80% 的计算时间。而 $A1$ 由于分块大小的固定,计算量不会改变,Step 1 所占的计算时间非常小。当 $A2$ 的行远大于分块值时,Step 2 几乎占据了剩余 20% 的计算时间。

2　使用 GPU 计算 Cholesky 分解

2.1　CUBLAS 库

CUBLAS 是由 NVIDIA 公司提供的并在 CUDA 构架的 GPU 上实现 BLAS 的函数库。利用 CUBLAS 库可以很方便地移植矩阵计算的 CPU 代码到 GPU 上进行计算。由于矩阵算法中主要涉及矩阵与矩阵相乘等的 Level 3 操作以及矩阵和向量相乘等的 Level 2 操作,且 BLAS 和 CUBLAS 库中的函数是经过高度优化的,并被全世界广泛认可,因此在计算 Cholesky 分解中,可直接使用该库中的函数在 CPU 或 GPU 上实现矩阵的 Level 2 或 Level 3 操作。这样,可以将主要精力放在 Cholesky 分解算法的任务划分及任务调度上。

2.2　使用 CUBLAS 计算 Cholesky 分解

由于 CUBLAS 库中提供了 *trsm* 和 *syrk* 函数,但没有提供 *potrf* 函数。对此,可以在每次循环时先将需要计算 Cholesky 分解的子矩阵块传回到 CPU 上并利用 MKL 中的 *potrf* 函数计算其 Cholesky 分解,然后在将结果传输至 GPU 对应的位置。再依次使用 CUBLAS 库函数在 GPU 上更新矩阵的剩余部分。然而这样做的结果是当 GPU(或 CPU)执行计算任务时,CPU(或 GPU)处于闲置状

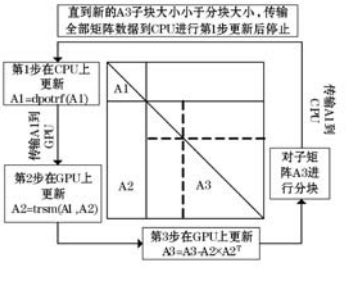

图 2　使用 CUBLAS 加速 Cholesky 分解

态,这导致了硬件资源的浪费。使用 CUBLAS 计算 Cholesky 分解的具体过程如图 2 所示。

这样直接使用 CUBLAS 库函数加速实对称正定矩阵的 Cholesky 分解的效果也是十分明显的。首先由于在 GPU 上的操作全为 Level 3 的 BLAS 操作;其次,相对较小的分块值 nb,如 $nb<512$,在本实验环境下,使用 MKL 库函数计算 512 阶对称正定矩阵的 Cholesky 分解耗时约为 4 ms,而来回传输 512 阶矩阵总共耗时约 2 ms。相比较大的对称正定矩阵而言,直接使用 CUBLAS 库函数已经可以称之为充分利用 GPU 进行计算了。

2.3 Volkov 的混合算法及分析

Volkov 等人于 2008 年给出了 CPU - GPU 混合计算 Cholesky 分解的算法[5],并在 NVIDIA 的 GTX 280 上获得了良好的加速效果。该算法随后被 MAGMA 软件包采纳,成为计算 Cholesky 分解的标准算法之一。图 3 - 图 6 以 4×4 的矩阵块为例,简单描述 Volkov 的混合算法。

