第 42 卷 第 6A 期
2015 年 6 月

计 算 机 科 学
Computer Science

Vol. 42 No. 6A
June 2015

CPU-GPU 协同计算的并行奇异值分解方法

周 伟 戴宗友 袁广林 陈 萍

(中国人民解放军陆军军官学院计算机教研室 合肥 230031)

摘 要 在目标跟踪应用中,常常采用奇异值分解(SVD)作为基本工具进行动态建库。然而当每秒处理的数据量较大、计算精度要求较高时,SVD 的计算耗时往往无法满足应用的实时性能要求。针对这一问题,提出了 CPU-GPU 协同计算的并行奇异值分解方法。该方法利用 GPU 与 CPU 间的异步执行,对奇异值分解过程进行划分从而构造软件流水线,大大挖掘软硬件的并行性。实验表明,该方法比一般的基于 GPU 的 Jacobi 方法有约 23% 的性能提升。相对于 CPU 上的 Intel MKL 的奇异值分解函数获得了 6.8x 的加速比,满足了应用中的实时性能要求。

关键词 GPU,协同计算,Jacobi 方法,奇异值分解

中图法分类号 TP391 文献标识码 A

Parallelized Singular Value Decomposition Method with Collaborative Computing of CPU-GPU

ZHOU Wei DAI Zong-you YUAN Guang-ling CHEN Ping

(Computer Teaching & Research Room, PLA Army Officer Academy, Hefei 230031, China)

Abstract In applications of tracking, singular value decomposition (SVD) is often used as a basic tool for dynamic library construction. However, facing the large amount of dataset per second and high accuracy requirements, the calculation of SVD is too time-consuming to satisfy the real-time constraint in the applications. This paper put forward the CPU-GPU collaborative method of parallel singular value decomposition. It utilizes the feature of asynchronous execution of GPU, and organizes the process of SVD into pipeline-style, with which we can largely exploit the parallelism. Experiments show that this method outperforms better than normal parallelized one-sided Jacobi method for SVD on GPU by 20%. Compared with the Intel MKL SGESVD on CPU, out approach can achieve a 6.8x performance improvement, which makes it satisfy the requirements of real-time applications.

Keywords GPU, Collaborative computing, Jacobi, SVD

1 引言

在目标跟踪应用中常常需要用奇异值分解作为基本的计算工具。而对每秒一定规模和数量的矩阵进行奇异值分解,其计算成本之高低往往成为整个应用能否达到实时性能的关键。

求解矩阵的 SVD 通常是基于 QR 分解和基于 Jacobi 旋转两种方法的变形[1]。前者是一个严格的串行过程,并行性有限,在 GPU 上获得的加速并不理想。而 Jacobi 旋转则具备内在的大量并行性,数据分布规整,计算时通信分布规则。尤其是单边 Jacobi 方法,不仅其计算和通信的并行性易于发掘,其访存需求也很容易利用 GPU 高效的 CUDA 访存模型。

目前关于 SVD 在 GPU 上的并行化工作也得到了很多关注[4-7,9]。然而,多数工作[4-6,9]利用了 GPU 对 SVD 主要迭代步骤进行计算,其余部分再交由 CPU,CPU 与 GPU 间串行工作,没有很好地并行工作。Ding Liu 等[7]则通过 Divide-and-Conquer 的方法均衡了 CPU 与 GPU 之间的任务负载,使两者能并行工作,对计算过程进行有效的时间隐藏。但这种方法并非是对 Jacobi 算法的直接并行化,而是采用分治的思想,其编程模式对算法改写及调试的难度较大,并不适合于编程者直接改造和设计出相应的并行算法。

本文提出了 CPU-GPU 协同计算的并行奇异值分解方法,利用 GPU 与 CPU 间异步执行的机制,对基于单边 Jacobi SVD 中的迭代计算过程进行分解,并将其组织成软件流水线。各个阶段以流水方式执行,大大挖掘算法在异构计算平台上的并行性,同时使 CPU 与 GPU 的计算负载得到了均衡。通过设计,在流水线中不但隐藏了算法的部分计算过程,同时也隐藏了主存与设备显存之间进行数据同步和拷贝的时间代价。

