Capítulo 1

Processamento de Pacotes Baseado em GPU

André G. Vieira, Gustavo Pantuza, João V. Soares, Filipe Pirola, Gustavo Viveiros, Marcos A. M. Vieira, Luiz F. M. Vieira

Abstract

The increasing volume of data traversing modern networks poses significant challenges to the performance and scalability of communication systems. In this context, leveraging Graphics Processing Units (GPUs) for packet processing has emerged as a promising solution, harnessing their massive parallelism to accelerate critical tasks such as routing, packet inspection, and intrusion detection. This work offers a comprehensive introduction to GPU-based packet processing, covering theoretical foundations in networking and parallel computing, as well as practical implementations using technologies like CUDA and frameworks such as DOCA, GPUNet, and PacketShader. Key technical aspects are explored, including memory management, load balancing, and energy efficiency, alongside real-world case studies and performance comparisons between CPU- and GPU-based approaches. The work aims to bridge theory and practice, focusing on applied scenarios and research opportunities, and is targeted at students and professionals in the field of Computer Science. Finally, emerging trends such as the integration with Software-Defined Networking (SDN) and GPU virtualization in distributed environments are discussed, highlighting the strategic role of these technologies in shaping the future of computer networks.

Resumo

O crescente volume de dados trafegados em redes modernas impõe desafios significativos ao desempenho e à escalabilidade de sistemas de comunicação. Nesse contexto, o uso de unidades de processamento gráfico (GPUs) para o processamento de pacotes surge como uma solução promissora, capaz de explorar o paralelismo massivo dessas arquiteturas para acelerar tarefas críticas, como roteamento, inspeção de pacotes e detecção de intrusões. Este minicurso apresenta uma introdução abrangente ao processamento de pacotes com GPUs, abordando desde fundamentos teóricos sobre redes e computação paralela até implementações práticas com tecnologias como CUDA e frameworks como

DOCA, GPUNet e PacketShader. Discutem-se também aspectos técnicos como gerenciamento de memória, balanceamento de carga e eficiência energética, além de apresentar estudos de caso aplicados e comparações de desempenho entre abordagens baseadas em CPU e GPU. A obra se propõe como uma ponte entre teoria e prática, com foco em aplicações reais e oportunidades de pesquisa. Por fim, são exploradas tendências emergentes como a integração com redes definidas por software (SDN) e a virtualização de GPUs em ambientes distribuídos, destacando o papel estratégico dessas tecnologias no futuro das redes de computadores.

1.1. Introdução

O crescimento exponencial do tráfego em redes de computadores modernas, impulsionado por aplicações de alta demanda como serviços em nuvem, inteligência artificial distribuída e análise de dados em tempo real, impõe desafios significativos ao desempenho e à escalabilidade dos sistemas de comunicação. Ao mesmo tempo, a necessidade por baixa latência e alta taxa de transferência torna cada vez mais crítica a eficiência no processamento de pacotes de rede. Nesse contexto, a utilização de Unidades de Processamento Gráficas (GPUs) para acelerar tarefas tradicionalmente executadas por CPUs tem emergido como uma abordagem promissora, explorando o paralelismo massivo das GPUs para tratar grandes volumes de dados simultaneamente [Li et al. 2020, Van Hauwaert et al. 2024].

Em função disso, o processamento de pacotes em GPUs apresenta desafios específicos relacionados à transferência de dados, ao balanceamento de carga e ao gerenciamento de memória [Huang et al. 2020], exigindo o desenvolvimento de novas arquiteturas [Sun and Ricci 2013], técnicas e frameworks que sejam capazes de integrar eficientemente esses dispositivos ao plano de dados das redes [Barney et al. 2010]. Este trabalho considera especificamente o problema de como explorar GPUs para o processamento eficiente de pacotes, com foco em tecnologias modernas como CUDA e o framework NVIDIA DOCA [NVIDIA Corporation 2022], além de soluções como GPUNet, PacketShader e DPDK [Pantuza et al. 2021a]. O objetivo é entender como tais ferramentas podem ser aplicadas, quais são suas limitações e de que forma contribuem para o avanço da área.

Neste minicurso, apresentamos uma visão abrangente do estado da arte no processamento de pacotes com uso de GPUs, com ênfase tanto nos fundamentos teóricos quanto em aplicações práticas. Discutimos a arquitetura das GPUs e seus modelos de programação, abordando os principais desafios relacionados ao paralelismo e ao gerenciamento de recursos. Também revisamos os principais frameworks e bibliotecas empregados na integração entre redes e GPUs [Sanders and Kandrot 2010]. Além disso, foram apresentados estudos de caso e comparações experimentais que demonstram os ganhos de desempenho possíveis com essas abordagens. Por fim, exploramos tendências emergentes no uso de GPUs em redes definidas por software (SDN) e em ambientes distribuídos com virtualização de GPU.

O presente trabalho tem como objetivo discutir os desafios e a interseção entre *frameworks* modernos, como DOCA, e o problema de performance no plano de dados. Ao enfatizar a viabilidade do processamento massivo e paralelo de pacotes de rede em GPU, este minicurso destaca avanços técnicos recentes bem como o impacto destas tecnologias

emergentes nos cenários das aplicações contemporâneas.

O restante deste trabalho estrutura-se da seguinte forma: A seção 1.2 discute os conceitos cruciais do processamento de pacotes. A seção 1.3 apresenta as unidades de processamento gráfico (GPUs). Na seção 1.4, são introduzidos os fundamentos de processamento paralelo. A seção 1.5 aborda frameworks, ferramentas e funcionalidades de rede baseadas em GPUs. Em seguida, a seção 1.6 trata de modelagem e implementação do processamento de pacotes em GPUs. A seção 1.7 ilustra casos de uso práticos dessa abordagem. Já a seção 1.8 discute os principais desafios enfrentados ao empregar GPUs nesse contexto. Finalmente, as conclusões e trabalhos futuros são apresentados na seção 1.9.

1.2. Processamento de Pacotes de Redes

O processamento de pacotes é um aspecto fundamental nas redes de comunicação, que envolve a análise e manipulação dos pacotes de dados enquanto transitam por uma rede. Cada pacote contém informações essenciais que precisam ser tratadas de maneira eficiente para garantir que a comunicação entre dispositivos seja realizada com rapidez e confiabilidade. O processo de manipulação dos pacotes inclui diversas etapas, como recepção, classificação, roteamento, tratamento de erros e controle. Nesta seção é apresentado uma fundamentação teórica sobre redes e processamento de pacotes. São discutidos diversas características e desafios no processamento de pacotes.

1.2.1. O que é o processamento de pacotes?

O processamento de pacotes é um conjunto de técnicas e algoritmos voltados para a manipulação eficiente de pacotes de dados que transitam pelas redes [Vieira et al. 2020]. Um pacote [Du et al. 2023a] é a unidade fundamental de transmissão de dados em redes digitais, e é composto por duas partes principais:

- Cabeçalho (header): Contém informações de controle, como os endereços IP de origem e destino, tipo de protocolo de transporte (TCP, UDP, etc.) e dados essenciais para o roteamento e gerenciamento do tráfego.
- Carga útil (payload): Representa o conteúdo real da transmissão, podendo ser uma fração de um arquivo, uma mensagem de texto ou qualquer outro tipo de dado.

O processamento eficiente desses pacotes é essencial para garantir comunicações de alta velocidade e baixa latência. Entre as principais etapas desse processo, destacam-se:

- Recepção: Captura dos pacotes conforme chegam à interface de rede (NIC).
- Classificação: Identificação e categorização dos pacotes com base em atributos como endereços IP, portas e protocolos, utilizando critérios como a classificação "5-tuple".
- Roteamento e Encaminhamento: Determinação do caminho adequado para o pacote, consultando tabelas de roteamento e aplicando políticas de gerenciamento de tráfego.

- Tratamento de Erros: Detecção e gerenciamento de falhas, como perdas de pacotes e problemas de verificação de integridade (checksum).
- Controle e Gerenciamento: Execução de funções administrativas, como protocolos de roteamento e operações de monitoramento.

O processamento de pacotes pode ser classificado em duas abordagens principais [Cerović et al. 2018]:

- Caminho Rápido (fast-path): Responsável por operações otimizadas para alto desempenho e baixa latência, como o encaminhamento e o processamento de cabeçalhos.
- Caminho Lento (slow-path): Envolve tarefas menos frequentes, como a correção de erros e a aplicação de políticas de gerenciamento, que não impactam diretamente o fluxo principal dos pacotes.

A implementação do processamento de pacotes pode ocorrer via software como o DPDK (do inglês - Data Path Development Kit) e o eBPF (do inglês - Extended Berkeley Packet Filter), via hardware FPGAs (do inglês - Field Programmable Gate Array) e ASICs (do inglês - Application-Specific Integrated Circuits) ou por meio de abordagens híbridas, que combinam ambas as técnicas para maximizar o desempenho [Bonola et al. 2022, Brunella et al. 2022, Vieira et al. 2020, Pantuza et al. 2021b].

1.2.2. Conceitos básicos de redes

Aqui é resgatado alguns conceitos básicos em redes de computadores que são importantes no desenvolvimento deste minicurso.

1.2.2.1. Modelo OSI

Com o crescimento da complexidade das redes de computadores, a International Organization for Standardization (ISO) criou o modelo OSI (Open Systems Interconnection). Esse modelo foi desenvolvido para estabelecer um conjunto de padrões de protocolos de comunicação, permitindo a interoperabilidade entre diferentes sistemas de rede.

O modelo OSI é composto por sete camadas, cada uma responsável por uma parte específica do processo de comunicação. No entanto, ele não é uma arquitetura de rede propriamente dita, pois não define quais tecnologias devem ser usadas em cada camada, apenas especifica suas funções.

A comunicação no modelo OSI é organizada nas seguintes camadas, da mais baixa para a mais alta: Física, Enlace de Dados, Rede, Transporte, Sessão, Apresentação e Aplicação [Tanenbaum and Wetherall 2010].

1. **Camada Física**

A camada física trata da transmissão dos sinais através do meio de comunicação. Seu principal objetivo é assegurar que os bits enviados por um transmissor sejam corretamente recebidos pelo receptor. Para isso, define aspectos como duração dos sinais, estabelecimento e encerramento de conexões, bem como as características físicas dos conectores de rede, como número de pinos e suas funções.

2. **Camada de Enlace de Dados**

   A camada de enlace de dados tem a função de detectar e corrigir erros que podem ter ocorrido na camada física [Day and Zimmermann 1983], garantindo que os dados cheguem à camada de rede de forma confiável. Para isso, os dados são divididos em quadros, enviados sequencialmente e confirmados pelo receptor. Além disso, essa camada pode implementar controle de fluxo para evitar sobrecarga do receptor caso a transmissão ocorra mais rápido do que ele pode processar.

   Em redes compartilhadas, essa camada inclui a subcamada Controle de acesso ao meio (MAC - do inglês Medium Access Control) [Association et al. 1997], que regula o acesso ao canal de comunicação para evitar colisões entre transmissões de diferentes dispositivos.

3. **Camada de Rede**

   A camada de rede é responsável pelo roteamento dos pacotes da origem ao destino, determinando o melhor caminho entre os dispositivos, mesmo que isso envolva várias redes intermediárias. As rotas podem ser estáticas, definidas manualmente, ou dinâmicas, ajustando-se automaticamente conforme mudanças na topologia da rede, como falhas em equipamentos.

   Embora o controle direto de congestionamento seja atribuição principal da camada de transporte, a camada de rede pode contribuir com mecanismos auxiliares, como a escolha de rotas menos congestionadas ou o envio de sinais de advertência.

   Além disso, essa camada enfrenta desafios como a compatibilização de esquemas de endereçamento, a fragmentação e remontagem de pacotes quando há diferenças no tamanho máximo de transmissão (MTU), e a interconexão de redes com tecnologias e protocolos distintos.

4. **Camada de Transporte**

   A camada de transporte, situada acima da camada de rede no modelo OSI, tem como principal responsabilidade assegurar a comunicação fim a fim entre aplicações situadas em dispositivos distintos. Seu propósito central é isolar as camadas superiores das particularidades da infraestrutura subjacente, promovendo uma transferência de dados eficiente, independente da topologia ou tecnologia da rede.

   Essa camada pode prover diferentes tipos de serviço à camada de sessão, sendo o mais comum um canal confiável e orientado à conexão, que garante a entrega ordenada e sem duplicação dos dados. No entanto, também é possível a oferta de serviços não confiáveis, nos quais não há garantias quanto à ordem ou à entrega dos dados, o que pode ser útil em aplicações que priorizam desempenho e baixa latência.

Diferentemente das camadas inferiores, que operam entre nós adjacentes (hop-by-hop), a camada de transporte atua de forma ponta a ponta (end-to-end), assegurando a comunicação direta entre a origem e o destino da informação.

5. **Camada de Sessão**

   A camada de sessão permite a criação e o gerenciamento de sessões entre dispositivos. Essas sessões oferecem funcionalidades como controle de concorrência (evitando que múltiplos usuários realizem operações críticas simultaneamente) e sincronização (possibilitando a retomada de processos após falhas no sistema).

6. **Camada de Apresentação**

   A camada de apresentação tem como principal função garantir que os dados transmitidos sejam compreendidos corretamente pelo sistema de destino, mesmo que os dispositivos envolvam formatos diferentes [Day and Zimmermann 1983]. Suas responsabilidades incluem a conversão de formatos de dados, a compressão para otimizar o tráfego e a criptografia para assegurar a confidencialidade das informações.

7. **Camada de Aplicação**

   A camada de aplicação tem como principal foco a semântica, fornecendo os protocolos que permitem a interação entre os usuários e os serviços de rede. Um dos mais conhecidos é o HTTP (HyperText Transfer Protocol), que permite a solicitação e transferência de páginas web, sendo fundamental para o funcionamento da World Wide Web.

1.2.2.2. Arquitettura TCP/IP

A arquitetura TCP/IP recebe este nome devido aos principais protocolos utilizados nesta arquitetura, o TCP(Transmission Control Protocol ou Protocolo de Controle de Transmissão) e o IP (Internet Protocol). O objetivo desta arquitetura é construir uma interligação de redes, permitindo a comunicação entre diferentes usuários em diferentes dispositivos, mesmo que separados geograficamente por uma grande área [Rodriguez et al. 2001].

O protocolo TCP é um protocolo fundamental da camada de transporte e foi projetado para estabelecer uma comunicação confiável e eficiente de ponta a ponta, mesmo em redes que podem apresentar falhas, atrasos e variações nos tamanhos dos pacotes. Sua função é lidar dinamicamente com essas incertezas para garantir a integridade dos dados transmitidos.

A arquitetura TCP/IP é composta por quatro camadas, organizadas da mais baixa para a mais alta: Enlace, Rede, Transporte e Aplicação [Tanenbaum and Wetherall 2010].

1. **Camada de Enlace**

   A camada de enlace, também chamada de interface de rede, é responsável pela transmissão física dos dados entre os dispositivos. Seu papel é estabelecer a interface com o hardware de rede, como conexões Ethernet ou linhas seriais, garantindo

que os pacotes possam ser enviados e recebidos corretamente. Diferente do modelo OSI, o TCP/IP não especifica protocolos exclusivos para essa camada, permitindo o uso de diferentes tecnologias de enlace de acordo com a infraestrutura disponível.

2. **A Camada de Rede**

   A camada de Rede [Forouzan and Fegan 2006], também conhecida como camada de Internet, projeta uma visão virtual da rede para suas camadas superiores, buscando proteger as camadas superiores das arquiteturas físicas de camadas inferiores, enquanto promove a definição de um esquema padronizado para o envio de pacotes de dados. Para isso, utiliza o protocolo IP (Internet Protocol), que transforma as informações em datagramas contendo um cabeçalho com os endereços de origem e destino, além da carga útil (payload) com os dados transmitidos.

   O IP é um protocolo sem conexão e não confiável, pois realiza a transmissão dos pacotes sem garantir que eles cheguem na ordem correta ou sem erros. Ele utiliza o princípio do melhor esforço (best-effort), o que significa que os pacotes podem chegar em sequência diferente, duplicados ou até mesmo se perderem durante a transmissão. O gerenciamento dessas inconsistências fica a cargo das camadas superiores. Além disso, a camada de rede também é responsável pelo roteamento, garantindo que os pacotes encontrem o melhor caminho até seu destino.

3. **A Camada de Transporte**

   Acima da camada de rede está a camada de transporte, cuja função é permitir que entidades pares de origem e destino mantenham uma comunicação, assim como na camada de transporte do modelo OSI.

   Dois protocolos de transporte principais operam nesta camada, TCP e UDP.

   O primeiro deles, o TCP (Transmission Control Protocol ou Protocolo de Controle de Transmissão), é um protocolo confiável, orientado à conexão, que permite que um fluxo de bytes originado em uma máquina seja entregue sem erros e em ordem a qualquer outra máquina na Internet. O TCP segmenta o fluxo de bytes de entrada em mensagens discretas e repassa cada uma delas à camada de Internet. No destino, o processo TCP receptor reorganiza as mensagens recebidas para reconstruir o fluxo original.

