**5.1 封面**

采用研究生院指定的统一封面。

**2中文目录**

目录由序号、标题名称和页码组成。包括：正文（含引言、结论）的一级、二级和三级序号和标题、参考文献等内容。

**3中文摘要 （300字左右）**

论文摘要由题目、摘要正文、关键词（3-5个）三部分组成。

**4 英文摘要**

内容同3。

1. **正文 (不得少于7,000字)**
   1. 课题背景

课题来源及背景、研究目的、理论意义和实际应用价值

5.2文献综述，

国内外在该研究方向研究现状及发展动态，引文应标注出自的参考文献序号。

5.3研究目标

5.4研究内容

5.5主要研究方案、技术路线与可行性分析

5.6预期研究成果和（或）创新点

至少列出与前人不同的成果和（或）创新点

5.7工作基础和条件

包括试验研究条件、已经开展的相关工作、存在的不足与问题等。

5.8研究进度安排

包括文献调研，工程设计，新工艺、新材料、新设备、新产品的研制和调试，实验操作，实验数据的分析处理，撰写论文等。以季度为单位列出。

**6 参考文献**

参考文献不少于30篇，其中外文文献不少于10篇，格式严格按照《华东理工大学学报自然科学版》格式要求著录。

**3中文摘要**

在高性能计算系统中构建和部署基于图形处理加速器的软件环境是一项极具挑战性的任务。高性能应用程序必须可靠地运行在多个平台和环境中，并在解决复杂的软件堆栈依赖关系的同时充分利用图形处理加速器的资源。容器通常用于在多台机器上无缝部署基于CPU的应用程序。有了这个用例，容器就是硬件无关的和平台无关的。使用英伟达图形处理加速器显然不是这样，因为它使用专用的硬件，并且需要安装英伟达驱动程序。因此，Docker Engine本身并不支持将英伟达图形加速器封装在容器内部。

容器是一种轻量级的虚拟化技术，具备便携式，易于构建和部署，占用空间小，运行时间少等优势。通过将应用程序及其环境封装到标准的软件单元中来解决部署复杂环境的问题。为了使Docker 容器在使用英伟达图形加速器的同时具备便携式和跨平台的特点，我们使用的解决方案是分离图像与英伟达驱动程序之间的联系。启动目标机器上的容器时，将安装所需的字符设备和驱动程序文件。

本文研究分析了基于Volta架构的图形处理器和统一计算设备架构，如合作组织，动态平衡度等编程模型。笔者通过在实际工作中遇到的问题和需求为Docker提供了运行时的扩展应用，它无缝隙的支持基于Volta架构的图形处理器，并在容器内能实现所有最新CUDA提供的技术方案，它为容器化应用程序提供了访问图形加速器和主机系统的专用机制，从而有效地解决了高性能计算资源在容器内的移植性问题。

在实际工作中，优化后的容器将应用程序包装到隔离的虚拟环境中，通过此项技术，大大简化了CUDA和操作系统在数据中心环境部署工作，运维和测试人员可以轻松地集成和隔离加速应用程序，无需任何修改，并将其部署在任何支持GPU的基础设施上,极大地节省了资源。

**关键字**：GPU，Docker，高性能计算，容器，虚拟化，异构系统，CUDA

**4 英文摘要**

Building and deploying a software environment based on a graphics processing accelerator in a high-performance computing system is a challenging task. High-performance applications must reliably run on multiple platforms and environments and take full advantage of the resources of the graphics processing accelerator while resolving complex software stack dependencies. Containers are typically used to seamlessly deploy CPU-based applications on multiple machines. With this use case, the container is hardware-independent and platform-independent. The use of NVIDIA graphics processing accelerator is clearly not the case because it uses dedicated hardware and requires the installation of NVIDIA drivers. As a result, the Docker Engine itself does not support encapsulating the NVIDIA graphics accelerator inside the container.

A container is a lightweight virtualization technology that is portable, easy to build and deploy, takes up less space and runs less time. Solve the problem of deploying complex environments by encapsulating the application and its environment into standard software units. In order to make the Docker image with portable and cross-platform features while using the NVIDIA graphics accelerator, the solution used by the container is to make the image irrelevant to the NVIDIA driver. When you start the container on the target machine, the required character device and driver files are installed.

This paper analyzes and analyzes the graphics processor based on Volta architecture and the unified computing device architecture, such as cooperative organization, dynamic balance and other programming model. The author provides Docker with run-time extended applications through the problems and requirements encountered in real work. It seamlessly supports the graphics processor based on the Volta architecture and enables the implementation of all the latest CUDA technology solutions in the container. Provides a specialized mechanism for accessing graphical accelerators and host systems for containerized applications, effectively addressing the portability of high-performance computing resources in containers.

