Revista de Sistemas de Informação da FSMA
n. 18 (2016) pp. 58-65

SISTEMAS DE INFORMAÇÃO
http://www.fsma.edu.br/si/sistemas.html

TRILHA ESTUDANTIL

Acelerando a Comparação de *Strings* do IDS Snort através da Programação Paralela: um Mapeamento Sistemático da Literatura sobre o Algoritmo Aho-Corasick Paralelizado

José Bonifácio da Silva Júnior, Edward David Moreno, Maria Augusta Silveira Netto Nunes

Departamento de Computação da Universidade Federal de Sergipe – DCOMP/UFS

Abstract*— The Intrusion Detection System (IDS) needs to compare the contents of all packets arriving at the network interface with a set of signatures for indicating possible attacks, a task that consumes much CPU processing time. In order to alleviate this problem, some researchers have tried to parallelize the IDS's comparison engine, transferring execution from the CPU to GPU. This paper identifies and maps the parallelization features of the Aho-Corasick algorithm, which is used in Snort to compare patterns, in order to show this algorithm's implementation and execution issues, as well as optimization techniques for the Aho-Corasick machine. We have found 147 papers from important computer science publications databases, and have mapped them. We selected 22 and analyzed them in order to find our results. Our analysis of the papers showed, among other results, that parallelization of the AC algorithm is a new task and the authors have focused on the State Transition Table as the most common way to implement the algorithm on the GPU. Furthermore, we found that some techniques speed up the algorithm and reduce the required machine storage space are highly used, such as the algorithm running on the fastest memories and mechanisms for reducing the number of nodes and bit maping.

***Index Terms*— Aho-Corasick, string matching, IDS, Systematic Literature Mapping, parallel processing.**

Resumo*—Um Sistema de Detecção de Intrusão (IDS) necessita comparar o conteúdo de todos os pacotes que chegam na interface da rede com um conjunto de assinaturas que indicam possíveis ataques, tarefa esta que consome bastante tempo de processamento da 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 tem como objetivo identificar e mapear o estado da arte sobre as características da paralelização do algoritmo Aho-Corasick, usado no IDS Snort para a comparação de padrões, a fim de mostrar as questões de implementação e execução do algoritmo, bem como técnicas de otimização para a máquina de Aho-Corasick. Foram mapeados pelo método de mapeamento sistemático 147 artigos de importantes bases de dados da área de computação. Destes, 22 foram selecionados e analisados para compor os resultados do artigo. Baseado nisto, a análise dos artigos mostrou, dentre outros resultados, que a paralelização do algoritmo AC é uma tarefa recente e que a Tabela de Transição de Estados é a forma mais comum de implementar o algoritmo na GPU. Além disso, verificou-se que algumas técnicas para acelerar o algoritmo e diminuir o espaço de armazenamento da máquina são usadas, como a execução do algoritmo em memórias mais rápidas, a redução de números de nós e o mapeamento de bit.

***Palavras-chave*—Aho-Corasick, comparação de *strings*, IDS, mapeamento sistemático da literatura, processamento paralelo.**

Artigo submetido em 08 de julho de 2016.
J. B. S. J. e-mail: boni14_gto@hotmail.com.
E. D. M. e-mail: edwdavid@gmail.com.
M. A. S. N. N. e-mail: gutanunes@gmail.com.

I. 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.

Para combater o acesso não autorizado aos sistemas de TI ou falhas ocasionadas por ataques de hackers, os profissionais da área de segurança da informação utilizam várias ferramentas, cada uma com uma função específica. Uma dessas ferramentas é o IDS (*Intrusion Detection System*), que tem a finalidade de detectar uma série de ataques na rede.

Um IDS é 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).

Um dos IDSs mais utilizados é o Snort, um software *open source* para UNIX e Windows. O Snort 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 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 Unidades de Processamento Gráfico (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 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) também paralelizaram o 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 trabalhos apresentam melhoras de desempenho com alto grau de divergência. Entretanto, fica claro que em todos os casos houve ganho significativo, o que, por si só, serve como justificativa para perseguir esta linha de estudo.

