硕士学位论文

基于云的分布式图像修复系统的研究与实现

院　　系：信息科学与工程学院
专　　业：微电子学与固体电子学
姓　　名：屠 昕
学　　号：13210720067
指导教师：陈更生 高级工程师
完成日期：2016 年 3 月 25 日

指导小组成员名单

陈更生　　高级工程师

目录

摘 要

如今随着物联网的快速发展，人们对实时处理视频的需求越发迫切，随之出现了众多基于实时处理视频的应用，这些应用能给用户提供多样化的视频处理服务。但是随着处理数据的增长，传统的单节点视频处理平台由于计算能力有限，已经很难应付大规模的数据处理。而大规模图像处理尤其是视频图像处理技术，对运算能力有着严格的要求，目前一些基于 Hadoop 的处理平台只能处理离线视频文件，无法满足实时处理视频流的需求。另一方面图像修复作为图像处理技术中重要的组成部分，在人们的生活中有着广泛的应用，但是目前的图像修复算法仍存在修复效果不理想和修复耗时过长等缺点。

针对上述情况，为了能够在实时处理大规模视频图像的同时对视频图像中的破损进行修复，本文实现了一套云计算处理平台，然后针对图像修复算法的一些不足之处进行了改进，并将其做了并行化改进。本文主要工作有：

1. 通过分布式消息发布订阅系统 Kafka、分布式流处理系统 Storm 和图形处理器 GPU 搭建了一套能够同时处理实时视频流和离线图像文件的云计算处理平台。其中 Kafka 用于图像数据的获取，Storm 用于处理图像数据的传递，GPU 则负责处理图像处理算法中的并行化计算部分。
2. 实现了一种改进的快速图像修复算法，用以提高传统算法的修复质量和修复速度。
3. 为了进一步减少修复耗时，使用了 GPU 对改进的快速图像修复算法进行了并行化改进。最后实现了一个基于云平台的图像修复应用，并在其中进行了相应的测试。

实验结果表明本文实现的云计算系统能够通过分布式的方式有效地部署图像处理算法并处理视频流和图像文件等数据，并且充分利用 GPU 的并行化方式来提高图像处理的速度。此外本文实现的改进图像修复算法在修复破损图像时效果更好，速度 1

关键字：云计算；Storm；Kafka；图像修复；分布式计算；分水岭分割；分类比较；GPU；并行计算
中图分类号：TP

Abstract

With the development of internet of things, the need for realtime video processing by public is becoming more and more urgent. Along with those needs, many applications focused on realtime video processing have appeared, these applications offer the users various video processing services. But along with the grow of the data needed to be processed, traditional method based on single processing node can hardly satisfy the demand for processing large amout of data for the limit of computing ability. And the massive image processing technology especially for video had strict requirement for computation abality. But those image processing platforms based on hadoop can only handle offline data and can't satisfy those need for realtime processing. On the other hand, as an important part of image processing, inpainting has been widely used in daily life. But there still exist a lot of drawbacks of inpainting such as poor inpaint quality and long time for executing.

According to the situation mentioned above, this paper implement a cloud computing platform for realtime processing of large sacle video processing and inpainting for the damage in those videos. And then this paper proposed a modified algorithm to improve the quality and speed of existing inpainting methods. The main work of this paper includes:

1. Build a cloud computing system for processing realtime video streams and offline image files based on Kafka, Storm and GPU. Kafka is responsible for retrieving and sending messages, Storm is responsible for processing and routing messages and GPU is responsible for the parallel part of image processing algorithm.

2. Implement an improved fast inpainting method to modify the existing method's inpainting quality and speed.

3. Use GPU to improve the inpainting speed in parallel. Implement an application for inpaitng based on the cloud computing platform and verify the experimental results.

The experimental results show that the cloud computing system can efficiently deploy image processing algorithms and process image data through distributed medthod and use GPU to improve the process speed in parallel. Besides the inpainting method this paper proposed has better inpainting quality and faster inpainting speed.

Keywords: cloud computing; Storm; Kafka; distributed computing; inpainting; watershed segmentation; classification matches; GPU; parallel computing

Classification Code: TP

第一章 绪论

1.1 研究背景和研究意义

云计算作为由分布式计算、负载均衡、存储系统等多种传统计算机技术和近年来新兴互联网技术结合的产物，已经被学术界和产业界广泛应用。云计算平台的底层是大量计算和存储资源，这些分布式资源被通过互联网的方式整合在一起，然后提供给用户。用户则根据自己的需求去申请资源并按自己的需求进行运算。云计算平台则会根据用户申请资源的实际运算情况来实时的管理底层资源。因此云计算平台能够解决用户无法通过个人电脑的计算能力解决的复杂问题，而且不需要用户亲自去搭建一套专门的处理系统。目前，包括FaceBook、Twitter、 Yahoo在内的许多知名互联网公司都搭建了自己的云计算平台系统[1]。

另一方面，随着物联网与云计算的快速发展，当前对于数据计算实时性的需求越来越高，其中一个重要的需求是对海量视频的实时性处理。目前许多基于Hadoop集群的视频处理系统只能处理离线数据，因为这种方式需要通过Map/Reduce对数据进行收集备份，而且在计算的过程中需要对HDFS进行读写操作，最终使实时性的需求很难实现[2]。

Storm作为如今最常用的分布式计算系统之一，能够对大规模流式数据进行高容错的实时计算，弥补了Hadoop只能以批处理的方式对离线数据进行计算的缺点，使得流数据的处理变得更加容易。因此，Storm能够满足为海量视频数据处理算法提供实时分布式处理平台的需求[3]。

尽管分布式流处理系统在计算延迟上有着天生的优势，但在处理大规模图像时，由于图像处理算法普遍需要极大的运算能力，因此仅仅依靠 CPU 的性能很难做到实时处理。即使是由多台服务器组成的集群，其运算能力也很难使得诸如视频图像修复这类极为耗时的算法达到秒级延时的要求。所以当 CPU 的计算能力已经无法满足计算需求时，GPU 作为专门为并行化计算开发出来的处理器，在处理并行计算问题的能力上已经赶超了 CPU[4]。而且 GPU 与 CPU 相类似，当依靠一个 GPU 无法满足计算能力的需求时，可以通过集群化来使得 GPU 集群拥有超越 CPU 集群的并行计算能力。因此，很多部署在 CPU 平台上的图像处理算法无法实现高清图像的实时处理，使得处理结果严重滞后于应用的需求。而采用 GPU 则可以相对 CPU 提高数十倍的性能，快速实现高清图像的实时处理，给图像处理带来了新的方法。

此外，图像作为获取和记录信息的主要途径，包含了许多重要的科研、经济和人文价值信息。数字图像处理技术作为信息技术中的重要组成部分，已经在人们生活的各方面得到了许多广泛的应用。图像修复即在保证图像的质量和自然效

果不受破坏的前提下，利用图像中未损坏的信息对图像中遗失信息进行修补或者对图像中特定的信息进行移除的图像处理。作为数字图像处理中的一个重要领域，已经被广泛应用于视频及图像资料编辑和 VR 技术等多个领域[5]。随着数字图像处理技术的发展，图像修复技术正被广泛的应用于人们的生活中。

传统意义上的图像修复主要针对绘画和相片等实物进行修复，通过手工的方式来更改图像中的画面，修复过程极其复杂。在进入信息化时代后，数字相片也逐渐替代了传统的胶卷，PS、美图秀秀等修图软件也越来越被人们所熟知和使用。如何能够自动对图像进行修复，成了很多人非常感兴趣的一个问题。

除了去除一些令人不满意的部分外，图像修复还可以扩展到从图像中移除特定物体，并用背景进行填补，使得最终效果达到让人感觉自然的成度。比较常见的数字图像修复场景主要有对数字化后的破损图像资料等进行修复；移走图像中的目标物；去除图像中的文字、划痕等。除此之外，在图像轮廓修复、气象云图处理、物联网等许多领域都有广泛的应用。

1.2 研究现状

近几年来，云计算产业的发展越加成熟，小到个人用户和创业公司，大到跨国企业和各大院校都对其极为重视。云计算最初起源于信息的存储服务，到目前为止已经发展成了一种通过对网络中计算和存储资源进行虚拟化处理并提供基于互联网应用服务的过程。通过这种方式，用户无需考虑提供云计算的平台如何搭建，只需专注于根据服务提供商提供的开放接口来实现自己基于互联网的应用服务即可。到目前为止，业界对云计算的定义可以归为三类：基础设施即服务（IaaS）提供给用户虚拟化的存储、处理、计算服务等基础设施和框架，比较具有代表性的平台有 Amazon 的 EC2 和 Nimbus；平台即服务（PaaS）则提供给用户在云环境下的程序应用开发平台，具有代表性的有 Google 的 AppEngine 和微软的 Azure；软件即服务（SaaS）则向用户提供了随时随地的应用程序调用，代表性的平台有 CRM 和 Mahout 等。而作为产业界和学术界最常用的一种基础云计算平台，Hadoop 整合了 HDFS 存储系统和 MapReduce 计算模型，使其同时具备了强大的处理功能、方便的使用方式、廉价的成本等优点，因此很快成为业界的标杆。后来在 Hadoop 的基础上陆续出现了许多类似的分布式计算系统，比如在 Hadoop 的基础上进行了一些架构改良并实现了基于内存计算的 Spark 和提供了实时运算特性从而可以实时处理大数据流的 Storm。因此在实现类似基于视频流的图像处理等实时性要求较高的应用时，Spark 和 Storm 会更加符合用户的需求。

图像修复技术主要分为两类。一类是基于 Bertalmio 等人[6]提出的基于偏微

分方程的方法，通过对破损区域的边界进行不同方向的扩散来对图像进行修复。在其基础上，Chan 等人[7]提出基于全变分的修复方法，改进了对有噪声图像的修复。该类算法的算法缺点是运算量大，不适合修复大面积的破损。

另一类方法是最早由 Efros A[8]提出的基于纹理合成的图像修复方法，通过对纹理的适当采样来对破损区域进行填补，适合大区域的图像修补。纹理合成的方法容易导致修复后的图像结构信息出现断裂。

Criminisi 等人[9]在纹理合成的基础之上结合了边界扩散算法的优点提出了基于样本块的图像修复算法，根据破损区域的边界信息确定修复优先级，在图像未破损的已知区域（源区域）中寻找与待修复区域的纹理信息最为接近的样本块来填充破损区域（目标区域），获得了较好的图像修复质量。在 Criminisi 算法的基础上，Wen 等人 [10]通过优化修复块的优先级计算公式使得在修复结构信息时取得了更好的效果，但并没有对修复速度做出改进；刘奎等人[11]提出了一种利用结构张量来决定目标区域修复顺序的改进算法，但是同样没有对修复速度做出优化。Masnou 等人[12]通过对图像中的等照度线进行匹配和连接来改进 Criminisi 算法中修复结构信息时产生的误差。Fr´ed´eric 等人[13]在 Masnou 等人工作的基础上通过使用欧拉曲线替代直线来修复图像的等照度线，使得修复效果更加自然。基于等照度线的改进算法缺点在于增加了修复时间，而且难以保证等照度线断点的正确匹配。

在提升修复速度方面，HUI 等人[14]则通过预先对源区域中的样本块进行分类来加快匹配速度。王昊京等人[15]通过放缩图像的方法显著地提升了 Criminisi 算法的修复速度，但是修复质量也随着缩放倍数的提升出现了明显的下降。而 Anupam 等人[16]针对 Criminisi 算法在修复质量和时间上同时进行了改进，相较于前述的几种算法取得了更为理想的处理效果，但仍然在修复时间、修复效果等方面存在不足。

之后 Criminisi 算法经过 V. Cheung, B. J. Frey 和 N. Jojic 改进，提出了 epitome 算法[17]，通过将二维图片的修复展开到三维从而适用于视频的修复问题。除了 epitome 算法外，Wexler, Shechtman 和 Irani 等人通过利用视频具有时空一致性的特性，通过在图像序列中取样并寻找最佳匹配块来对视频图像进行修复，使得匹配结果尽可能达到全局最优。Patwardhan 等人[18]在前两者工作的基础上，通过分别对视频的前景和背景进行有针对性的修复来改进了前者的工作，但是该算法要求视频背景是固定的。此后，Patwardhan 等人[19]又在之前工作的提出了一种改进方法来修复背景移动的视频，从而降低了修复过程中对视频背景固定的要求。

然而目前视频修复算法存在的一个普遍问题是修复时间过长，修复短短几秒的高清视频，修复时间可能长达几小时甚至数十小时，因此离真正的应用还有一定差距。但是随着 Hadoop 等用于大规模数据存储和分析的平台出现，为视频图

像处理提供了新的解决思路。但是由于 Hadoop 的工作方式类似于批处理操作，所以不适合用于搭建实时的分布式视频处理平台。和 Hadoop 类似，Storm 作为一个并行计算框架能够对大规模数据进行并行化处理，但是因为基于 Storm 的数据处理依靠的是网络直传和内存计算，一方面省去了 Storm 中等待和收集批处理数据的耗时和通过 hdfs 传输数据的耗时，另一方面也省略了作业在 Hadoop 中的调度延时，因此 Storm 处理数据的时延要比 Hadoop 低很多，更加适合于视频处理的流式计算[20]。

1.3 本文的研究内容

本文主要针对云计算平台进行了研究，并搭建了用于图像处理的分布式计算平台。然后针对图像修复算法对计算能力需求和海量视频处理并行化的趋势，基于 Storm 和 CUDA 平台搭建了一套分布式图像修复应用。在这套应用中，本文通过改进 Criminisi 算法来提高图像修复的质量和速度，对其中的特定部分进行 GPU 并行化改进。之后将 GPU 作为计算节点集成到 Storm 内部，使其能够运行在基于 Storm 搭建的并行化集群中，最终用 Storm 集群来对图像修复算法进行加速。本文主要工作包括以下几项。

1) 研究云计算平台中分布式系统的基本原理和结构，搭建基于 Strom 的图像处理集群，针对 Storm 间各个计算节点的拓扑结构重新编写调度算法。
2) 研究图像修复算法并对 Criminisi 算法进行改进。
3) 使用 GPU 对改进的图像修复算法进行并行优化，并对并行化后的图像修复的效果和加速比进行分析。
4) 基于 Storm 搭建云计算平台，将 CUDA 作为计算节点移植到 Storm 中，使得 Storm 能够对 CUDA 进行管理和调用，并将图像修复应用部署到集群中。
5) 对系统性能和修复效果进行评估，并总结和展望后续研究工作。

1.4 本文的组织结构

第一章为绪论部分，介绍了本文的研究目的和意义，然后讲述了下云计算平台和图像修复技术的概述和研究现状，并对文章的主要架构做了阐述。

第二章主要介绍了 Storm 的架构、模型及其相对于其他分布式架构在各方面的优势，讨论了如何通过修改 Storm 中的资源调度算法来提高资源的利用率。

第三章阐述了图像修复技术的基本原理，介绍了 Criminisi 算法的修复原理，并继续深入介绍了一种基于前者的改进算法。最后详细介绍本文提出的基于分水岭分割的快速图像修复算法。

第四章主要对如何用 Cuda 对图像修复算法进行并行化加速做出了详细的阐述。

第五章主要介绍了 Storm 的架构、模型及其相对于其他分布式架构在各方面的优势，讨论了如何通过修改 Storm 中的资源调度算法来提高资源的利用率。

第六章在完成系统设计后,对整个系统的性能进行客观的分析和总结。

1.5 本章小结

本章首先介绍了云计算和图像修复技术的背景和相关技术，综述了 Storm 和 Cuda 等平台的技术优势，对基于样本块的图像修复算法的发展背景做了简单的阐述，最后介绍了了本文的研究内容和文章的组织结构。

第二章 云计算平台相关技术介绍

云计算作为由分布式计算、分布式存储等多种传统计算机技术和新兴互联网技术结合产生的新事物，已经被学术界和产业界广泛应用。云计算平台的底层是大量的计算和存储资源，这些分布式资源被通过互联网的方式整合在一起，然后提供给用户。用户则根据自己的需求去申请需要的资源并按自己的需求进行运算。到目前为止，业界对云计算的定义可以归为三类：基础设施即服务（IaaS）提供给用户虚拟化的存储、处理、计算服务等基础设施和框架；平台即服务（PaaS）提供给用户在云环境下的程序应用开发平台；软件即服务（SaaS）则向用户提供了随时随地的应用程序调用。

2.1 Hadoop 处理框架介绍

Google 于 2004 年提出的 Hadoop 包含了 HDFS 存储系统和 MapReduce 计算模型，使其同时具备了强大的处理功能、方便的使用方式、廉价的成本，因此很快成为了产业界和学术界最常用的一种基础云计算平台，。

HDFS 的作用是用来搭建能够在廉价机器上部署的分布式文件系统。HDFS 和普通的分布式文件系统的区别在于 HDFS 能够为存储的数据提供更高的访问吞吐能力，非常适合大规模数据的计算。

但是比起 HDFS，更重要的是 Hadoop 的核心是 MapReduce 计算模型。MapReduce 旨在提供一种分布式计算模型的同时，提供一个批处理计算框架，使得可以通过大规模使用普通 PC 来处理 PB 级的数据[21]。除此之外，MapReduce 还提供了简易的应用接口，使得开发者只需关注业务逻辑本身就能完成任务，而不需要关注整个系统如何调度资源和系统的容错处理。MapReduce 的计算模型中提供了两个接口函数 Map 和 Reduce 给开发者来实现具体业务逻辑中的操作。

Map 函数通过接收键值对作为输入，当数据经过计算后又会以键值对形式将中间结果输出，并根据相同的 Key 值记录将数据分类聚合起来，然后作为输入传递给 Reduce 函数内的处理逻辑。

Reduce 函数在接收到 Map 函数传递来的中间结果之后，针对其中的 Key 值和 Value 集合进行相应的处理，然后再次生成键值对作为最终的计算结果。

MapReduce 框架具有很多优点，首先可以在部署在多台机器中并发工作，因此可以任意扩展，其次即使有机器发生故障也不会影响任务的正常运行，所以具有很高的容错性，。最后这个模型非常方便使用，用户只需编写对应的 Map 和 Reduce 函数即可完成对应的数据处理逻辑。

从上面的分析可以看出，MapReduce 的优势是数据的高吞吐量、支持海量数

据处理的大规模并行、细粒度的容错，但是这并不适合对时效性要求高的应用场景，主要原因有一下两个原因。

第一：Map 和 Reduce 的任务启动时间较长。因此，对于批处理任务来说，其任务启动时间相对后续任务执行时间来说所占比例并不大，但是对于时效性要求较高的应用，较长的启动时间会导致任务处理的延时变长，使得整体时效性降低。

第二：在 MapReduce 执行一次操作的过程中可能多次对磁盘和网络上的文件进行读/写及传输操作。比如读取初始数据块、将 Map 阶段的中间结果任务存储到本地磁盘中、Shuffle 阶段的数据传输、Reduce 阶段从磁盘读取数据和最终结果的写入等。当一个任务需要迭代操作时，需要同一个 MapReduce 任务反复执行，此时如果多次对磁盘和网络上的文件进行读/写及传输操作，将导致任务整体运行效率低下。

为了弥补 Hadoop 在处理时效性要求较高的应用时凸显出的缺陷，流式计算应运而生。与 Hadoop 类似，流式计算系统提供了完整的编程框架和编程接口，同时保证了系统性能和数据不丢失的同时，但流式计算系统在处理时效性要求较高的场景时更加适用。与目前大多数大数据处理系统一样，常见的流式计算系统架构分为两种:主从模式和 P2P 模式。大多数系统架构遵循主从模式，主要是因为主控节点在管理系统资源时比较方便。而 P2P 架构因为没有中心控制节点，所以在管理系统资源时更加复杂。

2.2 Storm 处理框架介绍

Storm 作为如今最常用的分布式计算系统之一，能够对大规模流式数据进行高容错的实时计算，弥补了 Hadoop 只能以批处理的方式对离线数据进行计算的缺点，使得对流数据的处理变得更加容易。因此，Storm 能够满足为海量视频数据处理算法提供实时分布式处理平台的需求。

2.2.1 Storm 物理模型

Storm 集群和 Hadoop 集群有着类似的特点。不过两者的工作方式有着很大的差别，Hadoop 上运行的工作称作 MapReduce job， MapReduce job 在 map 阶段需要等待所有任务完成并写入 HDFS 再开始 reduce 阶段，然后等待所有的 reduce 操作结束后，关闭所有的处理进程。而 Storm 中部署的 Topology 会一直留在内存中，使得处理进程保持运行下去，从而保证了 Storm 处理数据的效率。

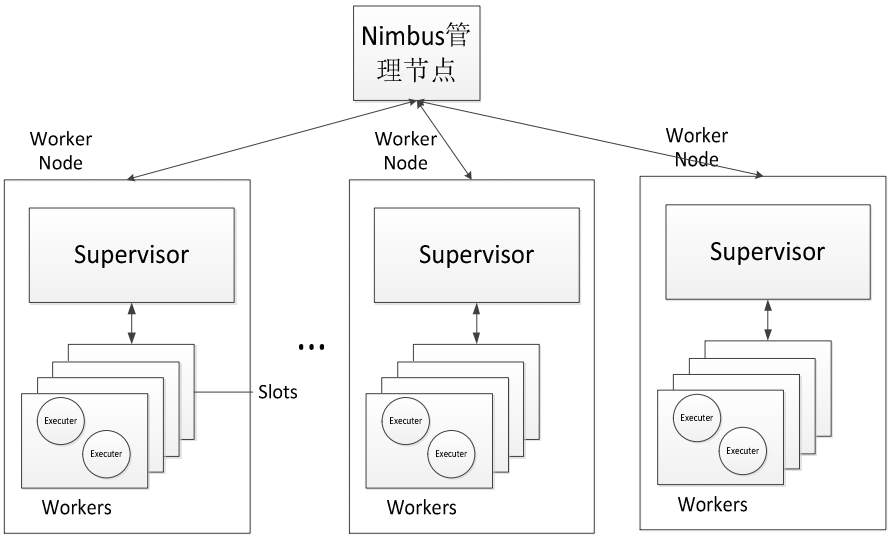


图 2-1 Storm 物理模型

Storm 采取了中心化的星形逻辑结构方式，在 Storm 集群中存在两种节点，一种是控制节点，在上面运行有一个 Nimbus 进程，用于在 Storm 中指定任务给各个工作节点，并监视各个节点的任务信息。另一种是工作节点，每个工作节点通过运行 Supervisor 进程来对当前节点处理的任务进度进行监听和管理。控制节点通过 Zookeeper 来发布消息指令给工作节点，工作节点则通过 Zookeeper 来获取这些指令并执行。Storm 的物理模型如图 2-1 所示，在每个工作节点中除了 Supervisor 进程外，还有许多 Worker 进程，每个 Worker 进程里又会运行一个或多个 executer 线程，每个 executer 线程负责执行一个 Topology 的 component 的 task 实例。