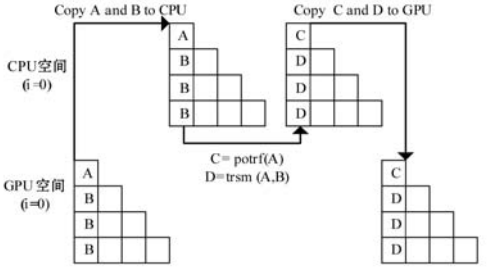

图 3 第 1 步更新

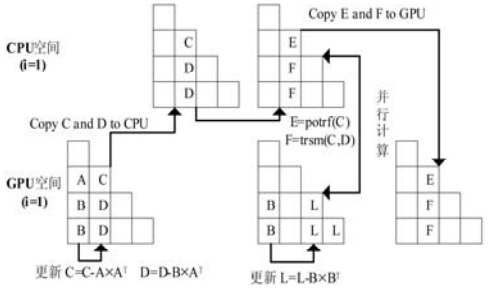

图 4 第 2 步更新

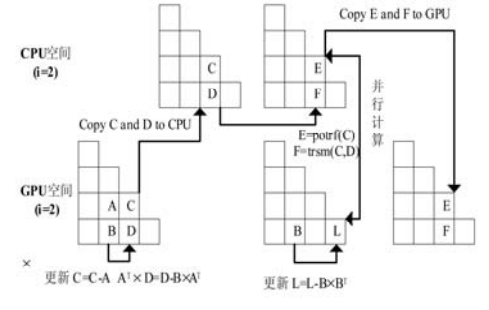

图 5 第 3 步更新

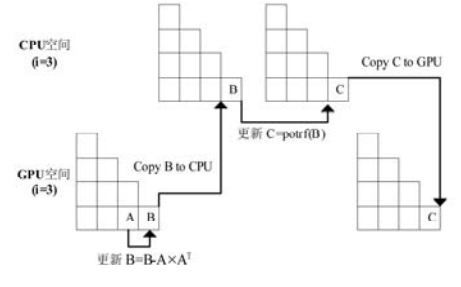

图 6 第 4 步更新

Volkov 的混合算法巧妙地移动了矩阵循环的更新顺序,这样余部矩阵的上次更新和本次子块的 Cholesky 分解及其所在列的更新可以同时进行。

考察 Volkov 算法中的混合并行部分,对 GPU 上的某次迭代更新后,有如图 7 所示的剩余矩阵:

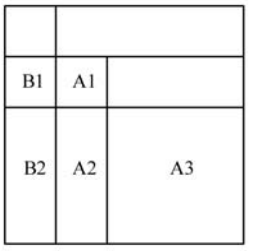

图 7 某次剩余矩阵

其中矩阵 $A1$、$A2$、$A3$ 为剩余待更新矩阵,子块 $B1$、$B2$ 为已求出的 Cholesky 分解部分。之后,在 GPU 上按如下公式对 $A1$ 与 $A2$ 进行更新:

$$A1 = A1 - B1 \times B1^{\mathrm{T}} \tag{10}$$

$$A2 = A2 - B2 \times B1^{\mathrm{T}} \tag{11}$$

然后将 $A1$ 与 $A2$ 异步传输到 CPU。同时,在 GPU 上对余部矩阵按如下公式进行更新:

$$A3 = A3 - B2 \times B2^{\mathrm{T}} \tag{12}$$

在 CPU 上再次对 $A1$ 和 $A2$ 更新:

$$A1 = potrf(A1) \tag{13}$$

$$A2 = trsm(A1, A2) \tag{14}$$

其混合并行的示意如图 8 所示。

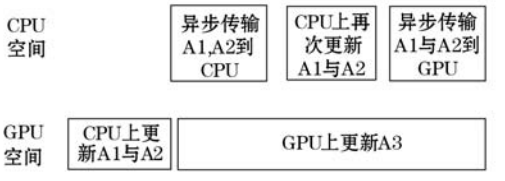

图 8 原 Volkov 的混合调度策略

可以发现在 GPU 上更新矩阵 $A1$、$A2$ 与更新 $A3$ 是相互独立的。而且当剩余待更新矩阵阶数较大时,可以考虑将 $A1$ 的首次更新移到 CPU 上进行。注意,对 $A2$ 的两次更新不能同时移到 CPU 上进行,因为对 $A2$ 的两次更新计算量都比较大,在 CPU 上计算时,耗时很多,极容易超过 GPU 上对 $A3$ 的更新耗时,从而导致 GPU 的空闲。

此外,当剩余待更新的矩阵较小时,在 GPU 上更新 $A3$ 的时间较短。此时不适合再将子矩阵 $A2$ 转移到 CPU 上进行更新,否则会使得 GPU 出现长时间空闲。

整个混合计算过程中对数据传输采用异步方式,使得每次数据传输时间被 GPU 计算或 CPU 计算隐藏。

对此,我们综合考虑了计算的 CPU 以及 GPU 的计算性能,以 Volkov 的混合算法为基础,给出了一种新的调度策略。

2.4 新的混合调度算法

记剩余待更新的矩阵 $A3$ 的阶数为 ns。将新的混合调度算法分为以下三个阶段:

阶段 1 当 $ns > x$ 时,更新子矩阵 $A3$ 耗时较长,因此 CPU 可以做更多的工作以减少空闲时间。由于对子矩阵块 $A2$ 的两次更新只能有一次移到 CPU 上进行,故有以下 2 种调度策略:

1) 将 $A2$ 的第一次更新放在 GPU 上执行,如图 9 所示。

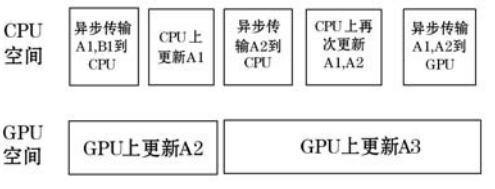

图 9 阶段 1 的混合调度策略 1

2）将 $A2$ 的第二次更新放在 GPU 上执行,如图 10 所示。

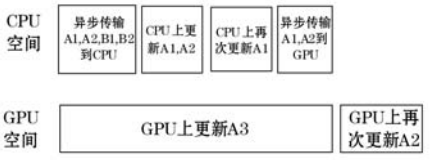

图 10　阶段 1 的混合调度策略 2

通过分析 $A2$ 两次更新的计算量,可以发现 $A2$ 的前次更新计算量为 $A2$ 再次更新计算量的两倍。为防止 GPU 出现空闲,只有当 $ns > x1\,(x1 > x)$ 时,GPU 上对 $A3$ 的更新任务能完全隐藏 CPU 上的计算任务,采用混合调度策略 2;否则,$x < ns < x1$ 时,将采用混合调度策略 1,以防止 GPU 的空闲。这样相互选择调度是最优的结果,其 $x1$、$x2$ 也需要根据计算机的 CPU 及 GPU 的计算性能大致估计出来。

阶段 2　当 $y < ns < x$ 时,遵循 Volkov 的混合并行策略。混合调度策略参见图 8 所示。

阶段 3　当 $ns < y$ 时,在 GPU 上首先更新 $A1$,然后启动异步传输 $A1$ 到 CPU 上进行再次更新操作。其余的更新操作在 GPU 上完成,如图 11 所示。

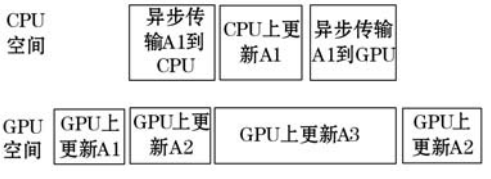

图 11　阶段 3 的混合调度策略

其中 x 与 y 的值需要依据各自计算的 CPU 和 GPU 的性能来大致确定,以保证 GPU 不会出现空闲时间并且 CPU 的空闲时间最小。这样的混合调度策略则是非常合适的。

同时,要对 Volkov 混合算法的第一步和最后一步进行调整。在第一步中,对第一主列块的更新与余部矩阵传输到 GPU 上可以异步进行,这样其中的一个耗时会被隐藏。最后一步更新时,可以先将最后一个子块先传回至 CPU,然后将更新最后一个子块与传回剩余矩阵数据异步执行。

对于分块大小 nb,可以参考 MAGMA 软件包中关于分块值的设定。针对 Kerpler 构架的 GPU,可将分块值设定如下:

$$nb = \begin{cases} 256 & n \leqslant 1500 \\ 512 & \text{其他} \end{cases} \tag{15}$$

3　数值实验及结果分析

3.1　实验环境

本文中描述的 GPU 计算过程均是在 NVIDIA 的 Kerpler 构架的 GPU 上实现的。有关数值试验的硬件和软件平台见表 1 所示。

表 1　实验环境

环境参数	数值
操作系统	Windows 7
CPU	Intel Core – i7 3.20 GHz
GPU	GTX 680
计算能力	3.0
SM	1 个
CUDA 核心数	448 个

3.2　实验结果与分析

实验测试 1000 ~ 10 000 阶对称正定矩阵,而且使用双精度数据类型,以保证计算结果的精确性。

首先分别给出本实验环境下 MKL 中执行 dgemm 以及 dtrsm 的耗时以及 CUBLAS 中一执行 dgemm,dtrsm 以及 syrk 的耗时的一个参考值,如表 2 所示。该数据用于估计划分阶段的 x(包括 $x1$)与 y 值。

表 2　各函数测试结果(单位:毫秒)

矩阵规模	mkl dgemm	mkl dtrsm	cublas dgemm	cublas dtrsm	cublas dsyrk
1000	23.620	13.795	4.71	4.232	5.721
2000	47.211	25.987	9.22	7.956	19.710
3000	68.212	38.264	13.734	11.840	43.543
4000	90.321	50.132	18.164	15.235	70.055
5000	114.311	61.831	22.843	19.123	107.848
6000	135.573	75.577	27.361	22.351	153.269
7000	127.576	86.446	31.792	24.314	210.781
8000	178.237	100.324	33.612	27.261	267.894
9000	201.363	111.211	40.875	32.163	341.252
10000	223.733	124.712	45.418	35.656	422.377

其中针对 dgemm 函数,m 为表格的第一列,$n = k = 512$。而 dtrsm 与 dsyrk 函数的 n 为表格第一列,$k = 512$。

表 3 给出了数据矩阵在 CPU 与 GPU 之间的传输耗时的一个参考值。

表 3　数据传输测试(单位:毫秒)