In actual work, the optimized container wraps the application into an isolated virtual environment, greatly simplifying the deployment of CUDA and the operating system in the data center environment, and the ease of integration and isolation Accelerate applications, without any modification, and deploy them on any GPU-enabled infrastructure, dramatically saving resources.

**Keywords**: GPU, Docker, High Performance Computing, Container, Virtualization, Heterogeneous System, CUDA

**5 正文 (不得少于7,000字)**

5.1课题背景

长期以来计算机科研人员最为担忧的是软件的可移植性问题。起初在20世纪50年代后期，人们创造出COBOL编程语言，这主要用来降低软件移植到新平台上去的复杂性和成本[1]。自那时起已经有60多年了，目前人们仍然在努力应对这些问题的根本性挑战。在高性能计算（HPC）领域，人们越来越多的把注意力聚焦在便携性上，这是因为在不损失高水平计算性能的前提下，高性能应用程序通常需要被运行在各种各样的平台和环境中。

从最初人们在COBOL中的工作，许多创新使得我们更加关注多年来软件的可移植性。让我们考虑下面的几个方面：编程语言（例如COBOL，Fortran，C，C++，Python），便携式库（例如Boost C++，PETSc），通用操作系统（例如嵌入式系统到超级计算机的Linux），计算机平台标准（例如IBM-PC,基于x86的架构），软件模式（例如通过可重用的软件组件的可移植性）。尽管这些努力已经成为人们处理软件的重要工具，但是由于对于如：性能，资源限制，依赖机器的功能，与操作系统有关的要求，软件库和各种工具的可用性日益增长的需求使得软件可移植性成为一个极为复杂的问题。因此，开发人员通常会将可移植性作为与其他要求的权衡。这里我们考虑使用Hadoop[2]的方法，Hadoop是一个开源框架用作大数据集的分布式处理和存储，通过抽象出平台软件和硬件的细节来实现可移植性[3]。如前所讨论的，Hadoop中对软件层的抽象就是一个十分有针对性的例子，它在性能和可移植性之间做了权衡[4][5]。

在高性能计算需求的背景下，人们要求程序同时能兼顾高性能和可移植性[6]。特别是在超级计算机架构中这项要求特别具有挑战，以为配置和软件环境在系统和供应商之间差异很大。我们要求高性能软件能适应各种不同的系统环境。工程师们最需要考虑的是在跨超级计算机站点移植代码任务时候产生的时间和金钱成本。然而，大多数情况下，用户和开发人员都希望将其时间花在新的科学研究和软件开发商，而不是在平台和环境之间移植应用程序。因此，简化和加速应用程序的移植的工作流程可以大大提高用户和开发人员的生产力。

虚拟化技术在过去十多年中呈现快速增长的趋势，它展现出可以胜任轻松移植和部署应用程序，尤其在云环境中被使用的最为广泛。

5.2文献综述

上世纪60年代开始，美国的计算机学术界就有了虚拟技术思想的萌芽。1959年克里斯托弗（Christopher Strachey）发表了一篇学术报告，名为《大型高速计算机中的时间共享》（Time Sharing in Large Fast Computers），他在文中提出了虚拟化的基本概念，这篇文章也被认为是虚拟化技术的最早论述。L.W.Comeau和R.J.Creasy创造性地设计了一种名为CP-40的新型操作系统，该操作系统实现了虚拟内存和虚拟机[7]。虚拟化技术在 20 世纪 60 年代首次出现，由IBM 率先实施：对大型机进行逻辑分区以形成若干独立虚拟机的一种方式。这些分区允许大型机进行“多任务处理”：同时运行多个应用程序和进程。原因是当时大型机是十分昂贵的资源，因此设计虚拟化技术来进行分区，作为一种充分利用投资的方式，解决了大型机的僵化和使用率不足的情况。

在 20 世纪 80 年代和 90 年代，由于客户端-服务器应用程序以及价格低廉的x86服务器和台式机组成了分散的计算机架构，大型机上的虚拟化技术处于停滞不前的状态。

在20世纪，虚拟化技术基本上都是服务器虚拟化，进入了21世纪，随着IT的发展，虚拟化的思路被借用到服务器以外的领域(包括存储，网络，桌面应用等)，形成了各种各样的虚拟化技术。本文重点从桌面应用来看，开始出现了应用虚拟化（也称桌面虚拟化）的技术，该技术把应用程序的人机交互逻辑（应用程序界面、键盘及鼠标的操作、音频输入输出、读卡器、打印输出等）与计算逻辑隔离开来，客户端无需安装软件，通过网络连接到应用服务器上，计算逻辑从本地迁移到后台的服务器完成，实现应用的快速交付和统一管理。

