基于Docker的高性能并行异构系统设计与实现

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，我们使用的容器就会独立于硬件和平台本身。但是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设备，并且需要安装英伟达驱动程序。因此，Docker Engine本身不可以直接访问到英伟达图形加速器(Nvidia GPU)并发挥其效率。

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。为了获得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，运维和测试人员也可以轻松地集成和隔离加速应用程序，无需任何修改，直接将其部署在任何支持GPU的基础设施上,极大地节省了资源。

Abstract

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

Contents

[Contents 2](#_Toc509347885)

[第一章 绪论 5](#_Toc509347886)

[1.1 课题的背景介绍 5](#_Toc509347887)

[1.2 国内现状和国外研究进展 6](#_Toc509347888)

[1.3 论文的创新点 9](#_Toc509347889)

[1.4 论文的主要目标 9](#_Toc509347890)

[1.5 本文的组织结构 9](#_Toc509347891)

[1.6 本章小结 9](#_Toc509347892)

[第二章 系统相关技术概述 9](#_Toc509347893)

[2.1 Docker虚拟化容器技术 9](#_Toc509347894)

[2.2 高性能异构平台 9](#_Toc509347895)

[2.3 Postgresql数据库技术 10](#_Toc509347896)

[2.4系统开发所用语言 10](#_Toc509347897)

[2.4.1 Go编程语言 10](#_Toc509347898)

[2.4.2 Python编程 10](#_Toc509347899)

[2.5 基于MVC的Django框架 11](#_Toc509347900)

[2.6 前端相关技术 11](#_Toc509347901)

[2.6.1 html 11](#_Toc509347902)

[2.6.2 css 11](#_Toc509347903)

[2.6.3 jquery 11](#_Toc509347904)

[2.6.4 wsgi 11](#_Toc509347905)

[2.6.5 nginx 11](#_Toc509347906)

[2.7 本章小结 11](#_Toc509347907)

[第三章 基于CUDA的高性能平台研究 11](#_Toc509347908)

[3.1 CUDA编程模型与环境 12](#_Toc509347909)

[3.1.1 CUDA接口 12](#_Toc509347910)

[3.1.2 CUDA中的线程 13](#_Toc509347911)

[3.1.3 CUDA设备端的实现 14](#_Toc509347912)

[3.2 Cooperative Groups(合作组) 16](#_Toc509347913)

[3.3 动态并行(Dynamic Parallelism) 17](#_Toc509347914)

[3.3.1 执行环境 17](#_Toc509347915)

[3.3.2 父子网格 18](#_Toc509347916)

[3.3.3 同步 18](#_Toc509347917)

[3.3.4 排序和并发 18](#_Toc509347918)

[3.5 GPU P2P ？？？ 19](#_Toc509347919)

[3.6 NVLINK ？？？ 19](#_Toc509347920)

[3.7 Unified memory(统一内存) 19](#_Toc509347921)

[3.8 CUDA tools 20](#_Toc509347922)

[3.8.1 NVCC 20](#_Toc509347923)

[3.8.2 Cuda-memcheck 21](#_Toc509347924)

[3.8.3 NVVP 21](#_Toc509347925)

[3.8.4 NSIGHT 21](#_Toc509347926)

[3.9系统硬件设计|基于Volta架构的硬件平台 (可能考虑放在第四章) 21](#_Toc509347927)

[第四章 容器内部的异构系统及Web服务 23](#_Toc509347928)

[4.2 Docker容器的外部扩展 23](#_Toc509347929)

[4.3 基于Django的Web框架开发 24](#_Toc509347930)

[4.4 后端数据库服务 24](#_Toc509347931)

[4.5 前端服务部署 24](#_Toc509347932)

[第五章 高性能异构系统的实现和测试 24](#_Toc509347933)

[5.1 可访问GPU的Docker容器实现 24](#_Toc509347934)

[5.2 Cholesky(PageRank)算法在系统中的实现 24](#_Toc509347935)

[5.3 性能调优(NVVP, cuda-memcheck) 24](#_Toc509347936)

[5.4 Linpack测试系统 24](#_Toc509347937)

[5.5 基于Django开发的Web服务 24](#_Toc509347938)

[第六章 结论与展望 24](#_Toc509347939)

[6.1 总结 24](#_Toc509347940)

[6.2 展望 24](#_Toc509347941)

目录

1. 绪论
   1. 课题背景和意义
   2. 国内现状和国外研究进展
   3. 课题研究的创新点
   4. 论文的主要目标
   5. 本文的组织结构
2. 系统相关技术研究

2.1 虚拟化容器技术

2.2 高性能异构技术

2.3 Postgresql数据库技术

2.4 Go编程

2.5 基于MVC的Django框架

2.6 Python编程

2.7 前端相关技术

1. 基于CUDA的高性能平台设计

3.1 CUDA编程模型与环境

3.2 合作组织

3.3 动态平衡度

3.4 NVML

3.5 GPU P2P

3.6 GPUDirect or NVLINK

3.7 基于Volta架构的硬件平台

第四章 容器内部实现的异构系统及Web服务

4.1 Docker容器的组织结构

4.2 Docker容器的外部扩展

4.3 基于Django的Web框架

4.4 数据库服务

4.5 前端服务

第五章 高性能异构系统的实现和测试

5.1 可访问GPU的Docker容器

5.2 Cholesky算法实现及移植

5.3 性能调优设计

5.4 Linpack测试系统

5.5 基于Django开发的Web服务

第六章 结论与展望

6.1 总结

6.2 展望

致谢

第一章 绪论

1.1 课题的背景介绍

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。

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。

本文从实际需求出发，结合基于Volta硬件架构与CUDA的配合使用，采用英伟达公司GPU来满足人民日益增长的对高性能计算的需求。在公司内部，我们又需要快速搭建异构环境，将平时工作中对GPU的测试记录写入数据库，并对其进行纵向对比，从而保证公司产品不会发生回归性质的性能衰退。本人使用市面上多种技术结合，选择Docker作为基础容器，将应用算法封装进入容器，并采用基于Django的web开发框架配合后端postgresql数据库和前端nginx服务器，为用户提供快速简便的操作界面，并提供稳定长久的数据支持。

1.2 国内现状和国外研究进展

1.2.1 虚拟化技术现状

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（ChristopherStrachey）发表了一篇学术报告，名为《大型高速计算机中的时间共享》（TimeSharinginLargeFastComputers），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。

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。

1.2.2 高性能计算平台现状

自1946年第一台电子计算机ENIAC问世至今，超级计算机的发展已先后经历了5个阶段或5代，即早期的单处理器巨型机、向量处理系统、大规模并行处理系统、共享内存处理系统和机群系统。

1983年，中国第一台被命名为“银河”的亿次巨型电子计算机在国防科技大学诞生。它的研制成功向全世界宣布：中国成了继美、日等国之后，能够独立设计和制造巨型机的国家。

1992年，国防科技大学研制出银河－II通用并行巨型机，峰值速度达每秒10亿次，主要用于中期天气预报。

1993年，国家智能计算机研究开发中心（后成立北京市曙光计算机公司）研制成功曙光一号全对称共享存储多处理机，这是国内首次以基于超大规模集成电路的通用微处理器芯片和标准UNIX操作系统设计开发的并行计算机。 1995年，曙光公司又推出了曙光1000，峰值速度每秒25亿次浮点运算，实际运算速度上了每秒10亿次浮点运算这一高性能台阶。曙光1000与美国Intel公司1990年推出的大规模并行机体系结构与实现技术相近，与国外的差距缩小到5年左右。

1997年，国防科技大学研制成功银河－III百亿次并行超级计算机系统，峰值性能为每秒130亿次浮点运算。

1997至1999年，曙光公司先后在市场上推出曙光1000A，曙光2000－I，曙光2000－II超级服务器，峰值计算速度突破每秒1000亿次浮点运算。 1999年，国家并行计算机工程技术研究中心研制的神威I计算机，峰值运算速度达每秒3840亿次，在国家气象中心投入使用。

2004年，由中科院计算所、曙光公司、上海超级计算中心三方共同研发制造的曙光4000A实现了每秒10万亿次运算速度。

2008年，“深腾 7000”是国内第一个实际性能突破每秒百万亿次的异构系统集群系统，Linpack能突破每秒106.5亿次。

2008年，曙光5000A实现峰值速度230万亿次、Linpack值180万亿次。作为面向国民经济建设和社会发展的重大需求的网格超级服务器，曙光5000A可以完成各种大规模科学工程计算、商务计算。

