Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

# Estudo de Viabilidade de Sistemas de Detecção e Intrusão em Redes baseados em GPUs

Lucas Polonio Ribeirinho
Tércio Cassiano Silva

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. João José Costa Gondim

Coorientadora
Prof.ª Alba Cristina M. de Melo

Brasília
2012

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Marcus Vinícius Lamar

Banca examinadora composta por:

Prof. João José Costa Gondim (Orientador) — CIC/UnB
Prof.ª Alba Cristina M. de Melo — CIC/UnB
Prof. Laerte Peotta de Melo — ENE/UnB

**CIP — Catalogação Internacional na Publicação**

Ribeirinho, Lucas Polonio; Silva, Tércio Cassiano.

Estudo de Viabilidade de Sistemas de Detecção e Intrusão em Redes baseados em GPUs / Lucas Polonio Ribeirinho, Tércio Cassiano Silva. Brasília : UnB, 2012.
129 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2012.

1. Detecção, 2. Intrusão, 3. Prevenção, 4. Redes, 5. GPU, 6. Snort

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

# Estudo de Viabilidade de Sistemas de Detecção e Intrusão em Redes baseados em GPUs

Lucas Polonio Ribeirinho
Tércio Cassiano Silva

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. João José Costa Gondim (Orientador)
CIC/UnB

Prof.ª Alba Cristina M. de Melo
CIC/UnB

Prof. Laerte Peotta de Melo
ENE/UnB

Prof. Marcus Vinícius Lamar
Coordenador do Bacharelado em Ciência da Computação

Brasília, 12 de dezembro de 2012

# Dedicatória

Dedico esta monografia à minha família, especialmente meus pais, Sueli e Romeu, e minha irmã, Thais. Dedico também aos meus amigos e especialmente à minha namorada, Kamila. Sem o apoio de vocês ao longo da vida e nos momentos mais importantes, esse projeto teria sido muito mais difícil de se concretizar.

**Lucas**

Gostaria de dedicar esse projeto a minha família, principalmente ao meu avô que faleceu no início do ano, pois foi ele quem me ensinou que o importante na vida é fazer as coisas que a gente ama. Outra pessoa que merece muito essa dedicatória é a minha mãe, que sofreu de uma doença muito grave durante o ano e nunca deixou de me apoiar a fazer o que eu precisava fazer, abdicando dos problemas dela para se preocupar com os meus.

**Tércio**

# Agradecimentos

Gostaríamos de agradecer aos membros da banca, os professores João José Costa Gondim, Alba Cristina M. de Melo e Laerte Peotta de Melo pelas considerações realizadas durante a defesa desse trabalho.
Além disso, gostaríamos de fazer um agradecimento a professora Alba, que aceitou ser nossa co-orientadora e que nos ajudou a resolver os maiores problemas da arquitetura do projeto.
Agradecemos especialmente aos nossos amigos e familiares, que sempre nos apoiaram e nos ajudaram a seguir em frente.

# Resumo

Sistemas de Detecção e Prevenção de Intrusão em Redes (IDPS) são importantes ferramentas para a análise do tráfego que circula por redes de computadores. O Snort é um dos IDPS mais utilizados mundialmente e considerado uma das melhores ferramentas disponíveis. A performance é um dos pontos principais de um IDPS, e no caso do Snort, a operação de maior custo computacional é a comparação dos pacotes da rede com milhares de *patterns* presentes em suas regras, existentes para detectar padrões conhecidos de ataques a computadores. Visto que cada pacote é comparado a milhares de *patterns*, uma análise cuidadosa mostrou que este problema poderia ser adaptado para execução em GPUs, em grande parte por sua arquitetura SIMD (*Single Instruction Multiple Data*). Para minimizar o *overhead* de transferência dos pacotes da CPU para a GPU, foi necessário acumular pacotes e enviar um buffer de pacotes para a GPU. O Snort, porém, é uma plataforma que não foi pensado para execução paralela. Para resolver estes problemas, e com a premissa da não modificação da arquitetura interna do Snort, foi criada uma arquitetura para execução de vários processos de Snort simultâneos, acúmulo de pacotes em um buffer e uso de GPU para o *pattern matching*. Os resultados obtidos com esta arquitetura não foram melhores do que a execução de um Snort não-modificado. Por outro lado, sugere-se ser possível excelentes ganhos de performance, de mais de 433%, caso a arquitetura do Snort seja modificada ou em outro IDPS que seja favorável ao acúmulo de um buffer de pacotes para *matching*.

**Palavras-chave:** Detecção, Intrusão, Prevenção, Redes, GPU, Snort

# Abstract

Intrusion Prevention and Detection Systems (IDPS) are very important tools for the analysis of network traffic across computers. Snort is one of the most famous and useful IDPS wordwide and is considered to be one of the best ones available. Performance is the key factor of an IDPS and, especially in Snort, the most computationally intensive operation is the matching of all the network packets against thousands of patterns present in its rules, responsible for detecting known patterns of malicious traffic. Based on the fact that every packet is compared to thousands of patterns, it was realized that this problem could be adpated for GPU execution, due to GPU's SIMD (Single Instruction Multiple Data) architecture. To minimize the overhead caused by transfering packets from CPU's to GPU's memory, it was needed to store packets and send whole buffers to GPU in only one transfer. However, Snort's arquitecture was not planned for parallel execution, so that it could acquire this buffer. To solve these problems, and considering we would not alter Snort's core architecture, it was created an architecture for executing multiple Snort processes simultaneously, for capturing packets ready for matching in a buffer and, finally, for using the GPU to perform the pattern matching. The results, achieved using this architecture, were not better than the standard Snort. Still, it was showed there could be a possibility of excelent performance boosts (over 433%) if Snort's core architecture was modified or whether it was used another IDPS whose architecture suits better the creation of the packet buffer for matching.

**Keywords:** Detection, Intrusion, Prevention, Networks, GPU, Snort

# Sumário


**1 Introdução** **1**
1.1 Problemas . . . 2
1.2 Objetivos . . . 2
1.3 Objetivos Especifícos . . . 3
1.4 Justificativa . . . 3
1.5 Organização do Documento . . . 3

**2 Sistemas de Detecção e Prevenção de Intrusão em Redes** **5**
2.1 Princípios . . . 6
2.2 Aplicação e uso de Sistemas de Detecção de Intrusão . . . 6
2.3 Funções de NIDPS . . . 7
2.4 Metodologias comuns de detecção . . . 8
2.5 Tipos de tecnologia . . . 9
2.6 Componentes e arquitetura . . . 9
2.6.1 Componentes típicos . . . 9
2.6.2 Arquitetura de rede . . . 10
2.7 Capacidade de detecção . . . 10
2.8 Capacidade de Prevenção . . . 11
2.8.1 Gerenciamento de prevenção . . . 12
2.9 Implantação de NIDPS . . . 15
2.9.1 Firewall de borda . . . 15
2.9.2 Na frente dos firewalls de borda . . . 16
2.9.3 Junto dos maiores *backbones* da rede . . . 16
2.9.4 Em sub-redes críticas . . . 17

**3 Unidades de Processamento Gráfico** **18**
3.1 Definição e Histórico . . . 18
3.2 GPUs da ATI/AMD . . . 19
3.3 GPUs da NVIDIA . . . 19
3.3.1 Modelo de programação . . . 20
3.3.2 Acesso a memória . . . 21
3.4 Implementação do Hardware . . . 22
3.5 Modelo de Execução . . . 23
3.6 Dispositivos CUDA . . . 24
3.7 Performance . . . 25
3.7.1 Controle de fluxo de execução e *Branch Divergence* . . . 25

3.7.2 Transferência de dados entre *host* e dispositivo . . . . . . . . . . . . 26
3.8 API - *Application Programming Interface* . . . . . . . . . . . . . . . . . . . 27

**4 Snort: Sistema Detecção e Prevenção de Intrusão de Código Livre 30**
4.1 Arquitetura Lógica do Snort . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.1.1 Aquisição de pacotes . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.1.2 Decodificação de pacotes . . . . . . . . . . . . . . . . . . . . . . . . 32
4.1.3 Preprocessadores . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.1.4 Etapa de detecção . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
4.1.5 Sistema de *log* e alertas . . . . . . . . . . . . . . . . . . . . . . . . 34
4.1.6 Módulos de saída . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.2 As regras do Snort e o uso de patterns para detecção de ataques . . . . . . 34

**5 Estratégia e Descrição da Implementação 36**
5.1 Tarefas de detecção de intrusão adequeadas à execução em GPU . . . . . . 36
5.1.1 Detecção baseada em assinaturas . . . . . . . . . . . . . . . . . . . 36
5.1.2 Detecção baseada em anomalias . . . . . . . . . . . . . . . . . . . . 37
5.1.3 Detecção baseada na análise dos estados do protocolo . . . . . . . . 37
5.2 As tarefas do Snort que são adequadas a execução em GPU . . . . . . . . . 37
5.3 As tarefas do Snort escolhidas para a execução em GPU . . . . . . . . . . 37
5.4 Descrição da arquitetura da solução proposta . . . . . . . . . . . . . . . . . 37
5.4.1 Particularidades das regras do Snort consideradas na Implementação 39
5.5 A estratégia de Implementação . . . . . . . . . . . . . . . . . . . . . . . . 40
5.5.1 Modificações no Snort . . . . . . . . . . . . . . . . . . . . . . . . . 40
5.5.2 Criação do *buffer* de Pacotes . . . . . . . . . . . . . . . . . . . . . . 41
5.5.3 Criação do Módulo da GPU . . . . . . . . . . . . . . . . . . . . . . 41

**6 Resultados Obtidos 43**
6.1 Descrição dos testes realizados . . . . . . . . . . . . . . . . . . . . . . . . 43
6.2 Descrição do ambiente . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
6.3 Descrição dos resultados . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
6.4 Discussão dos resultados . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

**7 Conclusão 53**
7.1 Trabalhos Futuros . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

**Referências 55**

# Lista de Figuras


2.1 Localizações mais frequentes de NIDPS em redes [SM01]. . . . . . . . . . . . 16

3.1 Arquitetura do *Stream Processor* ATI [ATI10] . . . . . . . . . . . . . . . . . . 19
3.2 Topologia da organização de um programa na GPU [NVI07] . . . . . . . . . 21
3.3 Organização do programa com visualização das memórias [NVI07] . . . . . 22
3.4 Um conjunto de multi-processadores SIMD com memória compartilhada [NVI07]. 24
3.5 No primeiro gráfico (*latency up*), latência na transferência de um *host* com diferentes *chipsets* para um dispositivo CUDA (NVIDIA Geforce GTX 280), variando-se a quantidade de dados transferida. No segundo gráfico (*latency down*), a mesma situação, mas na direção oposta [Hov08]. . . . . 27

4.1 Estrutura básica de funcionamento do Snort. . . . . . . . . . . . . . . . . . . . 31
4.2 Estrutura lógica de funcionamento do Snort. . . . . . . . . . . . . . . . . . . . 31

5.1 Arquitetura proposta para o Snort como módulo de comparação paralela de padrões de assinatura. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

6.1 Gráfico de Desempenho . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

# Lista de Tabelas

3.1 Código C para adiçao de matrizes . . . . . . . . . . . . . . . . . . . . . . . . 29
3.2 O código CUDA para a adiçao de matrizes . . . . . . . . . . . . . . . . . . . 29

6.1 Tempo de criação do lock, da espera por variável de condição e cópia do pacote do Snort (e seus metadados) para a sua respectiva área de memória compartilhada . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
6.2 Tempo de espera no *buffer* de pacotes . . . . . . . . . . . . . . . . . . . . . . . . 47
6.3 Tempo de cópia dos pacotes para formar o *buffer* que será enviado para a GPU . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
6.4 Tempo de transferência do *buffer* de pacotes da memória da CPU para a memória global da GPU . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
6.5 Tempo de execução do kernel na GPU com o loop de busca (*pattern matching*) 49
6.6 Tempo de percorrimento dos resultados escritos pela GPU na memória da CPU (via DMA) e escrita dos mesmos na memória compartilhada. . . . . . 50
6.7 Tempo de liberação do lock (*Mutex*) e envio de sinal para a variável de condição . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
6.8 Tempo de percorrimento dos resultados na área de memória compartilhada e *callback* das detecções no snort . . . . . . . . . . . . . . . . . . . . . . . . 51
6.9 Tempos totais gastos na execução da nossa modificação . . . . . . . . . . . 51
6.10 Snorts não-modificados, com PF_RING . . . . . . . . . . . . . . . . . . . . . 52
6.11 Comparativo do nosso com relação ao original. Os resultados se referem à porcentagem de redução de performance. . . . . . . . . . . . . . . . . . . . . . 52
6.12 Resultados das medidas realizadas considerando somente o módulo de GPU 52

# Capítulo 1

# Introdução

Detecção de intrusão é o processo de monitorar os eventos que ocorrem em um sistema computacional e analisá-los procurando por sinais de intrusão ou ações maliciosas – que podem ser definidos como as tentativas de comprometer confidencialidade, integridade, disponibilidade ou transpassar os mecanismos de segurança de um computador ou rede de computadores [SM01].

Prevenção de intrusão é o processo em que, além da detecção de intrusão, também será provida a tentativa de interrupção de possíveis incidentes detectados [SM07]. Um sistema de detecção e prevenção de intrusão em redes (NIDPS, do inglês *Network Intrusion Detection and Prevention System*) é um software que automatiza os processos de detecção e de prevenção de intrusão em redes [VSP+08].

A prevenção de intrusão em redes está normalmente associada a medidas de contenção que podem ser tomadas em conjunto a outros softwares ou pessoas. Por exemplo, quando uma ação maliciosa é detectada vindo de um certo IP, o NIDPS pode atualizar a lista de IPs não autorizados dos firewalls que monitoram essa rede, para que este o acesso a rede feito por este computador seja bloqueado. O NIDPS pode até acionar sistemas que tomam medidas mais drásticas, como tornar indisponíveis serviços públicos que possam ter sido comprometidos por um ataque [SM07].

Atualmente, percebe-se uma preocupação crescente com a segurança de sistemas computacionais. Enquanto a velocidade e a eficiência em todos os processos de negócios significam uma vantagem competitiva, a falta de segurança nos meios que habilitam a velocidade e a eficiência pode resultar em grandes prejuízos e falta de novas oportunidades de negócios [NdG07]. Um sistema de detecção e prevenção de intrusão em redes atua como uma medida de prevenção a ataques de negação de serviço, que podem comprometer a produtividade de um sistema, assim como atua, também, detectando ataques que possam comprometer o próprio sistema ou dados que este contém.

A análise de pacotes que trafegam na rede é a base dos sistemas de detecção e prevenção em redes mais utilizados atualmente. Essa análise ocorre, normalmente, pela inspeção dos pacotes aplicando-se regras (expressões regulares ou padrões de bits) sobre o seu conteúdo. Tais regras são conhecidas como assinaturas, e representam ameaças ou indícios de comportamentos já conhecidos e que podem representar um ataque em suas diversas fases, desde *probes* ao ataque em si [SM07]. Assim sendo, cada pacote que passa pelo NIDPS é comparado a um conjunto de regras previamente configuradas. Esses conjuntos tomam grandes proporções nos NIDPS atuais.

Grandes corporações enfrentam problemas para analisar o tráfego que passa por sua rede, visto que esse é um processo que consome muitos recursos computacionais para que se consiga analisar todo o tráfego em tempo real ou em tempo hábil para que ações de contenção sejam tomadas. Consequentemente, aumentar a performance desses sistemas é uma ótima maneira para redução de custos.

O processo de comparação de conteúdos de pacotes a várias regras determinadas se dá por algoritmos de comparação de cadeias de caracteres (*matching*) e por reconhecimento de strings em expressões regulares (*regular expression matching*). Esse trabalho é responsável, em média, por mais de 70% do tempo gasto na análise de um pacote [CGLM04], [AAM04].

Dada que essa etapa do processamento de pacotes por um NIDPS representa grandes porções do tempo utilizado, observa-se que um aumento de performance nela representa uma diminuição substancial no tempo total de processamento. Para tal, pode-se fazer uso de algoritmos paralelos que acelerem a comparação de *strings* e reconhecimento de *strings* em padrões representados por expressões regulares. Essa paralelização é especialmente útil visto que um único pacote é comparado a inúmeras regras.

Observa-se, então, que o uso de GPUs (do inglês, *Graphics Processing Unit*, processadores localizados em placas de vídeos) para propósitos gerais vem aumentando consideravelmente, no que vem sendo chamado de GPGPU (do inglês, *General Purpose Graphics Processing Unit*). Placas de vídeo com suporte a GPGPU são produzidas atualmente em larga escala, podem ser encontradas facilmente no mercado e possuem custo baixo, comparando-se o poder de processamento que elas fornecem (devido ao alto paralelismo) aos processadores comuns.

O NIDPS mais utilizado atualmente, SNORT, não tira proveito do paralelismo possível nos computadores atuais, sendo completamente sequencial na verificação das regras e em toda a sua análise do tráfego [BBEN07].

## 1.1 Problemas

- Como melhorar o desempenho da análise de pacotes do SNORT?
- Como melhorar o desempenho da análise de pacotes no SNORT utilizando uma solução baseada em GPU?

## 1.2 Objetivos

Dadas as considerações gerais sobre os IDPS e as GPGPUs, decidiu-se seguir por uma abordagem prática. Os trabalhos correlatos nessa área são raros e os existentes estão sobre patentes sobre o código fonte e dessa maneira dificultam maiores aprimoramentos nessas implementações.

Dessa forma, foi decidito adaptar o Snort existente para que uma parte dele funcionasse utilizando o processamento das GPUs. A funcionalidade escolhida foi a comparação dos *patterns* contidos no conjunto de regras do Snort e dessa maneira foi possível propor arquitetura para a utilização de GPUs. Vale ressaltar, que o objetivo nunca foi modificar o Snort existente, uma vez que caso essa fosse a abordagem, seria possível fazer praticamente

qualquer alteração no seu código para um módulo paralelo, mas sim adaptá-lo a execução paralela.

## 1.3 Objetivos Especifícos

São objetivos especifícos do trabalho:

- Adaptar o algoritmo de comparação de *patterns* do SNORT para execução em GPU;
- Medir e avaliar o desempenho de ambas as abordagens de comparação dos *patterns*;
- Avaliar se a solução proposta foi satisfatória ou não e apontar os pontos positivos e negativos'.

## 1.4 Justificativa

Atualmente o aumento do tamanho dos links de internet bem como o tráfego total de pacotes numa rede de computadores é crescente. Isso ocasiona o troca dos componentes de segurança constantemente, uma vez que o *throughput* é a medida de compra da maioria desses equipamentos e ele é objeto de aumento constante. Assim temos que:

- Quanto maior seja o aproveitamento da solução de análise de pacotes na rede, maior será o tempo de duração da solução, gerando assim uma economia significativa no orçamento das cooporações que contratam esse tipo de serviço;
- Além disso, 75% do tempo gasto na execução do SNORT é na análise de pacotes, que inclui a comparação de *patterns*, e assim quanto melhor for o aproveitamento da execução da solução, melhor será aproveitamento dela.

