中国科学技术大学

学士学位论文

基于 NFV 平台利用 GPU 构建高性能网络入侵检测系统的研究

姓　　名:	孟佳逸
院　　系:	计算机科学与技术系
学　　号:	PB12011054
导　　师:	华蓓 教授
完成时间:	二〇一六年五月

University of Science and Technology of China

A dissertation for bachelor's degree

Building High Performance Network Intrusion Detection System Using GPU on the NFV Platform

Author : Jiayi Meng

Department : Computer Science and Technology

Student ID : PB12011054

Supervisor : Prof. Bei Hua

Finished Time : May, 2016

致　谢

经历三个月的时间，我的毕业设计以这篇论文的完成而标志着结束。虽然毕业设计已经结束，但是基于 NFV 平台利用 GPU 构建高性能网络入侵检测系统的研究还远远没有结束。无论如何，在这里我要感谢一下对我的毕业设计工作基于帮助的老师和同学们。

首先要感谢在毕业设计过程中给予我悉心指导和细致帮助的华蓓老师。华蓓老师是我计算机网络课和网络算法学的任课老师。华老师细致的知识讲解和平易近人的教书方式，让我对系统网络产生极大的兴趣。在完成毕设的这一学期里，华老师每周都会专门开展组会，了解我们的毕设进展。令我印象深刻的是华老师在选题和阶段性审核中十分耐心细致。在最后修改论文的阶段，虽然华老师也十分繁忙，但是总能第一时间查阅我的邮件，并对我的毕业论文中每一句每一字都仔细审核，提出修改意见。更令我感动的是，在毕设进展的攻坚阶段，自己的出国申请出现问题。在自己特别举足无措之时，华老师能安慰我、帮助我、指导我、教育我。正是因为华老师对我的帮助，让我找到了自己前进的方向，不再迷茫彷徨，也收获了好的申请结果。谢谢华老师！谢谢华老师为我的付出！

然后要感谢的是华老师实验室的张凯师兄和胡嘉瑜师姐。第一次的见面，在了解我的研究兴趣之外，又和我轻松愉快地进行聊天，一下子拉近了我们的距离。在毕设开展阶段，张凯师兄和胡嘉瑜师姐对我的帮助也是非常之大，总能帮助我，解决研究中遇到的困难。尤其是张凯师兄，从开始帮助我理解毕设选题，再到对毕设实现的思路把关，再到对我们毕设的时间规划给予建议。张凯师兄可以称得上手把手地带我走上研究的道路。平常，张凯师兄还会分享给我他曾经留学的经验，把很多研究、生活的技能传授给我们。谢谢张凯师兄，胡嘉瑜师姐！

最后还要感谢在实验室里一起做毕设的好伙伴们，孙海沛、程元彬、杨丽杉。我们互相帮助、互相学习。正是有了这些朋友的陪伴，这三个月的毕设工作才充满了欢声笑语。谢谢你们！

三个月的时间转瞬即逝。尽管这三个月里充满了酸甜苦辣，但绝对是我本科阶段最难忘的一段回忆。喜欢实验室轻松但不随便的氛围，喜欢华老师，喜欢师兄师姐，喜欢一起研究的伙伴。

科大四年的生活即将结束。在此，我要感谢方瑾老师、钱海老师，感谢你们对我学习、工作、人生的帮助、指导和教育；感谢曾经和现在属于 1211 的所有同学，感谢你们的陪伴；感谢我的爸爸妈妈，永远爱你们。谢谢大家！

孟佳逸

mjycom@mail.ustc.edu.cn

2016 年 6 月 12 日

目 录

摘 要

基于 NFV 平台利用 GPU 构建高性能网络入侵检测系统的研究。

网络功能虚拟化 (NFV) 是通过虚拟化技术，将网络节点的功能，用软件的方式实现。因为能够摆脱硬件对传统网络功能的限制，所以网络功能虚拟化成为一种新型的网络架构的趋势。同时，网络带宽增长迅猛，千兆带宽已经普及，已然超过网络包传统处理的能力。因此 Intel 开发 DPDK 进行数据包的高速处理，可支持千兆以上带宽。另外，摩尔定律开始失效。CPU 发展减慢，不能满足网络数据包处理的速度要求。进而，GPU 成为加速网络数据包处理的热点。本文是利用 DPDK 和 GPU，在 NFV 平台上实现高性能网络入侵检测系统。

本文对开源的网络入侵检测系统 Snort 进行分析，对网络功能虚拟化技术进行浅析，对目前已有的高速处理网络包的工具 DPDK 进行介绍。然后，在网络功能虚拟化的平台上，利用 DPDK，设计实现在 GPU 上运行的高性能网络入侵检测系统。实验结果表明基于 NFV 平台利用 GPU 构建的高性能并行网络入侵检测系统的性能，较传统的网络入侵检测系统，有了可观的性能改进。在不影响网络入侵检测正确率的情况下，系统数据吞吐量、入侵检测速度等性能，相比于原有系统提升较大。接下来，对并行网络入侵检测系统进行进一步优化，对比不同参数下的性能，选择最佳参数，使其性能进一步提升。

关键词： NFV 平台, GPU, DPDK, 网络入侵检测系统

ABSTRACT

Network functions virtualization (NFV) is a virtualization technology, using software to achieve a network node function. Because it can get rid of the hardware limitations of traditional network functionality, the network function virtualization becomes a new popular network architecture. At the same time, the rapid growth of network bandwidth, gigabit bandwidth has been popular, already beyond the capacity of conventional network packet processing. So Intel developed DPDK high-speed packet processing which can support more than gigabits bandwidth. In addition, Moore's Law begins to fail. CPU slowing development can not meet the speed of the network packet processing requirements. Further, GPU becomes hot in accelerating network packet process. This article combines DPDK and GPU, realizing high-performance network intrusion detection systems on the NFV platform.

This project is based on the OpenNetVM platform, using Docker container, DPDK and GPU, high-performance network intrusion detection system. First, through the analysis of Snort rules, rules are fully configured. Then, OpenNetVM platform supporting DPDK, asynchronously receives data packets and packets for multiple-pattern string matching. We use Aho-Corasick algorithm as the multi-pattern string matching algorithm needed and realise it on the GPU. Finally, the system will return results to the user. The experimental results show, the performance of parallel network intrusion detection system greatly improves, almost close to the transmission speed of the network packets.

Keywords: NFV, GPU, DPDK, NIDS

第一章　概述

1.1　选题的背景与意义

传统模式下，电信运营商和网络运营商需要部署大量不同功能的网络设备，如服务器、路由器、防火墙等，来支持运营商提供的网络服务。随着运营商服务规模的逐渐增大，更大规模的网络设备需要被部署。然而，庞大数量的网络设备意味着，除了占用大量空间和能源外，运营商需要对不同的网络设备进行管理和维护。对硬件的管理和维护是困难、不易操作的。与此同时，运营商对硬件设备的定制开发的要求越来越高，速度越来越快。硬件设备的生命周期因此变得越来越短。但事实上，硬件设备的开发速度已然追赶不上运营商的要求和庞大用户群的需求，成为制约运营商扩大规模、开发更多更好更具创新性网络服务的瓶颈。

运营商联盟因此提出网络功能虚拟化 (NFV)，旨在通过标准的 IT 虚拟化技术，将不同类型的网络设备的功能整合，从而可以在数据中心、网络节点以及用户终端实现控制、调度等管理。网络功能虚拟化可以用软件的方式在类似虚拟机的平台上实现网络功能，并将这些功能运行在一系列标准工业服务器上。网络功能可以像软件一样被安装、并且可以和平台一起移动到网络中任何被需要的机器上。运营商不再需要安装新的设备，也不再需要安装特定网络功能的硬件设备。运行标准工业服务器所消耗的能源也要远小于运行特定网络功能设备所需要的能源。所以，借助网络功能虚拟化技术，比如实现网络入侵检测的功能模块，运营商能够大大降低运营、管理、维护等成本，并且能够加快业务成熟周期。

随着网络技术的快速发展，网络带宽迅猛增长，10Gbps 的网络带宽开始普遍，因此对网络数据包的处理速度要求越来越高。在目前 Linux 的架构下，网卡收到网络包后调用 netif_rx 或者 NAPI 层的 rx；然后挂载到对应 CPU 网卡的 queue 中；接着通过对 CPU 的中断将网络包向上传递。CPU 的中断，数据包在内存间的多次复制，内存管理的策略不当等原因，导致这种处理方式不适用于高速网络。尤其对实时性要求高的网络应用程序，比如网络入侵检测系统。如果网络入侵检测的探测引擎不能高速运行，入侵检测系统则不能达到实时性，进而带给系统无限的安全隐患。所以 Intel 开发一组能够快速处理数据包的平台和接口，即 DPDK，可以帮助解决传统处理数据包过程繁杂且速度慢等问题。