2009年10月29日，中国首台千万亿次超级计算机“天河一号”诞生。这台计算机每秒1206万亿次的峰值速度和每秒563．1万亿次的Linpack实测性能，使中国成为继美国之后世界上第二个能够研制千万亿次超级计算机的国家。 2011年11月，中国第一台实测性能超千万亿次的超级计算机曙光“星云”在国家深圳超算中心全面开通运行。“星云”系统峰值为每秒3000万亿次，实测Linpack性能达到每秒1271万亿次，是世界第三台、中国第一台实测双精度浮点计算超过千万亿次的超级计算机。

由日本政府出资、富士通制造的巨型计算机“K Computer”目前落户于日本理化研究所，并成功从中国手中夺回运算速度排行榜第一的宝座。“K Computer”目前的运算速度为每秒8000万亿次，而到2012年其完全建成时，运算速度将达到每秒一万万亿次。“K Computer”比现居第二的中国超级计算机速度快出约3倍，甚至比排名第2至第6的计算机运算速度总和还要快。

进入新世纪，随着研制高端计算机系统的诸多关键技术被攻克(尤其是机群技术)，我国自行研制的高端计算机系统已开始形成自己的品牌系列和一定的市场规模，其发展呈现星火燎原之势头。近两年，随着“神威”、“银河”、“曙光”、“深腾”、“天梭”等一批知名产品的出现，使我国成为继美、日之后第三个具备高端计算机系统研制能力的国家，被誉为世界未来高端计算市场的“第三股力量”。

1.2.3 容器技术现状

Docker利用Linux 的一些内核机制例如 [cGroups](https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt)、命名空间和 [SElinux](http://selinuxproject.org/page/Main_Page) 来实现容器之间的隔离。起初 Docker 只是 [LXC](https://linuxcontainers.org/) 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/)，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。

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。容器会比虚拟机更高效，因为它们能够分享一个内核和分享应用程序库。

最初一家法国公司DotCloud提供PaaS服务，它能对支持多种语言的运行环境，如Ruby、Java、Python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，所以就考虑将Docker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上收到超过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。

1.2.4 异构系统现状

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。基于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。

NVIDIA公司于2007年推广了基于CUDA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，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内部线性启动机制。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，在图像处理方面的性能提升了20-100倍。

1.3 论文的创新点

重点思路

支持访问NVIDIA GPU的Docker容器

算法的方法优化，在GPU上实现的算法方法优化，如Cholesky分解

待扩展

1.4 论文的主要目标

重点思路

Django的定制化服务，封装在容器内部，服务于测试任务

深入理解Votla的硬件特性

基于Go语言的Docker应用扩展

MVC模型的编程思想

前端框架实现订制服务

后端数据库模型设计

待扩展

1.5 本文的组织结构

待完善

1.6 本章小结

第二章 系统相关技术概述

2.1 Docker虚拟化容器技术

虚拟化技术是提高现有计算资源利用率的常用策略，传统上，使用它就意味着为了获得虚拟机的便利而需要接受性能大幅降低的代价。如今，这种性能上的降低已经得到很大的缓解。由于更快的处理器的出现以及更高效的虚拟化解决方案的诞生，现在我们也能把一台普通的台式计算机作为虚拟机的宿主机来使用。

Docker是一个开源项目，它基于来自操作系统研究的许多熟悉的技术：LXC容器，操作系统虚拟化，以及基于散列或类似git的版本控制和差异系统。

在日常工作中，处理客户问题要求能快速搭建出和客户问题想匹配的相关系统环境。另外在测试工作中，亦要求工程师能提高工作效率，不要把过多的精力放在搭建系统环境上，而是能更深入的对平台和GPU进行验证。

待扩展改进

2.2 高性能异构平台

支持高性能计算的平台十分宽泛，比如从台式计算机到大型并行处理系统都可以作为高性能计算的宿主平台。

由于GPU在硬件设计上有着对并行运算的天然优势，科学家和研究人员越来越依赖异构系统作为他们的底层运算基础设施，其中就把GPU作为核心处理器（比如CPU）的协作处理器来使用。一般流程中还是使用CPU控制复杂的逻辑过程，当遇到工作量很大但是处理单元又相对独立的大规模运算时候，可以通过并行框架提供的接口将这些任务分配给异构系统中的协处理器去运算，CPU端只需要等待需要的运算结果，然后继续后面的任务。这种组织架构极大的加速了核心单元的运算速度，但是由于异构系统中编程难度的加大，又容易增加使用成本。为了降低使用异构系统的成本和开发难度，行业设计公司需要提供软件开发基础设施，例如开发异构系统语言，但是这种语言又不能与市面上流行使用的编程语言有过大区别，另外还需要提供软件开发库，就像OpenGL的库一样，使得开发人员不需要过多了解底层硬件设计环节。事实上，类似于线程构建模块（TBB），OpenMP，CUDA这样的系统已经被越来越多的科学工程应用并加快了实现过程。本文着重研究基于NVNDIA公司的CUDA系统平台作为高性能计算系统的基础，后续实验都部署在此系统上完成。

待扩展改进

2.3 Postgresql数据库技术

在本系统后端我们使用Postgresql数据库来记录平时工作中的数据。现在被称为PostgreSQL的对象关系型数据库管理系统是从加州大学伯克利分校（University of California at Berkeley）编写的POSTGRES包中派生出来的。经过二十多年的开发，PostgreSQL成为了现在最先进的开源数据库。

美国国防部高级研究计划局（DARPA），陆军研究办公室（ARO），美国国家科学基金会（NSF）和ESL公司共同发起了由迈克尔·斯通布拉克教授领导的POSTGRES项目。POSTGRES起始与1986年，在POSTGRES的设计中给出了系统的最初概念，POSTGRES数据模型中出现了初始数据模型的定义。 POSTGES规则系统的设计中描述了当时规则系统的设计。在POSTGRES存储系统的设计中详细介绍了存储管理器的基本原理和体系结构。

待扩展改进

2.4系统开发所用语言

2.4.1 Go编程语言

Go编程语言是一个开源项目，可以使程序员的工作效率更高。

Go语言语法上具有清晰，简洁，干净，高效等特点。其并发机制使编写充分利用多核和联网机器的程序变得容易，而其新颖类型的系统则可实现灵活的模块化程序构建。 快速编译为机器代码，但具有垃圾收集的便利性和运行时反射的能力。 它是一种快速，静态类型的编译语言，感觉像是一种动态类型的解释型语言。

Docker开源容器也是使用Go语言编写。另外本系统需要为Docker开发可访问GPU的扩展程序。

2.4.2 Python编程

Python是一种广泛使用的高级编程语言，属于通用型编程语言，由吉多·范罗苏姆 创造，第一版发布于 1991 年。可以视之为一种改良 (加入一些其他编程语言的优点，如面向对象) 的 LISP。作为一种解释型语言，Python 的设计哲学强调代码的可读性和简洁的语法（尤其是使用空格缩进划分代码块，而非使用大括号或者关键词）。相比于 C++ 或 Java，Python 让开发者能够用更少的代码表达想法。不管是小型还是大型程序，该语言都试图让程序的结构清晰明了。

与 Scheme、Ruby、Perl、Tcl 等动态类型编程语言一样，Python 拥有动态类型系统和垃圾回收功能，能够自动管理内存使用，并且支持多种编程范式，包括面向对象、命令式、函数式和过程式编程。其本身拥有一个巨大而广泛的标准库。

Python 虚拟机本身几乎可以在所有的操作系统中运行。

待扩展改进

2.5 基于MVC的Django框架

Django是一个开放源代码的Web应用框架，由Python写成。采用了MVT的软件设计模式，即模型Model，视图View和模板Template。它最初是被开发来用于管理劳伦斯出版集团旗下的一些以新闻内容为主的网站的。并于2005年7月在BSD许可证下发布。Django的主要目标是使得开发复杂的、数据库驱动的网站变得简单。Django注重组件的重用性和“可插拔性”，敏捷开发和DRY法则（Don't Repeat Yourself）。在Django中Python被普遍使用，甚至包括配置文件和数据模型。

待扩展改进

2.6 前端相关技术

2.6.1 html

HTML即超文本标记语言，在没有超文本的时候，用文本来传输信息，但如果希望美化文本，例如文本加粗，换个颜色，又或者别出心裁的想在文本上传输图片，此时普通文本就力不从心了，于是超文本诞生了。超文本的用途非常多，最常见的就是描述一个网页，又或者是描述一个Word文档。

超文本语言构造的文件可以在本地或者Internet上打开，通过浏览器解释执行，去除了繁琐的编译过程，浏览器会将HTML渲染成我们希望的页面，每个浏览器不尽相同，不过大多都遵循W3C规范。

HTML语言需要按照规范的结构，当然有时浏览器也并不十分严格。结构的基本单位称为标记，标签一般都是成对出现，开始加上闭合代表一个节点标签。标记符<HTML>代表整个HTML文件的开始，而</HTML>代表结束。