   Além disso, o TCP implementa controle de fluxo, garantindo que um transmissor rápido não sobrecarregue um receptor mais lento com dados em excesso. O protocolo também realiza controle de congestionamento, ajustando dinamicamente a taxa de envio de dados com base nas condições da rede, a fim de evitar sobrecarga e perda de pacotes

   O segundo protocolo nesta camada, o UDP (Protocolo de Datagramas de Usuário), é um protocolo não confiável e sem conexão para aplicativos que não desejam a sequência ou o controle de fluxo do TCP e preferem fornecer seus próprios mecanismos, apresentando menor latência na comunicação. Ele também é amplamente utilizado em consultas tipo cliente-servidor, de solicitação-resposta, e em aplicações nas quais a entrega rápida é mais importante que a entrega precisa, como na transmissão de voz ou vídeo.

4. **A Camada de Aplicação**

   A camada de aplicação é usualmente responsável pelos programas e protocolos que possibilitam o TCP/IP dar início a transmissão de dados. Essa camada será responsável por determinar a finalidade específica da transmissão de dados.

   O modelo TCP/IP não possui camadas de sessão ou apresentação como no modelo OSI. Em vez disso, as aplicações simplesmente incluem qualquer função de sessão e apresentação que precisem. A experiência com o modelo OSI provou que essa visão está correta: essas camadas têm pouca utilidade para a maioria das aplicações [Tanenbaum and Wetherall 2010].

1.2.2.3. Roteamento

O roteamento é uma das principais funções da camada de rede do modelo OSI (Open Systems Interconnection) e consiste na formação de conexões entre diferentes redes físicas [Rodriguez et al. 2001], com o objetivo de entregar os pacotes de forma eficiente, reduzindo atrasos, lidando com falhas e congestionamentos.

O algoritmo de roteamento é responsável por definir qual será a rota utilizada entre os dispositivos da rede (roteadores) disponíveis, identificando os caminhos possíveis para alcançar diferentes destinos. Essa informação é armazenada nas tabelas de roteamento. Um roteador executa dois processos fundamentais: roteamento e encaminhamento. O processo de roteamento é responsável por determinar as melhores rotas para os pacotes, construindo e atualizando a tabela de roteamento. Já o encaminhamento utiliza essa tabela para analisar os pacotes recebidos e direcioná-los corretamente ao seu destino final ou ao próximo roteador no caminho [Tanenbaum and Wetherall 2010].

O roteamento pode ser dividido entre estático e dinâmico:

- Estático: O processo de roteamento não considera mudanças sobre a topologia ou o tráfego [Tanenbaum and Wetherall 2010], sua rota é definida manualmente pelo administrador da rede. Este modelo pode ser útil para pequenas redes, ou até mesmo em caráter de teste [Rodriguez et al. 2001], mas não para grandes redes, visto que pode ser frágil por não responder à falhas.
- Dinâmico: O roteamento passa a considerar medições e estimativas, como topologia e tráfego, para mudar suas rotas e atualizar sua tabela de roteamento dinamicamente [Forouzan and Fegan 2006].

Os algoritmos de roteamento dinâmico variam de acordo com a origem das informações utilizadas, que podem ser de visão local (informações próprias ou de vizinhos) ou de visão global (informações de todos os roteadores da rede), o momento em que as rotas são alteradas (se após mudanças na topologia ou periodicamente), e as métricas adotadas para otimização, como distância, número de saltos e tempo estimado de tráfego, entre outras.

Os protocolos de roteamento, como os protocolos do tipo vetor de distância e roteamento por estado de enlace, são responsáveis por determinar a melhor rota, além de

definir o que constitui a "melhor"rota, levando em consideração diferentes métricas como custo, largura de banda ou tempo de resposta [Forouzan and Fegan 2006].

O RIP [Hedrick 1988] (Routing Information Protocol) é um protocolo de roteamento baseado em vetores de distância onde a rota de menor custo é aquela com menos saltos (hops) e que cada nó mantém uma tabela com as distâncias mínimas para os outros nós da rede. Por exemplo, uma estratégia possível é atribuir um custo a uma passagem por rede. O RIP, especificamente, considera todas as redes como iguais, logo, se um pacote passa por 10 redes, (realizando 10 hops), seu custo total será de 10 hops.

Já o OSPF [Sidhu et al. 1993] (Open Shortest Path First) faz parte dos Protocolos de Roteamento de Estado de Enlace (Link) onde cada roteador constrói um mapa completo da topologia permitindo ao administrador configurar um custo de passagem pela rede com base nos serviços exigidos da rede e calcula as melhores rotas, usando algoritmos como Dijkstra [Dijkstra 1965] e, por exemplo, caso a velocidade seja crucial, uma conexão de fibra óptica terá um custo menor que uma conexão via satélite.

1.2.3. Desafios

O crescimento contínuo do tráfego na Internet, aliado à evolução dos serviços em redes de data centers, impõe a necessidade de um processamento de pacotes cada vez mais ágil e eficiente [Vieira et al. 2020]. Embora os avanços em hardware de rede e nas comunicações sem fio tenham elevado significativamente as taxas de transmissão, o desempenho do processamento de pacotes não tem acompanhado esse ritmo [Cerović et al. 2018]. Isso se deve, em grande parte, às limitações dos métodos tradicionais, que enfrentam diversos gargalos ao lidar com o volume e a complexidade crescentes do tráfego de rede.

O modelo convencional de processamento de pacotes segue cinco etapas principais [Du et al. 2023b]:

1. Recepção do pacote: Quando um pacote chega a uma placa de interface de rede (Network Interface Card - NIC), ocorre uma interrupção e a transferência do conteúdo para a memória é realizada via Direct Memory Access (DMA).

2. Manipulação da interrupção: O processador atende à interrupção acionando a função $netif\_rx()$, que cria uma estrutura sk_buff para representar o pacote recebido.

3. Encaminhamento para processamento: O pacote é inserido em uma fila para ser tratado posteriormente.

4. Processamento no kernel: O manipulador de interrupção por software processa o pacote na fila e o repassa para a pilha de rede do kernel.

5. Transferência para o espaço do usuário: A aplicação finalmente acessa o pacote, transferindo-o do espaço do kernel para o espaço do usuário, onde será processado.

Esse fluxo tradicional apresenta desafios significativos, especialmente em cenários de alta taxa de pacotes. Como cada pacote recebido gera uma interrupção, quando a frequência de pacotes é elevada, o processador pode gastar mais tempo respondendo a

interrupções do que realmente processando os dados [Du et al. 2023b]. Além disso, a necessidade de copiar pacotes entre diferentes áreas de memória representa um custo adicional [Cerović et al. 2018].

O uso de NICs tem sido uma abordagem comum para lidar com o processamento de pacotes, mas essa solução também apresenta limitações em ambientes de alta velocidade. Quando os pacotes chegam a uma NIC, eles são distribuídos em filas distintas e processados por diferentes núcleos do processador. No entanto, essa distribuição pode gerar desafios relacionados ao Cross-core scheduling [Du et al. 2023b] como:

- Atrasos na transferência de dados: Diferentes threads podem ser alocadas em diferentes núcleos, causando atrasos ao transferir pacotes entre caches locais e aumentando o tempo de acesso aos dados.
- Sobrecarga no gerenciamento de threads: O sistema operacional precisa coordenar quais pacotes serão processados por quais núcleos, aumentando o custo computacional.
- Desbalanceamento de carga: Alguns núcleos podem ficar sobrecarregados, enquanto outros permanecem subutilizados, reduzindo a eficiência do processamento.

Diante dessas dificuldades, novas abordagens têm sido desenvolvidas para otimizar o processamento de pacotes e minimizar gargalos, permitindo um melhor aproveitamento dos recursos computacionais disponíveis.

1.3. Introdução à Computação com GPU

Uma Unidade de Processamento Gráfico (GPU) é um dispositivo programável projetado para funcionar como um coprocessador, recebendo comandos, código e dados da CPU para executar tarefas altamente paralelas e intensivas em cálculo. Atuando como um aliviador de carga, a GPU acelera a solução de problemas complexos por meio da execução simultânea de cálculos matemáticos simples em larga escala. Esta seção apresenta uma visão geral da arquitetura das GPUs modernas e discute seu uso em aplicações de alto desempenho.

1.3.1. Arquitetura

As GPUs foram desenvolvidas para executar milhares de threads em paralelo, reduzindo o custo associado à execução de uma única thread e aumentando a taxa de transferência de dados (throughput).

A Figura 1.1 ilustra a arquitetura típica de uma GPU, composta por múltiplos núcleos, níveis hierárquicos de memória cache integrados no chip e uma memória global de alta largura de banda separada, referida como Memória de Acesso Aleatório de Vídeo (VRAM - do inglês Video Random Access Memory).

As GPUs modernas, apresentam arquiteturas paralelas compostas por um conjunto de Multiprocessadores de Streaming (SMs - do inglês Streaming Multiprocessor), projetados para executar operações de maneira simultânea e eficiente. Na Figura 1.1, uma linha

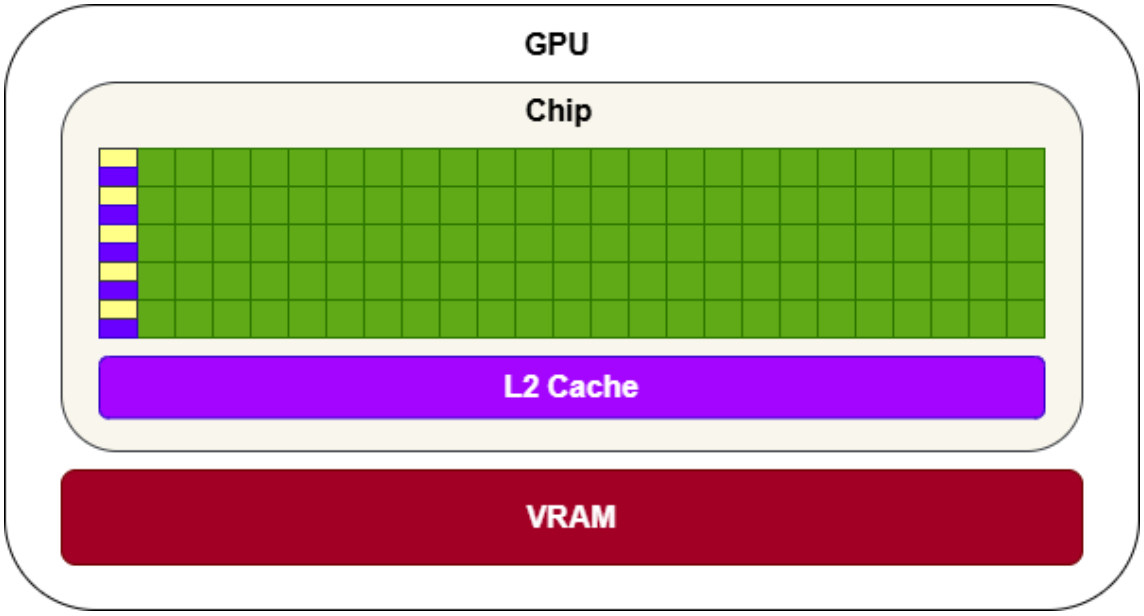

Figura 1.1. Esquema simplificado da arquitetura de uma GPU em hardware. As memórias caches estão em roxo, os núcleos em verde e os registradores de controle em amarelo.

de núcleos representa um único SM, que funciona como um processador independente e adota o modelo Single Instruction Multiple Threads (SIMT). Cada SM é equipado com um número fixo de núcleos especializados em operações específicas, como aritmética de ponto flutuante e aritmética inteira, além de múltiplos escalonadores, caches de dados L1, memória compartilhada e um conjunto de registradores de 32 bits.

A unidade básica de execução é um grupo de 32 threads, denominado warp ou wave. Um warp executa uma única instrução de forma síncrona para todas as suas threads, promovendo alta eficiência quando os fluxos de controle são semelhantes. Cada multiprocessador distribui os warps entre seus escalonadores e, a cada ciclo de emissão de instruções, cada escalonador despacha uma instrução para um dos warps atribuídos, desde que estejam prontos para execução. Uma característica fundamental dessa arquitetura é que a alternância de execução entre warps ocorre sem custo adicional, pois o contexto de execução de cada warp é mantido diretamente no chip durante toda a sua vida útil.

1.3.2. Processamento Paralelo: CPU vs GPU

Com as constantes melhorias no desempenho dos computadores, um número cada vez maior de aplicações com diferentes tarefas e demandas tem surgido rapidamente. Como consequência, o desenvolvimento de arquiteturas paralelas foi impulsionado, e os processadores passaram a evoluir em duas direções principais: os processadores de propósito geral (CPUs - do inglês Central Processing Unit) e os processadores de propósito específico (como as GPUs - do inglês Graphics Processing Unit). Embora ambos os tipos sejam capazes de executar tarefas paralelas, suas arquiteturas distintas os tornam mais eficientes para diferentes tipos de operações.

A Unidade Central de Processamento (CPU) é o componente principal respon-

sável por executar instruções e coordenar as operações de um sistema computacional. Projetada para lidar com uma ampla variedade de tarefas, a CPU é altamente eficiente na execução de operações sequenciais, sendo ideal para aplicações com lógica de controle complexa. As CPUs modernas têm aumentado seu desempenho historicamente conforme a Lei de Moore [Moore 1965], por meio do aumento da frequência de clock e da densidade de transistores. Esses transistores são responsáveis não apenas pela interpretação, execução e finalização de instruções aritméticas e lógicas, mas também pela coordenação e controle de diversos componentes do sistema.

Por outro lado, as GPUs são projetadas especificamente para computações intensivas e altamente paralelas. Em sua arquitetura, uma proporção significativamente maior de transistores é dedicada ao processamento de dados, em vez de à manipulação de cache ou controle de fluxo. Enquanto a CPU é especializada em desempenho por núcleo e tomada de decisões complexas, a GPU é otimizada para maximizar o throughput em tarefas altamente paralelizáveis, sacrificando a versatilidade em favor da eficiência em cargas de trabalho específicas, como processamento de imagens, aprendizado de máquina e simulações científicas.

Em diversos problemas com alta demanda computacional, as GPUs apresentam vantagens significativas em relação às CPUs [Wang et al. 2008]. No entanto, devido à natureza específica de sua arquitetura, nem todas as tarefas computacionais podem ser executadas exclusivamente na GPU. Instruções com forte dependência de controle ou com características essencialmente sequenciais ainda precisam ser processadas pela CPU.

Dessa forma, a GPU deve ser vista menos como uma substituta e mais como uma colaboradora da CPU. O verdadeiro potencial da computação moderna reside na utilização conjunta desses dois tipos de unidades de processamento, explorando suas forças complementares em um ambiente de computação heterogêneo e colaborativo. Essa abordagem, conhecida como GPGPU (General-Purpose computing on Graphics Processing Units), representa uma estratégia promissora para a resolução eficiente de problemas genéricos e complexos.

A Figura 1.2 ilustra visualmente as diferenças arquiteturais entre CPU e GPU. Observa-se que, enquanto a CPU possui poucos núcleos robustos voltados para tarefas de propósito geral, a GPU conta com centenas ou milhares de núcleos mais simples, organizados para permitir processamento paralelo massivo.

A Tabela 1.1 apresenta uma comparação entre as duas arquiteturas, destacando suas principais características e aplicações típicas.

1.3.3. Modelo de Programação

Embora as GPUs tenham sido originalmente criadas para realizar cálculos gráficos, elas também permitem acelerar o processamento de propósito geral. Essa abordagem é conhecida como computação de propósito geral em GPUs (GPGPU — do inglês General-Purpose computing on GPUs). Para desenvolver programas que executam em uma GPU, é necessário utilizar um framework de programação, ou seja, ambientes de desenvolvimento apropriado.

Uma das opções disponíveis é o **OpenCL** [The Khronos Group Inc. 2024], um

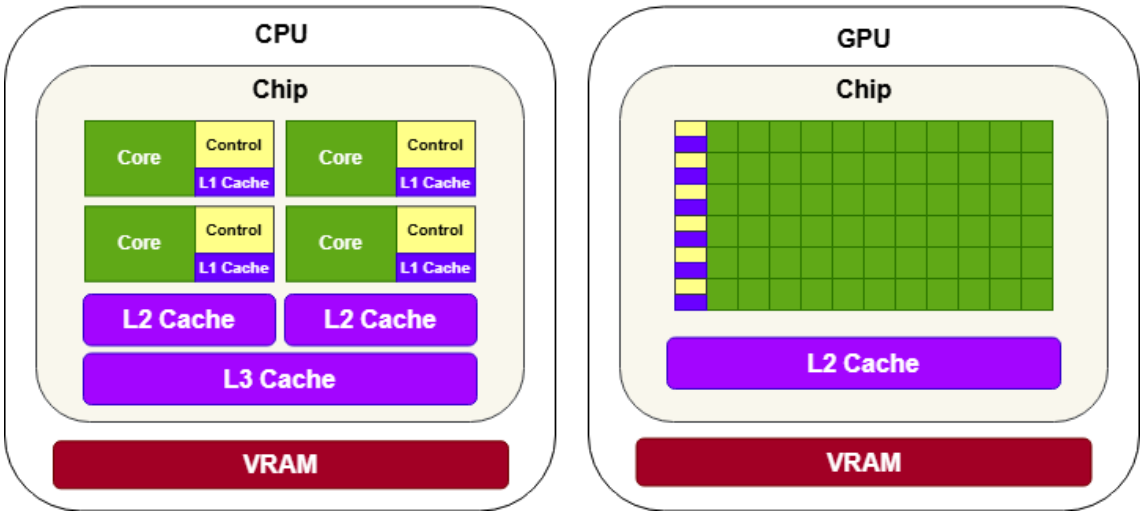

Figura 1.2. Comparação entre a arquitetura da CPU e da GPU em hardware.

padrão aberto que permite escrever aplicações paralelas para diferentes tipos de hardware, incluindo GPUs, CPUs e outros tipos de processadores. Ele é baseado na linguagem C e é compatível com diversas plataformas, como Intel, AMD, NVIDIA e ARM, o que o torna uma solução versátil para programação heterogênea.