O presente artigo tem como objetivo realizar um mapeamento sistemático (SLM, *Systematic Literature Mapping*) do estado da arte para mostrar de que forma os pesquisadores da área têm paralelizado e executado o algoritmo AC em GPUs e se os mesmos têm adotado técnicas de otimização para tornar a máquina AC paralela mais eficiente. Para isso, foram mapeados os artigos de importantes bases de dados da área de computação.

Assim, esse artigo está organizado da seguinte forma: a Seção 2 apresenta uma revisão sobre o algoritmo de Aho-Corasick. A Seção 3 traz o Método adotado nesse mapeamento. Já a análise dos Resultados é apresentada na Seção 4. Na Seção 5, é apresentada a Conclusão seguida pelas Referências.

II. ALGORITMO AHO-CORASICK E SUA IMPLEMENTAÇÃO EM GPUS

Nesta seção será dada uma breve descrição do algoritmo original desenvolvido por Alfred Aho e Margaret Corasick e de como as GPUs podem ser usadas para implementar eficientemente esse algoritmo.

O algoritmo AC tradicional é capaz de comparar múltiplos padrões simultaneamente percorrendo uma máquina de estado especial chamada máquina de Aho-Corasick, que difere de um Autômato Finito Determinístico comum, pois introduz uma nova transição definida como transição de falha para reduzir as transições de saída de cada estado (LIN, 2013).

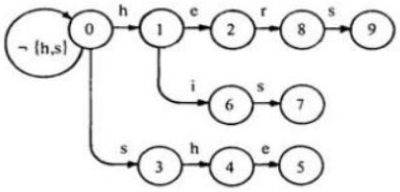
O algoritmo invoca três funções (TRAN, et al., 2012): a função *goto* (g), a função de falha (f), e a função de saída (*output*). A Figura 1 mostra as funções utilizadas pela máquina AC para o conjunto de padrões {"he", "she", "his", "hers"}:

- O grafo orientado da Figura 1 (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.

Como exemplo, assuma o texto “ushers”. A máquina AC da Figura 1 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 1b). 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 Tabela de Transição de Estados 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.



(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

Fig. 1. Funções usadas no algoritmo AC (Tran, et al., 2012).

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).

III. Método do Mapeamento Sistemático

Nesta seção apresenta-se o método de mapeamento sistemático realizado para mapear o estado da arte acerca das técnicas utilizadas na paralelização do algoritmo AC por meio da plataforma de computação paralela da Nvidia, CUDA (2016).

Segundo PETERSEN et al. (2008), o método de mapeamento sistemático corresponde à elaboração das questões de pesquisas, que definem o escopo da pesquisa; à realização da busca para estudos primários por meio da *string* de busca, que deve ser capaz de trazer todos os artigos dentro do escopo escolhido; à seleção dos estudos relevantes pelo uso de critérios de inclusão e exclusão dos artigos; à extração de dados e ao mapeamento e análise dos resultados, os quais são expostos a seguir.

A. Questões de Pesquisa

As seguintes questões de pesquisa foram elaboradas a fim de se atingir os objetivos propostos:

Q1) Qual o ano de publicação de cada artigo?

Q2) Quais são as métricas utilizadas para expor os resultados da execução do AC paralelo?

Q3) O autor faz a implementação da STT proposta por Aho-Corasick em sua forma original para representar a máquina AC?

Q4) Em qual memória da GPU a máquina AC é colocada?

Q5) Alguma técnica de compactação ou otimização é utilizada na máquina AC?

Q6) Em qual memória da GPU o texto de entrada é colocado?

B. Estratégias de Busca e Seleção

Para a execução da busca foi selecionada a base de dados do SCOPUS, uma vez que a mesma reúne artigos das revistas mais importantes na área da computação, associados a editoras e organizações profissionais como o IEEE, a ACM, ou a Elsevier, entre outras. Já para ter acesso aos artigos na íntegra, foi usado o portal de periódicos que a CAPES disponibiliza através do link *http://www.periodicos.capes.gov.br* e o site de busca do Google através do link *http://www.google.com.br*.