## 1.5 Organização do Documento

O documento está organizado de maneira que primeiramente oferece o referencial teórico para entender tanto o trabalho quanto o problema e onde são explicados os problemas, análise, resultados e conclusões do trabalho. São os capítulos os seguintes:

- Capítulo 2 - Sistemas de Detecção e Prevenção de Intrusão em Redes;
  Capítulo que revisa o referencial teórico dos IDPS;
- Capítulo 3 - Unidades de Processamento Gráfico;
  Capítulo no qual se explicam os principais conceitos das GPUs modernas;
- Capítulo 4 - Snort: Sistema de Detecção e Prevenção de Intrusão de Código Livre
  Capítulo onde são expostas as principais funcionalidades e usos do Snort, bem como aspectos de implementação;
- Capítulo 5 - Descrição do Trabalho Realizado
  Capítulo onde são explicadas as principais modificações no Snort, bem como as dificuldades encontradas no projeto;

- Capítulo 6 - Resultados Obtidos
  Capítulo em que são expostos os resultados obtidos e suas principais implicações;
- Capítulo 7 - Conclusão
  Fechamento do estudo e exposição dos trabalhos futuros.

O próximo capítulo revisa as teorias envolvidas nos IDPS atuais, a fim de fornecer os conceitos básicos do tema e facilitar o entendimento do trabalho como um todo.

# Capítulo 2

# Sistemas de Detecção e Prevenção de Intrusão em Redes

Desde os princípios da Internet, tentativas de comprometer os sistemas computacionais que fazem uso dela são frequentes. Nesse contexto surgiram os *firewalls* que são coleções de componentes interpostos entre duas redes de computadores que filtram o tráfego de acordo com alguma política de segurança [IKBS00].

*Firewalls* [Goe11] limitam o acesso entre redes com o uso de regras simples, com o objetivo de bloquear ou permitir determinado tráfego. Estas regras normalmente envolvem somente análise do cabeçalho de pacotes e filtram somente pacotes que trafegam de fora para dentro da rede.

Apesar da grande disseminação da utilização de *firewalls*, estes possuíam suas limitações, a exemplo das citadas anteriormente, além de que mostravam sua fragilidade quando um ataque era feito de uma maneira não prevista nas políticas. A partir disso, considerou-se criar um sistema novo, que fosse executado em conjunto com os *firewalls* existentes.

Surgiram, então, os Sistemas de Detecção de Intrusão em redes, conhecidos por NIDS (*Network Intrusion Detection Systems*). Tais sistemas tinham a função de analisar com maior profundidade o tráfego dentro de uma rede. Funcionam analisando os pacotes que passam nos dois sentidos pela borda da rede. O NIDS, ao detectar um evento malicioso ou atípico, gera um alerta para o administrador da rede.

Mais tarde, surgem os Sistemas de Detecção e Prevenção de Intrusão em redes, os NIDPS (*Network Instrusion Detection and Prevention Systems*), que permitem que sejam tomadas ações em resposta aos incidentes (além do alerta já gerado). Também aumentaram sua capacidade de detectar ataques, detectando ataques não conhecidos previamente, devido ao seu conhecimento sobre comportamentos genéricos de ataques.

Atualmente, os NIDPS são usados em conjunto aos *firewalls* para a proteção de redes de computadores. Existem, também, *Firewalls* UTM(*Unified Threat Management*), que, segundo o IDC (*International Data Corporation*), devem conter funcionalidades de *firewall*, sistemas de detecção e prevenção de intrusão, anti-vírus e suporte a Redes Privadas Virtuais (VPN, *Virtual Private Network*). Os grandes fabricantes de firewall tem se voltado para *Firewalls* UTM, visto que, por serem uma solução unificada, fornecem melhor agrupamento, maior facilidade de uso em conjunto e menor custo para os consumidores finais.

## 2.1 Princípios

Para entender o funcionamento dos NIDPS, é necessário compreender mais a fundo tanto a diferença entre detecção e prevenção de intrusão, assim como processos que utilizam detecção e prevenção juntas:

- **Detecção de intrusão** é o processo de monitorar os eventos que ocorrem num sistema computacional e analisá-los procurando por sinais de intrusão ou ações maliciosas, que podem ser definidos como as tentativas de comprometer confidencialidade, integridade, disponibilidade ou transpassar os mecanismos de segurança de um computador ou rede de computadores [SM01], [Wu09];
- **Prevenção de intrusão** é o processo em que, além da detecção de intrusão, também será provida a tentativa de interrupção de possíveis incidentes detectados [SM07];
- Um sistema de **detecção e prevenção de intrusão** em redes, NIDPS, é um software que automatiza os processos de detecção e de prevenção de intrusão em redes. [VSP+08].

Vale ressaltar que os sistemas de detecção e prevenção possuem muitas funções equivalentes, sendo possível que um sistema de prevenção atue como um de detecção por simples desativação das funções de prevenção [SM07].

Apesar das capacidades dos NIDPS em lidar com incidentes potencialmente maliciosos, muitos deles e muitas vezes esses incidentes são apenas erros de digitação de *url's* ou tentativas de conexão a um outro sistema ao qual o usuário não tem acesso por engano.

## 2.2 Aplicação e uso de Sistemas de Detecção de Intrusão

NIDPS têm como foco principal identificar possíveis incidentes potencialmente maliciosos. Um exemplo é o fato de um NIDPS ser capaz de identificar uma possível violação e comprometimento do sistema explorando uma vulnerabilidade [SM07]. A prevenção de intrusão em redes está normalmente associada a medidas de contenção que podem ser tomadas em conjunto a outros softwares ou pessoas. Nesse processo normalmente o sistema pode ser configurado para funcionar de duas maneiras:

- Reportar que uma possível violação das regras do sistema computacional ocorreu ao administrador da rede;
- Armazenar as informações da violação e reportar a *engine* que lida com os ataques.

Além disso alguns deles podem ser configurados com um conjunto de regras similar a um *firewall*, permitindo assim que um NIDPS analise o tráfego da rede com base na política adotada [SM07].

Por outro lado,um importante mecanismo que pode ser provido por esses sistemas é o de analisar uma atividade suspeita, que normalmente precede um ataque real, e tomar as medidas necessárias mesmo antes dele acontecer [SM07].

## 2.3 Funções de NIDPS

Atualmente existem muitos tipos de NIDPS disponíveis, caracterizados por diferentes maneiras de análise e monitoramento. Cada um desses sistemas tem suas vantagens e desvantagens cabendo ao usuário decidir qual deles utilizar [SM01]. Há porém, outras tecnologias que podem ser diferenciadas basicamente pelos tipos de eventos que estão aptos a reconhecer, não apenas pelas metodologias aplicadas [SM07].

Independentemente dos métodos e tecnologias aplicadas, geralmente os NIDPS fornecem as seguintes funções:

- **Coletar informações relacionadas aos eventos monitorados**;
- **Notificar os responsáveis pela segurança da rede sobre a relevância dos eventos observados**;
- **Geração de relatórios sobre os incidentes**.

Alguns desses sistemas ainda podem tomar alguma outra medida para contenção do ataque, como alterar a política de segurança do sistema e/ou armazenar informação extra sobre o incidente em questão. Essas atitudes tomadas são importantes, uma vez que o sistema desconheça aquela atividade, é possível identificar uma nova ameaça e possuir mais informações caso ela venha a ocorrer novamente [SM01][SM07].

É importante ressaltar que um simples detalhe diferencia um NIDS de um NIPS: um NIPS tem a capacidade de modificar seu funcionamento estando o sistema sobre ataque [SM07]. Uma variedade grande de técnicas de resposta são usadas e podem ser divididas nas seguintes categorias:

- **O NIPS age e contém o ataque com seus mecanismos**. Exemplos de como a contenção é feita são como os seguintes:
  - Termina a conexão com a rede ou a sessão do usuário por onde o ataque está sendo executado;
  - Bloqueia o acesso ao alvo(ou aos possíveis alvos) para o usuário, endereço IP ou outro atributo do atacante;
  - Bloqueia todos os acessos ao sistema, serviços, aplicação ou qualquer que seja o objetivo do atacante.
- O NIPS altera o nível de segurança do ambiente. O *IPS* pode reconfigurar outros serviços de segurança para interromper um ataque. Exemplos comuns de reprogramação de um dispositivo de rede(*firewalls*, roteadores, *switch*) para bloquear o acesso do atacante ao alvo e alterar um ***host-based*** *firewall* em um alvo para boquear possíveis ataques. Alguns deles ainda podem aplicar atualizações aos sistemas caso detectem que eles estão vulneráveis;
- O NIPS remove as partes maliciosas contidas no pacote. Essas remoções fazem com que o ataque não tenha eficácia, tornando-o assim "benigno". Um simples exemplo é um NIPS removendo um arquivo anexado infectado de um email e liberar o restante dele para o destinatário. Um outro exemplo, mais complexo, é um NIPS que age como um servidor proxy e normaliza as requisições de entrada, ou seja, o proxy

reempacota os dados das requisições, descartando as informações de cabeçalho. Isso pode causar a certos ataques que parte deles sejam descartados como parte do processo de normalização.

Um das principais características de NIDPS é a incapacidade de prover uma completa acurácia sobre os incidentes identificados. Ao identificar uma ameaça que não é uma ataque real, temos um **falso positivo** e caso a ameaça seja real e o sistema não a identifique, teremos um **falso negativo** [SM01][SM07]. É muito importante ter conhecimento desse tipo de comportamento em um sistema de detecção e prevenção, uma vez que é hipotética a possibilidade de uma acurácia completa, mas pode-se minimizar ao máximo a frequência desses alertas.

## 2.4 Metodologias comuns de detecção

Existem vários métodos para detecção de incidentes em Sistemas de Detecção e Prevenção de Intrusão. Um NIDPS normalmente usa múltiplos métodos, sejam eles integrados ou de forma separada durante o processo de detecção. Segundo o NIST (*National Institute of Standards and Technology*), é possível enumerar e definir as três principais classes de metodologias de detecção [SM07]:

- **Detecção baseada em assinaturas** é uma metodologia de detecção baseada na análise de padrões que correspondem a ameaças já conhecidas. Seu funcionamento o torna bastante efetivo contra ameaças já conhecidas e que podem ser definidas por padrões determinísticos (*i.e.:* não mutáveis). Porém, quando as ameaças não são conhecidas ou então quando as ameaças usam de técnicas de evasão ou mutação, essa metodologia não é capaz de detectá-las.
- **Detecção baseada em anomalias** é um método que tem conhecimento do comportamento e uso do sistema em condições normais, sem atividades maliciosas ocorrendo. Com base nesse conhecimento, o NIDPS analisa os eventos que ocorrem e decide se atividades maliciosas estão sendo realizadas de acordo com o desvio de comportamento observado. Essa metodologia necessita de um maior cuidado para que se determine o perfil que será considerado como comportamento padrão. Um perfil inicial é obtido da análise do sistema durante um tempo estabelecido (período de treinamento) e pode ou não ser alterado por novas análises ou por alterações manuais realizadas pelo administrador do sistema. Um cuidado para que atividades maliciosas não sejam incluídas no perfil padrão é essencial e é, também, um dos maiores problemas dessa metodologia de detecção [SM07]. Ao contrário da detecção baseada em assinaturas, a detecção baseada em anomalias pode ser altamente efetiva contra ameaças desconhecidas, se o perfil for obtido adequadamente.
- **Análise de estados de protocolo** (*stateful protocol analysis*) é uma metodologia que analisa o uso de protocolos usados através do sistema. O NIDPS deve observar e manter estados do uso desses protocolos. Deve ter conhecimento, também, dos perfis e condições de uso normais do protocolo, definidas pelo seu criador ou o responsável por mantê-lo. A partir disso, a análise de estados de protocolo é capaz de detectar possíveis ações maliciosas, que se sustentam sobre falhas ou fraquezas

nos protocolos ou no sistema que o implementa, mesmo que essas ações não sejam conhecidas previamente. Normalmente, essa metodologia não é usada sozinha em NIDPS, mas sim em conjunto com uma ou mais metodologias diferentes [SM07].

## 2.5 Tipos de tecnologia

Será descrito uma classificação em quatro grupos de tecnologias NIDPS. Essa classificação é baseada no tipo de eventos que o sistema monitora e em seu mecanismo de funcionamento. Foi realizada no Guia sobre Sistemas de Detecção e Prevenção de intrusão, do NIST [SM07], e no *Information Assurance Tools Report – Intrusion Detection Systems. Sixth Edition.*, do IATAC [Wu09].

- **Baseado em redes**. Esse tipo de tecnologia monitora o tráfego em segmentos de rede e analisa atividades nela e em seus dispositivos, buscando identificar atividades suspeitas.
- **Baseado em redes sem fio**. Trabalha tentando detectar atividades suspeitas envolvendo os protocolos de comunicação sem fio.
- **Análise de comportamento de redes**. Avalia o comportamento de redes em busca de identificar comportamentos incomuns no tráfego. Detecta, por exemplo, escaneamento de portas ou ataques distribuídos de negação de serviço [SM01].
- ***Host-based***. O tipo de tecnologia *host-based* monitora as características específicas em um único dispositivo. Normalmente, se refere a um único computador ou dispositivo de rede [SM07]. Neste caso, o *NIDPS* analisa os eventos que ocorrem internamente ao dispositivo.

Nesse trabalho, o foco será em Sistemas de Detecção e Prevenção de Intrusão em redes (NIDPS), que são sistemas que consideram os três primeiros itens dessa lista.

## 2.6 Componentes e arquitetura

Ainda de acordo com o NIST, serão descritas as principais soluções de componentes de um NIDPS e ilustradas as arquiteturas de redes mais comuns para esses componentes.

### 2.6.1 Componentes típicos

Os componentes típicos de um NIDPS são citados aqui, e podem estar localizados em dispositivos distintos ou unidos em um mesmo dispositivo.

- **Sensores ou Agentes** são os dispositivos que monitoram e analisam as atividades. No contexto de NIDPS, normalmente o termo sensor é usado.
- **Servidores de gerenciamento** são dispositivos centralizados que recebem as informações capturadas pelos sensores ou agentes e as gerenciam. Podem analisar mais profundamente as informações, além de correlacionar informações de um mesmo grupo capturadas por diferentes sensores.

- **Servidores de banco de dados** são repositórios para informações de eventos detectados pelos sensores, agentes ou servidores de gerenciamento. São fontes para consulta do histórico do sistema e podem ser utilizados para extrair dados estatísticos que determinarão o perfil padrão a ser utilizado em detecções baseadas em anomalias, por exemplo.
- **Consoles** são programas que fornecem interface para os administradores do sistema. Através deles, é possível configurar sensores ou agentes, aplicar atualizações de software do NIDPS, além de acessar informações de monitoramento e análise.

### 2.6.2 Arquitetura de rede

É importante observar que esta seção só é válida para NIDPS. Para o caso de NIDPS que utilizam tecnologia *host-based* todos os componentes são, normalmente, localizados no próprio dispositivo. Pode-se citar duas arquiteturas que podem ser utilizadas para distribuição dos componentes.

Na primeira, os componentes de gerenciamento (servidores de gerenciamento, de banco de dados e consoles) estão localizados em uma rede distinta (rede de gerenciamento) e os sensores ou agentes têm acesso à rede analisada e à rede de gerenciamento, através de duas interfaces distintas que não repassam tráfego entre si. Nesse tipo de arquitetura, é possível destacar a proteção da rede de gerenciamento por vários aspectos: a rede se mantém sem congestionamento em casos de ataques distribuídos de negação de serviço na rede analisada e, além disso, essa rede está isolada fisicamente da rede analisada, impossibilitando que o NIDPS seja atacado ou acessado diretamente. Esse último ponto é um dos inconvenientes, já que para acessar os consoles, é necessário que o acesso seja feito por um computador separado da rede analisada, que normalmente só terá essa função. Nota-se, então, que essa solução possui um custo elevado.

Na segunda, todos os componentes de gerenciamento estão localizados na mesma rede que é alvo da análise do NIDPS. Esse tipo de arquitetura tem custo reduzido e trás uma maior conveniência para os administradores. Porém, está sujeita aos problemas de segurança que a primeira opção não oferece. Em uma tentativa de se minimizar a questão da rede de gerenciamento ser inacessível externamente, é possível que seja utilizada uma rede virtual para gerenciamento, dentro da rede de análise.

## 2.7 Capacidade de detecção

Quanto a capacidade de detecção, novamente os NIDPS se utilizam de uma combinação de diferentes técnicas. Os tipos de eventos que são detectados e a precisão da detecção variam dependendo da tecnologia utilizada. A capacidade de detecção está diretamente associada à capacidade de se customizar o NIDPS [SM07]. A maioria dos NIDPS requerem que seja feita customização para que se melhore a precisão e a efetividade da detecção. Normalmente, é possível que se customize o sistema com vários tipos de parâmetros, alguns exemplos podem ser:

- **Limiar**. Limiar é um valor que determina o limite entre comportamento normal e anormal. O limiar normalmente especifica o nível máximo (ou mínimo) de aceitação.

São normalmente utilizados para detecção baseada em anomalias e em análise de estados de protocolo. Como exemplo, existem limiares de 10 conexões simultâneas a um servidor.

- ***Blacklists* e *Whitelists***. *blacklists* são listas de entidades que foram previamente associadas com atividades maliciosas. Essas listas podem incluir, por exemplo, identificadores de hosts, números de portas TCP ou UDP, aplicações etc. As *blacklists* podem ser populadas dinamicamente e temporariamente, caso seja detectado atividade maliciosa vinda de um certo IP, por exemplo. O NIDPS, então, ignora todas as atividades detectadas, desde que estas não estejam nesta lista. Por outro lado, *whitelists* são associadas a atividades conhecidamente normais e não-maliciosas. O NIDPS, portanto, irá gerar alertas para qualquer atividade que ele detecte e que não esteja nesta lista.

- **Configurações de alerta** são configurações que o administrador pode realizar que influenciam o modo como o NIDPS irá gerar alertas. Entre elas, podemos citar a atividade de ligar e desligar os alertas (por grupos de atividades, ou gerais), alterar um nível de segurança, especificar quais capacidade de prevenção devem ser utilizadas ou especificar como os alertas devem ser realizados (e-mail, celular, etc) [Wu09].

- **Visualização e edição de código**. A maioria dos NIDPS permitem a edição de códigos de assinaturas, normalmente com o uso de expressões regulares que ajudam a descrever padrões já conhecidos de atividades maliciosas. Outros, no entanto, possibilitam ao administrador adicionar código de detecção ou de análise, como, por exemplo, programas que realizam análise de estados de protocolos [SM07].

