Explorando o processamento paralelo na classificação de tráfego em redes de alta velocidade

Alysson F. Santos, Petrônio G. L. Júnior, Stenio F. L. Fernandes, Djamel F. H. Sadok

Centro de Informática – Universidade de Federal de Pernambuco (UFPE)
Recife – PE – Brasil

{afs,pglj,sflf,jamel}@cin.ufpe.br

Abstract. *Traffic Identification is a crucial function performed by ISPs administrators to evaluate and improve its network service. Deep Packet Inspection (DPI) is a well-known technique used to classify traffic and relies mostly in Regular Expressions (REs) evaluated by Finite Automata. No previous studies have discussed the impact of DPI on the overall system throughput. This work presents a novel technique to perform DPI on GPU called Flow-Based Traffic Identification (FBTI). Basically we want to increase DPI systems performance on commodity platforms to classify networked traffic at high speed links. We achieve a raw throughput classification of approximately 114 Gbps on GPUs. Our prototype solution reach a real total throughput of approximately 1 Gbps.*

Resumo. *Identificação de tráfego é uma tarefa importante para provedores de serviço de internet (ISP), com o objetivo de otimizar a qualidade de serviço (QoS) para os usuários. Inspeção Profunda de Pacotes (DPI) é a técnica mais utilizada para realizar a identificação e se baseia em expressões regulares (ER) para representar padrões de tráfego. Dentre os vários problemas, o principal gargalo ao identificar tráfego em tempo real de alta velocidade é o casamento das ER, que costuma ser computacionalmente custoso. A fim de aumentar o desempenho de um DPI, este trabalho utiliza uma unidade de processamento gráfico (GPU). Nesse contexto, a solução proposta alcançou uma vazão de processamento de 114 Gb/s na GPU, além de cerca de 1 Gb/s de vazão total.*

1. Introdução

A identificação de tráfego exerce um papel importante no gerenciamento das redes dos Provedores de Serviço da Internet (ISPs) visando, principalmente, melhorar a qualidade de serviço (QoS) de alguns serviços ou aplicações. Nesse contexto, a inspeção de pacotes (DPI) é uma técnica que fornece informações precisas sobre o tráfego de uma rede monitorada. Embora outras técnicas sejam utilizadas (como análise de fluxos, por exemplo), sistemas DPI apresentam maior flexibilidade e precisão. Um dos maiores problemas enfrentados por esses sistemas é, entretanto, a necessidade de analisar todos os pacotes que trafegam por um ponto da rede, o que demanda um grande esforço computacional [Dainotti et al. 2012]. A fim de melhorar o desempenho de um sistema DPI, várias soluções baseadas em *hardware* foram desenvolvidas. Por exemplo, circuitos FPGA ou ASIC [Becchi et al. 2007] [Yu et al. 2006] mapeiam expressões regulares para representar assinaturas de aplicações bem conhecidas em seus

hardwares, processando rapidamente. Essas soluções são custosas e pouco flexíveis. Por esse motivo, alguns trabalhos utilizam CPUs tradicionais para analisar pacotes [Antonello et al. 2012]. Outra plataforma que tem sido utilizada é a GPU (*Graphics Processing Unit*) em conjunto, principalmente, com a arquitetura CUDA (*Compute Unified Device Architecture*) [Sanders et al. 2010]. As GPUs apresentam baixo custo, grande poder computacional e flexibilidade (quando comparados com analisadores feitos em *hardware*). Trabalhos recentes trazem soluções eficientes para melhorar sistemas DPI [Cascarano et al. 2010], além de técnicas de aprendizagem de máquina [Sharp 2008] e roteamento IP [Mu et al. 2010] aplicados nessa área.

Sistemas DPI baseados em GPU frequentemente utilizam técnicas de casamento de cadeias de caracteres. Dentre eles, destacam-se alguns trabalhos que utilizam algoritmos baseados em casamento exato de cadeias de caracteres [Szabo et al. 2010]. Para esse fim, pesquisadores criam estruturas de dados capazes de representar as expressões regulares (ERs) e armazená-las em memória. As ERs oferecem maior expressividade quando comparadas a cadeias fixas de caracteres, além de serem mais flexíveis. Para representar ERs, são criados autômatos finitos determinísticos (DFA) [Wang et al. 2011] ou autômatos finitos não-determinísticos (NFA) [Cascarano et al. 2010]. De maneira geral, o DFA apresenta um melhor desempenho (em termos de velocidade de processamento dos pacotes), sendo bastante utilizado em sistemas DPI. Diante desse cenário, o escopo deste trabalho será restrito à construção de um sistema DPI baseado em GPU para identificação de tráfego. Nesse caso, é importante notar que o sistema DPI será projetado para fazer uso do grande poder computacional de uma GPU, se diferenciando, nesse ponto, de trabalhos anteriores, que utilizaram apenas CPUs. Dessa forma, este trabalho propõe novas arquiteturas baseadas em GPU que buscam a otimização do processo de classificação de tráfego realizado pelo DPI. A solução proposta alcançou resultados expressivos, como uma vazão de processamento (volume total de dados na GPU) de 114 Gb/s na GPU, além de cerca de 1 Gb/s de vazão total (volume total de dados processado), e uma vazão de fluxos classificados correspondente a um volume de mais de 20 Gb/s.

Este trabalho apresenta, na Seção 2, a teoria básica sobre GPU e CUDA. Em seguida, na Seção 3, alguns trabalhos relacionados são apresentados e, na Seção 4, a arquitetura proposta é descrita. A metodologia de avaliação utilizada no desenvolvimento do trabalho é exposta na Seção 5. Os resultados da avaliação estão na Seção 6. Por fim, as conclusões e trabalhos futuros são apresentados na Seção 7.

2. GPU e a arquitetura CUDA

Originalmente concebidas para auxiliar na renderização de componentes gráficos, reduzindo o uso de recursos da CPU, as GPUs evoluíram, aumentando o seu poder computacional, e se tornaram úteis na área da computação científica. Além da programação direta e intuitiva, essas unidades de processamento gráfico podem ser utilizadas como unidades complementares às CPUs, ampliando suas capacidades [Sanders et al. 2010]. É possível utilizar uma GPU para diminuir a carga na CPU, aumentando o paralelismo e o desempenho final do sistema. Tais benefícios são possíveis devido à grande quantidade de núcleos de processamento em uma única placa, na ordem de centenas. Elas evoluíram com o passar do tempo, fazendo com que as novas gerações de placas gráficas se tornassem mais facilmente programáveis pelos desenvolvedores, mantendo o alto poder de processamento [Ryoo et al. 2008]. Nesse

contexto, a NVIDIA desenvolveu a arquitetura de programação (CUDA), voltada para a execução de tarefas que precisam ser realizadas em paralelo sem constantes interrupções. CUDA se caracteriza por ser uma extensão da linguagem C/C++, possuindo funcionalidades e diretivas para processar tarefas diretamente na GPU. Tais funcionalidades e diretivas são inicializadas através de um programa principal que executa na CPU, chamado de *host* [Sanders et al. 2010], capaz de invocar o conjunto de instruções que será executado na GPU denominado *kernel.*.