Outra alternativa amplamente utilizada é o **CUDA** [NVIDIA Corporation 2024b], uma plataforma e modelo de programação proprietários da NVIDIA, desenvolvidos especificamente para suas GPUs. Atualmente, trata-se da principal tecnologia empregada para GPGPU. O ambiente de desenvolvimento baseia-se em extensões da linguagem C++ e oferece suporte tanto ao modelo de programação próprio quanto ao OpenCL. Além disso, disponibiliza bibliotecas otimizadas, ferramentas de depuração e recursos para análise de desempenho. O *CUDA C++ Programming Guide* [NVIDIA Corporation 2024a] é a principal referência técnica para o uso dessa plataforma.

No contexto do CUDA, as funções executadas na GPU são chamadas de kernels, e sua execução ocorre de forma massivamente paralela: ao serem chamadas, as kernels são executadas por múltiplas threads, cada uma com um identificador único (thread ID), o que permite comportamentos distintos.

Essas threads são agrupadas em blocos (blocks), que por sua vez compõem grades (grids), ambos com identificadores próprios. O programador pode definir a quantidade de threads por bloco e de blocos por grade, em até três dimensões. As threads devem operar de forma independente entre si, o que permite que sejam alocadas dinamicamente entre os diferentes multiprocessadores da GPU (SMs - do inglês Streaming Multiprocessors). A organização dessa hierarquia é ilustrada na Figura 1.3.

Durante sua execução, as threads acessam diferentes tipos de memória. Cada uma possui um espaço de memória local exclusivo e seus próprios registradores. Threads pertencentes ao mesmo bloco podem compartilhar uma memória comum, disponível apenas durante a execução daquele bloco. Além disso, todas as threads podem acessar a memória global da GPU. Com a introdução da Memória Unificada (Unified Memory), tornou-se possível utilizar um espaço de endereçamento comum entre CPUs e GPUs, eliminando a

Tabela 1.1. Comparação entre processamento de pacotes em CPU e GPU

Critério	CPU (Unidade Central de Processamento)	GPU (Unidade de Processamento Gráfico)
Arquitetura	Poucos núcleos otimizados para tarefas sequenciais e controle de fluxo	Centenas ou milhares de núcleos otimizados para execução paralela massiva
Paralelismo	Limitado (tipicamente 4–64 threads)	Massivo (milhares de threads simultâneas)
Latência de Resposta	Baixa latência em tarefas simples ou com controle complexo	Alta performance em tarefas paralelas; pode haver latência em cargas pequenas
Vazão (Throughput)	Limitada pela quantidade de núcleos e frequência	Alta vazão: pode processar milhões de pacotes por segundo
Consumo de Energia	Relativamente baixo	Mais elevado, especialmente em cargas intensivas
Facilidade de Desenvolvimento	Ambientes de programação bem estabelecidos (C, Go, Python, etc.)	Requer conhecimento em CUDA/OpenCL e paralelismo de dados
Escalabilidade	Limitada ao número de núcleos físicos e threads	Alta escalabilidade com milhares de threads paralelas
Gerenciamento de Memória	Simples; acesso direto à memória principal	Exige cópia entre memória do host e da GPU
Custo de Hardware	Menor custo unitário; geralmente já disponível	Maior investimento inicial em placas especializadas
Casos de Uso Recomendados	Tráfego moderado e lógica de controle complexa	Ambientes de alta performance com alto volume de tráfego

necessidade de cópias explícitas de dados entre os dispositivos.

As operações executadas pela GPU, como a chamada de kernels ou a movimentação de dados, podem ser organizadas em streams. As instruções em uma mesma stream são executadas em ordem, respeitando a sequência em que foram emitidas. No entanto, diferentes streams podem ser executadas simultaneamente no dispositivo, o que permite paralelismo em nível de operações.

Outro recurso que contribui para a sobreposição de tarefas é o uso de memória do host fixada em página (paged-locked memory). GPUs mais modernas conseguem, por exemplo, iniciar uma kernel ao mesmo tempo em que transferem dados de forma assíncrona, ou realizar transferências simultâneas de entrada e saída entre host e dispositivo. Esse tipo de paralelismo é especialmente útil em cenários em que novos dados precisam ser processados continuamente, sem aguardar a finalização completa de operações anteriores.

Além disso, o modelo CUDA permite o uso de eventos dentro das streams, que

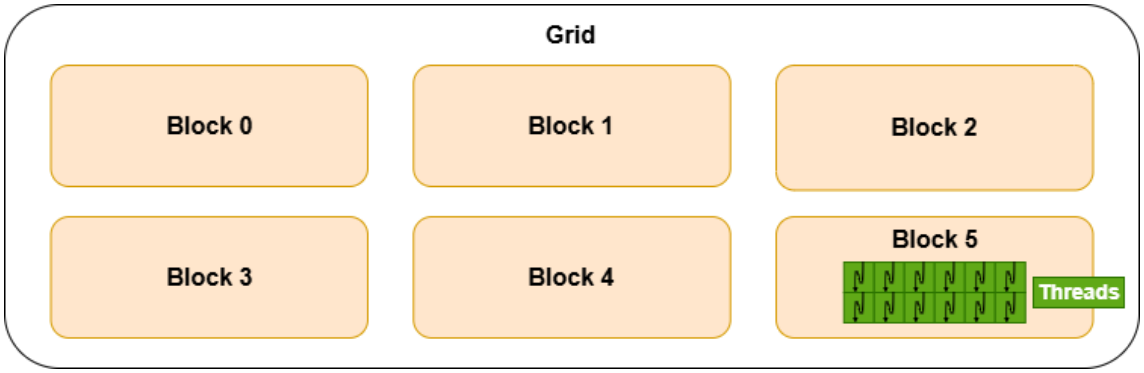

Figura 1.3. Hierarquia de threads na GPU. O grid contém 3 x 2 blocos e cada bloco contém 6 x 2 threads.

funcionam como marcadores temporais. Esses eventos permitem monitorar a progressão das operações tanto no dispositivo quanto no host, possibilitando mecanismos de sincronização entre diferentes kernels ou entre CPU e GPU.

1.4. Fundamentos de Processamento Paralelo

O processamento paralelo [Hwang and Briggs 1990] é uma técnica computacional que permite a execução simultânea de múltiplas tarefas, com o objetivo de aumentar a vazão de tarefas concluídas e, consequentemente, reduzir o tempo de processamento. Essa abordagem baseia-se na divisão de um problema em subproblemas menores, que podem ser resolvidos de forma concorrente por diferentes unidades de processamento.

1.4.1. Introdução

A lei de Moore [Moore 1965] previa que o número transistores em um circuito integrado dobraria a cada ano. Porém, tal lei está chegando a seu fim, visto que a medida que os transistores ficam menores a quantidade de corrente vazada aumenta, o que prejudica o consumo de energia. Além disso, a quantidade de potência dissipada aumenta com o quadrado da frequência do clock. Tais problemas impossibilitam a evolução da computação através apenas do aumento da frequência do clock.

Geer et al. [Geer 2005] explicam que, para superar essa estagnação e aumentar o desempenho dos processadores, a indústria adotou modelos multi-core, impulsionando o processamento paralelo, que se mostrou eficiente para aplicações de alto desempenho. Assim, as GPUs, que foram idealizadas para processamento gráfico, atualmente estão sendo usadas para propósito geral. Como exemplo há o PacketShader [Han et al. 2010], um framework de alto desempenho para roteadores em software, que acelera o processamento geral de pacotes utilizando GPUs.

Em clusters e super-computadores o uso de computação paralela é amplo. Notoriamente, há o supercomputador japonês Fugaku [Sato et al. 2020], que realiza simulações em medicina e previsão climática. Ele foi construído com mais de 150 mil processadores Fujitsu A64FX, que têm 48 núcleos cada e que implementam um conjunto de instrução SIMD, que permite a execução paralela da mesma instrução em múltiplos dados.

1.4.2. Modelos de programação paralela

O desenvolvimento de aplicações paralelas requer abstrações que auxiliem na classificação precisa de arquiteturas paralelas, bem como na comunicação e sincronização precisa entre múltiplas tarefas [Hwang and Briggs 1990]. Dessa forma, através desses modelos torna-se possível projetar algoritmos paralelos e explorar de forma eficiente os recursos computacionais disponíveis.

1.4.2.1. Taxonomia de Flynn

A **taxonomia de Flynn** [Flynn 1966] é um dos principais esquemas de classificação para arquiteturas de computação paralela e indica quatro modelos principais:

- **Instrução Única, Dado Único (SISD)**: Computadores que executam uma única instrução por vez em um dado específico. Processadores escalares são classificados como SISD, um exemplo deles é o Intel 486 [Fu et al. 1989].

- **Instrução Única, Múltiplos Dados (SIMD)**: Máquinas que executam a mesma instrução simultaneamente em múltiplos dados, como em cálculos vetoriais.

  Seiler et al [Seiler 2018] utilizaram a multiplicação rápida do AVX2, uma extensão SIMD para o conjunto de instruções x86, para aprimorar a criptografia baseada em Redes Ring-LWE. Essa criptografia consiste em resolver um sistema polinomial com coeficientes inteiros, reduzidos módulo um número primo, e com erros nas respostas, tornando o problema altamente resistente a ataques quânticos.

- **Instrução Única, Múltiplas Threads (SIMT)**: Nesse submcategória do modelo SIMD cada thread tem seu próprio contador de programa e pilha, por isso as execuções podem divergir, o que o torna assíncrono e mais flexível que o modelo SIMD.

  Liu et al. [Liu et al. 2010] apresentaram uma versão otimizada do algoritmo Smith-Waterman baseado na abstração SIMT, que é amplamente utilizado no sequenciamento de DNA e proteínas. O algorítimo se fundamenta no paradigma da programação dinâmica e tem como objetivo determinar regiões similares de duas sequências. Os resultados são alcançados ao comparar segmentos de tamanhos variados enquanto otimiza a medida de similaridade.

- **Múltiplas Instruções, Dado Único (MISD)**: Diferentes instruções executadas em paralelo no mesmo dado. Halaas et al. [Halaas et al. 2004] demonstram que, por meio de uma arquitetura MISD, o modelo pode ser empregado para realizar múltiplos reconhecimentos de padrões em bases de dados.

- **Múltiplas Instruções, Múltiplos Dados (MIMD)**: Executa instruções diferentes de forma assíncrona em diversos conjuntos de dados. Um exemplo de máquina MIMD foi o ultracomputador Nyu [Gottlieb et al. 1983], um pioneiro em computação paralela que possuía memória compartilhada e era composto por milhares de elementos de processamento autônomo.

1.4.2.2. Comunicação em sistemas paralelos

Memória compartilhada possibilita que múltiplos processadores troquem informações por meio de uma região de memória comum, o que permite uma comunicação rápida por não envolver cópias explicitas e alocações de buffers. Entretanto, esse modelo exige sincronização para que não ocorra complicações, como o uso de mutexes e semáforos para evitar condições de corrida e inconsistência de dados.

A OpenMP [Dagum and Menon 1998] é uma API multiplataforma para a criação de programas paralelos que utilizam memória compartilhada e está disponível em C, C++ e Fortran. Além disso, a simplicidade da API possibilita a criação de programas paralelos sem a complexidade de gerenciar threads e sincronizações.

Outra solução para o modelo de memória compartilhada, é a plataforma de computação paralela CUDA (Compute Unified Device Architecture), criada pela NVIDIA, focada em processamento paralelo de proposito geral em GPUs[Sanders and Kandrot 2010]. O CUDA oferece uma API para que desenvolvedores possam gerenciar a execução de kernels, que são funções executadas na GPU. Devido a essa interface flexível e poderosa, o CUDA é muito utilizado em diversas áreas, como classificação de pacotes em alto desempenho [Zhou et al. 2014].

Troca de mensagens possibilita os processadores se comunicarem sem compartilhar recursos, como memória. Essa independência facilita escalabilidade, sendo ideal para clusters e supercomputadores. Vale ressaltar que essa troca de mensagens sofre de latência adicional comparada ao modelo de memória compartilhada.

Nesse modelo, a sincronização é dependente dos requisitos da aplicação e pode ser feita com barreiras, algorítimos de exclusão mútua distribuídas e protocolos de ordenação de mensagens. O algorítimo de Lamport [Lamport 2019], amplamente usado em sistemas distribuídos, garante a ordenação causal, isso é, caso uma mensagem influencie outra, a mensagem influenciadora será entregue antes da influenciada.

Para o modelo de troca de mensagens, existe o padrão Message Passing Interface (MPI) [Walker and Dongarra 1996] que é comumente usado em computação paralela e distribuída. Ademais, esse padrão está disponível em diversas arquiteturas e linguagens de programação, o que facilita a escalabilidade de clusters heterogêneos.

1.4.2.3. Coordenação em memória compartilhada

Condições de corrida são, de acordo com Mazieiro [Maziero 2014], erros dinâmicos que podem acontecer quando mais de uma tarefa acessa um recurso compartilhado ao mesmo tempo, como uma área de memória compartilhada entre threads.

Esses erros podem acontecer, por exemplo, quando um contador é compartilhado entre threads e não exista mecanismos de exclusão mútua. Se duas threads tentam ler incrementar o contador ao mesmo tempo, elas podem ler e incrementar a partir do mesmo valor, resultando em um único aumento que gera uma contagem incorreta.

Outra maneira de definir as condições de corrida são as condições de Berns-

tein [Bernstein 1966], que definem dois cenários que podem ocasionar uma condição de corrida: uma tarefa escrever em um recurso compartilhado enquanto outra lê e duas tarefas escreverem no mesmo recurso ao mesmo tempo. Dessa forma, fica evidente que leituras concorrentes em um mesmo recurso não geram condições de corrida.

Seções críticas são áreas de código que modificam dados compartilhados e que podem gerar condições de corrida. Para evitar esses problemas, **exclusão mútua** assegura que apenas uma tarefa esteja presente em seções criticas a cada instante. Mecanismos de exclusão mútua sofisticados fazem uso amplo de operações atômicas, que em base são operações indivisíveis que ocorrem sem a interrupção do sistema operacional. Dessa forma, existem três principais métodos para exclusão mútua:

- **Semáforo**: Criado por Dijkstra [Dijkstra 1965], permite exclusão mútua entre múltiplas tarefas. Essa estrutura se resume a uma fila de tarefas e um contador que representa os recursos disponíveis. Esses recursos são acessíveis através de duas operações atômicas: down e up. Down decrementa o contador e suspende a tarefa, que fica na fila de tarefas, caso não haja recursos disponíveis. Up incrementa o contador e acorda a primeira tarefa disponível na fila de tarefas caso ela exista.

- **Mutex**: Mutexes são versões simplificadas dos semáforos, as vezes chamados de semáforos binários, que liberam acesso a seção crítica à apenas uma tarefa e que também definem seu acesso com duas funções, lock e unlock, que respectivamente tem o mesmo comportamento das operações down e up.

- **Variáveis de condição**: Além da exclusão mútua, algumas tarefa precisam de condições para serem executadas. Para isso, utiliza-se variáveis de condição, que permitem que tarefas sejam suspensas com a operação wait e sejam reativadas com a operação signal(acordar uma tarefa) ou notify(acordar todas as tarefas) quando alguma condição foi alcançada. Essas variáveis devem ser usadas junto a mutexes para garantir a exclusão mútua, uma vez que as tarefas revogam o acesso a seção crítica quando são desativadas.

1.4.3. Unidades de processamento

Nos contextos de programação paralela, as unidades de processamento representam as entidades responsáveis por executar tarefas simultaneamente. Elas podem ser implementadas e gerenciadas tanto pelo sistema operacional quanto diretamente pelo hardware, dependendo da arquitetura e do nível de abstração.

Processos, para Maziero [Maziero 2014], são mais do que apenas instâncias de um programa; eles representam uma unidade isolada de contexto compartilhado por uma ou mais tarefas. Esse contexto inclui aspectos de software (estado da execução), hardware (registradores, pilha etc.) e o espaço de endereçamento disponível.

Threads são sequências de instruções que executam independentemente dentro do mesmo processo, compartilhando seu espaço de endereçamento e contexto de software. No entanto, cada thread possui sua própria pilha e conjunto de registradores, permitindo a execução paralela por meio de memória compartilhada.

Diferentemente das threads, os processos possuem memória isolada e se comunicam por meio da troca de mensagens. Esse isolamento garante que, caso um processo falhe, os demais continuem em execução normalmente. Por outro lado, se uma thread falhar, o processo inteiro pode ser comprometido.

