Compactação do Algoritmo de Comparação de *Strings* do Snort para o uso na Memória Compartilhada de GPUs

José Bonifácio da Silva Júnior[1], Edward David Moreno[1], Ricardo Ferreira dos Santos[2]

[1]Departamento de Computação da Universidade Federal de Sergipe – DCOMP/UFS

[2]Departamento de Computação da Universidade Federal de Viçosa

boni14_gto@hotmail.com, {edwdavid,cacauvicosa}@gmail.com

Resumo. *A tarefa de comparar assinaturas de ataques com pacotes de redes em um Intrusion Detection System (IDS) consome bastante tempo de CPU. Para amenizar esse problema, tem-se tentado paralelizar o motor de comparação dos IDSs transferindo sua execução da CPU para a GPU. Este artigo mostra o processamento em paralelo dos dados no algoritmo de comparação de string Aho-Corasick e propõe compactar a Tabela de Transição de Estados desse algoritmo a fim de possibilitar o uso dele na memória compartilhada. A paralelização foi feita através da plataforma CUDA da NVIDIA e executada nas diversas memórias da GPU. O algoritmo AC foi compactado e executado na memória compartilhada, alcançando, em seu melhor resultado, um ganho de desempenho de 73% em relação às outras memórias da GPU e o algoritmo compactado chegou a ser 56 vezes mais rápido que sua versão serial. Com isso, pode-se perceber que o uso da compactação na memória compartilhada torna-se uma solução adequada para acelerar o processamento de IDSs que necessitem de agilidade na busca por padrões.*

1. Introdução

A tecnologia da informação (TI) tem sido cada vez mais determinante para o sucesso de empresas e organizações ao redor do mundo, sendo as redes de computadores como um dos seus principais meios de transmissão de dados. Juntamente com o aumento da importância da TI, cresceu também a necessidade de proteção do seu maior patrimônio, a informação.

Uma das formas de proteção passa pelo uso do IDS, uma ferramenta baseada em assinaturas que analisa os cabeçalhos dos pacotes e inspeciona as cargas, comparando-os com um grande conjunto de regras, ou seja, uma coleção de assinaturas de ataques conhecidos, tais como: vírus, *worms*, *spyware* ou código malicioso [Jaiswal 2014].

O Snort é um dos IDSs mais utilizados, sendo um software *open source* para UNIX e Windows. Ele é capaz de detectar quando um ataque está sendo realizado e, baseado nas características do ataque, alterar ou remodelar a configuração do sistema de acordo com as necessidades, e alertar o administrador do ambiente sobre esse ataque [Santos 2005]. Ele monitora o tráfego da rede pacote a pacote e em tempo real para verificar se o pacote que chega na interface coincide com algumas das suas assinaturas pré-configuradas.

Para alcançar isso, o Snort v2.9.7.3 usa o algoritmo de Aho-Corasick (AC) [Snort

Team 2015] para fazer a comparação do *payload* do pacote com a coleção de assinaturas. Isto é um problema, pois só essa atividade consome cerca de 70 a 80% do tempo de processamento [Jaiswal 2014]. Em redes de alta velocidade essa comparação pode sobrecarregar a CPU, fazendo-a deixar de executar os outros processos necessários para a execução do Snort ou de outra aplicação que estiver em execução no host.

Essas falhas fazem da paralelização da comparação de *strings* utilizando as *Graphic Processing Unit* (GPUs) uma solução adequada para o problema, já que as GPUs têm um maior poder de computação do que as CPUs, como pode ser visto em alguns trabalhos. Por exemplo, Lin, C. et al. (2013) paralelizaram o processamento do algoritmo de comparação de *string* Aho-Corasick (AC) e alcançaram uma velocidade 74,95 vezes maior que a versão serial do mesmo algoritmo. Tran, N. et al. (2012) paralelizaram os dados de entrada do algoritmo AC e obtiveram uma melhoria de 15,72 vezes na velocidade de comparação de *strings* em relação à versão serial. Já o trabalho de Thambawita, D. et al. (2014) mostrou que se o texto onde serão procurados os padrões for maior que 40000 bytes (o que tende a acontecer quando um *host* IDS é colocado em uma rede de alta velocidade) o desempenho da GPU supera o da CPU. Os resultados desses trabalhos mostraram que houve ganho significativo da GPU em relação à CPU, o que, por si só, serve como justificativa para continuar as pesquisas nesta linha de estudo.

