轻工学报® 2019 年 3 月 第 34 卷 第 2 期
JOURNAL OF LIGHT INDUSTRY Vol. 34 No. 2 Mar. 2019

引用格式:刘慧慧,闻萌莎,钱慎一,等. ARL 中 Clean 算法的并行化研究[J]. 轻工学报,2019,34(2):88 - 94.
中图分类号:TP301 **文献标识码**:A
DOI:10.3969/j.issn.2096 - 1553.2019.02.012
文章编号:2096 - 1553(2019)02 - 0088 - 07

ARL 中 Clean 算法的并行化研究

Research on parallelization of Clean algorithm in ARL

刘慧慧[1],闻萌莎[2],钱慎一[1],吴怀广[1], 张伟伟[1], 李代祎[1]
LIU Huihui[1],WEN Mengsha[2],QIAN Shenyi[1],WU Huaiguang[1],ZHANG Weiwei[1],
LI Daiyi[1]

1. 郑州轻工业大学 计算机与通信工程学院,河南 郑州 450001;
2. 华东师范大学 计算机科学与软件工程学院,上海 200241
1. *College of Computer and Communication Engineering*, *Zhengzhou University of Light Industry*, *Zhengzhou* 450001, *China*;
2. *College of Computer Science and Software Engiweering*, *East China Normal University*, *Shanghai* 200241, *China*

关键词:
ARL;去卷积算法;
CUDA;并行计算;
Clean 算法

Key words:
ARL; deconvolution algorithm; CUDA; parallel computing; Clean algorithm

摘要:针对 SKA 算法参考库 ARL 中的去卷积算法运行效率低、无法满足海量数据实时处理的问题,提出了 CPU 和 GPU 协同工作模式下的并行化 Clean 算法. 该方法将 Clean 算法中可以并行计算的步骤利用多线程在 GPU 上并行执行,将无法并行计算的步骤在 CPU 上串行执行. 验证实验结果表明,在数据逐渐增大的过程中,并行化 Clean 算法比在 CPU 上的串行处理运行时间显著减少,当图达到 4096 像素 ×4096 像素时,可以有 10 倍的提速. 这说明并行化 Clean 算法在处理海量数据时,能够显著提高运算效率.

收稿日期:2018 - 12 - 13
基金项目:国家重点研发计划政府间科技合作项目(2016YFE0100600;2016YFE0100300)
作者简介:刘慧慧(1994—),女,河南省郑州市人,郑州轻工业大学硕士研究生,主要研究方向为算法优化与算法并行化.
通信作者:钱慎一(1975—),男,江苏省扬州市人,郑州轻工业大学副教授,硕士生导师,主要研究方向为数据库、数据挖掘、信息集成.

Abstract: The deconvolution algorithm in the ARL of the SKA algorithm reference library is inefficient and cannot meet the needs of real-time processing of massive data. The parallelized Clean algorithm in the cooperative working mode of CPU and GPU was proposed. The steps of parallel computing in Clean algorithm were executed in parallel on GPU using multi-threads, and the steps in the Clean algorithm that couldn't be parallelized were executed serially on the CPU. The results showed that the running time of parallel Clean algorithm under CPU and GPU cooperative mode was significantly shorter than that under CPU. When the image size reached 4096 × 4096, the parallel Clean algorithm GPU cooperative mode could be speeded up by 10 times, which showed that the parallel Clean algorithm could significantly improve the efficiency of operation when dealing with massive data.

0 引言

射电望远镜是观测和研究来自天体的射电波的基本设备,但天线的数量有限,从而导致空间频率覆盖不完整[1],影响最终图像的构建. 平方公里阵列 SKA(square kilometre array)[2]是国际上建造的最大综合孔径射电望远镜. 与现有射电望远镜相比,SKA 的灵敏度提高 10 ~ 100 倍,测量速度提高 10^5 倍[3]. ARL(algorithm reference library)算法参考库是 SKA 的候选算法库,其中去卷积算法[4]是 ARL 的一个重要组成,在射电天文成像中起着至关重要的作用.