O SCOPUS disponibiliza alguns filtros de busca na sua página web. Sendo assim, para evitar que artigos fora do contexto entrassem nos resultados dessa primeira etapa do mapeamento, a busca foi executada utilizando o filtro de título, resumo e palavras-chave.

Inicialmente a *string* de busca foi composta apenas pelas palavras *"gpu"* e *"aho-corasick"* com o intuito de trazer a maior quantidade possível de artigos. Porém, a busca retornou apenas 27 artigos. Para melhorar o resultado, a sigla *"AC"* também foi inserida na *string*. Além disso, palavras como *"CUDA"*, que é a plataforma de programação paralela, e *"STT"*, que é o principal objeto de estudo desse trabalho, também foram inseridas. Já a inclusão de *"State Transition Table"* não alterou a quantidade de artigos retornados. Notou-se ainda que a palavra *"parallel"* tinha potencial para trazer artigos relevantes, mas que junto com *"AC"* e *"STT"* trazia mais de 7000 artigos e a maioria relacionados à área da engenharia elétrica.

Com isso, o termo de busca ficou definido como: *(gpu AND (aho-corasick OR ac OR stt)) OR (cuda AND (aho-corasick OR ac OR stt)) OR (parallel AND Aho-Corasick)*.

A busca dos artigos foi realizada entre 10 e 12 de junho de 2016. Neste intervalo temporal, a base de dados do SCOPUS retornou 147 artigos. Os títulos desses artigos podem ser consultados através do seguinte *link*:
https://gist.github.com/anonymous/d9e3bf7c005d0eda813def235ac3c5ec.

Com a finalização da busca, teve-se início o processo de filtragem dos artigos encontrados com base nos critérios de seleção.

C. Critérios de Seleção

Alguns critérios de inclusão e critérios de exclusão foram definidos a fim de filtrar os artigos relevantes desse mapeamento sistemático. Os critérios de inclusão foram:

1) Foram incluídos os artigos onde os autores paralelizaram o algoritmo AC em uma GPU através da plataforma CUDA e usou-o para fazer comparação de *strings*.
2) Foram incluídos os artigos que apresentaram a descrição da implementação do algoritmo ou de alguma técnica de otimização na máquina AC a fim de melhorar o desempenho da mesma.

A confirmação dos critérios de inclusão foi dada a partir da análise do resumo e da conclusão de cada um dos artigos encontrados.

O critério de exclusão adotado nesse mapeamento apenas exclui os artigos duplicados.

Após a aplicação dos critérios de inclusão e exclusão, restaram 22 artigos para fazerem parte dos estudos primários.

Após a seleção dos artigos, foi realizada uma análise aprofundada dos mesmos para responder às questões de pesquisa do Item *III.A*.

IV. Análise Dos Resultados

Nesta seção as questões de pesquisa delineadas no Item *III.A* são respondidas. As duas primeiras questões mostram aspectos gerais da pesquisa realizada, como: a quantidade de artigos por ano e o que se tem medido com os experimentos. Essas primeiras questões podem ser usadas como uma forma de justificar o uso de uma determinada métrica ou que pesquisas nessa área são atuais ou não, por exemplo.

As quatro questões posteriores conseguem responder importantes aspectos da implementação e execução do algoritmo paralelizado, como o uso de uma tabela para representar a máquina AC, localização dessa tabela, questões de otimização e localização do texto de entrada.

Q1) Qual o ano de publicação de cada artigo?

Com essa questão pode-se ver que as pesquisas relacionadas ao Aho-Corasick paralelo são pesquisas recentes devido ao período de publicações encontrado, o qual ocorreu de junho de 2010 a outubro de 2015. Esse era um resultado esperado uma vez que o uso de GPUs para aplicação de propósito geral (aplicação não gráfica) também é recente, justificando, assim, a pequena quantidade de artigos encontrados.