本文实现中的部分计算也使用了 NVIDIA 的 CUBLAS 和核函数。实验结果表明,本文方法相对于 Core i5 3330 CPU 上运行的 Intel MKL 的奇异值分解函数有 6.5x 的加速比,完全满足了应用中的实时性能要求。显然,这种基于 CPU-GPU 加速的方法对具有相似特点和实时性能要求的算法的并行化设计提供了一种有效的参考思路。

本文第 2 节介绍奇异值分解及 GPU 加速奇异值分解的相关工作;第 3 节介绍 CPU-GPU 协同计算加速奇异值分解的并行方法;第 4 节给出实验与分析,最后进行了总结。

本文受陆军军官学院科研学术基金项目(2012XYJJ-056)资助。

周 伟(1982—),男,讲师,主要研究方向为流处理器体系结构、并行计算,E-mail:greatzv@mail.ustc.edu.cn。

2 奇异值分解及并行化相关工作

2.1 奇异值分解及 Hestenes 算法

奇异值分解作为一个基本的数学工具广泛应用于 PCA、信号处理、目标识别与跟踪等领域。设 $A\in R^{m\times n}$，则必然存在正交矩阵：$U=[u_1,u_2,\cdots,u_m]\in R^{m\times m}$ 和 $V=[v_1,v_2,\cdots,v_n]\in R^{n\times n}$ 使得

$$U^{\mathrm{T}}AV=\begin{bmatrix}\Sigma r & 0\\ 0 & 0\end{bmatrix}_{m-r}^{r}$$

其中，$\Sigma r=\mathrm{diag}(\sigma_1,\sigma_2,\cdots,\sigma_r)$，$\sigma_1\geqslant\sigma_2\geqslant\cdots\geqslant\sigma_r>0$，将其称为矩阵 A 的奇异值分解，σ_i 是 A 的奇异值。u_i 和 v_i 分别是第 i 个左奇异向量和第 i 个右奇异向量。

单边 Jacobi 的 SVD 方法于 1958 年由 Hestenes 提出[3]。基本思想是利用一系列 Jacobi 平面旋转实现数据矩阵的单边正交化方法，可以直接获得右奇异向量和奇异值。具体如算法 1 所示。

算法 1　Hestenes 算法串行伪码

输入：A 为 m×n 矩阵，V 为单位矩阵

输出：A 奇异值分解项：U、Σ、V(A=UΣV^T)

```
Hestenes-SVD(A,Σ,U,V){
1. Load(A)
2. iter=iterMax
3. while(iter--->0){
4.    pass=TURE;
5.    for(i=0;i<n;i++){
6.       for(j=i+1;j<n;j++){
7.          if(inProd(col(i),col(j)==0) continue;
8.          pass=FALSE;
9.          if(norm(col(i)),norm(col(j)))
10.                exchange(col(i),col(j));
11.          //计算本次旋转角θ
12.          caculateθ(col(i),col(j),θ);
13.          //对 A 进行旋转,累积计算 V
14.          doRotate(A,V,θ);.}.}
15.   if(pass) break;.}
16.Σ= dilivery(A)//获取奇异值
17.U=(Ak,Σ)}//获取左奇异矩阵
```

该算法的计算复杂度为 $O(n^3)$[1,2]。由于其耗时较大，使其在实时性领域中的应用受到制约。例如，应用中需要每秒钟处理 2～4 个规模为 500×500 浮点数矩阵的 SVD 求解。目前在 CPU 上的串行算法处理单个 500×500 浮点数矩阵的 SVD 求解就已经超过 1.5s，远远无法达到实时处理要求。