Warps são, na arquitetura NVIDIA, um conjunto de 32 threads que executam paralelamente em um multiprocessador, seguindo o modelo SIMT, e são gerenciadas diretamente pela GPU. Warps são agrupadas em conjuntos chamados de blocos, que possuem memória compartilhada, o que permite a comunicação entre suas threads e outras warps.

1.4.4. Técnicas de paralelização

Para explorar todo o potencial da execução paralela, é necessário adotar técnicas que permitam decompor e organizar as tarefas de forma eficiente, visando minimizar seções críticas, evitar o desbalanceamento de carga entre as tarefas e reduzir a sobrecarga associada à criação e ao gerenciamento das unidades de execução.

1.4.4.1. Tipos de Paralelismo

O paralelismo é uma técnica fundamental para aumentar o desempenho de sistemas computacionais, explorando a execução simultânea de múltiplas tarefas. Existem diferentes formas de paralelismo, cada uma adequada a contextos específicos de aplicação. Abaixo, detalhamos três tipos principais:

Paralelismo de Dados. Esse tipo de paralelismo refere-se à divisão de um grande conjunto de dados em partes menores, que são processadas simultaneamente por diferentes unidades de execução. A ideia central é aplicar a mesma operação em diferentes segmentos dos dados de forma paralela. Por exemplo, Hillisetal. [Hillis and Steele 1986] demonstraram que é possível particionar um array de inteiros em blocos e processá-los paralelamente para calcular o somatório total e as somas parciais de cada segmento. Essa abordagem é bastante comum em aplicações de processamento vetorial, big data e aprendizado de máquina, onde grandes volumes de dados podem ser tratados de maneira eficiente com paralelismo em larga escala.

Paralelismo de Funções. Também conhecido como paralelismo de tarefas, este modelo envolve a execução simultânea de diferentes funções ou procedimentos. Ele é mais eficaz quando o sistema possui múltiplos tipos de tarefas que podem ser executadas de forma independente. Um exemplo prático é um servidor DNS recursivo, que pode utilizar diferentes grupos de threads: um grupo dedicado ao tratamento de requisições diretas e outro responsável por resolver requisições recursivas. Esse tipo de paralelismo melhora a responsividade do sistema, evitando que tarefas mais custosas bloqueiem a execução de outras mais simples.

Paralelismo em Pipeline. Neste modelo, diferentes etapas de processamento são executadas em paralelo, cada uma em uma "fase"do pipeline. Assim como em uma linha de

montagem industrial, enquanto uma etapa processa um conjunto de dados, outra já pode iniciar o processamento do próximo. O TensorPipe [Huang et al. 2019] é uma biblioteca que implementa esse tipo de paralelismo para redes neurais profundas organizadas como uma sequência de camadas. Por exemplo, enquanto um lote de dados está sendo processado pelas camadas iniciais da rede, outros lotes podem estar sendo processados nas camadas intermediárias ou finais. Essa abordagem maximiza o uso dos recursos computacionais e reduz significativamente o tempo total de treinamento ou inferência de modelos complexos.

1.4.4.2. Granularidade

A granularidade refere-se ao nível de divisão das tarefas ou dados entre múltiplas threads em um sistema paralelo. Ao distribuir dados para execução concorrente, o objetivo principal é balancear uniformemente a carga de trabalho entre as threads, de modo a maximizar o aproveitamento dos recursos disponíveis e minimizar o tempo ocioso de cada thread. Nesse contexto, a granularidade pode ser classificada em dois tipos principais:

- **Granularidade Grossa:** ocorre quando os dados são divididos em poucas partes relativamente grandes. Essa abordagem tende a reduzir a complexidade da implementação e minimizar o número de seções críticas (regiões que exigem sincronização entre threads), facilitando o gerenciamento do paralelismo. No entanto, ela pode gerar desbalanceamento de carga, já que uma thread pode terminar sua parte rapidamente enquanto outras ainda estão executando.
- **Granularidade Fina:** consiste na divisão dos dados em muitos subconjuntos menores. Isso possibilita uma distribuição de carga mais equilibrada entre as threads, favorecendo o aproveitamento máximo dos núcleos de processamento e aumentando a escalabilidade. Em contrapartida, essa abordagem demanda maior esforço de sincronização e controle, aumentando a complexidade do sistema e a possibilidade de condições de corrida (race conditions).

Não existe uma escolha universalmente ótima de granularidade — a melhor configuração depende diretamente das características do problema em questão, da arquitetura da máquina e do comportamento das tarefas. Por esse motivo, a definição do nível ideal de granularidade costuma ser feita por meio de testes e ajustes empíricos, buscando um equilíbrio entre desempenho e complexidade.

1.4.4.3. Escalonamento

Escalonamento. O escalonamento de threads é um aspecto fundamental da computação paralela que determina como as tarefas são distribuídas entre os recursos de processamento. Quando se distribui threads para tarefas existem três estrategias principais:

- Uma thread por tarefa: Embora sua simplicidade, a criação de diversas threads pode gerar uma sobrecarga alta. Além disso, o tempo de vida das threads pode variar, o que causa ociosidade.

- Escalonamento estático: Embora resolva o problema da sobrecarga na criação das threads, o desbalanceamento pode persistir.

- Escalonamento dinâmico: Esse modelo consegue balancear melhor a carga e reduzir os custos da criação de novas threads, embora seja mais complexo. Isso é possível através do pool de threads, que em base é uma abstração de uma fila que distribui tarefas para as threads, que uma vez que completam suas tarefas já recebem novas, o que maximiza seu uso computacional.

1.4.5. Estratégias de otimização para GPUs

Portar código para GPU com plataformas como CUDA pode ser relativamente simples, mas alcançar eficiência é um desafio que exige otimizações criteriosas. Assim, é necessário otimizações para obter o máximo de poder computacional possível da GPU.

1.4.5.1. Indicadores de desempenho

Ocupância é uma medida de desempenho em CUDA, que é a razão entre a quantidade de threads ativas em um multiprocessador (SM) e a quantidade máxima de threads que ele suporta. **ILP** (Instruction Level Paralelism) indica a quantidade de instruções que uma thread pode executar simultaneamente, utilizando, por exemplo, pipelining. Dado alguns fatores, a GPU pode estar limitada a três fatores:

- **Limitada por latência**: Quando a ocupância ou o ILP se encontra baixo. Aumentar o número de warps por bloco, e reduzir a dependência de dados entre as threads tende a minimizar esses problemas visto que ambas as técnicas aumentam a ocupância e ILP.

- **Limitada por instruções**: Essa dependência acontece quando o kernel executa instruções difíceis de paralelizar, como condicionais, instruções dependentes e instruções muito complexas. No geral, a solução dessa limitação é a refatoração do código de uma forma que evite esses problemas.

- **Limitada por largura de banda**: Contra-intuitivamente é um bom sinal. Ser limitado por largura de banda significa que não há desperdício em relação a capacidade de transferência de memória e mais importante, que a GPU passa a maior parte do tempo trabalhando, tendo o tempo ocioso minimizado. Nesse estado, melhor performance pode ser alcançada melhorando o hardware e reduzindo a quantidade de acessos a memória, como por exemplo, usando o acesso coalesido.

1.4.5.2. Branching

Como as warps operam no modelo SIMT, todas as threads devem executar a mesma instrução simultaneamente. Entretanto, caso ramificações não sejam levadas em conta, a quantidade de instruções executadas pode aumentar, reduzindo a eficiência da GPU.

Tal comportamento se deve porque a unidade de controle que gerencia a warp só consegue emitir uma única instrução para todas as threads, por isso as divergências são resolvidas de modo sequencial. As threads do warp são separadas em dois grupos, as que divergiram e as que não, então a unidade de controle desativará o grupo não divergente e executando o outro. Quando todas as threads novamente se encontrarem no mesmo ponto de execução ou o grupo divergente terminar sua execução, o grupo que foi desativado é reativado e a execução continua.

É evidente a ineficiência que tal comportamento pode trazer, uma vez que uma única thread em um warp pode fazer todas as outras esperarem por ela inativadas para que assim ela execute sua ramificação. Portanto, os algorítimos projetados para GPUs devem minimizar ramificações com o objetivo de maximizar o poder dos warps.

1.4.5.3. Acessos a memória

Embora plataformas como o CUDA gerenciem memória automaticamente, o programador pode determinar em qual camada da hierarquia os dados serão armazenados. Também é possível determinar se são dados constantes, otimizados para leitura rápida, ou dinâmicos, para o uso de DRAM. Da mesma forma, é essencial usar bem a memória compartilhada e os registradores para evitar o uso da memória global, que é mais lenta.

Devido a esse custo, o acesso à memória global deve ser coalescido entre as threads. Isso significa que, se cada thread em uma warp acessar endereços consecutivos de memória, a GPU pode enviar apenas uma solicitação de acesso, o que reduz a quantidade de transações. Uma técnica para maximizar o acesso coalescido é usar estruturas de dados SoA (Structure of Arrays), que organizam os dados de forma que cada elemento das estruturas seja contíguo.

Além disso, em GPUs NVIDIA, a memória compartilhada é dividida 32 conjuntos, o mesmo número de threads por warp, chamados de bancos. Isso acontece porque, se cada thread em uma warp tentar acessar a memória compartilhada via bancos diferentes esse acesso ocorre em paralelo. Caso contrário, o acesso será serializado, o que é ineficiente. Então, para fins de otimização, o programador deve garantir acesso disjuntos aos bancos através do uso de paddings, mapeamento de dados e acessos espaçados.

1.5. Redes de Alta Performance e GPUs

A necessidade de aperfeiçoamento no processamento de pacotes trouxe à luz a possibilidade de utilizar GPUs para acelerar diversas funções de rede [Han et al. 2010], graças ao paralelismo massivo das GPUs e ao paralelismo inerente ao processamento de pacotes [Cerović et al. 2018].

1.5.1. Introdução

Atualmente, aplicações modernas de rede — como serviços em nuvem [Hu et al. 2021], sistemas de inteligência artificial distribuída [Voigtländer et al. 2017] e plataformas de análise de dados em tempo real [Rahman et al. 2019] — impõem demandas cada vez mais rigorosas por alta largura de banda e baixa latência.

Para superar isso, as GPUs emergiram como aceleradores poderosos para computação de propósito geral [Wu and Liu 2008], como o processamento de pacotes de rede em alto desempenho [Cerović et al. 2018]. Aliado a isso, muitas tarefas de processamento de rede, como classificação, filtragem e inspeção profunda de pacotes, exibem paralelismo de dados inerente, tornando-as adequadas para execução em GPUs [Han et al. 2010].

Nesse sentido, as GPUs se mostraram eficazes para acelerar diversos dispositivos e funções de rede — como roteadores de software [Han et al. 2010, Sun and Ricci 2013], virtualização de funções de rede [Zhang et al. 2018, Guo et al. 2022] e sistemas de detecção de intrusão [Vasiliadis et al. 2008] — devido à sua arquitetura massivamente paralela, que permite lidar com inúmeros pacotes simultaneamente, e à alta largura de banda de memória, que possibilita o processamento eficiente de cargas com uso intensivo de memória [Go et al. 2017].

1.5.2. Ferramentas e frameworks para integração de GPUs com redes

A integração eficiente entre GPUs e sistemas de rede exige ferramentas e frameworks especializados, capazes de atender às exigências de alta largura de banda e baixa latência. Nesse contexto, diversas soluções foram desenvolvidas para facilitar a comunicação entre a GPU e a pilha de rede, otimizando o envio e recebimento de pacotes. Entre as principais ferramentas abordadas estão o DPDK (Data Plane Development Kit), o DOCA (Data Center and Cloud Automation) e o PF_RING, cada uma oferecendo recursos distintos para melhorar o desempenho da rede e a interação com as GPUs, tornando-as essenciais para ambientes de alta performance.

1.5.2.1. Data Plane Development Kit (DPDK)

O DPDK [DPDK Project 2024] é um conjunto de bibliotecas de código aberto gerido pela Linux Foundation, cujo principal objetivo é transferir o processamento de pacotes do kernel e sua tradicional pilha de protocolos para o espaço de usuário. Essa abordagem visa reduzir significativamente a sobrecarga provocada por interrupções de hardware, trocas de contexto e múltiplas cópias de dados entre o buffer da NIC (placa de rede - do inglês Network Interface Card), o kernel e o espaço de usuário. Como resultado, o DPDK permite alcançar baixos níveis de latência e alta vazão no processamento de pacotes, sendo uma solução fundamental para aplicações que exigem alta performance em redes de alto desempenho.

Para superar as sobrecargas citadas, o DPDK utliza acesso direto à memória (DMA) para movimentar pacotes entre a NIC e os buffers no espaço de usuário. Essa técnica elimina cópias intermediárias e reduz o tempo de acesso aos dados. Além disso, o DPDK, substitui as interrupções de hardware por polling ativo, que verifica continuamente a chegada de pacotes, o que diminui a latência e aumenta a vazão.

Com a crescente adoção da computação em nuvem, esforços têm sido feitos para migrar as NFVs (Network Function Virtualization) para um contexto de software, com ênfase na execução em máquinas virtuais ou containers. Nesse contexto, Kourtis et al. [Kourtis et al. 2015] demonstraram que, ao empregar o DPDK, as NFVs alcançam vazões significativamente superiores quando comparadas às suas versões baseadas na pilha

de redes tradicional do kernel.

Além disso, o DPDK GPUdev [Kumar et al. 2023] é uma biblioteca introduzida no framework DPDK com o objetivo de facilitar a integração das GPUs NVIDIA ao ecossistema. A interação entre a NIC e a GPU pode ocorrer com a CPU atuando como intermediária, ou de forma direta, por meio do uso do GPUdev RDMA, que elimina a necessidade de cópias intermediárias e reduz significativamente a latência.

O avanço dos telescópios tem ampliado significativamente o volume de dados coletados. Com projetos como o Square Kilometer Array, estima-se que as taxas de transferência atinjam dezenas de terabits por segundo, exigindo soluções mais eficientes de processamento. Nesse cenário, o DPDK GPUdev [Plante et al. 2022], com suporte ao GPUDirect e transferência via DMA para a GPU, mostrou-se eficaz para atender às demandas de alta vazão na aquisição de dados em astronomia.

1.5.2.2. NVIDIA DOCA

As Unidades de Processamento de Dados (DPUs) foram desenvolvidas para lidar com tarefas intensivas de processamento de dados, aliviando a carga das CPUs e permitindo que seus recursos sejam dedicados exclusivamente às aplicações. Nesse contexto, o NVIDIA DOCA é um framework que tem como objetivo aprimorar e acelerar as operações em data centers por meio do uso das DPUs NVIDIA BlueField e das NICs da família NVIDIA Mellanox ConnectX.

O DOCA Core [NVIDIA Corporation 2022] fornece uma camada de abstração de hardware que permite descobrir as unidades de aceleração de hardware disponíveis nos dispositivos, consultar suas capacidades e gerenciar a alocação e o compartilhamento desses recursos com as bibliotecas DOCA, facilitando o desenvolvimento de aplicações que exploram todo o potencial das DPUs BlueField.

As aplicações na DPU precisam de buffers, tanto de entrada quanto de saída, para o processamento de dados. Esses buffers podem ser criados e inicializados diretamente pela aplicação, utilizando o DOCA Software Development Kit (SDK), que se beneficia do modo zero-copy para se comunicar com o hardware. O DOCA SDK permite que o programador registre regiões de memória a serem usadas na criação dos buffers. Os buffers acessíveis pela DPU são, subsequentemente, gerenciados pelo DOCA Core, podendo referenciar regiões de memória da CPU, GPU ou via RDMA.

O DOCA GPUNetIO [NVIDIA Corporation 2024c] é um componente do framework DOCA que permite o processamento de pacotes em tempo real diretamente na GPU, viabilizando a comunicação direta entre a GPU e a NIC. Essa biblioteca também propõe diversas funcionalidades com foco na GPU, tais como:

- **GPUDirect Async Kernel-Initiated Network (GDAKIN)**: permite que kernels CUDA interajam diretamente com a NIC para envio e recepção de pacotes.
- **GPUDirect RDMA**: possibilita o recebimento direto de pacotes na memória da GPU, eliminando cópias intermediárias.

- **Ethernet Protocol Management on GPU (DOCA Ethernet)**: habilita o processamento de pacotes Ethernet diretamente na GPU.

e outras funcionalidades como semáforos, alocação inteligente de memória e gerenciamento de RDMA protocolado na GPU. A configuração inicial da aplicação — incluindo a alocação de memória e o lançamento dos kernels CUDA — é realizada pela CPU. No entanto, o caminho de dados, isto é, o fluxo dos pacotes de rede, ocorre exclusivamente entre a NIC e a GPU, eliminando a necessidade de envolvimento da CPU nesse processo e otimizando o desempenho.

O NVIDIA Aerial SDK para redes 5G demonstra o uso do DOCA GPUNetIO como peça central na aceleração do processamento de pacotes em Redes de Acesso de Rádio (RAN) [NVIDIA Corporation 2023]. Ao permitir a comunicação direta entre a memória da GPU e a DPU, o DOCA remove a CPU do caminho crítico de I/O, reduzindo a latência e aumentando a eficiência. O Aerial utiliza CUDA para processar as camadas cuPHY e cuMAC inteiramente na GPU, reforçando a viabilidade de redes 5G definidas por software com aceleração por GPU.