A paralelização do processamento de dados no algoritmo AC já mostra ser um caminho natural para acelerar a comparação de *strings* do Snort. Porém, com o passar do tempo, novos ataques vão surgindo e consequentemente o número de assinaturas tende a aumentar. Isso leva a uma preocupação a mais com a limitação da quantidade de memória das GPUs, principalmente das memórias mais velozes, como é o caso da memória compartilhada, por exemplo. Sendo assim, a compactação do algoritmo AC na GPU é uma alternativa para superar esse problema.

O presente artigo tem como objetivo principal realizar a paralelização do processamento dos dados do algoritmo AC, dividindo os dados a serem analisados em partes e processando-os em paralelo, explorar a hierarquia de memórias da GPU a fim de verificar onde melhor se encaixam os dados e o algoritmo e, também, aplicar uma compactação na STT do algoritmo a fim de possibilitar sua execução na memória compartilhada.

O artigo está organizado da seguinte forma: a Seção 2 mostra a hierarquia de memórias de uma GPU genérica. A Seção 3 traz o funcionamento do algoritmo AC e como o mesmo pode ser implementado em uma GPU. A Seção 4 mostra os trabalhos relacionados à paralelização do processamento de dados do algoritmo AC em GPUs. A Seção 5 explica a compactação eficiente desenvolvida neste trabalho. Os resultados experimentais utilizando a compactação são detalhados na Seção 6, seguidos da conclusão na Seção 7.

2. *Graphic Processing Unit* (GPU)

Nesta seção é dada uma breve descrição da arquitetura de uma GPU, com foco na sua hierarquia de memórias.

As Unidades de Processamento Gráfico são dispositivos de processamento de elementos gráficos introduzidos na década de 1980 para descarregar os processamentos gráficos relacionados às *Central Processing Units* (CPUs) [Kouzinopoulos e Margaritis

2008]. As GPUs modernas são programáveis e possuem processadores de *streams* capazes de fazerem computação de alto desempenho.

Pode-se ver pela Figura 1 que uma GPU possui alguns tipos de memórias que podem ser utilizadas em uma aplicação paralela. A memória global é uma memória localizada no *off-chip* DRAM e é através dela que a CPU se comunica com a GPU. A memória compartilhada fica dentro de cada bloco de *thread* e é compartilhada entre os *threads* em execução no bloco. O tempo de acesso coincide de perto com o tempo de acesso de um registrador, portanto, é uma memória muito rápida [Tran, et al. 2012].

Existem também as memórias constante e de textura localizadas na área *off-chip* DRAM. Estas memórias possuem, respectivamente, a cache constante e a cache de textura no chip, que podem armazenar dados somente leitura [Tran, et al. 2012].

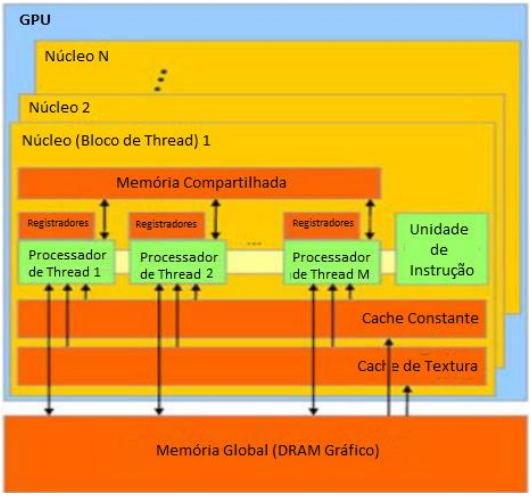


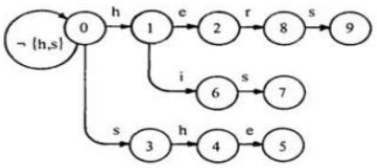
Figura 1: Arquitetura geral de uma GPU [Adaptada de Tran, et al. 2012].

3. Algoritmo Aho-Corasick e sua Implementação em GPUs

Nesta seção é dada uma breve descrição do algoritmo original desenvolvido por Alfred Aho e Margaret Corasick.