É importante que o administrador revise os parâmetros de customização do sistema. Regras, *whitelists* ou *blacklists*, por exemplo, devem ser checadas para verificar se estão adequadas e se ainda são necessárias, para que recursos do sistema não sejam utilizados sem necessidade. Mudanças no ambiente ou no comportamento devido a presença de novas tecnologias devem ser observadas com frequência pelo administrador.

Já quanto a capacidades de prevenção, o administrador normalmente pode customizar a capacidade de prevenção para cada tipo de alerta. Isso inclui habilitar ou desabilitar a prevenção, e indicar qual prevenção deve ser usada. Alguns NIDPS fornecem a capacidade de simular que uma prevenção seja tomada, o que na verdade só registra em um *log* que uma prevenção deveria ter sido tomada. Essa funcionalidade pode ser útil para que as configurações sejam ajustadas com maior precisão.

## 2.8 Capacidade de Prevenção

Muitos NIDPS oferecem múltiplas capacidades de prevenção; as capacidades específicas variam de acordo com topologia do NIDPS. Os NIDPS normalmente permitem que os administradores especifiquem as configurações da capacidade de prevenção para cada tipo diferente de alerta. Geralmente, são fornecidas as possibilidades de habilitação e desabilitação, assim como especificar que capacidade de prevenção deverá ser utilizada. Alguns sensores de NIDPS tem um modo de aprendizado ou simulação que oprime todas

as atividades de prevenção afim de apenas informar quando uma ação de prevenção seria tomada. Essas estratégias são utilizadas para que o administrador possa avaliar a aplicação de novas regras de prevenção, reduzindo o risco de alertar falsos negativos. [SM07]

### 2.8.1 Gerenciamento de prevenção

De acordo com o NIST, os NIDPS disponibilizados largamente utilizam capacidades similares de gerenciamento de prevenção. Os aspectos mais importantes de gerenciamento - implementação, operação e manutenção - e proverá recomendações para efetividade e eficiência. [SM07]

#### Implementação

Após decidir por um NIDPS, o primeiro passo é projetar a arquitetura do sistema, testar os componentes utilizados e garantir a segurança deles.

Alguns dos tópicos mais importantes da implementação são norteadas por:

- Onde os sensores ou agentes devem estar presentes;
- Quão resiliente o sistema deve ser, ou seja, como ele irá se recuperar de falhas;
- Em quais componentes da rede o NIDPS poderá agir;
- Em quais outros sistemas de segurança o NIDPS poderá agir. [SM07]

Após projetado o NIDPS, existem pré-requisitos para testar sua utilização, uma vez que a má utilização desse ambiente irá gerar um grande transtorno para usuário da rede. Isso se deve ao fato de que o sistema deve ser customizado para funcionar de acordo com as políticas definidas e sua instalação padrão possivelmente não atenderá a esse objetivos. Além disso, não se deve ativar o sistema como um todo em ambiente de teste, pois somente é necessário uma visualização de como ele irá se comportar. [SM07]

#### Operação e instalação

Na instalação dos componentes do NIDPS é altamente aconselhável que primeiro seja montado em um ambiente de testes, ao invés de fazê-lo em ambiente de produção. Isso se deve ao fato de que essa instalação pode gerar inconsistências na rede de computadores de produção. Após um período de testes, se iniciará a migração do sistema para produção. Mas ainda assim é importante que os componentes sejam transferidos de maneira gradual. Inicialmente nem todos os sensores ou agentes devem ser ativados, assim como as capacidades de prevenção. O sistema necessita de uma customização e identificação de como a rede funciona para diminuir o número de falsos positivos e desempenhar sua função. Além disso, os administradores da rede não conseguiriam configurar adequadamente o NIDPS caso ele fosse ativado com todo seu potencial, justamente por causa dos falsos positivos que não os permitiriam customizá-lo para um ambiente real. Isso justifica a presença de um ambiente de teste, pois boa parte dos falsos positivos poderão ser de identificados e ajudariam na customização de um ambiente real e ainda podem evidenciar os problemas de escalabilidade do sistema para a produção [SM07].

Os componentes do NIDPS podem ser baseados em *hardware* ou *software*. Caso sejam baseados em *hardware* normalmente somente é necessária a instalação dos componentes, como cabos e conexão a um fonte de energia, e a atualização do *software* que os manipulam, como por exemplo provê update de assinatura. Por outro lado, os componentes baseados em *software* costumam levar mais tempo para a instalação. Nesse caso o problema depende da arquitetura de *hardware* onde o NIDPS será instalado e suas principais considerações são: adquirir ou garantir que o *hardware* é robusto o suficiente para executar o sistema. Após esse passo inicial deve ser instalado um sistema operacional compatível com o *software* do NIDPS e o sistema como um todo deve ser reforçado para garantir robustez. O reforço do sistema é feito por meio das atualizações de todos os componentes de *software* que fazem parte do sistema. Além de tudo isso é necessária a configuração dos componentes instalados, assim como nos baseados em *hardware* [SM07].

Após a instalação e configuração dos componentes é necessário configurar suas capacidades de detecção e prevenção considerando os objetivos do sistema sendo disponibilizado. Apesar de não ser obrigatória, a não configuração dos componentes pode diminuir drasticamente a capacidades dos dispositivos, ficando praticamente restrito a ataques facilmente identificáveis [SM07].

### Garantindo a segurança dos componentes

A segurança dos componentes é um ponto crucial para um funcionamento desejável de um NIDPS. Eles são constantemente alvos dos ataques e caso haja um comprometimento o sistema pode ser tornar sem utilidade para detecções futuras. Além disso, alguns componentes da arquitetura contém informações sigilosas, como por exemplo vulnerabilidades conhecidas, que podem ser de grande utilidade para ataques futuros [SM07].

Por outro lado, os administradores do sistema deverão garantir que todos os componentes do sistema estão atualizados em suas versões mais recentes e também a segurança dos componentes do sistema. Algumas recomendações especificas para essa segurança são apresentadas:

- A designação dos privilégios, bem como a tipo de conta dos usuários, dos usuários deve ser bem definida. Isso evita que durante algum ataque sejam adquiridos privilégios que possam danificar o sistema, mesmo que aquele usuário não saiba que isso estava disponível para ele; [SM07]

- Os dispositivos de rede cujas funções são de filtragem de tráfego na rede, como *firewalls* e roteadores, devem ser configurados pelos administradores para que apenas os que realmente necessitam de acesso o tenham; [SM07]

- Os administradores também devem garantir que todas as comunicações do gerenciamento do NIDPS estão adequadamente protegidas [SM07]. Os modos mais comuns de se garantir isso são por meio de divisões físicas ou lógicas na comunicação, ou através da criptografia dela. Caso a solução escolhida seja pela criptografia devemos escolher qual algoritmo utilizar. A solução para criptografia normalmente é

pela utilização de TSL(*Transport Layer Security*); nas demais que não fornecem criptografia adequada, devemos considerar algum tipo de tunelamento dos pacotes, como uma VPN(*Virtual Private Network*). [SM07]

### Operação e Manutenção

Praticamente todos os sistemas de detecção e prevenção são modelados para operação e manutenção através de uma interface gráfica com o usuário - GUI(*Graphical User Interface*), também conhecido com *console*. A função básica do *console* é fornecer ferramentas aos administradores para configurar e atualizar os sensores e gerenciar os servidores, assim como monitorar o que está ocorrendo na rede, como pacotes descartados ou agente de falhas [SM07].

Outra função dos administradores da rede é gerenciar as contas de usuários, customizar relatórios bem como outras funções disponibilizadas pelo console. Os usuários também tem acesso a algumas funções através do *console* incluindo por exemplo monitoração e análise de dados do NIDPS e geração de relatórios. Uma prática comum é dividir as contas de acesso ao NIDPS entre administrador e usuário, mas caso seja necessário podem ser concedidas algumas funções de administrador aos usuários, mas somente as funções realmente necessárias. Essa operação é conhecida como *grant* de privilégios. A visualização dessa organização de privilégios pode ser vista nas opções disponibilizadas pelo *console* [SM07].

Outros produtos disponibilizam um acesso mais granularizado que especifica detalhes de quais agentes ou sensores um determinado usuário tem poder de alterar ou remover as configurações. Nesse caso a divisão se torna mais lógica, mas por outro lado temos um sistema com uma hierarquia bem definida de operação e manutenção [SM07].

Uma outra categoria de NIDPS fornece uma interface de linha de comando - CLI(*Command-line Interfaces*. Diferentemente das GUI *console* que podem e são tipicamente utilizadas para acesso remoto para gerenciamento dos sensores, agentes e servidores, as CLIs são tipicamente utilizadas para acesso local [SM07]. Nos casos em que uma CLI é utilizada com acesso remoto, isso deve ser feito através de uma conexão encriptada utilizando serviços semelhantes ou o próprio SSH(*Secure Shell*. Apesar de serem dispositivos de gerenciamento locais, normalmente são mais difíceis de operar que um *console* e não fornecem todas as funcionalidades possibilitadas por eles [SM07].

### Habilidades de manutenção e construção

Muitas são as habilidades requeridas para implementação, operação e manutenção de NIDPS. Algumas bem conhecidas são as seguintes:

- Os responsáveis pela implementação dos componentes do sistema necessitam ter uma base de administração desses sistemas, gerência de redes e segurança da informação;
- Os responsáveis por customizar o sistema precisam de conhecimento avançado em segurança da informação e dos princípios que norteiam os NIDPS. Conhecimento sobre princípios de resposta a incidentes e as políticas e procedimentos da organização sobre os incidentes também é recomendável. É necessário também conhecimento de protocolos de rede, de aplicações, bem como dos sistemas operacionais sendo monitorados;

- Habilidades de programação também são altamente recomendadas, caso se deseje customização do sistema a nível de codificação, escrita de relatórios e tarefas semelhantes [SM07].

Caso a solução de NIDPS seja adquirida de um fornecedor, existem algumas maneiras para obter treinamento em suas ferramentas. Algumas delas são:

- Treinamento fornecido pela empresa;
- Conferências técnicas sobre a ferramenta;
- Livros e relatórios técnicos;
- Contratação de assistentes técnicos para trabalhar na empresa.

Apesar dessas facilidades, o mais importante na implantação de uma solução de segurança baseada em um NIDPS é uma avaliação bem detalhada de uma equipe que realmente tem conhecimento na área para não adquirir produtos que não atendem a necessidade ou que tenham demasiadas funções que não são utilizadas.

## 2.9 Implantação de NIDPS

Existem muitas opções quanto à decisão sobre a localização dos NIDPS e de seus sensores em redes. Cada uma possui diferentes impactos [SM01]. As arquiteturas de posicionamento de NIDPS (um ou mais) em redes aqui mostradas podem ser usadas em conjunto umas das outras, e muitas vezes o são.

A escolha, ou as escolhas devem ser feitas levando em consideração os objetivos da instalação do sistema de detecção de prevenção de intrusão e os recursos disponíveis. Listaremos as arquiteturas mais frequentes e suas vantagens:

### 2.9.1 Firewall de borda

A zona Desmilitarizada (conhecida, também, por DMZ) é uma subrede em que os computadores podem se comunicar entre si sem a intervenção de firewalls. Nessa arquitetura, é posicionado um NIDPS atrás de cada firewall externo, que protege a comunicação da DMZ com o mundo externo. Um exemplo de localização com esta arquitetura pode ser visto na Figura 2.1, na seta indicada por *location 1*. A seguir estão as vantagens associadas a esta arquitetura:

- Consegue enxergar ataques originados de fora para dentro da rede, e que tenham ultrapassado as defesas da rede, como firewalls;
- Fornece uma visão geral e detecta problemas nas políticas e na performance do firewall.
- Enxerga ataques que tem como alvo serviços frequentemente disponibilizados dentro da Zona Desmilitarizada, como servidores *Web* ou FTP;

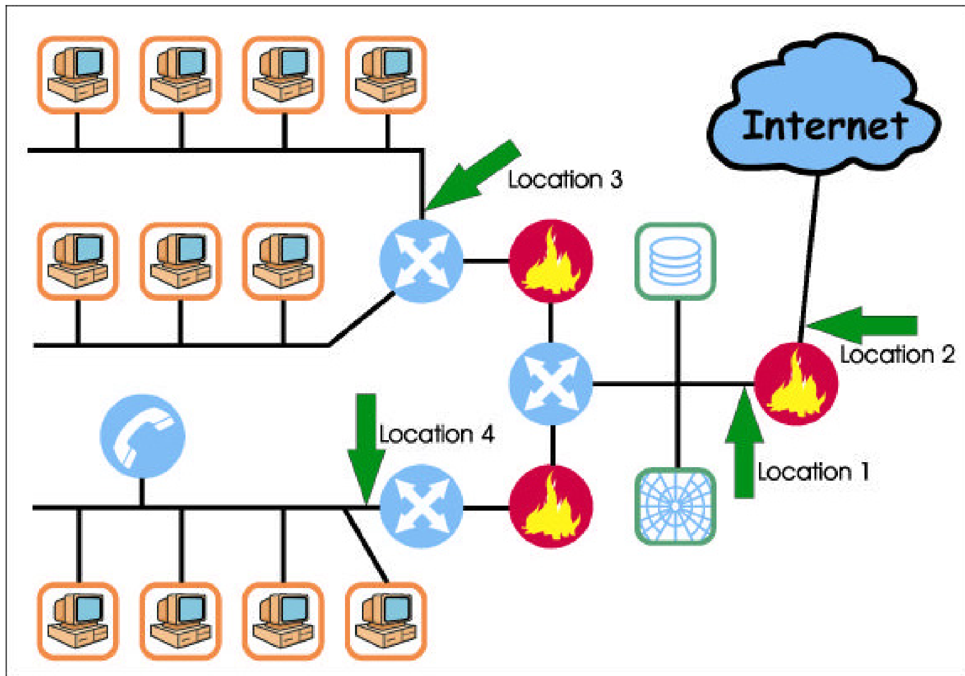


Figura 2.1: Localizações mais frequentes de NIDPS em redes [SM01].

- Por último, outra característica se destaca nessa arquitetura. Mesmo que um ataque consiga penetrar na rede, é possível que se reconheça resultados desse ataque pelo tráfego resultante gerado por ele e que sai da rede no sentido contrário ao ataque, para o caso de ataques que buscam por informações, não somente negação de serviços.

### 2.9.2 Na frente dos firewalls de borda

Nessa arquitetura, o NIDPS é posicionado entre o firewall de borda e a internet. É útil para documentar e analisar ataques externos. Um exemplo de localização com esta arquitetura pode ser visto na figura 2.1, na seta indicada por *location 2*. A seguir estão as vantagens associadas a esta arquitetura:

- É capaz de documentar a quantidade de ataques da internet que tem como alvo a sua rede.
- É capaz de documentar o número de ataques da internet que tem como alvo a sua rede.

### 2.9.3 Junto dos maiores *backbones* da rede

Nessa arquitetura, o NIDPS é posicionado nos maiores backbones da rede, ou seja, aqueles que apresentam maior quantidade de tráfego. Um exemplo de localização com esta arquitetura pode ser visto na figura 2.1, na seta indicada por *location 3*. A seguir estão as vantagens associadas a esta arquitetura:

- Monitora uma grande quantidade de tráfego da rede, aumentando a possibilidade de se detectar ataques.
- Detecta atividade não autorizada realizada por usuários internos ao perímetro de segurança da organização.

### 2.9.4 Em sub-redes críticas

Os NIDPS são posicionados na borda de sub-redes consideradas críticas, seja por disponibilizar serviços críticos ou conter informações importantes. Um exemplo de localização com esta arquitetura pode ser visto na figura 2.1, na seta indicada por *location 4*. A seguir estão as vantagens associadas a esta arquitetura:

- É capaz de detectar ataques cujo alvo são sistemas e recursos críticos dentro da organização.
- Possibilita focar o uso de recursos limitados de segurança em sub-redes consideradas como tendo maior valor e maior necessidade de segurança.

Após a revisão dos conceitos de básicos referentes aos IDPS atuais, serão analisadas as Unidades de Processamento Gráfico. Os conceitos estudados no próximo capítulo são altamente relevantes aos objetivos, uma vez que será a maior parte modificada tanto na arquitetura do Snort, quanto na arquitetura que será proposta para a solução do problema.

# Capítulo 3

# Unidades de Processamento Gráfico

## 3.1 Definição e Histórico

As Unidades de Processamento Gráfico (que chamaremos a partir de agora de GPU, do inglês *Graphics Processing Unit*) são dispositivos capazes de realizar processamento de cálculos e renderização de gráficos. Originalmente, e ao longo de muito tempo, teve suas capacidades voltadas para a execução de funções matemáticas comuns à área da computação gráfica (operações de transformação de matrizes, que representam espaços tri-dimencionais, são o exemplo mais comum). Possuem, historicamente, memórias independentes para texturas, constantes e demais dados de execução.

As primeiras GPUs remontam ao período entre 1970 e 1980, onde placas especializadas foram construídas somente com funções embarcadas, para que funções de computação gráfica fossem executadas. O objetivo era aumentar a performance dessas computações e diminuir a sobrecarga das CPUs. A arquitetura das primeiras GPUs era organizada em *pipeline*, do tipo *fixed function pipeline* [WPSAM10].

Na próxima década, as gerações seguintes de GPUs flexibilizaram o *pipeline* e permitiram que sequências personalizadas de instruções fossem executadas em determinados estágios do *pipeline*. Essas instruções foram chamadas de *shaders* e são capazes de manipular posições de vértices, primitivas geométricas ou atributos de *pixels* [FFY05].

Para que fosse possível a codificação desses *shaders*, foram criadas linguagens de programaçao específicas. A primeira que se tem notícia na literatura é do ano de 1984 e foi chamada de Shade Trees Language. No ano seguinte, foi proposta a Pixel Stream Editing Language, que finalmente permitiu o uso prático em GPUs. Em 1990, a empresa de animação por computação gráfica Pixar Animation Studios publicou a RenderMan Shading Language(RSL), linguagem que foi utilizada para a criação de filmes com computação gráfica de três dimensões.

Outras linguagens de *shaders* apareceram até que se estabelecessem os dois frameworks mais utilizados atualmente, OpenGL e DirectX, com suas próprias linguagens de *shaders*.

Com o advento e a evolução das linguagens de *shaders* as GPUs tornaram-se capazes de executar algoritmos de propósito geral, não somente funções pré-estabelecidas ou somente com poderes para computação gráfica. Essa nova forma de uso de GPUs é conhecida como GPGPU (do inglês, *General Purpose Graphics Processing Unit*).

Em seguida, será exposta a arquitetura CUDA, uma arquitetura para GPGPU produzida pelo fabricante de GPUs NVIDIA.

