UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Paralelização em CUDA do Algoritmo Aho-Corasick Utilizando as Hierarquias de Memórias da GPU e Nova Compactação da Tabela de Transição de Estados

José Bonifácio da Silva Júnior

SÃO CRISTÓVÃO/ SE

2017

UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

José Bonifácio da Silva Júnior

Paralelização em CUDA do Algoritmo Aho-Corasick Utilizando as Hierarquias de Memórias da GPU e Nova Compactação da Tabela de Transição de Estados

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PROCC) da Universidade Federal de Sergipe (UFS) como parte de requisito para obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Edward David Moreno Ordonez

SÃO CRISTÓVÃO/ SE

2017

José Bonifácio da Silva Júnior

Paralelização em CUDA do Algoritmo Aho-Corasick Utilizando as Hierarquias de Memórias da GPU e Nova Compactação da Tabela de Transição de Estados

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PROCC) da Universidade Federal de Sergipe (UFS) como parte de requisito para obtenção do título de Mestre em Ciência da Computação.

BANCA EXAMINADORA

Prof. Dr. Edward David Moreno Ordonez, Orientador
Universidade Federal de Sergipe (UFS)

Prof. Dr. Ricardo José Paiva de Britto Salgueiro
Universidade Federal de Sergipe (UFS)

Prof. Dr. Ricardo dos Santos Ferreira
Universidade Federal de Viçosa (UFV)

Paralelização em CUDA do Algoritmo Aho-Corasick Utilizando as Hierarquias de Memórias da GPU e Nova Compactação da Tabela de Transição de Estados

Este exemplar corresponde à Dissertação de Mestrado, sendo o Exame de Defesa do mestrando **JOSÉ BONIFÁCIO DA SILVA JÚNIOR** para ser aprovada pela Banca Examinadora.

São Cristóvão – SE, 21 de Junho de 2017

_____

Prof. Dr. Edward David Moreno Ordonez,
Orientador, UFS

_____

Prof. Dr. Ricardo José Paiva de Britto Salgueiro,
Examinador, UFS

_____

Prof. Dr. Ricardo dos Santos Ferreira,
Examinador, UFV

Júnior, José Bonifácio da Silva.

Paralelização em CUDA do Algoritmo Aho-Corasick Utilizando as Hierarquias de Memórias da GPU e Nova Compactação da Tabela de Transição de Estados / José Bonifácio da Silva Júnior; orientador Edward David Moreno Ordonez. – São Cristovão, 2017.
71f. ;

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Sergipe, 2017.

1. Computação de alto desempenho. 2. Arquitetura de Computadores. 3. Segurança da Informação. I. Ordonez, Edward David Moreno, orientador. II. Paralelização em CUDA do Algoritmo Aho-Corasick Utilizando as Hierarquias de Memórias da GPU e Nova Compactação da Tabela de Transição de Estados.

AGRADECIMENTOS

Agradeço primeiramente a Deus, pois Ele mais uma vez me deu a força necessária para vencer mais uma grande batalha da minha vida. Agradeço aos meus pais, Maria de Fátima Costa Silva e José Bonifácio da Silva, por me darem todo carinho e suporte necessários e por muitas vezes me compreederem com a paciência que apenas os pais e as pessoas que nos amam têm.

Agradeço ao professor Dr. Ricardo Salgueiro por fazer parte da minha banca de defesa do mestrado, mas, acima disso, pelo grande professor e ser humano que é (acho que é a pessoa mais feliz que conheço, nunca o vi de mau humor). Obrigado por ter me passado de forma clara e de fácil compreensão conhecimentos ainda na graduação em uma das áreas que mais gosto que é a área de segurança de redes. Isso me influenciou fortemente a dar continuidade nos estudos da área. Além disso, viabilizou a parceria com o Laboratório de Computação de Alto Desempenho – LCAD – da UFS, que possibilitou que eu realizasse os experimentos desta dissertação.

Agradeço ao professor Dr. Ricardo Ferreira, membro externo da banca, mas que (apesar de externo) me auxiliou “de perto” neste trabalho, sempre sendo solicito demais, respondendo às várias dúvidas que surgiram durante o trabalho em uma velocidade que mostrava o interesse dele em ajudar e passar o conhecimento sobre as GPUs. Foi essencial para o meu aprendizado na área de paralelização de algoritmos com GPUs ao ministrar o curso presencial da NVIDIA e a partir daí fazer surgir boas idéias para serem apicadas neste trabalho. Além disso, também viabilizou a parceria com a Universidade Federal de Viçosa (UFV), ao possibilitar que eu realizasse os experimentos em uma das GPUs dessa universidade.

Agradeço de forma especial ao meu orientador professor Dr. Edward Moreno por ter me acompanhado de perto neste desafio e pela competência de ter me mostrado os melhores caminhos para que eu concluísse com êxito este trabalho. Obrigado pela parceria e confiança, pela busca incansável das melhores soluções e também por ter sugerido que eu fizesse o mestrado ainda quando eu estava na graduação.

Agradeço aos meus irmãos Thiago, Danilo e Fábio que me ajudaram bastante nesta caminhada, meus familiares e minha namorada Ana Cristina, a qual esteve comigo não só nos bons momentos, mas também nos momentos que mais precisei de forças para continuar. Agradeço aos meus colegas e já amigos da equipe de TI da Corregedoria do Tribunal de Justiça de Sergipe pelos incentivos dados para que eu concluísse este trabalho e não desistisse em momento algum. Agradeço a todos os outros amigos, os que atualmente estão mais próximos como Marcelo, Marco e Nelmo e todos os outros que devido aos caminhos que a vida reserva para cada um de nós estão mais distantes fisicamente, mas que fazem parte da minha história.

Por fim agradeço a todos os professores da Universidade Federal de Sergipe, em especial aos professores do departamento de Ciência da Computação, por me concederem os conhecimentos que complementaram a minha formação de Mestre em Ciência da Computação.

Lista de Figuras

Lista de Tabelas

Sumário

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. Esta dissertação tem como objetivo fazer a paralelização dos algoritmos de comparação de *strings* Força-Bruta e Aho-Corasick e propor uma nova compactação da Tabela de Transição de Estados do algoritmo Aho-Corasick a fim de possibilitar o uso dela na memória compartilhada e acelerar a comparação de *strings*. Os dois algoritmos foram paralelizados utilizando a plataforma CUDA da NVIDIA e executados nas memórias da GPU a fim de possibilitar uma análise comparativa de desempenho dessas memórias. Inicialmente, o algoritmo AC mostrou-se mais veloz do que o algoritmo Força-Bruta e por isso seguiu-se para sua otimização. O algoritmo AC foi compactado e executado de forma paralela na memória compartilhada, alcançando um ganho de desempenho de 15% em relação às outras memórias da GPU e sendo 48 vezes mais rápido que sua versão na CPU quando os testes foram feitos com pacotes de redes reais. Já quando os testes foram feitos com dados sintéticos (dados menos aleatórios) o ganho chegou a 73% e o algoritmo paralelo 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.

Palavras-chaves: GPUS, CUDA, Algoritmos de Comparação de *Strings*, Aho-Corasick, IDS, Hierárquia de Memória da GPU, Técnicas de Compactação.

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 This dissertation aims to parallelize the Brute Force and Aho-Corasick string matching algorithms and to propose a new compression of the State Transition Table of the Aho-Corasick algorithm in order to make it possible to use it in shared memory and accelerate the comparison of strings. The two algorithms were parallelized using the NVIDIA CUDA platform and executed in the GPU memories to allow a comparative analysis of the performance of these memories. Initially, the AC algorithm proved to be faster than the Brute Force algorithm and so it was followed for optimization. The AC algorithm was compressed and executed in parallel in shared memory, achieving a performance gain of 15% over other GPU memories and being 48 times faster than its serial version when testing with real network packets. When the tests were done with synthetic data (less random data) the gain reached 73% and the parallel algorithm was 56 times faster than its serial version. Thus, it can be seen that the use of compression in shared memory becomes a suitable solution to accelerate the processing of IDSs that need agility in the search for patterns.*

Keywords: *GPUS, CUDA, string matching algorithms, Aho-Corasick, IDS, GPU Memory Hierarchy, Compaction Techniques.*

1. Introdução

A tecnologia da informação (TI) tem sido cada vez mais determinante para o sucesso de empresas e organizações ao redor do mundo e possui 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 função específica. Uma dessas ferramentas é o IDS (*Intrusion Detection System*), que tem a finalidade de detectar uma série de ataques na rede.

O IDS é uma ferramenta que analisa as atividades do sistema e da rede para verificar possíveis ataques, se possível em tempo real. Ele faz a coleta de informações sobre os pacotes que circulam na rede e verifica a existência de ataques conhecidos 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 caraterísticas do ataque, alterar ou remodelar sua configuração de acordo com as necessidades e alertar ao administrador do ambiente sobre esse ataque (SANTOS, 2005). O objetivo dele é monitorar 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 de ataques pré-configuradas. Em outras palavras, o Snort precisa manipular computacionalmente intensas operações de comparação de *strings*, onde uma grande quantidade de dados precisa ser comparada com suas assinaturas (OKE e VAIDYA, 2015). Esta atividade é bastante custosa para a CPU e, por isso, várias pesquisas de como acelerá-la estão sendo feitas e uma solução que tem ganhado força é a paralelização dos algoritmos que fazem a comparação de *strings* (ou comparação de padrões).