O algoritmo invoca três funções [Tran, et al. 2012]: a função *goto* (g), a função de falha (f), e a função de saída (*output*). A Figura 2 mostra as funções utilizadas pela máquina AC para o conjunto de padrões {"he", "she", "his", "hers"}:

- O grafo orientado da Figura 2 (a) representa a função *goto* (onde ¬ ('h', 's') denota todos os outros símbolos de entrada diferentes de 'h' e 's'). A função *goto* mapeia um par consistindo de um estado e um símbolo de entrada dentro de um estado ou uma mensagem de falha. Por exemplo, a borda marcada como h do estado 0 para o estado 1 indica que o g (0, 'h') = 1. A ausência de uma seta indica falha. A máquina AC tem a propriedade que g (0, σ) ≠ falha para qualquer símbolo de entrada σ.
- A função de falha mapeia um estado para outro estado. Ela é consultada sempre que a função *goto* relata uma "falha".
- A função de saída mapeia um conjunto de palavras-chave para a saída de acordo os estados finais alcançados.



(a) A função *goto*

i	1	2	3	4	5	6	7	8	9
$f(i)$	0	0	0	1	2	0	3	0	3

(b) A função de falha

i	*output* (i)
2	{he}
5	{she, he}
7	{his}
9	{hers}

(c) A função de saída

Figura 2: Funções usadas no algoritmo AC [Adaptada de Tran, et al., 2012].

Como exemplo, assuma o texto "ushers". A máquina AC trabalha da seguinte maneira: iniciando com o estado 0, a máquina volta para o estado 0 uma vez que g(0,'u')=0. Pela mesma razão, a máquina entra nos estados 3, 4 e 5 sequencialmente enquanto processa os caracteres 's', 'h' e 'e' (g(0,'s')=3, g(3,'h')=4, g(4,'e')=5) e emite a saída, indicando que ela encontrou as palavras "she" e "he". Depois a máquina avança para o próximo caractere de entrada 'r'. Uma vez que g(5,'r')=falha, a máquina entra no estado 2, já que f(5)=2 (Figura 2b). Então, uma vez que g(2,'r')=8 e g(8,'s')=9, a máquina AC entra no estado 9 e emite a saída "hers".

Para finalizar a discussão sobre o algoritmo AC, Aho e Corasick (1975) afirmam que a função *goto* pode ser armazenada das seguintes formas: em um vetor bidimensional (muitas vezes chamada de *State Transition Table* ou pela sigla STT), onde o acesso é em tempo constante, mas exige mais espaço de armazenamento, ou em uma lista linear, que necessita de menos espaço de armazenamento, porém o tempo de acesso para *g(s,a)* é proporcional ao número de valores que não levam a máquina a uma falha no estado *s*. Eles também sugerem uma solução mista, onde os estados mais frequentemente usados, tal como o estado 0, sejam armazenados na tabela e os menos usados, na lista linear.

No entanto, em se tratando de GPU, a estrutura de dados utilizada tende a ser o vetor bidimensional, já que o modelo de memória da GPU é restritivo quando se trata de implementar estruturas de dados bem conhecidas, tais como listas ligadas e árvores [Jacob e Brodley 2006].

4. Trabalhos Relacionados

Alguns trabalhos desenvolvidos recentemente são apresentados nesta seção a fim de mostrar como andam as pesquisas na área da paralelização de dados no algoritmo AC.

Tran, N. et al. (2012) apresentaram uma nova técnica de paralelização na qual armazenam de forma eficiente os dados do texto de entrada e os dados de referência (padrões de comparação) nas memórias da GPU. A partir dos dados de referência eles criaram uma STT (Figura 3) e colocaram esses dados na memória de textura de modo que a parte ativamente usada da STT pôde ser armazenada na cache de textura. A abordagem reduziu significativamente as latências médias de acesso à memória para

carregar ambos os dados de entrada e os dados de referência, e levou a melhorias de desempenho para o algoritmo AC, fazendo com que o algoritmo tivesse uma velocidade de processamento de 15,75 vezes maior se comparado com a versão serial em um de seus experimentos.

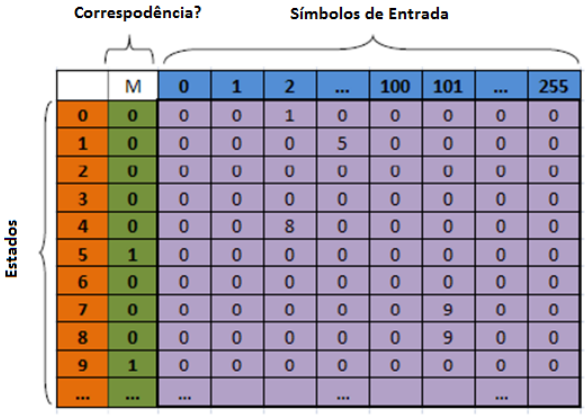


Figura 3: Ilustração da Tabela de Transição de Estado [Tran et al. 2012].