2.2.2 Storm 逻辑模型

Storm 逻辑模型主要由 Spout、Bolt 和 Topology 三个元素组成，其中 Topology 形成的有向无环图如图 2-2 所示。

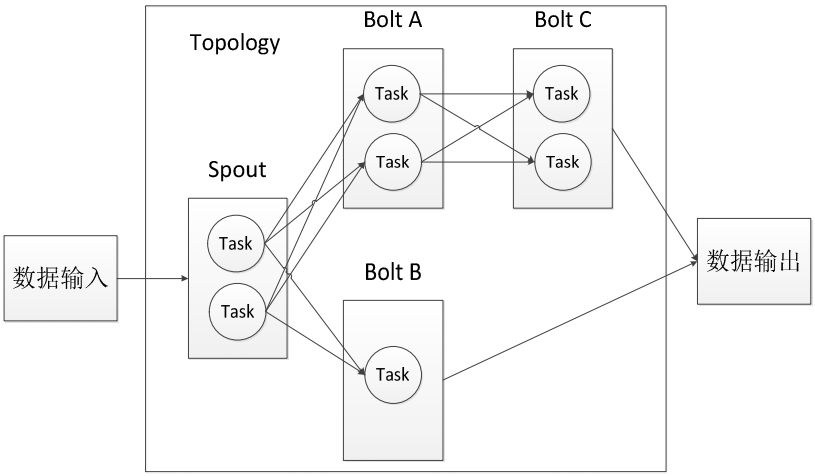


图 2-2 Storm 逻辑结构

Spout：Spout 是 Storm 中每个 Topology 里的消息源，是获取和分发消息的入口。Spout 的运行过程是先从一个外部管道读取消息，然后向 Topology 内部其他的节点发送消息。Spout 通过 nextTurple 发送消息，每条消息带有 stream 和 field 两个属性，分别代表了这个消息所属的流和域，系统会根据这两个属性将消息发送到对应的节点当中。当消息发送成功时，Spout 会收到 ack 消息，当消息发送失败时，会收到 fail 消息。

Bolt：Bolt 是 Topology 中的计算拓扑组件，它把 Spout 或者其他 Bolt 产生的消息作为输入，然后产生新的消息作为输出。

Topology：一个 Topology 结构由发送数据的 Spout 节点和处理数据的 Bolt 节点组成。各组件间的通信通过 stream 和 field 来实现数据流的控制和数据的传递，它们之间用于传递信息的数据包被称为 Tuples。一个 Topology 在部署之后将会持续运行，直到开发者选择手动 kill 掉该 Topology 服务。在消息的发送过程中，Storm 提供了 shuffle 和 field 等多种消息分发类型，能够满足各种业务逻辑的需求。除此之外，Storm 的高可靠性也保证了 Storm 中传递的每个消息都能得到处理。即使有节点出现故障，Storm 也能立即通过重新分配资源的方式来重启节点，避免了处理过程中消息的丢失。

2.2.3 Storm 优势

Storm 作为流式计算的主流架构在整体设计及其内部系统协调方面有着诸多优势，下面结合特点与优势对其进行分析。

简便的编程模型：与 Hadoop 中的 Map/Reduce 编程模型类似，Storm 提供了 Spout/Bolt 原语作为开发具体应用的接口。通过 Spout/Bolt 原语，开发者可以不用考虑如何实现一个分布式系统所涉及的消息通信、资源管理等操作，只需要把精力放在开发自身的应用上面。

可扩展：如图 2-3 所示，Storm 由一个 nimbus 节点和多个 worker 节点组成，nimbus 节点负责系统资源的管理的处理，工作节点则负责任务的执行。在一个工作节点中，会启动多个工作（worker）进程，每个工作进程又对应了一到多个工作线程（Executor）。而 Storm 中的任务 topology 则被以 component(spout 或 bolt) 的实例形式分配到各个 worker 的 executor 中。通过这种方式，Storm 实现了多个线程、进程和服务器之间并行进行，使得整个系统能够支持灵活的水平扩展。

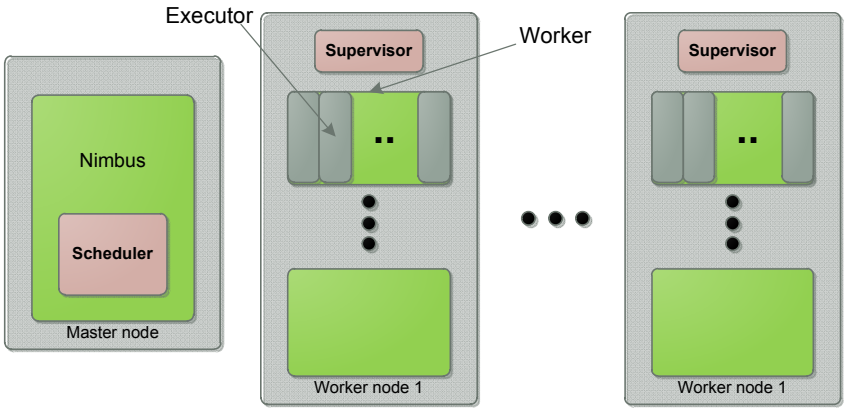


图 2-3 Storm 结构示意图

Worker、Executor 和 Task 的详细描述如下所示：

1. Worker（进程），在 Storm 中的处理节点中，一个 Worker 进程对应一个 slot，每个 Worker 进程默认分配一个计算机的物理 cpu 核心。当有 topology 被提交时，topology 中的 component 会依照调度器的要求被分配到各个 worker 的 executor 中执行，executor 是 worker 中的线程，每个线程对应了 component 的一个实例，最终每个 worker 都包含了一个由 component 组成的子集。开发者也可以根据实际情况手动的指定 topology 中 component 的分配来满足特殊的应用需求。
2. Executor（线程），由 worker 创建的一个独立线程，在每个 Worker 中可以同时运行多个 Executer 线程，每个 Executer 只会运行 1 个 Topology 中的 1 个 Componment 的 Task 实例。
3. Task（任务），最终执行 Componment 中逻辑的实例单元，具体数量由用户在配置文件中指定。在 Topology 开始运行以后，每个 Component 的实例数目固定不变，但是被分配 Executor 数目会根据系统资源的负载动态调整。

高可靠性：为了保证每条消息都能被最终处理，Storm 会针对每个 topology 任务构建了一棵消息树来追踪每条消息是否被处理。当一条消息从 spout 节点发出，经过各个 bolt 节点，直到最终到达 ack 节点后，这条消息才算完全被处理。如果这其中任何一步发生了错误导致消息处理终端，则这条消息就会被定义为发送失败，此时 spout 节点就会因为没有接收到 ack 消息或是接收到了中间节点的 fail 消息进而不停重新发送该消息，直到这条消息到达 ack 节点并被接收为止。于此同时，为了减少这种通过构建消息树来追踪消息的内存消耗，Storm 采用了异或计算的方式来对最终结果进行检测，Storm 在发送消息前，首先给每个消息一个唯一的 id，然后在发送的过程中通过判断异或计算的结果是否为 0 来判断该消息是否被完全处理，这样就达到了在保证消息传递的同时，又没有浪费过多内存的目的。下面详细说明了 Storm 在消息发送过程中遇到不同级别故障时的应

对策略：

1) 任务故障
   a) Spout 节点故障：Spout 节点故障会导致源数据的发送失败，解决办法是 Spout 节点重新发送消息直到消息 ack 为止，这样就保证了消息传递的完整性。
   b) Bolt 节点故障：Bolt 节点故障会导致所有经过这个 bolt 节点处理的消息无法继续传递，之后会导致 Spout 节点无法接收到 ack 消息，Spout 节点按照消息发送失败的错误进行处理。
   c) Acker 节点故障：和 bolt 节点故障类似，Spout 会按消息发送失败的情况再次进行发送，直到消息 ack 为止。
2) slot 故障
   a) supervisor 失败:只负责管理各节点的状态，本身无状态且不参与到 Storm 的任务处理当中。即使失败也不会影响当前部署任务的工作状态，重启 supervisor 即可。
   b) worker 失败:由于 worker 节点是由 supervisor 节点进行管理的，所以即使发生了故障，supervisor 节点也会马上发现故障并对该节点进行重启操作。
   c) nimbus 失败: nimbus 进程独立于其他进程，并且和 supervisor 进程一样没有状态，只负责提交任务，在发生故障时不会影响已经提交的任务执行，只会影响新任务的提交。当发生故障时，重新启动 nimbus 进程即可。
3) 集群节点故障
   a) Storm 集群中的节点故障: nimbus 会通过 scheduler 对故障节点中的任务进行重新分配，从而保证所有任务的正常运行。
   b) Zookeeper 集群中的节点故障:Zookeeper 自身的故障处理机制能保证在少于半数的机器发生故障时整个集群仍可正常运行。

2.2.4 StormCV

尽管 Storm 本身并没有提供图像处理接口，但是和 Hadoop 中的 HIPI(Hadoop Image Process Interface)等应用类似，Storm 中也有很多第三方开发的应用提供了图像处理接口。StormCV 作为 Storm 的一种扩展，专门被设计用来为分布式图像处理程序提供流水线式的处理环境。

StormCV 通过加入特定的计算机视觉操作和数据模型使得 Storm 可以处理图像。这些模型和操作包括图像帧的获取和特征提取，颜色统计等等。由于集成了 OpenCV，因此 StormCV 可以轻松地使用 OpenCV 库中的其他功能。此外，因为视频文件也是由连续的图像组成，所以 StormCV 在处理视频流的同时也可以处理对视频文件和批量图像进行处理。

与 Storm 不同，StormCV 主要由以下几个部分组成：

（1） Model:

Model 封装了一系列用于 Spouts 和 Bolts 之间传递的数据对象，包括视频帧、图像特征和描述类等等。这些对象最终被自动序列化到 Storm Tuples 中用于传输，因此开发者通过操作这些对象包含的数据而不是 Tuples 中的数据来实现开发过程。因此包含常见图像属性的 Model 对象构成了 StormCV 的核心。下面简单的描述了 Model 之间各个元素。

GroupOfFrames:将一帧或多帧图像封装在一个对象中，使得整个系统能够一次发送多帧图像到同一个目的地。

Frame:包含一个或多个特征的单帧图像对象。

Feature:用于描述图像的某个单一特征的描述符。

Descriptor:用于描述图像特征属性的描述符，包含了该描述符所作用的区域和对应的值。

所有上述对象中均包含有两个域用来表示节点间数据路由方式和图像帧的排列顺序:

streamId: 用于表示该对象属于哪个流。每个单独的视频流或文件都拥有一个对应的 streamId 来使得系统能够区分出当前处理的图像帧和特征属于哪个视频。

sequenceNr:用于表示当前的图像帧在流通道中的序号。

除此之外，StormCV 还提供了一个元数据映射用于给 Model 中的对象添加元数据信息。

（2） Fetchers:

Fetchers 运行在 Spouts 中，负责读取图像数据。Model 中的图像处理、产生和获取对象的操作由 Fetches 和 Operations 操作进行。其中 Fetchers 负责在 Spout 中对图像数据进行读取，StormCV 中常见的 Fetchers 实现有以下几种：

StreamFrameFetcher: 获取视频流，以特定的间隔提取图像然后作为帧对象发送出去。

FileFrameFetcher: 从远程地址读取视频文件，以特定的间隔提取图像然后作为帧对象发送出去。使用 FileConnector 来获取远程地址上的视频文件

ImageFetcher: 从特定源读取视频流，然后将图像作为帧对象发送出去。

RefreshingImageFetcher: 从不断刷新的数据源读取图像，例如网络摄像头等等。

FetchAndOperateFetcher: 在 Spout 中获取图像源的同时使用 Operation 对图像进行操作，从而降低 Spout 到 Bolt 间的数据传递。

（3） Operations:

从 Model 中读取数据，并产生用于输出的结果和对象，比如从 Model 中获取一帧图像，然后输出其中的多个特征对象。Operations 用于在 Bolts 内部执行数

据的分析和处理操作。一个 operation 从 model 中获取一个或多个对象作为输入，然后产生对应的输出。StormCV 中的 Operations 操作主要分为以下两种：

SingleInputOperations:对一个输入对象进行操作

BatchOperations: 需要两个或多个对象才能进行操作

（4） Batchers

Batcher 负责在进行具体的 BatchOperations 前对收到的数据进行整合直到能够作为之后 BatchOperations 的输入对象。Stormcv 包含以下几种 Batchers 对象：

a) DiscreteWindowBatcher: 创建指定长度的对象作为 batch。每当一个 batch 对象更新时，其原值随即被移除。这种 batch 通常按照特定的图像帧序号来创建，比如[0,25], [25,50], [50,75]。

b) SlidingWindowBatcher: 创建指定长度的对象作为 batch。但是每次更新时只移除第一个对象。这种 batch 通常也按照特定的图像帧序号来创建，比如[0,25], [25,50], [50,75]。

c) SeqeunceNrBatcher:通过顺序编号来创建 batches 输入对象，通常用于合成一张图像的多个特征。

Batcher 通过获取对象的特征来对图像进行分组。通常情况下这种分组是根据 streamID 来进行的，从而保证了同一个视频流的图像最终被分配和打包在一起。通常一个 Bolt 在处理数据的过程中通过 Batcher 中的 fieldsGrouping 操作来筛选需要的数据并进行打包。

（5） Groupings:

作为 Storm 的一部分，Grouping 控制着数据在平台中的传递路径。当一个 bolt 节点拥有许多实例时，Storm 需要决定哪些实例才是数据传输过程中真正的目的地。Storm 本身包含了一系列 groupings 方式比如 shuffleGrouping 和 fieldsGrouping，这些 groupings 方式也同时能够在 StormCV 中使用。StormCV 提供了几种特殊的的 groupings 方式，如下所示：

FeatureGrouping：可以通过特定的特征来进行分组操作。比如经过一个 Bolt 节点操作会产生多个特征，但之后的某个 Bolt 只需要‘SIFT’特征作为输入，因此这个 Bolt 就会选择 FeatureGrouping 的分类方式有选择的将图像和特征传送到对应的下个 Bolt 节点。

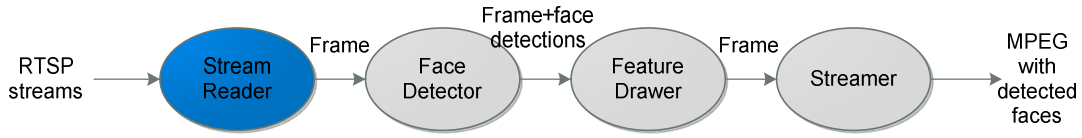
ModGrouping：作为另一种过滤输入对象的方式，ModGrouping 可以获取特定序号的对象。比如存在一个输入流，其中包含的输入对象拥有序列号{0,1,10,11,20,21}，当 ModGrouping 选择 mod 10 作为限制条件时，最终只有序号为 0、10 和 20 的对象被选择。

SupervisedGrouping: SupervisedGrouping 会在任务准备阶段将源任务和目标任务映射在一起，这样在运行的过程中所有的 tuples 都会经过该路径进行传递。

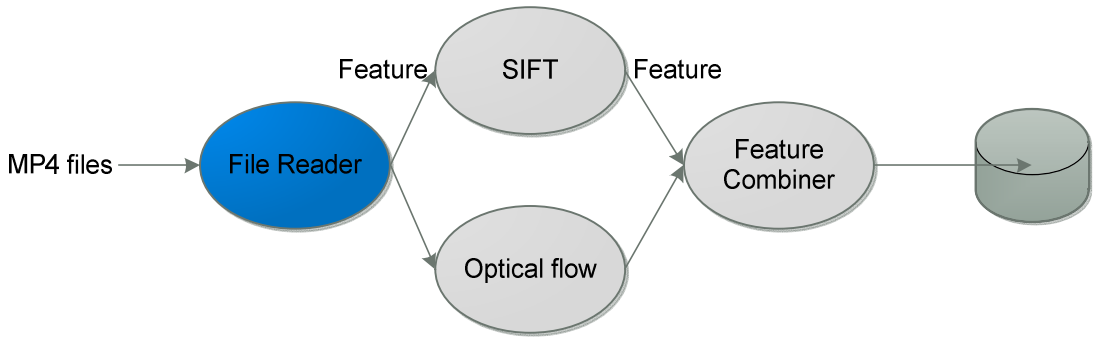
因此当一个 topology 中对象被传递的顺序要求非常严格时，可以使用这种方式来传递数据。

TileGrouping:当一帧图像被分割成碎片操作，并在操作完成后重新将结果合并在一起时使用。

下图显示了分别使用 Stormcv 读取视频流和视频文件的示例。其中读取视频流示例中，stormcv 通过 spout 节点读取 RTSP 视频流然后转化成视频帧传递到之后负责人脸检测和人脸提取的 Bolt 中，最后通过 Streamer 节点将处理完的视频帧转换为 MPEG 的视频流发送出去。而读取文件示例则通过 spout 读入文件，然后通过 FeatureGrouping 的方式将图像发往不同的节点进行处理，处理完之后又将提取出的特征进行合并，最后将提取出的图像特征存入数据库。



(a)stormcv 读取视频流



(a)stormcv 读取文件

图 2-4 stomcv 运行示例图

2.2.5 Storm Scheduler

Scheduler 是 Storm 中的资源分配器。当有新的 Topology 提交时，Scheduler 会将其中的各个 component 分配到不同的节点上进行运算。Storm 中的默认调度器被称为 EvenScheduler。EvenScheduler 通过简单的轮询策略来保证资源分配的公平。在分配资源的第一阶段，EvenScheduler 按照 component 分组遍历 topology 中所有的 executor，然后将他们按照轮询的方式为这些 executor 申请需要的 worker。在分配的第二阶段，EvenScheduler 会根据各个 worker node 的负载，将 worker 按照轮询的方式公平的分配给 worker node。这样就保证了每个 worker 被分配到了大致相等数量的 executor，同时也保证了每个 worker node 上分配了数量相等的 worker。

Storm 允许用户根据自己的需求去实现通用的 scheduler。通常一个通用的

scheduler 将一个 topology 结构（由 nimbus 提供）作为输入，这个 topology 表示了一张加权有图 G(V,T)。然后 scheduler 为这个 topology 制定一个计划来定义 executor 到 worker 和 worker 到 worker node 的分配。此外，Storm 的提供了 Ischeduler 接口来实现可插拔式的定制调度器。该接口需要两个参数作为输入，第一个是包含有所有 topology 当前运行信息的对象，第二个参数是包含当前所有物理集群运行信息的对象，包括所有 worker node、slot 和当前分配情况的信息[22]。

目前 Storm 提供了 3 种调度器：

1. EvenScheduler：根据集群中正在运行的 Topology 情况，平均的分配计算资源
2. DefaultScheduler：首先回收不再使用的资源，然后调用 EvenScheduler 进行平均分配
3. IsolationScheduler：可以单独为某些 Topology 指定它们需要的机器资源

此外，文献 [23]分别实现了 TOPOLOGY-BASED Scheduler 和 TRAFFIC-BASED Scheduler 来对 Storm 中的 Scheduler 进行改进。下面详细的介绍其中的三种 scheduler。

EvenScheduler

调度过程：

1. 获取所有需要进行任务调度的 Topology 的集合。当 Topology 需要的 Worker 数目大于已经分配给该 Topology 的 Worker 数目或者该 Topology 尚未分配的 Executer 的数目大于 0 时，任务调度被执行。
2. 在得到需要调度的 Topology 信息后，Storm 先计算出当前 Topology 中可使用的 Slot 个数，然后计算出集群当前的可用资源。
3. 对 Topology 进行任务分配，将计算出来的 Slot 分配给与该 Topology 相对应的 Executer。

调度示例：

假设当前集群有 3 个 Worker 节点 A、B、C，每个节点有 3 个进程（Slot），如图 2-5

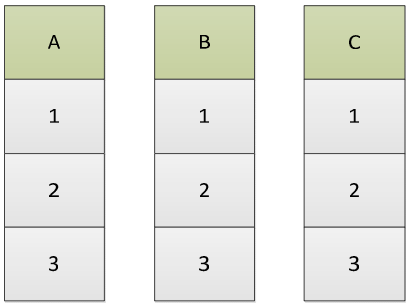


图 2-5EvenScheduler 调度示例

当一个新的 Topology 被提交到集群中时，EvenScheduler 将 Topology 的 component 分配到不同的 Slot 中用于计算。但在分配资源之前，为了保证公平，Storm 会在获取集群可用资源后会先对 Slot 进行排序。每次排序完之后，就会将 Topology 中的 component 按顺序分配到 Slot 中。

假设 3 台机器的 Slot 在经过 Scheduler 排序之后的顺序为[A，1]、[B，1]、[C，1]、[A，2]、[B，2]、[C，2]、[A，3]、[B，3]、[C，3]现在有两个 Topology T1、T2， T1 需要 2 个 Slot，T2 需要 4 个 Slot。

1. T1 提交后，Storm 会分配前两个 Slot [A，1]、[B，1]给 T1，如图 2-6

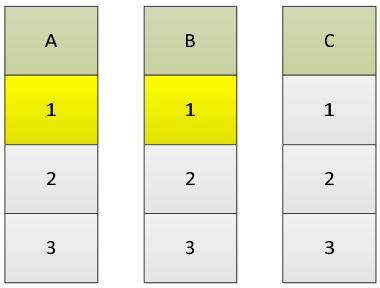


图 2-6 EvenScheduler 调度示例

2. 然后 T2 提交进行，这时集群可用资源为 [A，2]、[B，2]、[C，1]、[A，3]、[B，3]、[C，2]、[C，3]（重新排序之后的顺序），则 T2 会占用[A，2]、[B，2]、[C，1]、[A，3]，如图 2-7：