此外，摩尔定律也即将走到尽头。当硅片上线条的宽度达到纳米数量级后，采用现行工艺的半导体器件已经达到上限，无法正常工作。硬件发展速度减缓，CPU 主频、每秒运行的指令条数等性能指标增长减慢，已经不能满足高速网络的处理需求。2007 年 NVIDIA 提出用 GPU 并行处理来加速计算，即指同时采用图形处理单元 (GPU) 和 CPU，以加快程序速度。目前已有很多网络功能开始被广泛利用 GPU 进行实现。而网络入侵检测，作为网络功能中的重要一部分。系统采取积极主动的安全防护策略，实时对网络传输进行监控，通过对数据包内

容和规则集的规则进行的模式匹配分析，对可疑的数据传输向用户发出警报或是主动做出抵御行为。面对容量大的规则集，分析数据包的过程，在 CPU 上运行的效率也是远远低于 GPU。CPU 的处理速度也已不能满足高性能网络的要求。因此，在 GPU 上实现网络入侵检测系统具有必要性的。

当前，国内国际对网络入侵检测系统的主流用软件实现的方案，Snort，并没有利用 GPU 高速计算以及 DPDK 技术的特点实现，只是利用单个 CPU 核实现入侵检测的模式匹配的功能。最近的入侵检测系统的处理方案，如 Suricata 和 SnortSP，虽然实现了多个 CPU 核实现，但性能提升仍然受到没有对硬件最优使用的限制，导致性能不佳。还有一部分对网络入侵检测系统的研究，虽然利用 GPU 并行计算的优势，但大多不涉及网络功能虚拟化平台以及 DPDK 技术，例如：基于 Snort 利用 GPU 实现的网络入侵检测系统 Gnort，以及利用 GPU 实现多模式匹配的入侵检测系统 MIDeA。虽然速度提升明显，但缺点也比较明显，也就是可移植性差，难以简单移植到其他环境中，因此具有局限性。

从以上几点可以看出，在 NFV 平台上利用 GPU 构建高性能网络入侵检测系统具有开拓性的意义。

1.2 本文的主要工作

本文是基于 OpenNetVM 平台，利用 Docker 容器、DPDK 和 GPU，实现高性能网络入侵检测系统。首先，通过对 Snort 规则的解析，完成规则模式配置，并将规则配置到对应的数据结构中。然后，利用 OpenNetVM 平台支持的 DPDK 技术，异步实现接收数据包，和对数据包的多模式匹配。在 GPU 上实现的多模式匹配是利用 Aho-Corasick 算法 (AC 算法)，先将解析的数据结构转化成适用于 GPU 运行 AC 算法的数据结构，再将它传入 GPU 的全局存储器中，再利用 CUDA 实现 AC 算法。最后，系统会将匹配结果返回给用户。

1.3 本文的章节结构

第一章是绪论, 介绍研究的背景意义和本文的工作内容以及章节结构。

第二章是对研究过程使用的具体工具：开源网络入侵检测系统 Snort 和其使用的 Snort 规则集、虚拟化平台 Docker、DPDK 以及基于 Docker 和 DPDK 实现的网络功能虚拟化平台 OpenNetVM 进行简要介绍。

第三章是详细介绍基于 NFV 平台利用 GPU 构建高性能网络入侵检测系统的实现思路和实现细节，包括数据结构、实现算法、移植代码到平台等。

第四章是对构建的高性能网络入侵检测系统进行性能测试和性能分析，并与传统的实现网络入侵检测系统的方法的性能进行对比，得出性能评估结果。

第五章是对根据性能评估结果，对所完成的搭建工作进行总结，并对未来下一步的实现进行展望。

第二章　相关技术

2.1　Snort

Snort 是用 C 语言实现的多平台、实时开源入侵检测系统。Snort 拥有三种工作模式，分别是嗅探器、数据包记录器和网络入侵检测系统。Snort 的嗅探器模式只是接收数据包并作为连续不断的流显示在终端上；数据包记录器模式记录数据包内容到硬盘上；网络入侵检测模式是三个模式中最复杂，同时是可配置的。

Snort 是基于误用检测的网络入侵检测系统。通过 Snort 规则来描述带有攻击标识的数据包，再对网络中的数据包进行一一匹配，进而判断是否为“坏”包。所以，Snort 规则可称为 Snort 的核心。没有 Snort 规则文件，网络入侵检测模式就不能正常工作。遵照 Snort 规则的规范，用户可以根据自身需求，书写添加所需的新规则。

Snort 规则组成分为两大部分：规则头 (Rule Header) 和规则选项 (Rule Option) 。规则范例如下所示：

alert tcp any any -> 192.168.1.0/24 111 (content:”|00 01 86 a5|”; msg: ”mounted access”;)

在上例中，规则头部包含动作 (alert) 、协议 (tcp)、源和目的 IP 地址及掩码 (any 或 192.168.1.0/24) 、源和目的端口号 (any 或 111) 及方向 (->)。括号里是规则选项，包含报警的消息和要检查的包内容。Content 后面引号中的内容是需要匹配的特征字符串。msg 后面引号里面的内容是显示的报警消息。除 content 和 msg 标签外，规则选项部分还支持其他标签。在规则选项中占比最大的是 content 类的标签。content 类中除 content 标签外的其它标签如图2.1所示。

Table 3.5: Content Modifiers

Modifier	Section
nocase	3.5.5
rawbytes	3.5.6
depth	3.5.7
offset	3.5.8
distance	3.5.9
within	3.5.10
http_client_body	3.5.11
http_cookie	3.5.12
http_raw_cookie	3.5.13
http_header	3.5.14
http_raw_header	3.5.15
http_method	3.5.16
http_uri	3.5.17
http_raw_uri	3.5.18
http_stat_code	3.5.19
http_stat_msg	3.5.20
fast_pattern	3.5.22

图 **2.1**　Snort 规则选项中的 content 类标签

2.2 虚拟化平台 Docker

Docker 是一个开源的基于 LXC 构建容器的引擎，可以为任何应用创建一个可移植的、轻量级的容器。Docker 的容器只是对操作系统内核进行抽象处理。如图2.2所示，Docker 的容器 (container) 是基于 libcontainer，运行在宿主机上，只包含相关依赖库和应用程序，彼此之间又是相互隔离的，所以容器拥有轻量、高效、占用资源少等特点。

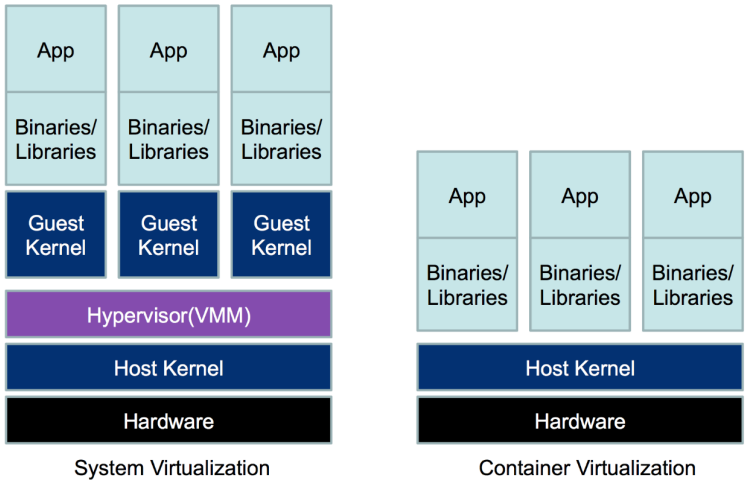

图 **2.2** 虚拟机和 Docker 容器的架构对比

Docker 有两个重要的基本概念：Docker image 和 Docker container。

1) image: 镜像。Docker 镜像是一个只读的模版，包含需要运行的文件。镜像可以创建 container，执行多个 container。

2) container: 容器。容器是 Docker 的运行组件，包含了运行应用程序所需要的所有东西。每一个容器都是从镜像中创建出来，相当于一个安全的、隔离的应用程序平台。