去卷积算法可以消除图像不完整、图像模糊对观测结果的影响,引发很多学者的关注和研究. 在 ARL 中,去卷积算法采用的是 Clean 算法. 1974 年,J. A. Högbom [5]首次提出了 Clean 算法,一种非线性去卷积的迭代方法,用以消除旁瓣干扰. 该算法虽然消除了旁瓣的干扰,但需要消耗大量的时间,运算效率较低. 2004 年,S. Bhatnagar 等[6]提出了一种用于无线电干涉图像的尺度敏感的反卷积(Asp-Clean)算法,将图像建模为自适应尺度像素的集合,可以更准确地重建非对称结构的天空图像,但是该算法增加了算法复杂性和计算成本,其计算时间是 Clean 算法的 3 倍多. 2008 年,T. J. Cornwell[7]提出了一种 Multiscale Clean 算法,该算法可以更好地处理扩展源,提高图像质量,但是比 Clean 算法需要的运行时间更长. 2011 年,U. Rau等[8]在 Multiscale Clean 算法的基础上,结合 Multi-frequency 算法,在更高的灵敏度和采样频率的情况下,重建了天空图像,但此算法仍存在运算时间较长、运行效率较低的问题. 2016 年,L. Zhang 等[9]提出自适应环路增益自适应规模 Clean(Algas-Clean)算法,该算法可以提供更准确的模型,具有更强的收敛性,比 Asp-Clean 算法的运算速度快了 50%. 2017 年,J. Cheng 等[10]为解决传统 Clean 算法出现的问题,提出了一种小波 Clean 算法,该算法对小波滤波器的参数进行了优化,提高了图像质量,但没有缩短运行时间,未提高算法运行效率. 目前,大部分学者都是针对提高图像质量进行研究,在运算效率方面研究成果很少. 然而,随着 SKA 项目的实施,海量的天文数据迅速增加,如何有效、快速地处理数据成为当今计算机领域研究的重点,这使去卷积算法运算效率的提高变得刻不容缓.

鉴于此,本文拟通过对 Clean 算法进行耗时分析,利用多线程来设计算法的 CUDA 核函数,提出一种基于 CPU 和 GPU 协同工作模式的并行化 Clean 算法,以期提高海量天文数据下的去卷积算法的运算效率.

1 Clean 算法在 ARL 中的应用

Clean 算法[11]是一种非线性去卷积方法,专门用于处理不完整的射电数据,将其运用于 ARL 中,可以消除图像中旁瓣(天线方向图通

常都有两个或多个瓣,其中辐射强度最大的瓣称为主瓣,其余的瓣称为旁瓣)的影响.

若天空亮度分布用 $I(x,y)$ 表示,其中 x,y 分别表示点源的横纵坐标,则与之对应的复可见度函数用 $V(u,v)$ 表示,其中 u,v 称为空间频率. 在 uv 平面(基线投影所在的平面)中,u 指向东,v 指向北,其基线如图 1 所示.

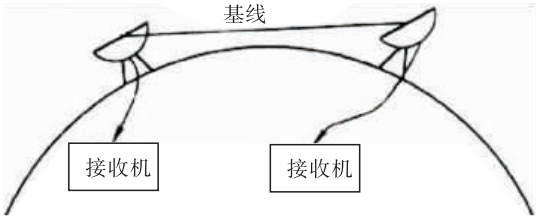


图 1　uv 平面基线示意图

Fig. 1　Baseline diagram of uv plane

由于 SKA 中天线数目是固定的,所以 uv 采样点的数目有限. 将 uv 采样点的分布函数称为采样函数,对采样函数进行傅里叶逆变换可得到脏束,将射电望远镜经过预处理后的数据进行傅里叶逆变换可得到脏图,通过一个迭代过程找到脏图中的最大亮度点及其位置,将上述结果与拟合光束(高斯光束)进行卷积即得到恢复的图像.