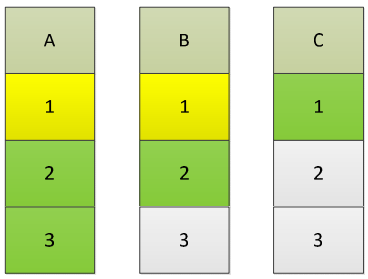


图 2-7 EvenScheduler 调度示例

从最终的分配结果可以看出 Storm 自带的分配策略会导致任务分布不均匀。

TOPOLOGY-BASED Scheduler（Offline Scheduler）

为了调整 EvenScheduler 可能造成的任务不均匀分配。文献 [23]首先提出了

一种 Offline Scheduler。Offline Scheduler 首先检查 Topology 的结构，然后查找最方便的 slot 来安排对应的 executor。首先可以通过 stream 的配置信息得到一个由 Topology 中的 compenent 组成的偏序集。如果存在一个 component 名为 c_i，它向另一个名为 c_j 的 component 发送 tuple，则存在关系 $c_i<c_j$。当同时存在 $c_i<c_j$ 和 $c_j<c_k$，则由于传递性存在关系 $c_i<c_k$。由于 Topology 都为有向无环图构成，所以可以根据偏序关系建立一个 component 间的线性关系集 Φ。如果 $c_i<c_j$ 存在，那么 c_i 在 Φ 中先于 c_j 出现。如果 $c_i<c_j$ 或者 $c_i>c_j$ 都不存在，c_i 和 c_j 在 Φ 中可以以任何顺序出现。Φ 中的第一个元素是 Topology 中的一个 Spout 节点。之后 Offline Scheduler 利用启发式搜索来迭代更新 Φ，对于任意 c_i，Offline Scheduler 只会将它的 component 放置在特定的 slot 中，这些 slot 必须满足包含向 c_i 发送消息的 component。

Offline Scheduler 特点：

1. 离线执行
2. 考虑 Topology 中 Component 之间的关联
3. 不考虑运行过程中的实时负载和分组

调度阶段：

1. Executor 排序后按顺序发配给 Slot
2. 将 Slot 指定给 Worker 节点

调度策略：

1. Component 间按偏序排列，得到一个线性的偏序集 Φ。
2. Scheduler 分配规则：存在两个 Component c_i 和 c_j（i<j），当 c_i 向 c_j 发送 Tuple，c_j 会将它的 executors 指定到正在运行 c_i 中 executors 的 slot 中。

调度示例

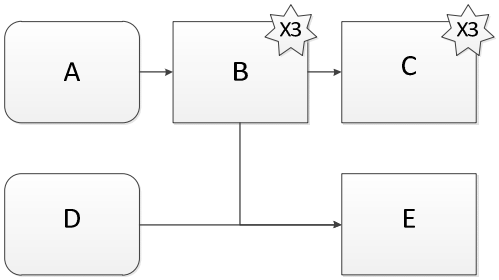


图 2-8 topology 示例

假设有 2 个 Spout 节点 A、D 和 7 个 Bolt 节点，3 个 B、3 个 C 和一个 E，其中 A 向 B 发送 Tuple，D 向 E 发送 Tuple，B 向 C 和 E 发送 Tuple。

按照偏序规则可以得到以下的顺序

$$A < B_{1,\ 2,\ 3} < C_{1,\ 2,\ 3}$$

$$A < B_{1,\ 2,\ 3} < E$$

$$D < E$$

得到的偏序集合为

$$S = \{A,\ B_{1,\ 2,\ 3},\ C_{1,\ 2,\ 3},\ D,\ E\}$$

假设 Storm 中有 4 个 slot，每个 slot 中有个 3 个 executer，将每个 component 分配到 slot 中得到的最终结果如下所示。可以看出，具有偏序关系的 component 都被尽可能的分配到了同个 slot 中。

	Executer1	Executer2	Executer3
Slot1	A	B	C
Slot2	B	C	E
Slot3	B	C	
Slot4	D		

表 2-1 topology-based scheduler 任务分配结果

由于 Offline Scheduler 是在 Topology 启动前执行的，所以既没有考虑负载和阻塞信息，也没有考虑内存和 cpu 的限制。

TRAFFIC-BASED Scheduler

与 Offline Scheduler 相比，Online Scheduler 能够根据运行时观测到的 executor 间的通信情况来动态的调整 node 之间和 slot 之间的负载。Online Scheduler 在给 executer 分配 node 节点时必须满足以下三个限制：（1）每个 Topology 必须分布在尽可能少的 worker 上（2）不超过每个 worker 节点上可获取的 slot 数量（3）不超过每个节点拥有的计算能力并使得节点间的通信开销最小。Online Scheduler 通过在运行时执行，使得 Storm 集群能够根据负载的变化动态的声明资源。

Online Scheduler 使用 CPU 利用率来测量每个节点应被赋予的负载和每个 executor 实际产生的负载。通过计算 CPU 利用率，我们可以合理的预测由异构节点组成的集群的负载信息。比如一个 executor 在 1GHz 的 CPU 上运行时占用了 10%的 CPU，那么把它移植到一个 2GHz 的 CPU 上运行时应该只占用 5%的 CPU。

在引入了 CPU 利用率的概念后，对于每个节点 n_i，一旦它的负载 L_i 和容量 B_i 满足 $L_i>B_i$ 并持续超过 X_i 秒后（X_i 是时间窗口），Online Scheduler 发起一次 reschedule 操作。同时通过测量 executor 间传递的消息数量来减少节点内部的阻塞。Online Scheduler 使用 $R_{i,\ j,\ k}$ 来表示 executor $e_{i,\ j}$ 和 executor $e_{j,\ k}$ 之间消息发送的频率，再通过统计所有节点上 executor 间的通信量可以得知节点间整体的通信情况。这样每隔一段时间，一旦发现节点间的通信量发生超过阈值的变化便发起一次 reschedule。

Online Scheduler 分为两阶段进行，（1）第一阶段 A1：ε->W，将 executer 分配到 worker 上，（2）第二阶段 A2：W ->N;，将 worker 分配到 node 上。其中

N={n_i}（i=1…N）表示节点的集合，W={$w_{i,j}$}(i=1…T，j=1…min(E_i，W_i))表示 worker 的集合，ε={$e_{i,j}$}(i=1…N，j=1…Ei)表示 executor 的集合。

Online Scheduler 在分配资源的同时需要满足以下三个条件：

（1） 一个节点上的总负载必须小于它的承载能力

$$\forall k=1...N \sum_{\substack{A_2(A_1(e_{i,j}))=n_k \\ i=1...T;j=1...E_i}} L_{i,j} \le B_k \qquad (2\text{-}1)$$

（2）每个 topology 的 executor 尽量运行在较少的 worker 上

$$\forall k=1...T|\{w \in W : A_1(e_{i,j}) = w, j = 1...E_i\}|=\min(E_i, W_i) \qquad (2\text{-}2)$$

（3）节点之间的信息量需要尽可能少

$$\min \sum_{\substack{j,k:A_2(A_1(e_{i,j}))\neq A_2(A_1(e_{i,k})) \\ i=1...T;j,k=1...E_i}} R_{i,j,k} \qquad (2\text{-}3)$$

在第一阶段中，先将每个 Topology 中的每对 executors 按照信息的发送频率降序排列。对于任意一对 executors，如果它们都还未被分配，则会将它们分配到负载最小的 worker 上。否则，建立一个 worker 集合 Π，Π 至多包含三个成员，每对 executors 分别所在的 worker 和集群中负载最低的 worker。然后计算重新分配 executor 的所有组合，选出 worker 之间通信量最少的一种组合作为重新分配 executor 的方案。在第二阶段中，采用和第一阶段相同的方式将 worker 分配到 node 中。

Online Scheduler 特点：

1. 在线执行
2. 能够通过调度 Worker node 来降低网络中的拥塞
3. 当 node 内部的拥塞超过设定的百分比时，调度被触发

调度阶段：

1. 启动 default scheduler
2. 持续监控 executer 内部的拥塞情况，当性能需要改善时，调度被触发，调度过程分为以下三步：
   i. 计算各节点上的负载
   ii. 指定 executors 给 workers
   iii. 指定 workers 给 nodes

分配策略：

1. 将所有流相连的 executors 组合按流量降序排列，按顺序分配给 worker
2. 如果一对中的 executor 都没有被调度过，则将它们同时放在负载最小的 worker 上
3. 否则就从负载最小的 worker 和两个 executor 所在的 worker 中选出 2 个 worker 重新分配 executor，分配要求要满足调度完成后 slot 内部

负载最小

4. 继续进行 2-3 直到所有 executor 被分配到合适的 worker 中
5. 按照分配 executor 到 worker 的方式，将 worker 分配到 node 中

调度示例

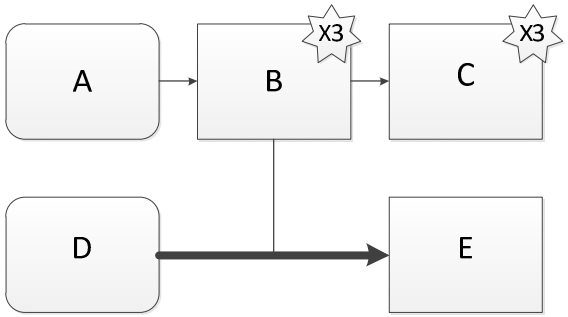


图 2-9 topology 示例

假设有 2 个 Spout 节点 A、D 和 7 个 Bolt 节点，3 个 B、3 个 C 和一个 E，其中 A 向 B 发送 Tuple，D 向 E 发送 Tuple，B 向 C 和 E 发送 Tuple。

每对相邻节点间的负载如下表

EX 1	EX 2	RATE
D	E	10
A	B	1
B	C	1
B	E	1

表 2-2 节点间传输频率

假设 Storm 中有 4 个 slot，每个 slot 中有个 3 个 executer，将每个 component 按照当前负载情况分配到 slot 中得到的最终结果如下所示。可以看出，负载最大的两个 component 被优先分配到相同的 slot 中，从而保证了数据传输的吞吐效率。

	Executer1	Executer2	Executer3
Slot1	D	E	
Slot2	A	B	C
Slot3	B	C	
Slot4	B	C	

表 2-3 traffic-based scheduler 任务分配结果

2.3 Kafka 介绍

由于在云计算平台中，常常需要对离线和实时的数据进行分析，这时就需要专门的工具对数据进行存储和管理。由于 Storm 的实时流计算完全基于内存运算，在数据的存储方面没有做任何实现。因此，需要一个用于连接数据源的消息中间件来将存储的数据传递到 Storm 中。本文中使用 Kafka 作为这个消息中间件，首先通过前端的采集程序将文件和视频流等数据源源不断地采集到队列中，然后通过 producer 将产生的图像数据发布到 Kafka 中，最后 Storm 的 Topology 作为消息的 Consumer 通过订阅的方式来拉取消息。作为一种分布式消息订阅发布系统，Kafka 具有以下几个优点[24]：

（1）速度快：一个简单的 Kafka 缓存能够每秒同时处理从数以千计客户端发送过来的上百兆的读写操作。

（2）可扩展：Kafka 的设计初衷就是为了能够实现仅仅依靠一个简单的集群就能为一个大规模项目提供中心数据的网络骨干。因此 Kafka 集群能够在运行过程中，简便和灵活的对集群进行扩充。在数据的传递过程中，由于 Kafka 会对数据流进行分割，因此即使数据流超过单一节点机器的带宽也通过分散流到集群中的多个发布节点进行传递和依赖多个订阅节点合作接收的方式来传递数据。

（3）可靠性：Kafka 通过将消息存储在硬盘里并在集群中进行备份的方式来防止数据发生丢失。每个缓存都能在不影响性能的情况下处理 TB 级的消息。

（4）分布式：Kafka 最初的设计就是为了提供一个具有高可靠性和高容错性的中心-集群式消息发布/订阅系统。

Kafka 作为一个基于副本的高可靠消息系统，在处理消息过程中主要用到以下几种组件：

Topic：用于区分发送消息源的分类。每个 topic 将被分成多个 partition(区),每个 partition 在存储层面是 append log 文件。如下图所示

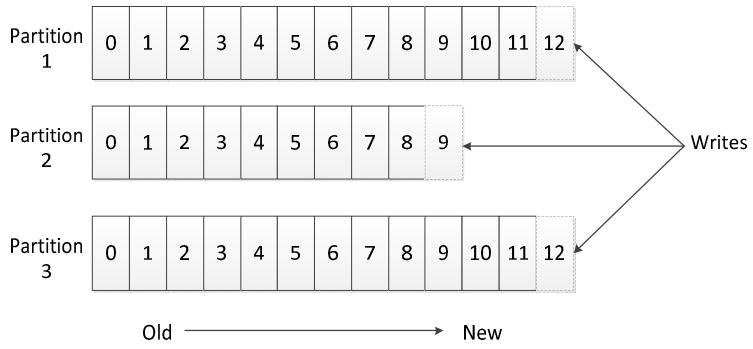


图 2-10 topic 分区示意图

Partition：partition 是一个不可修改的有序消息队列，只能不断地往该队列后面添加 log。这些队列中的消息都被分配到一个顺序 id 用于标记出它们处于队列

中的位置。Kafka 集群会保留所有的消息，不管它们是否已经被 consumer 读取。Partition 主要用于 log 的存储，由于在实际处理的过程中会产生大量的数据，这些数据有可能超过单一节点的存储上限，所以需要多个 partition 来对一个 topic 的消息 log 进行记录。

Producers：消息和数据的生产者，根据 topics 来发送消息。负责决定消息被发往对应 topic 中的哪个 partition。通常采取的是轮询的发送方式，或者根据自定义的语义分组策略发送消息。

Consumers：消息和数据消费者，consumers 负责订阅 Topic 并接收对应的消息并处理。Consumers 会通过组名进行分组，每条对应 topic 订阅的消息会被传递到该组中一个 consumer 的实例中。Consumer 的实例可能分散在不同的进程里，也可能被部署在不同的机器中。如果所有的 consumer 实例都包含在相同的分组中，consumers 间就组成了一个传统的消息队列方式来分配消息。如果 consumer 实例都分配在不同的分组中，那么分配消息的方式会按照发布/定阅的方式来进行，所有的消息会被广播到所有的消息源中进行。

Broker：缓存代理，Kafa 集群中的服务器统称为 broker。

图 2-11 显示了 Kafka 的交互流程，Kafka 首先在集群中定义了一个 tpoic 来区分消息源的分类，然后由 producer 根据对应的 topic 向 Kafka 的 broker 中写入数据，最后又 consumer 根据 topic 从集群中获取订阅的消息来进行处理。

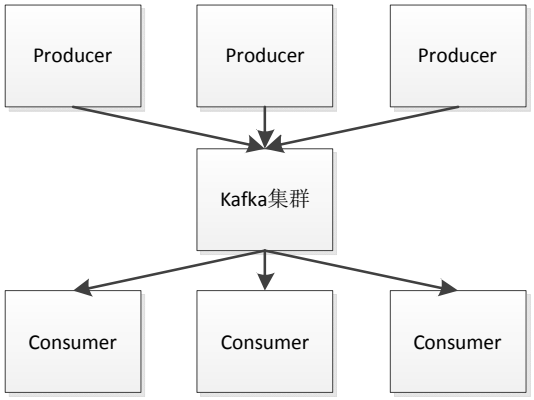


图 2-11 Kafka 消息处理模型

由于 Kafka 的特性是基于分布式的，所以 Kafka 内部缓存 broker 也是能够被多个节点分区和覆盖的。下图显示了一个两个 server 的 Kafka 集群，其中拥有的 4 个 partition 被分配到 2 个 consumer 分组中，分组 A 拥有 2 个 cousmer 实例，分组 B 拥有 4 个 consumer 实例。最终每组中的 consumer 根据 topic 去指定的 broker 中获取数据。

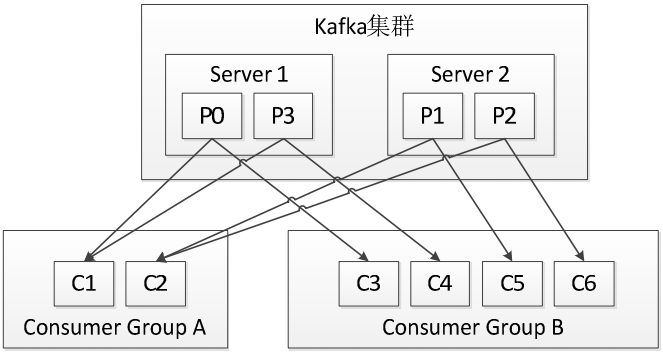


图 2-12 Consumer 获取发送消息

通过将 broker 进行多个节点分区可以使得 consumer 在读取消息时能够最大程度利用网络中的带宽。

2.4 本章小结

本章首先介绍了云计算平台中的常用框架，然后详细描述了 Storm 处理框架的物理和逻辑模型，并对 Storm 的优势进行了分析。之后介绍了 Storm 中一些与本文相关的组件，最后介绍了一种为云计算平台提供数据分发的分布式订阅消息系统 Kafka。

第三章 基于样本块的图像修复算法

图像修复技术主要分为两类。一类是基于 Bertalmio 等人[6]提出的基于偏微分方程的方法，通过对破损区域的边界进行不同方向的扩散来对图像进行修复。另一类方法是最早由 Efros A[8]提出的基于纹理合成的图像修复方法，通过对纹理的适当采样来对破损区域进行填补，适合大区域的图像修补。纹理合成的方法容易导致修复后的图像结构信息出现断裂。Criminisi 等人[9]在纹理合成的基础之上结合了边界扩散算法的优点提出了基于样本块的图像修复算法，根据破损区域的边界信息确定修复优先级，在图像未破损的已知区域（源区域）中寻找与待修复区域的纹理信息最为接近的样本块来填充破损区域（目标区域），获得了较好的图像修复质量。

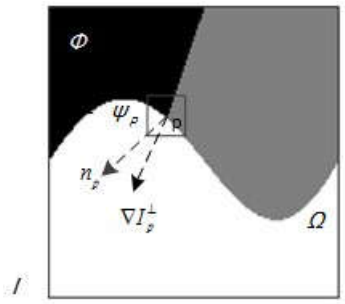
3.1 Criminisi 算法

Criminisi 算法的修复思路是首先确定图像的破损区域，然后通过寻找最相似的样本块来对破损区域进行填充。如图 3-1 所示，Ω为破损区域，Φ为源区域，I 为全图（$I=\Omega\cup\Phi$），$\delta\Omega$为破损区域的边界。Criminisi 算法大致可分为 3 部分。

Step1.计算$\delta\Omega$上的所有像素点的优先级。

为了确定边界$\delta\Omega$上最先需要被修复的样本块ψ_p，Criminisi 算法首先计算$\delta\Omega$边界上所有点的优先级，。公式(3-1) 定义了修复优先级。

图 3-1　Criminisi 算法修复过程



$$P(p)=C(p)\times D(p) \tag{3-1}$$

其中，$C(p)$为信任项，$D(p)$为数据项，具体定义如下：

$$C(p)=\frac{\sum_{q\in\Psi p\cap\Phi}C(q)}{|\Psi p|} \tag{3-2}$$

$$D(p)=\frac{\left|\nabla I_p^{\perp}\times n_p\right|}{\alpha} \tag{3-3}$$

式(3-2)中，ψ_p 为图 3-1 中以 p 点为中心的样本块中，$\Psi p\cap\Phi$ 为 ψ_p 里的已知区域。$|\psi_p|$ 为待填充块 ψ_p 中像素点的个数。$C(q)$ 为点 q 的信任项，开始修复前，位于源区域的点信任项为 1，位于破损区域的点其信任项为 0。$C(p)$ 为点 p 通过 ψ_p 中位于源区域中所有点的信任项求得的信任项，$C(p)$ 越大表示对应的 ψ_p 破损成度越小，越值得被优先修复。

$D(p)$ 为点 p 的数据项。式(3-3)中，$\nabla I_p^{\perp}$ 为点 p 的等照度线的方向和强度向量，n_p 则表示点 p 在边界 $\delta\Omega$ 上的单位外法向量，α 为归一化因子，$\left|\nabla I_p^{\perp}\times n_p\right|$ 表示两个向量的乘积。$D(p)$ 越大表示 p 点存在结构信息的可能越大，以保证图像结构信息被优先修复。

Step2.寻找最佳匹配块 ψ_q。

在找到优先级最高的样本块 ψ_p 之后，需要确定与 ψ_p 最接近的匹配块 ψ_q，然后用 ψ_q 来对 ψ_p 中的破损区域进行填充。其中 ψ_q 的选择条件如下：

$$\Psi q=min_{\Psi q\in\Phi}d\left(\Psi p,\Psi q\right) \tag{3-4}$$

式(3-4)中，$d(\psi_p,\psi_q)$ 表示样本块 ψ_p 中未破损区域的像素点与匹配块 ψ_q 对应区域的像素点之间的欧式距离之和[4]。

Step3.填充待修复区域并更新边界。

在找到与 ψ_p 最接近的样本块 ψ_q 之后，ψ_p 中的破损区域被 ψ_q 中的对应像素填充。在填充完成后，更新破损区域的边界 $\delta\Omega$。以更新后的边界为新的边界，重复 Step1~Step3 的计算直到破损区域被全部修复。

Criminisi 算法的优点是在保证图像纹理的到修复的同时，能够使原图的结构信息也得到修复。但 Criminisi 算法仍旧存在两个问题: 1.在选定填充块的过程中只考虑了匹配块中未破损区域与待填充块对应部分的相似度，没有考虑匹配块中用来填充的部分对修复结果可能产生的影响；2.每次找到用于填充的目标样本块之前，都要进行全图搜索，严重影响了修复速度。