Para que a GPU seja eficiente, é necessário entender o paradigma SIMD (*Single Instruction Multiple Data*), para o qual ela foi projetada. Esse paradigma consiste na execução de um mesmo conjunto de instruções sobre múltiplos dados de entrada. Uma soma de dois vetores poderia ser feita em um único passo, obtendo o vetor resultado, por exemplo. Sendo composta por várias *threads* (ramificações paralelas da execução do programa), a colaboração entre elas deve ser a menor possível. O objetivo desse modelo é ter várias *threads* executando o mesmo conjunto de instruções simultaneamente, aumentando o paralelismo. Em CUDA, as *threads* são agrupadas em conjuntos de blocos, que são reunidos em conjuntos de *grids*. *Grids* e blocos são apenas *containers* lógicos criados para dividir os componentes de CUDA e facilitar o escalonamento das *threads*. Além disso, a arquitetura é composta por diversos processadores simétricos, chamados *streaming multiprocessors* (SM), capazes de executar diversos *grids* de blocos simultaneamente. *Threads* de um mesmo bloco compartilham o mesmo espaço de endereçamento, a região de memória compartilhada e os registradores do SM em que o bloco executa. Elas se comunicam unicamente através da memória compartilhada de cada bloco, não sendo possível *threads* de blocos distintos trocarem informações. Um conjunto de 32 *threads* é chamado de *warp*. Em resumo, o paradigma SIMD prevê que as *threads* executem o mesmo conjunto de instruções sem divergência entre elas, garantindo um bom desempenho, comparado com um paradigma utilizado na CPU.

3. Trabalhos relacionados

Embora a maioria dos sistemas DPI utilizem DFAs para representar suas ERs, (devido ao desempenho de processamento), os NFAs consomem muito menos memória. Em [Cascarano et. al. 2010], foi criado o iNFAnt, um mecanismo para realizar casamento de ERs em GPU baseado em NFAs. Nesse caso, o iNFAnt foi criado com o intuito de permitir que várias *threads* pudessem executar simultaneamente, sendo cada uma delas responsável por realizar uma transição para um determinado estado ativo. No entanto, devido às características de CUDA, o algoritmo proposto se torna bastante lento, pois é preciso que as *threads* sempre estejam sincronizadas ao longo da execução, o que não é previsto pelos autores. No melhor caso, o algoritmo alcançou uma vazão de 1 Gb/s, ao realizar o casamento de pacotes TCP com apenas duas ERs simples, em contraste com sistemas DPI reais que possuem centenas de assinaturas (como L7-Filter, Snort, e Bro).

Um dos primeiros trabalhos que tratou do casamento de expressões regulares utilizando uma GPU foi apresentado em [Smith et. al. 2009]. Eles realizaram uma investigação aprofundada visando encontrar a melhor plataforma para realizar casamento de ERs de forma eficiente com foco em sistemas DPI. Foram analisadas as plataformas CUDA, FPGA e arquiteturas baseadas em vários núcleos (*multicore*). Alguns testes com diferentes fluxos de controle de execução e estratégias de paralelismo em nível de pacote foram realizados. No entanto, nos experimentos realizados, o

desempenho do sistema era degradado quando os tamanhos dos pacotes eram diferentes. Os autores concluíram que os pacotes deviam possuir o mesmo tamanho (sendo normalizados), para que não houvesse divergência durante a execução das *threads*. Além disso, em [Smith et. al. 2009], o impacto da utilização de diferentes autômatos finitos para representar as ERs na memória da GPU foi investigado. Os resultados finais apontaram que a GPU alcançou um desempenho 9 vezes maior que um processador Pentium 4. Porém, eles não apresentaram os valores para a vazão geral do sistema DPI.

Em [Vasiliadis et. al. 2009], os autores desenvolveram o Gnort, uma versão do Snort que utiliza uma GPU para realizar o casamento de expressões regulares. No Gnort, apenas o casamento de ERs é realizado em uma GPU. A solução proposta armazena um conjunto de pacotes de rede em filas temporárias. Quando esta fila encontra-se cheia, ela é transferida em lote para a GPU para que a análise seja realizada. O Gnort mantém a fila com posições de memória de tamanho fixo. Os autores não mostraram qual foi a organização dos blocos e *threads* utilizados no algoritmo. O desempenho de melhor caso de execução para o Gnort foi de aproximadamente 16 Gb/s, para a vazão de processamento dos pacotes e a vazão total foi de 800 Mbps.

O Gregex [Wang et. al. 2011] é capaz de atingir uma vazão de processamento de 122 Gb/s e um desempenho geral para identificar pacotes de 25.6 Gb/s. Os autores utilizaram um subconjunto de assinaturas do Snort, agrupando-as em um único DFA para representar todo o conjunto. Entretanto, não foi informada qual técnica de agrupamento foi utilizada, assim como outros detalhes de implementação não foram descritos. O Gregex utiliza uma fila de pacotes normalizada, onde cada pacote ocupa um espaço de 2048 Bytes (2 kB). Caso um pacote tenha carga útil menor que 2 kB, o restante da posição de memória da fila para onde o pacote está sendo copiado é preenchida com zeros. Isso gera imprecisões durante o cálculo da vazão, já que os zeros adicionados não fazem parte do tráfego original, e não deveriam ser contabilizados.

Considerando a solução apresentada neste trabalho, apesar dos benefícios associados ao uso da arquitetura CUDA, a criação de uma solução eficiente para o casamento de expressões com ER não é uma tarefa simples. Para implementar algoritmos paralelos, é necessário explorar diferentes *layouts* de memória de CUDA e diferentes modelagens do problema. A partir da melhor utilização da memória, diversas otimizações podem ser implementadas para maximizar o desempenho da GPU. Neste trabalho, também foram realizadas otimizações através da transposição de memória e da utilização de memória fixa sem paginação. Além disso, este trabalho prevê a utilização de fluxos, a fim de alcançar uma maior vazão total referente ao volume agregado.