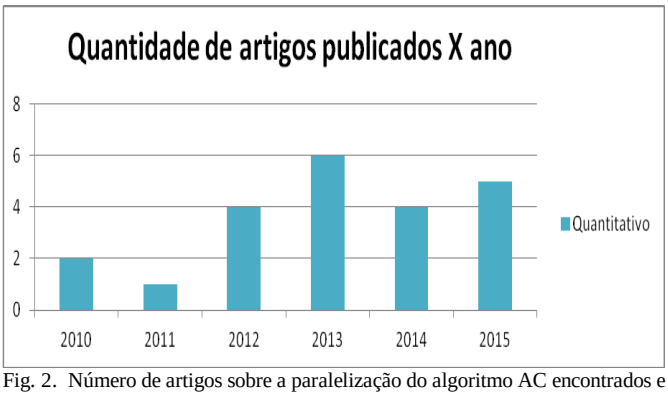


Fig. 2. Número de artigos sobre a paralelização do algoritmo AC encontrados e seus anos de publicação.

Ainda vale frisar que o quantitativo mostrado no gráfico da Figura 2 traz apenas artigos que passaram pelos filtros de inclusão e exclusão do mapeamento sistemático que foi descrito na Seção 2.

Q2) Quais são as métricas utilizadas para expor os resultados da execução do AC paralelo?

Nota-se pelo gráfico da Figura 3 que os autores têm analisado a quantidade de dados processados na GPU através da taxa de transferência, representada normalmente em Gbps ou Mbps, mostrando a quantidade de bits que o algoritmo desenvolvido consegue executar por segundo. Com isso, é possível, por exemplo, saber se o algoritmo suportará determinada velocidade de rede ou se ocorrerá perda de dados.

Métricas diretamente relacionadas à velocidade como o tempo de execução e o *speedup* (ganho de velocidade de um algoritmo em relação a outro) também são frequentemente usadas, já que a maior motivação de se usar a GPU é tentar diminuir o tempo que o algoritmo precisará para executar determinada quantidade de dados.

Outras métricas também apareceram com menos frequência, como: consumo de energia (Kwh) analisado por TAKAHASHI e INOUE (2012), já que este é um dos pontos negativos da GPU; quantidade de memória utilizada (KB) analisada por KOUZINOPOULOS et al. (2015) para mostrar a eficiência da sua compressão no DFA, entre outras.

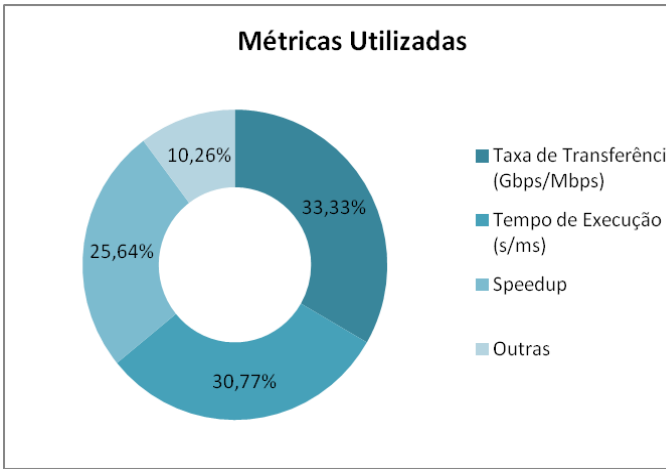


Fig. 3. Métricas utilizadas nos trabalhos filtrados para exibir os resultados da paralelização.

Q3) O autor faz a implementação da STT proposta por Aho-Corasick em sua forma original para representar a máquina AC?

A Questão 3 traz a resposta de uma das questões mais importantes desse mapeamento: 72,7% dos autores implementaram a STT na GPU e os outros 27,3% restantes não comentaram sobre como representaram a máquina AC na GPU. Com isso pode-se perceber que a forma mais comum de se executar a máquina de Aho-Corasick em uma GPU até a

data desse mapeamento é montando uma STT em alguma de suas memórias.

Q4) Em qual memória da GPU a máquina AC é colocada?

Essa questão, que é um complemento da Questão 3, mostra em qual memória a estrutura de dados utilizada para representar o DFA tem sido colocada.

Pode-se ver no gráfico da Figura 4 que a maioria dos autores implementaram a máquina na memória de textura da GPU. 11,11% implementaram na memória global e 5,56% implementaram na memória compartilhada. O restante não informou em qual memória ela foi colocada.