1.1. Motivação

Os IDSs, em geral, possuem duas limitações críticas que fazem diminuir sua velocidade de processamento. A primeira é a limitação de entrada e saída causada pelo *overhead* de leitura de pacotes na interface de rede (JACOB e BRODLEY, 2006). A segunda limitação fica por conta das operações de comparação do *payload* do pacote com a coleção de assinaturas, as quais consomem cerca de 70% a 80% do tempo de processamento da CPU

(JAISWAL, 2014). Esses dois problemas poderão causar uma perda considerável de pacotes ou até mesmo derrubar o *host* onde ele está instalado, ferindo dois príncipios básicos da Segurança da Informação que são a integridade e disponibilidade. Perder dados ou tornar o *host* responsável por manter a segurança de uma rede indisponível pode ser intolerável a depender da situação.

Esse cenário tende a ficar cada vez pior uma vez que novos ataques vêm sugindo, necessitando de novos padrões e, consequentemente, aumentando a quantidade de dados para serem comparados. Além disso, as redes estão atingindo níveis de velocidades cada vez maiores, necessitando de algoritmos mais velozes. Os algoritmos tradicionais não conseguem manipular essa grande quantidade de dados em uma velocidade aceitável, chegando ao ponto de tornarem-se inadequados para essa tarefa (OKE e VAIDYA, 2015). E como já foi mencionado anteriormente, a comparação consome cerca de 70% a 80% de todo processamento do Snort, sendo considerada a operação mais crítica do mesmo.

A limitação de *overhead* não parece ser um bom problema para tentar uma aceleração inicialmente, já que chegariam mais pacotes para o Snort processar e criaria-se um gargalo computacional devido à limitação da comparação de *strings*. Já a aceleração desta última mostra-se mais capaz de aumentar o desempenho do Snort, já que não há identificação de outra atividade lenta que utilize os resultados da comparação de *strings*.

O uso das Unidades de Processamento Gráfico (GPUs) é vista como uma solução adequada para acelerar o processamento da comparação de *strings* do Snort, uma vez que as GPUs têm um maior poder de computação do que as CPUs, como pode ser visto em alguns trabalhos como o de LIN, C. et al. (2013) que 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 e no trabalho de TRAN, N. et al. (2012) que 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. Por fim, Kouzinopoulos e Margaritis (2008) paralelizaram vários algoritmos de comparação de *strings* obtendo uma melhora de 20 vezes do algoritmo Boyer-Moore-Horspool e, posteriormente, ao usar a memória compartilhada da GPU, aumentou mais 24 vezes essa velocidade.

Outros autores também paralelizaram esses algoritmos, porém usando outras tecnologias, como Jiang, Salamatian e Mathy (2013) que exploraram o grande poder

computacional do processador de múltiplo núcleo TILERAGX36 PCIe, um processador com 36 núcleos desenvolvido pela Tilera Inc e instalado em um servidor x86 com sistema operacional Linux, tendo destacado como melhor resultado o fato do protótipo completo ter exibido velocidade quase linear e pôde suportar até 7,2 Gbps de tráfego com pacotes de 100 bytes. Huang, N. et al. (2008) desenvolveram o PixelSnort, que faz a comparação usando o algoritmo Knuth-Morris-Pratt paralelizado em uma GPU e sendo executado através de chamadas OpenGl no código fonte do Snort. Neste trabalho, o pixelSnort superou o Snort convencional em 40%.

Como se observa há uma tendência em paralelizar algoritmos de comparação de *strings* tradicionais, principalmente quando a tarefa deles é buscar padrões em grandes quantidades de dados, como ocorre no Snort.

1.2. Objetivos da Dissertação

O objetivo principal desta dissertação de mestrado é paralelizar algoritmos de comparação de *strings* tradicionais através de GPUs usando a linguagem de programação C na plataforma de computação paralela CUDA. Após paralelizar os algoritmos, acelerar o tempo de execução deles fazendo o uso da memória compartilhada das GPUs através de uma nova técnica de compactação.

1.2.1. Objetivos Específicos

Para alcançar o objetivo primário, alguns objetivos específicos têm que ser atendidos, a saber:

- Fazer um estudo na literatura sobre a paralelização do algoritmo de comparação de *strings* do Snort, o Aho-Corasick;
- Compactar a máquina de estados do Aho-Corasick a fim de possibilitar que o algoritmo seja executado na memória compartilhada da GPU;
- Paralelizar o algoritmo em uma GPU fazendo versões nas memórias global, de textura e compartilhada;
- Comparar o desempenho dos algoritmos paralelizados, além de compará-los com a versão serial do algoritmo;

1.3. Organização do Trabalho

A seção 2 apresenta uma breve revisão bibliográfica fazendo uma abordagem sobre o IDS Snort na primeira subseção. A segunda subseção detalha as características de uma GPU genérica, dispositivo necessário para a parelização dos algoritmos. A terceira subseção traz

alguns algoritmos de comparação de *strings* que foram paralelizados e poderão ser utilizados no Snort em substituição aos algoritmos seriais que nele estão. Por fim, a quarta subseção mostra o Estado da Arte do tema proposto, onde é feito um resumo de artigos publicados em renomadas revistas na área de computação sobre o Snort e a paralelização de algoritmos de comparação de *strings*. A Seção 3 explica a Metodologia adotada para atingir os objetivos propostos. A Seção 4 traz a descrição completa dos algoritmos desenvolvidos, os resultados iniciais dos algoritmos Naive e Aho-Corasick executados na placa GT210 e a nova técnica de compactação. A Seção 5 tem foco na melhoria do algoritmo Aho-Corasick e expõe os resultados obtidos com as placas TESLA K20 e TITANX, além da CPU. Por último, é feita a conclusão do trabalho na Seção 6.

2. Snort, GPUs e Algoritmos de Comparação de Strings

As seções seguintes têm por finalidade apresentar conceitos sobre o Snort, as GPUs, os algoritmos de comparação de *strings* Força-bruta e o Aho-Corasick e mostrar o Estado da Arte sobre a paralelização de algoritmos de comparação de *Strings*.

2.1. Snort

Uma ferramenta IDS serve basicamente para trazer informações sobre a rede, como: a quantidade de tentativas de ataques por dia, qual tipo de ataque foi usado, qual a origem dos ataques. A partir delas vai se tomar conhecimento do que se passa na rede e, em casos extremos, podem ser tomadas medidas cabíveis para tentar solucionar qualquer problema.

Nessas ferramentas existem dois tipos de limitações: 1) limitações de CPU que surgem devido à comparação de *strings* e 2) as limitações de Entrada e Saída causadas pelo *overhead* de leitura de pacotes na interface de rede (Jacob e Brodley, 2006). Para resolver o primeiro problema e amenizar o segundo, as pesquisas atuais vêm tentando fazer melhorias nos motores de comparação de *string* dos IDSs.

O Snort é um IDS que possui tecnologia de código aberto (Santos, 2005) para a detecção e prevenção de intrusão. Este IDS utiliza uma linguagem dirigida por regras, que combina os benefícios da assinatura de protocolo e os métodos de inspeção baseados em anomalia. Podemos ver na Figura 2.1 uma das várias regras que temos disponíveis no site do Snort. Basicamente a regra faz um alerta quando o protocolo tcp é usado e algum pacote é enviado de um dispositivo da rede externa por qualquer porta para algum dispositivo da rede interna pela porta 7597. Além disso, o alerta só será lançado se todas as regras que estão entre parêntesis forem atendidas. Por exemplo, o pacote enviado deve possuir a *string* “qazwsx.hsq” como indica a palavra-chave *content*.

```
1 alert tcp $EXTERNAL_NET any -> $HOME_NET 7597 (msg:"MALWARE-BACKDOOR QAZ Worm Client Login access";
2 flow:to_server,established; content:"qazwsx.hsq"; metadata:ruleset community; reference:mcafee,98775;
3                             classtype:misc-activity; sid:108; rev:11;)
```

Figura 2.1: Exemplo de regra disponível no site do Snort.

Com o intuito de ajudar na paralelização deste IDS, (Junta e Pereira, 2015) separaram e analisaram os componentes do Snort, os quais foram classificados em Mecanismo de captura, pré-processadores, mecanismo de detecção e plug-ins de saída, como pode ser visto no fluxograma de funcionamento do Snort da Figura 2.2.

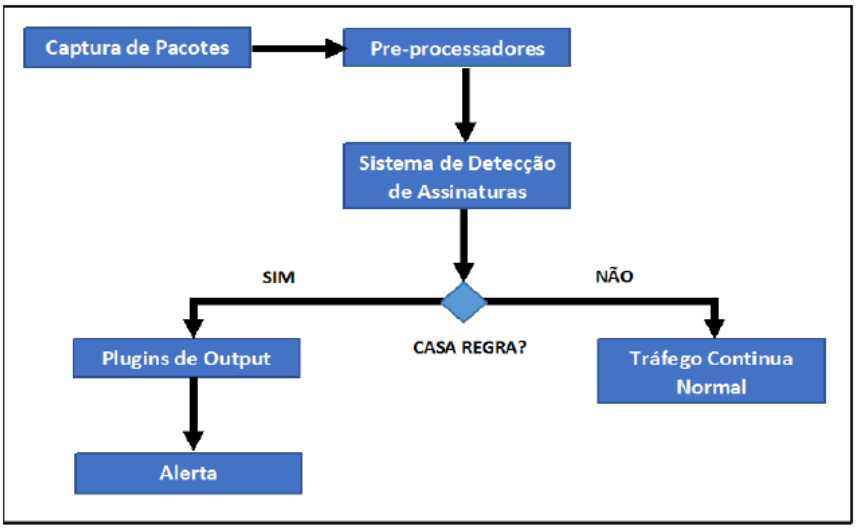


Figura 2.2: Fluxograma de execução do Snort (Junta e Pereira, 2015).