3.2 Anupam 算法

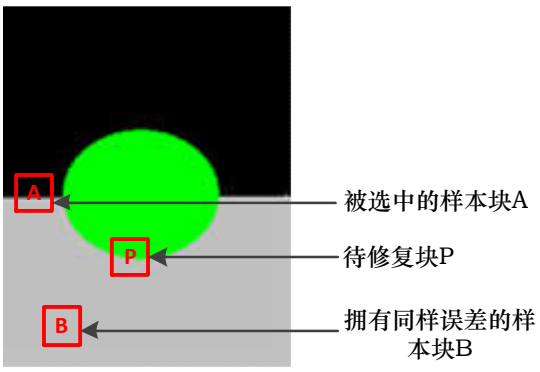


图 3-2 相似块的误匹配示意图

图 3-2 中，在修复破损区域 P 时，Criminisi 算法会先在源区域中寻找与 P 最接近的样本块进行填充。由于搜索顺序的原因，当 A 和 B 同时与 P 具有同样差异时，尽管选用样本块 B 进行修复更符合视觉效果，但 B 后于 A 被搜索到，导致匹配时永远被忽略，造成视觉上的错误。针对这种误匹配情况， Anupam 算法通过比较匹配块中与待修复块缺损区域对应的部分和待修复块中已知部分平均值的均方误差（MSE）来判断填充部分与待修复块已知部分的差异大小来改进匹配结果，当根据式(3-4)再次搜索到距离同样的匹配块时，则继续计算两者的 MSE，并选择 MSE 较小的匹配块进行填充。判断的条件如下所示：

$$M = \frac{\sum f_{p\in\Phi\cap\Psi}}{\#\{p \mid p \in \Phi \cap \Psi\}} \tag{3-5}$$

$$V = \frac{\sum (f_{p\in\Phi-\Psi} - M)^2}{\#\{p \mid p \in \Phi - \Psi\}} \tag{3-6}$$

其中，$f_{p\in\Phi\cap\Psi}$ 表示待修复块源区域中点 p 的颜色，$\#\{p \mid p \in \Phi \cap \Psi\}$ 为待修复块已知区域中像素的数量，M 为待填充区域中已知颜色的平均值，$f_{p\in\Phi-\Psi}$ 表示待修复块破损区域中的点 p 在匹配块中对应位置的颜色，$\#\{p \mid p \in \Phi - \Psi\}$ 表示待修复块破损区域中像素的数量。V 表示由匹配块中对应破损区域位置的颜色与待修复块中已知部分的平均值 M 计算得到的均方误差（MSE）。V 越小说明用于修复的匹配块颜色均值越接近破损块中已知区域的颜色均值。改进后的修复结果如图 3-3 所示。

a.待修复图　　b.Criminisi算法结果　　c.Anupam改进结果

图 3-3 修复结果比较

Anupam 算法在改进了 Criminisi 算法误匹配的同时，还对 Criminisi 算法的修复耗时做出了优化。由于 Criminisi 算法采用的全图匹配方式会搜索所有源区域，而最终选定的匹配块大多分布在距离待填充块较近的区域，因而存在许多多余的搜索操作。Anupam 算法通过把 Criminisi 算法中的全图搜索方式改为局部搜索来减少匹配次数，从而达到加速修复的目的。Anupam 算法的局部搜索范围由坐标(*startX*, *startY*)和(*endX*, *endY*)加以限定，其坐标定义如式 3-7~式 3-10 所示：

$$startX = \max\left(0, p - \frac{n}{2} - c_r - \frac{D_x}{2}\right) \tag{3-7}$$

$$startY = \max\left(0, p - \frac{m}{2} - c_c - \frac{D_y}{2}\right) \tag{3-8}$$

$$endX = \min\left(w, p + \frac{n}{2} + c_r + \frac{D_x}{2}\right) \tag{3-9}$$

$$endY = \min\left(h, p + \frac{m}{2} + c_c + \frac{D_y}{2}\right) \tag{3-10}$$

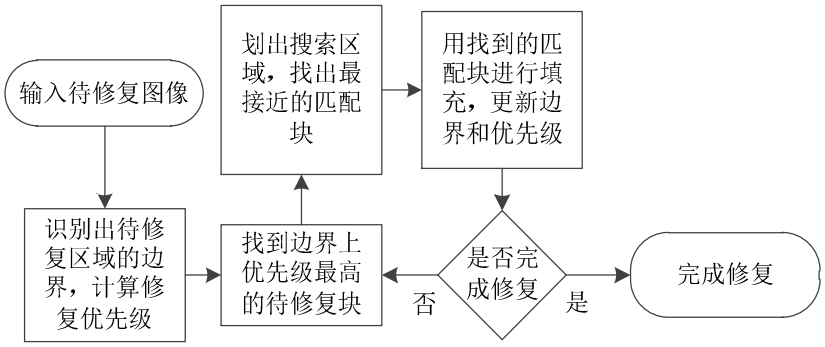
其中，*h* 和 *w* 表示图像的高和宽，*m* 和 *n* 表示样本块中包含的列数和行数，*Dx* 和 *Dy* 为常量，分别表示搜索范围在 *X* 和 *Y* 方向上的最小直径。

与 Criminisi 算法相比，Anupam 算法在修复时考虑了填充部分对修复的影响，所以在修复结构区域时效果更好，同时因为缩小了搜索区域，使得修复耗时得到了显著改善。但是该算法的改进同时也弱化了结构信息对选取的影响，使匹配块的选取更倾向于颜色接近的区域，进而引入了新的误差。另外经过该算法缩小后的搜索区域，依然含有进一步缩小的空间。

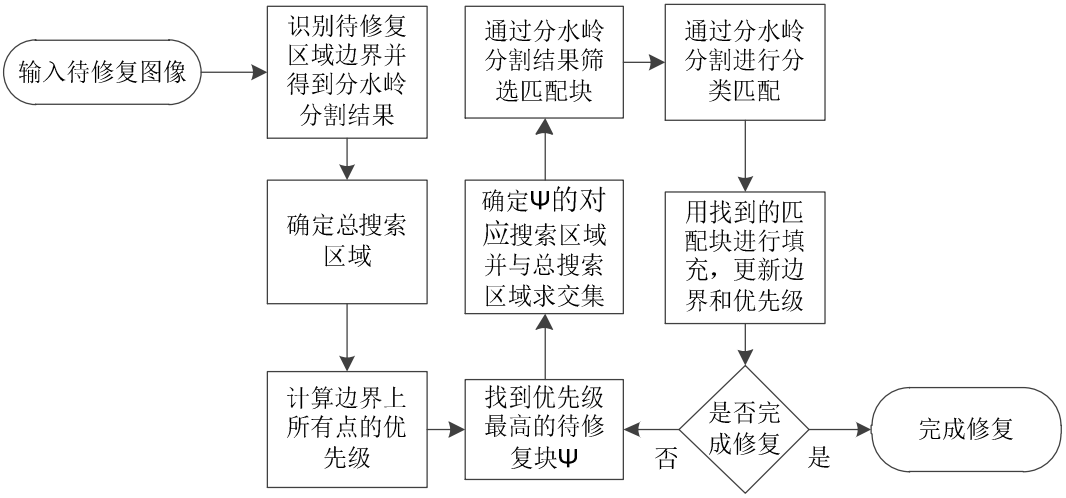
3.3 基于分水岭分割的快速图像修复算法流程

针对 Criminisi 算法在图像修复质量和修复速度上存在的不足，本文算法提出了 3 个方面的改进：①改进的匹配区域搜索，加快了修复速度。②通过分水岭

算法对图像进行分割，判断样本块和待修复块是否属于同一个区域来筛选进行匹配的样本块，从而实现加速。③利用改进 2 中的分水岭分割结果，提出了一种分类修复的匹配方法来改进原算法在修复结构时产生的误差。算法的整体流程如下图所示。



(a) Anupam 算法流程图



(b)本文算法流程图

图 3-4　Anupam 算法和本文改进算法的流程图

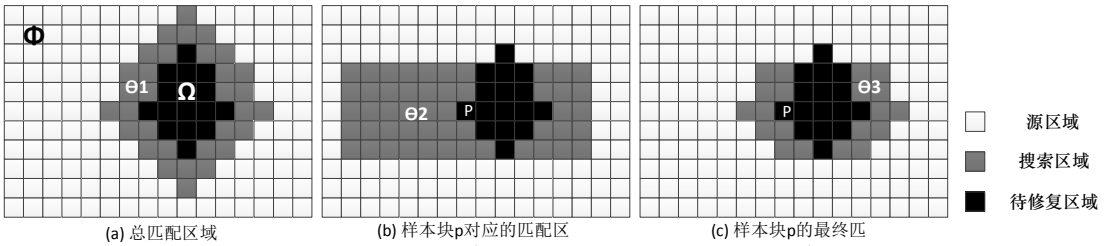
3.3.1　改进的匹配区域搜索算法

由于 Criminisi 算法在修复过程中选中的匹配块大部分集中于修复边界的周围， Anupam 算法通过在待修复块周围划定矩形区域的方式缩小搜索区域，达到提升修复速度的目的。但是 Anupam 算法在划定搜索区域时，为了保证每个待修复块都能找到对应的匹配块，分别使用样本块中心点对应的行和列中的待修复像素数量作为修复区域宽和高的半径。因此对于位于一些图像边缘区域的待修复块，Anupam 算法所采用的矩形框会包含大量无用的区域。

为了修复破损图像，首先需要在图像中寻找与待修复块最接近的样本块。由于 Criminisi 算法采用全图搜索的方式，所以非常耗时。但是根据马尔科夫随机场模型(MRF)的假设，一张图像中某一点的特征与其附近的邻域有关，因此可以

用局部搜索来代替全局搜索。根据文献[25]中的统计结果，相似块大部分位于破损区域附近，因此本文在 Anupam 算法的基础上通过改进匹配区域的方法来加速搜索：

本文首先标记出整个待修复区域的邻域作为总匹配区域$\theta 1$，该步骤仅在开始修复时执行一次，如图 3-5(a)所示，图中白色表示源区域Φ，黑色表示待修复区域Ω，灰色表示总匹配区域$\theta 1$，该区域的像素由$\forall \mathrm{p} \in \phi, \underset{q \in \Omega}{\operatorname{dist}}(p,q) < d$组成，其中$\underset{q \in \Omega}{\operatorname{dist}}(p,q)$表示像素 p 与像素 q 之间的距离，d 表示待修复区域邻域的宽度，经过对大量图像的实验和验证，考虑到本文实现时每个样本块的大小固定在图像面积的 1‰，对应的样本块边长占图像边长的 3%~4%，而根据文献[12]中的结果显示，大部分匹配块位于距离待修复区域 30%-40%边长的邻域内，因此 d 取为 10 个样本块的边长；然后针对不同的待修复块 p，标记出以该样本块为中心的矩形匹配区域作为待修复块 p 对应的匹配区域$\theta 2$，如图 3-5(b)所示，图中的灰色区域表示以 p 点为中心展开的矩形匹配区域，匹配区域大小由(*startX*, *startY*)和(*endX*, *endY*)坐标确定，其定义如式 3-7~式 3-10 所示；最后在标注出每个待修复块 p 对应的匹配区域后，与总匹配区域求交集，得到图 3-5(c)中由图 3-5(a)与图 3-5(b)中灰色区域的相交部分$\theta 3=\theta 1 \cap \theta 2$，即当前为了修复样本块 p 所确定的最终匹配区域。



(a) 总匹配区域　(b) 样本块p对应的匹配区域　(c) 样本块p的最终匹配区域

图 3-5　改进的匹配区域搜索过程

表 3-1 给出了分别使用 Anupam 算法和本节中改进的匹配区域搜索方法进行修复的耗时对比（原图见图 3-6 第一列）。可以看出，本文的改进方法使得 Anupam 算法取得了 40%左右的平均加速比。

表 3-1　修复耗时的比较

测试图片	t_{Anupam}/S	$t_{\text{本文}}$/S	加速比例/%
北极熊 (481×321)	25.558	14.603	42.86
骑车 (481×321)	12.896	7.672	40.51
钟楼 (300×400)	24.402	13.111	46.27
海岛 (325×248)	17.562	12.518	28.72

为了进一步验证改进的匹配区域对修复质量影响，本文引入文献[26]中的感知哈希值（图像指纹）作为相似性的评测标准，针对每个待修复块，通过同时在

Anupam 算法的匹配区域和改进区域中分别搜索匹配块并计算对应的哈希值来进行验证。根据标准，如果感知哈希值不相同的数据位数小于 5，就说明两张图像相似；如果大于 10，就说明这两张图像不同。根据表 3-2 中的结果显示，针对同样的待修复块，两种算法选中的匹配块有高度的相似性。由此可以推测出，改进后的加速算法可以获得与 Anupam 算法基本一致的图像修复质量。图 3-6 显示了经过本节算法改进后的修复效果和原算法的结果对比，从图中可以看出改进前后的修复质量基本没有变化。因此改进的匹配区域搜索算法在不影响修复质量的同时可以大幅提高图像的修复速度。

表 3-2　使用 Anupam 算法的搜索结果与改进的匹配区域搜索结果的相似比例

测试图片	修复次数	感知哈希值位数差异<5 次数	相似比例%
北极熊 (481×321)	234	212	90.6
骑车 (481×321)	114	112	98.2
钟楼 (300×400)	216	214	97.2
海岛 (325×248)	257	248	96.5

3.3.2　利用分水岭分割结果的匹配块筛选算法

由于 Crminisi 算法在搜索匹配块的过程中会遇到大量与待填充块明显不符合的样本块，为了能够快速的分辨出这些样本块，可以先对图像进行分割操作，将颜色相近的区域用同一种颜色表示，然后在匹配的过程中通过判断是否属于同一颜色区域来快速剔除明显不符合的匹配块。文献 [27]提出了一种通过模糊聚类对图像进行分割的算法，但是使用模糊聚类对图像进行分割需要预先设定图像的聚类数目，并且在进行分类的过程中会由于样本集的选取不理想导致分割结果不理想，当存在远离各聚类中心的像素时，由于模糊聚类的约束条件会使得该像素对各聚类的隶属度都相当大，最终影响迭代的结果。此外模糊聚类的计算时间较长，因此不适合用于对速度要求较高的图像修复算法。文献[28]提出了一种通过 kmeans 算法来分割图像的算法，和文献[27]中的改进一样，使用 kmeans 分割图像必须先指定要聚类数目，而且对选取的初值敏感，对于不同的初值，可能产生不同结果。kmeans 同样对于孤立的像素敏感，个别孤立像素的存在会极大影响分类结果[29]。

鉴于上述分析，本文基于分水岭分割的结果，提出了一种新的匹配块筛选方法，通过比较分水岭分割结果来筛选掉不合适的匹配块，进一步加快了修复速度。

分水岭分割是一个迭代标注过程，该过程首先把图像当作测地学中的拓扑地貌，然后从低到高按灰度值对每个像素进行排序，然后对海拔高度上的局部极小值执行膨胀操作，在膨胀的过程中判断和标注出在不同高度处局部极小值的影响域。如图 3-6(a)所示，为了得到两个集水盆间的分水岭，需要对连通区域进行多次膨胀。在每一次膨胀的过程中每个点都必须满足结构化元素的中心只能位于当前连通分量的条件，结果如图中的第一次膨胀边界所示。在下一次膨胀时，由于膨胀边界上的部分像素点造成了不同连通分量的集合聚合，如图中的第二次膨胀边界所示。为了划分出不同的连通分量，这些点就被标记为分水岭。使用分水岭分割图像的好处在于能够在自适应地对微弱边缘做出良好响应的同时，保证被分割的图像边界封闭连续。

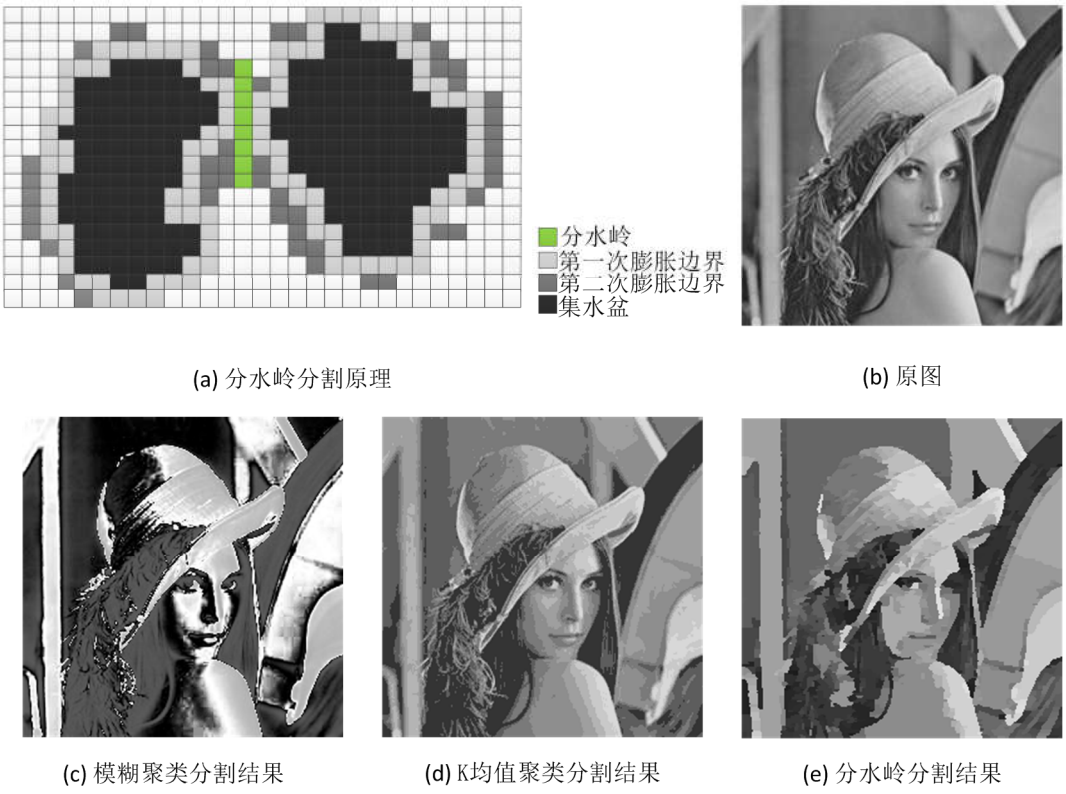


图 3-6 分水岭分割图像示意图

图 3-6(c)~(e)为分别对 LENA 图使用模糊聚类、k 均值聚类和分水岭分割的结果，其中模糊聚类分割耗时 4.891s，k 均值聚类耗时 173ms，而分水岭分割耗时 21.499ms，明显优于其他两种算法。从图 3-6(c)~(e)中可以看到，使用分水岭分割得到的结果在整体颜色区域的识别上有着更优秀的效果，既保证分割区域的连续封闭，又保留了图像大致的结构信息，从而为之后分析图像结构信息提供了便利。为了避免分水岭分割中过分割现象，本文将用于提取图像边缘信息的分割阈值设为 20。

在得到分水岭分割结果后，图像被分割成多种单一色块。之后匹配块的搜索由公式 3-11 加以筛选：

$$\Psi a \text{ for } \exists q: R(p)=R(q)\ ,\ q \in \Psi a,\ p \in \Psi p \cap \Phi \tag{3-11}$$

其中 Ψa 表示搜索中有效的候选样本块， $R(p)$ 表示待修复块已知区域中的 p 点在分割图像中包含的颜色区域集合， $R(q)$ 表示候选的样本块 Ψa 中的 q 点在分割图像中包含的颜色区域集合。通过公式 3-11 的限制，只有搜索到拥有和待匹配块相同颜色区域的样本块才进行误差计算，否则直接忽略该样本块。

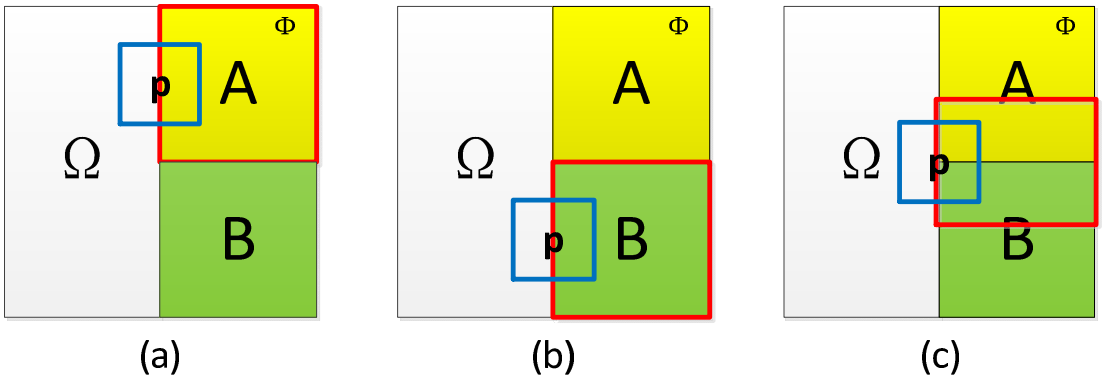


图 3-7 匹配块筛选示意图

筛选过程如图 3-7 所示，Ω 为待修复区域，Φ 为未破损区域，A、B 为不同颜色的匹配区域，中心位于 p 的方框为待修复块。如图 8(a)所示，当待修复块中的颜色只包含区域 A 中的颜色时，只搜索区域 A 中的样本块，区域 B 被忽略；如图 8(b)所示，当待修复块中的颜色只包含区域 B 中的颜色时，只搜索区域 B 中的样本块，区域 A 被忽略；如图 8(c)所示，当待修复块中的颜色同时包含两种以上的颜色时，则只搜索同时覆盖对应颜色区域的样本块。通过筛选匹配区域，匹配操作的算法复杂度由计算欧几里得距离时的 O(8n)下降到遍历样本块中颜色种类的 O(n)(其中 8 表示计算欧式距离时每个点需要执行的算术运算次数，n 表示样本块中的像素个数)，从而大大降低了修复耗时。

筛选匹配区域的实例如图 3-8 所示，图 3-8(c)中的小方框为待修复块，大方框为按照 Anupam 算法划分出的匹配区域，本文采用的基于分水岭分割的匹配块筛选，筛选后的匹配区域被限制在图 3-8(d)中的着色部分，因为这部分的颜色和待修复区域中的颜色相同。通过对比图 3-8(c)和图 3-8(d)，可以看出经过筛选后，需要匹配的区域明显减少。