Lin, C. et al. (2013) criaram o PFAC, uma versão que faz a comparação de *strings* de forma paralela utilizando o algoritmo AC. O PFAC não utiliza as transições de falhas existentes no algoritmo original. Sua STT foi colocada na memória de textura, porém eles afirmaram que como o estado zero é o mais acessado, as transições referentes esse estado foram inseridas na memória compartilhada. O algoritmo alcançou um *speedup* de 74,95 vezes se comparado com a versão serial do AC.

Um ponto interessante a se observar é que nem sempre é possível colocar a STT completa na memória devido à limitação de espaço de armazenamento. LIN et al. (2010), que implementaram a STT na memória compartilhada, dividiram a STT por grupos de ataques. Villa et al. (2012), implementaram na memória de textura, mas se a STT for muito grande ela é lançada por partes.

Além disso, Villa et al. (2012) aproveitaram a própria STT para dizer se o próximo estado era um estado final ou não. Para isso, cada posição da STT tem 32 bits, sendo que os 31 primeiros indicam o próximo estado e o último indica um *flag* de estado final. Dessa forma retira-se a necessidade de construir outra tabela para informar se é um estado final (ou estado de aceitação).

O espaço de armazenamento limitado também fez com que alguns autores compactassem a STT. Tanto Pungila (2013, 2015) como Bellekens (2014) fizeram uma compressão no DFA e usaram a técnica de mapeamento de bits para representá-lo. Pungila (2013) usou comparação de prefixos afirmando que um prefixo de profundidade igual a 8 é suficiente para produzir uma taxa de falso positivo de apenas 0,0001%. Já Pungila (2015) usou o algoritmo de compressão chamado Lempel-Ziv-Welch, conhecido como LZW.

Na técnica de mapeamento de bits cada nó do DFA tem um bitmap associado (um *array* de 8 células de 32 bits cada) representando os 256 valores do alfabeto ASCII. Cada bit no bitmap do nó que é setado como 1 representa uma transição válida para aquele nó. Com isso, os autores não necessitaram mais representar todas as combinações possíveis entre estados e caracteres de entrada, conforme é feito na STT

comum.

5. Compactação da *State Transition Table*

Nesta seção a estrutura da compactação é apresentada juntamente com os passos necessários para compactar uma STT.

Como pôde ser visto na Figura 3 uma STT comum possui poucos valores diferentes de zero. A fim de retirar os zeros desnecessários da STT, a compactação descrita a seguir foi desenvolvida e executada em pacotes de redes reais, mostrando que seu poder de compactação possibilita que milhares de regras sejam armazenadas na memória compartilhada da GPU, aumentando a velocidade da procura por padrões.

A tabela de transição passará a ser representada por três vetores, conforme a explicação seguinte (ver Figura 4):

a) Vetor de Índices (VI): O valor armazenado neste vetor, ou seja, VI[i], significa o índice inicial no vetor VE correspondente ao estado i. O tamanho de VI será igual à quantidade de estados, sendo que o índice i de VI corresponde ao estado i da máquina AC. Além disso, se ∀ caractere de entrada α, o estado i levar a máquina a uma falha, VI[i] é igual a -1. Possui relação 1 para N com VE, sendo que um estado pode ter várias entradas que o levem a outro estado válido, mas uma entrada leva a máquina apenas a um estado.

b) Vetor de Entrada (VE): armazena todas as entradas que levam um estado qualquer para outro estado válido em conformidade com os valores de VI. Possui relação 1 para 1 com VS.

c) Vetor de Saída (VS): armazena o estado de saída devido à entrada VE de mesmo índice.

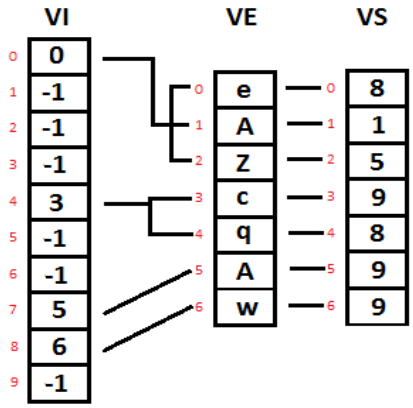


Figura 4: STT compactada

Para exemplificar como a transição de estado ocorre nos vetores de compactação, suponha que entre na máquina AC o caractere ‘q’ e o estado atual seja 4. VI[4] indica que os caracteres de entrada que podem levar a máquina a um estado válido se iniciam no índice 3 de VE. Agora, deve-se obter o valor posterior a VI[4] diferente de -1, que nesse caso é 5 (armazenado em VI[7]). Diminuindo 5 de 3 obtém-se a quantidade de caracteres de entradas que deve ser analisada em VE, ou seja, duas entradas. Portanto, a análise em VE deve ser feita iniciando no índice 3 até o índice 4. Partindo do índice 3

de VE, nota-se que ‘c’ é diferente de ‘q’. Avançando para o índice 4, encontra-se o caractere ‘q’. Como estamos no índice 4 de VE, o valor correspondente em VS, ou seja, VS[4], é o estado 8.