高性能计算泛指快速、量大和性能高的一类计算，诸如向量计算、并行与分布式计算、网格计算等[8][9]。它需要大量计算能力(power) 和强大的计算设施在很短的时间周期内完成给定的计算任务。所以在并行系统中，常通过增加处理器数目来提升计算能力(capability), 即计算速度。

从历届Top500排名可见，位于前列的机器中不少采用了GPU异构[10]加速体系架构，这说明采用CPU+GPU[10]混合异构架构对整机计算能力的贡献很大，从而使其逐步进入了主流。

容器技术需要解决最为核心的问题是针对软件的创建、发布和运行。它通过将运行环境和应用程序打包到一起，来解决部署的环境依赖问题，整整做到跨平台的发布和使用。与虚拟化相比，这样既不需要指令级模拟，也不需要即时编译。容器可以在核心CPU本地运行指令，而不需要任何专门的解释机制。容器[11]会比虚拟机更高效，因为它们能够分享一个内核和分享应用程序库。

容器是在OS级别进行的一种虚拟化[12]。 容器通过对内核的系统调用与主机操作系统进行通信。 因此，OS内核是用户空间容器化应用程序与应用程序在部署期间访问的主机系统的硬件资源之间的接口层。容器有效地将单个操作系统管理的资源划分到孤立的组中，以便更好的在孤立的组之间平衡有冲突的资源使用需求。

最初有一家叫DotCloud的法国公司提供PaaS[13]服务，他能对支持多种语言的运行环境，如Java、Python、Ruby等。可是在PaaS领域有太多巨头已经布局，DotCloud考虑如果不开源，很难与巨头竞争，所有就干脆将Docker[14]项目开源，至少能在开源社区得到个好名声。2013年3月，Docker正式以开源形式发布，此举让容器领域有了新的春天，截止2015年11月，Docker在Github上收到25600个赞，超过6800次克隆，以及超过1100名代码贡献者，成为20个最具影响力的Github开源项目。目前，世界上几乎所有的科技公司都在拥抱以Docker为代表的虚拟化生态圈。

Docker利用Linux的一些内核机制例如[cGroups](https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt" \t "_blank)、命名空间和[SElinux](http://selinuxproject.org/page/Main_Page)来实现容器之间的隔离[15]。起初Docker只是 [LXC](https://linuxcontainers.org/) [16]容器管理器子系统的前端，但是在0.9版本中引入了[libcontainer](http://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/" \t "_blank)，这是一个原生的go语言库，提供了用户空间和内核之间的接口[17]。

当前图形设备处理器易于获得并且价格可以被广泛接受，被认为是搭建异构系统的理想硬件设备。由于GPU硬件架构的特点，如：晶体管数量庞大，只有少数逻辑控制和缓存部分，特别适合用来做高任务量且逻辑相对独立的矩阵运算。所以CPU是专门为顺序串行处理而优化的几个核心组成，而GPU则有用一个数以千记的更小、更高效的核心组成的大规模并行计算架构[18]。

NVIDIA公司于2007年发明了基于CUDA[19]的并行编程生态系统，它对C语言进行了扩展，实现了异构系统模型，因为CUDA提供了一整套开发生态系统，包括基于编译器，调试器，调优器，集成开发环境[20]，和各类加速应用数学库等，让基于GPU的编程变得从此简单。

在2017年GPU技术会议上，NVIDIA向全球发布了CUDA 9，最新版本的CUDA提供了空前强大的并行计算平台和编程模型。比如合作组织，它重新定义了在kernel内部线程启动机制。还有针对NPP运行库的优化，相比Intel公司的多核至强系列处理器，NVIDIA GPU在图像处理方面的性能提升了20-100倍。

5.3研究目标

本文通过对国内外有关虚拟化技术，容器技术，Docker，异构系统，高性能系统，以及基于最新Volta架构的CUDA的研究并将其成果梳理总结。继而研究和阐述Cholesky[21]和LU[22]分解算法，PageRank算法[23]，通过基于Volta架构的GPU和CUDA编程模型相结合，将这些算法部署到GPU上进行加速并发运算，大大提高计算速度，实现了方法优化。异构系统的环境部署及其复杂，耗费时间和人力，我们希望在目前市场上已有的虚拟化技术将异构编程模型封装到容器内部，加快开发和运行环境的部署，让工程师们更加专注在功能的开发是实现上，同时运维人员也不需要耗费过多精力在搭建运行环境上。

5.4研究内容

本文主要研究了虚拟化技术，容器技术，异构系统，CUDA的最新编程特性，通过对Docker的扩展，将运行在异构系统上的高性能程序如Cholesky，LU等分解算法和PageRank算法等移植进入容器内，并充分利用GPU的硬件资源。