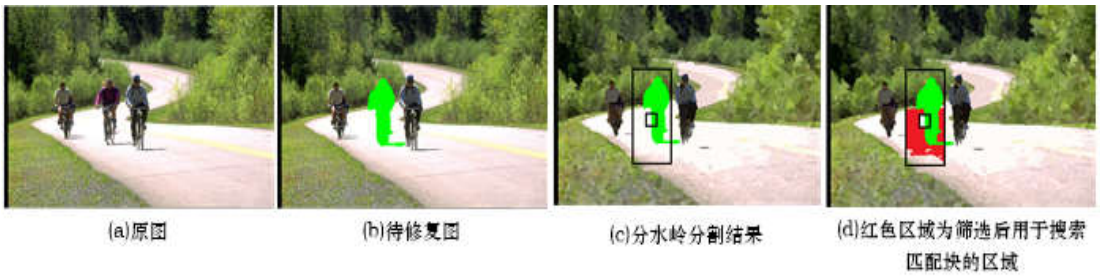


图 3-8　通过分水岭分割筛选搜索区域的过程

表 3-3 给出了修复耗时的比较。可以看出，与 Anupam 算法相比，采用分水岭分割结果筛选匹配块的改进方法，可以取得 25%左右的平均加速比，明显降低了图像修复时间。当破损区域所跨越的源区域中颜色对比越明显，使用这种方法的加速效果越为突出。图 3-10 显示了加速前后图像修复质量的对比结果，可以看出，图像修复的质量基本保持一致。

表 3-3　修复耗时比较

测试图片	*t*Criminisi/s	t_{Anupam}/s	$t_{本文}$/s	加速比/%
北极熊 (481×321)	33.109	25.558	24.204	5.30
骑车 (481×321)	14.047	12.896	6.461	49.90
钟楼 (300×400)	31.440	24.402	20.054	17.82
海岛 (325×248)	34.455	17.562	13.364	23.90

3.3.3　利用分水岭分割结果的分类修复算法

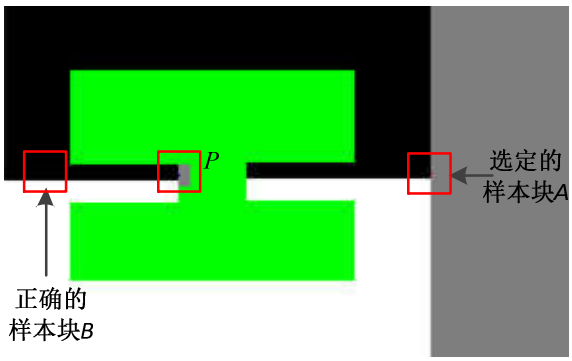


图 3-9 相似块的误匹配示意图

Anupam 算法针对 Criminisi 算法在进行修复时只考虑对已知部分进行比较而造成的误差，提出了一种比较样本块间 MSE 的改进方法，改进效果如图 3-3 所

示。尽管该算法在一定程度上解决了相似块的误匹配问题，但是 Anupam 算法在修复含有结构信息的待填充块时会造成新的误差。产生这种错误主要是因为当样本块间的欧式距离相等时，MSE 就成了选取匹配块的决定因素，从而导致了结构信息在修复时出现断裂。如图 3-9 所示，当同时存在与待填充块 *P* 中已知区域欧式距离相等的样本块 *A* 和样本块 *B* 时，由于样本块 *A* 中存在与待填充块 *P* 中已知部分的平均值一致的部分，使得样本块 *A* 与待填充块 *P* 的 MSE 较小，导致在修复结构区域时会去选取样本块 *A* 来填充，而不是结构上更加一致的样本块 *B*。

在图 3-2 中，出现错误匹配是因为没有考虑待填充块中未知区域的纹理信息，而在图 3-9 中，出现错误匹配是由于忽略了图像的结构信息，因而得到了错误的匹配结果。为了能够快速分辨待填充块是由单一纹理构成的区域还是包含结构变化的区域，并进行针对性修复，本文提出了一种利用分水岭分割结果的分类修复方法。

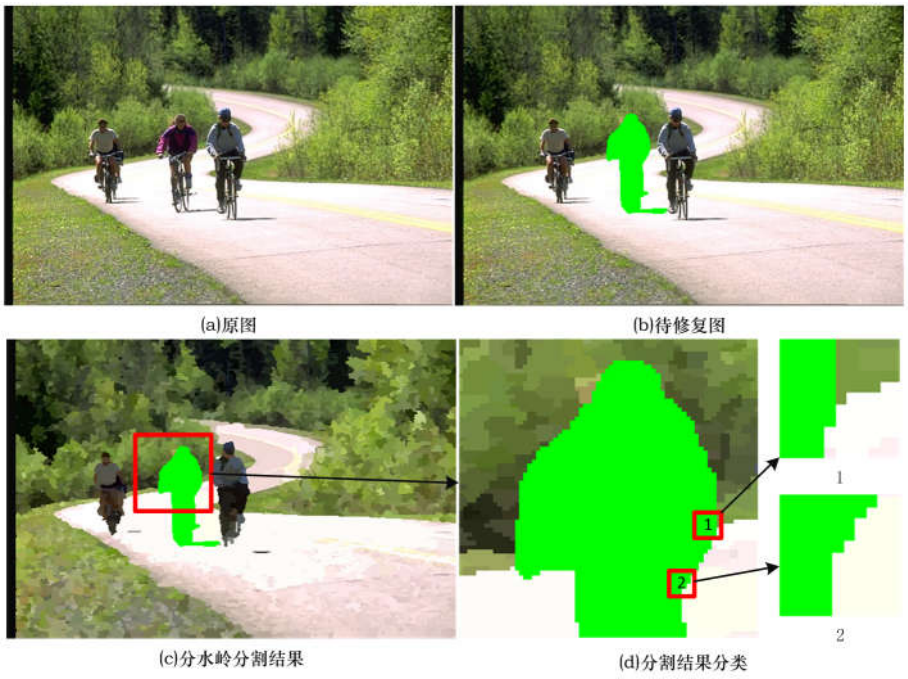


图 3-10 分水岭分割分类结果示意图

本文首先根据之前的分水岭分割算法得到一张分割结果，用来快速划分近似的颜色区域。如图 3-10 所示，图 3-10(a)为原图，图 3-10(b)为待修复图，图 3-10(c)为经过分水岭分割后的效果，图 3-10(d)为待修复区域中的两个样本块。其中，图 3-10(d)中标识为 1 的样本块为结构变化区域，标识为 2 的样本块为单一颜色区域。

在得到如图 3-10(c)所示的分水岭分割结果后，为了分辨出待填充块是由单一颜色构成的纹理区域还是由多种颜色构成的结构区域，首先在分水岭分割结果中找到与待填充块对应的区域并统计其中的颜色数目，当含有多种颜色时，说明该

区域内可能存在结构变化，否则是单一颜色的纹理区域。在识别出结构变化的区域之后，为了降低 Anupam 算法在修复结构时因为均值的引入而造成的误差，本文在公式 3-6 的基础上引入了颜色梯度变化的差异比较来进一步修正匹配的准确度。由于图像的变化情况可以用图像分布的梯度来反应，所以通过比较颜色梯度的变化，能够更加准确地识别出拥有相同结构的区域来进行修复。具体过程如下：

Step1.首先得到分水岭分割结果，寻找优先级最高的待填充块；

Step2.在找到优先级最高的待填充块 ψ_p 后，首先按照公式 3-4 计算出 $d(\psi_p,\psi_q)$，如果同时存在点 q 和 q'使得 $d(\psi_p,\psi_q)=d(\psi_p,\psi_{q'})$，则使用公式 3-12 和公式 3-13 来进一步分类寻找最匹配的填充块 ψ_q

$$\Psi q=argmin\,d'\left(\Psi p\ \Psi q\right) \tag{3-12}$$

$$d'\left(\Psi p,\Psi q\right)=\begin{cases} V(\Psi p,\Psi q) & \Psi p\text{只有一种颜色(1)} \\ g(\Psi p,\Psi q) & \Psi p\text{存在多种颜色(2)} \end{cases} \tag{3-13}$$

式 3-13 中 $d'(\Psi p,\Psi q)$ 为分类匹配后得到的差异结果。为了对填充块 ψ_p 进行有针对性的匹配，本文根据待填充块 ψ_p 在分水岭分割结果中包含的颜色数量来选择相应的匹配方法，具体可分为下列两种情况：

a. 如图 3-10(d)的样本块 2，当 ψ_p 在分水岭分割结果中只包含一种颜色时，表示 ψ_p 处于单一的纹理区域中。在计算匹配块间距离时，引入 $V\left(\Psi\mathrm{p},\Psi\mathrm{q}\right)$ 做修正。$V\left(\Psi\mathrm{p},\Psi\mathrm{q}\right)$ 为匹配块间的 MSE，计算方式如公式 3-6 所示。$V\left(\Psi\mathrm{p},\Psi\mathrm{q}\right)$ 越小，说明两个样本块颜色的平均值越接近，越有可能属于同一区域。

b. 如图 3-10(d)的样本块 1，当包含多种颜色时，表示在 ψ_p 中可能存在结构变化。在计算匹配块间差异时，引入 $g\left(\Psi p,\Psi q\right)$ 来修正结构区域的修复结果。$g\left(\Psi p,\Psi q\right)$ 为匹配块和待填充块中对应的已知部分在颜色梯度上差值的平方和，$g\left(\Psi p,\Psi q\right)$ 值越小，说明两个样本块间的结构越接近，反之，差异越大说明两个样本块在结构上差异越大。$g\left(\Psi p,\Psi q\right)$ 的计算方式如公式 3-14 所示：

$$g\left(\Psi p,\Psi q\right)=\sum_{p\in\Phi\cap\Psi p}((\mathrm{R}_p-\mathrm{R}_{p'})^2+(G_p-G_{p'})^2+(B_p-B_{p'})^2) \tag{3-14}$$

其中，R_p、G_p、B_p 分别表示在 RGB 颜色空间中利用 Sobel 算子求得的点 p 处的颜色梯度，p'为待修复块中的点 p 在匹配块 ψ_q 中对应的像素点。

Step3.在找到最合适的匹配块后，用匹配块中的已知部分对待填充块中缺失的部分进行填充，同时依照待填充块和选中的匹配块的坐标，对分水岭分割图像中的对应部分进行相应的填充。之后更新边界$\delta\Omega$的优先级并重复 Step1~ Step3 来继续寻找优先级最高的待填充块进行修复。

该方法在修复图像的过程中，既保证了同一纹理区域在寻找匹配块时的准确性，同时也保证了修复结构区域时，图像缺失的结构信息得到延伸。修复结果的对比如图 3-11 所示，图 3-11(d)为 Anupam 算法的修复结果，方框圈出的结构区域出现了明显的误差。而使用本节算法的结果如图 3-11(e)所示，由于在修复时采取了分类修复方法，因此结构区域得到了更好的修复。

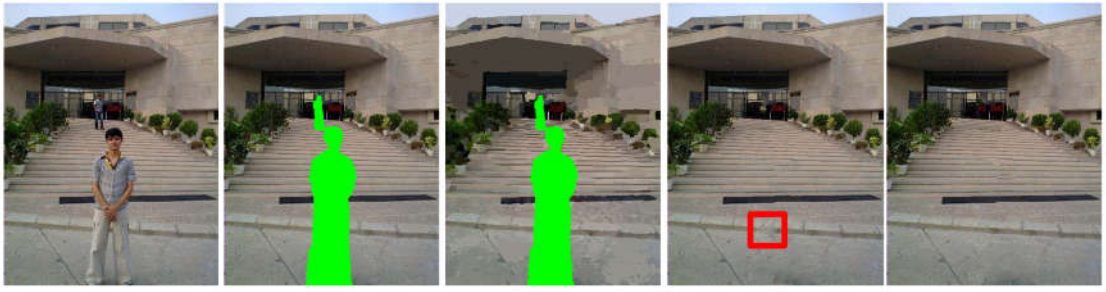


图 3-11　修复结果对比

3.4 实验结果分析

本文所用算法工作均在如下硬件测试平台完成：操作系统为 ubuntu13.04，CPU 为 AMD A4-5000 APU with Radeon(TM) HD Graphics，主频为 1.5GHz。实验中用到的测试图片来自于 berkeley 的 BSR_bsds500 测试集和参考文献。

3.4.1　算法耗时比较

表 3-4 为 Criminisi 算法、Anupam 算法和本文算法的整体耗时的结果比较。表中第 4 列给出了本文改进算法的整体修复耗时。可以看出，经过本文算法的改进后的修复速度平均提高了 40%左右。第 3.3.3 节中的分类修复算法为更加准确地识别出拥有相同结构的区域来进行后续的修复，因此增加了分类匹配和梯度差异计算，导致在原算法基础上增加了部分计算耗时（如表 3-4 中第 4 列所示）。

表 3-4　修复耗时比较

测试图片	Criminisi 算法耗时/s	Anupam 算法耗时/s	经 3.3.3 改进耗时/s	整体改进算法耗时/s	加速比/%

骑车(481×321)	14.047	12.896	15.656	7.274	43.595
钟楼(300×400)	31.440	24.402	37.739	13.104	46.299
蹦极(182×272)	23.766	17.562	27.905	13.214	24.758
印度人(324×432)	131.222	56.707	103.333	28.916	49.008

3.4.2 主观视觉比较

图 3-14 为修复结果对比，从图 3-12(a)和图 3-12(b)中可以看出在修复有明显边界区域区分的修复区域时，本文算法在加快修复速度的同时保持了原算法的修复效果。此外，在修复图 3-12(c)中拥有复杂背景的破损区域时，本文算法改进了 Anupam 算法在结构区域处的修复误差（红框圈出部分）。在修复图 3-12(d)中边界模糊的破损区域时，本文算法同样改进了结构区域处的修复误差（红框圈出部分）。从修复结果中可以看出，经过本文算法修复的破损图片在主观视觉效果上更加自然平滑，图像修复的质量更高。

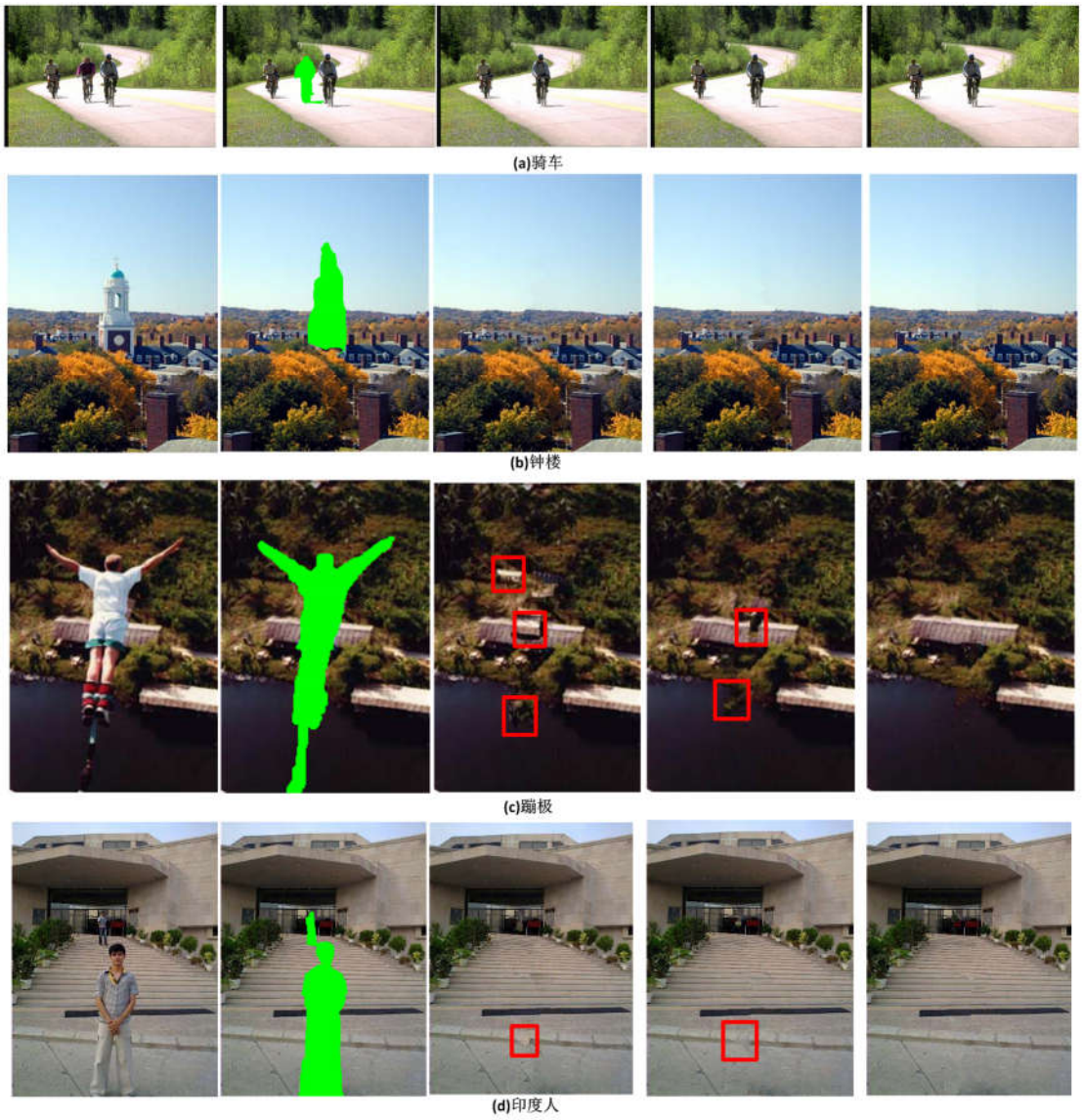


图 3-12 修复结果对比，从左至右依次为原图、待修复图、本文实现的 Criminisi 算法结果、本文实现的 Anupam 算法结果、改进后的修复结果

3.4.3 客观标准评价

为了更加客观的评价图像的修复结果，本文使用文献[15]提到的峰值信噪比 PSNR 值（图像在每个颜色通道上的峰值与噪声方差之比）和 MSE 来衡量图像的修复质量。图 3-13 显示了分别使用 Anupam 算法和本文算法对加入随机划痕后的图片进行修复的对比结果。分别对两种算法结果的 PSNR 和 MSE 值进行计算和比较，结果如表 3-5 和表 3-6 所示。

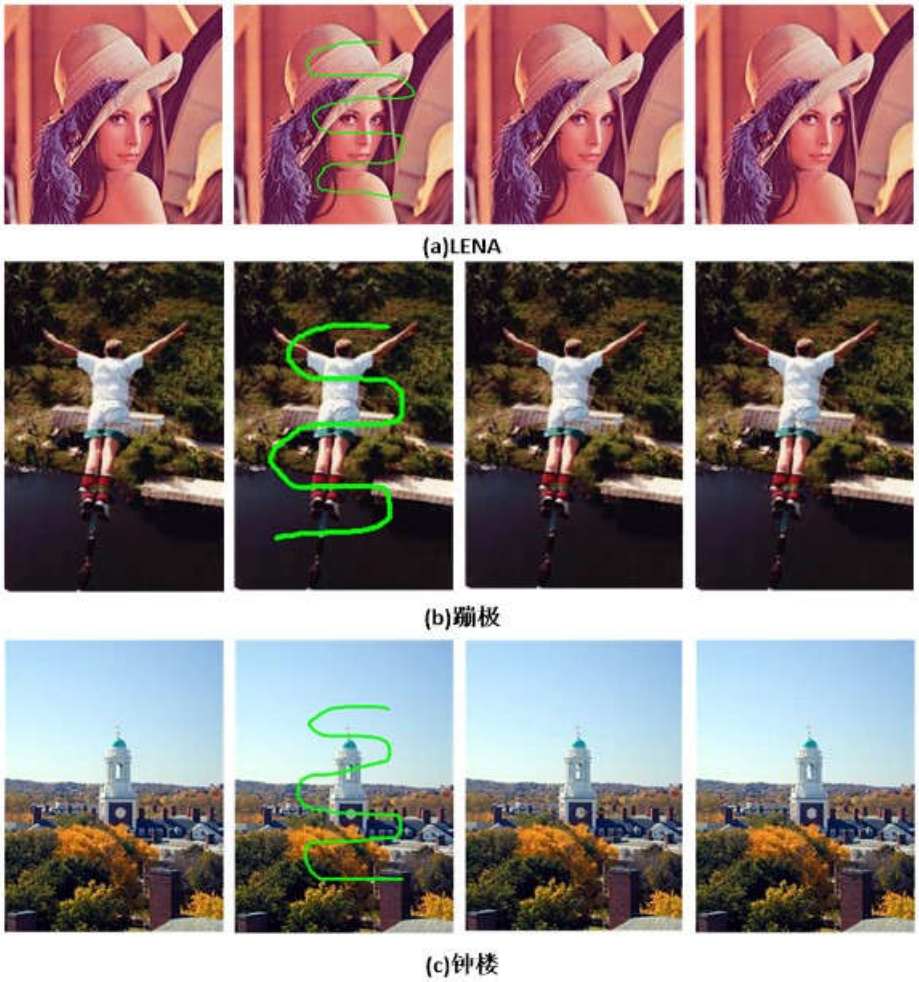


图 3-13 修复结果对比，从左至右依次为原图、待修复图、本文实现的 Anupam 算法结果、本文算法结果

表 3-5　Anupam 算法和本文算法的 PSNR 结果比较

测试图片	红色通道		绿色通道		蓝色通道	
	Anupam 算法	本文算法	Anupam 算法	本文算法	Anupam 算法	本文算法
LENA(512×512)	44.82410	47.84083	44.61742	48.08182	45.68211	48.43725
蹦极 (182×272)	35.71787	35.73580	34.98953	35.57598	34.42000	35.71291
钟楼 (300×400)	38.33161	39.52731	39.65222	40.70267	40.65593	41.39761

表 3-6　Anupam 算法和本文算法的 MSE 结果比较

测试图片	红色通道		绿色通道		蓝色通道	
	Anupam 算法	本文算法	Anupam 算法	本文算法	Anupam 算法	本文算法
LENA(512×512)	2.14126	1.06905	2.24563	1.01134	1.75740	0.93187
蹦极 (182×272)	17.42983	17.35801	20.61233	18.00871	23.50067	17.44975