Docker 的层次结构如图2.3所示。最底层是由核心系统和多种文件系统构成，底层之上是镜像，镜像分为基础镜像和普通镜像，镜像之上是容器。Docker 的基础镜像上可以建立容器，并在容器上添加应用，如图2.3中所示的 Apache 或 emacs，再通过提交操作可以直接生成镜像。从而在任何地方，用户都可以启动此镜像，运行添加的应用程序。

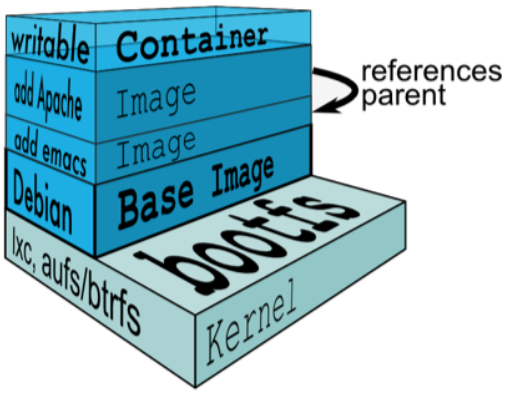

图 **2.3**　Docker 的基本结构

Docker 具有以下主要特性：

- 文件系统隔离：每个进程容器运行在完全独立的根文件系统里。
- 资源隔离：可以使用cgroup为每个进程容器分配不同的系统资源，例如CPU和内存。
- 网络隔离：每个进程容器运行在自己的网络命名空间里，拥有自己的虚拟接口和IP地址。
- 写时复制：采用写时复制方式创建根文件系统，这让部署变得极其快捷，并且节省内存和硬盘空间。
- 日志记录：Docker将会收集和记录每个进程容器的标准流（stdout/stderr/stdin），用于实时检索或批量检索。
- 变更管理：容器文件系统的变更可以提交到新的映像中，并可重复使用以创建更多的容器。无需使用模板或手动配置。
- 交互式Shell：Docker可以分配一个虚拟终端并关联到任何容器的标准输入上，例如运行一个一次性交互shell。

图 **2.4**　Docker 的主要特性

2.3　Intel DPDK

DPDK 是由 Intel 开发的一个开源的数据平面开发工具集，包含一系列库函数和驱动，为 Intel 架构的处理器提供高速处理数据包的支持。Linux 内核会将 DPDK 应用程序看作一个普通的用户态进程。它绕过 Linux 内核协议栈，直接依靠数据平面库进行数据包收发。DPDK 框架如图2.5所示，它通过环境虚拟层 (EAL) 为 DPDK 应用程序提供通用接口，实现 DPDK 运行的初始化工作：基于大页的内存分配、PCI 设备地址映射到用户空间等初始化操作，以方便应用程序访问。

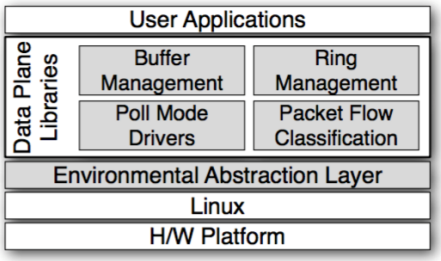

图 **2.5**　DPDK 的架构

DPDK 的库函数主要包括：

1) 队列管理：为了减少不同软件组件访问队列的等待时间，DPDK 实现安全无锁的队列来代替 Linux 底层提供的“自旋锁”(spinlock)。

2) 内存管理：在大页 (huge page) 内存空间中分配内存池给目标文件；并利用环 (ring) 来存储目标文件。

3) 缓存管理：为了减少操作系统大量消耗在分配和取回缓存的时间，DPDK 提前分配好固定大小的缓存，存放在内存池中。

4) 轮询模式驱动：为了提高网卡收发性能，DPDK 利用轮训模式驱动可以消除大量中断导致的响应延迟。

5) 流分类：为了更快地将数据包放到对应的流中进行处理，Docker 基于多元组信息利用 Intel SSE 实现高效的哈希算法处理流分类。

DPDK 的基本技术主要包括：

1) 大页技术：核心是将正常标准页的大小 4KB 扩展为更大的页表尺寸，从而 TLB (Translation Lookaside Buffer) 可以指向更大的内存区域，减少 TLB miss 的情况，加快读取内存的速度。DPDK 使用大页技术，所有内存从大页中分配，并且预先为数据包分配等大小的 mbuf，从而实现对内存池的良好管理。

2) 轮询技术：传统模式是当数据包到来时，需要 CPU 中断，通知应用程序处理报文。在 DPDK 的实现中，当网卡收到数据包时，直接将数据包存放到 cache 或内存，并设置报文到达的标志位。DPDK 利用轮询技术，使得应用程序轮询报文到达的标志位，判断是否有新的报文等待处理，从而不需要 CPU 中断。中断次数的大大减少，极大提升了应用程序处理报文的能力。

3) CPU 亲和技术：现代操作系统是基于分时的任务调度方式。在多核处理器下，是多个进程或线程在一个核上交替执行，导致进程或线程之间的切换开销较大。DPDK 把系统相应的线程和某一个核进行绑定，使该线程尽可能使用独立的资源进行处理。

2.4 OpenNetVM

OpenNetVM 是一个基于 Intel DPDK 和 Docker 容器实现的高性能网络功能虚拟化 (NFV) 平台。如图2.6所示，OpenNetVM 基于两种通信通道进行实现：一种是较小的环状的共享内存，用于实现数据包描述符在管理者和网络功能模块的客户之间的传递；另一种是大页，支持网络功能模块直接读写数据包。网卡接收到数据包后，会把数据包的内容通过 DMA 直接存放到大页区域。网络功能管理模块为每个网络功能模块都建立一个数据包描述符，并将描述符存放到环状缓存中。描述符标识不同的网络功能模块，标记数据包在大页中的偏移等等信息。描述符的应用更好实现了网络功能模块对共享内存中数据包的读写和管理。网络入侵检测系统模块，正是利用数据包描述符，通过数据包在大页共享内存的位置，访问数据包，并对数据包内容进行判断。

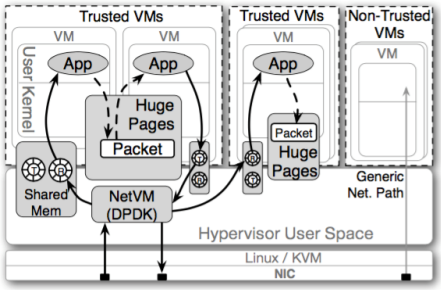

图 **2.6**　OpenNetVM 的内存移动示意图

OpenNetVM 具有以下主要特点：

1) 基于容器的网络功能模块：由于网络功能模块在 Docker 容器中是以标准用户进程来运行，所以对网络功能模块的写和管理都很容易。

2) 拥有网络功能管理模块：网络功能管理模块负责跟踪网络功能模块的运行情况，并且在数据包到来时，分发给对应的网络功能模块。

3) 支持软件定义网络 (SDN)：网络功能管理模块能够和 SDN 的控制器协同配合，使控制器能够具体控制处理一个数据流的多个网络功能模块。

4) I/O 的零复制策略：网络数据包会通过 DMA 直接被放入共享内存中，网络功能管理模块能够允许网络功能模块直接从共享内存中读取数据包，而不需要多余的复制过程。

5) 能够识别非统一内存访问架构 (NUMA)：OpenNetVM 能够识别 NUMA，即确保数据包被存放在对应 socket 的本地 DIMMs 的大页中，而且只被运行在此 socket 的线程处理。

6) 无中断：OpenNetVM 使用 DPDK 的轮询模式驱动，取代原有的基于中断模式，进而使得系统处理数据包的速度能够超过 10Gbps。

7) 大规模性：网络功能模块可以被复制使用以扩展其规模；同时，网络功能管理模块可以自动实现线程处理数据包的负载均衡，从而达到性能最大化。

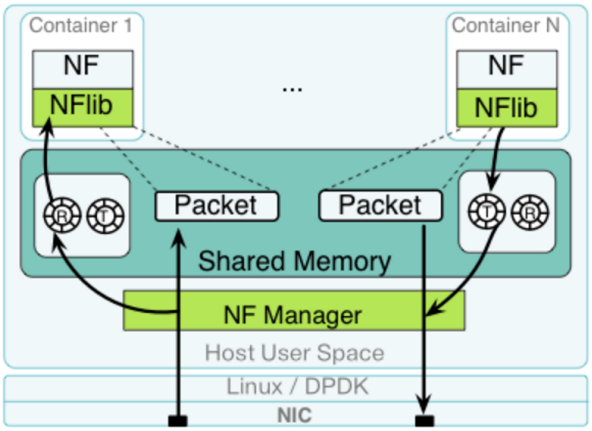

图 **2.7**　OpenNetVM 的基本结构

第三章　基于 NFV 平台利用 GPU 构建高性能网络入侵检测系统的实现

3.1　在 OpenNetVM 上实现 NIDS

3.1.1　整体思路

OpenNetVM 是基于 Docker 容器和 Intel DPDK 实现的平台。在 OpenNetVM 上实现将网络入侵检测系统 (NIDS) 的架构如图3.1所示。针对实现架构，思路首先是需要在 OpenNetVM 上，对 OpenNetVM 所需要的组件进行配置，如：配置大页内存、建立容器等。然后，将网络入侵检测系统的三大模块移植到 OpenNetVM 的基于容器的网络功能模块中。