HTML发展至今已经升级为HTML5.0，现代浏览器已经很好的支持了最新特性，例如炫酷的canvas、矢量图SVG、本地化的WEB存储以及一些最新的API。根据W3C的发言稿称：“HTML5是开放的Web网络平台的奠基石。”

2.6.2 css

2.6.3 jquery

2.6.4 wsgi

2.6.5 nginx

2.7 本章小结

第三章 基于CUDA的高性能平台研究

近年来，计算机行业普遍转向并行计算。2010年前后，几乎所有的消费电脑都配备多核中央处理器。越来越多的软件开发人员需要应对各种并行计算平台和技术，以便为越来越复杂的用户群体提供新颖丰富的体验。从并行计算的角度来看，NVIDIA在2001年发布的GeForce 3系列可以说是GPU技术中最重要的突破。 GeForce 3系列是计算行业首款实现微软当时新DirectX 8.0标准的芯片，该标准要求兼容硬件包含可编程顶点和可编程像素着色阶段。这是第一次，开发人员可以控制在GPU上执行的精确计算。早期将GPU用作通用计算(GPGPU)设备是非常一项复杂的任务。因为像OpenGL和DirectX这样的标准图形API仍然是与GPU交互的唯一方式，所以任何在GPU上执行任意计算的尝试都将受到图形接口(API)内编程的限制。正因为如此，研究人员试图通过图形API探索通用计算，试图让他们的问题在GPU上呈现为传统渲染。实质上，21世纪初的GPU被设计为使用称为像素着色器的可编程算术单元为屏幕上的每个像素生成一种颜色。但是由于对输入颜色和纹理进行的算术完全由程序员控制，研究人员观察到这些输入“颜色”实际上可能是任何数据。从本质上讲，我们可以分配给GPU一些任务并让其看起来像是标准渲染一样，这样GPU实现执行非渲染任务或者说是通用计算任务。2006年11月，NVIDIA推出业界首款DirectX 10 GPU，即GeForce 8800 GTX。GeForce 8800 GTX也是第一款采用NVIDIA CUDA架构的GPU。CUDA体系结构包括严格设计用于GPU计算的几个新组件，旨在缓解以前的图形处理器无法合法用于通用计算的许多限制。与先前将计算资源划分为顶点和像素着色器的概念不同，CUDA架构包含统一的着色器流水线，允许芯片上的每个算术逻辑单元（ALU）被旨在执行通用计算的程序封送。由于NVIDIA希望将这一全新图形处理器系列用于通用计算，因此这些ALU的构建符合IEEE对单精度浮点运算的要求，并被设计为使用专为通用计算量身定制的指令集，而不是专门用于图形。此外，GPU上的执行单元允许对内存的任意读取和写入访问以及对称为共享内存的软件管理缓存的访问。所有这些新特性将被添加到CUDA架构中去，以便创建一个在计算方面非常出色的同时在传统图形任务中表现良好的GPU。

自2006年英伟达公司(NVIDIA)推出基于显卡(GPU)的编程模型CUDA以来已经有10余年时间，CUDA也发展到第九代，其中越来越多的技术，编程工具的涌现都为了减轻程序员的编程工作量。CUDA带有一个软件环境，允许开发人员使用C作为高级编程语言。CUDA支持多种编程语言，应用程序编程接口或基于指令的方法，例如FORTRAN，DirectCompute，OpenACC。

CUDA编程环境我们推荐使用NSIGHT作为软件开发的继承环境，它提供友好的代码高亮功能，同时也嵌入了许多CUDA內建所支持的函数，比如atomicAdd，，开发人员可以Linux和Mac OS X上使用，它旨在整个软件开发生命周期中帮助开发人员提高工作效率，加快开发产品速度。

3.1 CUDA编程模型与环境

3.1.1 CUDA接口

为了发挥图形设备(GPU)的最大潜力，CUDA提供了多种编程语言接口，其中包括了C/C++、Python、Java、Fortran等主流编程语言用于科学计算。本篇论文重点讨论研究的对象是基于C/C++语言的编程环境。

CUDA C本身就是在传统C语言的基础上做了扩展，其中被修饰为\_\_global\_\_的函数会被CUDA Runtime推送到GPU上去执行。一般而言，如果要开发一个基于CUDA的高性能应用程序，我们会把复杂的逻辑部分分配给CPU去管理，当遇到需要加速的数学模型时，将此模型设计为核心函数并在主机端(Host)调用，CUDA中我们需要使用扩展限定符<<<…>>>作为调用核心函数的接口。执行内核的每个线程都被赋予一个唯一的线程ID，可通过内置的threadIdx变量在内核中访问。

一个典型的对CUDA的异构系统的调用如下图；

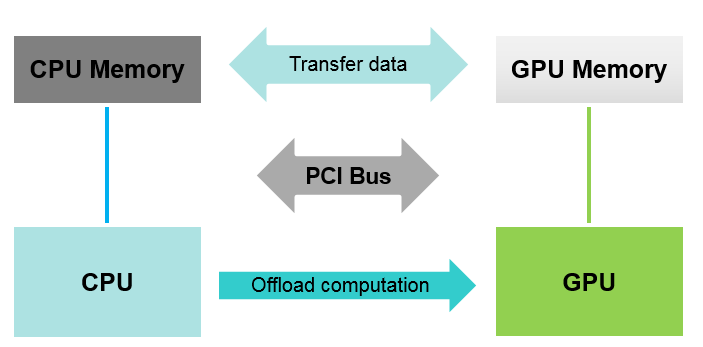


图1.1

从上图可知，主机端内存和设备端内存我们用PCI协议将它们联系在一起进行相关数据传送。当主机端有遇到需要进行高性能计算的任务时，通过接口将任务分配给设备(GPU)。

这里我们来看一个同型矩阵做加法是如何在GPU上实现的。

若假设有矩阵A，B求矩阵C=A+B。根据矩阵加法公式，矩阵C中的每个元素的值为矩阵A和矩阵B对应位置元素的和。

C=A+B==;

所以每个元素的算式就是C[i,j] = A[i,j] + B[i,j]

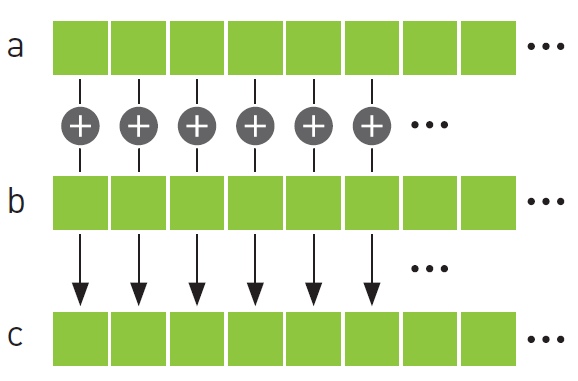


图1.2

我们首先定义一个核心函数，这里我们起名为VecAdd，它被限定符\_\_global\_\_修饰并且此函数没有返回值。它有三个形式参数，都是float类型的指针，表示它可以读取内存中的一片地址。在主机端的入口函数main体内，当我们需要调用VecAdd时候，就可以用VecAdd<<<..>>>来调用，让VecAdd在设备端执行。

这边还需要强调当我们在主机端调用Kernel函数时候，需要在尖括号内分别指定两个参数，第一个参数表示需要在设备端启动的块数(block)，第二个参数表示需要在设备端启动的线程数(thread)。

// Kernel definition

\_\_global\_\_ void VecAdd(float\* A, float\* B, float\* C)

{

int i = threadIdx.x;

C[i] = A[i] + B[i];

}

int main()

{

...

// Kernel invocation with N threads

VecAdd<<<1, N>>>(A, B, C);

...