本文分析了虚拟机和容器的区别和其中各自优劣，通过研究发现一种基于Docker的容器就像一种沙盒。每个容器内运行一个应用，不同的容器相互隔离，容器之间也可以建立通信机制。容器的创建和停止都十分快速，容器自己对资源的需求也十分有限，远远低于虚拟机。基于GPGPU的异构系统，最初GPGPU只能处理图像计算任务，随着统一渲染架构的出现，GPU的可编程性也随之加强，越来越多的研究会利用GPU来完成非图像计算的任务。目前主流的异构系统基于GPU通用计算架构主要使用NVIDIA的CUDA架构、Khronos Groupde OpenCL[24]（Open Computing Language，开放的计算语言）以及微软的Direct Compute（DirectX的GPGPU解决方案）[25]。目前CUDA是最流行的GPU通用计算架构，同时它也是目前较为完整的GPU通用计算解决方案。

CUDA研究，了解了CUDA是NVIDIA的GPGPU模型，为加速图像的实时处理而设计的一种运行在GPU上的开发平台，NVIDIA公司推出该平台的目的是让用户可以利用GPU加速解决复杂的计算问题。CUDA通常采用CPU+GPU的异构架构，充分调度与发挥CPU和GPU的合作，让GPU成为CPU的协作处理器，负责处理并行计算中的大规模运算，充分利用了两种各自的优势[26]。

5.5主要研究方案、技术路线与可行性分析

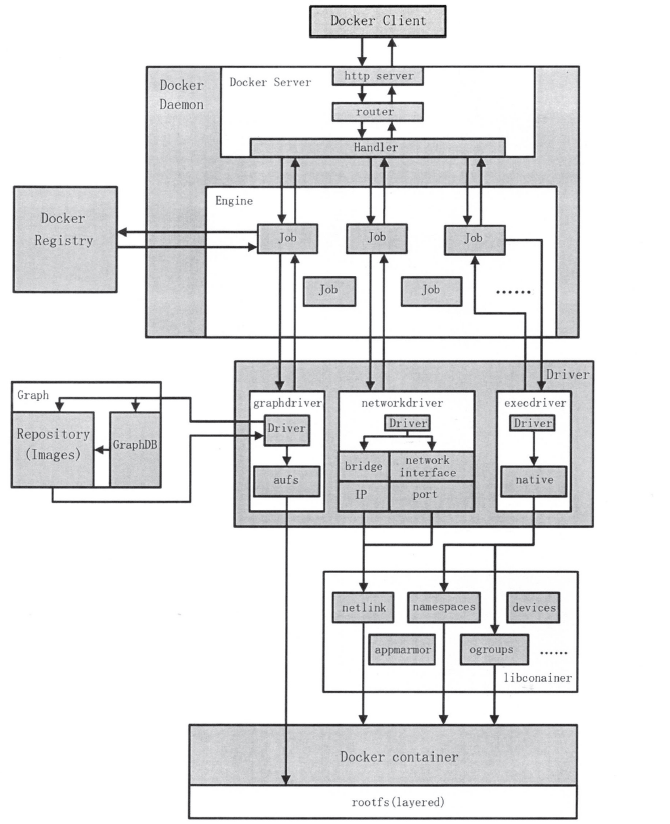
Docker技术研究：

Docker定义了一种统一标准的打包格式，将应用及其依赖打包进单个镜中，该镜像可以在任何Docker可运行的机器之间进行便捷移植，并且在不同机上，应用与平台将独立开来，这意味着不管在什么平台下，Docker容器中运行的应用其运行环境都完全相同。镜像是Docker容器运行的基础，当容器通过镜像创建之后，系统会在镜像层上创建一个可写层，每一次变更将记录到容器的文件系统中，而不会对镜像产生任何形式的改变[30]。

Docker服务端是通过Docker Daemon守护进程来与用户进行通信并对程序完成调度工作。其中的Docker server提供接口来与用户Docker C1ient服务进行交互，并通过路由器找到相应的Handler，最终调度Engine来执行工作[16]。Docker中的每件工作以Job的形式存在的。