网络入侵检测的三大基本模块是：1) 前期在网络功能 CPU 模块，对于 Snort 规则的预处理模块；2) 中期在网络功能 GPU 模块，利用 GPU 实现的模式匹配的核心模块；3) 后期在网络功能 CPU 模块，对于检测数据包的行为处理。

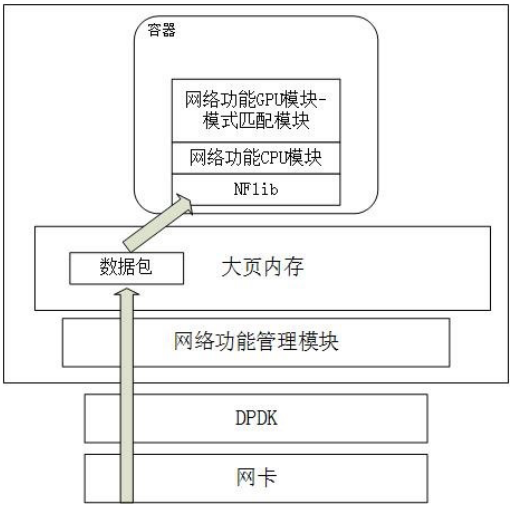

图 **3.1**　网络入侵检测系统的实现架构

3.1.2　OpenNetVM 的相关配置

DPDK 的相关配置：插入 IGB_UIO 模块；检查系统是否还有空闲的大页。如果有，则从内存中分配 1024 个 2048kb 大小的大页内存；查看所有的网卡状态，再从中绑定 DPDK 所需要的网卡。

Docker 的相关配置：首先搭建支持 CUDA 运算平台的镜像；然后便是在镜像上建立映射宿主机的相关设备配置的容器；最后将入侵检测系统的应用程序映射到 Docker 容器内。其中，宿主机的相关设备指的是 DPDK 支持的网卡、系统分配的大页内存。

3.1.3 网络入侵检测系统的模块移植

网络入侵检测系统的三个模块依次是：前期的规则预处理模块、中期核心模式匹配模块和后期对检测的包行为处理模块。前期是在 CPU 上执行，执行结束后则保存在 CPU 内存和 GPU 内存中。中期是在 GPU 上运行，运行结果返回给 CPU。后期是在 CPU 上运行。

3.1.3.1 前期规则预处理模块

从规则文件中按字符读取规则。由于 Snort 规则遵照严格的格式要求，根据格式要求将读取的每一块信息整合后存入数据结构即可。Snort 规则分为规则头部和规则选项。规则头部是括号前的内容，规则选项是括号内的内容。示例如下。

alert tcp any any -> 192.168.1.0/24 111 (content:”|00 01 86 a5|”; msg: ”mounted access”;)

规则头部的每一块信息是由空格进行区分。信息块的内容依序是规则动作、协议 (tcp)、左侧 IP 地址与网络掩码、左侧端口号、方向、右侧 IP 地址与网络掩码、右侧端口号。方向信息块有三种类型：左方向 (->) 、右方向 (<-) 、双方向 (<>) 。方向信息块的存在，增加用户书写 Snort 规则的便利。左侧 IP 地址与网络掩码、左侧端口号，既有可能是源 IP 地址与网络掩码、源端口号，也有可能是目的 IP 地址与网络掩码、目的端口号。右侧 IP 地址与网络掩码、右侧端口号同理。源和目的的确认是通过方向信息块决定。方向信息块总是由源信息指向目的信息。所以在上例中，方向信息块属于左方向，即左侧的第一个 any 是源 IP 地址与网络掩码，左侧的第二个 any 是源端口号，右侧的 192.168.1.0/24 是目的 IP 地址与网络掩码，右侧的 111 是目的端口号。字符串 any 可以应用在 IP 地址与网络掩码和端口号两种信息块。any 表示对于任何的 IP 地址与网络掩码或对于任何的端口号都是匹配的。

规则选项的每一块信息是由分号区分。每一块信息的内部由于标识的内容不同，所以内部格式有所区别。具体来说，规则选项的每一块信息内是由两种组织方式。一种是直接由一个部分组成，此部分为一个标识符标签，如 nocase 标识符，表示匹配不区分大小写。另一种是由两个部分组成，标识符标签和带引号的字符串，两个部分是由冒号区分开，如 msg: “mounted”，表示标识符 msg 标签的内容是 mounted。所以对于不同组织方式的规则选项内容，要用不用的存储方式。

模块的流程图：

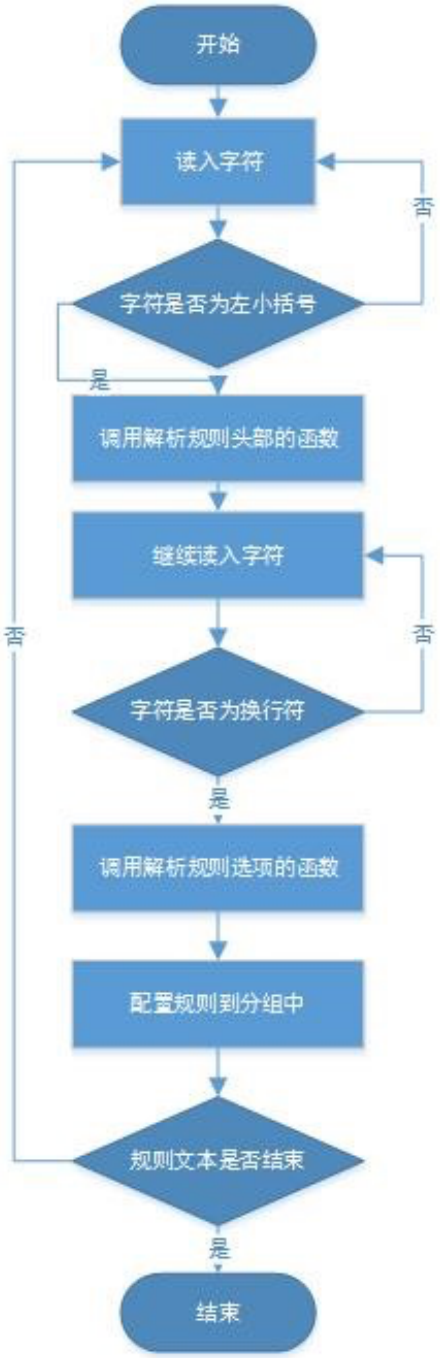

图 **3.2**　前期规则预处理模块的流程图

规则预处理模块首先单个单个字符读入，每个规则占一行，所以遇到换行符之后则进入下一条规则处理。在处理本行规则时，当第一次读入左小括号时，则判断规则头部读入结束，将规则头部的字符串传入负责解析规则头部的函数中。当读到换行符之前的右小括号时，则判断规则选项读入结束，将规则选项的字符串传入负责解析规则选项的函数中。

进入解析规则头部的函数时，函数主要根据空格区分不同的信息块。且 Snort 中严格规定信息块的顺序依次为：规则动作、协议 (tcp)、左侧 IP 地址与网络掩码、左侧端口号、方向、右侧 IP 地址与网络掩码、右侧端口号。所以，可以把

字符串中对应的信息块的信息，存放到规则头部的节点中。

进入解析规则选项的函数时，函数主要根据分号区分不同的信息块。对于信息块内部，存在两种组织方式：一种是只有一个标签组成的信息块；另一种是由标签和对应内容组成的信息块。第二种组织方式的内容部分往往是由引号标识。通过识别不同的信息块和信息块对应的标签，就可以将对应信息存放到规则选项的节点中。