2.2 GPU 加速 SVD 的相关工作

目前，GPU 加速 SVD 有了很多的工作。文献[4-7,9]都对 SVD 在 GPU 上的并行化进行了研究。

Sheetal 等人[5]在 GPU 上并行化了传统的 QR 迭代算法。他们的方法中，假定输入数据已经存储在 GPU 的显存中，并且数据一直保持在显存中。这样就忽略了数据在主机内存与显卡的显存间传输的代价。

Vedran[6]，张舒等[9]都在 GPU 上并行化了单边 Jacobi 的 SVD 方法。Charlotte 等[4]则比较了单边 Jacobi SVD 方法和基于 QR 的方法在 GPU 上的并行化情况。以上这些方法都是将主要的计算步骤交给 GPU 完成。CPU 接下来串行地完成其余步骤。在 GPU 执行任务时，CPU(主流是 2～4 核心，4～8 线程)则处于闲置状态，导致负载的不均衡。

Ding Liu 等[7]提出了针对 CPU-GPU 异构系统的分而治之的算法，算法在获得良好的数值稳定性的同时，通过重叠 CPU 与 GPU 的计算过程实现隐藏，并考虑到了主存到设备端的传输带宽。但这种方法并不是对 Jacobi 算法直接并行化，而需要依据分而治之的思想重新设计算法，改写的编程及调试难度较大，并不适合于对现有算法进行改造。

目前，大多数基于 GPU 的并行化 SVD 方法都是将主要的迭代步骤交由 GPU 进行计算。而 GPU 与 CPU 间执行方式基本是串行同步的方式。GPU 运算时，由于算法的依赖性制约，多核 CPU 总处于闲置状态。为获得应用中要求的实时性能，同时不失编程模型的易用性，我们提出了采用 CPU-GPU 协同计算的奇异值分解的并行方法。利用 GPU 与 CPU 间异步并行执行的特性，使算法以流水线的方式在 CPU 与 GPU 间并行执行，大大挖掘了算法在异构计算平台的并行性。

3 CPU-GPU 协同计算加速奇异值分解

这一节，首先给出单边 Jacobi 的 SVD 求解的 GPU 并行化方法。第 2 节中，引入 CPU-GPU 异构平台下的流水线方法来加速奇异值分解，并讨论了在流水线中算法的计算过程的时间隐藏，数据集在设备显存与主存间的同步与传输的时间代价等问题。

3.1 单边 Jacobi SVD 的 GPU 并行化

下面给出 CUDA 平台的一种基本的并行化的 Hestenes 算法。

对于 Hestenes 算法，主要耗时在第 3－15 行的迭代过程。其中每一轮迭代将矩阵 A 中的所有“列对”(矩阵 A 中的两列 p,q)都进行一次计算。具体对每个列对的计算包括：

Step1　根据列对求内积，从而计算旋转角 θ(计算中求得 $\sin\theta,\cos\theta$ 值即可)

Step2　根据旋转角 θ，对矩阵 A 与 V 的列对进行旋转。

由以上分析可知，其余第 16，17 行求右奇异向量和求模获得奇异值的时间相对于这个迭代过程来说，几乎可以忽略不计。因此，在 GPU 上的并行化主要是对算法 1 中的这个 while 循环，其一次循环称为一轮迭代。

在 Hestenes 算法中，一轮迭代的所有列对都将进行一次 Jacobi 旋转。由于数据依赖，一个矩阵一次同时并行执行的 Jacobi 旋转只能达到 $n/2$ 个，那么在一轮迭代中，就需要串行执行 $(n-1)$ 次这样的并行执行的操作。

在 CUDA 计算模型中，将一次能够同时并行执行的 $n/2$ 个列对的 Jacobi 旋转映射到一个线程 Grid 中，这样，对所有列对完成一次正交化旋转需要 $n-1$ 次核函数调用。在一个线程 Grid 中，$n/2$ 个列对相互间没有数据依赖，因此在执行过程中相互不产生通信，即为 $n/2$ 个独立的子问题。可以将每个列对映射到一个线程块(Block)中，则逻辑上 $n/2$ 个线程块同时并行执行。