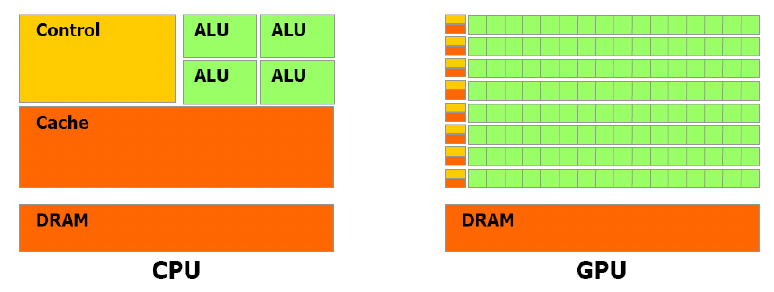
Docker Daemon需要完成的任务种类很多，若需要下载容器镜像，则会创建一个名为“pull”的Job，运行时从Docker Registry中下载镜像，并通过镜像管理驱动graph driver将下载的镜像存储在graph中；若用户需要为Docker容器创建网络环境，则会创建一个名为“a11ocate interface”的Job，通过网络驱动network driver分配网络接口的资源[31]。

libcontainer是一套独立的容器管理解决方案，这套解决方案涉及了大量Linux内核方面的特性，如：namespaces、cgroups以及capabilities等。libcontainer很好地抽象了Linux内核特性，并提供完整、明确的接口给Docker Daemon。当用户执行运行容器这个命令之后，一个Docker容器就处于运行状态，该容器拥有隔离的运行环境、独立的网络栈资源以及其他受限的资源等[32]。下图是关于Docker的整体架构图：



CUDA技术研究：

CPU和GPU的硬件结构的差异造成了它们之间运算能力的差异，如下图所示，GPU将芯片中更多的晶体管用来进行数据处理，而CPU是用很多晶管作为数据缓存和进行流控制。因而GPU是特别为计算密集度高，具有高并行度的计算（如同图像渲染）设计的。



从硬件设计图上可以看到，GPU非常适合处理具有低相互依存度，高可并行度的问题，计算密度非常高。由于同一程序在每个计算部分上执行，因此对负载流控制的要求非常少，更为在多个计算部分上同时执行并且计算密度高，访存延迟可以被计算隐藏，因此用不着大的数据缓存。CUDA全称是Compute Unified Device Architecture，是Nvidia 公司在2007 年推出的一种在GPU上进行通用计算的架构。这是一种可以使用类C语言而不再需要进行图形的API映射进行开发的通用计算开发环境和软件体系[33]。CUDA是一个完整的以GPU作为运算工具的并行计算架构，包含由底层的硬件部分和基于硬件之上的软件部分。硬件部分是支持CUDA计算架构的具有多个图形处理流处理核心的GPU，软件部分包括编译工具、驱动Driver API 库、运行时runtime API库和高层数学函数库。在CUDA架构下，开发者为了充分利用GPU中的硬件资源的并行计算能力，通常采取使用大量的并发线程进行并行计算的方式。这些并发线程的创建和管理在CUDA中是依靠硬件来实现的，软件的执行时间并不会被占用。访问GPU中硬件资源在CUDA中是通过调用接口函数来实现访问的，这些接口函数包含在运行时runtime API库中。CUDA开发环境中提供了DRMA内存寻址方式，这是一种在C语言中通用的寻址方式。故而CUDA C是一种扩展的C语言，拥有极强的编程灵活性。操作系统负责管理所有访问GPU的程序，不论是基础的图形应用程序，还是利用GPU进行通用计算的多个并发运行的CUDA程序。

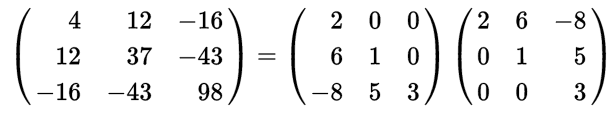
在CUDA的编程模型中，CPU是主机（Host）或称为宿主，GPU是计算设备（Compute Device），是CPU的协处理器。在GPU上可以并行运行多个线程。CUDA程序由专门的编译器nvcc编译后，分成了两个部分：主机部分和设备部分。主机部分运行在CPU上，设备部分运行在GPU上。主机部分一般为控制程序执行流程的串行程序，设备部分则是负责计算的密集性并行程序[34]。

设备（Device）端程序实际上是由大量的并发线程组成，这些线程由GPU硬

件调度，在不同的流处理器分布执行。在CUDA中，线程（Thread）、线程块（Block）和线程网格（Grid）构成了线程组织的三个层次[35]。由线程块组成线程网格，线程块可以是三种维度的一维、二维或者三维，在不同的流处理器簇上执行不同的线程块；线程块是一维或者二维的，由流处理器簇内不同的流处理器执行同一线程块内的线程。大量的并发线程能够在GPU 上同时执行是CUDA 的优势所在，提高CUDA应用性能的基本手段是增加并发线程的数目。