## 3.2 GPUs da ATI/AMD

Lançadas no fim de 2006, as GPUs ATI tem o CAL(*Computing Abstract Layer*) como é a parte do *Software* do ATI *Stream Computing* cujo objetivo é abstrair os detalhes de *hardware* para o *Stream Processor* da ATI [ATI10]. Suas principais funções são:

- Gerenciamento de Dispositivos;
- Gerenciamento de Recursos;
- Geração de código;
- Carregar e executar o kernel;

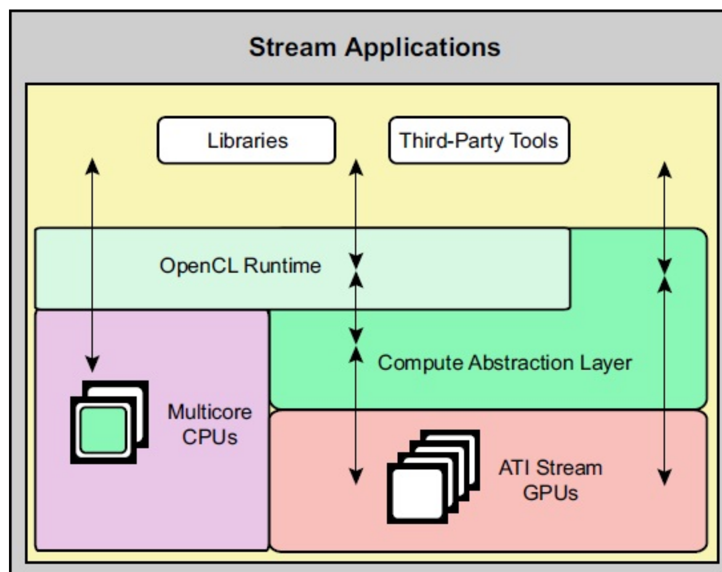


Figura 3.1: Arquitetura do *Stream Processor* ATI [ATI10]

Além disso, o CAL fornece uma biblioteca de dispositivos de driver que permite que as aplicações interajam com a GPU no mais baixo nível com performance otimizada enquanto mantém a compatibilidade de alto nível para o desenvolvedor.

O CAL é a base dos *Stream processors* ATI cuja organização pode ser vista na figura 3.1.

Apesar de os *Stream processors* ATI fornecerem uma maneira interessante de lidar com programação para GPGPU, não será profundo o seu estudo, visto que elas não serão utilizadas no no trabalho.

## 3.3 GPUs da NVIDIA

Também no fim do ano de 2006, a NVIDIA lançou sua *Compute Unified Device Archictecture*(CUDA) que é uma arquitetura de *hardware* e *software* para desenvolvimento e gerenciamento de problemas computacionais em GPU. Nesse arquitetura a GPU é utilizada como um dispositivo de processamento paralelo, mas sem as complicações de mapeá-lo

por uma API gráfica. Foi disponibilizada para as séries: *GeForce 8 Series*, *Tesla solutions* e algumas *Quadro solutions*.[NVI07].

A pilha do *software* CUDA é composta por várias camadas: um driver de *hardware*, uma Interface de Programação de Aplicação (API, do inglês) e seu ambiente de execução(*runtime*, e duas bibliotecas matemáticas de alto nível e uso comum - CUFFT e CUBLAS [NVI07]. O *hardware* foi desenvolvido para o suporte a processos leves (*lightweight*), também conhecidos como *threads* e possui camadas de ambiente de execução para assim maximizar a performance [NVI07].

Uma funcionalidade muito importante da CUDA API é que ela é apenas uma extensão da linguagem de programação C. Isso foi feito com o intuito de minimizar a curva de aprendizado do programador [NVI07].

### 3.3.1 Modelo de programação

Em CUDA, a GPU será vista como um dispositivo computacional capaz de executar um grande número de *threads* em paralelo [NVI07]. Ela será operada como um co-processador do processador principal, a CPU ou *host*. Sendo mais específico, as porções com grande paralelismo serão carregadas na GPU para sua execução [NVI07].

A idéia principal é isolar partes de código que executam dados independentemente e executá-lo na GPU para maximizar a vazão dos dados. Para que isso seja possível é necessário compilar a função para o conjunto de instruções do dispositivo, chamado de *kernel*, e inserí-lo nele [NVI07].

A organização interna do *kernel* é de um *grid* de blocos de *threads*.

Um **bloco de *threads*** é um conjunto de threads que podem cooperar eficientemente compartilhando dados através de uma memória compartilhada de alta velocidade e sincronizar sua execução para coordenar o acesso a memória [NVI07]. Cada *thread* é identificada por seu *thread ID*, que é o número da thread e o número do bloco. Normalmente esse endereçamento é complicado, sendo que muitas vezes são definidos vetores de duas ou três dimensões para facilitar esse endereçamento [NVI07].

Apesar da estrutura de blocos ser muito útil, ela é limitada no número de threads que um bloco pode conter. Por outro lado, blocos de mesma dimensão e tamanho e que executam o mesmo *kernel* podem ser colocados em conjunto dentro de um *grid*, fazendo assim com que o número de *threads* que são executadas pelo *kernel* seja muito maior. É formado assim um ***grid* de blocos de *threads***. Apesar de contornar o problema de quantidade de threads, essa abordagem faz com que seja reduzida a cooperação entre as *threads* simplesmente porque *threads* em diferentes blocos, mas no mesmo *grid* as threads só podem se comunicar pela memória global, que é uma memória de acesso lento [NVI07].

Assim como as *threads* os blocos também são identificados por um *blocoID*, que é o número do bloco como o número do *grid*. O endereçamento dos blocos enfrenta as mesmas dificuldades e é resolvido pela mesma técnica que nas *threads* [NVI07].

A topologia da organização das *threads*, blocos e *grids* pode ser visualizada na figura 3.2.

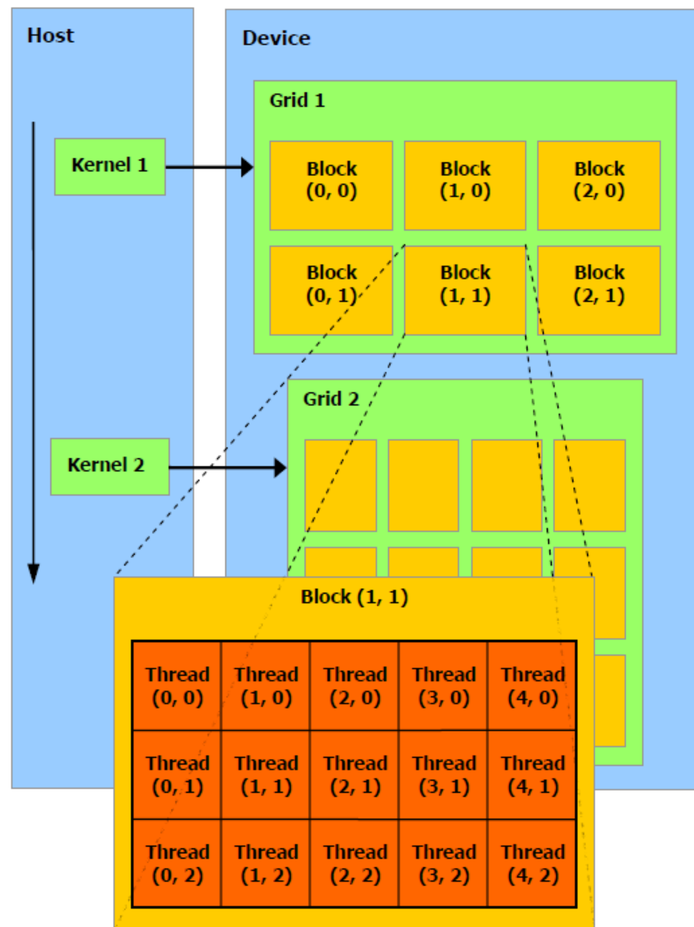


Figura 3.2: Topologia da organização de um programa na GPU [NVI07]

### 3.3.2 Acesso a memória

Uma *thread* em execução no dispositivo somente tem acesso a memória DRAM dele e as memórias *on-chip* através dos seguintes espaços de memória, que podem ser visualizadas na figura 3.3:

- Leitura/escrita por *thread*: *registradores*;
- Leitura/escrita por *thread*: *memória local*;
- Leitura/escrita por bloco: *memória compartilhada*;
- Leitura/escrita por *grid*: *memória global*;
- Somente leitura por *grid*: *memória constante*;
- Somente leitura por *grid*: *memória de textura*.

Os registradores, memórias locais e memórias compartilhadas são memórias internas aos blocos e são utilizadas durante a execução do *kernel*. Já as memórias global, constante e de textura além de serem persistentes durante a execução do *kernel* e poderem ser lidas e escritas pelo processador *host*, são otimizadas para diferentes usos de memória [NVI07].

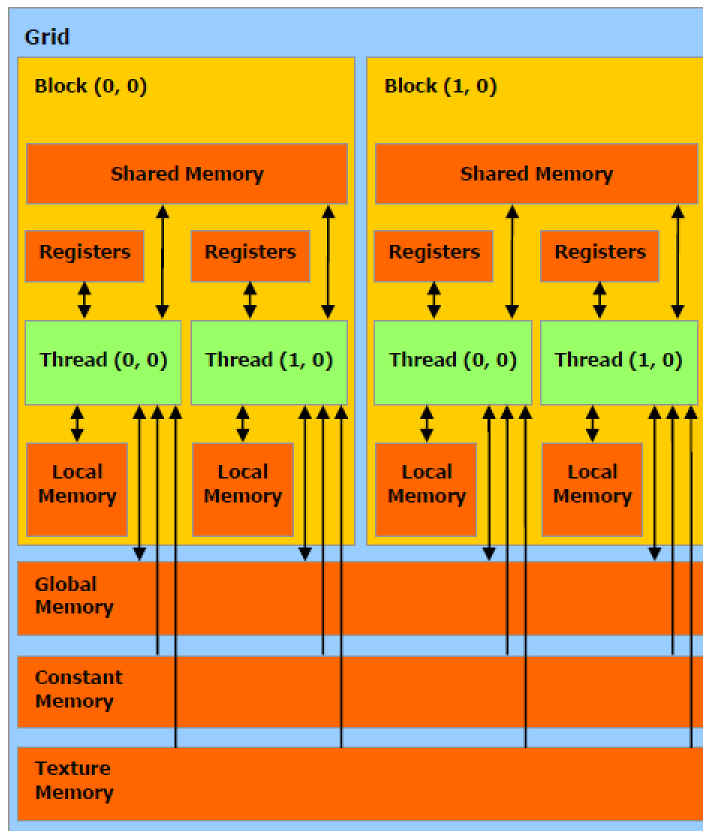


Figura 3.3: Organização do programa com visualização das memórias [NVI07]

O espaço de **Memória Global** não tem um sistema de cache e por isso é recomendado que seu acesso seja sequencial para conseguir a leitura da maior quantidade de dados em apenas um acesso a memória [NVI07].

Ambas, **Memória Constante e Memória de Textura** possuem um sistema de *cache* e portanto, ao menos que ocorra um *cache miss*, o acesso a elas custará apenas uma operação de leitura de *cache* [NVI07]. O acesso a Memória Constante será rápido caso todas as threads acessem o mesmo endereço, caso contrário o custo cresce linearmente. Já o acesso a Memória de Textura é otimizado para localização num espaço 2D, então caso os endereços de leitura sejam próximos o suficiente, teremos um acesso otimizado [NVI07].

## 3.4 Implementação do Hardware

Um dispositivo que implementa a arquitetura CUDA é um conjunto de multi-processadores. Cada multi-processador é organizado segundo uma arquitetura SIMD (Single Instruction, Multiple Data), como pode ser observado na figura (figura 3.4),.

A arquitetura SIMD é determinada por uma característica principal: durante a execução de um determinado programa, cada processador do multi-processador executa a mesma instrução, mas estas instruções atuam sobre diferentes dados. Essa arquitetura foi definida por Flynn, em 1972 [Fly72]. Foram apontados, também, problemas que estão associados a esta arquitetura, como o problema da comunicação entre os processadores

e o problema de degradação devido a *branches* no programa, chamado de *Branch Divergence* [Fly72]. Esses problemas serão mencionados ao decorrer deste trabalho.

Os dados que cada multi-processador pode acessar estão localizados em memórias internas ao multi-processador, que podem ser dos seguintes tipos [NVI07]:

- **Registradores**. Um conjunto de 32 registradores locais em cada processador.
- **Memória compartilhada**. Uma memória compartilhada por todos os processadores e que implementa o espaço de memória compartilhada.
- ***Cache* de constantes**. Também compartilhada por todos os processadores, é uma memória que permite somente sua leitura, e atua como aceleradora na leitura de constantes do espaço de memória para constantes.
- ***Cache* de texturas**. Assim como as anteriores, é uma memória *read-only* compartilhada por todos os processadores, e acelera a leitura de um espaço de memória para texturas.

Os dois primeiros itens citados são memórias implementadas como regiões de leitura e escrita, não possuindo *cache*, em oposição às memórias dos dois últimos itens da lista. O *cache* de texturas é acessado por cada multi-processador através da unidade de textura, mencionada anteriormente.

## 3.5 Modelo de Execução

Um *grid* de blocos de *threads* é executado em um dispositivo agendando-se blocos para execução nos multi-processadores. Cada multi-processador processa *grids* de blocos de *threads* e pode executar, assim, um *grid* de blocos seguido de outros, um por vez. Por questões de performance, um bloco é processado por somente um multi-processador, para que se utilize somente a memória interna ao multi-processador, levando a acessos à memória extremamente rápidos [NVI07].

O número de registradores e o tamanho da memória compartilhada de um multi-processador influência diretamente na quantidade de blocos que ele pode processar por vez. Os blocos que são processados por um multi-processador de uma vez são chamados de blocos ativos.

Caso não seja possível processar todas as *threads* de um bloco simultaneamente, cada bloco ativo é dividido em grupos de *threads*, chamados *warps*, contendo o mesmo número de *threads* cada. Através de um escalonador de *threads*, *warps* de tamanho escolhido para maximizar o uso dos recursos são executados de cada vez por uma determinada fatia de tempo (*quantum*), periodicamente dando espaço para outros, mantendo uma metodologia justa de escalonamento por *time-slice*.

As *threads* em um *warp* são sempre consecutivas (ordenadas pelo ID da *thread*), onde o primeiro *warp* contém a *thread* de ID 0. A ordem de execução dos *wraps* dentro de um bloco é indefinida, mas caso seja necessário sincronizar os *wraps*, para que acessem as memórias de forma ordenada, é possível que se defina pontos de sincronia programaticamente.

Já a questão de ordenação entre blocos dentro de um *grid* de blocos de *threads* é indefinida e não existe forma de sincronização [NVI07]. Como consequência disso, *threads*

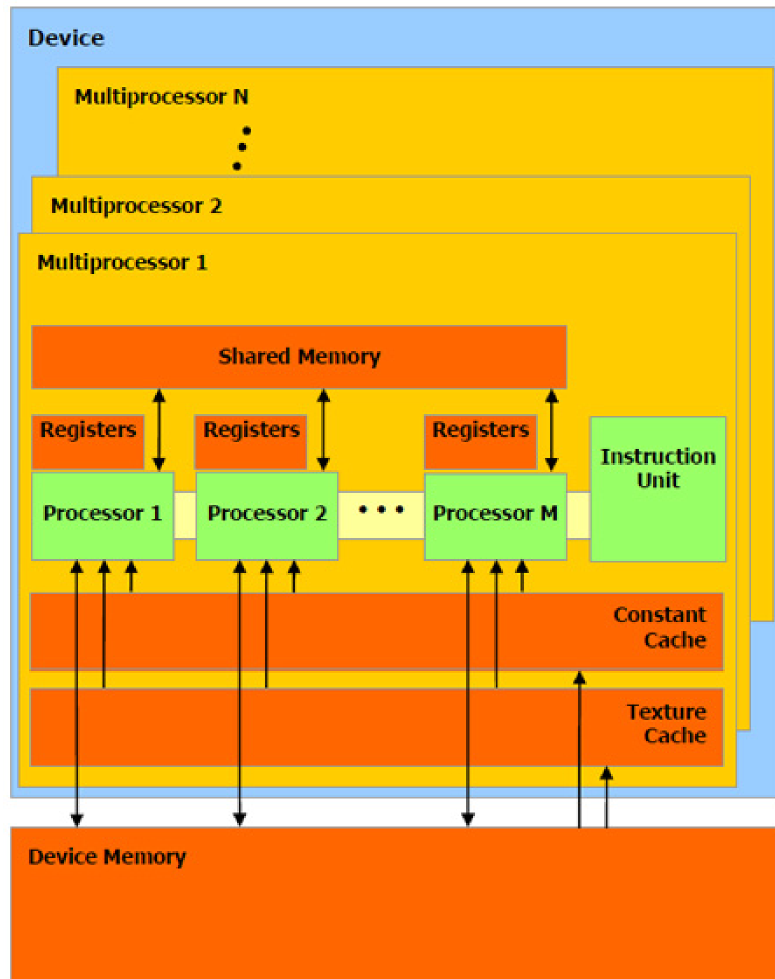


Figura 3.4: Um conjunto de multi-processadores SIMD com memória compartilhada [NVI07].

de blocos diferentes dentro de um mesmo *grid* não podem se comunicar com segurança pela memória global durante a execução de um conjunto de blocos.

Instruções não-atômicas que escrevem na mesma localidade da memória global ou compartilhada e são executadas por mais de uma *thread* em um *warp* o número de escritas serializadas que ocorrerão naquela localidade e a ordem em que elas ocorrerão é indefinida. Porém, é garantido que uma dessas escritas seja efetivamente realizada na memória. Para o caso de instruções atômicas, que leem da memória, modificam e escrevem de volta na mesma localidade, as instruções de diferentes *threads* são todas serializadas e é garantido que todas ocorram, mas a ordem também não é definida.

## 3.6 Dispositivos CUDA

A capacidade de computação em dispositivos CUDA pode ser comparada por um número principal de revisão e por um número secundário de revisão. Números principais iguais especificam que os dispositivos tem a mesma arquitetura, e números secundários representam melhorias incrementais.

Múltiplas GPUs podem ser usadas como dispositivos CUDA em um único *host* que tenha suporte a isso. O fabricante de placas de vídeo que atuam como dispositivos CUDA, a NVIDIA, implementa uma solução SLI (*Scalable Link Interface*) para aumento de performance de múltiplas GPUs. Porém, nesse modo, somente um único dispositivo CUDA é permitido. Logo, para que se use múltiplas GPUs como dispositivos CUDA, a solução SLI deve ser desativada.