3.1.3.2 中期模式核心匹配模块

网络入侵检测系统的核心模块是模式匹配检测。面对庞大的 Snort 规则集，传统的模式匹配算法的时间复杂度呈指数级，已然不能满足网络入侵检测实时性的要求。所以本系统采用时间复杂度呈线性的 Aho-Corasick 算法，大大加快模式匹配算法的运行速度。

Aho-Corasick 算法保证了匹配算法的时间复杂度不会由 Snort 规则集构成的模式的数量决定，而是由网络传来的数据包的大小决定。Aho-Corasick 算法近似与有限状态机。通常使用树的数据结构。在构建状态树时，每一个节点代表一个状态。若有对应的状态则选择这条路径走下去；若没有对应的状态则在当前节点后面添加新的状态。在使用状态树时，根据对应的状态沿着相应的路径，往叶子节点的方向前进。但如果此条路径的下一个状态和目标状态不一致，Aho-Corasick 算法会提前构造失败函数，根据失败函数的状态转移到新的路径上，继续向前，从而保证 Aho-Corasick 算法的时间复杂度是 O(n)。

3.1.3.3 后期对检测的包行为处理模块

在中期模式核心匹配模块完成后，会根据匹配情况返回匹配的规则模式的 ID 和在数据包中匹配的位置。根据规则模式 ID 找到对应的规则动作。

Snort 支持的处理操作共有五种，包括发出警报 (alert)，不处理 (pass)，生成日志记录 (log)，动态处理 (dynamic) 和激活 (activate)。Snort 规定，处理操作显示在规则头部的第一个信息块，即规则动作中存放。在入侵检测系统的实际应用中，规则动作以发出警报为主。发出警报则是把对应规则的规则选项中 msg 标签对应的内容发送给用户，以起到警示的目的。

3.2 利用 GPU 加速 NIDS 的模式匹配

3.2.1 整体思路

由于在 GPU 上实现 Aho-Corasick 算法，GPU 不支持从 CPU 传地址到 GPU，另外为了防止实现 Aho-Corasick 算法时线程发散，所以入侵检测系统采用二维数组代替状态树。二维数组的一个维度是状态，另一个维度是 256 个字符。而二维数组里的每一个元素都是 4 字节的数据，其中前 2 个字节存放下一个状态的

索引，后 2 个字节存放是否为最后匹配状态的标识。另外，利用一维数组构建失败函数，即对每一个状态都拥有一个失败函数值。失败函数值指向的是另一个状态。两个状态的关联性在于，到达这两个状态的两个匹配拥有相同的后缀。失败函数值指向的状态是拥有较短匹配的状态。匹配结束后，返回给 CPU 的结果是匹配规则模式的 ID 和在数据包中匹配的位置。

Aho-Corasick 算法构造的二维矩阵的规模较大，占用 GPU 大量内存的同时，也会增加模式匹配时的时间开销。而从网络中获得的数据包的端口号是可以预先获得，因此根据端口号，将数据包提前分组，再利用该组中规模较小的二维数组进行模式匹配，可以加快处理速度。

此外，在 GPU 上实现 Aho-Corasick 算法，还拥有两种并行方案：

1) 将每个数据包拆分成固定的长度，每个线程处理被拆分的一部分。但是为了防止因为拆分而导致“跨越”多个部分的模式无法正常匹配，每个线程在匹配结束拆分的固定长度后，还会再向后面进行一定长度的匹配。此扩展匹配的长度是由最长模式的长度决定；

2) 每个线程单独执行一个数据包的模式匹配。但这种方案会导致有些线程先完成匹配后需要等待其他线程完成后再开始匹配新的数据包。但根据 xxx 在实验中的测试分析，

这两种方案的性能之间并没有什么差别。所以最后由于第二种方案具有更易实现的特点，所以才用第二种并行策略。

所以要对数据结构和各个功能模块都进行优化。虽然构造 Aho-Corasick 二维状态数组的消耗较大，但由于构造二维数组的过程是在预处理过程中进行，所以实际在 GPU 上实现的只有 Aho-Corasick 算法的匹配模块。因此入侵检测系统的模式匹配模块的算法复杂度依然只于数据包的长度有关，性能依旧出色。

3.2.2 基本数据结构

对于从规则文件读入的规则，由于规则分类情况较多，如对于不同的规则动作，对于不同的处理协议，不同的端口号，都决定了不同的规则子集。所以为了方便利用 GPU 实现模式匹配，所以采用多层链表式数据结构。具体结构如下图所示。

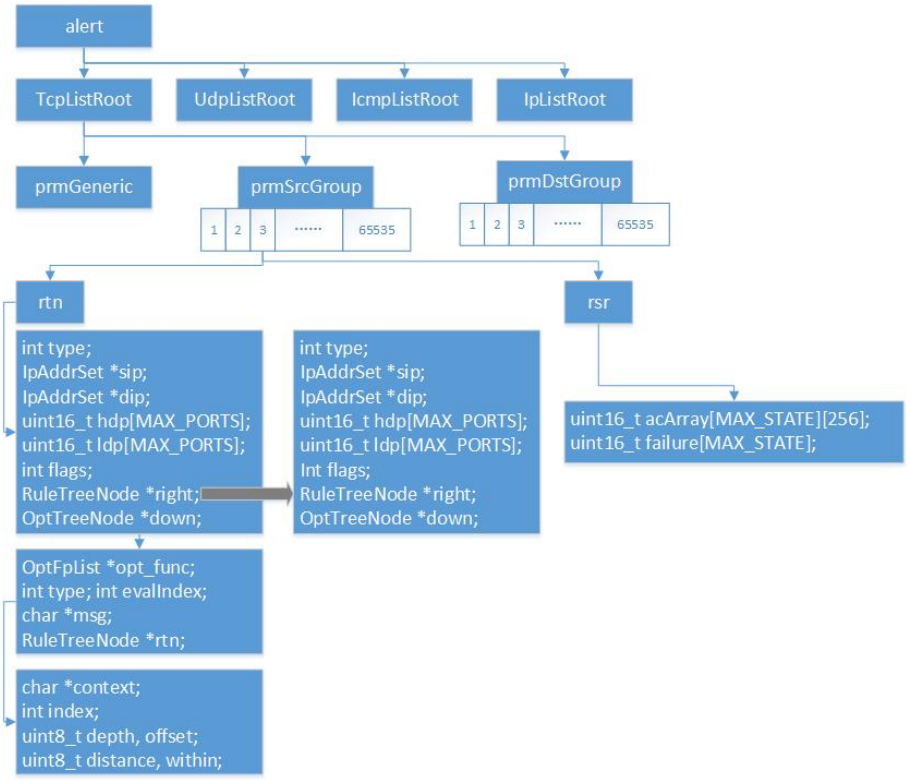

图 **3.3**　基本数据结构

对于规则动作为 alert，即发出警告类型的规则，第一层是根据 Snort 规则集目前支持的四大类协议进行分类，分别是 TCP, UDP, IP 和 ICMP。不同协议的规则对于规则头部端口号的限制范围也进行分类，构成第二层。如果规则的源端口号给定具体的值的情况下，则将规则添加到源端口固定值的分类数组 (prmSrcGroup) 中的对应元素中；如果规则的目的端口号给定具体值的情况下，则将规则添加到目的端口固定值的分类数组 (prmDstGroup) 中的对应元素中；如果规则的某一端口给定的是一个范围，则将规则放到最后一个分类组 (prmGeneric) 中。

第二层数据结构的下一层，即第三层，是分成两个元素，分别为 rtn 和 rsr。rtn 指向的下一层，即第四层，是读取规则文件构造的树状结构的，类型为 RuleTreeNode 的节点。rsr 指向的下一层是读取规则结束后，将已有的规则数据结构，即 rtn 指向的树状结构，转化为 Aho-Corasick 算法需要的适用于 GPU 的数据结构。其类型为 RuleSetRoot。

具体来说，第四层类型为 RuleTreeNode 节点是存储的规则的头部信息，如高/低位的 IP 地址喝网络掩码、高/低位的端口号等信息。类型为 RuleTreeNode 节点同时包含指向这一层数据结构右边的包含另一个规则头部信息的同类型的节点，和指向下一层数据结构的类型为 OptTreeNode 的节点。类型为 OptTreeNode 的节点记录规则类型、规则索引号、规则 msg 标签以及下一层数据结构中的 content 类标签的类型为 OptFpList 的节点。OptFpList 的节点除了包含指向下一个 OptFpList 类型的节点外，还有 content 类标签的基本信息和修饰信息，如 content