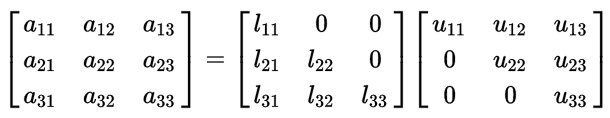
CUDA MPS，多进程服务（MPS）是CUDA应用程序编程接口（API）的二进制兼容实现。 MPS运行时体系结构旨在透明地启用合作多进程CUDA应用程序（通常是MPI作业），以在最新的基于NVIDIA的GPU上使用Hyper-Q功能。 Hyper-Q允许在同一GPU上同时处理CUDA内核; 当GPU计算容量由单个应用程序未充分利用时，这可以使性能受益。Volta架构引入了新的MPS功能。 与Volta GPU前的MPS相比，Volta MPS提供了一些关键的改进：Volta MPS客户端直接向GPU提交工作而不通过MPS服务器。 每个Volta MPS客户端拥有自己的GPU地址空间，而不是与所有其他MPS客户端共享GPU地址空间。 Volta MPS支持服务质量（QoS）的有限执行资源配置。

Cholesky定义：给定一个对称正定矩阵A，则存在主对角元素全为正数的下三角矩阵L满足：A=LLT，通常称下三角矩阵L为矩阵A的Cholesky因子。其中L是具有实数和正对角项的下三角矩阵，LT表示L的共轭转置。每个Hermitian正定矩阵（以及每个实值对称正定矩阵）都具有独特的Cholesky分解[27]。如果矩阵A是Hermitian和正半定的，则如果允许L的对角条目为零，则它仍然具有形式为A = LLT的分解。下面对称实矩阵的Cholesky分解：



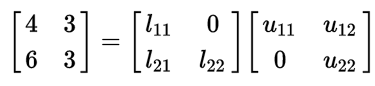
Cholesky分解主要用于线性方程的数值解Ax=B，如果A是对称和正定的，那么我们可以通过首先计算Cholesky分解A = LLT来求解Ax = b。然后通过正向替换求解Ly = b，最后通过反向替换求解x的LTx=y。对于可以置于对称形式的线性系统，Cholesky分解（或其LDL变体）是一种可以被选择的方法，因为它具有优异的效率和数值稳定性。与LU分解相比，它的效率大约快两倍。

LU定义：令A为方阵。 LU分解是指A的分解，具有适当的行和/或列排序或排列成两个因素，即下三角矩阵L和上三角矩阵U，A=LU，在下三角矩阵中，对角线以上的所有元素都为零，在上三角矩阵中，对角线以下的所有元素均为零[29]。 例如，对于3×3矩阵A，其LU分解看起来像这样：

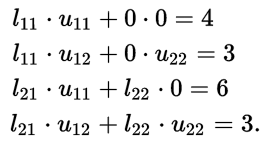


在矩阵中没有正确的排序或排列，因式分解可能无法实现。 例如，很容易通过扩展矩阵乘法来验证a11=l11u11。 如果a11=0，则l11u11中至少有一个必须为零，这意味着L或U是单数。如果A是非奇异的（可逆的），这是不可能的，可以通过简单地重新排序A的行来移除它，以便排列的矩阵的第一个元素不为零。

我将以下2乘2矩阵进行因式分解来举例：



找到这个简单矩阵的LU分解的一种方法是通过检查简单求解线性方程。扩展矩阵乘法如下所示，



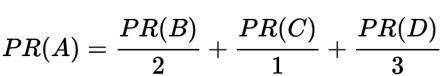
这个方程式是不确定的。 在这种情况下，L和U矩阵的任何两个非零元素都是解的参数，并且可以任意设置为任何非零值。 因此，为了找到唯一的LU分解，有必要对L和U矩阵进行一些限制。 例如，我们可以方便地要求下三角矩阵L成为单位三角矩阵（即将其主对角线的所有条目设置为一个）。

PageRank：PageRank是一种链接分析算法，它为诸如万维网的超链接文档集的每个元素分配数字加权，目的是“测量”其集合内的相对重要性[30]。该算法可以应用于具有相互引用和引用的实体的任何集合。它分配给任何给定元素E的数字权重称为E的PageRank，由PR(E)来表示。PageRank是基于由所有万维网页面作为节点和超链接创建的网页的数学算法，考虑到权限集线器，如cnn.com或usa.gov。等级值表示特定页面的重要性。一个页面的超链接将被视为支持投票。一个页面的PageRank被递归地定义，并且取决于链接到它的所有页面的数量和PageRank度量（“传入链接”）。由多页与较高PageRank相关联的页面本身具有高排名。

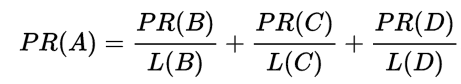
如果系统中唯一的链接是从B，C和D到A，每个链接将在下一次迭代时将0.25 PageRank转移到A，总共为0.75。