钟楼 (300×400)	9.54815	7.25019	7.04463	5.53112	5.59097	4.71323

PSNR 越高、MSE 越小，说明修复图和原图间的差异越小，相应的图像修复的质量也越好。由表 3-5 和表 3-6 可以看出，本文算法的修复结果优于 Anupam 算法。

3.5 本章小结

本章首先介绍了基于样本块的图像修复算法原理，并详细分析了现有算法中存在的缺陷。然后详细介绍了本文提出的一种基于分水岭分割的图像修复算法，并对修复结果进行了对比和分析。结果显示，本文的改进算法在修复速度和修复质量两方面有着更优秀的效果。

第四章 GPU 通用计算和 CUDA 介绍

4.1 GPU 简介

随着人们对计算机硬件（如 CPU、主存）的性能要求越来越高，各种处理器的发展速度也越来越快。但就目前技术而言，现代处理器的问题之一是已经接近时钟速度的极限，由于电子设备和芯片的封装过程中功耗墙的存在，会导致在提升性能的过程中出现电力浪费或者无法有效冷却设备的情况出现。

为了进一步提高处理器的性能，制造商门纷纷将改进方向从提升时钟频率转移向了增加处理器中的核数，现在已经陆续出现了双核、4 核、8 核乃至 32 核的处理器。然而，这种方法仍存在一定的缺陷，当使用多核处理器解决问题时，编程人员需要考虑多个核心之间的程序通信和资源共享问题。当核数较多时，为了支持多核运算会导致软件程序的开发周期变长和开发成本提升，进而迫使软件开发者只开发支持单核运行的程序，大大浪费了处理器原本的性能。

随着 GPU 设备的性能越来越强大，CPU 与 GPU 之间的差距也越来越大，为了能使得 GPU 能够获得一个进入主流的契机，英伟达公司于 2007 年推出了全称为 Compute Unified Device Architecture 的统一计算架构 CUDA。

CUDA 作为一种通用的并行计算平台和编程模型，允许开发人员同时在 CPU 和 GPU 上利用一种类 C 环境开发通用计算程序，这样既可以更加高效的利用硬件进行计算，又加快了开发效率。最终使得编程人员可以通过 CPU 来控制 GPU 中的多线程，然后由 GPU 内部的调度器来协调不同的资源来完成任务。随着 CUDA 一起出现的，是一些专门用于科学计算的计算卡，这些卡可以安装在常规的 PC 上，也可以被安装在专门的服务器机架上它们可以为科学计算提供强大的加速性能。可以这么说，CUDA 和 GPU 的出现，正在改变着高性能计算领域的形式。

4.2 CUDA 结构

并行程序通常与硬件紧密相连，引入并行程序的目的是为了获得更高的性能，但是需要以牺牲软件的可移植性为代价，这样会导致在下一代并行计算机中，原先为计算某个特性编写的特定并行程序失效，进而重新编写所有相关程序。显而易见的一点是，如今的计算机技术正在朝着多核处理器的方向前进。如何处理好多核中的线程管理正成为程序编写中的一个关键问题。线程作为一个单独的执行过程在每个程序的进程中执行，并且随时可以根据当前 CPU 中任务的执行情况被调整。与单核处理器中相同的一点是，在多核处理器中，操作系统分配在每个

核心上的任务都是分时和轮流运行的，每个任务只占用处理器运行时间的一小部分，从而实现数量多余物理核数的线程任务同时运行。

因而在 CPU 中，随着线程数的增加，操作系统需要对一组寄存器频繁的进行上下文切换操作来保证所有任务的合理运行。然而上下文切换是一项非常耗时的操作，通常需要上千个时钟周期才能完成一次，所以 CPU 中应用程序的线程数受到了极大的制约。一般情况下，CPU 拥有的活动线程数不会超过这个 CPU 中的物理核心数量的两倍。

为了减少线程间和线程块间的通信，CUDA 利用了片上资源来支持线程间的通信，然后通过按顺序调用多个内核程序来实现块间通信，但是整个块间通信的过程需要用到片外的全局内存。尽管可以通过原子操作来实现对全局内存的访问，这种方法在实际的操作过程中还是会受到一定的限制。

CUDA 作为 C 语言的一个扩展，开发者可以使用类似 C 语言的开发方式来对 CUDA 程序进行开发。一个 CUDA 应用主要分为两部分：主机端和设备端。主机端 host 负责执行 CPU 中的代码，设备端又称为 kernel，负责执行 GPU 中的代码。一般情况下，CUDA 会先执行主机端的程序，当执行到需要 GPU 来运算的部分时，主机端会先根据 GPU 中需要的显存进行相应的分配，然后设备端将需要执行的数据从 CPU 中复制到 GPU 中，并在运算完成之后，发送回 CPU 的内存中继续运行。

在 CUDA 中最基本的运行单位是线程，每个线程都有对应的 Id， Id 可以通过 CUDA 的内建变量 threadIdx 获取。多个线程（16 的倍数）会被分配在一起组成一个线程块，threadIdx 可以是 1~3 维的，这样可以分别用来索引 1~3 维的线程块。这种索引方式使得开发者对各种向量、矩阵的操作更加方便和直观。每个线程块内部的线程共用一块显存，因此可以快速的进行同步和共享内存操作。不同的线程块又可以组成线程网格，每个线程网格中的线程块执行同一个程序的核函数。尽管线程网格内不能像线程块内那样共享内存，但是通过以线程块为单位对资源进行分配能照顾到运行时 GPU 的实际进程数量。当 GPU 负载较高时，不同的线程块不会因为无法分配到线程资源而被抛弃，而是采用顺序等待执行的方式处理，这样就保证了所有任务都能得到处理。

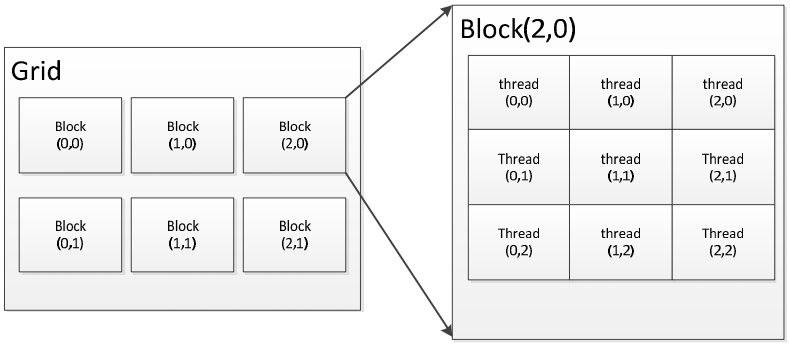


图 4-1 CUDA 编程结构

图 4-1 显示了 CUDA 在运行过程中各种资源间的对应关系。为了获取线程、线程块和线程网格等资源的索引和大小等信息，CUDA 提供了一批内建变量用于开发者直接调用，如下所示：

1. gridDim 为线程网格的尺寸，gridDim.x、gridDim.y、gridDim.z 分别对应为第一维至第三维的大小。在图 4-1 中 gridDim.x、gridDim.y、gridDim.z 分别取值 3、2、1。
2. blockIdx 为线程块索引。在图 x 中，Block（0,0）对应的第一维索引 blockIdx.x 为 0，第二维索引 blockIdx.y 为 0。
3. blockDim 代表了线程块的大小，blockDIm.x、blockDIm.y 分别对应为第一维和第二维的大小。在图 4-1 中，Block（1,1）包含了 4 * 3 个线程，因此 blockDIm.x、blockDIm.y 分别取值 4、3。
4. threadIdx 为线程索引，假设存在一个线程的 threadIdx 为(x, y, z)，CUDA 中的线程和线程索引关系如下所示：

    1、假设索引的是一维线程块，线程块大小用（L1），则 ID 为 x

    2、假设索引的是二维线程块，线程块大小用（L1，L2），则 ID 为 x + y * L1

    3、假设索引的是二维线程块，线程块大小用（L1，L2，L3），则 ID 为 x + y * L1+ z * L1* L2

CUDA 中的核函数的调用可以用 kernel<<<X，Y>>>表示，其中 X 表示了线程网格的大小，用 dim3 表示（dim3 相当于 3 个 unsighed int 定义的结构体），dim3 可以根据实际需要选择一维到三维大小。Y 则表示了线程块的大小，同样可以用 dim3 类型表示。最终的调用形式：

函数名称<<<block 大小，thread 大小，共享内存大小（可以省略）>>>（参数）；

4.3 JCUDA——面向 Cuda 的 JAVA 接口

由于 Storm 的应用开发依赖于 Java，为了能使 Storm 和 GPU 能够实现直接通信，本文在实现的过程中使用了 JCuda 作为 Storm 和 GPU 作为消息中间件。JCuda 作为一个提供 CUDA 绑定的 java 库集合，提供了两种不同的 API：运行时 API 和驱动 API。CUDA 与 JCUDA 最大的不同在于 JCuda 对 CUDA 内核进行操作的方式。

在 CUDA 中，核函数的定义和编译都是和 C 文件包含在一起的。其中源代码通过 NVIDIA CUDA 的编译器 NVCC 来编译。这个编译器通过使用另一个 C 编译器（GCC 或者 VS 编译器）来编译源代码中普通的 C 语言部分，并将 CUDA

的核函数定义和调用作为特殊部分在编译时区分对待。NVCC 的编译结果通常包含一个由整个程序构成的可执行文件。

显而易见的一点是，NVCC 无法直接编译 JAVA 代码。核函数调用的语法无法直接在 Java 中执行，编译完成后也无法只有一个单一的可执行文件。因此，使用 JCuda 运行时 API 无法直接调用 CUDA 核函数，而是需要调用驱动 API 才能实现。

JCuda 运行时 API 主要为了 Java 和 CUDA 运行时的库集合的通信提供便利。通过运行时 API，程序编写者并不需要自己去编写 CUDA 核函数而只需要知道如何使用运行时 API 提供的通信方法[30]。

下面的步骤说明了如何搭建一个 JCUDA 项目

（1） 创建核函数

JCUDA 中的核函数和 CUDA 中的写法类似，并且会存放在一个单独的文件中。通过指定核函数的名称，程序可以通过调用驱动 API 的方式来执行对应的操作。下面是一个核函数示例，该函数实现了一个简单的向量加法。

```
testGPU.cu

extern "C"
__global__ void add(int n, int *A, int *B, int *O)
{
int i = blockIdx.x * blockDim.x + threadIdx.x;
if (i<n)
{
O[i] = A[i] + B[i];
}
}
```

（2） 编译核函数

核函数的编译通过 NVCC 编译器执行。编译完成后会生成一个可用于被驱动 API 加载和执行的文件。具体的编译方式可以分为以下两种：

1) 生成一个包含特殊格式汇编源码的可读 PTX 文件
2) 生成一个已被编译完成并可以被 GPU 直接加载的 CUBIN 文件

由于编译 CUBIN 文件需要事先知道 GPU 的型号和架构等设备信息，并且一旦 CUBIN 文件编译完成就只能适用于对应的机器而不能用于其他的 GPU。而 PTX 文件可以在运行时针对不同的机器自动编译，

所以本文采用编译 PTX 文件的方式来执行 CUDA 的调用。一个 PTX 文件的编译过程如下：

```
nvcc -ptx testGPU.cu -o testGPU.ptx
```

（3） 载入和执行核函数

在 JCUDA 中通过 PTX 文件调用一个核函数的方式和 CUDA 中类似，下面展示了一个简单的示例：

首先载入 PTX 文件，并得到一个指向对应核函数的指针：

```
CUmodule M= new CUmodule();
cuModuleLoad(M, " testGPU.ptx");
CUfunction F= new CUfunction();
cuModuleGetFunction(F, M, "add");
```

在获取到核函数的指针之后，为了调用核函数，还需要构造一个指向输入参数的数据指针。通过该数据指针和核函数指针，调用 cuLaunchKernel 函数来执行 GPU 中的运算操作。

```
Pointer K= Pointer.to(
Pointer.to(new int[]{numElements}),
Pointer.to(A),
Pointer.to(B),
Pointer.to(O)
);
cuLaunchKernel(function,
gridSizeX,  1, 1,      // Grid dimension
blockSizeX, 1, 1,      // Block dimension
0, null,               // Shared memory size and stream
K, null // Kernel- and extra parameters
);
```

（4） 测试函数

```
public class testGPU
{
    public static void main(String args[])
    {
        Pointer pointer = new Pointer();
        JCuda.cudaMalloc(pointer, 4);
        ……
//CPU操作
……
        JCuda.cudaFree(pointer);
    }
}
```

使用下列命令对文件进行编译：

```
On Windows:
javac -cp ".:jcuda.jar" testGPU.java
On Linux:
javac -cp ".:jcuda.jar" testGPU.java
```

编译完成后会在同一目录下生成" testGPU.class"文件

使用下列命令对文件进行执行：

```
On Windows:
java -cp ".:jcuda.jar" testGPU
On Linux:
java -cp ".:jcuda.jar" testGPU
```

这条命令会执行程序里函数指针所创建出的内容。由于 Storm 中的 Topology 均由 Java 编写，因此通过这种方式 Storm 可以将本身运行在 CPU 中的代码移植到 GPU 中并行执行。

4.4 基于 GPU 的图像修复算法的并行化改进

由于 criminisi 算法在寻找匹配块的过程中需要与以图片中已知区域的每个像素点为中心展开的样本块计算欧式距离，并找出距离最近的一个样本块作为匹配块来修复破损区域。之后更新图像破损区域边界，继续重复搜索和修复操作直到所有破损区域被填补。具体的搜索过程如图 4-2 所示。

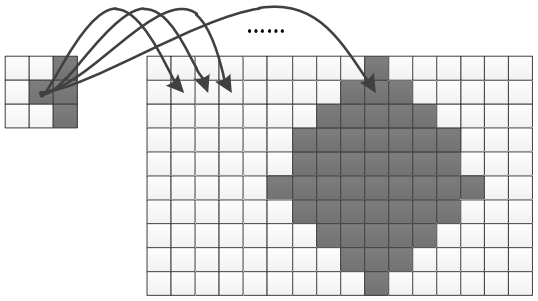


图 4-2 匹配块匹配过程

为了加速这个匹配过程，本文通过分割图片的方式将匹配块的寻找过程平均分配给 gpu 上的众多线程来同时进行寻找，以此来加快修复速度。具体的分配方式如下图所示。

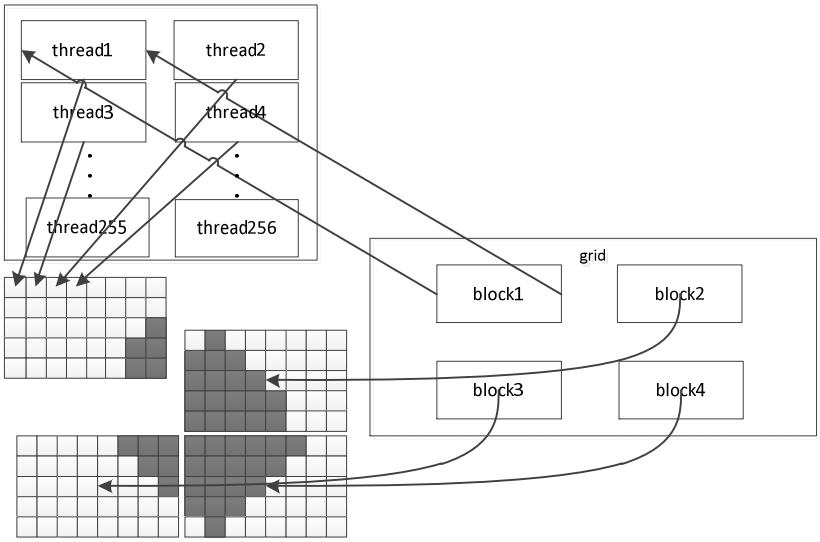


图 4-3 GPU 加速匹配块搜索原理示意图

（1）首先将 cpu 内存中的图像拷贝到 gpu 中。

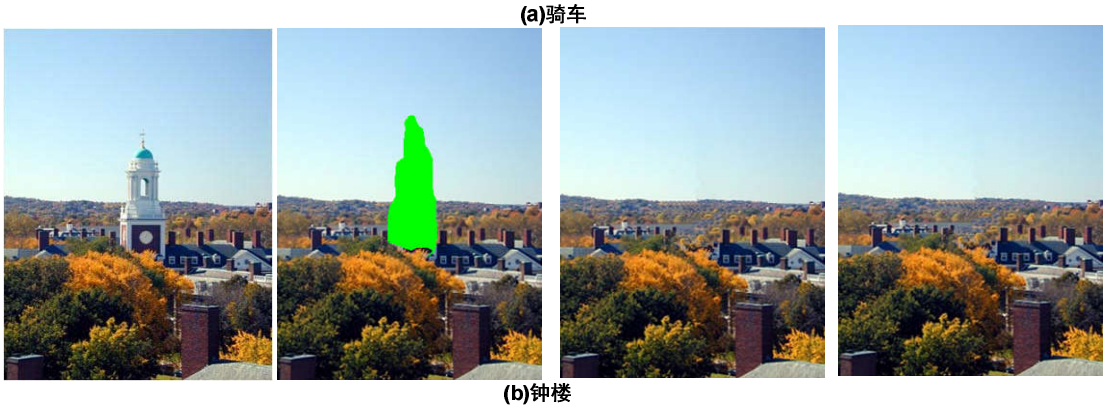
（2）然后根据图像的大小分配 block 和 thread 的数量，实验取去 thread 数量为 256，block 的数量为 M/256，其中 M 为图像的大小。

（3）分配完 block 和 thread 后开始以这个点作为样本块的中心与待修复块计算欧式距离。

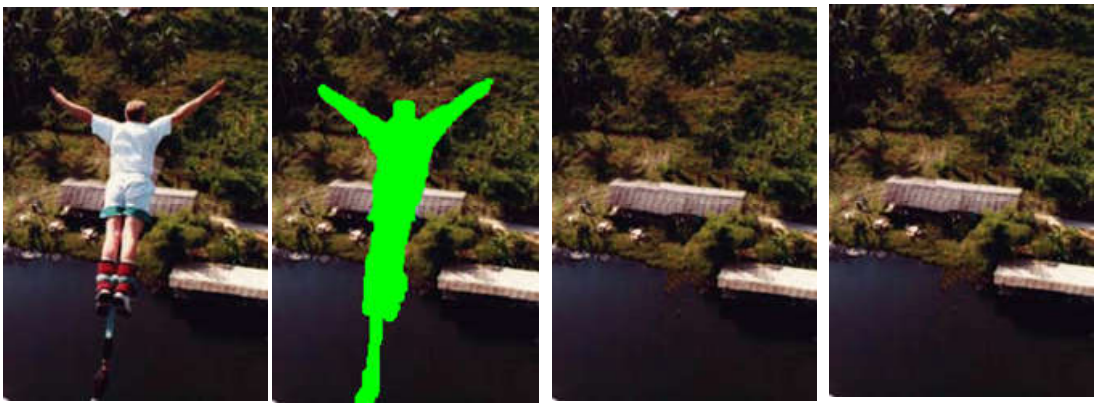
（4）在显存中声明一个共享变量用于保存当前最接近待修复块的中心位置像素的坐标，如果有线程发现比这个样本块更接近待修复块的坐标，随即更新该变量。

（5）当所有线程都完成匹配之后，共享变量被从 gpu 中拷贝回 cpu 的内存中用于修复图像，之后图像破损区域被更新，继续进行匹配和修复操作。

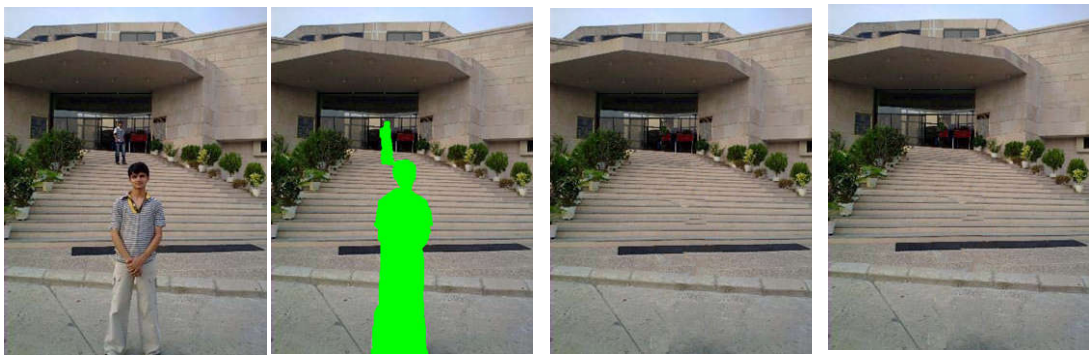
(a)骑车



(b)钟楼



(c)蹦极



(d)印度人

图 4-4 修复结果对比，从左至右依次为原图、待修复图、CPU 处理结果，GPU 处理结果

图 4-4 显示了使用 GTX970 型 GPU 和 CPU 的修复效果对比，具体的测试结果如下表所示，根据结果可以看出，经过 gpu 的加速，修复耗时普遍缩短了 5 倍以上：

表 4-1 消耗时间对比结果

测试图片	CPU 算法耗时/s	GPU 算法耗时/s	MSE
骑车 (481×321)	7.274	1.36	0

钟楼 (300×400)	13.104	1.53	0
蹦极 (182×272)	13.214	1.97	0
印度人 (324×432)	28.916	3.03	0

4.5 本章小结

本章首先对 GPU 计算和基于 CUDA 的编程模型进行了详细的介绍，然后介绍了 CUDA 中一种面向 JAVA 编程的库文件集合 JCUDA。最后本文基于 CUDA 对第三章中的图像修复算法进行了并行化改进，并给出了实验结果进行对比。