在 uv 平面中,误差项 r[12] 可表示为

$$r = \sum_{k=1}^{N} w_k \left| V_k - \hat{V}_k \right|^2 \qquad ①$$

其中,N 是采样点的数量,w_k 是第 k 个采样点的权重,V_k 是可见度函数,$\hat{V}_k$ 是图像平面中具有坐标(x,y) 的点源网格亮度模型的傅里叶变换,可表示为

$$\hat{V}_k = \sum_{i=1}^{M} I_i e^{j2\pi(v_k x_i + v_k y_i)}$$

其中,I_i 是第 i 个图像点的位置,M 是图像点的数量. 则式 ① 在图像平面中可表示为

$$r = \sum_{i=1}^{M}\sum_{p=1}^{M} B_{ip} I_i I_p - 2\sum_{i=1}^{M} D_i I_i + \sum_{k=1}^{N} w_k \left| V_k \right|^2 \qquad ②$$

其中,$D_i = \dfrac{\sum_{k=1}^{N} w_k \mathrm{Re}\left(V_k e^{j2\pi(v_k x_i + v_k y_i)} \right)}{\sum_{k=1}^{M} w_k}$,称为脏图;

$$B_{ip} = \frac{\sum_{k=1}^{N} w_k \cos(2\pi(v_k(x_i - x_p) + v_k(y_i - y_p)))}{\sum_{i=1}^{N} w_k},$$

称为脏束.

然后,最小化 r,并取式 ② 的导数,可以得到卷积方程

$$D_i = \sum_{p=1}^{M} B_{ip} I_p \qquad ③$$

式③说明了脏图是真实图像和脏束的卷积. 具有旁瓣的脏束称为点扩散函数 PSF(point spread function). Clean 算法可以从已知的脏图和 PSF 中得到图像点 I_p 的一系列位置,其主要计算步骤如下:

1)对采样函数进行傅里叶逆变换,得到脏束;

2)将射电望远镜经过预处理的数据进行傅里叶逆变换得到脏图,找到脏图中的最大值点,并记录该坐标;

3)将脏束的中心点移到这个最大值点的位置上,且使脏束乘以因子 γ;

4)从脏图中减去该脏束;

5)重复步骤 2)—4),直到剩余图像的最大值小于给定的噪声水平;

6)对一拟合光束和 δ 函数(被减去的最大值点的总和及其位置称为 δ 函数图)进行卷积;

7)将步骤 6)的结果加上剩余图像,得到最终的清晰图像.

2　Clean 算法的 GPU 并行化实现

ARL 中的 Clean 算法是基于 CPU 的,运行时间较长,本文根据 GPU 并行的特点,利用多线程,设计并实现基于 GPU 并行化的 Clean 算法.

2.1 Clean 算法的并行化分析

研究表明,去卷积算法在数据处理过程中迭代计算耗时较长,对计算机内存和 CPU 要求较高. GPU[13] 是具有数千个计算核心的大规模并行架构,可以在有限时间内执行多次浮点运算,因此在 GPU 上并行实现去卷积算法可以大大提升运算速度,节省时间. NVIDIA 公司提出了一种并行计算架构 CUDA(computer unified device architecture)[14],通过 CUDA C 编程在 GPU 上实现并行计算,其运算速度比 CPU 提高了几倍甚至上百倍[15].

通过对 ARL 中 deconvolution 模块的分析可知,在 Clean 算法的实现过程中,需要进行多次迭代计算,而每次迭代都需要使用上一次迭代的结果,因此无法将 Clean 算法的整个过程写成内核函数,但可以将其中的部分操作写成内核函数,使其可以在 GPU 上进行并行计算,以此来提高计算效率,而将无法进行并行计算的操作在 CPU 上串行执行.

并行处理主要分为任务并行处理和数据并行处理. 基于 CUDA 的组织结构,结合 Clean 的算法分析,本文将采用数据并行处理方式. 并行算法的执行过程从 CPU 开始,当内核函数被调用时,执行过程转移到 GPU 设备上. 另外,GPU 的内存分为全局存储器、常量存储器、共享存储器和寄存器[16]. 在整个程序中,数据需要被 Grid 网格中的所有线程访问,因此需要将数据放置在全局存储器中;将在脏图中找到的最大值放入共享存储器,方便存取;每一个点源数据都需要相同的高斯核,将高斯核放置在常量存储器中,以此来加强程序的快速访问.

2.2 Clean 算法的并行化实现

Clean 算法的并行化实现步骤如下.

1)初始化洁图,并将脏束和脏图作为形参传入.

2)在 GPU 上调用 CUDA 编写的 dirtyFabsMax 核函数,找到脏图中的最大值点,并记录该点的位置坐标.

3)将脏束的中心点移到这个最大值点的位置上,且使脏束乘以因子 γ.