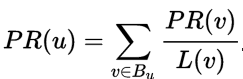
假设页面B链接到页面C和A，则页面C具有到页面A的链接，页面D链接到所有三个页面。因此，在第一次迭代时，页面B将其现有值的一半（0.125）转移到页面A，将另一半（0.125）传送到页面C。页面C将其所有现有值0.25传递到唯一 页面链接到A.由于D有三个出站链接，它会将其现有价值的三分之一（或大约0.083）转移到A.在此迭代完成后，第A页的PageRank约为0.458。



换句话说，出站链接授予的PageRank等于文档自己的PageRank得分除以出站链接数L()：



在一般情况下，任何页面的PageRank值可以表示为：



技术路线总结：首先研究Docker以及基于Docker的插件开发案列，搭建开发环境。我们知道Docker是基于Go语言编写的，Go语言是最近发展起来的开发语言，Docker的插件又需要用Go来编写，所以本人需要先学习掌握Docker插件的开发机制，并能熟练运用Go语言编写程序。另外，高性能架构CUDA已经发布了第九个版本，其中也有很多新技术，比如合作组织，全新的统一内存功能，结合研究的算法，在实现方法上进行优化，利用CUDA最新特性，发挥GPU最大性能。另外CUDA的环境部署极为复杂并且耗费时间，高性能计算本身对性能和可靠性的要求极高，传统的虚拟化技术不能满足需求。现在需要把Docker扩展出来的插件配合硬件，将整个开发运行环境部署到容器内部，并且又能充分发挥GPU特性，将其整合为一个可用系统，解决实际需求，事实上在实际工程中，本人还遇到各种版本编译器的测试问题，本人也在主机上安装过其他版本的编译器，过程十分复杂繁琐，如果能无缝地将系统移植进入容器，这个工作难点在今后也被迎刃而解。本篇论文从实际需求出发，研究了容器和异构系统并将其扩展，之后需要把成熟算法移植到新的硬件上实现并做对比测试，目前市场上Volta GPU的良品率极低且价格昂贵，但由于本人供职于英伟达公司并且支持全球客户，所以可以搭建出CUDA 9.0+Volta的环境。

5.6预期研究成果和（或）创新点

1. CUDA 9.0 + Volta是上个月才发布的异构系统架构，在此环境下实现算法移植是之前研究人员没有实施过的。
2. Docker的扩展已经有很多，但是针对英伟达专有GPU的优化并没有，本论文讨论并需要实现对专有硬件的支持会与前人不同。

5.7工作基础和条件

基于本人供职的公司和工作内容，已经搭建出CUDA+GPU的工作实验环境，平时工作也基于CUDA C开发和测试应用于GPU的程序，具备可行的实验环境。目前还需要继续研究Docker和Go语言，同时CUDA的最新特性也需要加强理解，才能将最新硬件发挥出最大功效。

5.8研究进度安排

第一季度：CUDA 9.0 学习研究，CUDA C开发环境搭建，CUDA新特效单元模块测试。

第二季度：容器技术深入研究，Go语言学习和开发环境搭建，Docker插件开发技术实战拓展，使得容器内部可以访问外接设备，这里我们主要考虑GPU。

第三季度：算法学习研究，将Cholesky，LU分解算法和PageRank算法移植性学习和实现。参考前人的移植方法，将上诉算法用CUDA C在基于Volta架构的GPU上实现。

第四季度：系统整合阶段，测试验证阶段。让系统在主机端和容器端分别运行，证明其性能不会降低并且部署环境时间被大量节约。提交Docker的定制镜像文件到云端，方便以后工作中使用。