1.5.2.3. PF_RING

O PF_RING [ntop 2025] é um módulo do kernel Linux e um framework open source em espaço de usuário que permite o processamento de pacotes em altas taxas, ao mesmo tempo em que fornece uma API consistente para aplicações de processamento de pacotes.

O PF_RING otimiza a captura de pacotes por meio da utilização de um buffer circular [NTOP 2018]. Os pacotes são copiados das NICs por meio da NAPI (do inglês New API) [Salim 2005]) para este buffers, também chamados de "rings", que podem ser lidos por aplicações no espaço de usuário.

O PF_RING traz como algumas de suas vantagens a possibilidade entre distribuir pacotes recebidos por diversos "rings"(criados com base na quantidade de aplicações), além de que pacotes não são enfileirados nas estruturas de dados de rede do kernel [Deri et al. 2004]. Assim como o DPDK, este framework permite processar pacotes em velocidade de linha (line speed) em hardware comum, ao reduzir o custo computacional por contornar o kernel durante o processamento [Dai and Wang 2019].

Como os drivers padrão não oferecem desempenho ideal na captura e transmissão de pacotes — especialmente em cenários com pacotes pequenos e em altas taxas —, a solução PF_RING, combinada com o driver DNA (Direct NIC Access), permite a cópia eficiente de pacotes diretamente da NIC para a memória via DMA (Direct Memory Access). Com base nessa infraestrutura, foi implementado um escalonador parcialmente preemptivo para gerenciar a transferência de dados entre a NIC e a memória da GPU, utilizando fluxos paralelos. Essa abordagem permite mascarar a latência de transferência ao explorar a execução e a cópia de dados de forma concorrente — um recurso suportado por GPUs de última geração [Ammendola et al. 2014].

1.5.3. O papel das GPUs na aceleração de redes.

Ao incorporar GPUs em tarefas de rede, é possível otimizar operações como inspeção de pacotes, filtragem de tráfego e balanceamento de carga, reduzindo significativamente a latência e aumentando a eficiência do sistema. Essa aceleração é especialmente relevante em contextos de segurança, onde a resposta rápida a ameaças e a análise em tempo real são essenciais. A seguir, são apresentados alguns dos principais usos das GPUs em redes, com foco em suas contribuições para o desempenho e a segurança dos sistemas.

1.5.3.1. Inspeção Profunda de Pacotes

Sistemas de segurança de rede convencionais, como os firewalls, provêm segurança por meio da inspeção de cabeçalhos de pacotes, prática que não garante a segurança principalmente na perspectiva da camada de aplicação [Lee et al. 2015]. A Inspeção Profunda de Pacotes (do inglês Deep packet inspection - DPI) é uma técnica utilizada para examinar o conteúdo de pacotes a fim de garantir a segurança da rede [Lin et al. 2016], com o objetivo de inspecionar cada byte da carga útil do pacote e detectar possíveis intrusões [Sharma and Singh 2015].

Sendo assim, Sistemas de Detecção de Intrusões na Rede (do inglês network intrusion detection systems - NIDSs) foram criados para promover melhor segurança. Estes sistemas podem ser baseados na detecção de anomalias ou assinaturas. Os sistemas baseados em anomalias monitoram a rede e detectam possíveis comportamentos anômalos, enquanto os sistemas baseados em assinatura buscam encontrar padrões maliciosos na carga útil dos pacotes. Os sistemas baseados em assinatura, por promover maior detecção contra ataques, foram o foco para um número maior de estudos [Lee et al. 2015].

Os sistemas baseados em assinatura, por outro lado, apresentaram consumo de aproximadamente 70% do tempo de execução do sistema, o que fez com que os estudos voltassem a buscar muitas soluções de software e hardware, futuramente levando à utilização de unidades de processamento gráfico (do inglês Graphical Processing Unit - GPU) [Lee et al. 2015]. Essas unidades, que possuem poder de processamento paralelo superior em comparação com as CPUs, aumentam a eficiência da tarefa de correspondência de padrões, já que a tarefa é dividida em segmentos paralelos, resultando em uma busca por padrões mais rápida do que a realizada em uma CPU [Sharma and Singh 2015].

1.5.3.2. Filtragem de tráfego

A filtragem de tráfego (do inglês traffic filtering) é uma técnica essencial para a segurança de redes, voltada para a inspeção dos pacotes e controle de fluxo que circulam entre sistemas. Seu objetivo principal é permitir apenas o tráfego legítimo, bloqueando comunicações não autorizadas e prevenindo atividades maliciosas.

A filtragem de pacotes (packet filtering) avalia o cabeçalho de cada pacote com base em critérios como endereços IP, portas, protocolos e o conteúdo da carga útil dos pacotes usando DPI [Lin et al. 2016]. A principal implementação da filtragem de pacotes são os firewalls [Abie 2000, Gouda and Liu 2005], que protegem os pontos de entrada de

uma rede inspecionando os pacotes de dados e decidindo, com base em regras predefinidas, se eles devem ser permitidos, rejeitados ou descartados.

Durante ataques coordenados, como os de negação de serviço distribuída (DDoS), que geram um volume massivo de dados, firewalls que utilizam filtragem por assinatura, que detectam padrões conhecidos desses ataques, podem se tornar ineficazes dado a dificuldade em diferenciar tráfego legítimo do malicioso em padrões desconhecidos. Embora ferramentas como NetFlow [Hofstede et al. 2014] e sFlow [Ujjan et al. 2020] possibilitem a análise de fluxos e a detecção de padrões anômalos em ataques em larga escala, essa análise só ocorre durante ou após o ataque.

Para detectar padrões irregulares, filtros como o NetBouncer [Thomas et al. 2003], que valida a legitimidade do usuário ao exigir que ele resolva desafios específicos, o SIFF [Yaar et al. 2004], que realiza uma negociação prévia via handshake com os usuários e descarta tráfego não autorizado através de roteadores intermediários, e o Hop-Count Filter [Jin et al. 2003], que detecta tráfego falsificado verificando se o TTL do cabeçalho IP é consistente o endereço de origem, foram projetados para mitigar o tráfego malicioso, evitando que ele cause danos significativos à rede.

Para lidar com períodos de tráfego intenso — especialmente quando causado por ataques DDoS - diversas soluções utilizando GPUs foram apresentadas. Sahoo et al. [Sahoo et al. 2014] e Keskin et al [Keskin et al. 2016] desenvolveram algoritmos paralelizados para firewalls, sendo que o trabalho de Keskin foca especificamente na mitigação de ataques DDoS, utilizando CUDA para execução em GPUs NVIDIA. Esses algoritmos possibilitam uma redução significativa no tempo de processamento de pacotes, além de melhorar a detecção de ameaças e intrusões.

1.5.3.3. Balanceamento de carga

O balanceamento de carga é uma técnica fundamental na computação paralela, cujo objetivo é distribuir de maneira uniforme o trabalho entre os recursos computacionais disponíveis, evitando a sobrecarga em unidades específicas e garantindo a utilização eficiente do sistema [Barney et al. 2010]. Com o avanço do uso de GPUs como aceleradores de uso geral, novas estratégias passaram a ser adotadas para o balanceamento de carga, sobretudo em aplicações de redes.

As diferenças arquitetônicas entre CPUs e GPUs levaram a desafios significativos para as técnicas tradicionais de balanceamento de carga baseadas em CPU. Enquanto CPUs operam com poucos núcleos potentes e threads pesadas, GPUs são projetadas para executar milhares de threads leves em paralelo, exigindo um paralelismo de grão fino. Essa assimetria expôs novas formas de desequilíbrio na carga de trabalho, especialmente em aplicações irregulares, onde a distribuição dinâmica e eficiente das tarefas torna-se ainda mais crítica.

As GPUs operam com paralelismo massivo baseado em SIMD, onde o processamento eficiente depende que threads de um mesmo warp (grupo de 32 threads) devem seguir o mesmo fluxo de execução. Divergências entre elas causam ociosidade e reduzem o desempenho. Para evitar isso, é fundamental que as threads dentro de um mesmo warp

realizem aproximadamente a mesma quantidade de trabalho e sigam o mesmo conjunto de instruções. Além disso O escalonador da NVIDIA também contribui para o balanceamento ao alocar blocos de threads nos Streaming Multiprocessors (SMs) conforme os recursos disponíveis [Osama 2022]. A eficiência da GPU, portanto, depende da uniformidade da carga entre os warps, tornando o balanceamento essencial para o uso eficiente dos núcleos e alto rendimentom, pois cargas desbalanceadas implicam subutilização de núcleos, comprometendo o throughput total da aplicação.

No contexto de redes, as GPUs oferecem uma alternativa eficiente para acelerar funções computacionalmente intensivas, como a verificação de redundância cíclica (CRC), espelhamento de tráfego e buscas em tabelas de roteamento. Nesses cenários, a CPU deixa de executar diretamente essas funções e passa a gerenciar a comunicação com a GPU. Para evitar gargalos, o balanceamento de carga envolve não apenas a distribuição entre núcleos da CPU, mas também o uso de múltiplas filas independentes de envio de dados entre CPU e GPU permitindo que diferentes núcleos da CPU enviem pacotes simultaneamente à GPU, promovendo maior paralelismo e melhor aproveitamento dos recursos.

A eficácia desse balanceamento depende da natureza da tarefa e do tamanho dos pacotes. Em funções simples, como Ethernet Mirroring, o desempenho é limitado pela taxa de comunicação CPU-GPU, o que torna mais eficiente ampliar o número de núcleos da CPU. Já em tarefas como CRC, o tamanho dos pacotes é decisivo: pacotes pequenos mantêm o gargalo na comunicação, mas pacotes maiores se beneficiam do uso de múltiplas filas, que podem ser processadas paralelamente pela GPU, aumentando a taxa de processamento [Van Hauwaert et al. 2024].

1.6. Implementação e Modelagem

Esta seção descreve o processo de implementação do processamento de pacotes de rede utilizando GPUs, especificamente com CUDA e a biblioteca NVIDIA DOCA. Além de um tutorial prático, a seção explora criticamente desafios de gerenciamento de memória, balanceamento de carga, limitações das abordagens atuais e potenciais avanços futuros.

1.6.1. Tutorial Prático de Programação com CUDA e DOCA

Para implementar uma aplicação básica de processamento de pacotes utilizando CUDA e DOCA, é necessário que o ambiente esteja corretamente configurado. Isso inclui a instalação do CUDA em uma GPU NVIDIA compatível com DOCA, executando em um sistema operacional como o Ubuntu, por exemplo. Inicialmente, devem-se instalar as dependências necessárias utilizando o gerenciador de pacotes:

Listing 1.1. Como instalar as dependências do DOCA

```
$> sudo apt update
$> sudo apt install nvidia-cuda-toolkit nvidia-driver nvidia-
   doca-sdk
```

Após a instalação, é importante configurar as variáveis de ambiente no arquivo ".bashrc"para assegurar o funcionamento adequado das ferramentas:

Listing 1.2. atualizando o PATH do sistema através do .bashrc

```
export PATH=/usr/local/cuda/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH
```

O passo seguinte é criar um kernel CUDA simples que será responsável pelo processamento direto dos pacotes recebidos na GPU. Um exemplo prático pode ser uma operação de inversão de bits utilizando XOR:

Listing 1.3. Implementação simples de inversão de bits

```
__global__ void process_packets_kernel(char *packet_data, int
   packet_size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < packet_size) {
        packet_data[idx] = packet_data[idx] ^ 0xFF;
    }
}
```

A integração com DOCA é feita inicializando o GPUNetIO para criar um caminho de dados direto entre a NIC e a GPU. O GPUNetIO é uma tecnologia da NVIDIA que possibilita a transferência direta e eficiente de pacotes da NIC para a memória da GPU sem a necessidade de cópias intermediárias na CPU [NVIDIA Corporation 2024c]. Essa técnica reduz a latência e melhora o desempenho global do processamento de pacotes, permitindo uma utilização mais eficiente dos recursos computacionais disponíveis (detalhado na seção 1.5.2.2). Os comandos abaixo exemplificam o carregamento e utilização da GPUNetIO:

Listing 1.4. Configuração e carregamento do GPUNetIO

```
doca_gpu_init();
doca_gpu_netio_t *gpu_netio;
doca_gpu_netio_create(&gpu_netio, device, cuda_stream);
```

Para executar a aplicação, uma *stream* CUDA é criada para organizar e gerenciar as operações que serão executadas em paralelo na GPU. O comando *cudaStreamCreate(&stream)* inicializa uma nova *stream* CUDA que permite lançar *kernels* CUDA de maneira assíncrona. A chamada *process_packets_kernel* realiza efetivamente o processamento paralelo dos pacotes recebidos diretamente na memória da GPU, onde *num_blocks* indica o número de blocos CUDA e *block_size* define a quantidade de *threads* por bloco.

Esses parâmetros devem ser definidos levando em consideração o tamanho total dos dados (*packet_size*) e a capacidade da GPU utilizada. Por fim, o comando *cudaStreamSynchronize(stream)* garante que a execução do *kernel* seja completamente finalizada antes que o programa continue, evitando possíveis inconsistências ou acessos indevidos à memória que ainda esteja em uso pela GPU. Essa estrutura de execução proporciona um controle eficiente e seguro sobre o processamento paralelo de pacotes na GPU. O código abaixo mostra esta utilização do CUDA *Stream*:

Listing 1.5. Utilização do Stream CUDA

```
cudaStream_t stream;
cudaStreamCreate(&stream);

process_packets_kernel<<<num_blocks, block_size, 0, stream>>>(
   packet_buffer, packet_size);
cudaStreamSynchronize(stream);
```

1.6.2. Adição de cabeçalho HTTP

Nesta seção é demonstrado como utilizar a biblioteca DOCA para implementar uma função que insere um novo cabeçalho a uma requisição HTTP. *insert_custom_http_header()* é uma função que pode ser implementada no contexto de um kernel CUDA operando sob a infraestrutura do DOCA GPUNetIO. Esta implementação assume um cenário em que os pacotes de rede contêm uma requisição HTTP completa, ou seja, não há necessidade de remontagem de fluxo TCP (TCP *reassembly*). Além disso, considera-se que os pacotes estejam armazenados diretamente na memória acessível pela GPU, tipicamente utilizando estruturas como `doca_gpu_buf_arr_t`, e que as modificações no *payload* possam ser realizadas diretamente na memória, desde que haja espaço reservado para a inserção do novo cabeçalho.

Os pressupostos adicionais incluem o fato de que o ponteiro `char*` para o *buffer* do pacote aponta diretamente para o início do payload HTTP, e que o *kernel* já tenha executado a operação de *parsing* dos cabeçalhos IP e TCP para localizar a região correta do payload. Também se assume que a carga útil da requisição HTTP esteja completamente contida dentro de um único *buffer* na GPU (por exemplo, 1500 bytes, o tamanho típico de um quadro Ethernet). A tarefa do *kernel* consiste, portanto, em localizar o final dos cabeçalhos HTTP e inserir, nesse ponto, um novo cabeçalho personalizado, como por

exemplo `X-Custom-Header: HelloFromDOCA`.

Listing 1.6. Função de inserção de cabeçalho HTTP

```
__device__ void insert_custom_http_header(char* payload, int
    payload_len, int max_buf_size) {
  const char* custom_header = "X-Custom-Header: HelloFromDOCA\r\
      n";
    const int custom_header_len = 31;

    // Encontra final do cabecalho HTTP: "\r\n\r\n"
    int i;
    for (i = 0; i < payload_len - 3; ++i) {
        if (payload[i] == '\r' && payload[i+1] == '\n' &&
            payload[i+2] == '\r' && payload[i+3] == '\n') {
            break;
        }
    }

    if (i >= payload_len - 3) return; // Cabecalho imcompleto

    int header_end_index = i;
    int insertion_point = header_end_index;

    // Verifica se ha buffer overflow
    if (payload_len + custom_header_len >= max_buf_size)
        return;

    // Libera espaco para novos cabecalhos
    for (int j = payload_len-1; j >= insertion_point + 4; --j) {
         payload[j + custom_header_len] = payload[j];
    }

    // Adiciona o novo cabecalho
    for (int k = 0; k < custom_header_len; ++k) {
        payload[insertion_point + 2 + k] = custom_header[k];
    }
}
```

1.6.3. Ferramentas e *Frameworks*

O PacketShader é um framework projetado para acelerar o processamento de pacotes em software através do uso intensivo de GPUs [Han et al. 2010]. Ele utiliza a capacidade massiva de paralelismo das GPUs para realizar operações de rede com alta performance, como roteamento e criptografia, superando significativamente abordagens tradicionais baseadas apenas em CPUs.

O APUNet é uma abordagem que aproveita GPUs para processamento eficiente de pacotes em redes virtuais. O *framework* otimiza o desempenho das funções de rede virtualizadas (VNFs) ao explorar o poder computacional paralelo das GPUs, proporcio-

nando assim melhorias significativas em desempenho e escalabilidade em ambientes de redes virtualizadas [Go et al. 2017].