每个线程块对应处理一个列对的计算任务，即上文的 Step1 和 Step2。在 Step1 中，采用并行规约的方式求内积，然后再将结果存入 Shared Memory。改由 Block 中一个单线程来计算旋转角的正余弦。Step2 是比较规范的并行化，其中，

每个线程对应矩阵 A 和 V 的一个元素的旋转计算。

在算法的 CUDA 实现时，矩阵 A 和 V 都是用两块空间。用乒乓操作，避免大量的显存读写。关于同时并行执行的列对的选取问题，采用的是 Brent 和 Luk[8] 提出的列调度算法。实现见算法 2。

算法 2　Hestenes 算法 CUDA 伪代码

输入：A 为 m×n 矩阵，V 为单位矩阵

输出：A 奇异值分解：U、Σ、V(A=UΣV^T)

```
1. Dim3 Block(2m)
2. Dim3 Grid(n/2)
3. p<—BrentIndex[2 * Block. Idx]
4. q<—BrentIndex[2 * Block. Idx+1]
5. Load A[p], A[q] to Sharedmemory
6. { ParallelReduction—InnerProdect }
7. —syncthreads();
8. if(ThreadIdx==1)
9. { Caculate RotateAngle θ }
10. —syncthreads();
11. { Rotate the p,q columns of A and V}
```

在 CPU 端，则需要申请存储空间，通过 while 循环反复调用 GPU 的核函数，完成 Step1 和 Step2 的计算。在算法 2 中，Step1 对应第 5—9 行，Step2 为第 11 行。原算法执行时间为 $o(n^3)$，并行化后，理想情况下，GPU 执行核心足够多，不考虑同步和数据传输等代价，并行度为 $n/2$，则执行时间应该降至 $o(n^2)$。

算法执行每轮迭代时，CPU 作为辅助单元，进行数据准备、内存与设备存储（显存）间的数据同步等工作。而当 GPU 进行计算时，CPU 则处于闲置状态。如图 1 所示。

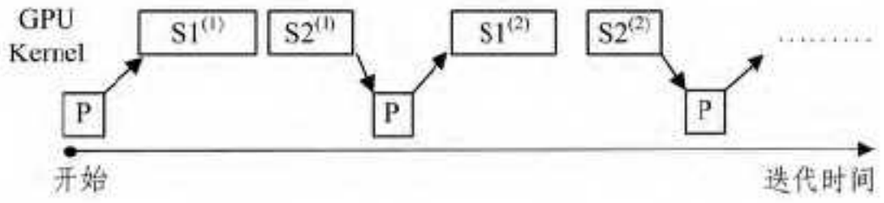

图 1　基于 GPU 的 Hestenes 算法执行过程

其中，上标 (i) 代表第 i 次迭代。S1，S2 分别代表对 p，q 列对进行的计算步骤 Step1 与 Step2。CPU 端则进行数据准备等辅助工作。可以看出，在迭代过程中 CPU 大部分时间处于闲置状态。然而由于 Step1 与 Step2 之间的数据依赖关系，两个步骤间必须串行执行，无法将 Step2 放在 CPU 上与 Step1 并行执行。

因此，引入 CPU-GPU 流水线机制，通过同时输入多个实例，在 CPU 与 GPU 间将 Step1 与 Step2 组织成软件流水线，理想情况下，通过流水线可以将 Step2 的计算过程完全隐藏，并隐藏引入流水线而带来的数据同步与传输的代价。执行时间的加速，则取决于 Step2 占整个算法的比重，以及 CPU 的计算能力。