4. Arquitetura proposta

Este trabalho tem como objetivo aumentar a vazão final de sistemas DPI que utilizam ERs como assinaturas através de uma GPU. Para isso, foram criadas 3 arquiteturas diferentes, chamadas 1CTB (1-CUDA *Thread per Block*), 1CTP (1-CUDA *Thread per Packet*) e ITBF (Identificação de tráfego baseada em fluxo), que utilizam diferentes disposições e combinações de blocos e *threads* em CUDA para aplicar o conjunto de expressões regulares a vários pacotes simultaneamente. Em resumo, queremos achar a melhor configuração para processar paralelamente os pacotes de uma determinada rede.

Como pode ser observado na Figura 1, a arquitetura geral é composta pela interação entre *host* (CPU) e GPU. Quando o *host* precisa classificar um conjunto de

pacotes utilizando uma determinada ER, eles são previamente copiados para o espaço de endereçamento da GPU, uma vez que CUDA acessa posições de memórias locais, estejam elas na memória global, constante, compartilhada ou de textura. Além de copiar os pacotes, também deve ser indicado qual DFA deverá ser utilizado para realizar o casamento. A tabela de transição dos DFAs utilizados é copiada e armazenada na memória global da GPU durante a fase de inicialização do DPI. O conjunto de pacotes que será inspecionado é armazenado na memória global da GPU, durante a fase de processamento. Durante a execução do código no *host*, o *kernel* é chamado e a GPU inicia o processamento, já tendo sido informada sobre a quantidade de blocos e *threads* que deverão ser instanciados. A disposição das *threads* e blocos varia de acordo com a solução utilizada (1CTB, 1CTP, ITBF). Após o término da execução do *kernel*, os resultados são copiados da GPU para o *host*, a fim de que os pacotes que casaram com alguma ER possam ser identificados. Essa arquitetura geral foi desenvolvida através de um protótipo funcional, sem paralelizar atividades desempenhadas no *host*. Todo o tráfego encontra-se armazenado em disco, de modo que o protótipo executa em modo *off-line* (lendo o disco e efetuando a classificação com a GPU).

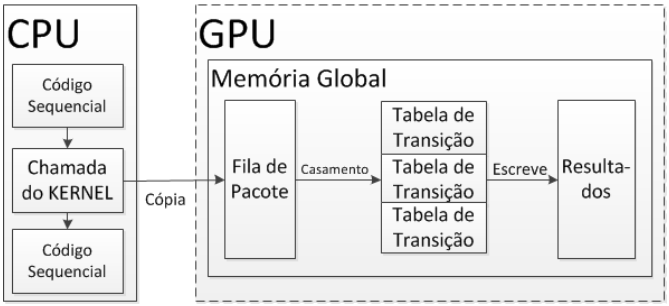

Figura 1. Arquiterura Geral de um sistema DPI utilizando uma GPU para realizar casamento de ER.

A Figura 2 ilustra a arquitetura 1CTB, onde é instanciado um único bloco CUDA por cada pacote da fila de entrada e cada um desses blocos possui apenas uma única *thread*. Ela é responsável por comparar o conteúdo de um único pacote com apenas um DFA, escrevendo o resultado final da classificação num vetor de resultados. Além da arquitetura 1CTB, também foi criada a arquitetura 1CTP, a qual cria várias *threads* por bloco, sendo cada uma das *threads* responsáveis por um único pacote. Dessa forma, várias *threads* são executadas em um mesmo bloco, porém cada uma trata uma posição diferente da memória global. Dessa forma, os acessos à memória podem ser disparados por *warps*, diminuindo a latência inerente ao processo.

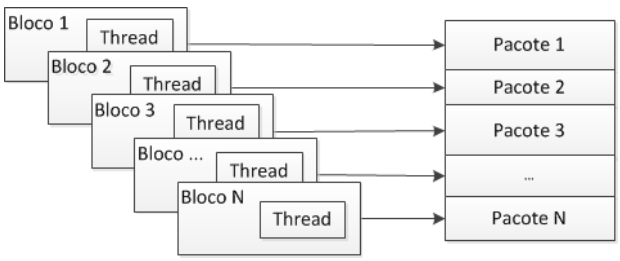

Figura 2. Solução 1 - Um *thread* por Bloco

A Figura 3 ilustra a solução 1CTP. A solução 1CTP baseia-se no correto posicionamento de cada uma das *threads* criadas através do conjunto de pacotes, de

modo a garantir que uma *thread* seja responsável apenas por classificar um único pacote. Percebe-se que a primeira *thread* do bloco 1 fica responsável por inspecionar o primeiro pacote, a segunda thread desse mesmo bloco é responsável pelo pacote 2 e consequentemente o *thread* N deste bloco é responsável pelo pacote N.

Por fim, a última versão criada denomina-se IBTF. Nessa solução, foi implementada a transposição de memória [Ruetsch et al. 2009], que é comumente utilizada na área de computação gráfica para aumentar o desempenho de algoritmos que executam em CUDA. A Figura 4 ilustra a solução IBTF. Vários estudos anteriores [Fernandes et al. 2009] [Bernaille et al. 2006] investigaram os benefícios obtidos ao realizar inspeção de pacotes agregando os dados em fluxos para identificar trafego, inspecionando apenas os primeiros bytes de cada fluxo. Mesmo analisando apenas parte dos dados trocados pelo canal de comunicação, os autores provaram que é possível obter uma boa classificação quanto às aplicações responsáveis pelos fluxos. No entanto, nenhum trabalho anterior analisou o impacto que utilizar essa técnica produz na vazão do sistema DPI.

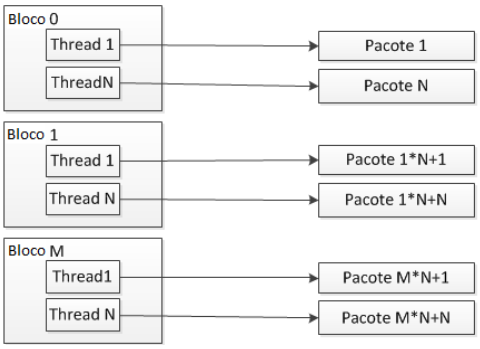

Figura 3. Solução 1CTP alocando um *thread* por pacote da fila de entrada.

Neste trabalho, essa técnica é aplicada através da arquitetura IBTF utilizando uma GPU para paralelizar o processamento. O IBTF aloca um conjunto de *threads* por bloco. Porém, ao invés de uma *thread* por pacote, uma *thread* para cada um dos fluxos que será classificado é definida.