Como foi visto na Seção II, o problema de se implementar a STT está relacionado ao espaço de armazenamento. Esse fato pode explicar a opção dos autores em não escolher a memória compartilhada, que apesar de ser uma memória de acesso mais rápido, possui menos espaço de armazenamento em relação às memórias globais e de textura. Já o grande uso da memória de textura acredita-se que ocorra pelo aumento de desempenho que a cache de textura pode proporcionar em relação à memória global.

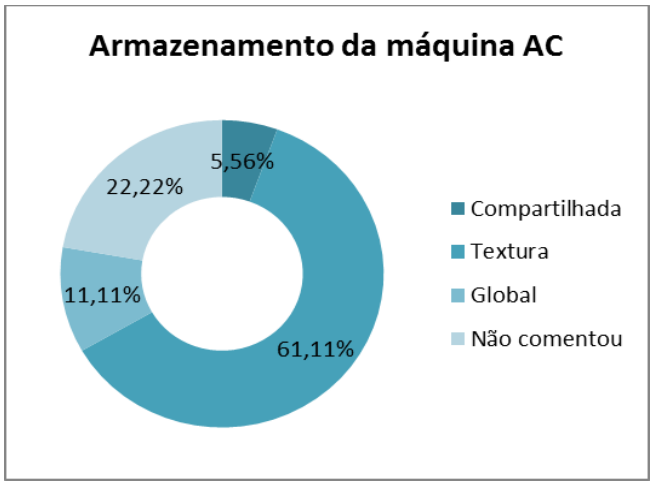


Fig. 4. Memória da GPU onde a máquina AC é colocada para ser executada.

Q5) Alguma técnica de compactação ou otimização é utilizada na máquina AC?

Para aumentar o desempenho do algoritmo, TRAN et al. (2012, 2013) fizeram com que a parte mais usada da STT ficasse na cache de textura. Já 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). LIN et al. (2013) implementam a STT na memória de textura, mas afirmam que como o estado inicial é o mais acessado, a primeira linha da STT (que representa o estado inicial) é colocada na memória compartilhada, já que o acesso a essa memória é mais rápido que à memória de textura.

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.

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) usa 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) usa o algoritmo de compressão chamado Lempel-Ziv-Welch.

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 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.

Q6) Em qual memória da GPU o texto de entrada é colocado?

Outra questão que deve ser levada em conta na hora de implementar e executar o algoritmo AC paralelo é onde colocar o texto a ser processado, ou seja, o texto onde serão buscados os padrões.

Nos artigos analisados, a maioria dos autores implementaram e executaram o algoritmo com o texto na memória global, alguns usaram a memória compartilhada para isso e o restante não citou a localização. Este resultado pode ser visto na Figura 5.

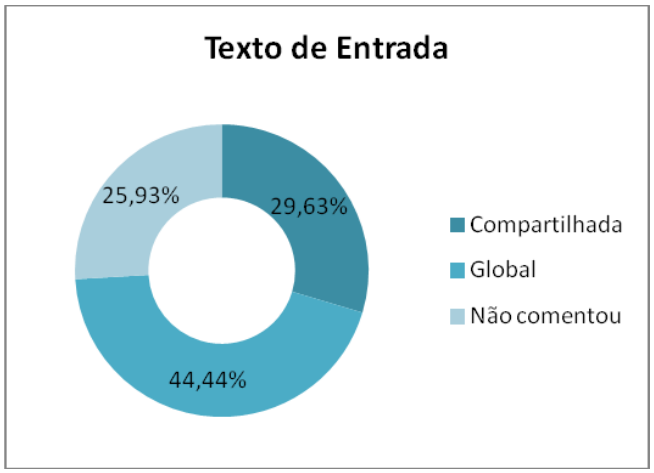


Fig. 5. Memória da GPU onde o texto de entrada é colocado para os padrões serem encontrados.

Acredita-se que a maioria dos autores prefere usar a memória global porque é uma memória com grande capacidade de armazenamento e como o tamanho do texto a ser processado normalmente é na casa dos GigaBytes, essa é a memória mais indicada. Por outro lado, autores como TRAN et al. (2012, 2013) ou ZHA e SAHNI (2011, 2013) usaram a memória compartilhada a fim de deixar o processamento mais rápido, já que o acesso a essa memória é mais rápido que o da global. Porém, para fazer isso, eles precisaram pré-processar os dados de entrada que o host envia para a GPU.