}

3.1.2 CUDA中的线程

如前所述，CUDA中把活动线程数分别用两个参数来表示：块(block)、线程(thread)。这样区分的目的是因为CUDA规定每个块中的线程数最大不能超过1024。所以总的活动线程数为:

为了方便把活动线程和硬件设备上的每个核对应起来，CUDA提供了內建数据类型dim3。Dim3是一个由三个成员组成的结构体，我们在申明变量时候可以用实际需要使用的块数和线程数去初始化dim3的某个实例。在kernel内部，我们可以使用blockIdx、threadIdx和blockDim三个內建变量方便的访问到每个活动线程。在实际使用过程中会分别出现一下两种情况，

1. 如果所有线程都包含在一个块内启动，那么每个线程id就是tid=threadIdx;
2. 前面所述，每个块内线程不能超过1024个，当出现需要启用的活动线程超过1024的时候，就需要分配指定块和每个块内的线程，这种情况下，每个访问到的线程id可以用算式 tid=threadIdx.x+blockIdx.x\*blockDim.x来表示。

3.1.3 CUDA设备端的实现

由于CUDA需要在异构系统上运行，虽然CUDA也能支持非NVIDIA公司的设备，但是为了获得最大性能发挥，NV仍然推荐使用英伟达公司的GPU产品。在异构系统中执行程序之前，必须将主机端和设备端代码剥离区分。

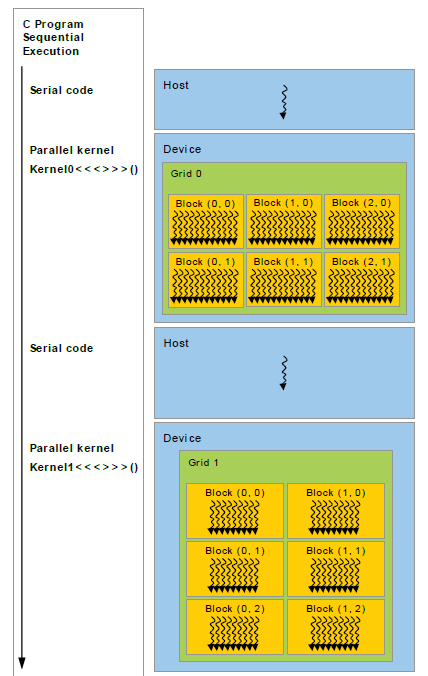


图1.3

如上图所示，主线程从主机端开始执行，经过一些逻辑判断整理后，把计算量巨大的工作分配给设备执行，上图中的Kernel<<<..>>>就承担起这样的任务。

类似于核心函数的调用和普通C语言很相似，可以看作是对传统C语言的扩展。

CUDA为熟悉C编程语言的用户提供了一个简单的途径，方便开发人员编写能在设备上运行的程序。内核函数必须由nvcc编译为二进制代码才能在设备上执行。用nvcc编译的源文件可以由主机代码(主机上执行的代码)和设备代码(在设备上执行的代码)混合组成。

nvcc的基本工作流程：

1. 将设备代码与主机代码分离。
2. 将设备代码编译成汇编形式(PTX)代码和/或二进制形式(cubin object)。
3. CUDA在运行时刻查找<<<..>>>的语法并使用函数culaunchkernel将其替换，然后从PTX代码或者cubin对象中启动kernel。

当设备驱动程序即时编译某些应用程序的某些PTX代码时，它会自动缓存生成的二进制代码的副本，以避免在后续调用应用程序时重复编译。缓存在升级设备驱动程序时会自动失效，因为应用程序需要适配新的驱动程序的性能，所以在新的驱动环境下再重新编译一次PTX代码。

3.2 Cooperative Groups(合作组)

CUDA支持合作组是为了扩展编程模型的灵活性并有效同步线程组之间的通信。合作组允许开发人员表达线程通信的粒度，帮助他们实现更丰富，更高效的分解并行粒度。以往CUDA模型只提供了两个相当粗糙的方法来同步线程，分别使用\_\_syncthreads()和cudaDeviceSynchronize()函数，这两种方法最多能同步一个warp单位的线程数量。但是，程序员希望以其他粒度定义和同步线程组，以“集体”组范围功能接口的形式实现更高的性能，设计灵活性和软件重用。为了提供更通用的并行交互模式，越来越多的程序员从性能方面考虑，会编写属于自己专用的、临时的、不安全的基元来同步单个warp内的线程，或者是在跨越单个GPU上运行的线程块集。虽然实现的性能改进是有价值的，但GPU端代码会变得越来越冗余，以至于在后续开发过程中生成，编写，调优和维护项目的代价非常昂贵。合作组通过提供安全且面向未来的机制来实现高性能代码来解决这个问题。

合作组由以下的元素组成：

1. 设计合理数据结构来抽象的表示合作线程组
2. 定义一系列方法，获得由CUDA接口启动的内部组
3. 将现有组划分为新组的操作
4. 子组之间的同步屏障
5. 检查子组和自定义组内的各个属性

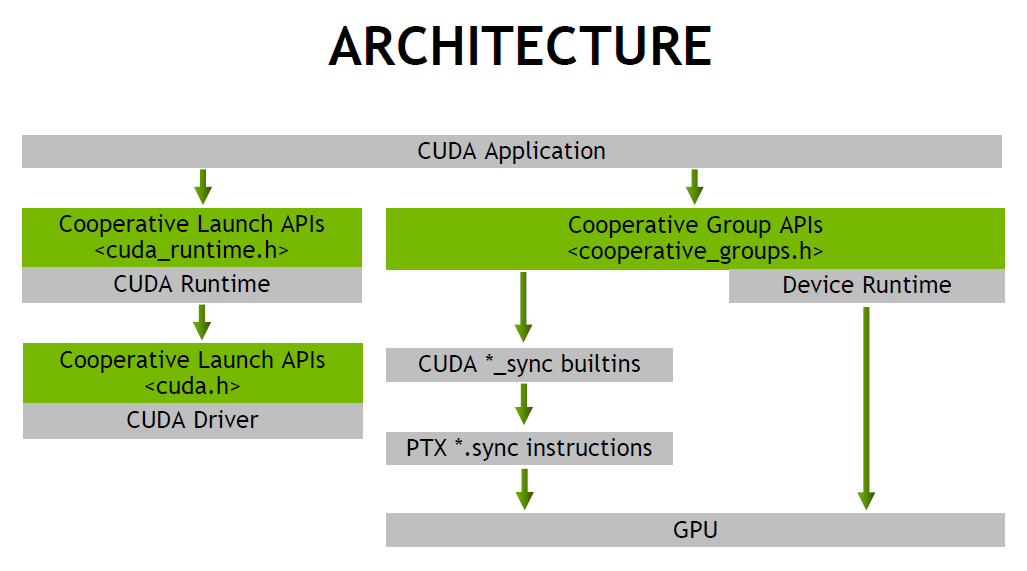
当主机端启动核模块并在设备上执行后，我们仅能通过函数在设备端同步某个流多处理器(streaming multiprocessors)的资源，在多数程序员看来这种同步方式过于粗放。按实际工程需求，人们通常需要在某个SM内部将线程再分为子组处理不同任务。由于某个块中能启用的线程数量有限制，开发人员需要合并不同的GPU上启动线程的资源，将他们统一计算，一次获得结果，避免数据在内存多次拷贝，降低运算效率。在合作组内部，CUDA分别使用tiled\_partition，this\_grid和this\_multi\_grid在逻辑层面分别来表示某个warp，某个GPU和多个GPU上的资源。从内部组的层面来看，分别可以看作以下组织：

1. Multi-grid group
2. Grid group
3. Thread group
4. Thread block tile
5. Thread block
6. Coalesced group

同时提供一些方法读取各线程组的属性，或在各组之间提供通信及同步功能：

|  |  |
| --- | --- |
| Sync() | 同步组内线程 |
| Size() | 获得组内线程数 |
| Thread\_rand() | 用0-size为线程索引 |
| Isvalid() | 该组是否违反了任意接口的限制条件 |

？？？



3.3 动态并行(Dynamic Parallelism)

早期的CUDA程序是一种符合扁平流程的编程模型，程序必须执行一套内核启动接口。为了获得最佳性能，每个线程都应该清晰地表达并行的意图才能发挥GPU的最大潜力。对于那些能在循环内部表示并行逻辑的单元并不会有太多限制条件。但是想表达并行内部嵌套的执行单元不是容易实现的。嵌套并行性在很多应用程序中自然产生，例如那些使用自适应网格的应用程序，这些应用程序通常用于实际应用程序，以降低计算复杂性，同时捕获相关细节。并行应用程序在批量处理任务时必须使用细网格，这会增加多余的计算量。如果采用粗粒度的网格来完成任务则会丢失相关的细节内容。动态并行性使得从设备上运行的线程启动内核成为可能。线程可以启动更多的线程，应用程序可以启动粗粒度内核，然后在需要的地方再次细化内核来完成工作。这可以有效避免重复计算内容，同时又能捕获所有有关细节。

动态并行是对CUDA编程模型的扩展，它允许CUDA内核直接在GPU上创建和同步新的任务，程序具备在kernel启动时动态地创建工作的能力。

开发人员直接在GPU上创建并行工作组，这将大幅减少主机和设备之间的数据传输和逻辑控制流程，因为通过动态并行技术允许直接在设备上执行线程并在运行时刻启动配置决策。另外，并行数据任务在运行时刻由内核内部生成，这可以动态地利用GPU的硬件调度程序和负载平衡器，并根据数据驱动的决策或工作负载进行实时调整。在引入动态并行技术之前，开发工程师需要在内核中去除递归，不规则循环或任何不符合单层并行的算法和程序模式。