## 3.7 Performance

### 3.7.1 Controle de fluxo de execução e *Branch Divergence*

Qualquer instrução de controle de fluxo de execução presente no programa a ser executado em um dipositivo CUDA (instruções condicionais, como *if* e *switch*; ou de iteração, como *do*, *while* e *for*) podem causar impacto negativo na velocidade de processamento. Isso ocorre porque instruções de controle de fluxo, ao serem executadas, podem causar *threads* de um mesmo *wrap* a divergir, isto é, seguir diferentes caminhos de execução.

Como as várias *threads* de um mesmo *wrap* são executadas de modo SIMD, as mesmas instruções devem ser executadas simultaneamente por todas essas *threads*. Quando um subconjunto das *threads* segue um caminho diferente de outro subconjunto, um dos subconjuntos deve esperar que o outro seja executado, para que, então, chegue sua vez de executar e o outro subconjunto tenha sua vez de esperar. Dizemos que diferentes caminhos de execução são serializados, e somente quando todos os caminhos de execução são completos, o programa pode convergir novamente e continuar a execução simultânea.

O impacto negativo no tempo de execução causado por esse fenômeno, que é chamado de *branch divergence*, é linear em relação ao número de divergências criadas pela execução [WPSAM10].

Em alguns casos, a condição que define qual o caminho a ser tomado em uma instrução de controle de fluxo depende do ID da *thread*. Nessas situações, é possível minimizar o número de *threads* divergentes. Isso é possível devido à característica determinística e ordenada da distribuição de *threads* em *warps*. Deve-se tentar ao máximo associar diferentes caminhos a serem tomados a *threads* que pertencem a um mesmo *warp*. Para este caso, e dependendo da quantidade de divergências necessárias, é possível que se elimine completamente o problema de *branch divergence*.

Em outros casos, o compilador pode desenrolar iterações (*loop unroll*). Isso pode ser controlado também pelo programador, caso necessário, com uma diretiva de compilação (*#pragma unroll*). Nessas situações, onde é possível que haja *loop unroll*, nenhum *warp* irá divergir.

Outra situação em que nenhum *warp* poderá divergir é quando o compilador otimiza instruções condicionais utilizando-se de *branch predication*. Neste caso, nenhuma instrução que dependa de uma condição deixa de ser executada. Em vez disso, cada *thread* possui um predicado para cada ponto de divergência, que é ativado ou não (de forma booleana), baseado na condição necessária para a divergência. Logo, todas as *threads* executam todos os caminhos de execução divergentes, mas quando o predicado está desativado todas as instruções de escrita, avaliação de endereços ou operações de leitura não são executados.

Existem várias vantagens associadas ao uso de *branch predication* e *loop unrolling* e entre elas destaca-se a possibilidade para o compilador de manter mais variáveis em registradores (por exemplo, quando diferentes índices de um vetor são acessados em diferentes iterações) e a remoção de execução de instruções passadas (tipicamente lenta em GPUs) [FFY05, WPSAM10] .

O compilador somente substitui uma instrução condicional por instruções com predicados caso o número de instruções controladas pela condição seja menor ou igual a um certo limite definido pelo compilador, que varia se o compilador avalia que a condição pode produzir poucos ou muitos *warps* divergentes.

### 3.7.2 Transferência de dados entre *host* e dispositivo

A largura de banda entre o dispositivo e memórias localizadas no dispositivo é muito maior do que a largura de banda entre dispositivo e memória de *host* [NVI07]. Outro fator que varia drasticamente, devido à distância e a largura de banda, é a latência, que é muito maior em transferências entre dispositivo e *host*. Transferência entre host e dispositivo devem, então, ser evitadas ao máximo.

Atualmente, as placas de vídeo produzidas pela NVIDIA são conectadas aos seus *hosts* por um barramento PCI Express 2.0 [NVI]. E as transferências entre *host* e dispositivo são realizadas por meio de DMA(*Direct Memory Access*) [NVI07].

O DMA permite que transferências sejam feitas com uma mínima atuação da CPU do *host*, de modo que somente um comando que inicia a transferência é efetuado pela CPU. A transmissão fica então, em sistemas convencionais, a cargo da *northbridge*, um componente da placa-mãe [Hov08], que avisa o *host* ou a GPU, por meio de uma interrupção, de que a transferência está completa. Esse mecanismo, além de liberar a CPU e o dispositivo para outras ações, torna a transferência mais rápida, pois CPUs convencionais só conseguem copiar áreas de memória em porções pequenas de *bytes*.

Embora a transferência ocorra com uso de DMA, ainda existe um grande *overhead* na transferência, como foi observado por Hovland, em 2008. Em testes realizados em um sistema convencional (figura 3.5), utilizando-se uma placa de vídeo do fabricante NVIDIA, foi aferido que uma transferência de poucos bytes do *host* para o dispositivo levou por volta de 9,6$\mu$s. Relativamente, pouca diferença foi observada ao realizar-se uma transferência de 512KB, que teve latência ainda menor que 9,7$\mu$s. Comportamento semelhante foi observado em transferências do dispositivo para o *host* [Hov08].

Logo, pequenas transferências devem ser evitadas, devendo ser substituídas por uma única grande transferência, caso isso seja possível. Deve-se, ainda, levar em consideração este *overhead* para que se decida se determinada computação será mais rápida rodando em um ambiente massivamente paralelo como a GPU, porém com uma penalização de tempo devido às transferências entre *host* e dispositivo, ou rodando em um ambiente sequencial ou pouco paralelo como a CPU, porém sem *overhead* extra. A Figura (figura 3.5) ilustra a latência para uma placa GTX 280.

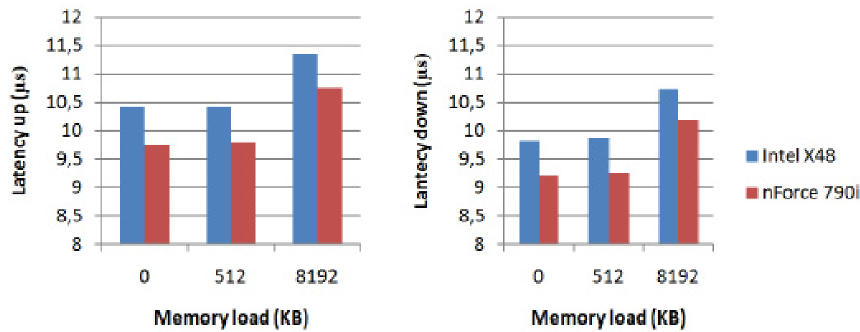


Figura 3.5: No primeiro gráfico (*latency up*), latência na transferência de um *host* com diferentes *chipsets* para um dispositivo CUDA (NVIDIA Geforce GTX 280), variando-se a quantidade de dados transferida. No segundo gráfico (*latency down*), a mesma situação, mas na direção oposta [Hov08].

## 3.8 API - *Application Programming Interface*

O principal objetivo da API é fornecer um caminho relativamente simples para usuários da linguagem C e para facilitar a escrita de programas para execução no dispositivo [NVI07]. Pode ser dividida em:

- Um conjunto mínimo de extensões que permitem ao programador se concentrar em porções do código fonte para a execução no dispositivo;
- Um biblioteca de execução dividida em:
  - Um componente de *host* que executa na CPU e provê controle e acesso ao dispositivo;
  - Um componente do dispositivo que executa na GPU e provê as funções específicas dela;
  - Um componente de uso geral que provê alocação de vetores e um subconjunto das bibliotecas padrão do C que são suportadas por ambos - GPU e CPU.

Para ilustrar a similaridade da linguagem C convencional e dos programas compilados para a GPU usaremos os exemplos ilustrados nas tabelas 3.1 e 3.2.

Como uma extensão da linguagem C, existem quatro características principais que podem ser destacadas por diferenciarem a API CUDA:

- Devem existir qualificadores de tipo de função que especificam se uma função executa no *host* ou no dispositivo e se a função pode ser chamada de cada um destes.
- Qualificadores de tipo de variável devem estar presentes para que seja especificado se uma variável deve ser alocada na memória do *host* ou do dispositivo CUDA (e em qual memória, neste caso).
- Uma nova diretiva para especificar como um kernel é executado no dispositivo a partir do *host*.

- Quatro variáveis novas que especificam as dimensões do *grid* e de blocos, além dos índices de blocos e threads.

Quanto aos qualificadores de tipo de função, existem três tipos. O qualificador *__device__* declara uma função que é executada no dispositivo e chamada somente a partir do dispositivo. O qualificador *__global__* declara uma função como um *kernel*, que é uma função chamada somente pelo *host* para que seja executada no dispositivo. E por fim, o qualificador *__host__* declara uma função que é chamada e executada somente pelo *host*. Os qualificadores de tipo de função devem estar presentes em todas as funções, e caso não sejam especificados, o compilador CUDA irá assumir que as funções são do tipo *host*.

No exemplo em CUDA (tabela 3.2), usamos a função *main* do programa sem qualificador, o que implica que ela será do tipo *__host__*. E a outra função é do tipo *__global__*. Essa função será nosso único *kernel*, e realizará a computação da nossa adição de matrizes no dispositivo CUDA.

Algumas restrições são impostas sobre cada tipo de função, devido às limitações da arquitetura do dispositivo. Funções especificadas como *__device__* ou *__global__* não suportam recursão, não permitem a declaração de variáveis estáticas dentro de seu escopo e não podem ter número variável de argumentos. Funções especificadas como *__device__* não podem ter seu endereço atribuído a um ponteiro. Funções declaradas como *__global__* (ou *kernels*), por outro lado, podem ter seu endereço obtido (os ponteiros dessas funções são válidos), e além disso, sempre devem ter retorno do tipo *void*.

Uma importante observação pode ser feita com relação à chamada de funções de *kernels*(a sintaxe para essa chamada pode ser vista no exemplo da tabela 3.2): essas chamadas são assíncronas, ou seja, retornam o controle ao programa no *host* que as chamou possivelmente antes da execução do *kernel* ser completa no dispositivo [NVI07].

Quanto aos qualificadores de variáveis, tem-se, também, três tipos. O qualificador *__device__* declara uma variável cuja memória é alocada no dispositivo. Deve ser usada em conjunto com no máximo um outro qualificador de variável. Caso não seja especificado nenhum outro, essa variável irá residir no espaço de memória global, irá durar durante todo o tempo de vida da aplicação e é acessível por todas as *threads*(por *grid*) e pelo *host*(usando-se chamadas de funções pertencentes à biblioteca de execução CUDA).

Junto do qualificador *__device__* podem aparecer um dos seguintes outros qualificadores. O primeiro é o qualificador *__constant__*, que determina que a variável declarada residirá no espaço de memória constante do dispositivo, terá a duração do tempo de vida da aplicação e possui a mesma acessibilidade do tipo *__device__* puro. O segundo qualificador que pode ser utilizado na associação é *__shared__*, que especifica que a variável residirá na área de memória compartilhada de cada bloco de *threads*, terá duração pelo tempo de vida do bloco e só pode ser acessada por *threads* que pertencem ao bloco.

Por outro lado, se nenhum qualificador aparecer associado a uma variável, e esta pertencer a uma função do localizada no *host*, então a variável residirá na memória do host. Caso a variável sem qualificador pertença a uma função localizada no dispositivo, geralmente residirá em um registrador do dispositivo, ou então, para o caso de grandes estruturas ou outros determinantes de compilação, na memória local.

Voltando ao exemplo, para especificarmos as dimensões de nossos blocos e grids, foram utilizadas as variáveis novas introduzidas pela API CUDA. A variável *gridDim* contém as

dimensões dos *grids*, *blockidx* contém o índice do bloco dentro do grid, *blockDim* contém as dimensões do bloco, *threadIdx* contém o índice de uma thread dentro de seu bloco.

| Programa C |
|---|

```
void add_matrix_cpu
(float *a, float *b, float *c, int N)
{
 int i, j, index;
 for (i=0;i<N;i++)
 {
  for (j=0;j<N;j++)
  {
   index = i+j*N;
   c[index] = a[index] + b[index];
  }
 }
}

void main()
{
 add_matrix_cpu(a,b,c,N);
}
```

Tabela 3.1: Código C para adiçao de matrizes

| Programa CUDA |
|---|

```
__global__ void add_matrix_gpu
 (float *a, float *b, float *c, int N)
{
 int i,j;
  i = blockIDx.x*blockDim.x+threadIDx.x;
  j = blockIDx.y*blockDim.y+threadIDx.y;
 int index = i+j*N

 if(i<N && j<N)
  c[index] = a[index] + b[index];
}

void main()
{
 dim3 dimBlock(blocksize, blocksize);
 dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);

 add_matrix_gpu<<<dimGrid,dimBlock>>>(a,b,c,N);
}
```

Tabela 3.2: O código CUDA para a adiçao de matrizes

No próximo capítulo serão comentados particularidades do Snort, um dos IDPS mais utilizados atualmente. Sendo ele o objeto de estudo, é importante entender seu funcionamento, arquitetura, decisões de projeto para que seja possível maximizar tanto a produtividade quanto a implementação de uma nova solução.

# Capítulo 4

# Snort: Sistema Detecção e Prevenção de Intrusão de Código Livre

Para os testes da proposta, foi escolhido um NIDPS amplamente conhecido e utilizado em grande parte das redes que contam com sistemas de detecção e prevenção de intrusão.

Snort é um Sistema de Detecção e Prevenção de Intrusão em Redes *open-source*, desenvolvido pela empresa americana Sourcefire, mesma produtora do anti-vírus de código-fonte aberto mais utilizado mundialmente, o ClamAV.

Esse sistema combina os benefícios da inspeção de pacotes por assinatura (padrões de bits), inspeção por estados de protocolos (*stateful protocol inspection*) e inspeção baseada em anomalias [Sno].

A figura 4.1 mostra a estrutura básica de funcionamento do Snort. O sistema deve ser previamente instalado e configurado com regras de detecção de anomalias e de alarmes. A sequência de passos pode ser definida como segue:

- Recebimento do pacote. O Snort recebe todos os pacotes que chegam a ele dependendo da arquitetura da rede.
- Análise do pacote. Para cada pacote, é verificado se ele ativa alguma regra pré-definida.
- Caso alguma regra seja ativada, seja pelo último pacote recebido ou por algum comportamento observado ao longo do tempo, o Snort emite sinais de alerta ao administrador, ou então realiza ações de contenção (caso seja programado para tal).

Neste trabalho, o foco será na atenção na segunda etapa citada anteriormente. O interesse é em aumentar a performance da análise de pacotes, descarregando a CPU dessa tarefa. Para isso, será feita a análise do pacote em uma GPU, que irá comparar cada pacote paralelamente às centenas ou milhares de regras que são definidas normalmente no Snort.

## 4.1 Arquitetura Lógica do Snort

O Snort é logicamente dividido entre múltiplos módulos de acordo com a figura 4.2. São componentes que trabalham em conjunto par detectar ataques de intrusão e fornecer

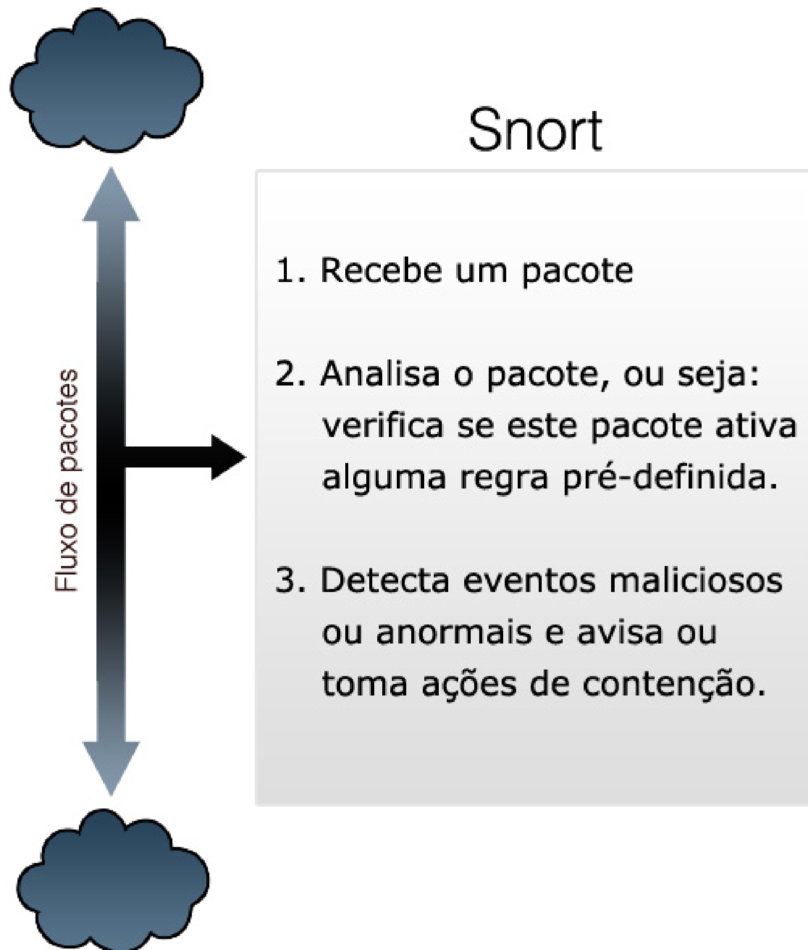


Figura 4.1: Estrutura básica de funcionamento do Snort.

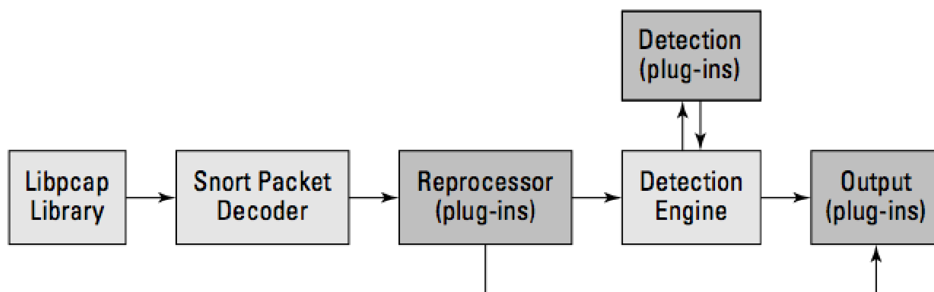


Figura 4.2: Estrutura lógica de funcionamento do Snort.

uma saída num formato conhecido para o sistema de detecção. O Snort pode ser dividido nos seguintes componentes:

- Aquisição de pacotes
- Decodificação de pacotes
- Preprocessadores

- Etapa de detecção
- Sistema de *log* e alertas
- Módulos de saída

### 4.1.1 Aquisição de pacotes

A aquisição de pacotes do Snort foi construída utilizando a biblioteca libpcap. Ela é uma biblioteca para C/C++ para captura de tráfego de rede [Pca].

Após a captura do pacote de rede, esse pacote é passado para a próxima etapa de detecção do Snort, a decodificação de pacotes.