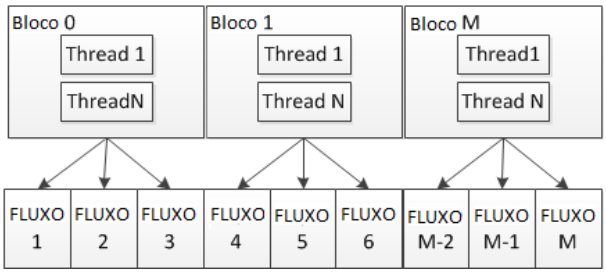

Figura 4. IBTF – Identificação de Tráfego Baseada em Fluxo

Como já reportado por estudos anteriores, é preciso que a fila de entrada seja formada por um vetor de dados uniforme, onde a área de memória ocupada por cada elemento é igual. Desse modo, o IBTF aloca um espaço fixo de dados por fluxo, onde os primeiros *K* bytes do fluxo são armazenados. A arquitetura IBTF também implementa a técnica de transposição de memória para diminuir o tempo de acesso a memória, distribuindo-os entre *warps*. Para que seja possível realizar a transposição, é necessário uma matriz quadrada. Desse modo, o IBTF trabalha com um vetor de fluxos *K* x *K*, contendo *K* fluxos, cada um com os *K* primeiros bytes de cada fluxo.

5. Metodologia de Avaliação

A fim de construir as soluções propostas, uma arquitetura Intel Quad-Core foi utilizada, onde cada núcleo é composto por um processador Core i7-2600. Além disso, foi utilizada a GPU NVIDIA GeForce GTX 480 com CUDA 2.0. A GPU GTX 480 é composta por 480 processadores e permite a criação de *kernels* com no máximo 65536 blocos, onde cada um dos blocos só é capaz de alocar 1024 *threads*. Para criar os cenários de avaliação, foi construído um gerador sintético de *traces* [Santos et al. 2011]. Esse gerador é capaz de criar arquivos de pacotes contendo carga útil segundo um conjunto de ERs, de modo que um sistema DPI possa encontrar ou não um casamento, de acordo com o *trace* gerado. Adicionalmente, este trabalho utilizou diferentes *traces* sintéticos de pacotes, assumindo cenários de melhor e de pior caso. Os cenários de melhor caso são aqueles em que um casamento irá ocorrer com alguma assinatura. Os cenários de pior caso são aqueles em que não ocorre casamento com nenhuma ER do conjunto. Nesse caso, foram gerados *traces* utilizando o gerador sintético de pacotes sobre um conjunto genérico de assinaturas, composto por 15 ER que especificam protocolos de aplicações comuns de rede, tais como HTTP, MSN Messenger, SSH, POP3, SMTP e RTSP. Além de realizar testes de melhor e de pior caso, também foi utilizado um *trace* real*,* identificado como caso médio, capturado em um enlace de alta velocidade de um provedor de serviços brasileiro, correspondente a uma coleta com 24 horas de duração, realizada entre novembro e dezembro de 2008. O *trace* completo possui 263 GB, porém apenas 75 GB foram utilizados nos testes, pois apenas pacotes IP, carregando segmentos TCP foram considerados. Mais informações sobre o *trace* utilizado podem ser encontradas em [Fernandes et al. 2009].

A principal métrica utilizada para medir o desempenho dos protótipos construídos neste trabalho é a vazão, que representa a quantidade de informações que a arquitetura proposta consegue processar por unidade de tempo. Nesse contexto, a vazão total (VT) trata do volume total de dados processado considerando um determinado período de tempo, contabilizando ainda as transferências de dados entre *host* e GPU. O presente trabalho trata não apenas da vazão de execução total da arquitetura proposta, mas também de outras medições a fim de obter diferentes tipos de vazão. Nesse caso, foram definidos outros dois tipos de vazão a serem avaliados.

A vazão de processamento (VP) caracteriza-se por ser a vazão para classificar todos os dados na GPU. A VP é dada pela razão entre a quantidade total de bytes copiados para a GPU e o tempo total necessário para processar e realizar o casamento de ERs. É importante salientar que esse tempo de processamento não inclui o tempo necessário para transferir os dados do *host* para a GPU e da GPU de volta para o *host*. Para medir a VP de maneira precisa, é necessário medir apenas o tempo gasto para processar os pacotes na GPU. Para isso, CUDA possibilita a utilização de eventos de medição. A Figura 5 retrata o ponto de medição utilizado para obter a duração do processamento dos pacotes na GPU e as diretivas de CUDA utilizadas.

A vazão de fluxo (VF) remete ao volume total de processamento do tráfego representado pelos fluxos. A VF será contabilizada apenas para a arquitetura IBTF, já que as demais não possuem agregação de tráfego por fluxos. De forma resumida, a VF é a razão entre o volume total de tráfego agregado dos fluxos que estão sendo classificados e o tempo necessário para processá-los na GPU. O mesmo intervalo de tempo utilizado no cálculo da VP é usado para a VF.

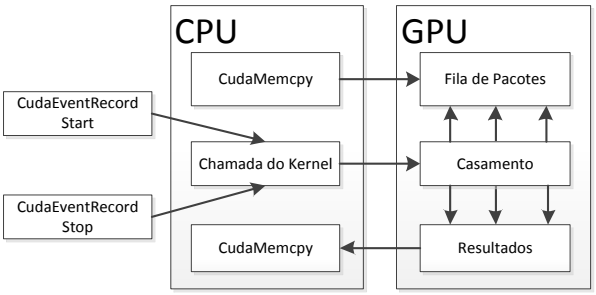

Figura 5. Ponto de medição para calcular a vazão de processamento

6. Resultados Experimentais

Conforme descrito, foram avaliadas as vazões de processamento e total para cada versão da arquitetura proposta, além da vazão de fluxo para a arquitetura IBTF. Assim, os resultados serão apresentados, analisados e uma breve discussão acerca dos resultados será exposta. Os testes foram realizados utilizando os *traces* de melhor e pior caso, além do *trace* do provedor de serviço, o qual representa o caso médio de execução. Em todos os casos, os valores obtidos para os diferentes tipos de vazão tratam-se de médias referentes a 10 execuções independentes.

O protótipo desenvolvido para testar a arquitetura 1CTB é bastante simples. Cada um dos pacotes é lido sequencialmente do *trace* e são, em seguida, armazenados em filas ainda na CPU. Quando uma fila possui uma quantidade pré-determinada de pacotes, ela é enviada para a GPU e o casamento de expressões regulares é executado. Na Figura 6-a, estão as médias da vazão de processamento (VP) referentes à execução da arquitetura 1CTB para diferentes tamanhos de fila de pacotes. A análise da Figura 6-a aponta que quanto maior a fila, maior é o valor da vazão de processamento para cada um dos *traces* utilizados. Como esperado, o *trace* do melhor caso de execução possui a melhor vazão de processamento e o *trace* de pior caso teve o menor desempenho.