标签的内容，content 标签内容的深度、偏移、最大距离、最小距离等。由于一个规则的内部可能包含多个 content 标签，也就是存在多组修饰不同 content 的标签。所以在类型为 OptFpList 的节点中还包含指向下一个 OptFpList 类型的节点。

第三层中另一个元素 rsr 指向的是第四层的类型为 RuleSetRoot 的节点。该节点保存的信息是经过处理后的规则信息，包含 Aho-Corasick 的二维状态数组和由失败函数值构成的一维数组等信息。

3.2.3 规则预处理模块的 GPU 优化

规则预处理模块是对 Snort 规则进行读入、解析、转化为 Aho-Coraisck 算法需要的并适用于 GPU 运行的数据结构。在完成对规则头部和规则选项的解析后，需要将规则头部类型为 RuleTreeNode 的指针指向的节点和规则选项类型为 OptTreeNode 的指针指向的节点连接起来，并根据规则的类型和规则的端口号情况，将规则放到对应的分组中。

3.2.4 模式匹配模块的 GPU 优化

3.2.4.1 模式匹配算法 Aho-Corasick 的预处理的实现细节

不同规则协议和不同端口号分组的规则分别实现 Aho-Corasick 算法的预处理。Aho-Corasick 算法的预处理分为两部分：一部分是构造状态转换的二维数组；另一部分是构造失败函数的一维数组。

1. 构造状态转化的二维数组：

构造过程的伪代码如下所示。

```
1  i := 0
2  state := 0          // 初始状态为 0
3  while 1 do
4          ch := packet[i]
5          if packets[i] == ‘\0’ then break
6          else if ACarray[state][ch] == 0 then
7                  break
8          else
9                  state := ACarray[state][ch]
10                 i++
11         endif
12 end while
13
14 if packets[i] != ‘\0’ then
15         while 1 do
16                 ACarray[state][ch] := newState
17                 // 增加新状态
18                 state := newState
19                 i++
20                 ch := packet[i]
21         end while
22 endif
23
```

二维数组 ACarray 的两个维度分别是转化的状态和 256 个字符。而二维数组里的每一个元素都是 4 字节的数据，其中前 2 个字节存放下一个状态的标号，后

2 个字节存放是否为最后匹配状态的标识。在构建状态转化的二维数组时，每一行代表一个状态。若在当前状态下，下一个输入的 256 个字符中的一个字符已经存在一个状态，则转化到此状态中继续操作。如果不存在对应的状态，则扩展一行新状态行，作为新状态存储到二维数组的元组中。

2. 构造失败函数的一维数组：

构造过程的伪代码如下所示：

```
1  q_num := 0
2  q_head := 0
3  state := 0
4
5  for i := 0 until 256 do
6          if ACarray[state][i] != 0 then
7                  queue[q_num] := ACarray[state][i]
8                  // 从初始状态能转移到的状态放入队列中
9                  q_num++
10         endif
11 endfor
12
13 while queue not empty do
14         state := queue[q_head]
15         for i := 1 until 256 do
16                 if ACarray[state][i] != 0 then
17                         queue[q_num] := ACarray[state][i]
18                         q_num++
19                         fail_state = failure[state]
20                         while ACarray[fail_state][i] == 0 do
21                           fail_state = failure[ACarray[fail_state][i]]
22                         end while
23                 endif
24         endfor
25 end while
```

基本思路是：按照深度由低到高进行计算，在计算深度为 d 的所有状态时候，首先考虑每一个深度为 d-1 的状态。迭代调用深度越来越低低的失效函数，直至有个最近的状态能匹配下一个字符。如果一直不能找到匹配时，则会到根状态。

具体实现如上伪代码所示，先讲深度为 1 的状态放入队列中，即将从初始状态能转移到的状态放入队列中。再进入 while 循环中，取出队列的头元素。将头元素能转移到的状态放入队列中。再对新加入队列的元素就失败函数值。每个失败函数值都是从上一层的失败函数值依次迭代，直到找到能够匹配的转移。

3. 状态转化的二维数组和失败函数的一维数组合一转化为一维数组：

构造一维数组的长度是状态数 x 257。将二维数组和失败函数的一维数组合一转化的目的是便于 Aho-Corasick 算法在 GPU 上的传参和运行。

3.2.4.2 模式匹配算法 **Aho-Corasick** 在 **GPU** 上的实现细节

实现过程的伪代码如下所示。

```
1  bid := blockIdx.x
2  tid := threadIdx.x
3  i = tid + bid * blockDim.x
4  k := 0
5  state := 0
6
7  while 1 do
8          ch := packets[i * MAX_LEN + k]
9          k++
10         while ACarray1D[state * 256 + ch] == 0
11                 state := failure[state];
12         end while
13         state := ACarray[state][ch]
14 end while
```

根据预处理构造转化的一维数组 ACarray1D，凭借数据包内的下一个字符 ch 和当前状态 state 共同决定下一个状态。如果下一个字符和当前状态决定的下一个状态已经通过预处理构造，则下一个状态为已经构造的状态；如果没有通过预处理构造，即不存在，则下一个状态为失败函数的状态。当数据包的字符全部匹配结束，如果找到匹配的模式，则返回匹配的模式 ID 和数据包中发生匹配的位置；如果没有找到匹配的模式，则都返回 0。

3.3 对检测数据包的行为处理模块的实现细节

GPU 上的模式匹配模块的匹配结束后，返回给 CPU，一个 batch 的数据包的发生匹配的模式 ID 和数据包中发生匹配的位置。每个数据包的结果找到对应规则后，产生规则动作，将警告信息显示给用户。

第四章　性能检测与分析

4.1　实验设置

测试网络入侵检测系统的性能，需要搭建的网络拓扑如下图所示。

图 **4.1**　实验设置的网络拓扑图

网络入侵检测系统运行在服务器端。服务器上首先需要安装 Docker，配置入侵检测系统所需要的 CUDA 运算平台，分配大页内存，挂载 10Gbps 的网络端口。然后，在 Docker 上搭建 openNetVM 平台，并把网络入侵检测系统的工程文件链接为 openNetVM 的网络功能模块的库文件。接着就可以运行服务器。当服务器端接收来自客户端发来的数据包，就对每一个数据包进行解析，提取数据包内容，再进行网络入侵检测的模式匹配。最后将入侵检测结果返回给用户。

在客户端上，也需要配置可以使用 DPDK 实现发包的程序，从而程序支持 10Gbps 的网络吞吐量。

服务器的配置参数，如下表所示。

操作系统	Ubuntu 15.04
内存大小	64 G
CPU 类型	Intel Xeon CPU
CPU 主频	2.3 GHz
CPU 内核数	20

表 **4.1**　服务器配置参数

客户端的配置参数，如下表所示。

操作系统	Ubuntu 15.04
内存大小	32 G
CPU 类型	Intel Xeon CPU
CPU 主频	2.4 GHz
CPU 内核数	16

表 **4.2**　客户端配置参数

此外，服务器端的入侵检测系统模式匹配功能，使用的 GPU 是 NVIDIA Tesla K40c。Tesla K40c 的性能指标如下表所示。

CUDA 驱动版本/运行库版本	7.5 / 7.5
全局存储器的大小	11520 MBytes (12079136768 bytes)
流处理器的个数	15
每个流处理器的 CUDA 核心数	192
GPU 的最大时钟频率	745 MHz (0.75 GHz)
每个流处理器支持的最大线程数	2048
每个线程块支持的最大线程数	1024

表 **4.3** Tesla K40c 性能指标

由于网络入侵检测系统的规则集固定，所以在保证匹配正确率为百分之百的情况下，实验首先关注系统在 GPU 上运行的模块的实现性能。通过调整不同批处理大小和不同线程数，观察 GPU 上运行的网络入侵检测系统模式匹配的性能指标。接着，将模式匹配模块放到 openNetVM 平台上，观察在平台上运行的 GPU 模式匹配加速模块的性能和之前比较是否有变化。判断在 openNetVM 平台上实现模式匹配是否对性能会有影响。然后，把网络入侵检测系统移植到 openNetVM 平台上，利用上述的网络拓扑结构，一个客户端向网络入侵检测系统所在的服务器发包，观察系统总性能。最后，将利用 GPU 实现的网络入侵系统总性能和利用 CPU 实现的网络入侵检测系统进行对比，比较得出利用 GPU 实现的性能总提升。