3.3.1 执行环境

CUDA执行模型是基于线程，线程块(thread blocks)和网格(grids)的原语，内核函数定义由线程块和网格中的各个线程执行的程序。当调用内核函数时，网格的属性由执行配置来描述，该配置在CUDA中有特殊的语法。对CUDA中的动态并行性的支持扩展了在新网格上配置，启动和同步设备上运行的线程的能力。

3.3.2 父子网格

配置和启动新网格的设备线程属于父网格，并且由调用创建的网格是子网格。直到子网格所属的线程启动的网格执行完成后父网格才结束退出，这表面动态并行完成了对子网格的正确嵌套调用。即使调用线程没有显式同步启动的子网格，运行时也会保证父级和子级之间的隐式同步。

3.3.3 同步

CUDA运行时刻操作的线程在整个线程块中都是可见的。这表示父网格中调用的线程可以在该线程发起的网格中，线程块中的其他线程上执行同步操作，或同步同一线程块内的CUDA工作流。直到块中所有线程的启动都完成后，在线程块上执行的任务才算完成。如果一个块中的所有线程在所有子线程启动完成之前退出，则会自动触发同步操作。

3.3.4 排序和并发

设备运行时刻内核的启动必须遵循CUDA 工作流(Stream)的语义顺序。在一个线程块内，同一个工作流中都会按顺序或启动核执行内核函数。由于同一个线程块中的多个线程会启动到同一个流中，工作流中的排序取将决于块内的线程调度，可以使用同步原语(比如\_\_syncthread())来控制该线程。动态并行可以更容易的表达程序中的并发指令。但是这不能确保设备上任意不同线程块之间都能执行并发操作。当一个父线程块启动子网格时，子节点并不开始执行任何工作，直到父节点线程块达到显式同步点(例如cudaDeviceSynchronize())，才能确保子节点开始执行任务。虽然通常很容易实现并发特性，但这可能会随着设备配置、应用程序工作负载和运行时间调度的变化而变化。因此任何依赖不同线程块之间的并发是不安全的。

3.4 NVML(NV设备管理)

NVML是一套基于C语言的编程接口，它可以用来监视和管理英伟达GPU设备的各种状态。通过nvidia-smi，可以由命令直接访问和查询设备状况。NVML运行版本会随着英伟达显卡驱动程序一起发布，这套软件开发套件(SDK)提供了经典的组件：头文件，动态库和编程案列。每个新版本的NVML都向后兼容，旨在成为构建第三方应用程序的平台。

通过NVML，我们可以获取设备相关的信息：

|  |  |
| --- | --- |
| 项目名称 | 信息意义 |
| ECC错误计数 | 获得可纠正的单个位和可检测到的双位错误。 记录当前引导周期和GPU的生命周期错误数量 |
| GPU利用率 | 获取GPU和存储器接口的计算资源的当前利用率信息 |
| 活动计算进程 | 运行在GPU上的活动进程列表，以及相应的进程名称/ID和进程占用的GPU内存信息 |
| 时钟和Pstate | 几个重要时钟域的最大和当前时钟速率，以及当前的GPU性能状态的信息 |
| 温度和风扇速度 | 当前的核心GPU温度，以及非无源产品的风扇速度的信息 |
| 电源管理 | 如果设备支持这项功能，就显示当前的功耗和功率限制的信息 |
| 识别 | 各种动态和静态信息，包括主板序列号，PCI设备ID，VBIOS / Inforom版本号和产品名称 |

同时，通过接口调用，还能修改以下设备参数：

|  |  |
| --- | --- |
| 接口参数 | 修改意义 |
| ECC模式 | 启用或禁用ECC |
| ECC重置 | 清除单位和双位ECC错误计数 |
| 计算模式 | 控制计算进程是否可以在GPU上运行，以及它们是独立运行还是与其他计算进程同时运行 |
| 持久性模式 | 控制当没有活动客户端连接到GPU时NVIDIA驱动程序是否保持加载状态 |

按照接口功能可以划分为以下5类方法：

1. 支持类方法
2. 查询类方法
3. 控制类方法
4. 事件处理方法
5. 错误报告方法

3.5 GPU P2P ？？？

GPUDirect包括一系列不断发展的技术，以提高性能并扩展其可用性。英伟达公司于2010年6月首次推出，GPUDirect Shared Access支持通过共享固定主机内存加速与第三方PCI Express设备驱动程序的通信  
待改进扩展

3.6 NVLINK ？？？

3.7 Unified memory(统一内存管理)

统一内存(Unified Memory)也是CUDA编程模型中全新的组件，它定义了一个托管内存空间，在这片空间中所有处理器都可以访问公共地址空间上单个连贯内存映像。这里提到的处理器是指具有任何专用MMU的独立执行单元，包括任何类型和架构的CPU和GPU。在CUDA程序中使用此项技术，使得底层系统对数据访问和获取数据位置都不再需要显示的内存拷贝调用，这将在两个方面利于GPU端的编程：

1. CUDA程序员可以为系统中所有GPU和CPU的内存空间提供更紧密和更直接的语言描述从而得失GPU编程得到简化。
2. 同时通过将数据透明地迁移到使用它处理器，可以最大限度地提高数据访问速度。

简而言之，统一内存不需要通过常规的函数调用，比如cudaMemcpy()等内存操作函数进行显式地数据迁移，这不会因为将所有数据放入零拷贝内存而导致性能损失。当然数据移动仍然会发生，所以程序的运行时间通常不会减少，但使用统一内存编写程序可以使代码看上去更简单易懂，且更易于维护。

Unified Memory提供了一个“单指针到数据”的模型，它在概念上类似于CUDA的零拷贝内存。两者之间的一个关键区别在于，使用零拷贝分配时，内存的物理位置被固定在CPU系统内存中，程序对其是进行快速访问还是慢速访问是由内存中的位置决定的。而统一内存会将内存和执行空间分开，以便能快速访问所有数据。

统一内存描述了一个系统，它提供为各种各样的程序提供内存管理服务，从定位运行时API到使用虚拟ISA(PTX)程序。 该系统的一部分定义了选择统一内存服务的托管内存空间。

托管内存可与设备内存互相操作、交换数据，例如使用cudaMalloc()申请到的内存资源。所有在设备内存上合法的CUDA操作在托管内存上同样有效。它们之间的主要区别在于程序的主机部分也能够引用和访问托管内存空间。

存储空间的统一意味着主机和设备之间不再需要显式存储器传输。 在托管内存空间中创建的任何分配都会自动迁移到需要的位置。

统一内存通过将数据迁移到正在访问的设备来尝试优化内存性能，如果CPU正在访问它则将数据移动到主机内存，如果GPU将要访问它就将数据移动到设备内存。数据迁移是统一内存的基础，程序编写不会受到底层数据迁移的影响。系统在保证一致性的原则上会尝试将数据放置在最有效率的访问空间上。

数据的物理位置对程序是不可见的，并且可能被随时更改，但对于访问虚拟地址上的数据而言，不如论是在本地处理器的哪个位置空间，都将保持对这片地址的合法占用。保证对一片内存地址上的数据持续有效访问是比提升系统性能更为重要的任务。另外需要提到的是在主机操作系统的限制下，为了保持各处理器之间的全局一致性，系统有时候不被允许访问或移动数据。

3.8 CUDA tools

CUDA工具包是一套应用程序，其控制部分在通用计算设备上作为进程运行，并使用一个或多个NVIDIA的协处理器来加速单个程序，多个数据(SPMD)并行作业。这些作业是独立的，因为它们可以由一批GPU线程执行完成，完全不需要主机进程进行干预，从而获得并行图形硬件的加速性能。

一般我们把GPU代码制作成C ++语言的函数集合来实现，写一些注释将它们与主机代码区分开，还有些注释用于区分在GPU上的不同类型的数据存储器。 这些函数一般需要参数才行运行，并且可以调用方法与常规C函数的调用非常相似，会做一些略微扩展为的是能够指定调用GPU线程上矩阵的函数。在其生命周期中，主机进程可能会派发许多并行的GPU任务。

3.8.1 NVCC

编译器需要做多个步骤，如每个CUDA源文件拆分，编译，预处理和合并等。 Nvcc作为CUDA编译器驱动程序，其目的是用于隐藏开发人员编写CUDA加速程序的复杂细节。它接收一系列常规编译器选项，例如定义宏和包含/库路径，以及用于指导编译过程。所有非CUDA编译步骤都转发给nvcc支持的C ++主机端编译器，并且nvcc将其选项转换为合适的主机端编译器选项。

一次编译执行是通过命令行选项向nvcc选择的逻辑转换步骤。单个编译阶段仍然可以通过nvcc分解为更小的步骤，但是这些更小的步骤仅能实现某一编译环节。如果在编译阶段碰到问题，nvcc还提供详细日志描述整个编译步骤和详细的错误信息，当然这只能用于调试目的，一般不把这个功能加到正式发布的安装脚本中。