4)在 GPU 上调用 CUDA 编写的 subPsf 核函数,从脏图中减去该脏束.

5)判断剩余图像的最大值是否小于给定的噪声水平,若未满足条件,则继续进行迭代计算;若满足条件,则退出迭代,继续向下执行.

6)对一拟合光束和 δ 函数进行卷积.

7)将步骤 6 得到的结果加上剩余图像,得到最终的清晰图像.

8)把 GPU 上经过处理得到的数据传给 CPU 并进行图像绘制.

Clean 算法的步骤 2 主要是为了找到脏图的最大值点及其位置,该操作可以采用并行方式进行处理. 初始化最大值点,将图像中的每个点与该点进行比较,若图像中的点大于该点的值,则把图像中的点赋值给最大值点. 因为找到脏图中最大值点及其位置的操作都是比较操作,可以同时进行,所以能够并行计算. 由于 ARL 中的 Clean 算法是采用 Python 语言实现的,因此本文采用 PyCUDA 来实现 Clean 算法的并行化. 具体实现方式是内核函数采用 CUDA C 编程,其余部分仍采用 Python 编程,通过 PyCUDA 使 Python 程序可以访问 Nivida 的 CUDA 并行计算 API,从而达到 GPU 并行的目的. dirtyFabsMax 核函数将 Blocksize 和 Gridsize 作为相关参数传入,指定网格和线程块的大小,通过计算 Blocksize 和 Gridsize 的乘积得出网格中线程的数量,进而指定并行处理多线程运算 GPU 上创建线程的个数. 另外,需要 syncthreads 函数来保证线程同步,否则由于线程执行具有无序性和异步性,可能会出现元素覆盖的问题,导致运行结果不正确. dirtyFabsMax 核函数的伪代码为

Input dirty

Initialize row ← threadIdx. x + blockIdx. x * blockDim. x

col ← threadIdx. y + blockIdx. y * blockDim. y

t ← row + col * blockDim. x * gridDim. x

For stride ← blockDim. x * blockDim. y * gridDim. x * gridDim. y, stride > 1, stride←stride / 2 do__syncthreads( )

If t < stride Then

If psf[t] < psf[t + stride]

Then temp ← psf[t]

psf[t] ← psf[t + stride]

psf[t + stride] ← temp

End If

End If

End For

Clean 算法中的步骤 4 是从脏图中减去脏束,此操作可以进行并行处理. 因为脏图和脏束上的每个点是相互独立的,可以同时进行减法操作,而 GPU 可以为每一个元素分配一个线程,因此每一个元素的相减操作就可以并行执行,提高了 Clean 算法的计算效率. 编写 subPsf 核函数用来实现从脏图中减去脏束的操作. 因为实验所用的 GPU 每个线程块最多支持 1024 个线程,而处理大小为 1024 像素 × 1024 像素的图像则需要 1024 × 1024 个线程,所以需要在一个网格中划分多个线程块来得到 1024 × 1024 个线程,函数被调用时的执行配置为 < < < 1024,1024 > > > ,这样就可以满足 GPU 为每一个数据元素分配一个线程. subPsf 核函数的伪代码如下:

Input res, psf, a1o, a2o, mval

Initialize row ← threadIdx. x + blockIdx. x * blockDim. x

col ← threadIdx. y + blockIdx. y * blockDim. y

If row > = a2o[0] AND row < a2o[1] AND col > = a2o[2] AND col < a2o[3]

Then t1 ← (row + a2o[0]) * blockDim. x * gridDim. x + col - a2o[2]