矩阵规模	512	1000	2000	3000	4000
CPU – To – GPU	1.121	3.353	12.637	20.382	50.546
GPU – To – CPU	0.942	2.746	11.421	17.204	40.065

下面考虑 $x = 6000$ 时,使用阶段 1 的混合调度策略 1 是合适的。首先依据表 3 可得到,异步传输 $A1$,$B1$(矩阵规模均为 512×512)以及在 CPU 上更新 $A1$ 的时间被 GPU 上更新 $A2$ 隐藏。而异步传输 $A2$(矩阵规模 6000×512)耗时 $t1 < 20$ ms,在 CPU 上再次更新 $A2$ 耗时 $t2 < 80$ ms,异步传回 $A1$,$A2$ 耗时 $t3 < 20$ ms,而 GPU 上对矩阵 $A2$(规模 6000 阶)的更新耗时为 $t \approx 150$ ms $> (t1 + t2 + t3)$,故可取 $x = 6000$。类似,我们可以由此得到比较合理的 $x1$,x 以及 y 的值,分别如下:

$$x1 = 8000 \quad x = 6000 \quad y = 4000$$

最后,对 1000 ~ 10000 阶对称正定矩阵分别使用 4 种方法进行计算。表 4 给出了 4 种方法计算 1000 ~ 10 000 阶对称正定矩阵的 Cholesky 分解的结果。其中可以看到基于新调度策略的函数 New_Sche 明显优于使用 CUBLAS 直接计算 Cholesky 分解以及基于 Volkov 的混合算法的函数 Volkov_S。

表 4　各算法测试结果(单位:秒)

矩阵规模	MKL	CUBLAS	Volkov_S	New_Sche	New_Sche 加速比
1000	0.023	0.018	0.022	0.014	1.643
2000	0.114	0.062	0.083	0.058	1.966
3000	0.46	0.160	0.162	0.144	3.194
4000	1.054	0.309	0.295	0.271	3.889
5000	1.977	0.552	0.500	0.485	4.076
6000	3.374	0.873	0.773	0.702	4.806
7000	5.279	1.315	1.188	1.021	5.170
8000	7.726	1.825	1.705	1.488	5.192
9000	10.477	2.603	2.375	2.015	5.199
10000	14.335	3.385	3.152	2.774	5.167

(下转第 305 页)

由表 5 可知,在最优聚类数目 $K=1500$ 情况下,本文提出的 Fisher-VSM 聚类算法相比于 VSM 聚类算法,其分类准确率提高了 10.57%。因此,本文提出的聚类算法具有较好的性能。

4 结　语

本文提出一种基于多约简 Fisher-VSM 和 SVM 的文本情感分类算法。借助于 TF-IDF 权重函数兼顾文档特征项局部和全局分布信息的优势,采用 Fisher 准则选择高判别性的低维的 TF-IDF 特征,降低文档的维度,建立低维 Fisher-VSM。根据 Fisher-VSM 之间的相似度,对文档模型进行聚类,从而减少文档集的数量。从文档的维数及数量两个方面的约简,提高了 SVM 的分类性能和训练速度。实验结果表明,本文提出的算法维度约简率为 44.7%,文档数目约简率为 37.5%,其分类准确率为 93.31%,是一种可行的高效的文本情感分类算法。高效准确的文本评论观点的判定,有利于决策支持。本文提出的多约简文本聚类算法,不仅有利于 SVM 的训练,也适用于其他分类方法,期望对机器学习算法在文本情感分类领域的应用有所借鉴。

参 考 文 献

[1] 樊小超. 基于机器学习的中文文本主题分类及情感分类研究[D]. 南京理工大学,2014.

[2] Pang B, Lee L, Vaithyanathan S. Thumbs up? Sentiment classification using machine learning techniques[C]//Proceedings of the Conference on Empirical Methods in Natural Language Processing, Philadelphia, 2002:79 - 86.

[3] Whitelaw C, Garg N, Argamon S. Using appraisal groups for sentiment analysis[C]//Proceedings of the ACM Conference on Information and Knowledge Management, Bremen(DE), 2005:625 - 631.

[4] 陈培文,傅秀芬. 采用 SVM 方法的文本情感极性分类研究[J]. 广东工业大学学报,2014,31(3):95 - 101.

[5] 肖正,刘辉,李兵. 一种基于语义距离的 Web 评论 SVM 情感分类方法[J]. 计算机科学,2014,41(9):248 - 252,284.

[6] 杨经,林世平. 基于 SVM 的文本词句情感分析[J]. 计算机应用与软件,2011,28(9):225 - 228.