### 4.1.2 Decodificação de pacotes

O módulo de decoficação de pacotes recebe os pacotes vindos da libpcap, capturados de diferentes interfaces de rede e prepara esses pacotes ou para o preprocessamento ou para a etapa de detecção.

### 4.1.3 Preprocessadores

Os preprocessadores, quando utilizados, desempenham um papel quase tão importante quanto a detecção no funcionamento do Snort. São componentes ou *plugins* que podem ser utilizados juntamente com o Snort para planejar ou modificar a estrutura dos pacotes antes que aconteça qualquer operação de detecção. Caso exista detecção baseada em anomalias, é nessas etapa que ela irá ocorrer, pois alguns preprocessadores contém essas regras de detecção, e produziram alertas.

Nota-se a importância dos preprocessadores pois os ataques de intrusão podem conter mudanças sutis na maneira como o ataque irá ocorrer, sem comprometer a lógica do ataque. Isso ocorre pois normalmente o NIDS procura por um *exact match*, ou seja, um comparação exatamente igual entre o pacote de entrada e a assinatura. Por exemplo, se for utilizada a assinatura "scripts/iisadmin" nos pacotes HTTP, o ataque poderia conter outras strings que não são exatamente a assinatura, mas executariam o mesmo ataque. São exemplos os seguintes:

- "scripts/./iisadmin"
- "scripts/examples/../iisadmin"
- "scripts\iisadmin"
- "scripts/.\iisadmin"

Esses problemas se tornam ainda maiores quando são consideradas URIs hexadecimal caracteres ou Unicode caracteres que são largamente aceitos por servidores. Assim, os servidores conseguem interpretar esses caracteres e extrair a string inicial deles. Como mencionado, se o NIDS estiver procurando por *exact match*, ele não encontrará. Uma das funções dos preprocessadores é de rearranjar os pacotes de uma maneira que os ataques sejam detectáveis.

Além disso, os preprocessadores também são responsáveis por lidar com a desfragmentação de pacotes. Quando uma grande quantidade de dados é transferida por meio da rede, o pacote é fragmentado, por causa do tamanho do MTU da camada de rede. Em NIDS, caso haja fragmentação, é primeiro é necessário remontar o pacote, pois uma parte do ataque pode estar em um fragmento e outra parte em outro fragmento. Essa é uma técnica bem comum em ataques de intrusão.

Os preprocessadores são a primeira barreira contra os ataques de intrusão, uma vez que identificam as formas mais básicas dos ataques. Caso não sejam utilizados, formas bem simples de ataque podem comprometer o sistema computacional que se deseja proteger.

### 4.1.4 Etapa de detecção

Essa é a parte mais importante de todo o Snort. É responsável por identificar se existe alguma tentativa de intrusão nos pacotes. Para tanto, o Snort utiliza regras para descrever as assinaturas dos ataques. Essas regras são carregadas dentro de estruturas de dados internas para que cada pacote seja comparado com elas. Se um determinado pacote comparado determinar que houve um *match* de assinatura a ação apropriada será tomada. As ações que podem ser tomadas são salvar um *log* do pacote ou gerar um alerta.

O tempo de execução da etapa de detecção do Snort é crucial para seu funcionamento. É necessário considerar a quantidade de regras utilizadas bem como o poder de processamento da máquina em que o Snort está rodando para que ele realmente forneça uma resposta em tempo real. O tempo de resposta do Snort depende basicamente dos seguintes fatores:

- Número de regras
- Capacidade da máquina onddee o Snort está sendo executado
- Velocidade do barramento interno da máquina onde o Snort está sendo executado
- Capacidade de leitura da interface de rede

Esses fatores sempre devem ser considerados no planejamento de um NIDS. Note também que a aplicação das regras podem ocorrer em diversas partes dos pacotes. São elas:

- O cabeçalho do pacote IP
- O cabeçalho do protocolo da camada de transporte. Normalmente são UDP ou TCP, mas pode ser outro protocolo da camada de transporte. Eventualmente pode-se considerar o ICMP.
- O cabeçalho do protocolo da camada de aplicação. São exemplos desses protocolo: FTP, SMTP, SNMP etc. Pode ser necessário utilizar métodos indiretos para analisar esse cabeçalho, como um *offset* onde a busca deve começar.
- O *payload* do pacote é onde se encontram os dados que se deseja transportar pela rede. Pode-se procurar uma *string* específica nele para determinar se um algum ataque está ocorrendo.

Em versões antigas do Snort, a procura por assinaturas era feita apenas até que uma delas fossem encontrada. Porém, isso poderia gerar problemas, uma vez que o pacote poderia conter várias assinaturas de vários níveis de comprometimento do sistema computacional. Por conta disso, ocorriam situações onde ataques de alta prioridade não eram identificados porque um ataque de baixa prioridade era identificado anteriormente. Nas versões mais modernas do Snort isso não ocorre. Primeiro ocorre a busca por todas as assinaturas de ataque e somente após isso as prioridades são avaliadas para que o sistema utilize a de maior periculosidade para gerar um alerta.

### 4.1.5 Sistema de *log* e alertas

Dependendo do que é encontrado em algum pacote na etapa de detecção, esse pacote pode ser utilizado para ser guardado em um *log* ou gerar um alerta. *Logs* são mantidos em arquivos de texto simples. Caso o diretório onde esses arquivos são salvos não sejam alterados, eles podem ser encontrados em:

```
/var/log/snort
```

### 4.1.6 Módulos de saída

Os módulos de saída ou os *plugins* podem realizar diferentes tarefas dependendo da forma como se quer que os alertas ou *logs* sejam salvos. Basicamente esse módulo controla como essa saída ocorrerá. Dependendo da configuração, eles podem realizar as seguintes tarefas:

- Salvar os *logs* dos pacotes em /var/log/snort/alerts ou em algum outro arquivo
- Enviar SNMP *traps*
- Enviar mensagens para o syslog
- Salvar os *logs* em um bando de dados MySQL ou Oracle
- Fornecer saídas em XML
- Modificar a configuração de *firewalls* e roteadores
- Enviar SMB(*Server Message Block* para máquinas Windows.

## 4.2 As regras do Snort e o uso de patterns para detecção de ataques

A maior parte das atividades maliciosas possuem algum tipo de assinatura. Informações sobre essas assinaturas são usadas para a criação de regras no Snort, que será responsável pela detecção das atividades maliciosas na rede.

Um conjunto de regras criadas pela equipe responsável pelo Snort, da *SourceFire*, está disponível para *download* no site snort.org. Essas regras são disponibilizadas gratuitamente com 1 mês de atraso, e as regras atualizadas são disponíveis mediante pagamento.

Regras podem ser conseguidas de outras fontes, também gratuitas, ou criadas pelo administrador da rede.

As assinaturas que as regras detectam podem estar presentes no cabeçalho dos pacotes ou então em seu *payload*. O Snort é capaz de analisar num pacote as camadas de rede, transporte e de aplicação e as regras são divididas em duas partes.

A primeira parte da regra é um cabeçalho, que define a ação que deve ser tomada caso a assinatura seja detectada, o protocolo e os endereços e portas dos pacotes que devem ser avaliados pelo Snort contra esta regra. Existem opções avançadas para melhor especificar os elementos citados anteriormente.

A segunda parte da regra são opções, que podem ser múltiplas e uma assinatura só é satisfeita se todas as opções forem satisfeitas. Um exemplo de opção são *flags* para detectar ACK, que é uma opção existente no cabeçalho dos pacotes do protocolo TCP.

Outros exemplos são a opção "content"e opções que a acompanham. É possível definir texto ou *bytes* específicos que devem aparecer no *payload* do pacote, assim como a posição em que eles podem aparecer e características como se a procura deve ser sensível ao caso ou não. Novamente, informações mais específicas podem ser adicionadas através de opções mais avançadas.

É possível, também, que sejam definidas opções que são expressões regulares para reconhecimento dos padrões no conteúdo dos pacotes. Devido ao elevado custo computacional para que se compare textos ou *bytes* com expressões regulares, o *matching* da expressão regular só irá ser realizado pelo Snort caso as demais opções presentes na regra sejam satisfeitas.

As regras são definidas em arquivos de configuração próprios para isso. A escrita dessas regras é bem intuitiva. Em geral, o Snort irá alertar o usuário sobre regras incorretas ou que possam não fazer sentido. E em geral, irá tomar atitudes para elevar a performance da comparação das regras, como a citada anteriormente.

No próximo capítulo será feita uma análise da implementação. Ela será composta pela análise do problema e suas particularidades relacionadas a GPU, a proposta e projeto da solução, bem como a descrição da implementação em detalhes.

# Capítulo 5

# Estratégia e Descrição da Implementação

## 5.1 Tarefas de detecção de intrusão adequeadas à execução em GPU

Para definir quais as tarefas da detecção de intrusão que são adequadas à execução em GPU é preciso considerar e classificar pelas diferentes metodologias de detecção.

### 5.1.1 Detecção baseada em assinaturas

Esse tipo de detecção envolve a comparação de diversos padrões pré-definidos de ataques de intrusão. Portanto, essa detecção é feita por meio da comparação de strings com uma lista de ameaças de intrusão conhecidas. Isso se deve ao fato de que a procura é exatamente a assinatura na análise dos pacotes ou *log* de entrada, logo a operação que será realizada é a de *pattern matching* por meio da comparação de strings.

Esse tipo de comparação parece adequado a execução em GPU porque a arquitetura delas é SIMD, ou seja, em um determinado momento de execução, cada multi-processador está executando a mesma instrução, mas sobre dados diferentes. Logo seria possível realizar a operação de comparação do pacote ou *log* de entrada contra vários padrões paralelamente, diminuindo assim o tempo necessário para realizar todas essas comparações contra as regras.

A fase de comparação de padrões é uma fase que, muitas vezes, ocorre para determinar se uma comparação mais profunda e computacionalmente mais cara, com expressão regular, deve ser realizada. Apesar de ser uma fase posterior, essa operação também é passível de ser portada para uma GPU. De acordo com a arquitetura da GPU, seria possível transformar as expressões regulares em AFD (Autômato Finito Determinístico). Assim, ocorrerá exatamente o que acontece no caso do *pattern matching*. Seriam executada sempre a mesma instrução nas *threads* do multi-processador da GPU, mas sobre dados diferentes, nesse caso os dados para criar os AFDs e os pacotes ou *logs* de entrada.

Assim, nas duas partes do NIDPS onde são executadas as buscas para identificar uma intrusão é possível portá-la para a GPU.

### 5.1.2 Detecção baseada em anomalias

Já a detecção baseada em anomalias não irá buscar exatamente um padrão pré-definido, mas analisará o perfil adquirido na fase de aprendizagem para buscar por intrusões. Apesar disso, a maneira como a busca irá acontecer não necessita de nenhum tipo de alteração com relação a detecção baseada em assinaturas, apenas a interpretação dos dados obtidos pela busca.

Portanto, como a procura por intrusão é idêntica a detecção baseada em anomalias, ela também pode ser implementada em uma GPU.

### 5.1.3 Detecção baseada na análise dos estados do protocolo

Na análise dos estados do protocolo, temos os perfis pré-determinados de execução dos protocolos de rede. São criadas sequências de passos para determinados protocolos para se avaliar de os protocolos não estão sendo utilizados de forma suspeita. Essas sequências de passos podem ser mapeadas como AFDs. Como observado anteriormente, será criado um algoritmo de percorrimento de autômato e ele será sobre os dados dos protocolos e dos dados de entrada da rede.

Porém, nessa metodologia somente um ou poucos AFDs existirão, e a cada pacote novo na rede, apenas uma transição de estados ocorreria. Logo, a detecção baseada na analíse dos estados do protocolo não fornece condições suficientes para criação de uma adaptação de sua análise para uma GPU.

## 5.2 As tarefas do Snort que são adequadas a execução em GPU

De acordo com a *Source Fire*, empresa que desenvolve Snort, ele é um NIDPS que combina os benefícios de busca por assinaturas, baseada em protocolos e anomalias [Sno].

Assim sendo e de acordo com as premissas vistas na seção anterior, todas essas metodologias de detecção poderiam ser escolhidas para a implementação do módulo de busca em uma GPU, com exceção da baseada na análise dos estados do protocolo.

## 5.3 As tarefas do Snort escolhidas para a execução em GPU

O maior esforço do Snort se concentra na detecção baseada em assinaturas. Além disso, como visto anteriormente, a detecção baseada em assinaturas se adapta facilmente à arquitetura SIMD, de GPUs.

## 5.4 Descrição da arquitetura da solução proposta

O Snort, por ser um NIDPS de código aberto e como um bom tempo de desenvolvimento, foi escolhido como o objeto de trabalho para implementar um módulo de comparação paralela de padrões de assinatura.

Na figura 5.1, observa-se um esquema resumido da arquitetura proposta para a solução.

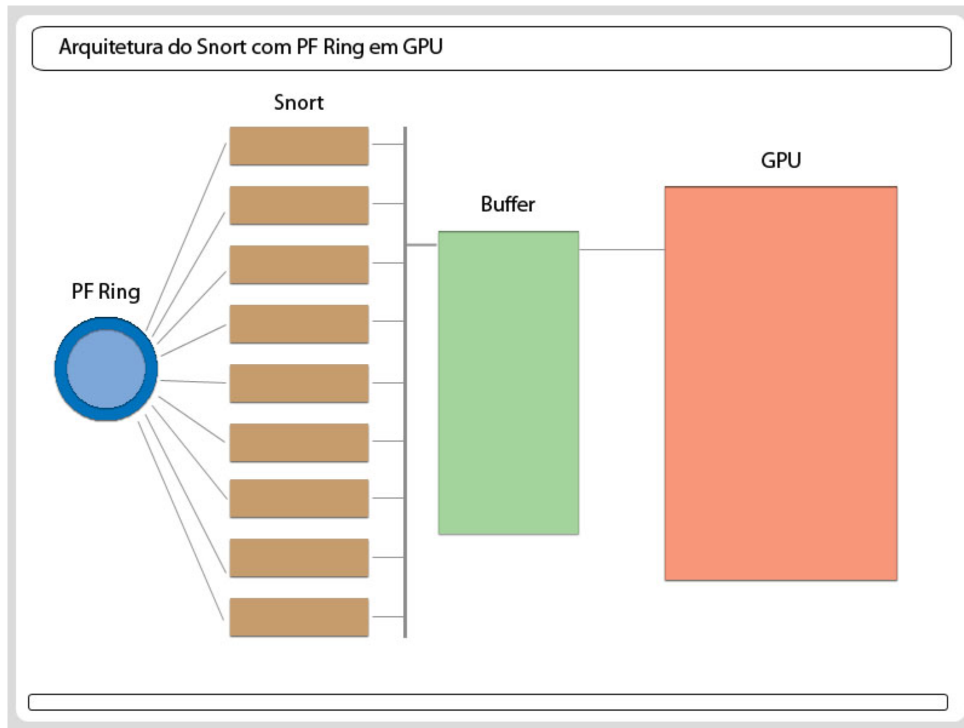


Figura 5.1: Arquitetura proposta para o Snort como módulo de comparação paralela de padrões de assinatura.

Apesar de ser um NIDPS bem aceito e aclamado, o Snort tem um código fonte bastante complicado de ser alterado. Inicialmente, a arquitetura iria ser composta apenas por um processo do Snort, que iria criar um *buffer* de pacotes a serem transferidos à GPU para realizar o *matching* com os *patterns*. Esse *buffer* se faz necessário para diminuir o tempo gasto na transferência de pacotes da memória da CPU para a memória da GPU. Infelizmente, a execução do Snort funciona de maneira altamente sequencial e para cada pacote lido da interface de rede, ele espera o resultado das comparações para adquirir outro pacote, impossibilitando que fosse construído um buffer com esses pacotes.

A partir desse problema, para maximizar a performance da solução com uso de GPU para detecção, seria necessária outra arquitetura que possibilitasse a criação desse *buffer* de pacotes.

Após uma nova rodada de pesquisa, decidiu-se utilizar um novo tipo de *socket* para redes que aumenta bastante a capacidade de captura de pacotes da rede, o PF_RING [PFR]. Modificações no Snort já foram realizadas para que ele funcione com o PF_RING. Nesse caso, o PF_RING funciona como um buffer circular, encaminhando cada pacote para um dos diversos processos independentes de Snort em execução. Uma observação importante é que o PF_RING não encaminha pacotes de um mesmo fluxo (conexão) para processos diferentes do Snort, não comprometendo, assim, a capacidade do Snort de detecção de pacotes sequenciais em um fluxo. Isso permite aumentar a quantidade de processos do Snort em execução paralela (dependendo do número de núcleos da CPU), maximizando o uso da CPU. E por outro lado, como vários Snorts terão pacotes para serem analisados simultaneamente, seria possível criar um buffer de pacotes para serem enviados à GPU.

Resumidamente, o *design* da solução utiliza o PF_RING como *socket* para uma melhor aquisição dos pacotes. Esse *socket* é responsável por distribuir de maneira uniforme os pacotes entre várias instâncias (processos) de Snort, tantas quanto o processador suportar. No Snort, logo antes de onde ocorreria a comparação na CPU, eram capturados os pacotes e formados *buffers* que seriam enviados para a GPU executar a rotina de comparação. Com os resultados obtidos na GPU de volta aos Snorts, eles continuariam no seu fluxo, sem realizar a comparação na CPU, até que um novo pacote seja adquirido do PF_RING e o ciclo se repita novamente.

Esse *buffer* será mantido em um processo independente, que será responsável pela aquisição de pacotes dos diferentes Snorts, envio deles para a GPU, coleta dos resultados obtidos pela GPU e passagem desses resultados de volta para cada Snort que continha pacote no *buffer*. Esse funcionamento será detalhado mais adiante.

### 5.4.1 Particularidades das regras do Snort consideradas na Implementação

Nessa seção, não serão descritas especificamente nenhuma regra do Snort, apenas a explicação dos modificadores que devem ser considerados e como devem ser considerados na implementação dessa solução. São cinco modificadores:

- *Offset*;
- *Depth*;
- *Distance*;
- *Within*;
- *nocase*

Esse modificadores atuam sobre um outro parâmetro das regras do Snort, o *content*. Ele é um dos mais simples dentro do Snort, uma vez que apenas define uma *string* que deve ser procurada nos pacotes. Porém os modificadores mudam a forma como o Snort deve procurar essa *string* nos pacotes.

1. Offset
   De acordo com o manual do Snort, é a posição que a especificação da regra especifica para começar a procurar por um *pattern* nos pacotes. Quando existe mais de um *content*, normalmente há uma confusão sobre a ordem de ocorrência, mas o modificador somente indica a posição de início da procura, não importando a ordem.

2. Depth
   O modificador *depth* indica até qual posição do pacote o Snort deve procurar por aquele *pattern*, uma vez que o modificador *offset* apenas define onde a busca se inicia.