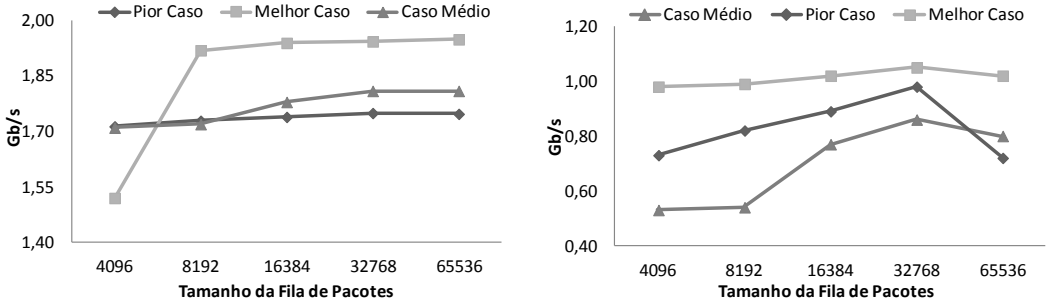

Figura 6. Vazão relacionada ao 1CTB. (a) Vazão de processamento da arquitetura 1CTB. (b) Vazão total para a arquitetura 1CTB.

No entanto, o aumento na vazão é muito pequeno quando a fila possui mais que 16384 pacotes. Isso se deve ao fato de que como a arquitetura utiliza apenas uma *thread* por bloco, não é vantajoso criar uma grande quantidade de blocos, pois CUDA escalona uma quantidade fixa de blocos por SMs. Nesse caso, a concorrência entre os blocos passa a ser alta, resultando em uma maior quantidade de mudanças de contexto e fazendo com que o ganho em desempenho não seja tão relevante, apesar do aumento. Assumindo os resultados obtidos para o *trace* de caso médio como sendo o comportamento esperado, a vazão total com a melhor configuração possível para a

arquitetura 1CTB foi de aproximadamente 1,8 Gb/s. Para a vazão total (VT), cujos resultados encontram-se ilustrados na Figura 6-b, o mesmo comportamento encontrado para a vazão de processamento ocorreu para valores de fila até 32768, ou seja, quanto maior a fila de pacotes, maior a vazão total. No entanto, para valores de fila maiores que 32768 pacotes, o desempenho começou a piorar. Isso se deve ao fato de que é preciso copiar a fila de pacotes do *host* para a GPU, acarretando em atrasos que reduzem o tempo total de processamento. A vazão total com a melhor configuração possível para a arquitetura 1CTB foi de 0,98 Gb/s com os resultados de caso médio.

A Figura 7-a ilustra os resultados obtidos para a vazão de processamento para a arquitetura 1CTP. É possível verificar que a variação da quantidade de blocos e *threads* alocados causa um forte impacto no desempenho, pois cada um dos blocos será responsável por um pacote diferente. De maneira geral, a vazão de processamento aumenta à medida que mais *threads* são alocados dentro de cada bloco, reduzindo a quantidade de blocos (a vazão de processamento aumenta quando a relação *threads*/bloco aumenta). Entretanto, o melhor resultado obtido foi com a configuração que utiliza 1024 blocos, cada um deles contendo 64 *threads*. Isso se deve ao fato de que CUDA escalona *threads* de um mesmo bloco em *warps* e um mesmo *warp* ocupa todos os processadores de um SM, provocando uma ocupação ótima de recursos. A melhor vazão de processamento alcançada, ao utilizar um *trace* de melhor caso, foi de aproximadamente 23 Gb/s. Como o desempenho esperado da arquitetura é aquele produzido pelo caso médio, a vazão de processamento para a arquitetura 1CTP é de aproximadamente 14 Gb/s.

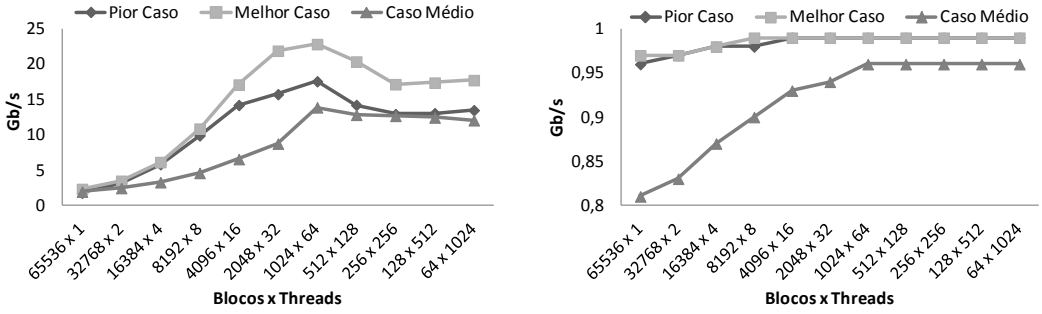

Figura 7. Vazão relacionada ao 1CTP. (a) Vazão de processamento para a arquitetura 1CTP. (b) Vazão total para a arquitetura 1CTP.

Os resultados obtidos para a vazão total na arquitetura 1CTP encontram-se na Figura 7-b. Conforme esperado, assim como nos resultados obtidos para a vazão de processamento, em todos os *traces* utilizados, quanto maior a quantidade de *threads* por bloco (a relação *thread*/bloco), maior a vazão total. Da mesma forma que na vazão de processamento, o melhor resultado obtido foi com a configuração que utiliza 1024 blocos, cada um com 64 *threads*. No entanto, para casos de teste contendo mais *threads* por bloco, a vazão total manteve-se a mesma, e não mais diminuiu como ocorreu para a vazão de processamento. Seguindo a mesma abordagem utilizada para os demais testes, o resultado considerado para a vazão total foi aquele encontrado na melhor configuração para o caso médio, que foi de aproximadamente 0,96 Gb/s.