第五章 基于云的分布式图像修复系统的设计与实现

5.1 基于 Storm 和 GPU 的分布式视频处理系统框架

为了能够使搭建的云计算平台能够分别对实时视频流和离线图像文件进行处理，本文首先使用 Kafka 分布式消息系统作为消息的获取和分发节点，通过 producer 来执行数据的获取，然后通过 broker 对数据进行缓存，最后 Storm 中的 spout 节点能够通过 Kafka 种的 consumer 获取到对应的信息，从而进行后续的图像处理操作。在 Storm 中，流数据按照 topology 定义的结构传递，但是每个 bolt 节点对应的操作由在 cpu 中的进行改为拷贝数据到 gpu 中进行计算，通过 gpu 强大并行计算能力达到加速运算降低延时的目的。最后 Storm 将处理完的视频流发送到 RTMP 服务器，使得客户端能够浏览最终的处理效果。

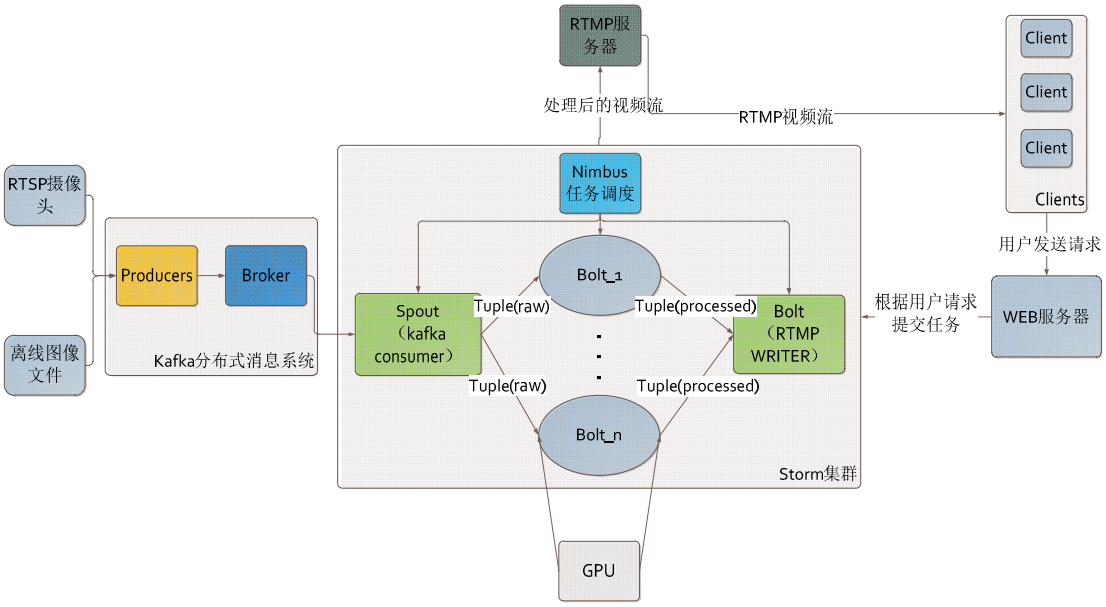


图 5-1 基于云的视频处理系统框架示意图

框架解读如下：

1) 通过 Kafka 分类读取 RTSP 摄像头的实时视频流和离线图像数据并缓存在 broker 中；
2) Spout 节点通过 consumer 从 broker 中读取缓存数据，然后发送到对应的 bolt 节点
3) bolt 节点接收到数据后，通过 JCUDA 实现调用 gpu 进行图像处理操作 ,每个 gpu 根据需要负责一至多个 bolt 节点上的操作；

4) Storm 经过一系列的图像处理操作后将处理后视频流发送给 RTMP 服务器；
5) RTMP 服务器作为视频的网页点播服务器；
6) 用户借助浏览器查看原生视频流和处理后的视频流；

5.2 图像采集和分发模块

由于 Storm 设计初衷是作为基于内存运算的快速实时流计算解决方案，因此任何实现任何数据的存储，需要依赖外部模块来提供消息的输入。Kafka 作为一种基于发布/订阅模式的分布式消息发布系统提供了相应解决方案。通过 Kafka 作为采集和分发输入图像模块，可以在缓存在线视频流和储存离线图像文件的同时，作为消息队列向 Storm 进行数据的分发和管理。基于 Kafka 的消息发布系统架构如下图所示

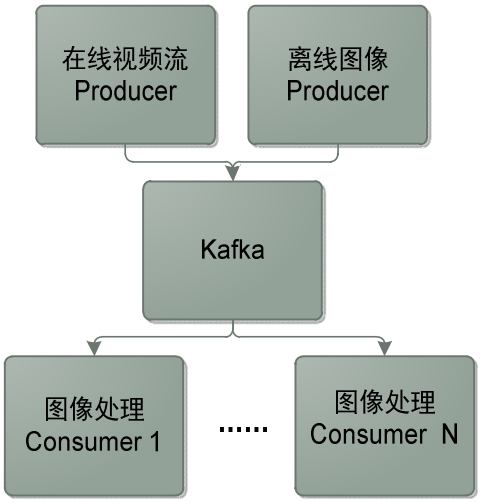


图 5-2 图像分发系统架构示意图

5.2.1 Producer 模块

为了能够处理不同形式的输入，本文实现了两种数据读入方式，分别对在线视频流进行读入和缓存的在线处理模式和对离线图像和视频文件进行存储和发布的离线处理模式。

在线处理模式

为了对在线的视频流进行缓存，本文实现了 StreamProducer 类用于读取指定的视频源并写入 Kafka 缓存，具体步骤如下图所示。

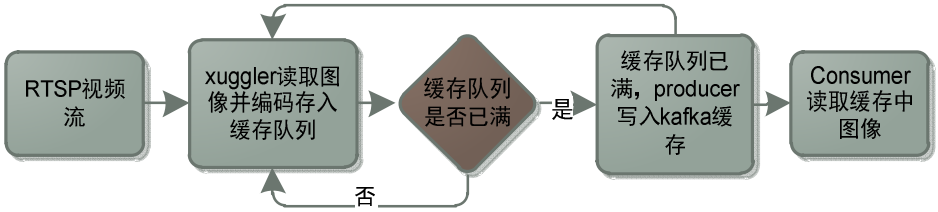


图 5-3 在线图像处理流程图

StreamProducer 首先根据传入的 rtsp 视频流地址和对应的 topics 建立连接，然后通过 Xuggle 作为解码器，读取每一帧图像，并存入一个缓存图像的队列中。当该队列被填满时，FrameProducer 类将缓存队列中的图像一次读出并清空队列，然后通过 produce()函数将每帧图像写入 Kafka 缓存中，对应的 Consumer 通过 topics 找到对应消息源中得数据，其中 produce()的代码示例如下。

```
@Override
    public void produce(Frame value) throws ExecutionException,
            InterruptedException {
        byte[] bytes = frameSerializer.toBytes(value);
        logger.debug("The bytes of Frame is " + bytes.length);
        ProducerRecord<String, byte[]> record = null;
        if (assignPartition) {
            record = new ProducerRecord<String, byte[]>(topic, bytes);
        } else {
            record = new ProducerRecord<String, byte[]>(topic, partition,
                    Integer.toString(partition), bytes);
        }
        this.produce(record);
    }
```

离线处理模式

为了对存储的离线图像和视频文件进行分发，本文实现了 ImagesProducer 类用于读取指定文件夹下的图像和视频文件。读取过程如下图所示，首先根据输入的离线图像文件位置获取需要处理的文件，然后通过 FileReader 读入图像文件，并用 opencv 对图像进行编码，之后通过 FrameProducer 类中的 produce()函数将编码过后的图片写入 Kafka 缓存中。

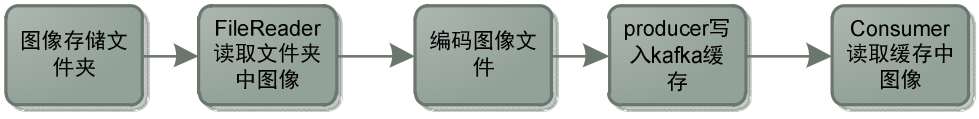


图 5-4 离线图像处理流程图

5.2.2 Consumer 模块

为了让 Storm 从 Kafka 集群中获取缓冲数据，需要向项目中倒入由 apache Storm 官方专门提供的 jar 包，并修改在项目中 maven 文件的配置，添加信息如下所示：

```
<dependency>
    <groupId>org.apache.Storm</groupId>
    <artifactId>Storm-Kafka</artifactId>
    <version>0.11.0-SNAPSHOT</version>
</dependency>
```

然后在 Topology 定义代码中设置 Kafka spout，其中 zks 指定了缓存代理的地址，topics 指明了该 consumer 需要从 Kafka 处理的消息源的哪个分类中获取缓存数据：

```
// for Storm-Kafka test
String zks = "10.134.143.51:2181,10.134.143.55:2181,10.134.142.111:2181";
String zkRoot = "/" + topic; // default zookeeper root configuration for Storm
String id = UUID.randomUUID().toString();

BrokerHosts brokerHosts = new ZkHosts(zks);
SpoutConfig spoutConf = new SpoutConfig(brokerHosts, topic, zkRoot, id);
spoutConf.scheme = new SchemeAsMultiScheme(new MessageScheme());
spoutConf.forceFromStart = false;
spoutConf.zkServers = Arrays.asList(new String[] {"10.134.143.51","10.134.143.55","10.134.142.111"});
spoutConf.zkPort = 2181;
spoutConf.zkRoot = "/" + topic;
spoutConf.bufferSizeBytes = 104857600;
spoutConf.stateUpdateIntervalMs = 100;
spoutConf.useStartOffsetTimeIfOffsetOutOfRange = true;

//create the topology
TopologyBuilder builder = new TopologyBuilder();
String preOperation = "spout";
//KafkaSpout KafkaSpout = new KafkaSpout(spoutConf);

// create Kafka-spout
builder.setSpout(preOperation, new KafkaSpout(spoutConf), n1);
```

由于在实现的过程中，本文从 Kafka 中获取的消息类型不是默认的 String 类型，而是自定义的图像类型，因此需要实现 spoutConfig.scheme. spoutConfig.scheme 接口用于实现将 Kafka-spout 中接收到的 Kafka message 转换为 Storm 的 tuple 类型。实现的代码如下所示。

```
public class MessageScheme implements Scheme {
private static final long serialVersionUID = -67564994110664053531;
private static Logger logger = LoggerFactory.getLogger(MessageScheme.class);
FrameSerializer serializer = new FrameSerializer();
public List<Object> deserialize(byte[] bytes) {
    Frame frame = serializer.fromBytes(bytes);
    //check
    if (frame != null) {
        //logger.info("spout receive a frame with streamId " + frame.getStreamId() + " seq " + frame.getSequenceNr());
    } else {
        logger.error("spout can not get a not null frame!");
    }
    try {
        if (frame.getImage() == null) {
            logger.error("spout: image is null!");
        }
    } catch (IOException e1) {
        e1.printStackTrace();
    }

    try {
        return serializer.toTuple(frame);
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}
public Fields getOutputFields() {
    return serializer.getFields();
}
}
```

5.3 GPU 计算模块

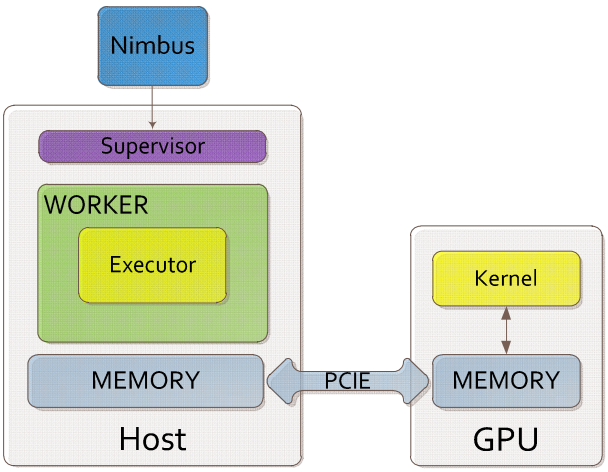


图5-5 Storm与GPU通信结构示意图

图 5-5 显示了 Storm 和 GPU 交互的整体设计。为了实现在 Storm 中运行用户提交的 CUDA 程序，首先需要用户在部署 topology 的同时，将每个 bolt 对应的 PTX 文件一同打包上传至每个含有 GPU 的工作节点，PTX 文件中包含了如何在 CPU 中处理每次接收到 tuple 的核函数。之后才能通过 executor 在 CPU 中所分配的线程来启动 GPU 中的核函数来处理收到的 tuple，每个 executor 为 bolt 的一个实例，一个 bolt 可以指定多个 executor 实例。同原生的 Storm 类似，nimbus 作为主控节点中的守护进程负责集群中代码的分发和 executor 线程在 worker 节点

间的分配。而 Supervisor 作为每个 worker 节点中的守护进程则负责接收 nimbus 分配的工作[31]。

经过本文改进后的编程模型与 Storm 的编程模型非常相似，都需要定义一个由 spout 和 bolt 组成的 topology 数据结构，并实现 spout 和 bolt 类中的 emit 和 process 方法。唯一的区别在于，本文使用了 JCUDA 作为 Storm 和 GPU 间通信的桥梁。JCUDA 作为一个访问 GPU 的库文件集合，拥有一套和 CUDA 对应的运行时 API 和驱动 API，为基于 java 的 Storm 程序提供的可靠简便的交互方式。本文通过将调用 GPU 的操作集成在 bolt 中来达到降低运行耗时的目的。下图显示了一个调用 GPU 的 executor 的函数调用过程。

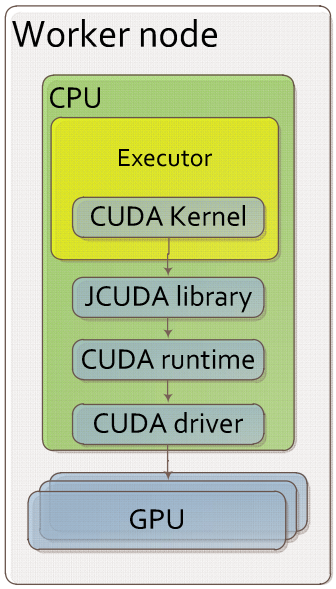


图5-6 CUDA调用过程

为了能够将调用GPU的过程完全集成到Storm中，本文首先针对每个图像处理算法实现了Stormcv中的ISingleInputOperation<Frame>接口。与Storm中BaseRichBolt的逻辑类似，该接口声明了两个抽象方法prepare()和execute()用于用户定制针对每帧图像的操作，只是传递的消息由tuple变为了图像。在定义了需要进行的图像处理操作后，为了能够在Storm的worker节点中调用jar包中封装的gpu代码，本文实现了GPUutils类来调用和java代码一起打包在jar中上传的cu文件，cu文件中包含了需要在gpu中执行的核函数，其中java函数检测和调用cu文件的代码如下所示。

1) 首先将包含处理图像核函数的cu文件名称发给GPUutils。

```
String ptxFileName;
ptxFileName = GPUUtils.preparePtxFile("compareAndFind.cu");
```

2) GPUUtils找到对应的cu文件，并去寻找对应的ptx文件，如果不存在，则

执行nvcc命令编译运行所需的ptx文件，getSelfJarDenpendencyFileTmp()方法会从一同上传的jar包中找到一起打包的cu文件。

```
public static String preparePtxFile(String cuFileName) throws IOException
{
    …
    String tmpCuFile = NativeUtils.getSelfJarDenpendencyFileTmp(cuFileName);
    File cuFile = new File(tmpCuFile);

    if (!cuFile.exists())
    {
        throw new IOException("Input file not found: "+cuFileName);
    }
    String modelString = "-m"+System.getProperty("sun.arch.data.model");
    String command =
            "nvcc " + modelString + " -ptx "+
                    cuFile.getPath()+" -o "+ptxFileName;

    System.out.println("Executing\n"+command);
    Process process = Runtime.getRuntime().exec(command);
    …
    System.out.println("Finished creating PTX file");
    return ptxFileName;
}
```

经过上述的操作，Storm的计算节点就可以轻易的调用JCUDA来实现与GPU之间的通信和计算。

5.4 视频图像修复应用

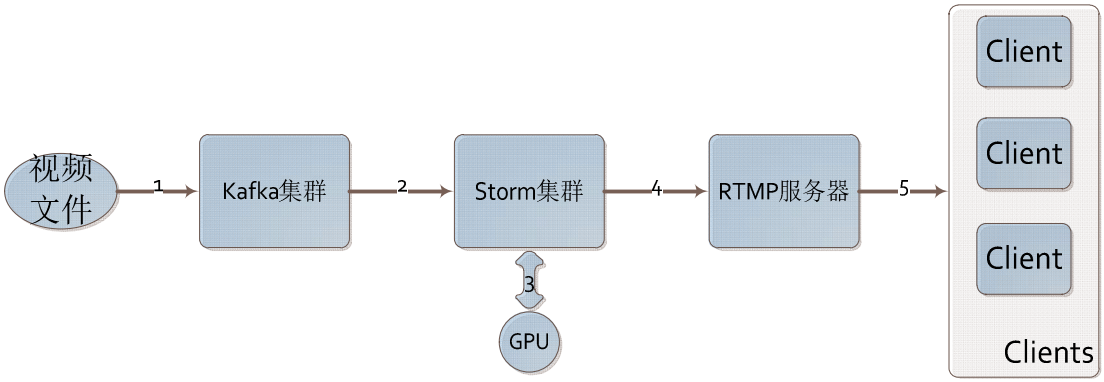


图 5-7 图像修复应用整体架构图

由于修复破损视频图像需要构建每帧图像对应的掩模图像并和原始图像进行合并，并且测试时耗时较长，所以测试时采用读取离线视频文件的方式来验证系统的功能。应用整体结构如图 5-7 所示，其中用于图像图像修复测试的 Topolgy 结构如图 5-8 所示，spout 为发送节点，会将读入的视频图像按帧数先后编号然后发送给处理节点 Bolt_P，处理节点 Bolt_P 在完成图像修复后将结果传递给

Bolt_E 节点缓存。最后 Bolt_E 节点将缓存的修复图像合并成 RTMP 视频格式，并发送到 RTMP 服务器用于播放。

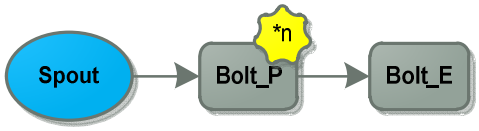


图 5-8 图像修复 topolopy 结构图

视频修复步骤如下所示：

1. 读取离线视频文件到 Kafka 缓存中，视频中的每帧图像都加入了对应的掩模作为该帧图像的破损区域；
2. Storm 集群启动负责图像修复的 topology，该 topology 中的 spout 节点通过消息订阅的方式从 Kafka 集群中的缓存中读出需要修复的图像，通过 Stormcv 对读出图像的帧数顺序进行编号然后传递给进行修复操作的 bolt 节点；
3. Bolt 节点在得到输入的图像后，将图像从内存拷贝到显存中，并调用 cu 文件中的核函数进行修复操作，在修复操作完成后，结果又由 GPU 拷贝回内存中用于下一次操作；
4. 在图像完成修复后，会被缓存在 topology 中的最后一个 bolt 节点中。该 bolt 节点会通过接收之前的处理节点修复的图像，并根据每张图像的编号来合成 RTMP 视频流发送至 RTMP 服务器；
5. 最后用户通过访问 RTMP 服务器的方式，可以从网页上观看到对应的处理结果；

5.5 本章小结

本章详细描述了基于云的分布式图像修复系统的设计与实现。首先阐述了该系统的整体框架，然后分别对图像采集和分发模块、gpu 计算模块和视频修复应用做了详细的阐述。通过 Kafka 分布式消息订阅系统，图像采集和分发模块解决了 Storm 动态获取数据的需求；GPU 计算模块通过为 Storm 加入 GPU 计算节点，提高了云计算平台并行处理图像数据的能力；最后本文利用上述已搭建的云计算平台部署了一套针对视频修复的应用，实现了通过云平台调用 Storm 的 bolt 节点中得 GPU 来对图像数据进行修复。

第六章 测试与分析

6.1 开发环境介绍

本文的测试均在三台服务器和两个 GPU 上进行，其中三台服务器分别配置成一个管理节点和两个运算节点，两个 GPU 则被分别安装在两个计算节点中，对应的硬件和软件配置如下所示：

6.1.1 硬件环境

表 6-1 服务器硬件配置

	dellr910c1	dellr920	dellt610
逻辑 CPU	64 Intel(R) Xeon(R) CPU E7- 4820 @ 2.00GHz	64 Intel(R) Xeon(R) CPU E7-4820 v2 @ 2.00GHz	16 Intel(R) Xeon(R) CPU E5520 @ 2.27GHz
物理 CPU	4x8=32	4x8=32	2x4=8
线程数	2	2	2
主存	128G	256G	4

表 6-2 GPU 配置

	GT730	GTX970
制作工艺	28 纳米	28 纳米
核心频率	1033MHz	1051/1127MHz
显存频率	5000MHz	7010MHz
显存类型	GDDR3	GDDR5
显存容量	1024MB	4096MB
显存位宽	64bit	256bit
接口类型	PCI Express 3.0 16X	PCI Express 3.0 16X
流处理单元	384 个	1664 个

6.1.2 软件环境

表 6-3 服务器软件配置

	dellr910c1	dellr920	dellt610
系统	redhat6.4	redhat6.4	redhat6.4
内核	2.6.32	2.6.32	2.6.32
ip	10.134.143.51	10.134.143.55	10.134.142.111
zookeeper	leader	follower	follower
Storm	Storm 控制节点	Storm 计算节点	Storm 计算节点

6.2 吞吐性能测试

为了验证 Kafka 在分发消息的过程中能够快速地对 broker 进行读写操作，保证服务器间的带宽能有效的被利用起来，本文对 Kafka 中 producer 的写操作和 consumer 的读操作进行了相应的测试：

6.2.1 Producer 吞吐率测试

（1）测试用例：单个 producer 写入数据

测试脚本：

```
$KAFKA_HOME/bin/Kafka-topics.sh --zookeeper 10.134.143.51:2181, 10.134.143.55:2181, 10.134.142.111:2181
--create --topic test --partitions 3 --replication-factor 1

$KAFKA_HOME/bin/Kafka-run-class.sh org.apache.Kafka.clients.tools.ProducerPerformance test 50000000 100
-1 acks=1 bootstrap.servers=lab36951:9092 buffer.memory=67108864 batch.size=8196
```