V. Conclusão

Este trabalho teve como objetivo mapear o estado da arte identificando e analisando as questões de implementação, execução e otimização do algoritmo Aho-Corasick no paradigma paralelo com uso de GPUs.

O método utilizado foi o mapeamento sistemático e foi conduzido por meio de um protocolo de busca e seleção de artigos que especificou o método utilizado nesse artigo. Com os termos de busca definidos, foram realizadas as buscas em inglês na base de dados do SCOPUS. Foram encontrados 147 artigos ao final das buscas. Foi realizada uma filtragem em cima desses 147 artigos através dos critérios de seleção que foram definidos no mapeamento, e ao final restaram 22 artigos para compor os estudos primários e serem analisados.

A partir dos estudos primários, foi possível responder às questões de pesquisa levantadas, e assim, pôde-se confirmar que tabelas de transição de estados são usadas frequentemente na memória de textura para executar a máquina AC. No momento da execução, a maioria dos autores armazenaram o texto a ser processado na memória global.

Notou-se também que diversas técnicas de otimização e compactação são adotadas com dois objetivos bem definidos: primeiro a execução mais rápida da máquina AC, ao transferir parte dela para a memória compartilhada, por exemplo, e, segundo, a redução do espaço de armazenamento, usando compactação dos nós e o bitmapeamento, por exemplo.

Com isso, acredita-se que esta pesquisa apresenta resultados relevantes à academia, fornecendo uma análise de como tem ocorrido a paralelização do algoritmo de comparação de *strings* Aho-Corasick, podendo ser utilizada como fonte de consulta para pesquisas futuras com esse tema.

A Tabela 1 da folha em anexo detalha a análise realizada nos artigos selecionados.

Referências

[1] AHO, A.; CORASICK, M. Efficient string matching: an aid to bibliographic search, Communications of the ACM, v.18 n.6, p.333-340, Jun. 1975.

[2] BELLEKENS, X. J. A.. et al. 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. Sep. 2014.

[3] CUDA: Programación Paralela Facilitada. Programación Paralela Facilitada. Disponível em: <http://www.nvidia.com.br/object/cuda_home_new_br.html>. Acesso em: 23 ago. 2016.

[4] HU, L. et al. An Efficient AC Algorithm with GPU. Elsevier: 2012 International Workshop on Information and Electronics Engineering (IWIEE). [s. L.], p. 4249-4253. Jan. 2012.

[5] JACOB, N.; BRODLEY, C. Offloading IDS Computation to the GPU. The 22nd Annual Computer Security Applications Conference, pp 371-380, Dec. 2006.

[6] JAISWAL, M. Accelerating Enhanced Boyer-Moore String Matching Algorithm on Multicore GPU for Network Security. International Journal of Computer Applications, Vol. 97 – No. 1, 2014. Acessado em: 20/06/2015. Disponível em: < http://research.ijcaonline.org/volume97/number1/pxc3896934.pdf >

[7] KOUZINOPOULOS, S. et al. A Hybrid Parallel Implementation of the Aho-Corasick and Wu-Manber Algorithms Using NVIDIA CUDA and MPI Evaluated on a Biological Sequence Database. International Journal On Artificial Intelligence Tools. [s. L.], p. 1-32. Feb. 2015.

[8] KOUZINOPOULOS, S.; MICHAILIDIS, D.; MARGARITIS, G. Multiple String Matching on a GPU Using CUDA. Scalable Computing: Practice And Experience: Scientific International Journal for Parallel and Distributed Computing. [s. L.], p. 121-137. Mar. 2015.

[9] LEE, C.; LIN, Y.; CHEN, Y. A Hybrid CPU/GPU Pattern-Matching Algorithm for Deep Packet Inspection. PLoS ONE 10(10): e0139301, Oct. 2015.

[10] LIN, C. et al. Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs. IEEE Transactions on Computers, Vol. 62, pp 1906-1916, Oct. 2013.