O mecanismo de captura é a parte que monitora, em modo promíscuo, todo o tráfego do segmento onde ele está inserido, através das bibliotecas de captura de pacotes WinPcap no Windows e LibPcap no Linux. Os pré-processadores definem o tipo do pacote e como ele será tratado, através da ativação de *plugins* específicos para cada tipo de pacote, por exemplo: um pacote Telnet é tratado diferentemente de um pacote RPC neste componente. O Mecanismo de Detecção é o principal componente e onde ocorre a comparação de *strings*. Já os *Plugins* de Saída ficam responsáveis em decidir o que fazer caso a *string* seja encontrada.

Inicialmente eles verificaram que o componente que mais consome CPU é o mecanismo de detecção com 76% do tempo de execução do Snort. Sendo assim, eles dividiram este componente de forma não aleatória em trechos de códigos e realizaram a medição do tempo de execução de cada trecho. Após obter os resultados, os trechos que tiveram tempos mais relevantes foram divididos novamente, realizando um aprofundamento no trecho. Assim por diante até que se obtivesse um trecho relativamente pequeno e que demandasse um tempo de execução relativamente grande. Deste modo, sugeriram começar a paralelização pelo trecho menor e com tempo de execução relevante, e posteriormente estender a paralelização para o trecho maior que contém o trecho já paralelizado. O resultado dessa metodologia adotada por eles pode ser visto no gráfico da Figura 2.3.

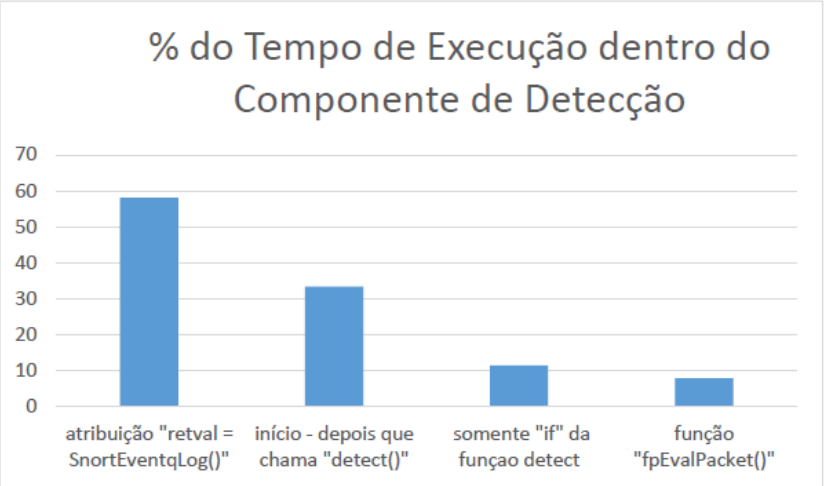


Figura 2.3: Percentual do tempo de execução dos trechos do componente de detecção (Junta e Pereira, 2015).

Com isso, eles sugerem que se inicie uma paralelização pela função "fpEvalPacket()", que está contida dentro da função "Detect()" e que demanda aproximadamente 77% do tempo de execução de "Detect()" e 7% do tempo de execução do componente de detecção. Após isso, paralelizar "Detect()" que está contido dentro do trecho "inicio do componente até depois que chama Detect()" e que demanda 33% da execução do trecho "inicio do componente até depois que chama Detect()" e 11% do tempo de execução do componente detecção. Para depois paralelizar este último trecho todo que demanda aproximadamente 33% do tempo de execução do componente de detecção.

2.2. Unidade de Processamento Gráfico (GPU)

As Unidades de Processamento Gráfico (GPUs) são dispositivos processadores de elementos gráficos introduzidos na década de 1980 para descarregar os processamentos gráficos relacionados às Unidades Centrais de Processamento (CPUs) (KOUZINOPOULOS e MARGARITIS, 2008). As GPUs modernas são programáveis e possuem processadores de *streams* capazes de fazerem computação de alto desempenho, a qual tem evoluído consideravelmente depois do surgimento da arquitetura heterogênea CPU-GPU.

A computação heterogênea utiliza um conjunto de processadores com arquiteturas diferentes para executar uma aplicação (CHENG, et al., 2014). A Figura 2.4 ilustra esse cenário.

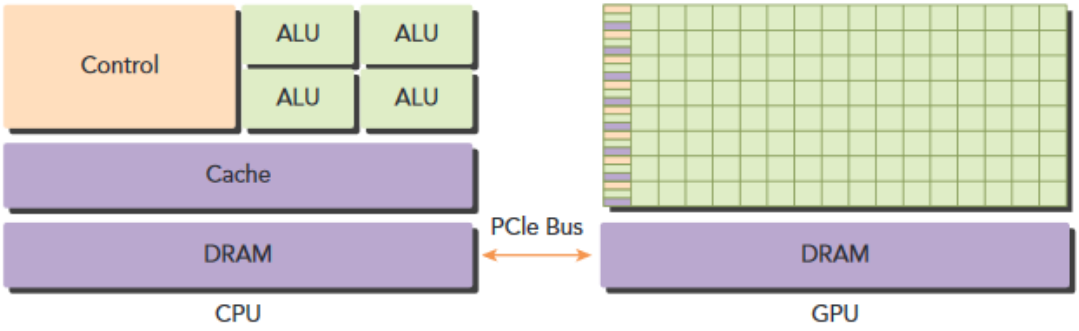


Figura 2.4: Ilustração de uma arquitetura heterogênea com CPU e GPU (CHENG, et al., 2014).

A CPU, também chamada de *core*, é a componente chave na computação de alto desempenho. Se há tempos atrás se colocava apenas um *core* em um chip de processamento formando as CPUs tradicionais, nos dias atuais tem-se colocados vários *cores* em um único chip, formando-se as CPUs *multicore* e as GPUs.

Diferentemente da arquitetura serial, que é classificada como Single Instruction Single Data (SISD), a GPU utiliza uma arquitetura Single Instruction Multiple Data (SIMD), onde uma única instrução é replicada e executada em vários *cores* da GPU de forma simultânea, e cada *core* opera sobre uma *stream* de dado diferente. Tenta-se com isso diminuir a latência (tempo necessário para uma operação começar e terminar – normalmente dada em milissegundos), e aumentar a largura de banda (quantidade de dados que pode ser processada por unidade de tempo – normalmente dada em bytes por segundo) e vazão (quantidade de operações que pode ser processada por unidade de tempo – normalmente dada em operações de ponto flutuante por segundo).

No trabalho com computação paralela, deve-se conhecer de forma sólida tanto os aspectos do hardware do dispositivo (arquitetura) como também seus aspectos de software (programação paralela).

A NVIDIA fornece algumas famílias de produtos, como a Tegra que foi projetada para dispositivos móveis e embarcados, Geforce para consumidor gráfico comum, Quadro para visualização profissional, e Tesla para computação paralela em Datacenter. Oferece a Fermi, o acelerador de GPU na família de produtos Tesla, que mostrou ser capaz de acelerar aplicações em muitas áreas como processamento sísmico, simulações de bioquímica, modelagem climática, processamento de sinais, entre outras. Após a Fermi lançou a Kepler que possui um poder de processamento muito maior que as gerações anteriores (CHENG, et al., 2014). Mais recentemente a Nvidia lançou as arquiteturas Maxuell e Pascal possuindo ainda mais poder de processamento.

	FERMI (TESLA C2050)	KEPLER (TESLA K10)
CUDA Cores	448	2 x 1536
Memory	6 GB	8 GB
Peak Performance*	1.03 Tflops	4.58 Tflops
Memory Bandwidth	144 GB/s	320 GB/s

Figura 2.5: Comparação entra as arquiteturas Fermi e Kepler (CHENG, et al., 2014).

Para dar suporte à execução de uma aplicação CPU-GPU, a Nvidia projetou CUDA, um modelo de programação para computação heterogênea que permite acessar tanto a GPU como a CPU através de linguagem de programação. Seu compilador, no caso do CUDA C, o nvcc, é quem vai dividir o código que executará no host (CPU) e o que executará no device (GPU).

A arquitetura da GPU é construída em torno de uma matriz escalável de Multiprocessadores de Streams (SM). Alguns componentes que fazem parte desta arquitetura são:

- CUDA Cores;
- Memória Compartilhada/ Cache L1;
- Registradores;
- Unidades de Carga/Armazenamento;
- Unidades de Função Especial;
- Agendador de Warp.

Cada SM em uma GPU é projetada para dar suporte à execução concorrente de milhares de *threads*. Quando um grid é lançado, os blocos de *threads* daquele grid são distribuídos entre os SMs disponíveis para execução. Após serem agendados em um SM, os *threads* de um bloco executam simultaneamente apenas na SM atribuída. Múltiplos grupos de *threads* podem ser atribuídos ao mesmo SM de uma vez e são agendados com base na disponibilidade de recursos SM. As instruções dentro de um único *thread* utilizam pipeline para alavancar o paralelismo no nível de instrução, somando-se ao já conhecido paralelismo em nível de dados (CHENG, et al., 2014).