3.8.2 Cuda-memcheck

CUDA工具箱还提供了CUDA-MEMCHECK，用于检查GPU正确性的软件。 该软件包含多个可执行不同类型检查的工具。 内存检查 (memcheck)工具能够精确检测和分配CUDA应用程序中的越界和未对齐的内存访问错误。 该工具还可以发现GPU遇到的硬件异常。racecheck工具可以检查出可能导致数据竞争的共享内存数据访问的潜在危险。 initcheck工具可以报告GPU执行未初始化访问全局内存的情况。同步检查工具可以报出应用程序无效使用同步原语的情况。调试的时候可以将用户的应用程序作为cuda-memcheck的参数使用，让检查程序自动查找潜在错误或问题。

3.8.3 NVVP

图形监测器(NVIDIA Visual Profiler)能提供可视化地展示并行应用程序运行的效果。Visual Profiler能分别显示CPU和GPU上应用程序活动的时间线，为开发人员分析和改进程序的性能提供便利。此外，Visual Profiler还能分析GPU应用程序，以检测潜在的性能瓶颈，并指导研究人员采取何种措施消除或减少这些瓶颈。

3.8.4 NSIGHT

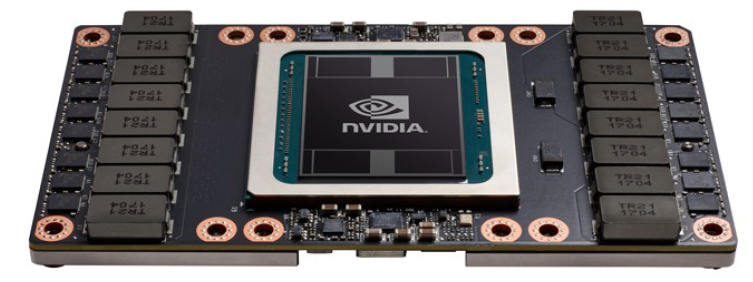
Nsight是一个统一CPU和GPU代码的集成开发环境(IDE)，能在x86架构的Linux和Mac OS X上和POWER以及ARM平台上开发CUDA应用程序。它能在软件开发过程的所有阶段都为开发人员提供帮助。Nsight Eclipse Plugins需要安装在Eclipse 4.4或更高版本上才能运行，它的主要功能有以下几点：

1. 编辑，构建，调试和配置基于CUDA-C的应用程序
2. 支持CUDA的源代码编辑器，如语法高亮显示，代码补全和内联帮助
3. 用于调试异构应用程序的图形用户界面
4. Profiler集成，使用Nsight中内置的CUDA应用程序将Visual Profiler作为外部应用程序启动

3.9系统硬件设计|基于Volta架构的硬件平台 (可能考虑放在第四章)

为了搭建实验环境，我们在此选用了nvidia公司最新的基于Volta架构的设备(GV100)来运行我们的程序。GV100不仅仅是它的上一代产品Pascal架构的改进，事实上它显着地提高了性能，丰富了可扩展性，并增加了许多可改善和可编程的新功能。

GV100是一块包含211亿个晶片，芯片尺寸为815 mm2的GPU。它采用为NVIDIA定制的新型12nm FFN(NVIDI专属的FinFET)高性能制造工艺制造。与前代产品Pascal GPU相比，GV100具有更高的计算性能，并增加了许多新功能。它进一步简化了GPU编程和应用程序的移植，GV100还可以提高GPU资源利用率。GV100还是一款非常节能的处理器，可提供出色的每瓦性能。

Volta对GPU核心的SM处理单元进行了重新设计。最新的Volta SM比上一代Pascal产品节能50％，在相同功率范围内实现了FP32和FP64性能的重大提升。可实现并行线程之间的更细粒度的同步和协作的独立线程调度功能也是首次被运用。而且，新组合的L1数据缓存和共享内存单元显着提高了性能，同时简化了编程。

Volta支持第二代NVIDIA NVLink高速互连技术，为多GPU和多GPU / CPU系统配置提供了更高的带宽，更多的链接和更灵活的可扩展性。与GP100上的四条NVLink链接和160 GB/s总带宽相比，Volta GV100支持六条NVLink链接和300 GB/秒的总带宽。

Volta高度调校的16 GB HBM2内存子系统可提供900 GB/秒的峰值内存带宽。 三星的新一代HBM2存储器和Volta的新一代存储器控制器相结合，可提供1.5倍的内存带宽，而Pascal GP100只能提供95％的内存带宽利用率。

多进程服务(MPS)也是Volta GV100架构的一项新功能，它为CUDA MPS服务器的关键组件提供了硬件加速，同时又增强了共享多GPU的应用程序的性能，隔离性和服务质量(QoS)。它还将MPS客户端的最大数量从Pascal的16个增加到Volta的48个。

在统一内存技术方面它包括了新的访问计数器，可以更准确地将内存页面迁移到访问它们最为频繁的处理器上，从而提高了处理器之间共享内存范围的效率。 另外在IBM Power平台上，新的地址转换服务(ATS)允许GPU直接访问CPU的页表。

在最高性能模式下，V100加速器将运行至300瓦的热设计功率水平，以加速那些需要快速计算和高数据吞吐量的应用程序。最高效率模式允许数据中心管理人员调整V100加速器的电源，以获得最佳的每瓦性能。可以将集群中的所有GPU的功率都设置在上限范围内，从而大幅降低功耗，同时仍可获得出色的整体性能。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| GPU | Kepler GK180 | Maxwell GM200 | Pascal GP100 | Volta GV100 |
| Compute Capability | 3.5 | 5.2 | 6 | 7 |
| Threads / Warp | 32 | 32 | 32 | 32 |
| Max Warps / SM | 64 | 64 | 64 | 64 |
| Max Threads / SM | 2048 | 2048 | 2048 | 2048 |
| Max Thread Blocks / SM | 16 | 32 | 32 | 32 |
| Max 32-bit Registers / SM | 65536 | 65536 | 65536 | 65536 |
| Max Registers / Block | 65536 | 32768 | 65536 | 65536 |
| Max Registers / Thread | 255 | 255 | 255 | 2551 |
| Max Thread Block Size | 1024 | 1024 | 1024 | 1024 |
| FP32 Cores / SM | 192 | 128 | 64 | 64 |
| Ratio of SM Registers to FP32 Cores | 341 | 512 | 1024 | 1024 |
| Shared Memory Size / SM | 16 /32 / 48 KB | 96 KB | 64 KB | 96 KB |

Volta架构GPU和历代GPU的性能比较

第四章 容器内部的异构系统及Web服务

现如今，越来越多的公司开始关注“云”的世界并怀抱Docker技术。描述Docker的最好方法我想到的是使用Docker网站中的短语：Docker“是一个开源项目，可将任何应用程序作为轻量级容器进行打包，发布和运行”。这个想法是提供一个全面的抽象层，允许开发人员 以“容器化”或“封装”任何应用程序并使其在任何基础架构上运行。当然这里所谓的容器是使用一种标准打包格式将应用程序和它所需要的执行环境封装起来，而不是指任何一种现有的底层技术。为了快速说明Docker的主要作用，我可以打个比方：Docker是一种运用在发布时候的容器，它提供了一种标准的，一致的环境，组件和接口等，在容器内部，开发人员不需要在考虑任何关于环境的因素，只需要将精力放在开发项目上。容器十分轻量级，因为它们不需要管理程序的额外负载，而是直接在主机的内核中运行。 这意味着，与使用虚拟机相比，用户可以在给定的硬件组合上运行更多的容器。甚至可以在虚拟机的主机中运行Docker容器！

Docker 最初是 dotCloud 公司创始人 Solomon Hykes 在法国期间发起的一个公司内部项目，它是基于 dotCloud 公司多年云服务技术的一次革新，并于 2013 年 3 月以 Apache 2.0 授权协议开源，主要项目代码在 GitHub 上进行维护。Docker 项目后来还加入了 Linux 基金会，并成立推动 开放容器联盟（OCI）。

Docker 自开源后受到广泛的关注和讨论，至今其 GitHub 项目已经超过 4 万 6 千个星标和一万多个 fork。甚至由于 Docker 项目的火爆，在 2013 年底，dotCloud 公司决定改名为 Docker。Docker 最初是在 Ubuntu 12.04 上开发实现的；Red Hat 则从 RHEL 6.5 开始对 Docker 进行支持；Google 也在其 PaaS 产品中广泛应用 Docker。