t2 ← row * blockDim. x * gridDim. x + col

res[t1] ← res[t1] - psf[t2] * mval

End If

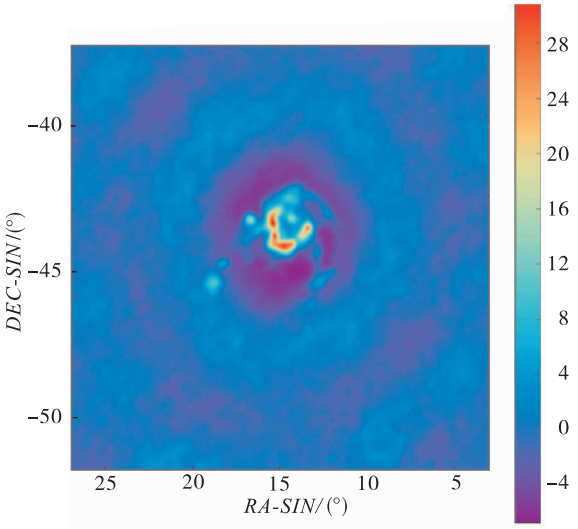
3　验证实验结果与分析

验证实验所用平台是高性能计算平台,配备有型号为 NVIDIA Tesla K80 的 GPU,内存共 128 GB,存储器为 512 GB 的 SSD 硬盘和 1 TB 的 SAS 高速硬盘,操作系统为 CentOS7,使用 PyCUDA 作为开发语言. 实验的测试数据来源于 ARL 提供的天文测试数据.

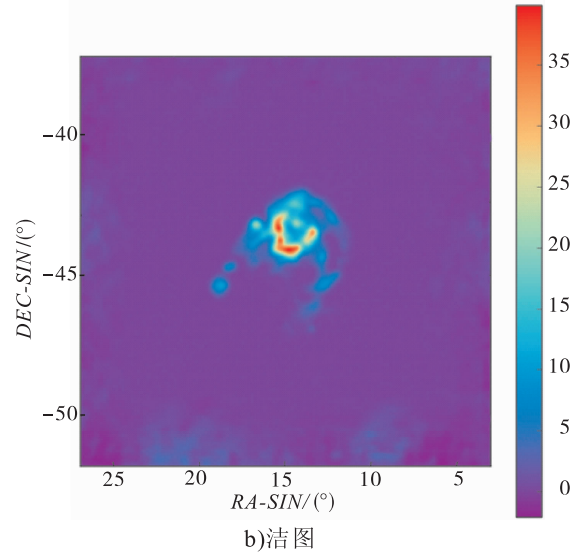
首先将 ARL 中的数据进行预处理,生成脏图和脏束,然后调用并行化后的 Clean 算法对脏图进行处理,生成洁图,结果如图 2 所示. 从图 2 可以看出,脏图中大量的模糊点被清除,洁图更清晰地反映出天空亮度的分布情况.

将 ARL 中的 Clean 算法用 PyCUDA 语言在 CPU 和 GPU 协同工作条件下实现,用 Python 语言在 CPU 条件下实现,测试迭代次数均为10^4次,两个实现过程的计算时间如表 1 所示,加速比是 Clean 算法在 CPU 下的运行时间与在 GPU 下运行时间的比值.

从表 1 可以看出,在 CPU 与 GPU 协同作用下,用 PyCUDA 实现的 Clean 算法对数据的处理速度更快,用时远远小于 CPU 下用 Python 实现 Clean 算法的处理数据时间. 当图像大小为 512 像素 × 512 像素时,数据量较小,只有 26 × 10^4 条左右,加速的效果不是很明显,只有两倍



a)脏图



b)洁图

图 2　实验测试数据的脏图与洁图

Fig. 2　Dirty image and Clean image of experimental data

表 1　Clean 算法计算时间的比较

Table1　Comparison of Clean algorithm execution time

图像大小/(像素 × 像素)	计算时间/s		加速比
	Python	PyCUDA	
512 × 512	8.875	3.666	2.42
1024 × 1024	27.289	5.263	5.19
2048 × 2048	95.240	12.369	7.69
4096 × 4096	457.186	40.712	11.23
8192 × 8192	1 803.011	170.817	10.56

左右的加速效果. 但是当图像大小增大后,特别是大于 4096 像素 × 4096 像素后,此时的数据量增加到 10^7 条以上,加速比急剧增大,加速效果有了明显提升. 当图像大小为 8192 像素 × 8192 像素时,Clean 算法在 CPU 上需要运行 30 min才能得出结果,但本文实现的并行算法只需要 2 min 左右就可以得到结果,可以达到 10 倍的提速,极大地节省了时间,提高了算法的运行效率. 从并行化实现的 Clean 算法在不同图像大小下的加速比可以明显看出,当图像大小增大时,加速比增大,在 GPU 上运行的加速效果非常显著.