3. Distante
   O modificador *distance* indica a partir de qual posição o Snort deve começar a procurar por um segundo *match* relativo ao último. Assim como no offset, não define o fim da busca no pacote.

4. Within
   Esse modificador permite definir um intervalo em *matches* relativos ao modificador *content*. Usualmente é usado pra definir um segundo ponto de parada para a busca no pacote.

5. nocase
   O mais simples de todos os modificadores, simplesmente define se a busca deve ser feita considerando o caso.

## 5.5 A estratégia de Implementação

### 5.5.1 Modificações no Snort

Para que cada Snort conseguisse se comunicar corretamente com o nosso *buffer* de pacotes e, consequentemente, com o módulo da GPU, é preciso atribuir um identificador único a cada instância do Snort. Essa foi a primeira modificação que foi necessária para adequar o Snort a nova execução. Esse ID agora é recebido como argumento na inicialização de cada Snort.

Um ponto importante da arquitetura deriva da arquitetura do próprio Snort. O Snort cria grupos de regras, separado suas regras em grupos baseados no protocolo, portas de origem e de destino de cada regra. Diversos grupos são, então, criados. Para cada grupo desses, o Snort cria o que é chamado de máquina de busca de *patterns*. O objetivo desta divisão em várias máquinas é minimizar o trabalho durante o *pattern matching*, visto que cada máquina só deve cuidar dos patterns das regras a que ela se refere. Na hora da execução, o Snort repassa o trabalho de *matching* para somente uma das máquinas, já que, por exemplo, uma regra para pacotes TCP não precisa ser analisada contra pacotes UDP.

Analisando o conjunto padrão de regras do Snort e os resultados produzidos por ele na execução do programa, foi descoberto que do conjunto de cerca de 1500 *patterns* presentes no Snort, 1368 *patterns* são relativos a um só grupo, de HTTP. Então, considerando que o maior esforço do Snort se concentra na análise desse grupo de *patterns* HTTP e que boa parte do tráfego que circula nas redes atuais é utilizando o protocolo HTTP, foi escolhido como objeto de estudo somente a comparação dos *patterns* da máquina de busca que representa esse grupo. As outras máquinas continuarão sendo executadas na CPU, visto que o tempo de execução delas em CPU é bem pequeno e não justificaria o *overhead* do tempo de transferência de dados da CPU para a GPU, nem a volta. Com esse número de *patterns* sendo comparados paralelamente na GPU, foi possível atingir uma quantidade significativa de paralelização da comparação, como será descrito adiante.

Por fim, foi necessário alterar a máquina de busca de *patterns* para que uma chamada, que pode ser de criação da máquina, adição de *patterns*, finalização ou a busca propriamente dita, se comunicasse corretamente com o nosso módulo de *buffer*, para que este realizasse o *matching* na GPU.

### 5.5.2 Criação do *buffer* de Pacotes

Como dito anteriormente, o módulo de *buffer* de pacotes é um módulo escrito por nós que roda em um processo independente dos Snorts. Suas principais funcionalidades são:

- Inicialização da GPU com os *patterns* desejados.
- Criação e mantenimento de um *buffer* de pacotes ou frames enviados pelos Snorts que estão execução em diversos processos.
- Envio desse *buffer* para a GPU para que os pacotes sejam comparados com todos os patterns.
- Retorno dos resultados para os Snorts que tinham pacote em *buffer*.

A comunicação dos Snorts com o processo de *buffer* se dá pela criação de uma área de memória compartilhada entre esses processos. Cada Snort escreve em um espaço reservado especificamente a ele dentro desta área. O processo de *buffer* percorre a área de memória em busca de novos pacotes a serem enviados para a GPU.

Para a sincronia de comunicação entre um Snort e o *buffer*, foram criados contadores de leitura e de escrita na área de memória compartilhada. Quando um Snort possui um novo pacote a ser enviado ao *buffer*, ele escreve o pacote na memória compartilhada e aumenta o contador de escrita. O *buffer* percebe, então, que os contadores de leitura e escrita estão diferentes, e sabe que existe um pacote a ser processado para aquele Snort. Dessa maneira foi possível garantir a consistência das operações de leitura e escrita.

A partir deste ponto, o Snort que passou o pacote para o *buffer* entra em uma espera por uma variação de condição, implementada com o uso de *Mutex* e da função *cond_wait()*. Optou-se por essa opção para que os processos de Snort não ficassem em *busy waiting*, atrapalhando o funcionamento dos demais Snorts e demais processos do sistema operacional. As variáveis do *Mutex* e da *cond_wait()* residem na área de memória compartilhada de cada Snort e foram criadas com as devidas flags para funcionamento inter-processos. Uma vez processado o *buffer*, o processo de *buffer* envia um sinal para que os Snorts saiam da espera pela variável de condição e possam continuar seu trabalho, já com os resultados esperados em memória.

No processo de *buffer* foi criado um mecanismo de *timeout* para que o *buffer* seja enviado para a GPU. Caso o *buffer* esteja cheio, ele é enviado automaticamente à GPU. Porém, caso exista algum pacote no *buffer* mas o *buffer* não esteja cheio, após um tempo pré-determinado de *timeout* o(s) pacote(s) serão enviados à GPU da mesma forma.

### 5.5.3 Criação do Módulo da GPU

Para a inicialização da GPU, o processo de *buffer* é iniciado e espera até que um primeiro Snort seja iniciado também. Este primeiro Snort passa seu conjunto de *patterns* para o *buffer* através de um arquivo, utilizando um mecanismo de sincronia. Após essa etapa, o processo de *buffer* copia os *patterns* para a memória global da GPU, inicializa demais memórias, incluindo requisição por memória *page-locked* para a comunicação dos resultados da GPU de volta para a CPU utilizar DMA (*Direct Memory Access*).

Após a etapa de inicialização, os Snorts que forem sendo iniciados já estarão prontos para se comunicar com o *buffer* e receber os *patterns* encontrados pela GPU de volta em sua memória compartilhada.

Quando o buffer de pacotes está cheio, ele é copiado para a memória global da GPU e o kernel de comparação de strings é disparado na GPU. Esse kernel tem como inputs um vetor de patterns e um vetor de pacotes (vindos do buffer) na memória global do dispositivo. Seu output é um vetor de patterns detectados, que reside na memória da CPU e é acessado do dispositivo via DMA.

Na GPU, cada thread irá realizar a comparação de strings para um pattern específico. Sendo assim, todos pacotes serão analisados por todas as threads, de maneira paralela. Em cada thread, a comparação é feita em um loop, para 1 pacote por vez, com o pattern específico daquela thread.

Uma thread sabe qual pattern utilizar do vetor pelo seu ID, calculado baseado no número do bloco e da thread dentro do bloco no qual ela está rodando na GPU. Os IDs são números sequenciais e equivalentes aos IDs que atribuímos aos patterns, sendo assim, todos os patterns são facilmente acessados no seu vetor residente na memória global.

Ao término da comparação para cada pacote, a thread escreve, via DMA, se houve ou não detecção naquela thread (pattern) para aquele pacote.

Quando a execução do kernel termina, o processo de buffer percorre o vetor de patterns detectados para determinar se houveram detecções de patterns na análise. Os resultados são escritos na memória compartilhada e libera os Snorts que estavam em espera, para que possam continuar sua execução e futuramente capturar mais pacotes.

No próximo capítulo, serão comentados os testes realizados com a nova implementação, afim de avaliar o desempenho da solução, bem como os detalhes referentes as diversas métricas realizadas para a mensuração dos dados.

# Capítulo 6

# Resultados Obtidos

## 6.1 Descrição dos testes realizados

O sistema desenvolvido foi projetado para analisar uma grande quantidade de tráfego de rede e caso seja utilizado para a analisar apenas o tráfego que passa por um único computador, por exemplo, não utilizará todo o seu poder de processamento e dessa maneira não mostrará melhora significativa em relação ao Snort que já é utilizado. Isso se dá devido à melhora significativa de desempenho caso o *buffer* de pacotes enviado à GPU esteja cheio.

Para contornar esse problema e considerando que a arquitetura apenas considera as requisições e respostas do protocolo HTTP, foi gerada e capturada uma grande quantidade desses dados para utilizar a máxima capacidade da modificação.

Também foi adicionada uma regra HTTP simples ao arquivo de regras. Apesar de o Snort ser planejado para evitar intrusão em sistemas computacionais, caso o tráfego gerado não contivesse conteúdo malicioso não seria possível verificar como o *feedback* do Snort se comportava para o propósito esperado. Assim, foi adicionada uma regra que do ponto de vista de intrusão não é representativa, mas do ponto de vista de testes permite verificar a performance do Snort com as modificações realizadas.

Para efeitos de comparação, os mesmos testes realizados no Snort utilizando o PF_RING foram realizados na adaptação do algoritmo.

Em testes preliminares, foi verificado que a solução apresentava resultados abaixo dos esperados, logo a opção foi não realizar testes de *throughput* do sistema como um todo. Foi decidido separar a execução da solução em 8 etapas independentes, e realizar aferições de tempo para cada uma delas, sob o nosso tráfego capturado para testes.

O objetivo desses testes foi identificar os pontos que não apresentavam desempenho razoável e, além disso, determinar se a causa do problema de performance era a parte do *pattern matching* na GPU ou a solução que foi criada em volta do Snort, para adaptar a solução, que não possui arquitetura favorável à execução na GPU.

Após a determinação dessa causa, foram realizados testes para avaliar somente a perfomance da GPU para comparação de string com *patterns* reais (advindos do próprio Snort) e pacotes reais capturados da rede. Sendo assim, para este teste houve desacoplação da solução do Snort e aferição dos resultados do tempo total gasto para a comparação de diversas quantidades de pacotes. Neste teste, foi possível obter dados que permitiram calcular um throughput máximo utilizando a nova solução de *pattern matching* na GPU.

Esses testes foram realizados utilizando 1, 2, 3 e 4 processos de Snort com PF_RING. Como foi utilizado um processador (CPU) com 2 cores, foi escolhido uso da ferramenta do sistema operacional *schedtool* para a definição dos processos divididos da maneira mais uniforme possível entre os 2 cores.

Para cada execução do teste, foi executado o(s) Snort(s) e o processo de *buffer* e a alimentação do sistema com os pacotes do nosso conjunto de pacotes de teste. Calculou-se a média do tempo de execução da etapa desejada para todos estes pacotes. Cada teste foi repetido por 10 vezes para que fosse possível obter resultados mais realistas. Para as 10 rodadas, foram calculadas a média e o desvio padrão do tempo obtido.

Como segunda parte desses resultados, foram realizados testes desacoplando a nova solução do Snort. Ou seja, foi fornecido o *buffer* de pacotes diretamente ao processo de *buffer*. Para estes testes foram fornecidas diversas quantidades de pacotes para o *buffer*, aumentando-as exponencialmente. O tempo de aferição foi o tempo para processamento total, de todos os pacotes fornecidos ao *buffer*. Os testes foram repetidos por 10 vezes para cada quantidade de pacotes.

## 6.2 Descrição do ambiente

O ambiente de testes utilizado deve ser um computador com sistema operacional UNIX-like que tenha suporte ao PF_RING e que contenha uma GPU Nvidia e com *cuda capability* maior ou igual a 1.1. Além disso, é necessário que exista suporte à flag PTHREAD_PROCESS_SHARED na *Pthread*, o que não ocorre em alguns sistemas operacionais. Dito isso, o ambiente de testes utilizado nos testes foi o seguinte:

- Placa mãe Abit A9wd-Max
- Processador Intel Core 2 Duo 3Ghz, 2mb de cache
- Memória ram 4Gb ddr2 1066mhz
- Placa de vídeo GTX 480 1536 MB GDDR5
- Sistema Operacional Ubuntu 11.10 64 bits
- Placa de rede Ethernet 10/100

É importante notar que o sistema só funciona corretamente caso a interface de rede esteja funcionando adequadamente e o Snort instalado esteja apto a capturar os pacotes que passam por ela. Essas premissas são válidas também para o Snort convencional não modificado.

Para a captura do conjunto de pacotes de teste foi escolhida a ferramenta AB (da *Apache*), que gera requisições HTTP automaticamente. A ferramenta foi configurada para enviar 1000 requisições, com paralelismo (quantidade de requisições enviadas simultaneamente) de 10 requisições. Executou-se o AB para enviar requisições para um servidor de testes local. Ocorreu a captura das requisições e das respostas para a formação do conjunto de pacotes de teste.

## 6.3 Descrição dos resultados

Para cada uma das seguintes etapas (divididas independentemente da execução da nova solução), foram realizados testes independentes:

- Tempo de criação do lock, da espera por variável de condição e cópia do pacote do Snort (e seus metadados) para a sua respectiva área de memória compartilhada (tabela 6.1).
- Tempo de espera no *buffer* de pacotes (tabela 6.2).
- Tempo de cópia dos pacotes da área de memória compartilhada para formar o *buffer* que será enviado para a GPU (tabela 6.3).
- Tempo de transferência do *buffer* de pacotes da memória da CPU para a memória global da GPU (tabela 6.4).
- Tempo de execução do kernel na GPU com o loop de busca - *pattern matching* (tabela 6.5).
- Tempo de percorrimento dos resultados escritos pela GPU na memória da CPU (via DMA) e escrita dos mesmos na memória compartilhada (tabela 6.6).
- Tempo de liberação do lock (*Mutex*) e envio de sinal para a variável de condição (tabela 6.7).
- Tempo de percorrimento dos resultados na área de memória compartilhada e *callback* das detecções no snort (tabela 6.8).

Nas tabelas abaixo, é possível conferir os resultados dos testes aferindo o tempo de cada uma das etapas descritas acima.

É possível, também, visualizar o tempo total gasto pela nossa solução (soma das 8 etapas) em uma tabela independente (tabela 6.9).

Para efeitos de comparação, foram realizados testes com o Snort utilizando apenas o PF_RING, sem as nossas modificações (tabela 6.10).

Tabelas comparativas entre o desempenho do nosso tempo total e do tempo total do Snort não-modificado foram produzidas (tabela 6.11).

Os resultados do segundo teste, desacoplando o módulo da GPU do Snort, podem ser observados em uma tabela (tabela 6.12) e um gráfico (figura 6.1).

## 6.4 Discussão dos resultados

De acordo com a análise do tempo gasto em cada etapa, foi possível notar alguns fatos:

- A etapa que levou mais tempo, por grande margem, foi uma grande surpresa. O envio de sinal para os Snorts, através da sinalização de uma variável de condição (função cond_signal), leva muito tempo e não é viável na essa nova arquitetura, apesar de ter sido uma solução necessária.

Tabela 6.1: Tempo de criação do lock, da espera por variável de condição e cópia do pacote do Snort (e seus metadados) para a sua respectiva área de memória compartilhada

| **Medida** | **Um Snort($\mu$s)** | **Dois Snorts($\mu$s)** | **Três Snorts($\mu$s)** | **Quatro Snorts($\mu$s)** |
|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 |
| 2 | 1 | 0 | 0 | 1 |
| 3 | 0 | 1 | 1 | 0 |
| 4 | 1 | 1 | 1 | 2 |
| 5 | 0 | 0 | 1 | 0 |
| 6 | 0 | 0 | 0 | 1 |
| 7 | 1 | 1 | 1 | 1 |
| 8 | 0 | 1 | 0 | 2 |
| 9 | 1 | 0 | 1 | 0 |
| 10 | 1 | 1 | 1 | 0 |
| Média | 0.5 | 0.5 | 0.7 | 0.7 |
| Desvio Padrão | 0.52705 | 0.52705 | 0.48305 | 0.82327 |

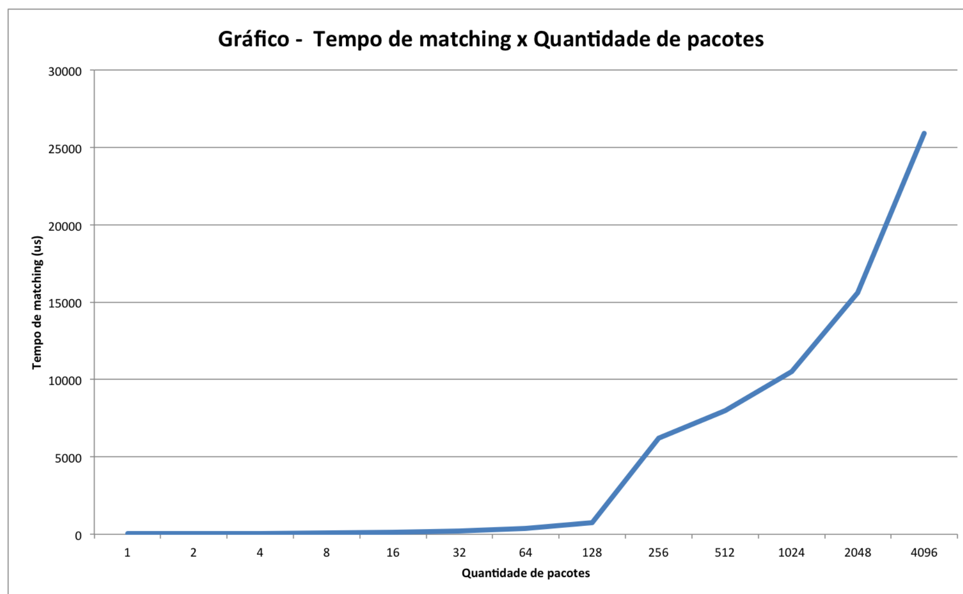


Figura 6.1: Gráfico de Desempenho

- Foi possível observar que o tempo médio de espera no buffer também é um fator limitante. Porém, esse valor pode ser amenizado com a diminuição do tempo de *timeout* do processo de *buffer*, mas somente caso haja tráfego suficiente na rede para que nesse tempo ainda seja possível preencher completamente o *buffer*.

- Dentre os demais tempos, os únicos com influência na soma total foram tempos realmente necessários para a realização do *pattern matching* na GPU, são eles: a transferência do *buffer* de pacotes para a GPU, a execução do kernel na GPU, e a transferência dos resultados de volta para a CPU. Esses tempos, na realidade, não tiveram tanta influência no tempo total, representando de 25 a 35% deste.