A arquitetura IBTF utiliza a técnica transposição de memória para aumentar o desempenho no acesso à memória global, fazendo com que *threads* de um mesmo *warp* realizem acessos contíguos. O mesmo modelo é utilizado para avaliar as outras

arquiteturas, realizando testes de melhor caso, pior caso e caso médio. Nesse, caso a quantidade de blocos e *threads* foi modificada de modo que seja possível encontrar o conjunto ótimo para esses parâmetros. Devido ao uso da técnica de transposição, é preciso que a fila de fluxos seja uma matriz quadrada, ou seja, para fluxos de *K* bytes de dados, é preciso que existam *K* fluxos na fila. Para certo valor *K* de fluxos na fila, valores diferentes de *threads* e blocos foram aplicados. Percebe-se na Figura 8 que, para cada um dos valores de *K*, diferentes configurações produzem resultados distintos. Por exemplo, ao utilizar 8 kB de dados dos fluxos (Figura 8-d), a configuração que produziu a melhor vazão foi com 64 blocos, cada um contendo 128 *threads*. Já para testes onde foram inspecionados os 4 kB de dados (Figura 8-c), a melhor configuração foi de 64 blocos, com 64 *threads*. Os gráficos apontam que quanto maior for a quantidade de dados analisados de cada fluxo, maior a vazão de processamento. No entanto, podemos perceber que apenas na situação em que os primeiros 8 kB de dados foram analisados a vazão para o melhor caso é muito acima do pior caso e do caso médio.

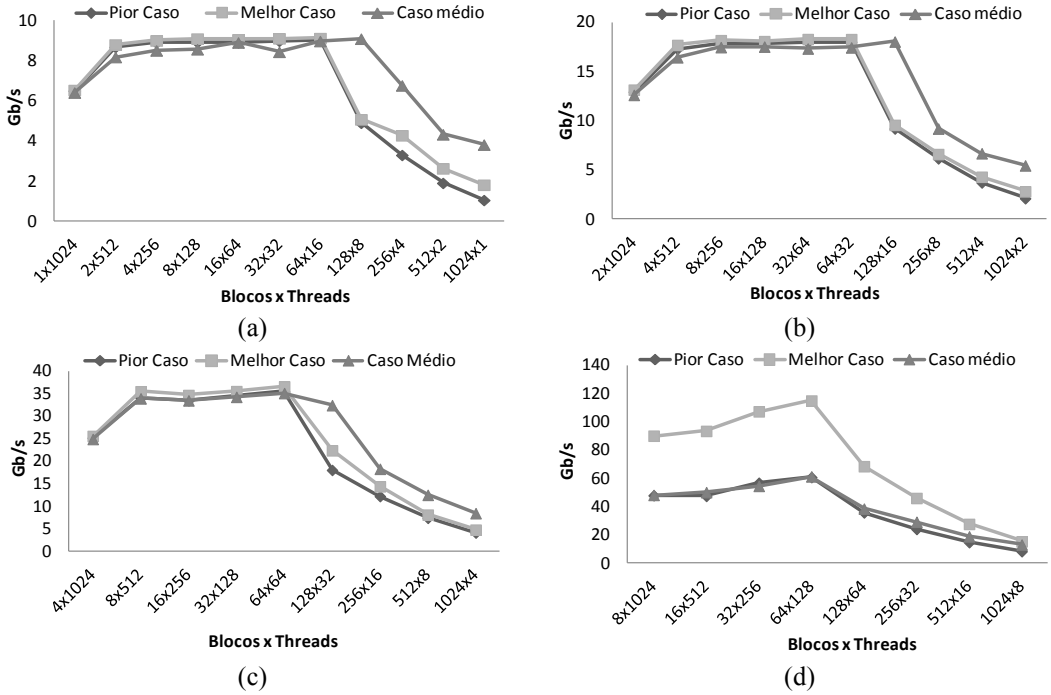

Figura 8. Vazão de processamento para a arquitetura IBTF. (a) Vazão para K = 1kB. (b) Vazão para K = 2kB. (c) Vazão para K = 4kB. (d) Vazão para K = 8kB.

Isso se deve ao fato de que o protótipo construído para executar os testes da arquitetura IBTF classifica 100% do tráfego do *trace* gerado sinteticamente para representar o melhor caso apenas quando os primeiros 8 kB estão presentes. Inspecionar menos que 8kB não é suficiente para conseguir identificar todo o *trace*, fazendo com que o pior caso, melhor caso e caso médio possuam processamento semelhantes. Porém, é importante frisar que mesmo no caso médio e no pior caso a vazão de processamento foi bastante próxima. Isso se deve ao fato de que grande parte do *trace* de caso médio não pode ser identificado, fazendo com que o cenário de pior caso tivesse desempenho semelhante.

Os gráficos da Figura 9 ilustram os resultados obtidos para a vazão de fluxos, realizando a mesma variação entre blocos e *threads* feita para avaliar a vazão de processamento, para cada um dos níveis e para cada um dos *traces*. Uma vez

armazenados em memória, os fluxos continuam possuindo apenas os seus valores de quantidade de pacotes e o seu volume de dados atualizados. Quando eles são classificados, todo o volume agregado é utilizado para os cálculos da vazão de fluxo. Esse é um limite teórico, pois depende que todos os fluxos estejam armazenados em memória e que haja um mecanismo eficiente. Esse mecanismo é capaz de lidar com todo o tráfego que chega até o ponto de medição, sem que haja perda ou descarte de pacotes, e de garantir que todos os fluxos não serão removidos da memória até que sejam identificados. Caso haja esse mecanismo e a GPU possam receber os fluxos na mesma taxa utilizada em nossos experimentos, podemos dizer que o sistema DPI é capaz de lidar com a mesma vazão de dados reportada pela vazão de fluxos. Assim, com o aumento no tamanho da quantidade de *bytes* analisados por fluxo, a vazão de fluxo manteve-se estável, onde apenas o *trace* de caso médio obteve uma vazão de fluxo maior para K = 8 kB. Isso acontece porque os *traces* gerados sinteticamente possuem fluxos com volume de dados fixos e menores. Para os testes utilizando 1 kB, 2 kB e 4 kB, a vazão de fluxo obtida para os testes de pior caso foi maior do que aquela encontrada para os testes de melhor caso. Isso se deve ao fato de que o *trace* que representa o pior caso é composto apenas por pacotes grandes de 1500 *bytes*, fazendo com que seus fluxos possuam um volume agregado maior. Já o *trace* de melhor caso possui fluxos contendo tanto pacotes de 1500 quanto 64 *bytes*, fazendo com que a vazão de fluxo seja menor. Já para os testes de melhor caso onde os primeiros 8 kB foram inspecionados (o *trace* é totalmente identificado), o resultado da vazão de fluxo foi acima dos 20 Gb/s.