4.2 NIDS 在 GPU 上运行的性能

网络入侵检测系统在 GPU 上运行的模块是模式匹配模块。模式匹配模块是利用 GPU 实现的 Aho-Corasick 算法。服务器在本地读取数据包内容后，开始计时。数据包的内容先传给 GPU 全局内存，再进行模式匹配，最后将结果传回 CPU。整个过程进行指定次数后，计时结束。计算平均吞吐量。

4.2.1 在不同批处理大小下的 GPU 性能

在测试 GPU 上运行模式匹配的性能时，固定 GPU 的线程块数和每个线程块的线程数。改变批处理大小。最后计算求得平均吞吐量。

下图是在线程块数为 15，每个线程块的线程数为 1024，即单个线程块所能承载的最大线程数。批处理大小从 15360 (15 * 1024) 个数据包开始一直测到 1013760 (15 * 1024 * 66) 个数据包。每个数据包的大小为 1518 B。平均吞吐量如下图所示。

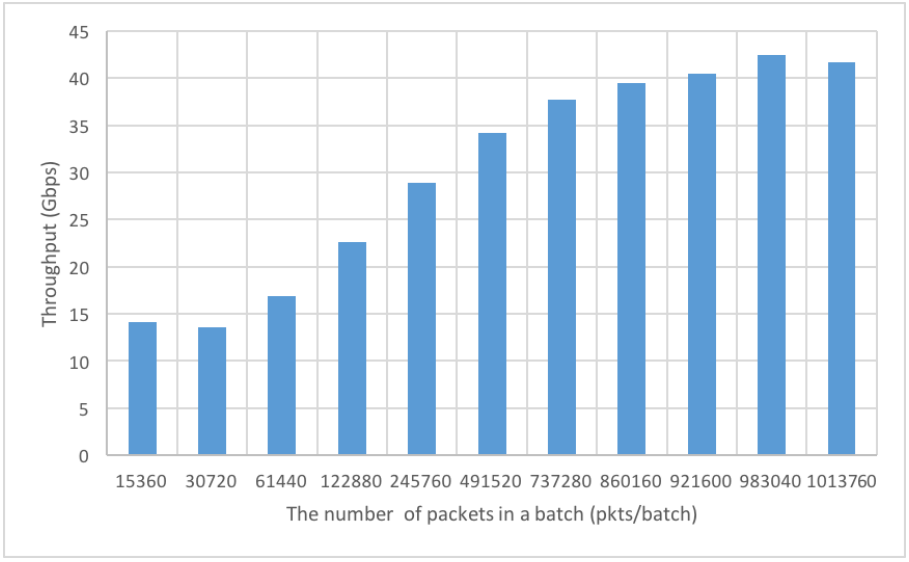

图 **4.2**　在不同批处理大小下的 GPU 吞吐量

图中显示，随着 GPU 核心函数承载的批处理大小的逐渐增大，GPU 运行的吞吐量也逐渐增高。但随着批处理大小的增大，GPU 运行的吞吐量的增大速度逐渐减慢。从图中可发现，当批处理大小达到 983040 (15 * 1024 * 64) 的情况下，GPU 运行的吞吐量达到最大值，约为 42.405 Gbps。当批处理大小继续增大时，GPU 的吞吐量则不再增加，基本维持在 40 Gbps 左右。

批处理大小增加意味着，在处理相同数量的数据包的情况下，从 CPU 向 GPU 拷贝数据的次数就会减少，进而减少了数据传输开销。同时，运行一次 GPU 的核心函数处理的数据越多。所以整体的吞吐量会逐渐增加。

但当批处理大小达到峰值之后，如果批处理大小继续增加，而 GPU 的计算性能已经达到最大，那么吞吐量会基本维持，不再改变。

4.2.2 在不同线程数大小下的 GPU 性能

在第一个实验中，测得 GPU 所能到达性能最好的批处理大小为 983040 个数据包。再固定批处理大小，和每个线程块所能运行的最大线程数 1024。每个数据包的大小为 1518 B。改变线程块的数目从 1 一直测到 30。最后计算求得平均吞吐量。下图是求得的平均吞吐量的变化图。

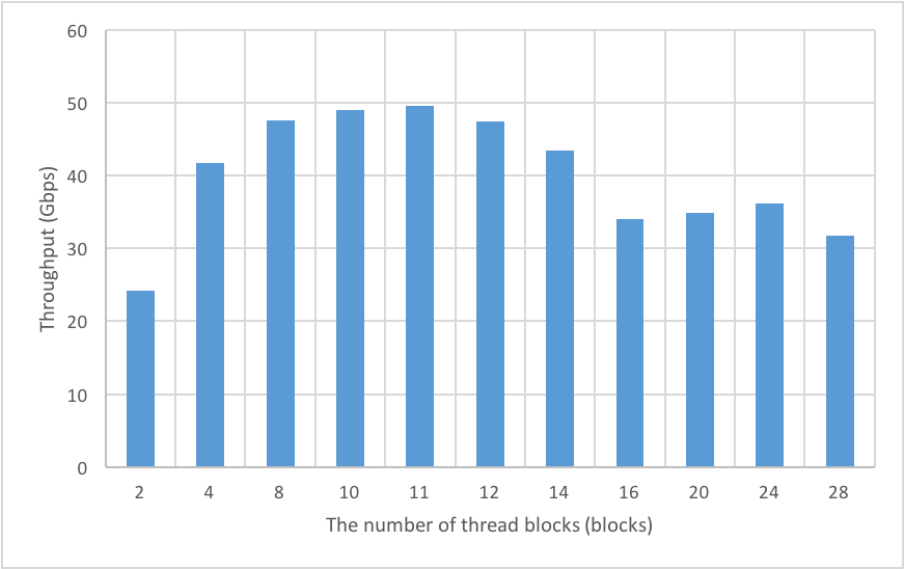

图 **4.3**　在不同线程数大小下的 GPU 吞吐量

由图所示，发现，随着线程块数量的不断增加，GPU 的吞吐量逐渐增大。但当吞吐量达到峰值后，继续随着线程块数量的增加，GPU 的吞吐量开始减少，最后维持不再变化。

图中显示，当线程块 8 ~ 12 的情况下，GPU 的吞吐量基本维持在 45 Gbps 以上。最佳情况是在线程块数为 11 时，吞吐量达到 49.506 Gbps。

线程块大小的增加，意味着 GPU 使用的流处理器的数量越来越多。流处理器越来越多，则 GPU 进行模式匹配的计算速度会越来越大。但是由于 GPU 所有的流处理器最多只有 15 个，所以当流处理器数量超过 15 个时，继续增大线程块数量，意味着，存在一部分流处理器需要处理的线程块的数量增加，另一部分流处理器处理的线程块数量不变。不变的流处理器需要等待改变的流处理器处理结束后，再结束此次 GPU 核心函数运行的模式匹配，进而导致 GPU 吞吐量减少。同时，增加线程块数，意味着单个线程需要处理的数据包数量是在减少。所以在多个因素的共同影响下，GPU 吞吐量最后基本维持不变。

理论上，当线程块数量为流处理器数量时，GPU 的吞吐量应该达到最大。但实际测试中，当线程块数为 11，小于流处理器数量时，GPU 的吞吐量就已经达到峰值。原因预估是多个因素互相影响导致。线程块数增多时，单个线程的访存的跳跃程度会越高。访存顺序的跳跃程度越高，开销也会增加。所以，线程块数未达到流处理器数量时，已经达到 GPU 处理性能的最优值。

4.3 NIDS 在 Docker 容器内用 GPU 运行的性能

在本地测试结束后，把网络入侵检测系统移植到 OpenNetVM 平台上，也就是 OpenNetVM 平台所基于的 Docker 容器中。观察 GPU 运行的性能是否会有改变。

4.3.1 在不同批处理大小下的 GPU 性能

同上，固定 GPU 的线程块数为 15，每个线程块的线程数为 1024。批处理大小从 15360 (15 * 1024) 个数据包开始一直测到 1013760 (15 * 1024 * 66) 个数据包。每个数据包的大小为 1518 B。在 OpenNetVM 平台上的平均吞吐量和在本地运行的 GPU 平均吞吐量对比如下图所示。