4　结论

针对 SKA 算法参考库 ARL 中去卷积算法处理海量数据负担过重,时间太长的问题,对 ARL 中 Clean 算法进行并行化研究,提出了 CPU 和 GPU 协同工作模式下的并行化 Clean 算法. 将 Clean 算法中耗时较长的两个部分,即在脏图中找到最大值点和从脏图减去脏束,采用多线程并行处理,并使用 syncthreads 函数来保证线程同步,无法并行处理的步骤在 CPU 上串行执行,从而完成对海量数据的处理. 验证实验结果表明,处理同样的天文数据时,本文基于 GPU 并行化的 Clean 算法与 CPU 上的串行 Clean 算法相比,可以达到 10 倍的提速.

本文使用的是单 GPU 与 CPU 协同工作,没有实现算法的最大并行化,因此,今后的工作是将单 GPU 上的算法移植到多 GPU 上,以便进一步优化 ARL 中去卷积算法的并行化.

参考文献:

[1] DABBECH A,FERRARI C,MARY D,et al. Moresane: model reconstruction by synthesis-analysis estimators-a sparse deconvolution algorithm for radio interferometric imaging[J]. Astronomy & Astrophysics,2015,576:7.

[2] BROEKEMA P C, VAN NIEUWPOORT R V, BAL H E. The square kilometre array science data processor: Preliminary compute platform design[J]. Journal of Instrumentation, 2015, 10(7):14.

[3] VAN HEERDEN E, KARASTERGIOU A, ROBERTS S J, et al. New approaches for the real-time detection of binary pulsars with the Square Kilometre Array (SKA)[C]// General Assembly and Scientific Symposium. Piscataway: IEEE, 2014:1 - 4.

[4] LA CAMERA A, SCHREIBER L, DIOLAITI E, et al. A method for space-variant deblurring with application to adaptive optics imaging in astronomy[J]. Astronomy & Astrophysics, 2015, 579:1.

[5] HÖGBOM J A. Aperture synthesis with a non-regular distribution of interferometer baselines [J]. Astronomy and Astrophysics Supplement Series, 1974, 15:417.

[6] BHATNAGAR S, CORNWELL T J. Scale sensitive deconvolution of interferometric images-I: adaptive scale pixel (Asp) decomposition[J]. Astronomy & Astrophysics, 2004, 426(2):747.

[7] CORNWELL T J. Multiscale CLEAN deconvo-lution of radio synthesis images[J]. IEEE Journal of Selected Topics in Signal Processing, 2008, 2(5):793.

[8] RAU U, CORNWELL T J. A multi-scale multi-frequency deconvolution algorithm for synthesis imaging in radio interferometry[J]. Astronomy & Astrophysics, 2011, 532:71.

[9] ZHANG L, ZHANG M, LIU X. The adaptive-loop-gain adaptive-scale Clean deconvolution of radio interferometric images[J]. Astrophysics & Space Science, 2016, 361(5):153.

[10] CHENG J, XU L, LU Z, et al. Application of wavelet clean for Mingantu Spectral Radio-heliograph imaging[C]//2017 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS). Piscataway: IEEE, 2017:81.

[11] BOSE R. Lean Clean: deconvolution algorithm for radar imaging of contiguous targets[J]. IEEE Transactions on Aerospace and Electronic Systems, 2011, 47(3):2190

[12] CHEN L, LI L M, WAN G C, et al. A modified Clean algorithm for improving aperture synthesis observations of radio astronomy[C]// Progress in Electromagnetic Research Symposium. Piscataway: IEEE, 2016:144.

[13] HUANG X, WANG K, HUANG L, et al. GPU implementation for lo-regularized blind motion deblurring[C]// IEEE International Conference on Progress in Informatics and Computing. Piscataway: IEEE, 2016:597 - 601.

[14] SHERRY M, SHEARER A. IMPAIR: massively parallel deconvolution on the GPU[C]// Image Processing: Algorithms and Systems XI. [S. l.]: [s. n.], 2013, 8655(1):84 - 94.

[15] SANDERS J, KANDROT E. CUDA by example [M]. Boston: Addison-Wesley Professional, 2010:18 - 25.

[16] Nvidia. CUDA C best practices guide[M]. American: Nvidia, 2012.