**方法措施**：(1）通过网络、图书馆、期刊查阅相关资料。  
 (2）和老师、同学进行相关的交流。  
 (3）运用所学知识以及前沿理论展开论述。   
 (4）尽量案例、数字和图文结合。

文献参考

1. Nanda S, Chiueh T C. A Survey on Virtualization Technologies[J]. Rpe Report, 2005.
2. J. Shafer, S. Rixner, and A. L. Cox, “The hadoop distributed filesystem: Balancing portability and performance,” in Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on. IEEE, 2010, pp. 122–133.
3. JP Walters, V. Chaudhary, M. Cha, S. Guercio Jr, and S. Gallo, "A comparison of virtualization technologies for HPC," in Advanced Information Networking and Applications, 2008. AINA 2008. 22nd International Conference on. IEEE, 2008, pp. 861-868.
4. N. L. Ensmenger, The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise (History of Computing).The MIT Press, 2010.
5. R. Morabito, J. Kj¨allman, and M. Komu, “Hypervisors vs. lightweight virtualization: a performance comparison,” in Cloud Engineering (IC2E) 2015 IEEE International Conference on. IEEE, 2015, pp. 386–393.
6. L. Benedicic, M. Gila, S. Alam, and T. Schulthess, “Opportunities for container environments on Cray XC30 with GPU devices,” in Cray Users Group Conference (CUG’16), 2016.
7. Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield, "Xen and the art of virtualization," in Proceedings of the nineteenth ACM Symposium on Operating systems principles. , pp. 164-177, ACM Press.
8. Y. Allusse, P. Horain, A. Agarwal and C. Saipriyadarshan, GpuCV: A GPU-accelerated framework for image processing and computer vision, in: Intl. Symp. on Advances in Visual Computing, Springer-Verlag, Berlin, 2008, pp. 430–439.
9. D. M. Jacobsen and R. S. Canon, “Shifter: Containers for HPC,” in Cray Users Group Conference (CUG’16), 2016.
10. W. Li, X. Wei, and A. Kaufman. Implementing lattice Boltzmann computation on graphics hardware. Visual Computer, 19(7-8):444–456, December 2003.
11. Linux Containers project, “LXC 1.0,” available at: https: //linuxcontainers.org/lxc/ (Jan. 2017).
12. Lombardi F, Pietro R D. CUDACS: Securing the Cloud with CUDA-Enabled Secure Virtualization[C]// International Conference on Information and Communications Security. Springer-Verlag, 2010:92-106.
13. Beimborn D, Miletzki T, Wenzel S. Platform as a Service (PaaS)[J]. Business & Information Systems Engineering, 2011, 53(6):371-375.
14. Merkel D. Docker: lightweight Linux containers for consistent development and deployment[J]. 2014, 2014(239).
15. Soltesz S, Pötzl H, Fiuczynski M E, et al. Container-based operating system virtualization[J]. Acm Sigops Operating Systems Review, 2007, 41(3):275-287.
16. Bernstein D. Containers and Cloud: From LXC to Docker to Kubernetes[J]. IEEE Cloud Computing, 2015, 1(3):81-84.
17. Love R. Linux Kernel Development, 2nd Edition[J]. 2005.
18. Asanovic K, Bodik R, Catanzaro B C, et al. The Landscape of Parallel Computing Research: A View from Berkeley[J]. Technical Report Uc Berkeley, 2006, eecs-2006-183.
19. Sanders J, Kandrot E. CUDA by Example: An Introduction to General-Purpose GPU Programming[M]. Addison-Wesley Professional, 2010.
20. 佚名. 集成开发环境[J]. 计算机应用文摘, 2004(5).
21. George A, Heath M T, Liu J, et al. Sparse Cholesky factorization on a local-memory multiprocessor[J]. Siam Journal on Scientific & Statistical Computing, 1988, 9(2):327-340.
22. Davis T A, Duff I S. An Unsymmetric-Pattern Multifrontal Method for Sparse LU Factorization[J]. Siam Journal on Matrix Analysis & Applications, 2006, 18(1):140-158.
23. Haveliwala T H. Topic-sensitive PageRank: a context-sensitive ranking algorithm for Web search[J]. Knowledge and Data Engineering, IEEE Transactions on, 2003, 15(4):784-796.
24. Munshi A, Gaster B, Mattson T G, et al. OpenCL Programming Guide[J]. 2011.
25. 张益男, 袁杰. DirectCompute加速图像处理方法的研究[J]. 现代电子技术, 2012, 35(22):55-58.
26. 成思远. 异构(CPU-GPU)计算机系统性能评测与优化技术研究[D]. 国防科学技术大学, 2011.
27. 包金山. 交换正定Hermitian矩阵平均矩阵的半正定性[J]. 内蒙古民族大学学报(自然汉文版), 2000(1):14-16.
28. 黄德才, 戚华春. PageRank算法研究[J]. 计算机工程, 2006, 32(4):145-146.
29. 张永杰, 孙秦. 大型稀疏线性方程组符号LU分解法[J]. 计算机工程与应用, 2007, 43(28):29-30.
30. 王德广, 周志刚, 梁旭. PageRank算法的分析及其改进[J]. 计算机工程, 2010, 36(22):291-292.
31. 伍阳. 基于Docker的虚拟化技术研究[J]. 信息技术, 2016(1):121-123.
32. 王飞. 基于Docker的研发部署管理平台的设计与实现[D]. 北京交通大学, 2015.
33. Che S, Boyer M, Meng J, et al. A performance study of general-purpose applications on graphics processors using CUDA[J]. Journal of Parallel & Distributed Computing, 2008, 68(10):1370-1380.
34. Manavski S A, Valle G. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment[J]. Bmc Bioinformatics, 2008, 9 Suppl 2(2):S10.
35. Bell N, Garland M. Efficient Sparse Matrix-Vector Multiplication on CUDA[J]. 2008.