Caso no exemplo anterior em vez de ‘q’ a entrada fosse ‘y', não haveria uma saída correspondente em VS. Neste caso ocorreria uma falha na função *goto* e o vetor de falhas deveria ser consultado para o estado 4.

Como o Snort possui milhares de regras, uma máquina AC real pode chegar a ter milhares de estados, fazendo com que a compactação seja bastante necessária, quer seja para colocar a máquina AC na memória compartilhada, quer seja para colocá-la em memórias maiores que não suportem uma STT tão grande, necessidade que ocorreu nos experimentos de Villa et al. (2012), que apesar de implementarem a STT na memória de textura, tiveram a necessidade de lançá-la por partes caso ela fosse muito grande.

A compactação descrita acima é bem semelhante às Matrizes Esparsas CSR (*Compressed Sparce Row*). VE e VS são idênticos. A diferença está em VI, que nas matrizes esparças

Através das regras *free* do Snort, foi construída uma STT a partir de mais de cinco mil regras, gerando um autômato de 48317 nós. Podemos ver na Figura 5 que a compactação proposta neste artigo reduz significativamente a quantidade de KB necessários para armazenar a STT.

Abordagem	Quantidade de kilobytes necessários
STT comum	24738
Bitmapeamento	1540
Compactação desse trabalho	145

Figura 5: Redução da quantidade de dados através da compactação.

Para construir os vetores VI, VE e VS a partir de uma determinada STT, o seguinte algoritmo deve ser seguido:

1) Declare a variável *contador_de_entradas* e atribua zero a essa variável;
2) Inicie um loop a partir da linha zero da STT;
3) Percorra todas as células da linha *i* da STT da esquerda para a direita;
4) Se o valor da célula for diferente de -1, armazene o valor da célula em *VS*, armazene o valor da coluna *j* em *VE* e incremente a variável *contador_de_entradas*;
5) Depois de percorrer a linha *i* completamente, armazene *contador_de_entradas* em *VI[i]* (caso este contador não tenha sido incrementado para a linha *i*, armazene -1 em *VI[i]*);
6) Vá para a próxima linha da STT;
7) Volte para o passo 2 até que toda STT tenha sido percorrida.

6. Experimentos e Resultados

Esta seção mostrará como os experimentos foram realizados e quais resultados foram obtidos.

Foram utilizadas duas GPUs e uma CPU para executar o algoritmo AC a fim de comparar o desempenho em cada uma das plataformas, conforme as seguintes descrições:

- GPU TITAN X: GPU com 3584 cores CUDA e memória global de 12 GB. Seu host possui um processador Intel Core i5 1,2 GHz da 6ª geração, memória RAM de 8 GB em um sistema operacional Linux 3.19.0-80-generic #88~14.04.1-Ubuntu;
- GPU TESLA K20: GPU com 2496 cores CUDA e memória global de 5 GB. Seu host possui um processador Intel Xeon Ten-Core E5-2660v2 de 2.2 GHz, memória RAM de 64 GB em um sistema operacional RedHat 6.4;
- CPU: Intel Core i3-4005U 1,70 GHz. Opera com uma memória RAM de 4 GB em um sistema operacional Windows 8.1 de 64 bits.

Os dados de entrada foram divididos em pacote sintético e pacote real. O pacote sintético é composto por 1000 caracteres de um texto em inglês. Por ter 1000 caracteres, seu tamanho bruto é de 1 KB. Para simular dados maiores, este texto foi replicado a fim de obter o tamanho desejado. Por exemplo, no caso do experimento de 1MB de texto de entrada, o texto bruto foi replicado 1000 vezes. Cada thread processa um dos textos replicados, ou seja, no caso de 1 MB serão necessárias 1000 threads para processar o texto completo. O pacote real foi formado pelos caracteres dos pacotes obtidos na rede da Universidade Federal de Sergipe, com o auxílio do software Wireshark. Seu tamanho bruto de 1 MB. Esse texto foi dividido em 10 partes de tamanhos iguais para simular a chegada de vários pacotes de dados para serem processados paralelamente. Neste texto também houve replicações para se alcançar o tamanho de texto desejado. Para o experimento de 10 MB, por exemplo, esse texto foi replicado 10 vezes e processado por 100 *threads*.