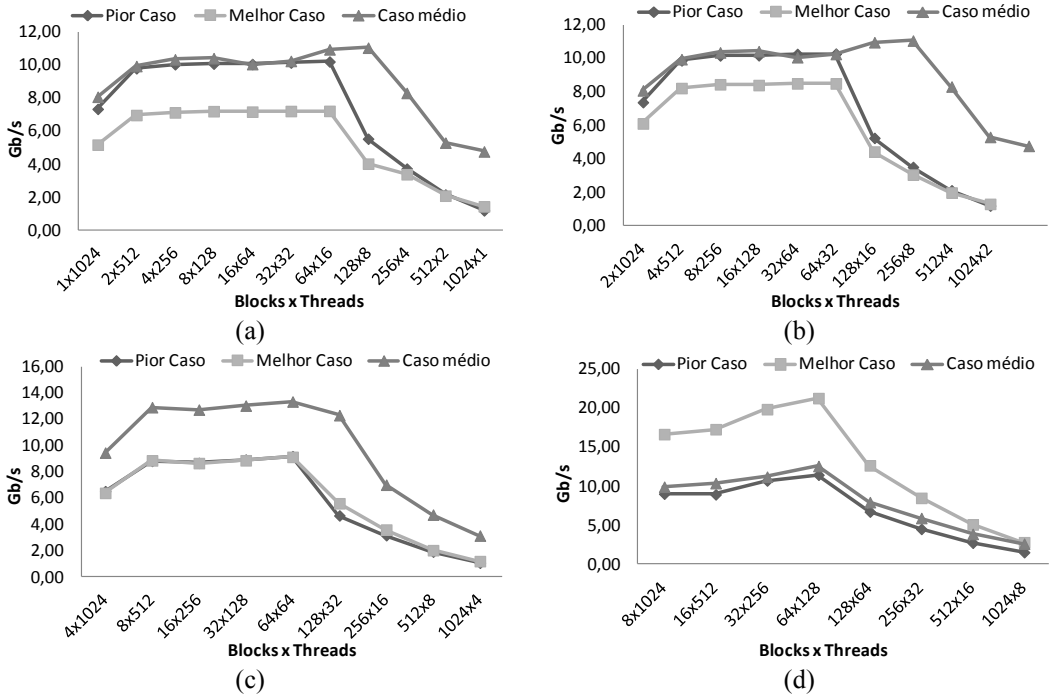

Figura 9. Vazão de Fluxo para a arquitetura IBTF. (a) Vazão para K = 1kB. (b) Vazão para K = 2kB. (c) Vazão para K = 4kB. (d) Vazão para K = 8kB.

Os gráficos da Figura 10 ilustram os valores da vazão total para cada um dos testes realizados, variando a quantidade K de fluxos e o *trace* utilizado. É importante frisar que o arquivo de *trace* do melhor caso de execução possui uma mistura de pacotes grandes e pequenos. Além disso, o *trace* que representa o melhor caso possui vários fluxos contendo pacotes de 1500 *bytes* e outros com apenas pacotes de 64 *bytes*, o que

acarreta um aumento na quantidade de pacotes que não serão enviados a GPU para processamento, causando um impacto negativo na vazão total de processamento. O *trace* do caso médio possui fluxos com diferentes tamanhos, o que também causará impacto negativo na vazão total. Já o *trace* do pior caso possui apenas fluxos grandes, contendo 1500 *bytes*, os quais sempre serão enviados para processamento. Analisando os gráficos somos capazes de perceber que, ao contrário do esperado, a vazão total foi maior, para todos os valores de *K*, no *trace* de pior caso. Apenas nos testes realizados utilizando os primeiros 8 kB de dados é que a vazão total do *trace* de caso médio e do *trace* de pior caso assumiram valores próximos. Os testes realizados com o *trace* de melhor caso apontaram o pior desempenho na vazão total de processamento devido à presença de vários fluxos pequenos, os quais não possuem ao menos *K* bytes de dados. O mesmo ocorre para os resultados obtidos com o *trace* de caso médio.

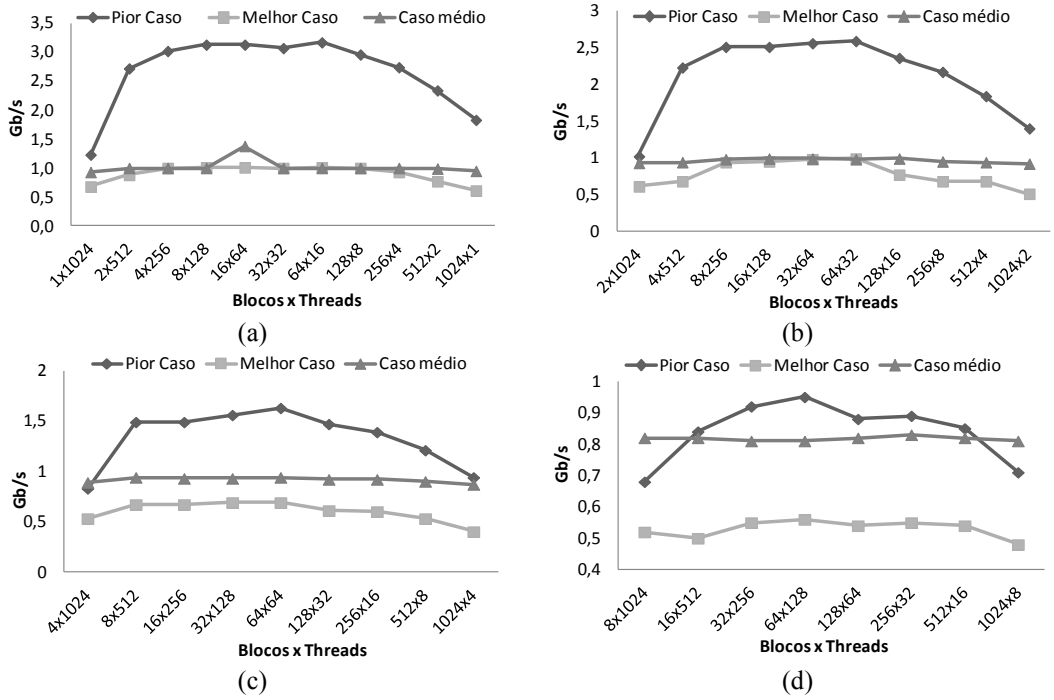

Figura 10. Vazão total para a arquitetura IBTF. (a) Vazão para K = 1kB. (b) Vazão para K = 2kB. (c) Vazão para K = 4kB. (d) Vazão para K = 8kB.