CUDA utiliza uma arquitetura para gerenciar e executar *threads* em grupos de 32 chamados *warps*. Todos os *threads* em um mesmo *warp* executam a mesma instrução ao mesmo tempo, ou seja, em uma estrutura condicional, se a *thread* 1 estiver no *if* e todas as outras 31 *threads* já estiverem no *else*, as 31 *threads* esperarão o fim da execução do *if* da *thread* 1 para que todas as 32 *threads* executem o *else*. Isso significa que quanto mais existirem estruturas que mudem o fluxo normal do programa, a tendência é deixar o algoritmo menos eficiente.

Um bloco de *thread* é agendado em apenas um SM. Uma vez que ocorre o agendamento, ele permanece até a execução se completar. A Figura 2.6 mostra a correspondência dos componentes na visão lógica e na visão de hardware da programação em CUDA.

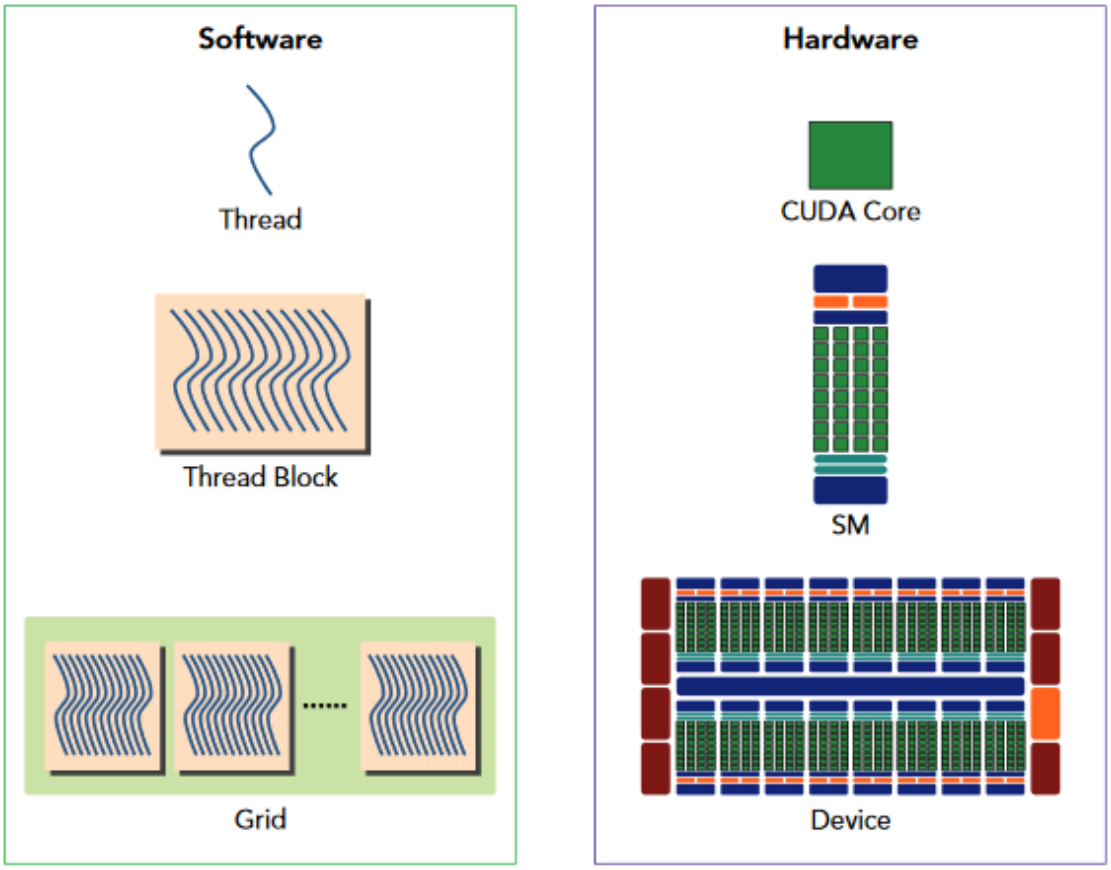


Figura 2.6: Visão lógica x visão de hardware da programação em CUDA.

A Figura 2.6 mostra que a programação de uma *thread* significa o uso de um determinado *core* da GPU. O bloco de *threads* significa o conjunto de *cores* de um dos SMs disponíveis. E o grid corresponde ao próprio SM.

Já a Figura 2.7 dá uma visão detalhada da arquitetura de uma GPU (aspecto físico), voltada para a hierarquia de memórias da mesma.

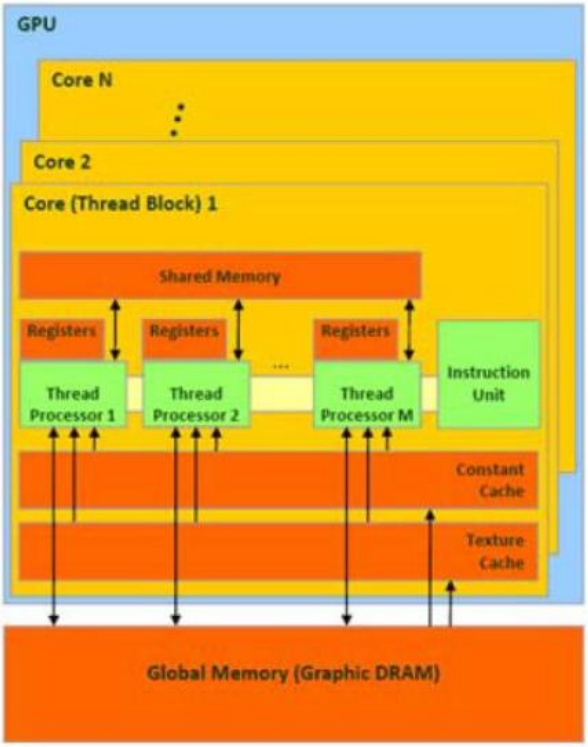


Figura 2.7: Arquitetura geral de uma GPU (Tran, et al., 2012).

- A memória global é uma área no *off-chip* DRAM gráfico DDR (G-DRAM ou comumente chamado como a memória do dispositivo). Através da memória global a GPU pode se comunicar com a CPU do *host*. Todos os dados do aplicativo residem inicialmente na DRAM e os pedidos a essa memória normalmente são atendidos usando transações de memória de 128 bytes ou 32 bytes, ou seja, a GPU lerá pacotes de 128 ou 32 bytes por vez.
- A memória compartilhada usa o mesmo epaço de 64 KB da cache L1, porém pode ser programada. Reside dentro de cada bloco de *thread* e é compartilhada entre os *threads* em execução por múltiplos processadores de *thread*. O tempo de acesso a essa memória é mais rápido se comparado com o da memória global.
- Os registradores são as memórias mais rápidas da GPU e são usados para armazenar temporariamente os dados usados nos cálculos da GPU em cada processador de *thread*, similar aos registradores da CPU. Este é um recurso escasso, tendo um limite de hardware de 63 registradores por *threads* na arquitetura Fermi e 255 na Kepler, por exemplo. É importante observar que arrays que não podem ter o índice determinado em tempo de compilação ou que são muito grandes para os registradores são colocados na memória local (uma memória residente no mesmo espaço da memória global).
- Cada *thread* tem sua própria memória local para carregar e armazenar os dados

necessários para os cálculos. A memória local é também uma área na G-DRAM. Assim, é também uma memória lenta.

- Além das memórias mencionadas anteriormente, há a memória constante e a memória de textura no G-DRAM. Os dados na memória constante e memória de textura podem ser armazenados em cache como dados somente leitura no chip do cache constante e do cache de textura, respectivamente. Vale ressaltar que a memória de textura é otimizada para localização espacial 2D, assim os *threads* que usam a memória de textura para acessar dados 2D alcançarão um melhor desempenho.
- Além das caches de textura e local, podem existir outras memórias caches não programáveis que ajudam no aumento do desempenho de memórias como a global e compartilhada.

Existem padrões de acessos a essas memórias que podem influênciar no desempenho da aplicação. O padrão de acesso alinhado ocorre quando o primeiro endereço de uma transação de memória do dispositivo é um múltiplo da granularidade da cache usada para atender a transação (32 bytes para cache L2 ou 128 bytes para cache L1). A realização de uma carga desalinhada causará perda de largura de banda. Já o padrão de acesso *coalesced* ocorre quando todos os 32 *threads* em um warp acessam um pedaço contíguo de memória.

O poder computacional da GPU é maior que o da CPU. Seu paralelismo em nível de dados além de um modelo de memória simplificado oferecidos por esses dispositivos tem atraído muitos pesquisadores para resolver problemas de desempenho em algoritmos convencionais (ou seja, não gráficos) que possuem desempenho mais baixo por causa da sua execução sequencial. Porém, existem desafios a se enfrentar quando uma aplicação convencional é transferida para a GPU. Em particular, o modelo de memória é restritivo quando se trata de implementar estruturas de dados bem conhecidas, tais como listas ligadas e árvores (Jacob e Brodley, 2006).

2.3. Algoritmos de Comparação de Strings

Os algoritmos podem ser classificados em uma faixa que vai desde facilmente paralelizáveis até completamente não paralelizáveis (ou programas inerentemente seriais). Essa classificação se aproximará de facilmente paralelizável a depender de alguns fatores, como a facilidade de dividir o problema em partes e a indepêndencia entre os resultados de cada parte. Já os algoritmos inerentemente seriais exigem os resultados de um passo anterior para realizer de forma eficaz a próxima etapa.