Tabela 6.2: Tempo de espera no *buffer* de pacotes

| **Medida** | **Um Snort($\mu$s)** | **Dois Snorts($\mu$s)** | **Três Snorts($\mu$s)** | **Quatro Snorts($\mu$s)** |
|---|---|---|---|---|
| 1 | 0 | 10 | 22 | 45 |
| 2 | 0 | 9 | 22 | 45 |
| 3 | 0 | 8 | 22 | 45 |
| 4 | 0 | 10 | 22 | 45 |
| 5 | 0 | 7 | 22 | 45 |
| 6 | 0 | 10 | 22 | 45 |
| 7 | 0 | 9 | 22 | 45 |
| 8 | 0 | 8 | 22 | 45 |
| 9 | 0 | 10 | 22 | 45 |
| 10 | 0 | 7 | 22 | 45 |
| Média | 0 | 8.8 | 22 | 45 |
| Desvio Padrão | 0.00000 | 1.22927 | 0.00000 | 0.00000 |

- O tempo de execução do kernel na GPU pode ser consistentemente estimado por uma função matemática da forma $f(X) = a + Xb$, onde $a$ é uma constante de valor alto, e $b$ um valor bem pequeno, se comparado a $a$. Esse grande tempo de inicialização se deve à preparação para execução do kernel na GPU, porém o tempo de real execução do algoritmo dentro do kernel é bem pequeno. Isso gera a conclusão de que quanto maior for o buffer passado para a GPU, mais haverá amortização do tempo de inicialização.

Como resultado geral, foi obtida redução na performance de 160 a 220% se comparada a nova solução ao Snort padrão utilizando o PF_RING. Porém, esses resultados são relativos à soma de todas as etapas da solução, sendo que muitas delas foram utilizadas devido a limitações do Snort. Tal fato esconde que por trás destes resultados, é possível visualizar ganhos reais de performance caso não fossem necerrária uma arquitetura tão grande e com tanto *overhead* para a adequação um IDPS sequencial a acumular um buffer de forma paralela.

Seguindo este pensamento, e considerando que outras arquiteturas de IDPS seriam mais adequadas à essa modelagem e não seria necessária a arquitetura de adaptação, foi realizada uma segunda rodada de testes que levavam em consideração somente o tempo de realização do *pattern matching* na GPU. Nesse tempo foi incluido os tempos de transferência de pacotes para a GPU, de execução do kernel e de transferência e percorrimento na CPU dos resultados obtidos pela GPU. Os resultados podem ser observados na tabela 6.12 e na figura 6.1.

Nesses testes, pode-se notar o aumento exponencial da quantidade de pacotes no buffer analisado pela GPU , utilizando de 1 a 4096 pacotes no mesmo buffer. Também foi possível observar o tempo gasto em crescimento linear quase horizontal até 128 pacotes. A partir de 256 pacotes se obteve um aumento considerável no tempo de execução total, mesmo que os resultados ainda se mantenham bons se comparados a execução na CPU. Esse aumento se deve a um limite na quantidade de memória *page-locked* que pode ser alocada no sistema, tornando o uso de DMA mais lento, já que para desempenhos ótimos requer-se que toda a memória do DMA esteja alocada em memória *page-locked*.

Tabela 6.3: Tempo de cópia dos pacotes para formar o *buffer* que será enviado para a GPU

| **Medida** | **Um Snort($\mu$s)** | **Dois Snorts($\mu$s)** | **Três Snorts($\mu$s)** | **Quatro Snorts($\mu$s)** |
|---|---|---|---|---|
| 1 | 1 | 1 | 6 | 4 |
| 2 | 1 | 2 | 4 | 5 |
| 3 | 2 | 5 | 2 | 4 |
| 4 | 2 | 4 | 2 | 6 |
| 5 | 3 | 3 | 2 | 5 |
| 6 | 2 | 3 | 4 | 4 |
| 7 | 1 | 3 | 4 | 5 |
| 8 | 2 | 1 | 5 | 3 |
| 9 | 1 | 2 | 6 | 2 |
| 10 | 1 | 1 | 3 | 6 |
| Média | 1.6 | 2.5 | 3.8 | 4.4 |
| Desvio Padrão | 0.69921 | 1.35401 | 1.54919 | 1.26491 |

Para efeitos comparativos, caso fosse executado o *pattern matching* na CPU, com o algoritmo do Snort, para 128 pacotes o tempo médio gasto seria de 3507.2$\mu$s (27.4$\mu$s para cada pacote). No segundo teste foi obtido um tempo médio de 752.7$\mu$s, ou seja, um ganho de performance de cerca de 366%.

Novamente para efeitos de comparação, com 4096 pacotes no buffer o tempo gasto na CPU seria de 112230.4$\mu$s, e o tempo gasto na GPU foi de 25890.9$\mu$s. Ou seja, novamente um excelente ganho em performance, de cerca de 433%.

No próximo capítulo, serão realizadas nossas considerações finais sobre o estudo, bem como o fechamento dos objetivos relacionados aos resultados obtidos.

Tabela 6.4: Tempo de transferência do *buffer* de pacotes da memória da CPU para a memória global da GPU

| **Medida** | **Um Snort($\mu$s)** | **Dois Snorts($\mu$s)** | **Três Snorts($\mu$s)** | **Quatro Snorts($\mu$s)** |
|---|---|---|---|---|
| 1 | 7 | 28 | 16 | 25 |
| 2 | 7 | 25 | 22 | 86 |
| 3 | 9 | 38 | 37 | 28 |
| 4 | 12 | 26 | 22 | 33 |
| 5 | 8 | 36 | 45 | 33 |
| 6 | 8 | 23 | 21 | 34 |
| 7 | 9 | 40 | 40 | 36 |
| 8 | 7 | 31 | 39 | 34 |
| 9 | 21 | 19 | 37 | 32 |
| 10 | 13 | 14 | 15 | 75 |
| Média | 10.1 | 28 | 29.4 | 41.6 |
| Desvio Padrão | 4.35762 | 8.37987 | 11.20714 | 20.90827 |

Tabela 6.5: Tempo de execução do kernel na GPU com o loop de busca (*pattern matching*)

| **Medida** | **Um Snort($\mu$s)** | **Dois Snorts($\mu$s)** | **Três Snorts($\mu$s)** | **Quatro Snorts($\mu$s)** |
|---|---|---|---|---|
| 1 | 7 | 8 | 17 | 11 |
| 2 | 12 | 9 | 10 | 8 |
| 3 | 13 | 12 | 15 | 13 |
| 4 | 13 | 18 | 11 | 15 |
| 5 | 6 | 19 | 9 | 13 |
| 6 | 10 | 20 | 12 | 18 |
| 7 | 17 | 18 | 17 | 18 |
| 8 | 18 | 10 | 11 | 18 |
| 9 | 18 | 10 | 19 | 18 |
| 10 | 10 | 8 | 15 | 8 |
| Média | 12.4 | 13.2 | 13.6 | 14 |
| Desvio Padrão | 4.29987 | 4.93964 | 3.43835 | 4.05518 |

Tabela 6.6: Tempo de percorrimento dos resultados escritos pela GPU na memória da CPU (via DMA) e escrita dos mesmos na memória compartilhada.

| **Medida** | **Um Snort($\mu$s)** | **Dois Snorts($\mu$s)** | **Três Snorts($\mu$s)** | **Quatro Snorts($\mu$s)** |
|---|---|---|---|---|
| 1 | 18 | 22 | 24 | 25 |
| 2 | 15 | 24 | 23 | 24 |
| 3 | 15 | 25 | 23 | 24 |
| 4 | 16 | 22 | 22 | 22 |
| 5 | 16 | 22 | 23 | 31 |
| 6 | 18 | 25 | 23 | 22 |
| 7 | 15 | 22 | 24 | 24 |
| 8 | 19 | 24 | 23 | 24 |
| 9 | 16 | 22 | 25 | 24 |
| 10 | 17 | 22 | 25 | 22 |
| Média | 16.5 | 23 | 23.5 | 24.2 |
| Desvio Padrão | 1.43372 | 1.33333 | 0.97183 | 2.61619 |

Tabela 6.7: Tempo de liberação do lock (*Mutex*) e envio de sinal para a variável de condição

| **Medida** | **Um Snort($\mu$s)** | **Dois Snorts($\mu$s)** | **Três Snorts($\mu$s)** | **Quatro Snorts($\mu$s)** |
|---|---|---|---|---|
| 1 | 45 | 75 | 92 | 136 |
| 2 | 71 | 146 | 74 | 165 |
| 3 | 47 | 57 | 86 | 307 |
| 4 | 23 | 129 | 113 | 171 |
| 5 | 45 | 51 | 177 | 283 |
| 6 | 46 | 177 | 114 | 104 |
| 7 | 27 | 74 | 87 | 334 |
| 8 | 76 | 169 | 145 | 109 |
| 9 | 56 | 68 | 217 | 246 |
| 10 | 25 | 95 | 119 | 102 |
| Média | 46.1 | 104.1 | 122.4 | 195.7 |
| Desvio Padrão | 18.13192 | 47.23335 | 45.28478 | 89.12420 |

Tabela 6.8: Tempo de percorrimento dos resultados na área de memória compartilhada e *callback* das detecções no snort

| **Medida** | **Um Snort($\mu$s)** | **Dois Snorts($\mu$s)** | **Três Snorts($\mu$s)** | **Quatro Snorts($\mu$s)** |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 |
| Média | 0 | 0 | 0 | 0 |
| Desvio Padrão | 0.00000 | 0.00000 | 0.00000 | 0.00000 |

Tabela 6.9: Tempos totais gastos na execução da nossa modificação

| | **Um Snort($\mu$s)** | **Dois Snorts($\mu$s)** | **Três Snorts($\mu$s)** | **Quatro Snorts($\mu$s)** |
|---|---|---|---|---|
| Etapa 1 | 0.50000 | 0.50000 | 0.70000 | 0.70000 |
| Etapa 2 | 0.00000 | 8.80000 | 22.00000 | 45.00000 |
| Etapa 3 | 1.60000 | 2.50000 | 3.80000 | 4.40000 |
| Etapa 4 | 10.10000 | 28.00000 | 29.40000 | 41.60000 |
| Etapa 5 | 12.40000 | 13.20000 | 13.60000 | 14.00000 |
| Etapa 6 | 16.50000 | 23.00000 | 23.50000 | 24.20000 |
| Etapa 7 | 46.10000 | 104.10000 | 122.40000 | 195.70000 |
| Etapa 8 | 0.00000 | 0.00000 | 0.00000 | 0.00000 |
| **Total** | **87.20000** | **180.10000** | **215.40000** | **325.60000** |
| Desvio Padrão | 29.44939 | 64.99651 | 62.93433 | 118.79201 |

Tabela 6.10: Snorts não-modificados, com PF_RING

| Medida | **Um Snort($\mu$s)** | **Dois Snorts($\mu$s)** | **Três Snorts($\mu$s)** | **Quatro Snorts($\mu$s)** |
|---|---|---|---|---|
| 1 | 22 | 26 | 243 | 36 |
| 2 | 9 | 14 | 21 | 87 |
| 3 | 21 | 75 | 38 | 109 |
| 4 | 11 | 51 | 62 | 28 |
| 5 | 7 | 46 | 27 | 228 |
| 6 | 57 | 162 | 66 | 321 |
| 7 | 81 | 115 | 105 | 92 |
| 8 | 23 | 14 | 69 | 141 |
| 9 | 35 | 35 | 24 | 78 |
| 10 | 8 | 25 | 171 | 32 |
| **Média** | **27.4** | **56.3** | **82.6** | **115.2** |
| Desvio Padrão | 24.25879 | 48.34149 | 72.44799 | 93.86610 |

Tabela 6.11: Comparativo do nosso com relação ao original. Os resultados se referem à porcentagem de redução de performance.

| | | | | |
|---|---|---|---|---|
| Média | 218.25% | 219.89% | 160.77% | 182.64% |
| Desvio Padrão | 53.71% | 113.34% | 135.38% | 212.66% |

Tabela 6.12: Resultados das medidas realizadas considerando somente o módulo de GPU

| N$^o$ de pacotes | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 50 | 51 | 51 | 63 | 49 | 60 | 68 | 64 | 52 | 47 |
| 2 | 55 | 52 | 54 | 54 | 55 | 53 | 53 | 54 | 53 | 54 |
| 4 | 63 | 62 | 64 | 63 | 61 | 64 | 64 | 61 | 65 | 64 |
| 8 | 101 | 112 | 107 | 97 | 105 | 112 | 91 | 82 | 108 | 79 |
| 16 | 117 | 163 | 156 | 152 | 113 | 152 | 112 | 112 | 152 | 155 |
| 32 | 180 | 187 | 198 | 249 | 191 | 192 | 251 | 246 | 240 | 252 |
| 64 | 305 | 320 | 423 | 423 | 335 | 336 | 442 | 453 | 451 | 336 |
| 128 | 847 | 811 | 845 | 625 | 595 | 936 | 576 | 615 | 837 | 840 |
| 256 | 7716 | 6403 | 6564 | 7593 | 1541 | 6447 | 6458 | 6455 | 6415 | 6493 |
| 512 | 7511 | 9503 | 7742 | 7709 | 7516 | 7620 | 7702 | 7720 | 7779 | 9083 |
| 1024 | 12639 | 10487 | 6232 | 12840 | 12705 | 10038 | 10309 | 9972 | 9921 | 10081 |
| 2048 | 14246 | 14519 | 21192 | 22311 | 14610 | 20252 | 9329 | 14543 | 12452 | 12632 |
| 4096 | 28225 | 38096 | 23186 | 17918 | 17889 | 30852 | 30673 | 23577 | 23799 | 24694 |

# Capítulo 7

# Conclusão

O objetivo principal desse trabalho era propor uma possível adaptação do Snort para a utilização da capacidade de processamento das GPUs na sua lógica de comparação de pacotes com *patterns* presentes em suas regras e com a arquitetura proposta na figura 5.1.

O objetivo secundário era mostrar a viabilidade de um Snort ou um outro IPDS com algoritmos de busca por padrões de assinatura ou expressões regulares para sistemas reais. Um primeiro ponto da viabilidade era o ganho real de processamento, sendo que a utilização de placa gráfica poderia não ser compensatório dependendo desse ganho. Outro ponto importante é o valor investido em hardware, também necessita ser compensatório.

É possível dizer que o objetivo principal foi alcançado, porém o secundário não. Entretanto os resultados são de certa maneira intrigantes. Os resultados apresentados no capítulo 6 somente nos mostram que a utilização de GPUs em IPDS é completamente possível, já que conseguimos ganhos reais no tempo de comparação dos pacotes contras os *patterns*. Porém os mesmos resultados mostraram que a nossa implementação não é viável.

Isso demonstrou que o problema é sim adequado a uma roupagem de GPU, mas a arquitetura proposta não foi adequada a proporcionar ganhos reais, uma vez que ela foi proposta pensando em não modificar o código atual do Snort. O Snort foi utilizado como objeto de estudo por ser uma das melhores ferramentas de detecção e prevenção de intrusão, porém foi também um limitante a melhores desempenhos da adaptação. Uma série de considerações e decisões tiveram que ser tomadas no decorrer do projeto para que o foco da adaptação não fosse perdido e praticamente todas elas foram para não modificar o funcionamento atual do Snort.

Assim, conclui-se que boa parte dos IDPS atuais teriam um ganho potencial de desempenho caso utilizassem algoritmos de comparação de pacotes contra *patters* e expressões regulares em GPU, desde que sejam feitas modificações de arquitetura no próprio IDPS para suportar uma captura eficiente dos pacotes e sua transferência de maneira otimizada para a GPU. Além disso, os parâmetros de execução da GPU também devem ser calibrados para uma melhor performance, mas o problema é altamente adequado a GPU bem como uma solução bem desenhada acarretará num bom desempenho.

## 7.1 Trabalhos Futuros

Devido ao tempo de execução do projeto, bem como as dificuldades impostas pelo processo, algumas tentativas de adaptação da arquitetura nem sequer foram cogitadas. Além disso, seriam tentativas que poderiam melhor a execução da solução, bem como resolver os problemas da arquitura. São elas:

- Adaptação de toda a arquitetura do Snort;
- Utilização de outro IDPS para o estudo, algum cuja a arquitetura já fosse planejada para o paralelismo;
- Utilização de semafóros para o controle da escrita/leitura de memória

# Referências


[AAM04] S. Antonatos, K. G. Anagnostakis, and E. P. Markatos. Generating realistic workloads for network intrusion detection systems. In *Proceedings of the Fourth International Workshop on Software and Performance (WOSP)*, pages 72–79, Dezembro 2004. 2

[ATI10] ATI. *ATI Stream Computing*. March 2010. vii, 19

[BBEN07] J. Beale, A. R. Baker, J. Esler, and S. Northcutt. *Snort: IDS and IPS toolkit*. Syngress, February 2007. 2

[CGLM04] J. B. D. Cabrera, J. Gosar, W. Lee, and R. K. Mehra. On the statistical distribution of processing times in network intrusion detection. In *Proceedings of the 43rd IEEE Conference on Decision and Control*, pages 75–80, 2004. 2

[FFY05] J. A. Fisher, P. Faraboschi., and C. Young. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Electronics & Electrical. Morgan Kaufmann, 2005. 18, 26

[Fly72] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, September 1972. 22, 23

[Goe11] K. M. Goertzel. Information Assurance Tools Report – Firewalls. Seventh Edition. Technical report, Information Assurance Technology Analysis Center (IATAC), IATAC 13200 Woodland Park Road Herndon, VA 2017, May 2011. 5

[Hov08] R. J. Hovland. Latency and bandwidth impact on gpu-systems. Technical report, Department of Computer and Information Science, Norwegian University of Science and Technology, 2008. vii, 26, 27

[IKBS00] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a distributed firewall. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 190–199, New York, NY, USA, 2000. ACM. 5

[NdG07] E. T. Nakamura and P. L. de Geus. *Segurança de redes em ambientes cooperativos*. Novatec Editora, 2007. 1

[NVI] NVIDIA. http://www.nvidia.com. Acessado em: 10/11/2011. 26

[NVI07] NVIDIA. *NVIDIA CUDA Programming Guide 1.1*. 2007. vii, 20, 21, 22, 23, 24, 26, 27, 28

[Pca] Libpcap. http://www.tcpdump.org/. 32

[PFR] NVIDIA. http://www.nvidia.com. Acessado em: 10/11/2011. 38

[SM01] K. Scarfone and P. Mell. *Intrusion Detection Systems*. National Institute of Standards and Technology, 2001. vii, 1, 6, 7, 8, 9, 15, 16

[SM07] K. Scarfone and P. Mell. *Guide to Intrusion Detection and Prevention Systems (IDPS)*. National Institute of Standards and Technology, February 2007. 1, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

[Sno] Snort Intrusion Detection System. http://www.snort.org/. 30, 37

[VSP+08] G. Vasiliadis, S.Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 116–134, Berlin, Heidelberg, 2008. Springer-Verlag. 1, 6

[WPSAM10] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying gpu microarchitecture through microbenchmarking. *IEEE International Symposium on Performance Analysis of System & Software (ISPASS)*, 1(9):235–246, Março 2010. 18, 25, 26

[Wu09] T. M. Wu. Information Assurance Tools Report – Intrusion Detection Systems. Sixth Edition. Technical report, Information Assurance Technology Analysis Center (IATAC), IATAC 13200 Woodland Park Road Herndon, VA 2017, September 2009. 6, 9, 11