[7] 周城,葛斌,唐九阳,等. 基于相关性和冗余度的联合特征选择方法[J]. 计算机科学,2012,39(4):181 - 184.

[8] 朱艳辉,栗春亮,徐叶强,等. 一种基于多重词典的中文文本情感特征抽取方法[J]. 湖南工业大学学报,2011,25(2):42 - 46.

[9] Wang Suge, Li Deyu, Song Xiaolei, et al. A feature selection method based on improved fisher's discriminant ratio for text sentiment classification[J]. Expert Systems with Applications, 2011, 38(7): 8696 - 8702.

[10] 孙劲光,马志芳,孟祥福. 基于情感词属性和云模型的文本情感分类方法[J]. 计算机工程,2013,39(12):211 - 215,222.

[11] 王素格. 基于 web 的评论文本情感分类问题研究[D]. 上海大学,2008.

[12] 谷文成,柴宝仁,韩俊松. 基于支持向量机的垃圾信息过滤方法[J]. 北京理工大学学报,2013,33(10):1062 - 1066,1071.

[13] 张璇. 基于 Fisher 准则的说话人识别特征参数提取研究[D]. 湖南大学,2013.

[14] 王飒,郑链. 基于 Fisher 准则和特征聚类的特征选择[J]. 计算机应用,2007,27(11):2812 - 2813,2840.

[15] 刘靖明,韩丽川,侯立文. 基于粒子群的 K 均值聚类算法[J]. 系统工程理论与实践,2005,25(6):54 - 58.

[16] 谭松波. 中文情感挖掘语料-ChnSentiCorp [EB/OL]. [2012 - 08 - 10]. http://www.searchforum.org.cn/tansongbo/corpus-senti.htm.

(上接第 287 页)

4 结　语

科学计算中使用 CPU 与 GPU 混合并行计算的例子层出不穷,然而最困难的在于设计一种负载均衡的混合调度方案,从而使得 CPU 的空闲时间尽可能小,同时又不能让 GPU 空闲。糟糕的混合调度方案会使得整体计算性能下降,或者使得计算不正确,从而失去了混合计算的目的,这都是需要避免的。合适的混合调度策略往往需要依据自身计算机的 CPU 以及 GPU 的计算性能来共同确定。首先细致分析每步操作在 CPU 以及 GPU 上的计算速度,然后合理划分计算任务到 CPU 和 GPU 上执行。再者,算法中会大量使用异步传输操作,使得这些数据传输耗时被 CPU 和 GPU 上的计算时间隐藏,这也是优化加速的一个重要部分。最后,本文中没有考虑分块值对混合调度策略的影响。由实验的分析可知,分块值与调度策略有直接关系,影响阶段值 x1,x 以及 y 的确定。如何确定最优的分块值以及相应的调度策略将是后面要进行的工作。

参 考 文 献

[1] Chandrasekar J, Kim I S, Bernstein D S, et al. Reduced-Rank Unscented Kalman filtering using Cholesky-based decomposition[C]//American Control Conference, June, 2008:1274 - 1279.

[2] Yu H, Chung C Y, Wong K P, et al. Probabilistic Load Flow Evaluation With Hybrid Latin Hypercube Sampling and Cholesky Decomposition[J]. IEEE Transactions on Power Systems, 2009, 24(2):661 - 667.

[3] David S. Watkins. Fundamentals of Matrix Computations[M]. New York: John Wiley and Sons, 2013.

[4] Gene H Golub, Charles F, Van Loan. Matrix Computations[M]. Baltimore: Johns Hopkins University Press, 2013.

[5] Volkov V, Demmel J W. Benchmarking gpus to tune dense linear algebra[C]//Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, Nov, 2008:1 - 11.

[6] 胡鹏飞,袁志勇,廖祥云,等. 基于 CPU-GPU 混合加速的 SPH 流体仿真方法[J]. 计算机工程与科学,2014,36(7):1231 - 1237.

[7] 张健,焦良葆,陈瑞. CPU-GPU 混合平台上动态场景光线跟踪的研究[J]. 计算机工程与应用,2012,48(21):151 - 154.

[8] Yaohung M Tsai, Weichung Wang, RayBing Chen. Tunning Block Size for QR Factorization on CPU-GPU Hybrid Systems[C]//Proceedings of the IEEE 6th International Symposium on Embedded Multicore Socs, Sept, 2012:205 - 211.

[9] John Cheng, Max Grossman, Ty McKercher. CUDA C Programming[M]. Indianalpois: John Wiley & Sons, 2014.

[10] 刘金硕,邓娟,周峥,等. 基于 CUDA 的并行程序设计[M]. 北京:科学出版社,2014.