3.2　CPU-GPU 协同计算加速奇异值分解

我们提出一种 CPU-GPU 软件流水线的方法，其基本思想是：通过同时执行多个输入实例，并将每个输入实例的相应计算任务分割为多个“线程流水段”，使得整个过程的计算通量加大，这个过程类似于指令流水线中的多个指令以及每个指令的多个执行阶段。其中，“线程流水段”可以是 CPU 上执行的计算线程、总线上的数据传输任务过程或 GPU 上执行的核函数线程。通过对原始的 GPU 应用引入这种方法，使得 GPU 在执行核函数时，CPU 也能够异步地执行计算任务，而总线上也同时可以异步地进行数据的拷贝工作。这使得计算平台的诸多计算资源能够并行地工作以获得更高的程序性能。

显然，引入软件流水线后，首先要满足串行算法 1 同样的顺序一致性。在应用中，采用信号量、设备端的异步调用机制、核函数的流机制共同保证流水线能够符合顺序一致性的执行。如图 2 所示，使原来应用中需要逐个地输入待处理矩阵的方式变成交叉地输入两个待处理的矩阵，从而在填满流水线的同时，规避了数据依赖问题。如图 2 所示，细箭头表示矩阵 A 与矩阵处理过程中的依赖关系，宽箭头表示矩阵 B 处理过程中的依赖关系。算法 3 给出引入这种机制的伪代码。

算法 3　CPU-GPU 协同加速 Hestenes 算法

输入：A，B 为 m×n 矩阵

Semaphore[] 为信号量数组

Stream[] 为 CUDA 的流，控制核函数操作数据的一致性。

```
1. Trans(A)
2. Trans(B)
3. Semaphore[]={1}
4. While(Iter——>0){
5.    if(Semaphore[0]==1)
6.       gpu-eaculateTheta<<<…Stream[0]>>>(A)
7.    if(Semphore[1]==1)
8.       gpu-eaculateTheta<<<…Stream[1]>>>(B)
9.    if(Semaphore[0]==0){
10.      cpu-thread(doRotate(A))
11.      synchData(A);}
12.   if(Semaphore[0]==0){
13.      cpu-thread(doRotate(B))
14.      synchData(B);}
15. }
```

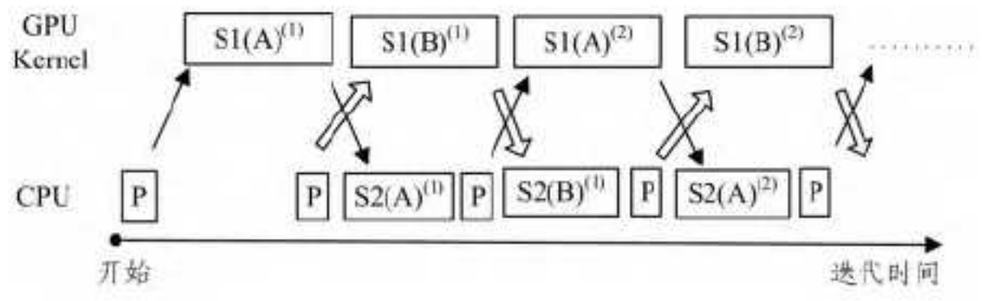

图 2　基于 GPU-CPU 流水线的 Hestenes 算法执行过程

在算法 3 中，Step1 由 GPU 的核函数执行实现，由函数 gpu-caculateTheta 完成。Step2 由 CPU 执行，由函数 doRotate 完成。信号量数组 Semaphore[] 在这里长度为 2，分别控制 GPU、CPU 的线程执行。在上面两个函数中，最后执行完都会对信号量数组进行更改，以允许下一个“流水段”的执行。而在 GPU 的执行上，采用 CUDA 的流 Stream[] 数组来保证不同数据实例间的区分。CPU 端和 GPU 端的线程启动都采用异步的形式调用。

对于任意 n 个矩阵，会分成 $n/2$ 个成对的矩阵，逐次执行。其原因如下：如果一个接一个地处理 n 个矩阵，那么由于每次迭代的 gpu-caculateTheta 步骤与 doRotate 步骤存在数据依赖（即旋转过程需要用到上一步计算的 theta 角度值），将导致 GPU 执行时 CPU 只能等待；同样，CPU 执行时 GPU 只能等待。而通过将 n 个矩阵分成 $n/2$ 组，每次同时处理两个矩阵，从而通过交错执行两个不同数据集矩阵，可以使得 GPU 与 CPU 间能以异步并行方式执行计算任务，即 gpu-caculateTheta 步骤与 doRotate 步骤在不同的矩阵数据集上同时执行。