[11] LIN, C. et al. Accelerating String Matching Using Multi-threaded Algorithm on GPU. 2010 IEEE Global Telecommunications Conference (GLOBECOM 2010), pp 1-5, Dec. 2010.

[12] PENG, J.; CHEN, H.; SHI, S. The GPU-based string matching system in adavanced AC algorithm. 2010 10th IEEE International Conference on Computer and Information Technology (CIT 2010), pp 1158-1163, Jun. 2010.

[13] PETERSEN, K., FELDT, R., MUFTABA, S. AND MATTSON, M., 2008. Systematic mapping studies in software engineering. In: Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering, 68-77.

[14] PUNGILA, C. Hybrid Compression of the Aho-CorasickAutomaton for Static Analysis in IntrusionDetection Systems.Springer. [s. L.], p. 1-10. jan. 2013.

[15] PUNGILA, C.; NEGRU, V. Real-Time Hybrid Compression of Pattern Matching Automata for Heterogeneous Signature-Based Intrusion Detection. Springer Link. Berlim, p. 65-74. May. 2015.

[16] PUNGILA, C.; REJA, M.; NEGRU, V. Efficient parallel automata construction for hybrid resource-impelled data-matching. Future Generation Computer Systems, Vol. 36, pp 31-41, Jul. 2014.

[17] SANTOS, B. R. 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, Lavras, 2005. Disponível em: <http://www.ginux.ufla.br/files/mono-BrunoSantos.pdf>. Acesso em: 14 set. 2016.

[18] SNORT TEAM. SNORT Users Manual 2.9.7: The Snort Project. Oct. 2014. 265 p. Disponível em: <https://www.snort.org/#documents>. Acesso em: 07 Jun. 2015.

[19] SOROUSHNIA, S. et al. High Performance Pattern Matching on Heterogeneous Platform. Journal Of Integrative Bioinformatics. [s. L.], p. 1-11. Oct. 2014. Disponível em: <http://journal.imbio.de/articles/pdf/jib-253.pdf>. Acesso em: 19 set. 2016.

[20] TAKAHASHI, R.; INOUE, U. Parallel Text Matching Using GPGPU. 2012 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing (SNPD), pp 242-246, Aug. 2012.

[21] THAMBAWITA, D. R. V. L. B.; RAGEL, R.; ELKADUWE, D. 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, Dec. 2014.

[22] TRAN, N.; LEE, M. High performance string matching for security applications. 2013 International Conference on ICT for Smart Society (ICISS), pp 1-5, Jun. 2013.

[23] TRAN, N.; LEE, M.; CHOI, D. Cache Locality-Centric Parallel String Matching on Many-Core Accelerator Chips. Hindawi Publishing Corporation: Scientific Programming. [s. L.], p. 1-20. Sep. 2015.

Disponível em: <https://www.hindawi.com/journals/sp/2015/937694/>. Acesso em: 04 Set. 2016.

[24] TRAN, N.; LEE, M.; HONG, S.; BAE, J. Performance Optimization of Aho-Corasick Algorithm on a GPU. 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, pp 1143-1152, Jul. 2013.

[25] TRAN, N.; LEE, M.; HONG, S.; SHIN, M. Memory Efficient Pararellelization for Aho-Corasick Algorithm on a GPU. The 14th International Conference on High Performance Computing and Communications, pp 432-438, Jun. 2012.

[26] TRAN, N.; LEE, M; HONG, S.; CHOI, J. High throughput parallel implementation of aho-corasick algorithm on a GPU. 2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), pp 1807-1816, May. 2013.

[27] VILLA, O.; CHAVARRÍA-MIRANDA, D. G.; TUMEO, A. Aho-Corasick String Matching on Shared and Distributed-Memory Parallel Architectures. IEEE Transactions on Parallel and Distributed Systems, Vol. 23, pp 436-443, Mar. 2012.

[28] ZHA, X.; SAHNI, S. GPU-to-GPU and host-to-host multipattern string matching on a GPU. IEEE Transactions on Computers, Vol. 62, pp 1156-1169, Jun. 2013.