A multiplicação de matrizes é um bom exemplo de algoritmo facilmente paralelizável

uma vez que é possível calcular separadamente cada posição da matriz resultante e o resultado de uma posição não interfere no resultado das outras. No caso da comparação de *strings*, nosso objeto de estudo, existem vários algoritmos e devemos analisar a particularidade de cada um deles.

2.3.1. Naive (Força-Bruta)

O método Naive, mais conhecido como Força-Bruta, alinha a extremidade esquerda do padrão P com a extremidade esquerda do texto T e então compara os caracteres de P e T da esquerda para a direita até que qualquer um dos dois caracteres comparáveis sejam diferentes ou até P se esgotar, caso em que uma ocorrência de P é relatada. No outro caso, P é deslocado uma posição para a direita, e as comparações são reiniciadas a partir da extremidade esquerda de P. Esse processo é repetido até a extremidade direita de P passar sobre a extremidade direita de T (DAN, 1997).

No pior caso, o número de comparações feitas por esse método é Θ(nm), onde n é o tamanho de P e m o tamanho de T. Esse número de comparações pode se tornar inviável à medida que os tamanhos de P e T forem aumentando (DAN, 1997).

Existem técnicas para melhorar o método Naive, fazendo-o passar de Θ(nm) para Θ(n + m) no pior caso. Uma dessas ideias é tentar deslocar o padrão P em mais de uma posição quando não há a correspondência dos caracteres de P e T. A Figura 2.8 mostra essas ideias, usando P = abxyabxz e T = xabxyabxyabxz. Nesta figura, o asterisco indica que não houve correspondência, enquanto que o circunflexo indica que houve.

No primeiro cenário da Figura 3 (cenário à esquerda), onde é ilustrada a comparação do Naive puro, o algoritmo imediatamente encontra uma diferença entre P e T, devido à primeira comparação dos caracteres deles, e desloca P em uma posição. Os próximos 7 caracteres correspondem e as comparações são bem-sucedidas. Porém na nona comparação não há correspondência, sendo P deslocado uma posição e as comparações reiniciadas a partir da extremidade esquerda de P. Apenas a partir da sexta posição de T será detectada uma ocorrência de P. Para este exemplo são necessárias 20 comparações através do Naive Puro.

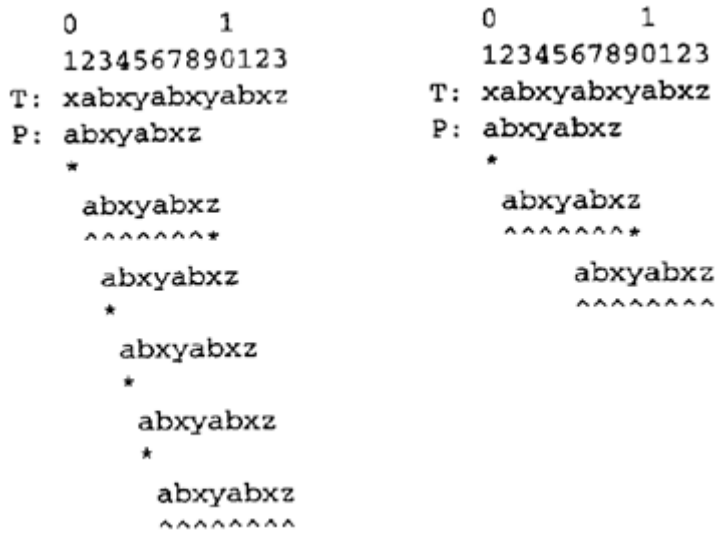


Figura 2.8: Ilustração do Naive Puro (à esquerda) e do Naive Melhorado (à direita) (DAN, 1997).

Um algoritmo mais inteligente pode perceber, depois da nona comparação, que as próximas três comparações não corresponderão. Este algoritmo, representado no segundo cenário da Figura 2.8 (cenário à direita), salta as três próximas comparações, e imediatamente move a extremidade esquerda de P para alinhar com a posição 6 de T. Para fazer isso, este algoritmo (chamado de Naive Melhorado) sabe que o primeiro caractere de P (especificamente 'a') não ocorre novamente em P até a posição 5 de P. Com isso ele conclui que o caractere 'a' não ocorre mais uma vez em T até a posição 6 de T, pois os três caracteres corresponderam e eram diferentes de 'a'.

2.3.2. Aho-Corasick

O Snort v2.9.7.3 usa o algoritmo de Aho-Corasick (AC) para comparar o conteúdo dos pacotes que chegam na interface de rede com o conjunto de assinaturas de ataques configurado. Esse algoritmo de comparação de *string* tradicional é capaz de comparar múltiplos padrões simultaneamente percorrendo uma máquina de estado especial chamada máquina de Aho-Corasick, que difere de uma DFA (*Deterministic Finite Automaton*) comum, pois introduz uma nova transição definida como transição de falha para reduzir as transições de saída de cada estado (LIN, 2013).

O algoritmo AC invoca três funções (TRAN, et al., 2012): a função *goto* (g), a função de *falha* (f) e a função de *saída*. Para construir a função *goto*, (AHO e CORASICK, 1975) criam o *grafo goto*, iniciando apenas com um vértice que representa o estado 0. Após isso, o padrão "y" é adicionado através de novos vértices e novas arestas de modo que haverá, iniciando do estado inicial, um caminho para este padrão no grafo. Neste ponto a função de saída começa a ser construída e armazena o padrão na posição correspondente ao último

estado adicionado.

A inclusão de novos vértices e arestas no grafo *goto* nem sempre é necessária. Pode-se ver na Figura 2.9 que a inserção do padrão *his*, por exemplo, não necessita de uma aresta com o rótulo *h* saindo do estado 0, uma vez que esse rótulo já existe no grafo. Sendo assim, apenas os rótulos com os caracteres *i* e *s* deverão ser adicionados partindo do estado 1.

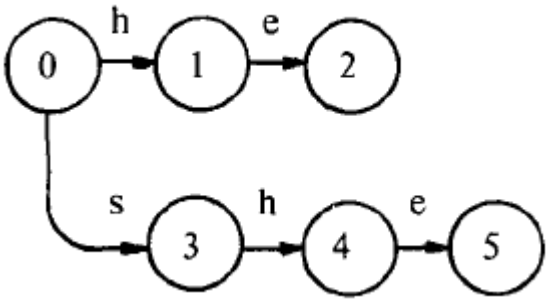


Figura 2.9: Construção do grafo goto (AHO e CORASICK, 1975).

Para finalizar a construção da função *goto*, eles adicionam um *loop* do estado zero para o próprio estado zero para todos os símbolos de entrada que não são capazes de realizar uma transição[1] do estado zero para outro estado.

(AHO e CORASICK, 1975) afirmam que a função *goto* pode ser armazenada das seguintes formas: em um vetor bidimensional (tabela), 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* (onde *s* é o estado atual e *a* o caractere de entrada). 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 função de falha é construída a partir da função *goto*. A variável *depth* é definida como a menor distância de um estado *s* até o estado inicial. Primeiro calcula-se a função de falha para todos os estados de *depth* igual a 1, depois para os estados de *depth* igual a 2, e assim por diante. Para calcular a função de falha dos estados de *depth* igual a *d*, considera-se cada estado r de *depth* igual a d - 1, e faz as seguintes ações:

1. Se *g(r,a) = falha* para todo *a*, não faz nada.
2. Caso contrário, para cada símbolo *a* tal que *g(r,a) = s*, faz o seguinte:
    a. Faz *state = f(r)*.
    b. Executa a instrução *state* ← *f(state)* zero ou mais vezes, até *a* ser obtido para *state* tal que *g(state, a)* ≠ *falha* (sendo que *g(0,a)* ≠ *falha* para todo *a*).

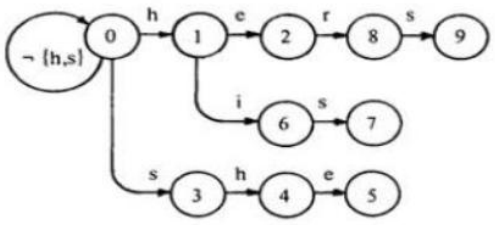
[1] Uma transição indica uma mudança de estado e é descrita por uma condição que precisa ser realizada para que a transição ocorra.

c. Faz *f(s) = g(state,a)*.

A função de saída também é atualizada durante a construção da função de falha. Ao determinar que *f(s) = s'*, faz-se a união das saídas do estado *s* com as saídas do estado *s'*.

Como exemplo, a Figura 2.10 mostra as funções utilizadas pela máquina AC para o conjunto de padrões {"he", "she", "his", "hers"}:

- O grafo orientado da Figura 2.10 (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 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-chaves para a saída com os estados designados.



(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 2.10: Funções usadas no algoritmo AC (Tran, et al., 2012).

Agora assuma o texto "ushers". A máquina AC trabalha da seguinte maneira: iniciando no estado 0, a máquina volta pro estado 0 uma vez que *g(0,'u')=0*. Pela mesma razão, a máquina entra nos estados 3,4,5 sequencialmente enquanto processa as strings 's', 'h', 'e' (*g(0,'s')=3*, *g(3,'h')=4*, *g(4,'e')=5*) e emite a saída, indicando que ele encontrou as palavras "she" e "he". Depois a máquina avança para o próximo símbolo de entrada 'r'. Uma vez que *g(5,'r')=falha*, a máquina entra no estado 2=f(5) (Figura 2.10b). Então, uma

vez que *g(2,'r')=8*, *g(8,'s')=9* a máquina AC entra no estado 9 e emite a saída "hers".