应用中，一批矩阵的大小都在 512×512 个元素。一个矩阵大小为 1M。同步时，按照现在的 PCI-E3.0 的规范，速度在 16GB/s，传输所有矩阵的时间也在 1ms 左右，从下一节的实验分析中可以看到，传输耗时远远小于两个阶段的执行时间，可以忽略不计。

从实验分析一节可以看到，由于 Step1 的执行时间要略大于 Step2 的执行时间，因此在较好情况下，迭代次数较大时，可以完全隐藏 Step2 进行旋转操作的执行时间。同时，由于主存与设备显存间同步和传输矩阵数据的时间在 1ms 左右，其耗时也基本被隐藏在流水线中。

4 实验与分析

实验平台采用 Intel Core i5-3330 四核处理器和 Geforce GTX650 的图形处理器。GTX650 含 384 个 SP 计算单元，计算峰值为 812.54G/FLOPS。实验使用 CUDA 版本为 3.0。实验输入大小为 512×512 的一组矩阵。在 CPU Corei5-3330 上运行的是 Intel MKL(Math Kernel Library 11.0 Evaluation Version)[10] 中的 SGESVD 函数。

(1)耗时分析

选取 3 个矩阵作为输入，每个矩阵重复计算 10 次后取平均值。结果如表 1、表 2 所列。

表 1 串行 Hestenes 算法执行时间(ms)

输入实例	Step1	Step2	误差稳定时迭代次数
Input-Matrix-A1	1411.6	229.1	9
Input-Matrix-A2	1442.5	238.2	10
Input-Matrix-A3	1418.2	235.2	10

表 2 GPU 并行化后的执行时间(ms)

输入实例	Step1	Step2	Step2 占比
Input-Matrix-A1	238.3	73.6	23.5%
Input-Matrix-A2	244.6	82.2	25.1%
Input-Matrix-A3	242.2	74.9	23.6%

(2)加速比比较

引入 CPU-GPU 后，Step1(计算旋转角的步骤)在 GPU 上执行，而 Step2(对矩阵 A 和 V 进行旋转的步骤)放到 CPU 上执行，此时 Step1 在 GPU 上的执行时间仍然足够掩盖 Step2 在 CPU 上的执行时间。表 3 列出了输入实例个数在 $n=1, n=2, n=4$ 时的计算结果。

表 3 CPU-GPU 协同计算比较(ms)

输入矩阵个数	GPU 并行 SVD 耗时	引入 CPU-GPU 耗时	数据传输开销	GPU 并行与串行加速比	CPU-GPU 协同与串行加速比
1	312.2	467	1.6	5.26	3.51
2	665.8	487.2	3.5	4.92	6.72
4	1349.6	977.6	6.7	4.89	6.76
6	1789.3	1432	11.3	5.32	6.64

图 3 给出了奇异值分解在 CPU 上的执行时间、在 GPU 上的执行时间以及引入 CPU-GPU 软件流水线后的执行时间。当同时参与运算的矩阵数量大于 2 时，其相对于原来的 GPU 并行版本能够达到 20% 以上的加速比，也基本符合预期。但当只有一个矩阵参与运算时($n=1$)，无法显示出引入 CPU-GPU 软件流水线的优势，相反由于 Step2 交由 CPU 执行，导致执行时间更长。因此当只有一个矩阵参与运算时，应避免采用这种方法。