Docker 使用 Google 公司推出的 Go 语言 进行开发实现，基于 Linux 内核的 cgroup，namespace，以及 AUFS 类的 Union FS 等技术，对进程进行封装隔离，属于 操作系统层面的虚拟化技术。由于隔离的进程独立于宿主和其它的隔离的进程，因此也称其为容器。最初实现是基于 LXC，从 0.7 版本以后开始去除 LXC，转而使用自行开发的 libcontainer，从 1.11 开始，则进一步演进为使用 runC 和 containerd。

Docker 在容器的基础上，进行了进一步的封装，从文件系统、网络互联到进程隔离等等，极大的简化了容器的创建和维护。使得 Docker 技术比虚拟机技术更为轻便、快捷。

下面的图片比较了 Docker 和传统虚拟化方式的不同之处。传统虚拟机技术是虚拟出一套硬件后，在其上运行一个完整操作系统，在该系统上再运行所需应用进程；而容器内的应用进程直接运行于宿主的内核，容器内没有自己的内核，而且也没有进行硬件虚拟。因此容器要比传统虚拟机更为轻便。

4.1 Docker容器的组织结构

Docker使用的是客户端-服务器体系结构。 Docker客户端与Docker守护进程通信，Docker守护进程负责构建，运行和分发Docker容器。Docker客户端和守护进程可以在同一个系统上运行，也可以将Docker客户端连接到远程Docker守护进程。 Docker客户端和守护进程使用REST API通过UNIX套接字或网络接口进行通信。



Docker engine

Docker container

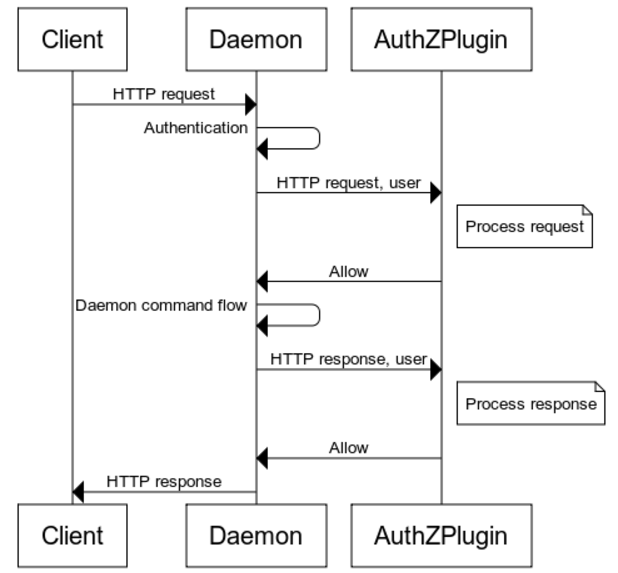
Docker service

Docker entry  
Docker image

Dockerfile  
带扩展改进

4.2 Docker的外部扩展开发

Docker插件是增强Docker引擎功能的进程外扩展。Docker本身提供的服务内容有时候并不能满足实际需求，为了拓展功能Docker还提供了进程扩展服务，可以通过插件查找机制完善容器技术。为了能读取英伟达驱动程序，我们需要开发一个访问授权的插件，访问授权流程图如下：

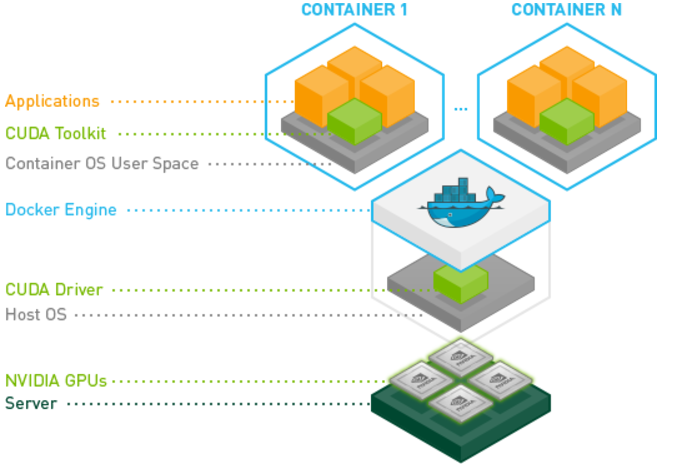


对于插件的使用，公司各组之间在开发过程中达成以下原则，后续工作都围绕这些原则进行：

1. Docker的插件基础架构可以通过使用通用API加载，删除和与第三方组件进行通信来扩展Docker。访问授权子系统是使用这种机制构建的。
2. 使用这个子系统，不需要重建Docker守护程序来添加授权插件。任何人可以将插件添加到已安装的Docker守护程序。另外需要重新启动Docker守护程序才能添加新的插件。
3. 授权插件基于当前认证上下文和命令上下文来批准或拒绝对Docker守护进程的请求。认证上下文包含所有用户详细信息和认证方法。命令上下文包含所有相关的请求数据。
4. 授权插件必须遵循Docker插件API中描述的规则。每个插件必须驻留在插件发现部分下描述的目录中。

这就表示，插件不会运行在Docker daemon中。我们可以随时随地（如果需要可以在另一台主机上）启动的插件。只需要通过Plugin Discovery通知Docker daemon这儿有一个新的插件可用即可。

下图直观的反应了整个扩展应用的架构：



4.2.1 开发环境配置

开发项目之前，各组同步了集成开发环境和代码管理工具。为了节约项目成本，集成开发工具使用微软公司提供的开源软件VSCode，代码管理工具使用公司现有的，已经购买了服务的Perforce。

市面上已经有许多成熟优秀的代码版本管理软件，比如Git，SVN，它们都广泛被各大公司采用，并帮助项目团队回顾代码。这里我们采用的英伟达公司已经使用多年的Perforce，它是基于服务器/客户端架构的，公司IT部门已经搭建了服务器端并为每位同事都创建了相应账号，我们只需要在本机上安装Perforce的客户端并做相应配置。

Perforce是一个企业级版本控制和内容协作平台，可以为现代开发团队负责构建的日益复杂的系统的各个方面提供灵活性和控制。Perforce可扩展以适应各种类型和规模的团队和资产，并支持高度自动化的持续交付实践。

环境搭建步骤如下：

* 1. 访问Perforce官网“https://www.perforce.com/downloads/helix-command-line-client-p4”并下载Unix x84\_64版本的P4 command。
  2. 使用编辑工具vim打开文件~/.barshrc并配置环境变量：export P4UER=username，export P4PASSWORD=password。
  3. 建立本地仓库，这里我导入一个之前创建好的模板库，用命令实现p4 client -t ocl02-All-Series
  4. 并将下列信息配置到p4 client中去：

Client: thinkStation

Owner: ffan

Host: thinkStation

Root: /home/test/p4

View:

//sw\_spec/... //dhcp-10-19-192-202/sw\_spec/...

//sw/... //dhcp-10-19-192-202/sw/...

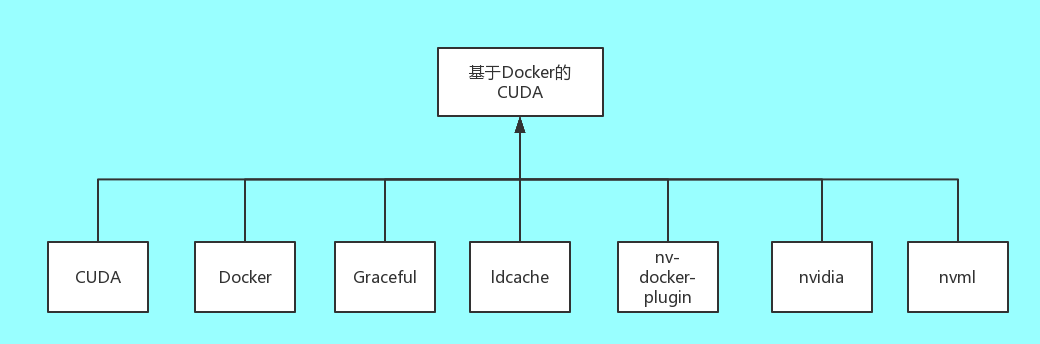
Visual Studio Code是Microsoft为Windows，Linux和MacOS开发的源代码编辑器。它包括对调试，嵌入式Git控制，语法高亮，智能代码完成，片段和代码重构的支持。它也是可定制的，因此用户可以更改编辑器的主题，键盘快捷键和首选项。它是免费且开源的，虽然官方下载是在专有许可下进行的。

VScode中搭建Go语言的环境及其安装步骤：

* 1. 从<https://code.visualstudio.com/>下载适合vscode版本，这里我选择的是支持Linux x86\_64的版本。
  2. 打开VScode，点击扩展按钮(Extension)，在搜索框内填入Go。
  3. 安装插件Go for Visual Studio Code点击重新加载。重启VSCode后在expoler界面新建docker-plugin-nvidia项目。