2.4. Estado da Arte

2.4.1. Algoritmo Paralelo PFAC

LIN, C. et al. (2013) propuseram uma máquina de estado que executa um algoritmo paralelo bastante eficiente chamado de *Parallel Failureless Aho-Corasick* (PFAC). Essa máquina é semelhante à máquina de Aho-Corasick, mas não utiliza as transições de falha existentes na mesma. Somado a isso, foram introduzidas as seguintes técnicas que otimizam as execuções da GPU: redução de transações do *buffer* de entrada da memória global, redução da latência de consulta à tabela de transição, eliminação da tabela de acessos de saída, evitando conflito de banco da memória compartilhada, e o melhoramento da transmissão de dados via PCI Express.

O PFAC aloca uma *thread* para cada caractere de uma *stream* de entrada a fim de identificar qualquer padrão começando na posição inicial da *thread*, O número de *threads* alocadas é igual ao tamanho da *stream*. Nesse algoritmo, se o caractere em análise não levar a máquina para outro estado, a *thread* imediatamente termina sua tarefa. Por exemplo: se a primeira letra da *stream* de entrada conseguir levar a máquina para outro estado, a segunda letra é analisada pela *thread*; se a segunda não for encontrada na máquina de estado toda a *string* é descartada.

As Figuras 2.11 e 2.12 exemplificam bem essa situação. Na Figura 2.11 é possível ver algumas *threads* com uma máquina de estado para cada caractere de entrada. A primeira *thread* chegará ao estado 2 (um dos estados finais da máquina) devido às entradas 'A' e 'B', mas, diferentemente do Aho-Corasick original, não irá para um estado de falha quando o caractere 'E' for analisado pela máquina. Já a segunda *thread* fará sua comparação de *string* a partir do caractere 'B', como pode ser visualizado nas Figuras 2.11 e 2.12.

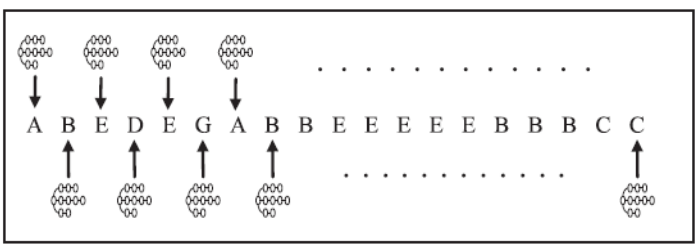


Figura 2.11: Algoritmo PFAC (LIN et al., 2013).

Resumidamente, notamos pela Figura 2.12 que em vez de usar uma *thread* para encontrar três padrões, ele usa três threads para encontrar três padrões em paralelo.

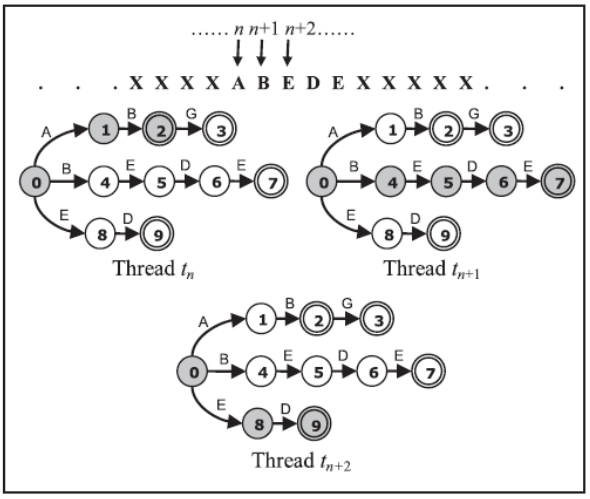


Figura 2.12: Exemplo do PFAC para os padrões "AB", "ABG", "BEDE", e "ED" (LIN et al., 2013).

Para realizar os experimentos, os autores compararam o desempenho do algoritmo Aho-Corasick no Intel Core i7-950 e o algoritmo PFAC na NVIDIA GTX580 (além de fazerem uma versão do PFAC para a CPU com o OpenMP chamada $PFAC_{OMP}$). Mais especificamente, eles tem um *host* com uma CPU Intel Core i7-950 executando o sistema operacional Linux, com memória DDR3 de 12GB em uma placa mãe ASUS P6T-SE, e um dispositivo equipado com uma GPU GTX580 GeForce da NVIDIA no mesmo sistema Core i7 com driver NVIDIA versão 260.19.29 e o CUDA versão 3.2.

Os padrões de teste foram retirados do Snort V2.8 e as *streams* de entrada foram extraídas de pacotes DEFCON que contém grandes quantidades de padrões de ataques reais e são amplamente utilizadas para testes de NIDS (*Network Intrusion Detection Systems*) comerciais.

As métricas utilizadas na comparação foram as seguintes (LIN et al., 2013):

1) Taxa de transferência de dados bruta: $\frac{8n}{T_{GPU}}$, onde 8n é o total de bits de uma stream de entrada de n caracteres e T_{GPU} o tempo de execução do *kernel* no núcleo da GPU.

2) Largura de banda efetiva do PCIe:

   $\frac{8n}{T_{host/dispositivo}}$, onde 8n é o total de bits de entrada (sendo n é o número de caracteres de entrada) e $T_{host/dispositivo}$ o tempo para mover o dado do host para o dispositivo via PCIe;

   e $\frac{16n}{T_{dispositivo/host}}$, onde 16n é o total de bits do resultado da comparação dos padrões e $T_{dispositivo/host}$ o tempo para mover os resultados da comparação do dispositivo para o host via PCIe.

3) Taxa de transferência do Sistema: $\frac{8n}{T_{total}}$, onde 8n é o total de bits de entrada e T é T_{GPU}, $T_{host/dispositivo}$ e $T_{dispositivo/host}$.

Após a execução dos testes, o desempenho do PFAC mostrou-se bastante superior ao do algoritmo Aho-Corasick. Para pacotes de entrada DEFCON de 256 MB, o $PFAC_{GPU}$ alcançou 143,16 Gbps de taxa de transferência de dados bruta, enquanto que o AC_{CPU} e o $PFAC_{OMP}$ alcançaram 1,91 e 12,61 Gbps, respectivamente. Ou seja, os resultados experimentais mostraram que o algoritmo proposto executado na GPU NVIDIA é 74,95 vezes mais rápido do que o algoritmo Aho-Corasick implementado em uma CPU se for contabilizado apenas a taxa bruta. O autor enfatizou bem este resultado, porém também foi importante mostrar a taxa do sistema (ver Figura 2.13) que, ao levar em consideração as transferências entre CPU e GPU, deu uma reduzida na taxa de transferência da versão de GPU, mas que mesmo assim foi superior a todas as outras versões implementadas.

DEFCON Packets		AC_{CPU}	$DPAC_{OMP}$	$PFAC_{OMP}$	$PFAC_{GPU}$		
Input Size	# of matched Patterns	Raw data Throughput (Gbps)	Raw data Throughput (Gbps)	Raw data Throughput (Gbps)	Raw data Throughput (Gbps)	H2D / D2H Effective Bandwidth (Gbps)	System Throughput (Gbps)
32 MB	458,491	1.55	8.73	9.51	110.64	47.92 / 51.20	14.48
64 MB	840,956	1.58	8.96	9.89	119.37	48.00 / 51.28	14.68
128 MB	933,759	1.82	9.73	11.79	135.02	48.08/ 51.28	14.87
256 MB	**1,118,653**	**1.91**	**9.71**	**12.61**	**143.16**	**48.08 / 51.28**	**15.00**

Figura 2.13: Comparação das taxas de transferências com pacotes DEFCON de tamanhos diferentes (LIN et al., 2013).

Em outro experimento LIN, C. et al. (2013) compararam o PFAC executando em outras gerações de GPUs, como: GTX 580, GTX 480, GTX 295 e Tesla M2050. Foi verificado que o PFAC alcança um maior desempenho nas novas gerações de GPUs Nvidia com mais núcleos, largura de banda de memória e maior capacidade de computação.

O lado negativo da abordagem é que o PFAC pode ter problemas em relação ao desbalanceamento de cargas, pois as *threads* que ficam responsáveis pelas maiores cargas dominam o tempo de execução de uma *warp*.

2.4.2. Memórias Compartilhadas e Caches de Textura da GPU

Em outro trabalho, TRAN, N. et al. (2012) apresentaram uma nova técnica de paralelização a qual armazena de forma eficiente os dados do texto de entrada e os dados de referência (padrões de comparação) nas memórias compartilhadas e caches de texturas da GPU. Além disso, a nova abordagem programou eficientemente os acessos à memória a fim de minimizar a sobrecarga de dados nas memórias compartilhadas.

Para o uso eficiente da memória, eles colocaram cuidadosamente os dados de entrada e

os dados de referência. Para os dados de entrada, cujo tamanho é de pelo menos algumas centenas de megabytes, eles podem ser total ou parcialmente colocados na memória global do DRAM-Gráfico (GDRAM), também conhecida como memória do dispositivo. No entanto, eles são grandes demais para caber nas memórias compartilhadas e caches de textura. Por conseguinte, a fim de carregar os dados de entrada de forma eficiente para a memória compartilhada, os autores organizaram cuidadosamente os pedidos de acesso à memória, de modo que o número de acessos à memória global pôde ser minimizado.