O GPUnet oferece uma camada de abstração que permite comunicação eficiente entre GPUs distribuídas através da rede [NVIDIA Corporation 2024c]. O objetivo principal do GPUnet é integrar comunicações de rede diretamente ao espaço de endereçamentos da GPU, possibilitando o envio e recebimento direto de pacotes pela GPU sem intervenção da CPU.

1.6.4. Desafios de Gerenciamento de Memória em GPUs

Um dos principais desafios na gestão de memória em GPU é a latência associada à transferência de pacotes entre CPU e GPU, que pode limitar severamente o desempenho geral da aplicação [Huang et al. 2020]. Além disso, acessos desalinhados à memória global, compartilhada ou registradores na GPU podem causar quedas significativas na eficiência computacional. Finalmente, a necessidade frequente de sincronização entre operações realizadas pela CPU e pela GPU pode reduzir o paralelismo disponível, comprometendo a vantagem inicial proporcionada pelas GPUs [Kim and Park 2024].

1.6.5. Balanceamento de Carga com DOCA e GPU

O balanceamento de carga utilizando GPUs em conjunto com a plataforma DOCA (Data Center on a Chip Architecture) tem se mostrado uma abordagem promissora para maximizar a utilização dos recursos computacionais em redes de alto desempenho. A arquitetura DOCA, desenvolvida pela NVIDIA, permite integrar diretamente a lógica de rede com capacidades de processamento acelerado, utilizando recursos como SmartNICs (DPUs) para descarregar tarefas da CPU. Ao empregar GPUs nesse contexto, é possível aproveitar o paralelismo massivo oferecido por esses dispositivos para processar múltiplos fluxos de dados simultaneamente, reduzindo significativamente a latência e aumentando a vazão do sistema [Osama et al. 2023].

Essa combinação entre DOCA e GPU viabiliza um balanceamento de carga mais inteligente e dinâmico, no qual o roteamento e o processamento de pacotes podem ser adaptados em tempo real com base em métricas de tráfego e disponibilidade de recursos. A CPU, aliviada de tarefas intensivas, pode se concentrar na orquestração de alto nível, enquanto a GPU assume operações de análise de pacotes, criptografia, compressão ou filtragem. Isso resulta não apenas em um melhor aproveitamento do hardware, mas também em maior eficiência energética e escalabilidade da infraestrutura, o que é especialmente relevante em datacenters que lidam com volumes massivos de tráfego.

1.7. Casos de Uso

Nesta seção, serão apresentados exemplos práticos de como as GPUs estão sendo utilizadas em cenários reais. Serão detalhadas aplicações em espelhamento de porta ethernet, sistemas de detecção e prevenção de intrusão (IDS/IPS) acelerados por GPU, pesquisa de protocolo da Internet e cálculo de verificações de redundância cíclica. Cada caso de uso será acompanhado de métricas de desempenho e discussões sobre os benefícios e limitações da abordagem com GPUs, fornecendo uma visão prática e aplicada das tecnologias discutidas nas seções anteriores.

1.7.1. Espelhamento de Porta Ethernet

O espelhamento de porta Ethernet é uma técnica utilizada para monitoramento e análise de tráfego em redes, permitindo a cópia de pacotes de uma interface para outra sem interferir no fluxo original da rede. Esse mecanismo é amplamente empregado em cenários de segurança, diagnóstico de falhas e monitoramento de desempenho, sendo essencial para administradores de rede que necessitam inspecionar pacotes em tempo real.

Tradicionalmente, essa funcionalidade é implementada diretamente em switches e roteadores, utilizando recursos dedicados de hardware. No entanto, com o crescimento do tráfego de rede e a necessidade de maior flexibilidade, soluções baseadas em software, como aquelas construídas sobre DPDK (Data Plane Development Kit), se tornaram populares [Shanmugalingam et al. 2016]. Entretanto, o processamento de pacotes exclusivamente na CPU apresenta limitações de desempenho, especialmente em redes de baixa latência e alta vazão.

Dessa forma, o uso de GPUs para o processamento de pacotes no espelhamento de porta Ethernet se apresenta como uma alternativa eficiente para lidar com grandes volumes de tráfego. As GPUs são capazes de processar pacotes em paralelo, reduzindo a latência e aumentando a escalabilidade do sistema. A abordagem geralmente segue o seguinte fluxo, como ilustrado na Figura 1.4:

1. Pacotes são capturados da interface de rede em rajadas (*batches*).
2. Os pacotes são transferidos para a GPU, onde o cabeçalho é analisado e modificado conforme necessário.
3. Os pacotes são reenviados para a interface de destino, replicando o tráfego para monitoramento.

Existem diferentes técnicas para otimizar a movimentação de pacotes entre CPU e GPU. Algumas abordagens incluem:

- **Listas de Comunicação (*CommLists*)**: Organização dos pacotes em estruturas otimizadas para transferência eficiente entre CPU e GPU.
- **Regiões de Interesse Coalescentes (*ROIs*)**: Processamento apenas das partes relevantes dos pacotes, reduzindo a movimentação de dados entre os dispositivos.

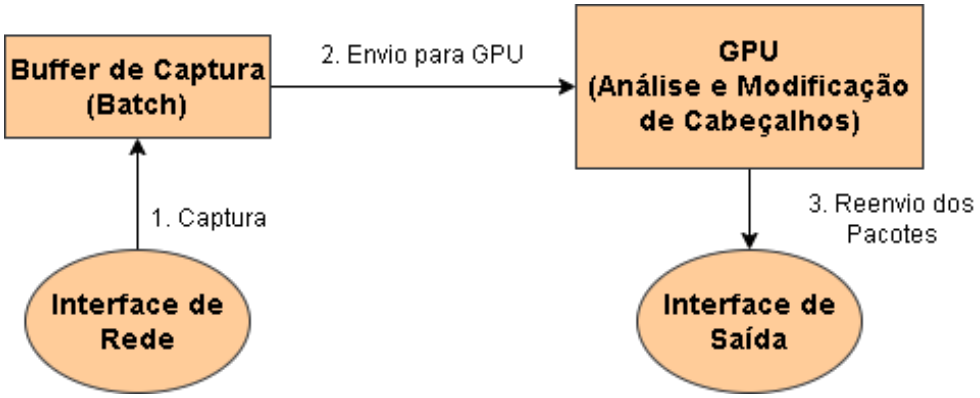

Figura 1.4. Etapas do processamento paralelo de pacotes com GPU no espelhamento de porta.

O espelhamento de porta Ethernet baseado em GPU proporciona alta taxa de transferência e baixa latência, tornando-se uma solução viável para redes modernas que exigem monitoramento em tempo real sem degradação de desempenho. Além disso, essa abordagem pode ser combinada com outras técnicas avançadas, como detecção de intrusões e análise de tráfego baseada em aprendizado de máquina, aproveitando o poder computacional da GPU para tarefas mais complexas.

1.7.2. Cálculo de Verificações de Redundância Cíclica

A verificação de redundância cíclica (CRC) é um código de detecção de erros amplamente utilizado em redes digitais para garantir a integridade dos dados transmitidos. No contexto das redes Ethernet, cada quadro termina com uma sequência de verificação de quadros (*Frame Check Sequence* – FCS), composta por um CRC de 32 bits (CRC-32), que permite ao receptor identificar possíveis erros na transmissão. O cálculo do CRC-32 FCS de um quadro Ethernet é realizado considerando todos os campos do quadro, com exceção do próprio FCS. Esse cálculo é baseado na divisão polinomial dos dados por um polinômio gerador de grau 32, conforme especificado na Seção 3.2.9 do protocolo IEEE 802.3 [Eth 2022].

Tradicionalmente, a computação do CRC é realizada diretamente no hardware das placas de rede, permitindo um cálculo incremental à medida que os quadros são recebidos ou transmitidos. No entanto, esse processo apresenta uma dependência sequencial, pois o cálculo do CRC de um byte depende do resultado obtido no byte anterior. Essa característica impõe desafios para arquiteturas multi-core convencionais, onde a capacidade de paralelização é limitada ao número de núcleos disponíveis. Além disso, a variação no tamanho dos quadros dificulta uma paralelização eficiente, devido à natureza cíclica do algoritmo de CRC.

Diante dessas limitações, o uso de unidades de processamento gráfico (GPUs) para a computação do CRC-32 surge como uma abordagem promissora, explorando o alto grau de paralelismo dessas arquiteturas. Para otimizar o desempenho, diferentes estratégias podem ser empregadas:

- **Pré-computação de tabelas**: A tabela de CRC pode ser previamente calculada na

CPU e transferida para a memória global da GPU, permitindo consultas eficientes durante o processamento dos pacotes.

- **Execução em lote (*batch processing*)**: Em vez de calcular o CRC de cada pacote individualmente, a GPU processa múltiplos pacotes simultaneamente, reduzindo a sobrecarga de transferência de dados entre CPU e GPU.

- **Coalescência de acessos à memória (*coalesced memory access*)**: A organização dos dados é otimizada para minimizar latências de acesso, garantindo maior eficiência no uso da largura de banda da memória.

Implementações baseadas na pré-computação de tabelas, por exemplo, podem oferecer ganhos significativos de desempenho, superando em até cinco vezes implementações tradicionais baseadas em CPU e processamento bit a bit [Perez 1983], dependendo da estrutura dos dados e da eficiência na movimentação entre memória principal e memória da GPU. Dessa forma, a computação de CRC-32 em GPUs possibilita um processamento paralelo altamente eficiente, reduzindo a carga sobre a CPU e viabilizando maior escalabilidade para aplicações de monitoramento e segurança em redes de alto desempenho.

1.7.3. Detecção de Intrusão (IDS/IPS) Acelerada por GPU

A internet se tornou um ecossistema essencial para transações financeiras, comunicações e armazenamento de informações pessoais e corporativas. No entanto, essa conectividade massiva também ampliou a superfície de ataque, tornando as redes vulneráveis a diversas ameaças, como exploração de vulnerabilidades, injeção de código malicioso e ataques distribuídos de negação de serviço (DDoS). Diante desse cenário, a implementação de mecanismos eficazes de detecção e prevenção de intrusões tornou-se fundamental para garantir a segurança dos sistemas computacionais e a integridade e segurança dos dados dos usuários.

Em vista disso, os sistemas de detecção de intrusão (*Intrusion Detection Systems* – IDS) e de prevenção de intrusão (*Intrusion Prevention Systems* – IPS) desempenham um papel essencial nesse contexto. O IDS é um mecanismo passivo que monitora e analisa o tráfego da rede e dos dispositivos conectados em busca de padrões que indiquem possíveis ataques ou atividades suspeitas. Já o IPS, além de monitorar o tráfego, atua de forma ativa, bloqueando automaticamente ameaças, encerrando conexões perigosas e mitigando potenciais ataques antes que comprometam o sistema. Embora ambos os sistemas compartilhem objetivos semelhantes, suas abordagens diferem: enquanto o IDS se concentra na detecção e alerta de incidentes, o IPS age preventivamente para conter ameaças em tempo real.

Com o aumento exponencial do tráfego de rede e da complexidade dos ataques cibernéticos, a detecção eficiente de ameaças tornou-se um desafio computacional significativo. A implementação de IDS/IPS tradicionais baseados em CPU pode enfrentar limitações de escalabilidade e desempenho, especialmente ao lidar com grandes volumes de pacotes em tempo real. Nesse contexto, o uso de unidades de processamento gráfico (GPUs) para acelerar a análise de pacotes tem se mostrado uma alternativa promissora,

permitindo a execução paralela de algoritmos de inspeção de tráfego e aprendizado de máquina para identificação de ameaças com maior eficiência.

1.7.3.1. Sistema de Detecção de Intrusões (IDS)

A aceleração de IDS por meio de GPUs tem sido bastante explorada visando aumentar a eficiência na detecção de ameaças em redes de alto desempenho. Em 2006, um estudo pioneiro propôs a delegação da operação de correspondência de assinaturas para a GPU, reduzindo a carga de processamento na CPU do sistema IDS [Jacob and Brodley 2006]. A solução implementada, denominada PixelSnort, demonstrou um desempenho significativamente mais robusto em cenários de alta carga computacional, superando a versão convencional do Snort em até 40%.

Com os avanços em aprendizado de máquina e computação paralela, abordagens mais sofisticadas passaram a explorar redes neurais para a detecção de anomalias em tráfego de rede. Em 2015, pesquisadores implementaram um IDS baseado em redes neurais utilizando GPUs para acelerar a classificação de comportamentos normais e anômalos [Thanh Van and Thinh 2015]. O experimento foi conduzido com o conjunto de dados *KDD Cup 1999* [Stolfo et al. 1999], amplamente utilizado para avaliação de sistemas de detecção de intrusões. Esse dataset contém registros simulando tráfego de rede real, categorizados como conexões normais ou ataques de diversos tipos (como DoS, probe, R2L e U2R).

A implementação em GPU apresentou desempenho até 32 vezes superior à versão equivalente em CPU, garantindo maior robustez, escalabilidade e capacidade de resposta em tempo real. Esses resultados evidenciam o potencial das GPUs na mitigação das limitações computacionais dos IDSs convencionais, viabilizando sistemas de segurança mais ágeis e eficazes.

1.7.3.2. Sistema de Prevenção de Intrusões (IPS)

No contexto de IPS, abordagens baseadas em aprendizado profundo (deep learning) acelerado por GPU têm demonstrado um potencial significativo na implementação de sistemas mais eficazes e escaláveis. Em uma dessas abordagens, pesquisadores usaram um Sistema de Detecção e Prevenção de Intrusões (IDPS) hierárquico de três camadas, no qual a validação de pacotes ocorre progressivamente em diferentes níveis da rede, para proteger as redes de possíveis ataques [Ali and Yousaf 2020].

Na primeira camada, o sistema realiza a validação de usuários por meio de etiquetas RFID e assinaturas criptografadas, assegurando a autenticidade dos dispositivos conectados. Em seguida, a segunda camada, emprega filtros nebulosos de segunda ordem (*Type-II fuzzy filtering*) para analisar as características dos pacotes e detectar padrões de tráfego suspeitos. Por fim, a terceira camada adota um modelo de deep learning acelerado por GPU para classificar os pacotes em normais, suspeitos ou maliciosos, permitindo uma resposta proativa a potenciais ameaças.

Os experimentos conduzidos no simulador OMNeT++ demonstraram ganhos ex-

pressivos na taxa de detecção de intrusos, além da redução da taxa de falha, latência e impacto no throughput. Dessa forma, a utilização de GPUs para acelerar a análise de pacotes viabilizou a processamento em tempo real, proporcionando escalabilidade e melhor desempenho em comparação a abordagens convencionais baseadas exclusivamente em CPU.

Em suma, a aplicação de GPUs na detecção e prevenção de intrusões representa um avanço significativo na cibersegurança, permitindo análises mais rápidas e escaláveis para redes de alto desempenho. À medida que ataques se tornam mais sofisticados e volumosos, o uso de GPUs para a execução paralela de algoritmos de inspeção e aprendizado de máquina se mostra essencial para garantir a proteção eficaz de sistemas críticos.

1.7.4. Pesquisa de Protocolo da Internet

Com o aumento significativo da capacidade das redes de comunicação e o crescimento contínuo do tráfego de dados na Internet, o processamento de pacotes, como a busca de endereços IP, tem se tornado uma preocupação central para a eficiência das redes. Funções cruciais de roteamento realizadas em switches e roteadores não conseguem acompanhar o aumento da velocidade dos links de comunicação, o que coloca pressão sobre os dispositivos para que possam processar pacotes de maneira mais rápida e eficaz.

Nesse contexto, as unidades de processamento gráfico (GPUs) emergem como uma solução promissora devido à sua alta capacidade de paralelismo e flexibilidade. Por conseguinte, GPUs oferecem uma plataforma robusta para implementar funções de roteamento de maneira escalável e com alto desempenho. A pesquisa de endereço IP, especificamente a correspondência do maior prefixo (Longest Prefix Match - LPM), é um desafio importante nesse cenário, pois requer a busca rápida de endereços em grandes bancos de dados de roteamento.

A utilização de GPUs para a busca de endereços IP foi explorada por diferentes abordagens, incluindo a implementação de estruturas de dados como tries. Em 2013, um estudo propôs uma solução inovadora utilizando o método de *Multi-bit Trie* para a pesquisa de IPs, aproveitando a paralelização das GPUs. Essa abordagem, chamada de *GPU-Accelerated Multi-bit Trie* (GAMT), permitiu alcançar velocidades de lookup impressionantes, com até 1072 milhões de buscas por segundo (MLPS) para IPv4 e 658 MLPS para IPv6, mesmo com tráfego altamente dinâmico, incluindo atualizações frequentes de até 70.000 por segundo. Além disso, mesmo com um tamanho de lote reduzido, a técnica manteve a latência média de pesquisa abaixo de 100 microssegundos [Li et al. 2013].

Mais recentemente, outra variante de solução, baseada em uma abordagem de trie adaptada, também foi proposta para resolver o problema de LPM, em que o banco de dados de endereços IP é particionado em tabelas diferentes, com base nos primeiros k bits do endereço IP. Essa abordagem mostrou uma melhoria significativa no desempenho, com uma aceleração de 64,46% em relação à implementação do binary trie e de 94,32% em relação à implementação de árvores de busca binária (BST), destacando o potencial da GPU em otimizar o processamento de pacotes em redes de alta velocidade [Sonai et al. 2023].