O motivo que explica os resultados obtidos para a vazão total terem sido muito abaixo dos encontrados para a vazão de processamento e para a vazão de fluxo é o fato de que há um custo computacional referente a tarefas de E/S ao realizar leitura dos pacotes do disco. Essa tarefa consiste em copiar o fluxo de dados para a GPU, em seguida aguardar o fim do processamento e copiar os resultados. Nesse caso, ainda existe todo processamento adicional para montar a estrutura de dados, adicionar e remover novos fluxos e para realizar as tarefas de manutenção dessa tabela em memória. Dessa forma, o custo adicional encontrado é um fator diretamente relacionado ao não paralelismo adotado na construção do protótipo. Segundo [Amdahl 1967], para que seja possível medir o ganho em desempenho obtido ao utilizar o conceito de paralelismo para aumentar o desempenho de uma determinada tarefa é preciso considerar o ganho em desempenho obtido ao realizar essa tarefa em paralelo, adicionada ao tempo que continua sendo realizado de maneira sequencial. É possível concluir que a arquitetura que ofereceu o melhor ganho em desempenho para a vazão de processamento foi a IBTF, produzindo um aumento significativo em relação ao 1CTB,

atingindo aproximadamente 114 Gb/s. Conforme apresentado na Tabela 1, percebe-se que a vazão de processamento alcançada é próxima ao máximo obtido na literatura.

Já para a vazão total, a solução que produziu o maior ganho de desempenho foi a 1CTP, com cerca de 1 Gb/s como pode ser observado em Tabela 2.

Tabela 1. Comparação da vazão de processamento de diferentes abordagens

Arquitetura	Vazão de Processamento
1CTB	1,8 Gb/s
1CTP	14 Gb/s
IBTF 8 kB (Caso Médio)	60 Gb/s
IBTF 8 kB (Melhor Caso)	114 Gb/s
Gnort	16 Gb/s
Gregex	122 Gb/s

Isso acontece porque, como apenas o processamento dos pacotes foi acelerado, mantendo as leituras dos pacotes de um disco, realizando cópias da fila de pacotes para a GPU e em seguida copiando os resultados de volta para o *host*, o ganho em desempenho na vazão total não acompanhou o aumento encontrado para a vazão de processamento. Por fim, é importante destacar que a vazão de fluxo, obtida pelo IBTF, de aproximadamente 20 Gb/s é uma abordagem inovadora, que permite que essa vazão seja correspondente a um grande volume real de dados. Em outras palavras, essa vazão deve ser comparada com a vazão total, representando um ganho superior a 20 vezes em comparação com os resultados apresentados na Tabela 2.

Tabela 2. Vazão total para diferentes abordagens

Arquitetura	Vazão total
Gnort	800 Mbps
1CTP	950 Mbps
IBTF 8 kB	930 Mbps

7. Conclusão e trabalhos futuros

O presente trabalho teve como objetivo descrever e desenvolver arquiteturas capazes de realizar identificação de tráfego utilizando expressões regulares em alta velocidade, utilizando o poder computacional proporcionado pelo uso de uma GPU. Além de utilizar uma GPU, as ERs foram transformadas em DFAs de modo que uma determinada cadeia de entrada pudesse ser avaliada através da execução de um único algoritmo de casamento, utilizando uma única tabela de transição que representa várias assinaturas simultaneamente.

O trabalho apresentou resultados que apontaram uma vazão de processamento de aproximadamente 114 Gb/s e uma vazão total acima de 960 Mb/s. É interessante destacar que a vazão obtida através da agregação de fluxos alcançou uma vazão real de mais de 20 Gb/s, mesmo considerando os custos computacionais relativos ao uso da GPU, superando de maneira satisfatória outras abordagens. Como trabalho futuro, consideramos explorar outras funcionalidades da arquitetura CUDA, que oferece um mecanismo para otimizar o agendamento de tarefas para serem processadas na GPU,

denominado *streams*. O uso de s*treams* permite que vários *kernels* sejam chamados em instantes diferentes, sendo uma ótima solução a ser adotada para arquiteturas *multithread*.

Referências

Amdahl, G., "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", AFIPS Conference Proceedings (30): 483–485, 2007.

Antonello, R. et al., “Deterministic Finite Automaton for Scalable Traffic Identification: the Power of Compressing by Range”, Proc. of IEEE NOMS 2012, Hawai, USA.

Becchi, M. e Crowley, P., “An improved algorithm to accelerate regular expression evaluation”, In Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems 2007. NY, USA, 145-154.

Bernaille, L. et al."Traffic classification on the fly" SIGCOMM CCR 36,2(2006),23-26.

Cascarano, N. et al., “iNFAnt: Nfa Pattern Matching on GPGPU Devices”, ACM SIGCOMM Computer Communication Review 40, 5, 20-26, 2010.

Dainotti, A. et al., "Issues and future directions in traffic classification," IEEE Network, vol.26, no.1, pp.35-40, January-February 2012.

Fernandes, S. et al., "Slimming Down Deep Packet Inspection Systems," INFOCOM Workshops 2009. IEEE Press, Piscataway, NJ, USA, 61-66.

Mu, S. et al., “IP routing processing with graphic processors”, Design, Automation and Test in Europe, 2010, pp. 93-99.

Ruetsch, G. e Micikevicius, P., “Optimizing matrix transpose in cuda”, NVIDIA 2009.

Ryoo, S. et al. “Optimization principles and application performance evaluation of a multithreaded GPU using CUDA”, PPoPP '08. ACM, New York, NY, USA, 73-82.

Sanders, J. e Kandrot, E. “CUDA by example: an introduction to general-purpose GPU programming”, Addison Wesley (2010).

Santos, A. et al., “High-Performance Traffic Workload Architecture for Testing DPI Systems”, IEEE GLOBECOM 2011, vol., no., pp.1,5, 5-9 Dec. 2011.

Smith, R. et al. "Evaluating GPUs for network packet signature matching" ISPASS 2009, vol., no., pp.175,184, 26-28 April 2009.

Szabo , G. et al., “Traffic Classification over Gbit Speed with Commodity Hardware” IEEE J. Communications Software and Systems, vol. 5, 2010.

Sharp, T., “Implementing decision trees and forests on a GPU”, in Proc. ECCV, 2008, Vol. 5305 (2008), pp. 595-608.

Yu, F. et al. ,“Fast and memory-efficient regular expression matching for deep packet inspection”, ANCS '06, ACM, New York, NY, USA, 93-102, 2006.

Vasiliadis, G. et al., "Regular expression matching on graphics hardware for intrusion detection" Proceedings of the 12th RAID, Saint-Malo, France, 2009, pp. 265-283.

Wang, L. et al. "Gregex: GPU Based High Speed Regular Expression Matching Engine" IMIS 2011, vol., no., pp. 366, 370, June 30 2011-July 2.