Eles organizaram os dados de referência, com o qual os dados do texto de entrada são comparados, em uma tabela bidimensional chamada Tabela de Transição de Estado (STT) e colocaram esses dados na memória de textura de modo que a parte usada ativamente 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.

A Figura 2.14 mostra um exemplo da STT, onde as linhas representam os estados e as colunas representam os caracteres de entrada. Ela é uma tabela comum que é acessada por todas as *threads* aleatoriamente para a comparação de padrão.

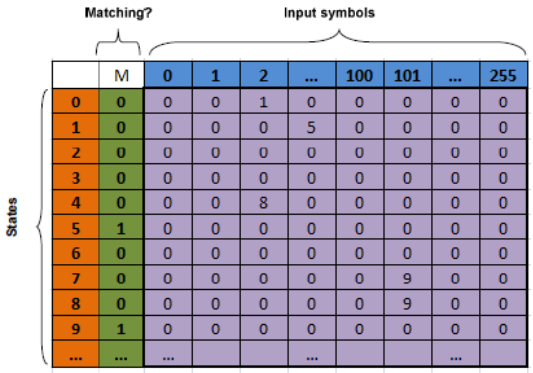


Figura 2.14: Ilustração da Tabela de Transição de Estado (TRAN et al., 2012).

Eles trabalharam com duas novas abordagens:

1) **Usando memória global (Figura 2.15):** Nesta abordagem o AC foi paralelizado de maneira direta, armazenando os dados do texto de entrada na memória global. Esses dados foram divididos em vários *chunks*. Cada *chunk* foi atribuído a cada processador de *thread* no mesmo bloco de *thread*. Ao final, cada *thread* aplica a comparação de padrão no seu próprio *chunk*.

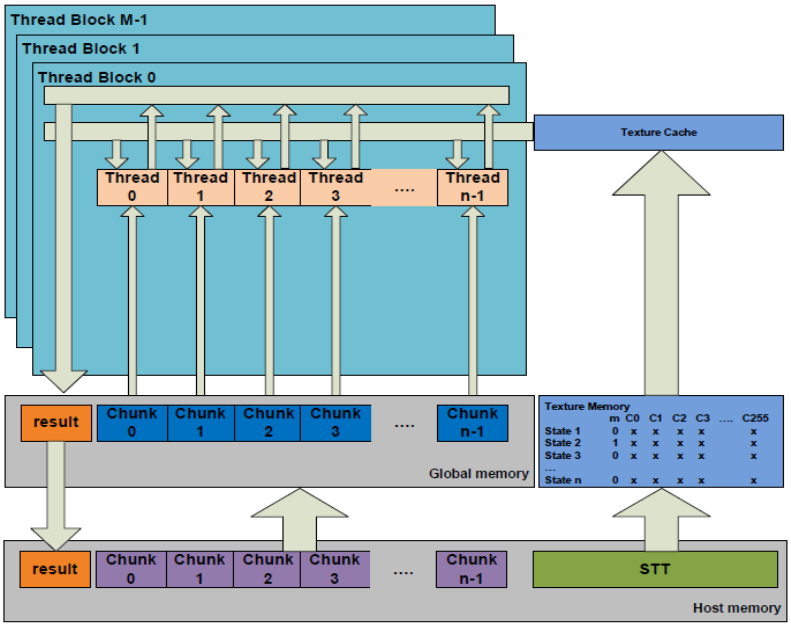


Figura 2.15: Ilustração da abordagem de memória global (TRAN et al., 2012).

2) **Abordagem de memória compartilhada (Figura 2.16):** Nesta abordagem os dados também foram divididos em *chunks*, mas foram levados da memória global para a memória compartilhada usando uma leitura coordenada chamada Acesso *Coalesced* (TRAN et al., 2012).

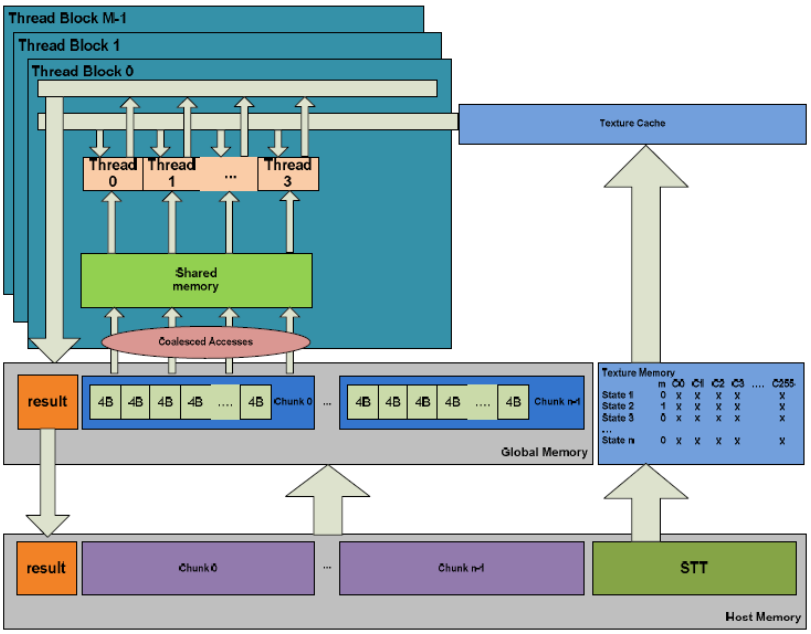


Figura 2.16: Ilustração da abordagem de memória compartilhada (TRAN et al., 2012).

Para a execução do experimento, usaram CUDA na GPU NVIDIA GeForce 9500 GT, uma placa mais antiga se comparada com a placa GTX580 utilizada nos experimentos do PFAC da subseção 2.4.1. Ela é uma placa de baixo desempenho, contem apenas 32 processadores de *streams* (ou processadores de *threads*) organizados em 4 multiprocessadores (ou blocos de *threads*), operando a 1,35 GHz com 256 MB de memória do dispositivo. Já o sistema experimental completo usou um processador multicore da Intel

(2,2 GHz Intel Core2Duo 4) com 2GB de memória principal. O SO usado foi o Linux 5.5 CentOS.

Os autores (TRAN, N. et al., 2012) realizaram três experimentos: Algoritmo AC na versão serial, na versão memória global e na versão memória compartilhada. Diferentes tamanhos de textos de entradas e quantidades diferentes de padrões foram usados para medir o desempenho.

A Figura 2.17 mostra a comparação do tempo de execução das três abordagens usando diferentes tamanhos de *strings* de entrada e fixando o número de padrões em 1000.

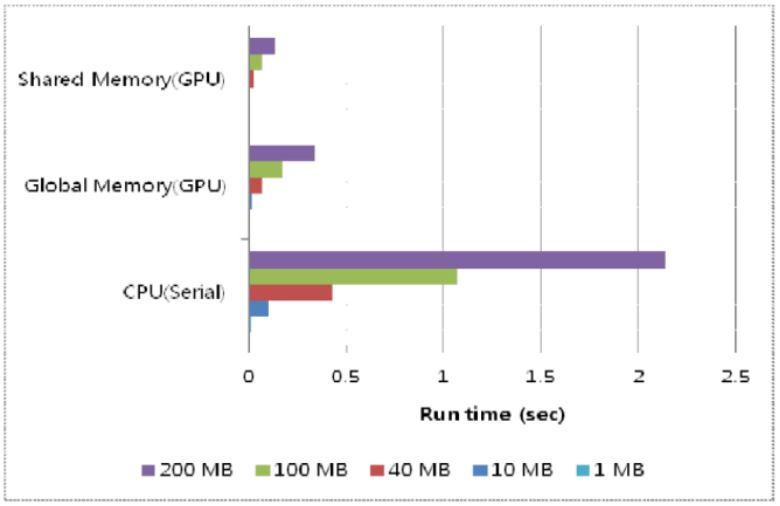


Figura 2.17: Resultado das três abordagens usando 1000 padrões (TRAN et al., 2012).

A Figura 2.18 mostra a comparação das velocidades da versão de memória global e da versão de memória compartilhada com a velocidade da memória serial. A velocidade máxima alcançada nessa comparação foi 15,72 vezes usando um texto de entrada de 200 MB e 1000 padrões na versão de memória compartilhada. Já na versão de memória global a velocidade foi de 4,71 a 5,76 vezes comparado com a versão serial.

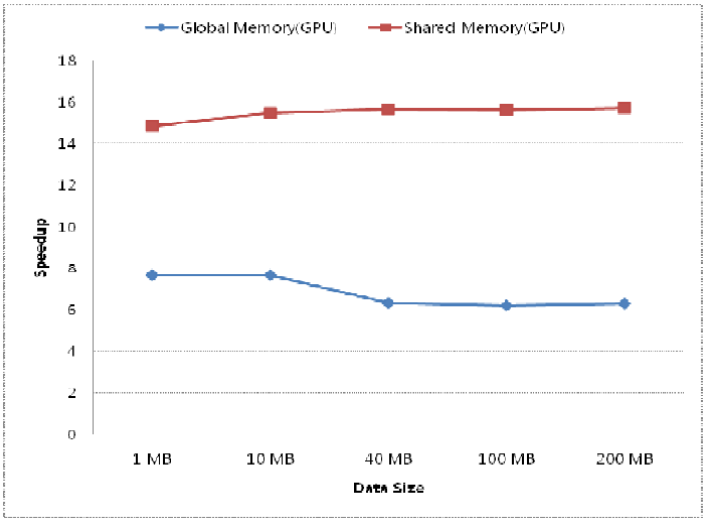


Figura 2.18: Comparação das velocidades (TRAN et al., 2012).