Kafka-topics.sh 脚本创建了一个名为 test 的 topic，这个 topic 包含了 3 个 partition。然后 Kafka-run-class.sh 脚本创建了一个线程，该线程会不停地向 Kafka 发送 payload 为 100 字节的消息，累计到 5000 万条为止。

（2）测试用例：三个 producer 同时写入数据

测试脚本：

```
$KAFKA_HOME/bin/Kafka-topics.sh --zookeeper 10.134.143.51:2181, 10.134.143.55:2181, 10.134.142.111:2181
--create --topic test --partitions 3 --replication-factor 1

$KAFKA_HOME/bin/Kafka-run-class.sh org.apache.Kafka.clients.tools.ProducerPerformance test 50000000 100
-1 acks=1 bootstrap.servers=lab36951:9092 buffer.memory=67108864 batch.size=8196

$KAFKA_HOME/bin/Kafka-run-class.sh org.apache.Kafka.clients.tools.ProducerPerformance test 50000000 100
-1 acks=1 bootstrap.servers=lab36955:9092 buffer.memory=67108864 batch.size=8196

$KAFKA_HOME/bin/Kafka-run-class.sh org.apache.Kafka.clients.tools.ProducerPerformance test 50000000 100
-1 acks=1 bootstrap.servers=lab369111:9092 buffer.memory=67108864 batch.size=8196
```

Kafka-topics.sh 脚本创建了一个名为 test 的 topic，这个 topic 包含了 3 个 partition。然后在 3 台服务器上同时运行 Kafka-run-class.sh 脚本创建三个线程，

每个线程都会不停地向 Kafka 发送 payload 为 100 字节的消息，分别累计到 5000 万条为止。

测试结果如下所示：

表 6-4 Producer 吞吐率测试

	单个 producer	三个 producer
Producer 吞吐率	9 *MB/second*	31 *MB/second*

6.2.2 Consumer 吞吐率测试

（1）测试用例：单个 Consumer 读取数据

测试脚本：

```
$KAFKA_HOME/bin/Kafka-topics.sh --zookeeper 10.134.143.51:2181, 10.134.143.55:2181, 10.134.142.111:2181
--create --topic test --partitions 3 --replication-factor 1

$KAFKA_HOMEbin/Kafka-consumer-perf-test.sh --zookeeper 10.134.143.51:2181 --messages 50000000 --topic test --threads 1
```

Kafka-topics.sh 脚本创建了一个名为 test 的 topic，这个 topic 包含了 3 个 partition。然后 Kafka-consumer-perf-test 脚本创建了一个线程，该线程会不停地从 Kafka 中消费 Broker 中得消息，累计到 5000 万条为止。

（2）测试用例：三个 Consumer 同时读取数据

测试脚本：

```
$KAFKA_HOME/bin/Kafka-topics.sh --zookeeper 10.134.143.51:2181, 10.134.143.55:2181, 10.134.142.111:2181
--create --topic test --partitions 3 --replication-factor 1

$KAFKA_HOMEbin/Kafka-consumer-perf-test.sh --zookeeper 10.134.143.51:2181 --messages 50000000
--topic test --threads 1

$KAFKA_HOMEbin/Kafka-consumer-perf-test.sh --zookeeper 10.134.143.55:2181 --messages 50000000
--topic test --threads 1

$KAFKA_HOMEbin/Kafka-consumer-perf-test.sh --zookeeper 10.134.142.111:2181 --messages 50000000
--topic test --threads 1
```

Kafka-topics.sh 脚本创建了一个名为 test 的 topic，这个 topic 包含了 3 个 partition。然后在 3 台服务器上同时运行 Kafka-consumer-perf-test.sh 脚本创建三个线程，该线程会不停地从 Kafka 中消费 Broker 中的消息，分别累计到 5000 万条为止。

测试结果如下所示：

表 6-5 consumer 吞吐率测试

	单个 consumer	三个 consumer
consumer 吞吐率	9 *MB/second*	28 *MB/second*

最后为了验证 Kafka 在写入和读出的过程中带宽的占用情况，本文通过 scp 命令拷贝文件对不同机器间的网络性能测试进行了测试，测试过程如下：

```
Clwang@node1:~$ scp test xtu@10.134.143.55:/home/xtu
test                          100%  927MB  10.3MB/s   01:30
```

测试时通过将一个 900M 左右的测试文件从一个服务器拷贝到另一个服务器，结果显示耗时 1 分 30 秒，平均传输速率在 10m/s 左右，与 Kafka 间单个节点接收和发送数据的速率相当，而由三台机器组成的 Kafka 集群其吞吐已经能够超过单个网卡的带宽限制，说明该 Kafka 集群已经充分利用了网络的带宽。

6.3 传输延迟测试

为了测试 Storm 内 component 间消息传递的延迟对最终的处理结果造成的影响，本文实现了一个用于测试的 Reference topology，其拓扑结构如下所示

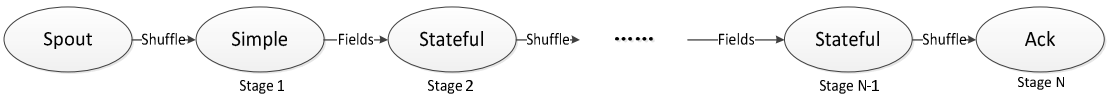


图 6-1 Reference topology 结构图

该 topology 中包含两种处理节点——simple 节点和 Stateful 节点，两种节点均不对传递的数据进行操作，并且传递的消息只包含发送消息的 spout 节点编号和消息发送时间的数据。其中 simple 节点对应分布式计算中的 Map 阶段，该类节点接收随机发送来的数据，并按照消息中 spout 的编号对信息进行分类然后发送给对应的 Stateful 节点；而 Stateful 节点对应分布式计算中的 Reduce 阶段，该类节点按照消息中 spout 的编号接收 simple 节点发来的数据，然后将消息随机的发送到下一级的 simple 节点中。这样通过交替设置多个 simple 节点和 Stateful 节点，可以有效模拟通用的流式 Map/Reduce 计算模型。在测试中，本文分别加入了多组 simple 节点和 Stateful 节点进行测试，每个节点分别实验了 2 个实例和 4 个实例对传输延迟的影响，除了 Spout 和 Ack 节点被分配在同一台机上用于测试发送到接收的延迟外，其余节点均随机分配，测试结果如下表所示：

表 6-6 处理延时测试

	Stage6/ ms	Stage8/ ms	Stage10/ ms	Stage12/ ms
每个节点 2 个实例	2.02	2.38	2.97	3.75
每个节点 4 个实例	1.71	2.23	2.84	3.28

由测试结果可以看出，经过该系统传输的数据，即使在经过了 12 个节点的传输之后，延迟依旧能够维持在毫秒级，并不会给数据的处理带来过多额外的开销，满足系统中对于实时视频处理部分的延迟需求。

6.4 图像修复应用测试

(a)海浪

(b)海滩行走

图 6-2 修复结果对比，左边为破损图，中间为 CPU 测试结果，右边为 GPU 测试结果

图 6-2 显示了使用该系统对分别对 352*288 和 720*1080 分辨率的视频进行修复的修复结果。经过 MSE 的对比结果显示，两者的修复结果完全一致。从结果中可以看出经过分布式系统和 GPU 处理过的图像和使用 CPU 进行修复后的图像拥有的修复结果。表 6-7 显示了分别使用单个 CPU、Storm 集群（CPU 作为计算节点）和 Storm 集群（GPU 作为计算节点）的修复耗时结果对比。通过对比结果可以看出，相较于使用单个 CPU 对视频进行修复所用耗时，使用 Storm 集群处理时修复所用耗时随着处理线程数的增加而线性的减少。而使用 GPU 作为计算节点后，修复耗时进一步降低。从结果中可以看出，同时使用两个 GPU 进行计算时，其耗时已经超越 Storm 通过 2 台机器共 16 个线程进行处理的速度，使得在修复分辨率较低的图像时平均每帧的处理速度达到秒级，而在处理分辨率较高的图像时也取得了不错的加速比，从而证明了该系统用于处理视频数据的可行性和在处理耗时方面的优越性。

表 6-7 处理耗时测试

	海浪（352*288*99）帧/s	海滩行走（720*1080*104帧）/s
CPU 修复耗时	977	44720
Storm 修复耗时（1 个 worker 共 8 线程）	125	5760
Storm 修复耗时（2 个 worker 共 16 线程）	69	3014

Storm 加入 GPU 计算节点修复耗时（服务器 113 中 1 个 GTX970 节点）	54	2600
Storm 加入 GPU 计算节点修复耗时（服务器 112 中 1 个 GT730 节点）	159	9256
Storm 加入 GPU 计算节点修复耗时（服务器 113 中 1 个 GTX970 节点+服务器 112 中 1 个 GT730 节点）	44	2045

6.5 本章小结

本章完成了对上一章实现的基于云的分布式图像修复系统的测试工作。首先对部署云计算系统使用的硬件和软件环境进行了介绍。然后对分别对系统的吞吐性能、传输延迟、修复速度和效果进行了测试。测试结果表明：

（1）本文用做云平台输入的 Kafka 分布式发布订阅消息系统能够有效的利用带宽资源来发送数据；

（2）本文搭建的 Storm 分布式图像处理环境在传输数据的过程中耗时较低，不会给图像处理应用造成过多额外的延迟

（3）通过云平台中 Storm 和 GPU 处理过的图像修复结果与用 CPU 处理的结果一致，并且由于采用了分布式处理和 GPU 加速，极大缩短了修复耗时，在处理分辨率较低的视频时可以达到秒级延迟。

最终的实验结果证明了该云计算系统能够通过分布式的方式处理基于视频图像的数据，并且能够充分利用 GPU 的并行化方式来提高图像处理的速度。

第七章 总结和展望

7.1 本文研究总结

本文首先针对云计算平台进行了研究，并详细介绍了 Hadoop 和 Storm 两种云计算处理框架和一些用于搭建云计算平台需要用到的组件。之后，本文提出了一种改进的 Criminisi 算法来提高图像修复的质量和速度，并对其中的样本块匹配部分进行了 GPU 并行化改进。最后本文针对图像修复算法对计算能力的需求和海量视频处理并行化的趋势，将 GPU 作为计算节点集成到 Storm 内部，使其能够运行在基于 Storm 搭建的并行化集群中，从而组成了一套基于 Storm 和 CUDA 的分布式计算系统，并将并行化后的图像修复算法作为应用部署在其中，最终达到使用云计算系统来对视频图像修复算法进行加速的目的。

7.2 未来工作展望

在后续的工作中，本人将对该系统做进一步的改进。首先是修改 Storm 的资源调度器，由于 GPU 的加入使得之前用于调度获取 cpu 资源并进行任务分配的 adaptive online 调度器也随即失效，而且目前没有很好的方法对 GPU 的运行情况进行实时监控，所以需要进一步的修改调度器使得任务的分配能够更加合理，而不是目前采用的随机分配策略。

其次在图像修复算法方面，本文的算法在修复高分辨率图像的时候修复速度依旧不理想，而且当图像结构比较复杂时，仅依靠待修复像素点上的等照度线方向作为结构区域的修复方向会造成结构的错误延伸。另外本文在进行分水岭分割的过程中发现分割的阈值越小，匹配越准确，但阈值过小后会出现一些孤立的区域，它们与待修复区域相交但无法在图像的其他区域中找到接近的匹配块。所以还需要进一步改进图像修复算法。

除此之外，目前本文设计的分布式图像处理系统在部署图像处理应用时，为了保证处理视频时实现不同帧图像的并行化，所以只能处理不依赖于视频帧间数据的算法。对于那些依赖前一帧处理结果的算法，本系统目前只能用一个节点进行顺序操作的方式进行处理。这样对整个系统的资源都是一种很大的浪费，需要日后更进一步的改进和研究来提高该系统处理上述图像算法时的效率。

综上所述,本文设计的基于 Storm 的分布式图像修复系统系统在目前阶段还有许多不足之处，并且需要进一步的优化来对这些不足之处进行改进。所以在日后的学习和工作中会有针对性的对每个不足之处一一进行研究与改进。

参考文献

[1] P. Chandarana,V.E.S.I.T., Mumbai, India,M. Vijayalakshmi. Big Data analytics frameworks[C]// Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference. Melbourne, VIC:IEEE,2013:1784 - 1787

[2] Paul Zikopoulos,Chris Eaton. Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data1st .McGraw-Hill Osborne Media ©2011 .ISBN:0071790535 9780071790536.

[3] W. Yang, Chengdu, China, X. Liu, L. Zhang , L. T. Yang. Big Data Real-Time Processing Based on Storm[C]//Proceedings of the 27th annual conference on Computer graphics and interactive techniques. New York, USA: ACM Press/Addison-Wesley Publishing Co, 2000: 417-424.

[4] 钱悦.图形处理器 CUDA 编程模型的应用研究[J]. 计算机与数字工程.2008,12:177-180.

[5] 方宝龙.基于纹理合成的图像修复算法研究[D].济南,山东大学,2013.

[6] Bertalmio M, Guillermo S, Vincent C, et al. Image Inpainting [C]//Proceedings of the 27th annual conference on Computer graphics and interactive techniques. New York, USA: ACM Press/Addison-Wesley Publishing Co, 2000: 417-424.

[7] Chan T, Shen F.Mathematical models for local non-texture inpainting[J]. SIAM Journal on Applied Mathematics, SIAM, 2002, 62(3): 1019-1043.

[8] Ef ros A , Leung T.Texture synthesis by non-paramet ric sampling[C]//Proceedings of IEEE Int ernational Conference on Comput er Vision. Kerkyra, Greece: [s.n.],1999 :1033-1038.

[9] Criminisi, Perez P, Toyama K. Region filling and object removal by exemplar-based image inpainting[J]. Image Processing , IEEE Transactions, 2004, 13(9): 1200 – 1212.

[10] Wen Huangcheng, Chun Weihsieh, Sheng Kailin, et al. Robust Algorithm for Exemplar-based Image Inpainting[C] // Proceedings of the International Conference on Computer Graphics, Imaging and Vision (CGIV 2005). Beijing, China: [s.n.], 2005:64-69.

[11] 刘 奎, 苏本跃, 王一宾. 基于样例的图像修复改进算法[J]. 计算机工程, 2012, 38(7): 193-194

[12] Masnou S. Disocclusion: A Variational Approach Using Level Lines [J]. Image Processing, IEEE Transactions, 2002, 11(2): 68 - 76.

[13] Fr´ed´eric Cao, Yann Gousseau, Simon Masnou, et al. Geometrically guided exemplar-based inpainting[J]. SIAM J. IMAGING SCIENCES, 2009, 4(4):1143–

1179.

[14] Hui Qinwang, Qing Chen, Cheng Hsiung Hsieh, et al. FAST EXEMPLAR-BASED IMAGE INPAINTING APPROACH[C]// Proceedings of International Conference on Machine Learning and Cybernetics, ICMLC 2012, International Conference. Xian, China: IEEE Computer Society Press, 2012: 1743 – 1747.

[15] 王昊京, 王建立, 王鸣浩，等. 采用双线性插值收缩的图像修复方法[J]. 光学精密工程, 2010, 8(5): 1234-1241.

[16] Anupam, Pulkit Goyal, Sapan Diwakar, "Fast and Enhanced Algorithm for Exemplar Based Image Inpainting",2010 Fourth Pacific-Rim Symposium on Image and Video Technology,2010 , 61,325- 330.

[17] Vincent Cheung , Brendan J. Frey, Nebojsa Jojic. Video Epitomes[J]. International Journal of Computer Vision, 2008, 76 (2): 141-152.

[18] K. A. Patwardhan ; Minnesota Univ., Minneapolis, MN, USA ; G. Sapiro ; M. Bertalmio. Video inpainting of occluding and occluded objects[C]// Image Processing, 2005. ICIP 2005. IEEE International Conference. IEEE, 2005: 69-72.

[19] K. A. Patwardhan ; Electr. & Comput. Eng., Minnesota Univ., Minneapolis, MN ; G. Sapiro ; M. Bertalmio. Video Inpainting Under Constrained Camera Motion[J]. IEEE Transactions on Image Processing, 2007, 16(2): 545 - 553.

[20] Q. Ning ; Dept. of Comput. Sci. & Eng., Texas A&M Univ., College Station, TX, USA ; C. A. Chen ; R. Stoleru ; C. Chen. Mobile storm: Distributed real-time stream processing for mobile clouds[C]// Cloud Networking (CloudNet), 2015 IEEE 4th International Conference. Niagara Falls, ON: IEEE, 2015: 139 - 145.

[21] 任凯,戴瑾.云计算概念、技术与开源项目[J] . 萍乡高等专科学校学报.2014,6:51-56.

[22] K. Danniswara ; KTH R. Inst. of Technokogy, Stockholm, Sweden ; H. P. Sajjad ; A. Al-Shishtawy ; V. Vlassov. Stream Processing in Community Network Clouds[C]// Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference. Rome: IEEE, 2015: 800 - 805.

[23] Leonardo Aniello, Roberto Baldoni, Leonardo Querzoni. Adaptive Online Scheduling in Storm [C]

[23] Leonardo Aniello, Roberto Baldoni, Leonardo Querzoni. Adaptive Online Scheduling in Storm [C]

[24] S. Zhao ; Comput. Sci. & Electr. Eng. Dept., Univ. of Missouri-Kansas City, Kansas City, MO, USA ; M. Chandrashekar ; Y. Lee ; D. Medhi. Real-time network anomaly detection system using machine learning[C] // Design of Reliable Communication Networks (DRCN), 2015 11th International Conference. Kansas City, MO: IEEE, 2015: 267 – 270.

[6] Bertalmio M, Guillermo S, Vincent C, et al. Image Inpainting [C]//Proceedings of the 27th annual conference on Computer graphics and interactive techniques. New

York, USA: ACM Press/Addison-Wesley Publishing Co, 2000: 417-424.

[8] Ef ros A , Leung T.Texture synthesis by non-paramet ric sampling[C]//Proceedings of IEEE Int ernational Conference on Comput er Vision. Kerkyra, Greece: [s.n.],1999 :1033-1038.

[9] Criminisi, Perez P, Toyama K. Region filling and object removal by exemplar-based image inpainting[J]. Image Processing , IEEE Transactions, 2004, 13(9): 1200 – 1212.

[25] Chunduck Park, BaekSop Kim. Distance weighted Bounding for Fast Exemplar-based Inpainting[J]. International Journal of Software Engineering and Its Applications,2015,9(2): 143-150

[26] Dr. Neal Krawetz. Looks Like It. [EB/OL]. http://www.hackerfactor.com/blog/index.php?/archives/432-Looks-Like-It.html.

[27] 王宇新, 贾 棋, 刘天阳等. 遮挡物体移除与图像纹理修补方法[J]. 计算机辅助设计与图形学学报, 2008, 20 (1): 43-49.

[28] 朱 霞, 李 宏, 张 卫. 一种基于颜色区域分割的图像修复算法[J]. 计算机工程, 2008, 34 (14): 191-193.

[29] The Watershed Transform: Definitions, Algorithms and Parallelization Strategies[j]. Fundamenta Informaticae, 2000, 41: 187-228.

[30] Yonghong Yan, Max Grossman, Vivek Sarkar. JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA[C]// Euro-Par 2009 Parallel Processing, Springer Berlin Heidelberg,2009,5704: 887-899.

[31] Z. Chen ; Dept. of Electr. Eng. & Comput. Sci., Syracuse Univ., Syracuse, NY, USA ; J. Xu ; J. Tang ; K. Kwiat more authors.G-Storm: GPU-enabled high-throughput online data processing in Storm[C]// Big Data (Big Data), 2015 IEEE International Conference. Santa Clara, CA: [s.n.], 2015: 307 - 312.

致谢

时光荏苒，三年研究生的时光就这样匆匆而逝，加上本科 4 年的时间，不知不觉已经在复旦生活了七个年头。我很感激在复旦的这七年的时间，因为在这七年里我体会了人生的苦辣酸甜，让我由当初懵懂的少年变成了如今能够独担一面的男子汉。回首过去三年的研究生生活和七年的复旦生活，我的内心感慨颇多，从刚踏进校园到进入实验室，从本科毕业到成为一名复旦的研究生，往事历历在目，心中感慨万千。

在临毕业之际，作为一名复旦学生，我第一个要感谢的人是我们敬爱的导师——陈更生老师。陈老师在平时除了具备严谨治学，教育学生严肃认真等复旦老师普遍的优点之外，为人更是随和热情，在专业知识上不可谓不博学多闻。在科研过程，当我每每遇到问题时，陈老师总是能够给学生指点迷津；在生活中，陈老师更是给了我无微不至的关怀，当我家里发生变故时，也是陈老师给予了我最大的帮助。所以借此机会我要对陈老师的精心指导和无私关怀说句衷心的感谢。

同时，我要感谢钮圣虓师兄。从我本科进入实验室起，钮圣虓师兄就在许多项目中给予了我非常多的帮助，尤其在小论文实现过程中每当遇到困难时，他都耐心的给予我指导，使得我的小论文工作能够顺利的进行下去。其次要感谢实验室已经毕业的王盛师兄和伍赛师兄，我还记得我刚到 308 实验室的时候，什么都不会，什么都不懂，每次遇到问题时总要麻烦他们，而他们也总是不厌其烦的给我提供热心的帮忙，让我感受到了这个实验室的人情味。最后感谢严健康、王磊、叶旻渊和郑杰等多位同学在我完成大论文时给予的帮助，是你们的辛勤劳动才能使得我能够完成这篇论文的测试工作！

最后，我要感谢我的母亲和父亲还有默默在背后支持我的家人，感谢你们在这 25 年里对我的养育之恩，感谢你们在我背后默默付出和承受的一切，将来待我成才之日定会以涌泉之恩相报。

论文独创性声明

本论文是我个人在导师指导下进行的研究工作及取得的研究成果。论文中除了特别加以标注和致谢的地方外，不包含其他人或其它机构已经发表或撰写过的研究成果。其他同志对本研究的启发和所做的贡献均已在论文中作了明确的声明并表示了谢意。

作者签名：_____ 日期：_____

论文使用授权声明

本人完全了解复旦大学有关保留、使用学位论文的规定，即：学校有权保留送交论文的复印件，允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其它复制手段保存论文。保密的论文在解密后遵守此规定。

作者签名：_____ 导师签名：_____ 日期：_____