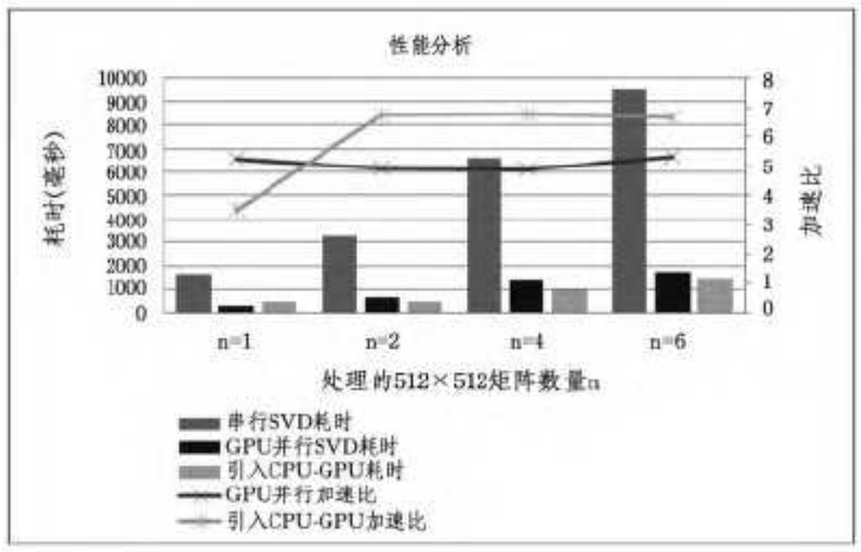

图 3 引入 CPU-GPU 流水线前后的加速比

在执行过程中，如图 2 所示，GPU 的执行时间足够长，可以将 CPU 上执行的矩阵旋转操作、CPU 与 GPU 间的数据传输操作完全隐藏。因而得到了更加理想的加速比，如图 3 所示。另外注意到，CPU 与 GPU 间的数据传输操作远小于 GPU 的耗时，一定程度上可以忽略。当输入 n 个矩阵时，将其分成 $n/2$ 组成对矩阵进行处理，每次迭代同时进行的矩阵数目为 2，因此，当 $n>2$ 以后，每个矩阵的平均加速比保持稳定。

结束语 本文提出了采用 CPU-GPU 协同计算加速奇异值分解的方法，该方法给出了一种通过流水线隐藏算法部分计算过程的方法。相对于单纯的 GPU 并行化 Hestenes 算法，获得了 20% 的性能提升，并相对于传统算法获得了 6.8 倍的加速比。

参 考 文 献

[1] Golub G, Van Loan C. Matrix Computations. Johns Hopkins University Press(2nd edition)[M]. Baltimore, MD, 1989

[2] Gu Ming, Demmel J, Dhillon I. Eficient Computation of the Singular Value Decomposition with Applications to Least Squares Problems[R]. Technical Report CS-94-257. Department of Computer Science, University of Tennessee, October 1994

[3] Hestenes M R. Inversion of matrices by biorthogonalization and related results[J]. JSoc. Indust. Appl., 1958, 6(1): 51-90

[4] Kotas C, Barhen J. Singular value decomposition utilizing parallel algorithms on graphical processors[C] // OCEANS 2011. Publication Year, 2011: 1-7

[5] Lahabar S, Narayanan P J. Singular Value Decomposition on GPU using CUDA[C]//IEEE International Parallel Distributed Processing Symposium. Hyderabad, 2009: 1-10

[6] Novakovi V, Singer S. A GPU-based hyperbolic SVD algorithm [M]. BIT 51, 2011: 1009-1030

[7] Liu Ding, et al. A divide-and-conquer approach for solving singular value decomposition on a heterogeneous system[C] // Proceedings of the ACM International Conference on Computing Frontiers. ACM, 2013

[8] Brent R P, Luk F. T. The solution of Singular Value and Symmetric Eigen problems on Multiprocessor Arrays[J]. Sct Stat Comput, 1985, 6: 69-84

[9] 张舒，窦衡. 基于 CUDA 的矩阵奇异值分解[J]. 计算机应用研究，2007，24(6)

[10] http://software.intel.com/en-us/intel-mkl/