Algumas configurações de bloco e grid da GPU foram testadas e, para a placa TESLA K20, a melhor configuração foi um bloco de tamanho 100 e grid de tamanho 10000. Já para a placa TITANX, a melhor configuração foi bloco igual a 1000 e grid igual a 1000.

Quatro versões do algoritmo AC foram criadas, sendo uma versão serial e três versões com processamento paralelo. Das versões paralelas, em uma versão a STT é colocados na memória global, em outra versão a STT é colocada na memória de textura e outra onde a STT é compactada com a técnica descrita na Seção 5 e colocada na memória compartilhada da GPU. O pacote de entrada foi colocado na memória global em todas as versões paralelas.

As métricas utilizadas nos experimentos são descritas abaixo:

- Tempo de Execução do Sistema (TE): É a soma do tempo de execução do *kernel* com o tempo de transferência de dados entre CPU e GPU medido em milissegundos;
- *Throughput* (TT): É a taxa de transferência do *kernel* medida em caracteres por segundo (Mcps);
- Percentual de Execução do *Kernel* (% TE): É a porcentagem do Tempo de Execução do Sistema necessária para a execução do kernel.

Pode-se ver na Tabela 1 que as três versões da GPU executaram de forma mais rápida que a CPU em todos os ensaios com pacotes sintéticos na TESLA K20. Nota-se também que entre pacotes de 10 MB e 50 MB a versão de memória compartilhada já consegue superar as outras duas versões paralelas.

Tabela 1: Resultados com pacote sintético - Tesla K20.

Tamanho do Texto de Entrada	Serial		Global			Textura			Compartilhada		
	TE	TT	TE	TT	% TE	TE	TT	% TE	TE	TT	% TE
1 MB	31	32,26	2,5	500,00	80	2,5	500,00	80	2,5	500,00	80
10 MB	250	40,00	9	3333,33	33	9	3333,33	33	9	3333,33	33
50 MB	1328	37,65	68	1470,59	50	67	1515,15	49	62	1785,71	45
125 MB	3453	36,20	171	1453,49	50	171	1453,49	50	155	1785,71	45
250 MB	6750	37,04	346	1461,99	49	342	1461,99	50	307	1838,24	44
500 MB	13297	37,60	684	1461,99	50	699	1461,99	49	613	1845,02	44
1000 MB	26818	37,29	1368	1459,85	50	1371	1461,99	50	1224	1855,29	44

Além disso, nota-se que até 10 MB o GCT (Ganho da memória Compartilhada em relação à memória de Textura) e o GCG (Ganho da memória Compartilhada em relação à memória Global) – ambos calculados através do Tempo de Execução - foram iguais a zero e as abordagens poderiam ser usadas sem distinção. Porém se o texto de entrada tiver um tamanho maior ou igual a 50 MB é válido usar a abordagem de memória compartilhada para ter uma execução mais veloz, podendo chegar a um ganho de velocidade acima de 14% em relação às outras duas abordagens.

Já na placa TITANX o desempenho da abordagem compactada se mostrou ainda mais eficiente. Diferentemente da placa Tesla K20, nesta placa o ganho da versão compartilhada em relação às outras abordagens já começa a aparecer a partir de 1 MB e a partir daí vai aumentando. Com 1 GB de texto de entrada, a memória compartilhada obteve um ganho de desempenho de 73% em relação às versões de memória de textura e memória global. Além disso, em termos de tempo de execução, a versão compactada foi 56 vezes mais rápida que a versão serial quando o tamanho do texto de entrada foi 1 GB, como pode ser visto na Tabela 2.

Tabela 2: Resultados com dados sintéticos - Titanx.