Esses avanços indicam que o uso de GPUs no processamento de busca de endereços IP não apenas oferece uma solução escalável para lidar com o aumento de tráfego e

complexidade das redes, mas também garante uma melhoria considerável no desempenho de dispositivos de roteamento, tornando-os mais eficientes e preparados para as demandas do tráfego de rede moderno.

1.8. Desafios e Tendências Futuras

O uso de GPUs no processamento de pacotes tem se consolidado como uma solução promissora para atender às crescentes demandas por desempenho e escalabilidade em redes de alta velocidade. Graças à sua capacidade de paralelismo massivo e alto throughput, as GPUs oferecem vantagens significativas em tarefas que requerem o processamento simultâneo de grandes volumes de dados, como na aceleração de roteamento, monitoramento de tráfego e filtragem de pacotes. Essas vantagens tornam as GPUs uma escolha atraente para cenários que exigem processamento em tempo real e em larga escala, frequentemente encontrados em redes de alta performance e data centers.

No entanto, apesar das melhorias substanciais na capacidade de processamento, o uso de GPUs também impõe desafios notáveis, especialmente no que diz respeito ao consumo energético e à eficiência térmica. O alto desempenho das GPUs vem com um custo considerável em termos de energia, o que pode resultar em problemas térmicos e dificuldades na gestão de grandes sistemas com múltiplas GPUs. Além disso, a escalabilidade desses sistemas continua sendo uma questão crítica, pois a necessidade de interconectar várias GPUs em arquiteturas distribuídas pode gerar gargalos que afetam o desempenho geral. Outro ponto crucial é a integração com arquiteturas emergentes, como ambientes virtualizados e infraestruturas em nuvem, onde a alocação eficiente de recursos e a compatibilidade com tecnologias de virtualização podem complicar ainda mais a implementação de soluções baseadas em GPU.

Dessa forma, o uso de GPUs no processamento de pacotes exige uma análise cuidadosa das vantagens e limitações envolvidas, sendo necessário superar desafios técnicos tanto em nível de hardware quanto de software. Nesta seção, discutiremos em detalhes esses desafios, além de explorar exemplos de aplicações e tópicos de pesquisa relevantes, com o intuito de apresentar uma visão mais aprofundada sobre as oportunidades e obstáculos no uso de GPUs em redes de alta velocidade. A partir disso, também serão destacados os aprendizados adquiridos até o momento e as perspectivas futuras para o desenvolvimento dessa tecnologia.

1.8.1. Consumo de Energia e Eficiência Térmica

Entre os principais desafios, destacam-se o consumo de energia e a eficiência térmica, principalmente em ambientes de larga escala, como os data centers. A crescente adoção de tecnologias como a Inteligência Artificial intensificou a demanda por maior capacidade de processamento, levando à expansão desses ambientes e aumentando a pressão sobre os sistemas de energia e resfriamento [Dayarathna et al. 2016]. Embora as GPUs ofereçam benefícios computacionais expressivos, esses ganhos vêm acompanhados de custos operacionais significativos para manter os dispositivos funcionando de forma estável e eficiente.

Além disso, o intenso processamento também gera grande dissipação de calor, exigindo sistemas de resfriamento avançados para evitar o superaquecimento e garantir

a longevidade do hardware, o que também consome muita energia [Alkrush et al. 2024]. Dessa forma, em data centers, a soma desses fatores pode comprometer a eficiência global do sistema, tornando essencial o desenvolvimento de soluções que otimizem o consumo energético e aprimorem a gestão térmica garantindo a sustentabilidade do sistema.

Em vista disso, estratégias como o ajuste dinâmico de voltagem e frequência (DVFS) [Hosseini Shirvani et al. 2020] e o investimento em sistemas de resfriamento inteligentes — como o *Two-phase cooling* [Kanbur et al. 2020] e o *TES-based cooling* [Chen et al. 2019] — surgem como alternativas viáveis para enfrentar esses desafios. Ademais, os fabricantes têm explorado o uso de novos materiais e arquiteturas, como a utilização de compostos avançados para dissipação térmica e soluções de resfriamento por imersão líquida [Pambudi et al. 2022]. Com isso, essas soluções não apenas otimizam o consumo energético e a gestão térmica, mas também promovem um equilíbrio sustentável entre desempenho, custo operacional e eficiência ambiental.

1.8.2. Limitações de Hardware

Além das questões energéticas e térmicas, o hardware das GPUs impõe limitações técnicas que afetam o desempenho de aplicações de larga escala no processamento de pacotes. Um dos principais gargalos é a capacidade limitada de memória, que pode restringir o processamento eficiente de grandes fluxos de dados em redes de alta velocidade. Para mitigar essa limitação, estratégias como a troca de memória entre GPU e CPU têm sido exploradas, mas muitas abordagens existentes não oferecem um desempenho satisfatório para todos os modelos [Huang et al. 2020].

Recentemente, a introdução da memória HBM (High Bandwidth Memory) tem se mostrado uma solução promissora para mitigar os gargalos de comunicação entre CPU e GPU em sistemas de alto desempenho. Essa tecnologia oferece uma largura de banda significativamente maior do que as memórias convencionais, ao mesmo tempo em que consome menos energia, o que a torna especialmente atrativa para aplicações que exigem grande volume de troca de dados e alto poder computacional. Como resultado, a HBM tem sido cada vez mais utilizada para melhorar a escalabilidade e a eficiência energética de arquiteturas heterogêneas, proporcionando uma interação mais fluida entre os diferentes componentes do sistema [Kim and Park 2024].

Apesar dos benefícios, a adoção generalizada da HBM ainda enfrenta obstáculos técnicos consideráveis. Questões relacionadas à dissipação térmica, complexidade de empilhamento de memória DRAM e o uso de tecnologias como TSV (Through-Silicon Via) continuam sendo desafios centrais para a viabilização em larga escala dessa tecnologia. Estudos recentes apontam que, mesmo com os avanços no empacotamento 3D e nas técnicas de interconexão vertical, ainda é necessário desenvolver soluções mais eficazes para controle térmico e redução dos custos de produção [Kim and Park 2024]. Esses fatores tornam a HBM uma tecnologia promissora, mas que exige contínua inovação para alcançar sua maturidade plena no mercado.

1.8.3. Integração com Redes Definidas por Software (SDN)

O processamento de pacotes baseado em GPU também precisa lidar com diversos desafios ao ser integrado com redes definidas por software (SDN) [Pantuza et al. 2014]. As

redes SDN, que oferecem flexibilidade e controle centralizado, precisam frequentemente de processamento de pacotes em alta velocidade para garantir a baixa latência e alta vazão necessárias para suas operações. Embora as GPUs possam oferecer um desempenho significativo em tarefas intensivas, a latência adicional introduzida pela comunicação entre a CPU e a GPU pode impactar diretamente o tempo de resposta das redes, comprometendo a eficiência do processamento [Li et al. 2020].

No contexto das SDNs, diversos frameworks têm sido adotados para explorar o potencial das GPUs e melhorar o desempenho geral do sistema. O P4, por exemplo, é um framework de linguagem de programação de rede que permite programar a camada de encaminhamento de pacotes de maneira altamente flexível. Da mesma forma, o eBPF (Extended Berkeley Packet Filter) oferece a possibilidade de realizar programações avançadas no nível do kernel, permitindo otimizações em tempo real e processamentos especializados diretamente na rede. Embora esses frameworks ofereçam boas perspectivas, a transmissão de dados entre diferentes camadas de hardware continua sendo um dos principais desafios, especialmente quando se trata da comunicação entre a CPU, a GPU e os dispositivos de rede.

A escolha da tecnologia de interconexão entre os diferentes componentes do sistema, como PCIe, NVLink e NVSwitch, pode ter um impacto significativo na escalabilidade e no desempenho geral do sistema de processamento de pacotes. O PCIe, amplamente utilizado, é eficaz, mas apresenta limitações em termos de largura de banda. Tecnologias como NVLink e NVSwitch oferecem maiores capacidades de comunicação e largura de banda, sendo particularmente benéficas em sistemas com múltiplas GPUs e uma alta demanda por transferência de dados. A adoção dessas tecnologias pode melhorar a eficiência da comunicação entre CPU, GPU e dispositivos de rede, mas sua implementação precisa ser cuidadosamente otimizada para garantir que os benefícios sejam efetivos, sem gerar gargalos no processamento.

1.8.4. Virtualização e GPUs na Nuvem

Por fim, a integração do processamento de pacotes baseado em GPU com ambientes virtualizados e infraestruturas em nuvem representa um desafio significativo para arquiteturas modernas de rede e computação. Embora as GPUs ofereçam alto desempenho para cargas de trabalho intensivas e paralelizáveis, sua utilização em contextos multitenant — como os encontrados na computação em nuvem — exige soluções robustas de virtualização. A alocação dinâmica e eficiente desses recursos, além do controle rigoroso sobre o compartilhamento de hardware, é essencial para garantir que múltiplas aplicações possam coexistir sem comprometer o desempenho ou a previsibilidade das execuções [Belkhiri and Dagenais 2024].

Nesse cenário, diferentes tecnologias têm sido exploradas para viabilizar a virtualização de GPUs de forma eficaz. Entre elas, destacam-se soluções como o NVIDIA vGPU (Virtual GPU), que permite particionar uma única GPU física em múltiplas instâncias virtuais; o Multi-Process Service (MPS), que permite que diferentes processos compartilhem a GPU de maneira eficiente, reduzindo a sobrecarga de inicialização e contexto; e o SR-IOV (Single Root I/O Virtualization), que facilita o acesso direto de máquinas virtuais aos dispositivos de hardware, minimizando a latência. Essas abordagens

buscam melhorar a flexibilidade na distribuição de recursos, aumentar a escalabilidade do ambiente e reduzir o custo operacional por meio de uma melhor utilização da infraestrutura [Hong et al. 2017].

Ainda assim, a implementação prática dessas tecnologias enfrenta limitações importantes. A complexidade do gerenciamento de recursos compartilhados, os desafios de isolamento entre usuários, e o impacto na latência e no throughput das aplicações são pontos que demandam avanços contínuos. Além disso, a integração entre as camadas de hardware, sistema operacional e orquestração em nuvem precisa ser cuidadosamente projetada para garantir que os benefícios do uso de GPUs não sejam anulados por gargalos de virtualização. O desenvolvimento de novos protocolos, APIs padronizadas e modelos de escalonamento conscientes do desempenho será crucial para permitir que a computação acelerada por GPU seja adotada de forma eficiente, segura e escalável no contexto da nuvem.

1.9. Conclusões e Discussões Finais

Ao longo deste trabalho, foi explorado de forma abrangente o uso de GPUs no contexto do processamento de pacotes de redes, destacando sua relevância em um cenário onde a demanda por desempenho e eficiência cresce continuamente. Inicialmente foi apresentando os fundamentos do processamento de pacotes, explicando seu papel central na comunicação em redes e os desafios enfrentados para garantir desempenho em ambientes de alta velocidade. O entendimento dos modelos OSI e TCP/IP, assim como dos mecanismos de roteamento, permitiu estabelecer uma base sólida para a discussão técnica que se seguiu.

Em seguida, foi introduzido o universo da computação com GPU, apresentando sua arquitetura, principais fabricantes e as ferramentas de programação que permitem explorar seu imenso potencial de paralelismo. Comparada às CPUs, a GPU oferece vantagens substanciais em tarefas altamente paralelizáveis, como a análise de pacotes em grande escala. A familiarização com ferramentas como CUDA e OpenCL foi essencial para compreender como essas unidades podem ser aplicadas efetivamente ao contexto de redes.

Nos fundamentos do processamento paralelo, foi discutido modelos de programação, sincronização entre threads e estratégias de otimização. Esse conhecimento é crucial para extrair o máximo desempenho das GPUs e garantir que as aplicações desenvolvidas sejam escaláveis e eficientes. Além disso, a compreensão dos gargalos e limitações comuns em sistemas paralelos permite o desenvolvimento de soluções mais robustas.

A seção sobre redes de alta performance aprofundou a discussão sobre como as GPUs podem ser integradas a frameworks como DPDK e PF_RING para alcançar maior rendimento no tráfego de dados. Foram analisadas aplicações específicas, como inspeção profunda de pacotes e balanceamento de carga, onde o uso de GPUs se mostra especialmente vantajoso. Esses casos práticos demonstram que a combinação de GPUs com bibliotecas de rede avançadas resulta em soluções altamente otimizadas.

Nas seções práticas, abordou-se a modelagem e implementação de soluções baseadas em CUDA para tarefas comuns no processamento de pacotes. Foram discutidas as particularidades do gerenciamento de memória em GPUs, além de técnicas de ba-

lanceamento de carga que permitem a distribuição eficiente de tarefas entre núcleos de processamento. Essa parte demonstrou como o conhecimento teórico pode ser aplicado diretamente em cenários reais.

Os casos de uso detalhados reforçaram ainda mais o potencial da abordagem. Aplicações como firewalls e sistemas de detecção de intrusão (IDS/IPS) mostraram-se especialmente beneficiadas pelo uso de GPUs, tanto em desempenho quanto em capacidade de escalabilidade.

Também foi discutido os desafios que ainda precisam ser enfrentados para consolidar o uso de GPUs em redes, como consumo energético, integração com SDN, e virtualização em nuvem. Apesar das vantagens, esses fatores exigem atenção para garantir que as soluções sejam sustentáveis e viáveis em ambientes de produção. A tendência é que, com o amadurecimento dessas tecnologias, novos paradigmas de rede surjam, mais adaptáveis e orientados à performance.

Por fim, é possível concluir que o uso de GPUs no processamento de pacotes representa uma fronteira promissora para redes de alta performance. Combinando alto poder de paralelismo, ferramentas de desenvolvimento maduras e uma crescente base de conhecimento acadêmico e industrial, a integração entre computação gráfica e redes de computadores abre caminho para novas soluções, mais rápidas, inteligentes e escaláveis. A continuidade das pesquisas nessa área será essencial para acompanhar as exigências de um mundo cada vez mais conectado e dinâmico.

Referências

[Eth 2022] (2022). Ieee standard for ethernet. *IEEE Std 802.3-2022 (Revision of IEEE Std 802.3-2018)*, pages 1–7025.

[Abie 2000] Abie, H. (2000). An overview of firewall technologies. *Telektronikk*, 96(3):47–52.

[Ali and Yousaf 2020] Ali, A. and Yousaf, M. M. (2020). Novel three-tier intrusion detection and prevention system in software defined network. *IEEE Access*, 8:109662–109676.

[Alkrush et al. 2024] Alkrush, A. A., Salem, M. S., Abdelrehim, O., and Hegazi, A. (2024). Data centers cooling: A critical review of techniques, challenges, and energy saving solutions. *International Journal of Refrigeration*, 160:246–262.

[Ammendola et al. 2014] Ammendola, R., Biagioni, A., Deri, L., Fiorini, M., Frezza, O., Lamanna, G., Cicero, F. L., Lonardo, A., Messina, A., Sozzi, M., et al. (2014). Gpus for real-time processing in hep trigger systems. In *Journal of Physics: Conference Series*, volume 523, page 012007. IOP Publishing.

[Association et al. 1997] Association, I. S. et al. (1997). Ieee standard for wireless lan medium access control (mac) and physical layer (phy) specifications. *IEEE Std 802.11-1997*, pages 1–445.

[Barney et al. 2010] Barney, B. et al. (2010). Introduction to parallel computing. *Lawrence Livermore National Laboratory*, 6(13):10.

[Belkhiri and Dagenais 2024] Belkhiri, A. and Dagenais, M. (2024). Analyzing gpu performance in virtualized environments: A case study. *Future Internet*, 16(3).

[Bernstein 1966] Bernstein, A. J. (1966). Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763.

[Bonola et al. 2022] Bonola, M., Belocchi, G., Tulumello, A., Brunella, M. S., Siracusano, G., Bianchi, G., and Bifulco, R. (2022). Faster software packet processing on FPGA NICs with eBPF program warping. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 987–1004, Carlsbad, CA. USENIX Association.

[Brunella et al. 2022] Brunella, M. S., Belocchi, G., Bonola, M., Pontarelli, S., Siracusano, G., Bianchi, G., Cammarano, A., Palumbo, A., Petrucci, L., and Bifulco, R. (2022). hxdp: Efficient software packet processing on fpga nics. *Communications of the ACM*, 65(8):92–100.

[Cerović et al. 2018] Cerović, D., Del Piccolo, V., Amamou, A., Haddadou, K., and Pujolle, G. (2018). Fast packet processing: A survey. *IEEE Communications Surveys & Tutorials*, 20(4):3645–3676.

[Chen et al. 2019] Chen, X., Zhang, Q., Zhai, Z. J., and Ma, X. (2019). Potential of ventilation systems with thermal energy storage using pcms applied to air conditioned buildings. *Renewable Energy*, 138:39–53.

[Dagum and Menon 1998] Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55.

[Dai and Wang 2019] Dai, W. and Wang, F. (2019). Study on processing performance of a dpdk and gpu combined pulsar data reduction system. *International Journal of Mechatronics and Applied Mechanics*, (6):213–224.

[Day and Zimmermann 1983] Day, J. D. and Zimmermann, H. (1983). The osi reference model. *Proceedings of the IEEE*, 71(12):1334–1340.