[29] ZHA, X.; SAHNI, S. Multipattern String Matching On A GPU. 2011 IEEE Symposium on Computers and Communications (ISCC), pp 277-282, Jun. 2011.

ANEXO

TABELA I
ARTIGOS SELECIONADOS PARA OS ESTUDOS PRIMÁRIOS

Artigo	Q1	Q2	Q3	Q4	Q5	Q6
[2] A Highly-Efficient Memory-Compression Scheme for GPU-Accelerated Intrusion Detection Systems	*2014*	*Tf/sp*	*s*	*-*	*Usa compresão de caminho e bitmapeamento*	*-*
[9] A hybrid CPU/GPU pattern-matching algorithm for deep packet inspection	*2015*	*Tf*	*s*	*t*	*-*	*g*
[7] A hybrid parallel implementation of the Aho-Corasick and Wu-Manber algorithms using NVIDIA CUDA and MPI evaluated on a biological sequence database	*2015*	*Te/sp*	*s*	*t*	*-*	*g*
[10] Accelerating pattern matching using a novel parallel algorithm on gpus	*2013*	*Tf*	*s*	*t*	*Primeira linha na compartilhada*	*g*
[11] Accelerating string matching using multi-threaded algorithm on GPU	*2010*	*Tf*	*s*	*c*	*É dividida em grupos*	*-*
[27] Aho-Corasick String Matching on Shared and Distributed-Memory Parallel Architectures	*2012*	*Tf*	*s*	*t*	*Insere flag na STT*	*g*
[23] Cache locality-centric parallel string matching on many-core accelerator chips	*2015*	*Tf/sp*	*s*	*t*	*STT divida em pequenas STTs*	
[16] Efficient parallel automata construction for hybrid resource-impelled data-matching	*2014*	*Te*	*-*	*-*	*-*	*-*
[28] GPU-to-GPU and host-to-host multipattern string matching on a GPU	*2013*	*Te/ sp*	*-*	*-*	*-*	*c*
[14] Hybrid compression of the Aho-Corasick automaton for static analysis in intrusion detection systems	*2013*	*Tf/mu*	*s*	*-*	*Usa compresão de caminho e bitmapeamento*	
[19] High performance pattern matching on heterogeneous platform	*2014*	*Tf*	*s*	*t*	*-*	*c*
[22] High performance string matching for security applications	*2013*	*Tf*	*s*	*g*	*-*	*g*
[26] High throughput parallel implementation of aho-corasick algorithm on a GPU	*2013*	*Te/tf/sp*	*s*	*t*	*Parte mais usada na cache de textura*	*g/c*
[25] Memory efficient parallelization for Aho-Corasick algorithm on a GPU	*2012*	*Te/tf/sp*	*s*	*t*	*Parte mais usada na cache de textura*	*g/c*
[29] Multipattern string matching on a GPU	*2011*	*Te/sp*	*-*	*-*	*-*	*c*
[8] Multiple string matching on a GPU using CUDA	*2015*	*Te/sp*	*s*	*g*	*-*	
[20] Parallel text matching using GPGPU	*2012*	*Te/tf/e*	*-*	*-*	*-*	*g*
[24] Performance optimization of aho-corasick algorithm on a GPU	*2013*	*Te/tf/sp*	*s*	*t*	*Parte mais usada na cache de textura*	*g/c*
[15] Real-time hybrid compression of pattern matching automata for heterogeneous signature-based intrusion detection	*2015*	*Tc*	*-*	*-*	*Usa Lempel-Ziv-Welch e bitmapeamento*	*-*
The GPU-based string matching system in adavanced AC algorithm	*2010*	*Te/sp*	*s*	*t*	*-*	*g/c*
[21] To use or not to use Graphics processing units (GPUs) for pattern matching algorithms	*2014*	*Te*	*s*	*-*	*-*	*-*

Onde: "/" significa que teve mais de uma abordagem; "tf" significa taxa de transferência; "te" significa tempo de execução; "sp" significa speedup; "tc" significa taxa de compresão; "e" significa consumo de energia; "mu" significa memória utilizada; "s" significa sim; "t" significa memória de textura; "c" significa memória compartilhada; "g" significa memória global.