Tamanho do Texto de Entrada	Serial		Global			Textura			Compartilhada		
	TE	TT	TE	TT	% TE	TE	TT	% TE	TE	TT	% TE
1 MB	31	32,26	1,87	729,93	73	1,85	740,74	73	1,52	980,39	67
10 MB	250	40,00	5,35	7407,41	25	5,46	6849,32	27	5,16	8620,69	22
50 MB	1328	37,65	42	2173,91	55	42	2173,91	55	26	7142,86	27
125 MB	3453	36,20	97	2450,98	53	96	2500,00	52	61	8333,33	25
250 MB	6750	37,04	201	2293,58	54	200	2314,81	54	122	8620,69	24
500 MB	13297	37,60	402	2252,25	55	402	2314,81	54	238	8928,57	24
1000 MB	26818	37,29	823	2183,41	56	823	2217,29	55	476	9090,91	23

Ao se usar os pacotes reais o desempenho de todas as versões diminuíram se comparado aos pacotes sintéticos devido aos dados apresentarem uma assimetria maior, fato que é ruim para o sincronismo das *threads* da GPU. Porém, os resultados obtidos continuaram satisfatórios, como pode ser visto no experimento com a placa TESLA K20 na Tabela 3 e com o TITAN X na Tabela 4. Em todos os ensaios as GPUs superaram a CPU. Na placa TESLA K20 a versão compartilhada foi mais veloz que a versão global em todos os ensaios, porém só foi mais rápida que a versão de textura quando o texto de entrada teve um tamanho entre 200MB e 500MB.

Tabela 3: Resultados com dados reais – Tesla K20.

Tamanho do Texto de Entrada	Serial		Global			Textura			Compartilhada		
	TE	TT	TE	TT	% TE	TE	TT	% TE	TE	TT	% TE
10 MB	375	26,67	268	38,31	97	246	41,84	97	265	38,76	97
50 MB	1906	26,23	296	190,84	89	275	208,33	87	293	193,05	88
100 MB	3828	26,12	331	380,23	79	310	414,94	78	327	386,10	79
200 MB	7641	26,17	449	729,93	61	388	793,65	65	399	760,46	66
500 MB	19062	26,23	736	1333,33	51	670	1533,74	49	656	1607,72	47
1000 MB	38329	26,09	1527	1182,03	55	1610	1128,67	55	1506	1283,70	52

Na placa TITANX, a versão compartilhada superou as outras duas versões paralelas em todos os ensaios, chegando a ter um ganho de 12,1% em relação à memória de textura e 14,9% em relação à memória global. Além disso foi 48 vezes mais rápida que sua versão serial.

Tabela 4: Resultados com dados reais – Titanx.

Tamanho do Texto de Entrada	Serial		Global			Textura			Compartilhada		
	TE	TT	TE	TT	% TE	TE	TT	% TE	TE	TT	% TE
10 MB	375	26,67	105	99,01	96	98	106,38	96	92	113,64	96
50 MB	1906	26,23	120	495,05	84	113	531,91	83	108	561,80	82
100 MB	3828	26,12	140	970,87	74	134	1041,67	72	128	1098,90	71
200 MB	7641	26,17	177	1923,08	59	170	2061,86	57	165	2173,91	56
500 MB	19062	26,23	456	1838,24	60	445	1923,08	58	397	2369,67	53
1000 MB	38329	26,09	905	1883,24	59	897	1926,78	58	831	2202,64	55

Os códigos-fontes desenvolvidos neste trabalho podem ser acessados através do link https://gist.github.com/anonymous/4957a9bc119be1c0514f26eca6e63266.

7. Conclusões

Este artigo mostrou que a comparação serial de padrões utilizada em softwares de segurança da informação como o IDS Snort tem se tornado cada vez mais problemática para a CPU devido a grande quantidade de dados para processar. Como uma solução viável, as GPUs fornecem uma estrutura de processamento de dados paralela e uma hierarquia de memórias que fazem aumentar a velocidade do processamento dos dados.

Sendo assim, foram extraídos 18 ataques das regras free do Snort a fim de construir uma STT do algoritmo Aho-Corasick para ser executada nas memórias das GPUs e, particularmente, compactá-la para ser usada na memória compartilhada.

Nos testes com dados sintéticos, as duas placas GPUs utilizadas tiveram ganhos de velocidades consideráveis podendo chegar a um ganho de 73% com a placa TITANX na abordagem compactada na memória compartilhada se comparada com as abordagens não compactadas nas memórias global e de textura. Se comparado com a CPU, o ganho da versão compactada é maior ainda, podendo ser 56 vezes mais rápida. Nos teste com dados reais, as abordagens tiveram ganhos menores (muito devido à aleatoriedade dos dados) se comparados com os dados sintéticos, mas evidenciou-se que mesmo assim os

ganhos são suficientes para justificar o uso da versão compactada, podendo chegar a 14,9% também na placa TITANX. Se comparada com a CPU, a versão compactada executou 48 vezes mais rápida.