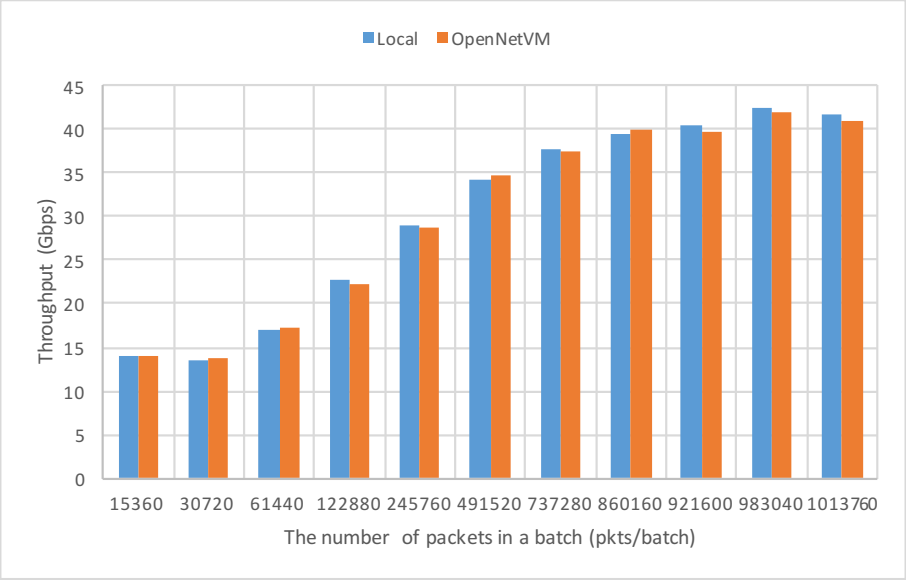

图 **4.4**　在本地和在 OpenNetVM 平台上不同批处理大小下的 GPU 吞吐量对比

图中，每一组左边的柱状表示的是在本地运行的 GPU 平均吞吐量，右边的柱状表示的是在 OpenNetVM 运行的 GPU 平均吞吐量。可以发现，在 OpenNetVM 平台上，即 Docker 平台上运行的 GPU 的性能没有太大影响，只有轻微的性能波动，可以忽略不计。

4.3.2　在不同线程数大小下的 GPU 性能

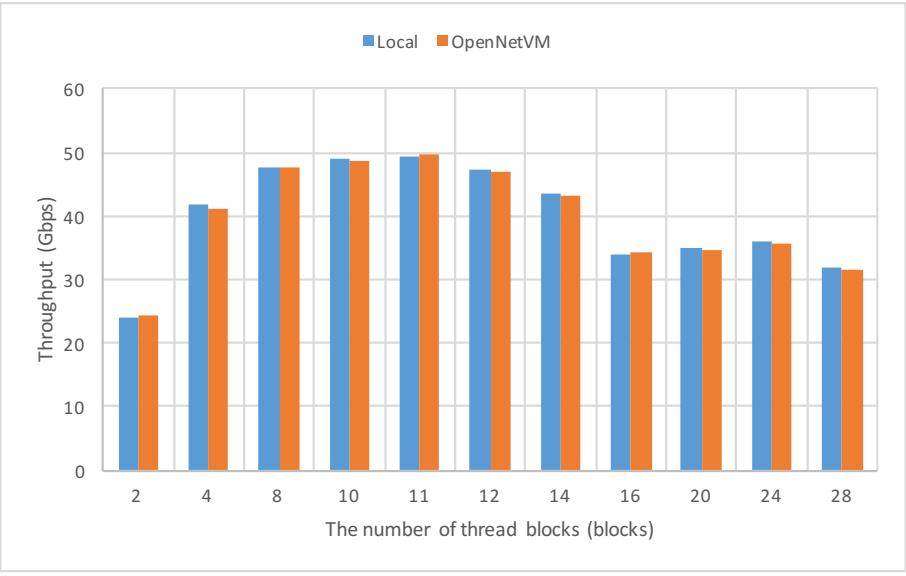

图 **4.5**　在本地和在 OpenNetVM 平台上不同线程数大小下的 GPU 吞吐量对比

图中，每一组左边的柱状表示的是在本地运行的 GPU 平均吞吐量，右边的柱状表示的是在 OpenNetVM 运行的 GPU 平均吞吐量。在能接受的性能数据浮动范围内，GPU 的性能没有受到太大影响。

所以在 OpenNetVM 平台上进行的两个实验表示，当把网络入侵检测系统的 GPU 运行模块移植到 OpenNetVM 平台上时，GPU 的运行性能不会有任何影响。

4.4 NIDS 在 OpenNetVM 平台上的整体性能

通过上述实验可以得出，GPU 运行的最好性能是在批处理大小为 983040 时，线程块数量为 11，每个线程块的线程数为 1024 时，GPU 运行性能达到最优。所以将网络入侵检测系统的参数确定为上值后，在 OpenNetVM 上，运行载有网络入侵检测系统的服务器。在客户端利用基于 DPDK 实现的 Pktgen 进行发包。客户端的发包速度达到 9692 Mbps。每个数据包的大小为 1518 B。测得服务器端的网络入侵检测系统的吞吐量达到 7.518 Gbps。由于异步实现服务器上的收包和模式匹配，所以在最开始运行时，服务器会丢部分包。但运行几秒后，系统就不会再丢包。这是因为系统启动会需要一定时间。导致部分包来不及处理，就要被丢掉。

在 OpenNetVM 上，发包速度达到 9692 Mbpss 时，利用 GPU 实现的网络入侵检测系统的性能约为 1.116 Gbps。所以整体利用 GPU 优化的性能达到 6.74 倍。性能提升非常明显。网络传输速度为近 10 Gbps。GPU 运行的模式匹配加速模块的吞吐量为 49.506 Gbps。最后的系统总性能为 7.518 Gbps。没有达到网络传输速度的原因，预估是当网络收到数据包后，在网卡和 DPDK 之间的数据传输过程还会损耗部分性能。

第五章　总结与展望

5.1　总结

本文是基于 OpenNetVM 平台，利用 Docker 容器、DPDK 和 GPU，实现高性能网络入侵检测系统。首先，通过对 Snort 规则的解析，完成规则模式配置。然后，利用 OpenNetVM 平台支持的 DPDK 技术，异步实现接收数据包，和对数据包的多模式匹配。多模式匹配算法是在 GPU 上实现 Aho-Corasick 算法。最后，系统会将匹配结果返回给用户。通过实验结果显示，构建的高性能并行网络入侵检测系统的吞吐量大大提高，提升近 7 倍，接近网络数据包的传输速度。

5.2　展望

利用 GPU 实现的匹配算法优化，还有提升性能的空间。一是目前 Aho-Corasick 算法构造的二维数组占用空间非常大，可以对数组压缩，进而减少从 CPU 向 GPU 拷贝数据的时间；二是 Aho-Corasick 是利用 GPU 的并行计算特点，可以通过调整传入 GPU 的一维数据的元素存储顺序，提高模式匹配的性能。总而言之，还有很多优化工作等待我们去完成。

参考文献

[1] What is NFV – Network Functions Virtualization – Definition? https://www.sdxcentral.com/nfv/resources/whats-network-functions-virtualization-nfv/.

[2] Network function virtualization. https://en.wikipedia.org/wiki/Network_function_virtualization.

[3] Docker 到底是什么？为什么它这么火！. http://cloud.51cto.com/art/201410/453718.htm.

[4] Docker：具备一致性的自动化软件部署. http://www.infoq.com/cn/news/2013/04/Docker.

[5] Docker 系列. http://tech.meituan.com/docker_introduction.html.

[6] 高性能网关设备及服务实践. http://blog.csdn.net/zhaqiwen/article/details/42619227.

[7] Snort 中文手册. http://man.chinaunix.net/network/snort/Snortman.htm.

[8] Aho-Corasick 算法实现类. http://blog.csdn.net/betabin/article/details/7423945.

[9] DPDK API. http://dpdk.org/doc/api/.

[10] Boyer R S, Moore J S. A Fast String Searching Algorithm (ACM). 1977..

[11] G Vasiliadis M P. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID). 2008..

[12] Huang N F, Hung H W. A GPU-Based Multiple-Pattern Matching Algorithm for Network Intrusion Detection Systems. In Proceedings of the 22nd International Conference on Advanced Information Networking and Applications Workshops. 2008..

[13] Vasiliadis G, Polychronakis M. MIDeA: A Multi-Parallel Intrusion Detection Architecture. In Proceedings of the ACM Conference on Computer and Communications Security (CCS). 2011..

[14] C Clark D S. A Hardware Platform for Network Intrusion Detection Patterns (FPL). 2004..