[Dayarathna et al. 2016] Dayarathna, M., Wen, Y., and Fan, R. (2016). Data center energy consumption modeling: A survey. *IEEE Communications Surveys & Tutorials*, 18(1):732–794.

[Deri et al. 2004] Deri, L. et al. (2004). Improving passive packet capture: Beyond device polling. In *Proceedings of SANE*, volume 2004, pages 85–93. Amsterdam, Netherlands.

[Dijkstra 1965] Dijkstra, E. W. (1965). Cooperating sequential processes, technical report ewd-123. Technical report.

[DPDK Project 2024] DPDK Project (2024). About DPDK. Accessed: 2025-04-09.

[Du et al. 2023a] Du, Y., Chang, K., Shi, J., Zhou, Y., and Liu, M. (2023a). A survey on mechanisms for fast network packet processing. In *Proceedings of the 2023 4th International Conference on Computing, Networks and Internet of Things*, pages 57–66.

[Du et al. 2023b] Du, Y., Chang, K., Shi, J., Zhou, Y., and Liu, M. (2023b). A survey on mechanisms for fast network packet processing. In *Proceedings of the 2023 4th International Conference on Computing, Networks and Internet of Things*, CNIOT '23, page 57–66, New York, NY, USA. Association for Computing Machinery.

[Flynn 1966] Flynn, M. (1966). Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909.

[Forouzan and Fegan 2006] Forouzan, B. A. and Fegan, S. C. (2006). *TCP/IP Protocol Suite*. McGraw-Hill Higher Education, 3nd edition.

[Fu et al. 1989] Fu, B., Saini, A., and Gelsinger, P. P. (1989). Performance and microarchitecture of the i486 processor. In *Proceedings 1989 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 182–183. IEEE Computer Society.

[Geer 2005] Geer, D. (2005). Chip makers turn to multicore processors. *Computer*, 38(5):11–13.

[Go et al. 2017] Go, Y., Jamshed, M. A., Moon, Y., Hwang, C., and Park, K. (2017). {APUNet}: Revitalizing {GPU} as packet processing accelerator. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 83–96.

[Gottlieb et al. 1983] Gottlieb, Grishman, Kruskal, McAuliffe, Rudolph, and Snir (1983). The nyu ultracomputer—designing an mimd shared memory parallel computer. *IEEE Transactions on computers*, 100(2):175–189.

[Gouda and Liu 2005] Gouda, M. and Liu, A. (2005). A model of stateful firewalls and its properties. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 128–137.

[Guo et al. 2022] Guo, L., Zhang, K., and Wang, X. S. (2022). Gaviss : Boosting the performance of gpu-accelerated nfv systems via data sharing. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):4472–4483.

[Halaas et al. 2004] Halaas, A., Svingen, B., Nedland, M., Saetrom, P., Snove, O., and Birkeland, O. (2004). A recursive misd architecture for pattern matching. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(7):727–734.

[Han et al. 2010] Han, S., Jang, K., Park, K., and Moon, S. (2010). Packetshader: a gpu-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 40(4):195–206.

[Hedrick 1988] Hedrick, C. L. (1988). Rfc1058: Routing information protocol.

[Hillis and Steele 1986] Hillis, W. D. and Steele, G. L. (1986). Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183.

[Hofstede et al. 2014] Hofstede, R., Čeleda, P., Trammell, B., Drago, I., Sadre, R., Sperotto, A., and Pras, A. (2014). Flow monitoring explained: From packet capture to data analysis with netflow and ipfix. *IEEE Communications Surveys & Tutorials*, 16(4):2037–2064.

[Hong et al. 2017] Hong, C.-H., Spence, I., and Nikolopoulos, D. S. (2017). Gpu virtualization and scheduling methods: A comprehensive survey. 50(3).

[Hosseini Shirvani et al. 2020] Hosseini Shirvani, M., Rahmani, A. M., and Sahafi, A. (2020). A survey study on virtual machine migration and server consolidation techniques in dvfs-enabled cloud datacenter: Taxonomy and challenges. *Journal of King Saud University - Computer and Information Sciences*, 32(3):267–286.

[Hu et al. 2021] Hu, S., Bai, W., Chen, K., Tian, C., Zhang, Y., and Wu, H. (2021). Providing bandwidth guarantees, work conservation and low latency simultaneously in the cloud. *IEEE Transactions on Cloud Computing*, 9(2):763–776.

[Huang et al. 2020] Huang, C.-C., Jin, G., and Li, J. (2020). Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1341–1355, New York, NY, USA. Association for Computing Machinery.

[Huang et al. 2019] Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. (2019). Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32.

[Hwang and Briggs 1990] Hwang, K. and Briggs, F. A. (1990). *Computer Architecture and Parallel Processing*. McGraw-Hill, Inc., USA, 1st edition.

[Jacob and Brodley 2006] Jacob, N. and Brodley, C. (2006). Offloading ids computation to the gpu. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 371–380.

[Jin et al. 2003] Jin, C., Wang, H., and Shin, K. G. (2003). Hop-count filtering: an effective defense against spoofed ddos traffic. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, page 30–41, New York, NY, USA. Association for Computing Machinery.

[Kanbur et al. 2020] Kanbur, B. B., Wu, C., Fan, S., Tong, W., and Duan, F. (2020). Two-phase liquid-immersion data center cooling system: Experimental performance and thermoeconomic analysis. *International Journal of Refrigeration*, 118:290–301.

[Keskin et al. 2016] Keskin, S., Erdogan, H. T., and Kocak, T. (2016). Graphics processing unit based next generation ddos prevention system. In *2016 4th International Symposium on Digital Forensic and Security (ISDFS)*, pages 59–62.

[Kim and Park 2024] Kim, K. and Park, M.-j. (2024). Present and future, challenges of high bandwith memory (hbm). In *2024 IEEE International Memory Workshop (IMW)*, pages 1–4.

[Kourtis et al. 2015] Kourtis, M.-A., Xilouris, G., Riccobene, V., McGrath, M. J., Petralia, G., Koumaras, H., Gardikis, G., and Liberal, F. (2015). Enhancing vnf performance by exploiting sr-iov and dpdk packet processing acceleration. In *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, pages 74–78.

[Kumar et al. 2023] Kumar, V., Rao, R., and Kim, Y. (2023). Optimizing inline packet processing using dpdk and gpudev with gpus. Accessed: 2025-04-09.

[Lamport 2019] Lamport, L. (2019). *Time, clocks, and the ordering of events in a distributed system*, page 179–196. Association for Computing Machinery, New York, NY, USA.

[Lee et al. 2015] Lee, C.-L., Lin, Y.-S., and Chen, Y.-C. (2015). A hybrid cpu/gpu pattern-matching algorithm for deep packet inspection. *PloS one*, 10(10):e0139301.

[Li et al. 2020] Li, A., Song, S. L., Chen, J., Li, J., Liu, X., Tallent, N. R., and Barker, K. J. (2020). Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110.

[Li et al. 2013] Li, Y., Zhang, D., Liu, A. X., and Zheng, J. (2013). Gamt: A fast and scalable ip lookup engine for gpu-based software routers. In *Architectures for Networking and Communications Systems*, pages 1–12.

[Lin et al. 2016] Lin, Y.-S., Lee, C.-L., and Chen, Y.-C. (2016). Length-bounded hybrid cpu/gpu pattern matching algorithm for deep packet inspection. In *Proceedings of the Fifth International Conference on Network, Communication and Computing*, pages 63–67.

[Liu et al. 2010] Liu, Y., Schmidt, B., and Maskell, D. L. (2010). Cudasw++ 2.0: enhanced smith-waterman protein database search on cuda-enabled gpus based on simt and virtualized simd abstractions. *BMC research notes*, 3:1–12.

[Maziero 2014] Maziero, C. A. (2014). Sistemas operacionais: conceitos e mecanismos. *Livro aberto*.

[Moore 1965] Moore, G. (1965). Moore's law. *Electronics Magazine*, 38(8):114.

[NTOP 2018] NTOP (2018). PF_RING. `https://www.ntop.org/products/packet-capture/pf_ring/`. Acesso em: 11 abr. 2025.

[ntop 2025] ntop (2025). PF_RING - GitHub Repository. `https://github.com/ntop/PF_RING`. Acesso em: 11 abr. 2025.

[NVIDIA Corporation 2022] NVIDIA Corporation (2022). *NVIDIA DOCA Core Programming Guide v1.5.2*. NVIDIA. https://docs.nvidia.com/doca/archive/doca-v1.5.2/doca-core-programming-guide/index.html.

[NVIDIA Corporation 2023] NVIDIA Corporation (2023). NVIDIA Aerial SDK - Product Brief. https://docs.nvidia.com/aerial/archive/aerial-sdk/23-4/text/product_brief/product_brief_intro.html. Acesso em: abr. 2025.

[NVIDIA Corporation 2024a] NVIDIA Corporation (2024a). Cuda c++ programming guide. Accessed on 2024.

[NVIDIA Corporation 2024b] NVIDIA Corporation (2024b). Cuda zone. Accessed on 2024.

[NVIDIA Corporation 2024c] NVIDIA Corporation (2024c). DOCA GPUNetIO SDK Guide. https://docs.nvidia.com/doca/sdk/doca+gpunetio/index.html. Acesso em: abr. 2025.

[Osama 2022] Osama, M. (2022). *GPU load balancing*. University of California, Davis.

[Osama et al. 2023] Osama, M., Porumbescu, S. D., and Owens, J. D. (2023). A programming model for gpu load balancing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 79–91.

[Pambudi et al. 2022] Pambudi, N. A., Sarifudin, A., Firdaus, R. A., Ulfa, D. K., Gandidi, I. M., and Romadhon, R. (2022). The immersion cooling technology: Current and future development in energy saving. *Alexandria Engineering Journal*, 61(12):9509–9527.

[Pantuza et al. 2021a] Pantuza, G., Bleme, L. A. C., Vieira, M. A. M., and Vieira, L. F. M. (2021a). Danian: tail latency reduction of networking application through an o(1) scheduler. In *2021 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6.

[Pantuza et al. 2014] Pantuza, G., Sampaio, F., Vieira, L. F. M., Guedes, D., and Vieira, M. A. M. (2014). Network management through graphs in software defined networks. In *10th International Conference on Network and Service Management (CNSM) and Workshop*, pages 400–405.

[Pantuza et al. 2021b] Pantuza, G., Vieira, M. A. M., and Vieira, L. F. M. (2021b). equic gateway: Maximizing quic throughput using a gateway service based on ebpf + xdp. In *2021 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6.

[Perez 1983] Perez, A. (1983). Byte-wise crc calculations. *IEEE Micro*, 3(3):40–50.

[Plante et al. 2022] Plante, J., Gratadour, D., Matias, L., Viou, C., and Agostini, E. (2022). A high-performance data acquisition on cots hardware for astronomical instrumentation. In *Software and Cyberinfrastructure for Astronomy VII*, volume 12189, pages 328–337. SPIE.

[Rahman et al. 2019] Rahman, M., Islam, M., Calhoun, J., and Chowdhury, M. (2019). Real-time pedestrian detection approach with an efficient data communication bandwidth strategy. *Transportation research record*, 2673(6):129–139.

[Rodriguez et al. 2001] Rodriguez, A., Gatrell, J., and Peschke, R. (2001). *TCP/IP Tutorial and Technical Overview*. Prentice-Hall, Inc., USA, 7th edition.

[Sahoo et al. 2014] Sahoo, A. K., Das, A., and Tiwary, M. (2014). Firewall engine based on graphics processing unit. In *2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies*, pages 758–763.

[Salim 2005] Salim, J. H. (2005). When napi comes to town. In *Linux 2005 Conf*. Citeseer.

[Sanders and Kandrot 2010] Sanders, J. and Kandrot, E. (2010). *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.

[Sato et al. 2020] Sato, M., Ishikawa, Y., Tomita, H., Kodama, Y., Odajima, T., Tsuji, M., Yashiro, H., Aoki, M., Shida, N., Miyoshi, I., Hirai, K., Furuya, A., Asato, A., Morita, K., and Shimizu, T. (2020). Co-design for a64fx manycore processor and "fugaku". In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15.

[Seiler 2018] Seiler, G. (2018). Faster avx2 optimized ntt multiplication for ring-lwe lattice cryptography. *Cryptology ePrint Archive*.

[Shanmugalingam et al. 2016] Shanmugalingam, S., Ksentini, A., and Bertin, P. (2016). Dpdk open vswitch performance validation with mirroring feature. In *2016 23rd International Conference on Telecommunications (ICT)*, pages 1–6.

[Sharma and Singh 2015] Sharma, J. and Singh, M. (2015). Cuda based rabin-karp pattern matching for deep packet inspection on a multicore gpu. *International Journal of Computer Network and Information Security*, 7(10):70–77.

[Sidhu et al. 1993] Sidhu, D., Fu, T., Abdallah, S., Nair, R., and Coltun, R. (1993). Open shortest path first (ospf) routing protocol simulation. *ACM SIGCOMM Computer Communication Review*, 23(4):53–62.

[Sonai et al. 2023] Sonai, V., Bharathi, I., and Noor Mahammad, S. (2023). A perspective of ip lookup approach using graphical processing unit (gpu). In Molla, A. R., Sharma, G., Kumar, P., and Rawat, S., editors, *Distributed Computing and Intelligent Technology*, pages 98–103, Cham. Springer Nature Switzerland.

[Stolfo et al. 1999] Stolfo, S., Fan, W., Lee, W., Prodromidis, A., and Chan, P. (1999). Kdd cup 1999 data. `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`. Accessed: 2025-04-13.

[Sun and Ricci 2013] Sun, W. and Ricci, R. (2013). Fast and flexible: Parallel packet processing with gpus and click. *ANCS 2013 - Proceedings of the 9th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 25–35.

[Tanenbaum and Wetherall 2010] Tanenbaum, A. S. and Wetherall, D. J. (2010). *Computer Networks*. Prentice Hall Press, USA, 5th edition.

[Thanh Van and Thinh 2015] Thanh Van, N. T. and Thinh, T. N. (2015). Accelerating anomaly-based ids using neural network on gpu. In *2015 International Conference on Advanced Computing and Applications (ACOMP)*, pages 67–74.

[The Khronos Group Inc. 2024] The Khronos Group Inc. (2024). OpenCL. Accessed on 2024.

[Thomas et al. 2003] Thomas, R., Mark, B., Johnson, T., and Croall, J. (2003). Netbouncer: client-legitimacy-based high-performance ddos filtering. In *Proceedings DARPA Information Survivability Conference and Exposition*, volume 1, pages 14–25 vol.1.

[Ujjan et al. 2020] Ujjan, R. M. A., Pervez, Z., Dahal, K., Bashir, A. K., Mumtaz, R., and González, J. (2020). Towards sflow and adaptive polling sampling for deep learning based ddos detection in sdn. *Future Generation Computer Systems*, 111:763–779.

[Van Hauwaert et al. 2024] Van Hauwaert, R., Vanliefde, M., and Barbette, T. (2024). Gpu-based packet processing.

[Vasiliadis et al. 2008] Vasiliadis, G., Antonatos, S., Polychronakis, M., Markatos, E. P., and Ioannidis, S. (2008). Gnort: High performance network intrusion detection using graphics processors. In *Recent Advances in Intrusion Detection: 11th International Symposium, RAID 2008, Cambridge, MA, USA, September 15-17, 2008. Proceedings 11*, pages 116–134. Springer.

[Vieira et al. 2020] Vieira, M. A., Castanho, M. S., Pacífico, R. D., Santos, E. R., Júnior, E. P. C., and Vieira, L. F. (2020). Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)*, 53(1):1–36.

[Voigtländer et al. 2017] Voigtländer, F., Ramadan, A., Eichinger, J., Lenz, C., Pensky, D., and Knoll, A. (2017). 5g for robotics: Ultra-low latency control of distributed robotic systems. In *2017 International Symposium on Computer Science and Intelligent Controls (ISCSIC)*, pages 69–72.

[Walker and Dongarra 1996] Walker, D. W. and Dongarra, J. J. (1996). Mpi: a standard message passing interface. *Supercomputer*, 12:56–68.

[Wang et al. 2008] Wang, L., Huang, Y.-z., Chen, X., and Zhang, C.-y. (2008). Task scheduling of parallel processing in cpu-gpu collaborative environment. In *2008 International Conference on Computer Science and Information Technology*, pages 228–232.

[Wu and Liu 2008] Wu, E. and Liu, Y. (2008). Emerging technology about gpgpu. In *APCCAS 2008 - 2008 IEEE Asia Pacific Conference on Circuits and Systems*, pages 618–622.

[Yaar et al. 2004] Yaar, A., Perrig, A., and Song, D. (2004). Siff: a stateless internet flow filter to mitigate ddos flooding attacks. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, pages 130–143.

[Zhang et al. 2018] Zhang, K., He, B., Hu, J., Wang, Z., Hua, B., Meng, J., and Yang, L. (2018). {G-NET}: Effective {GPU} sharing in {NFV} systems. In *15th USENIX Symposium on networked systems design and implementation (NSDI 18)*, pages 187–200.

[Zhou et al. 2014] Zhou, S., Singapura, S. G., and Prasanna, V. K. (2014). High-performance packet classification on gpu. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6.