Como trabalhos futuros, sugere-se que sejam retirados os conflitos de banco da versão compartilhada, já que isto deve aumentar consideravelmente o desempenho desta versão. Por fim, pode-se também fazer a integração do algoritmo desenvolvido com o Snort. Uma sugestão seria criar dois *buffers* de 200 MB, por exemplo. Quando o primeiro *buffer* estiver com os 200 MB das *strings* onde serão procurados os padrões, ele seria enviado para a GPU processar. Enquanto isso, o segundo *buffer* começaria a ser preenchido e quando estivesse completo (o processamento do primeiro já deveria ter acabado) seria enviado para a GPU e o processo recomeçaria no primeiro buffer.

Referências

Aho, A.; Corasick, M. (1975). "Efficient string matching: an aid to bibliographic search", Communications of the ACM, v.18 n.6, p.333-340, June.

Bellekens, X. J. A.. et al. (2014). "A Highly-Efficient Memory-Compression Scheme for GPU-Accelerated Intrusion Detection Systems". ACM: SIN '14 Proceedings of the 7th International Conference on Security of Information and Networks. [s. L.], p. 302-310. September.

Cuda. "CUDA: Programación Paralela Facilitada". http://www.nvidia.com.br/object/cuda_home_new_br.html.

Jacob, N.; Brodley, C. (2006). "Offloading IDS Computation to the GPU". The 22nd Annual Computer Security Applications Conference, pp 371-380, December.

Jaiswal, M. (2014). "Accelerating Enhanced Boyer-Moore String Matching Algorithm on Multicore GPU for Network Security". International Journal of Computer Applications, Vol. 97 – No. 1, 2014. http://research.ijcaonline.org/volume97/number1/pxc3896934.pdf.

Kouzinopoulos, C. S.; Margaritis, K. G. (2008). "String Matching on a multicore GPU using CUDA". The 13th Panhellenic Conference on Informatics, pp 14-18, September.

Lee, C.; Lin, Y.; Chen, Y. (2015). "A Hybrid CPU/GPU Pattern-Matching Algorithm for Deep Packet Inspection". PLoS ONE 10(10): e0139301, October.

Lin, C. et al. (2013) "Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs". IEEE Transactions on Computers, Vol. 62, pp 1906-1916, October.

Lin, C. et al. (2010). "Accelerating String Matching Using Multi-threaded Algorithm on GPU". 2010 IEEE Global Telecommunications Conference (GLOBECOM 2010), pp 1-5, December.

Pungila, C. (2013). "Hybrid Compression of the Aho-CorasickAutomaton for Static Analysis in Intrusion Detection Systems". Springer. [s. L.], p. 1-10, January.

Pungila, C.; Negru, V. (2015). "Real-Time Hybrid Compression of Pattern Matching Automata for Heterogeneous Signature-Based Intrusion Detection". Springer Link. Berlim, p. 65-74, May.

Pungila, C.; Reja, M.; Negru, V. (2014). “Efficient parallel automata construction for hybrid resource-impelled data-matching”. Future Generation Computer Systems, Vol. 36, pp 31-41, July.

Santos, B. R. (2005). “Detecção de Intrusos Utilizando o Snort”. 83 f. Monografia (Especialização) - Curso de Pós-Graduação Latu Sensu em Administração de Rede Linux, Departamento de Computação, Universidade Federal de Lavras, http://www.ginux.ufla.br/files/mono-BrunoSantos.pdf, September.

Snort Team. (2015). “SNORT Users Manual 2.9.7: The Snort Project”. 265 p, https://www.snort.org/#documents, June.

Thambawita, D. R. V. L. B.; Ragel, R.; Elkaduwe, D. (2014). “To Use Or Not To Use: Graphics Processing Units (GPUs) For Pattern Matching Algorithms”. The 7th International Conference on Information and Automation for Sustainability (ICIAfS), pp 1-4.

Tran, N.; Lee, M.; Hong, S.; Shin, M. (2012). “Memory Efficient Pararellelization for Aho-Corasick Algorithm on a GPU”. The 14th International Conference on High Performance Computing and Communications, pp 432-438.

Villa, O.; Chavarría-miranda, D. G.; Tumeo, A. (2012). “Aho-Corasick String Matching on Shared and Distributed-Memory Parallel Architectures”. IEEE Transactions on Parallel and Distributed Systems, Vol. 23, pp 436-443.