扩展应用后，Docker引擎上层依然是各类自定义容器，只是这用容器内部已经安装了CUDA的工具包，并且容器上层的应用程序一般都用于加速的应用程序。和一般系统不同，容器下层是支持CUDA硬件的驱动程序，通过插件发现机制，让去驱动程序和Docker通信，并能在容器内部访问硬件资源，驱动程序构建在主机操作系统上，主机的硬件设备仍然需要安装英伟达公司的图形加速处理器。

从模块组织形式上我们将分成以下7个模块分别实现相应功能。



从具体实现来说，基于Go开发环境为插件实现下列模块以及他们负责的职能。

CUDA模块：

对于每一代硬件架构，在一个map里面将其与键值一一对应。为了配合Volta，需要最高支持到计算能力7.0。还需要开发一个Go语言版本的接口能访问底层设备，getDeviceProperties代替cudaGetDeviceProperties的功能。还支持查询硬件设备的功能接口GetDriverVersion，接受一个字符串类型的参数，通过另一个参数error表明当前系统状态。我们还需要一个NewDevice接口让模块具备创建新设备的能力，在设备数量变化时候能访问每一块主机内的GPU。为了让用户在容器内容像直接使用CUDA Runtime时候一样方便，我们提供一些方法，逻辑上做出映射关系，让其具备与实体设备同样的功能。下图是功能对应表：

|  |  |  |
| --- | --- | --- |
| 容器内部方法 | CUDA Runtime方法 | 方法功能 |
| errorString | cudaGetErrorString | 通过调用返回人们容易读懂的错误信息 |
| driverGetVersion | cudaDriverGetVersion | 通过调用返回当前显卡驱动版本 |
| deviceGetByPCIBusId | cudaDeviceGetByPCIBusId | 通过主机PCI地址获得一个当然设备的句柄 |
| deviceCanAccessPeer | cudaDeviceCanAccessPeer | 判断设备是否具备点对点功能 |
| deviceReset | cudaDeviceReset | 动态地重设活动设备 |
| getDeviceProperties | cudaGetDeviceProperties | 获取设备的属性内容；比如计算能力，设备名称，内存容量等 |

Docker模块设计：

如前一小结所述，docker本身还具备数十个命令，不同字命令担负不同功能，在定制化的容器接口处，我们并不想让用户在使用本接口时候和原生docker感觉到有太大区别。所有通过docker模块，把主要一些字命令做了重新封装，类似于软件库的钩子，当用户通过docker-plugin-nvidia使用容器时候，能执行我们定义的命令，这些命令会少许更改原生指令，目的是为了和英伟达硬件直接的通信。

Docker接口把所有传递给主程序的参数保存在一个数组内，通过go的字符串处理函数去除噪声，然后在nvidia-docker中执行。扩展插件本身也是通过卷和外部联系，系统也要检查卷的状态。例如抽取镜像，镜像检测等原生方法在系统内部也需要支持。

Graceful模块设计：

插件发现机制是通过服务器/客户端形式通过socket接口在主机端实现，但是在每个模块文件中都插入机制代码会造成冗余度高。这模块中我们实现了recovery，NewHTTPServer，Handle，Stop，Error

定义HTTPServer结构，用成员变量sync.Mutex控制线程之间同步，变量network记录当前网络号，router和server都是指向网络服务的指针，err返回错误信息。

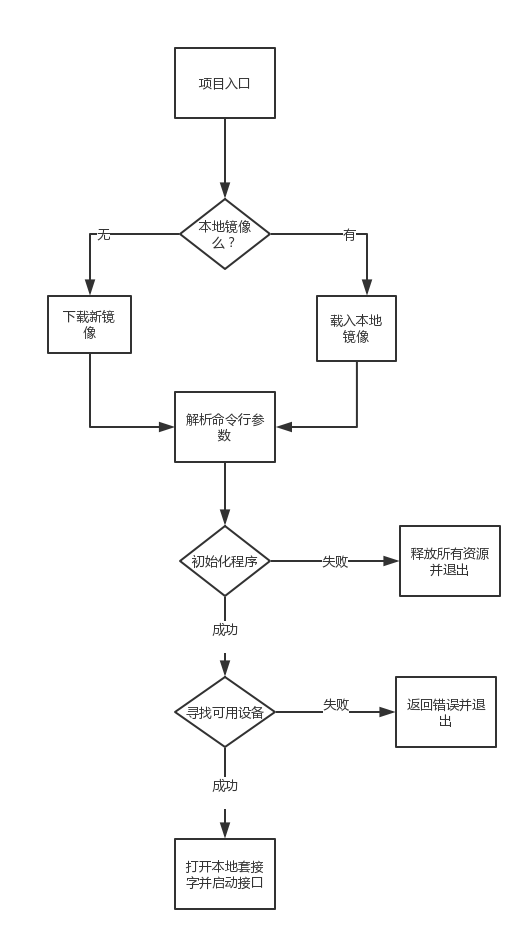
“func recovery(handler http.Handler) http.Handler”是一个恢复动作，持有http类型的句柄，当调用返回时把http类型中的Handler返回。

func NewHTTPServer(net, addr string, mw ...middleware.Constructor) \*HTTPServer: 方法让程序新设一个基于http协议的服务器，通过string参数网络net和地址addr以及中间件mw生成新的服务实例。

Ldcache模块：

Nv-docker-plugin模块设计：

整个程序的入口模块，负责控制启动流程，功能开启，前端与用户交互，后台和通过卷(volume)访问主机设备，实现了容器内部驱动程序利用硬件，在容器内实现高性能平台。工作流程图如下所示：



Nvidia模块设计：

在本模块内，按软件工程需求将重复代码抽取出来，并按照不同功能点分成4个子模块：环境管理，本地化，远程和工具，方便日后继续开发维护。实现了下列主要方法供程序使用：

func parseAddr(addr string) (host, sport, hport string)实现了解析转换ip地址功能，接收一个字符串类型的ip地址后获得主机名，套接字端口和主机通信端口号。

func LoadEnvironment() (err error)：将子程序实现的功能getHost，getGPU，getDocker做逻辑封装，并统一执行管理。

func getHost() (\*url.URL, error)：通过內建方法os.getenv获得nvhost和dhost，用正则模型”^[a-z0-9+.-]+://”匹配出合法的地址并连接字符串”tcp”。分析env获取url，判断url协议是否为tcp，http或ssh中的任意一种并建立相应连接。

func getGPU() []string：通过內建方法FieldsFunc回调函数指针c rune 获得GPU环境信息。

func getDocker() []string：主要通过內建方法strings.Field获得NVDocker的环境信息。

Nvml go模块设计：

在系统内会涉及到Go与C的互操作。这能提升局部代码性能，用C替换一些Go代码。C之于Go，好比汇编之于C。在系统中通过直接打开libnvml函数库并获得句柄的操作方式访问库中所有函数符号，通过操作这些合法符号使得系统具备管理，改变硬件属性的能力。下面是具体实现方式：

func init() error实现了初始化nvml句柄，并不初始化任何GPU设备。当系统中的GPU处于不稳定或处于不良状态时允许NVML与系统中的GPU通信。同时GPU会被nvmlDeviceGetHandleBy函数中发现并初始化。

func shutdown() error：实现了关闭并释放NVML，这些资源之前被init初始化时候占用的。

func systemGetDriverVersion() (string, error)可以获取系统中GPU的版本信息，该版本标识符是一个字母数字字符串。 它的长度不会超过80个字符（包括NULL终止符）。

func systemGetProcessName(pid uint)通过进程号参数unit尝试获得该进程名称，返回的进程名称被裁剪以提供长度。该名称是用ANSI编码的字符串。

func deviceGetCount()用来检索系统中计算设备的数量。

func deviceGetHandleByIndex(idx uint) 根据索引unit获取特定设备的句柄。有效索引是从nvmlDeviceGetCount()返回可访问设备数得出的。

func deviceGetTopologyCommonAncestor(h1, h2 handle)

func (h handle) deviceGetBAR1MemoryInfo()

func (h handle) deviceGetPowerManagementLimit()

func (h handle) deviceGetMaxClockInfo()

描述实现细节  
带扩展改进

4.3 基于Django的Web框架开发

主要描述web路由，响应方法实现等技术细节

4.4 后端数据库服务

主要描述数据表设计等技术细节

4.5 前端服务部署

主要描述基于nginx的http服务的实现细节

第五章 高性能异构系统的实现和测试

5.1 可访问GPU的Docker容器实现

5.2 Cholesky(PageRank)算法在系统中的实现

5.3 性能调优(NVVP, cuda-memcheck)

5.4 Linpack测试系统

5.5 基于Django开发的Web服务

第六章 结论与展望

6.1 总结

6.2 展望

致谢