Já em um segundo experimento, o tamanho do texto de entrada foi fixado em 200 MB e o número de padrões foi sendo aumentado até 1000 padrões, como mostra a Figura 2.19.

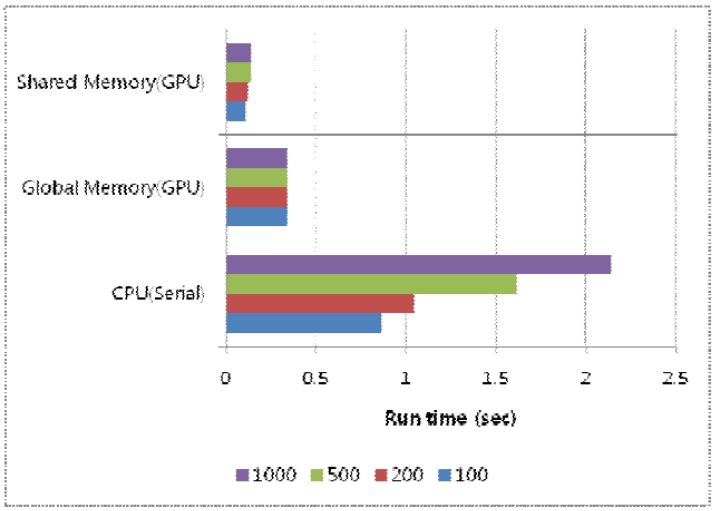


Figura 2.19: Resultado das três abordagens para 200 MB de texto de entrada (TRAN et al., 2012).

Como pode ser visto na Figura 2.19, o tempo de execução das duas versões paralelas na GPU alteraram insignificantemente com o número de padrões. Isto foi especialmente verdadeiro para a versão de memória global.

2.4.3. GPU x CPU

THAMBAWITA, D. et al. (2014) mostraram quando as GPUs podem ser usadas para aplicações de correspondência de *strings* usando o algoritmo de Aho-Corasick como referência. Para isso, eles compararam o desempenho do Aho-corasick em uma CPU e uma GPU a fim de identificar o dispositivo adequado para obter o melhor desempenho. Essa comparação foi feita tendo a mudança de tamanho dos dados de entrada como o principal parâmetro.

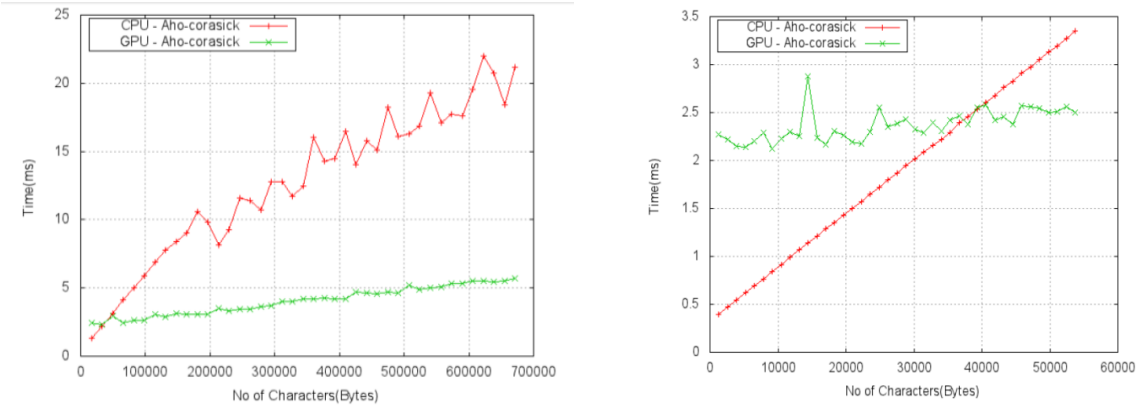
Para o experimento foi usada uma versão serial do algoritmo Aho-Corasick, chamada Multifast (que é o algoritmo geral do Aho-Corasick, com transições de falha e sem quaisquer técnicas de processamento paralelo) e na versão paralela foi usado o PFAC (implementado em CUDA), sendo que o multifast foi executado em uma CPU Intel Core i5-3230M @ 2.60GHz com 3 MB de cache e o PFAC em uma GPU NVIDIA GeForce GT 740M de 384 cores CUDA.

Mediu-se o tempo de execução do experimento em milisegundos, mas alguns fatores foram desconsiderados, como o tempo para inicializar a GPU e o tempo para execução da função *printf()*, porque esses fatores afetam diretamente o tempo medido. Portanto, ambos foram ignorados do tempo medido.

Eles usaram como padrões cinco permutações usando vinte e seis caracteres do alfabeto inglês (Exemplo: abcdefghijklmnopqrstuvwxyz). Também adicionaram o alfabeto inglês repetidamente dentro de um arquivo de texto para ser usado como arquivo de texto de

entrada para o experimento.

No primeiro experimento eles aumentaram o tamanho do arquivo de entrada de 16375 bytes para 671375 bytes e obtiveram o resultado mostrado na Figura 2.20a.



a) Carga normal. *b) Carga reduzida.*

Figura 2.20: CPU e GPU com Aho-Corasick (Thambawita et al., 2014).

Um importante ponto de mudança pôde ser identificado sobre o gráfico da Figura 2.20a próximo a 40000 bytes (onde o desempenho da GPU ficou abaixo da CPU). Por isso, eles fizeram outro experimento mudando o tamanho do arquivo de 1310 a 53710 bytes e obtiveram o resultado mostrado na Figura 2.20b.

Pode-se notar que o ponto de mudança ocorreu por volta dos 40000 bytes (40 kB). Portanto, para fazer comparação de *string* com um algoritmo como o Aho-Corasick e um arquivo de entrada de tamanho menor que 40000 bytes é melhor usar a CPU do que a GPU.

2.4.4. Paralelização de vários Algoritmos

Por fim, o trabalho de Kouzinopoulos e Margaritis (2008) analisou as parelizações dos algoritmos de comparação de *strings* Naive, Kanuth-Morris-Pratt, Boyer-Moore-Horspool e o Quick-Search, feitas usando a plataforma CUDA. O conjunto de dados e a metodologia experimental tiveram 3 sequências de referências no contexto da bioinformática: 1) o genoma bacteriano da Yersinia pestis (4,6 Mb de tamanho); 2) do Bacillus anthracis (5,2 Mb de tamanho); 3) um Cromossomo Artificial Bacteriano (BAC) simulado do Homo Sapiens. O padrão de consulta foi construído de subsequências escolhidas aleatoriamente a partir de cada sequência de referência com um comprimento de 25, 50, 200 e 800 caracteres. A partir desse conjunto de dados, eles compararam cada algoritmo com sua respectiva versão serial e seu melhor resultado foi quando paralelizou o Boyer-Moore-Horspool obtendo uma velocidade 20 vezes maior em relação a sua versão serial e 24 vezes maior quando passou a usar memória compartilhada em relação à sua versão com memória

global.

2.4.5. Mapeamento Sistemático do Algoritmo AC paralelo

JÚNIOR, J. B. S. et al. (2016) fizeram um mapeamento sistemático acerca do algoritmo AC paralelo em revistas renomadas na área de computação. Eles conseguiram levantar 22 artigos que fizeram a paralelização em uma GPU e mostraram que máquina AC é colocada na memória de textura em 61,11% dos experimentos, inclusive nos trabalhos citados acima. Apenas LIN et al. (2010) usaram a memória compartilhada, porém, devido ao pouco espaço de armazenamento disponível nessa memória, eles não conseguiram executar a STT completa e tiveram que dividir a STT por grupos de ataques, lançando determinada STT à medida que a *string* chegue na GPU. Dessa forma, é necessário realizar uma pré-filtragem na *string* que será comparada, para saber qual STT será lançada, e essa pré-filtragem faz o algoritmo ficar mais lento.

Além disso, eles mostraram também as técnicas de otimização aplicadas nos trabalhos encontrados, como por exemplo, 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) implementaram a STT na memória de textura e 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 bit-mapeamento 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 para diminuir o tamanho do DFA e o bit-mapeamento para compactar a STT formada a partir do DFA reduzido.

2.4.6. A técnica de compactação por Bit-mapeamento

Tanto PUNGILA (2013, 2015) e BELLEKENS (2014) usaram a técnica de bitmapeamento para representar o DFA. Nessa técnica, também conhecida como mapeamento por bit, cada nó do DFA tem um bitmap associado (um *array* de 8 células de 32 bits cada uma) onde seus filhos são representados nesse mapa de 256 bits, sendo que cada bit corresponde a uma letra do alfabeto ASCII (BELLEKENS, 2014). 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.

A Figura 2.21 ilustra como os nós são organizados. O primeiro nó (‘A’) tem dois filhos, ‘B’ e ‘D’. O *offset* do nó 1 apenas aponta para o seu primeiro filho (nó 3) correspondente a letra ‘B’, no entanto, para acessar o segundo nó (‘D’) é realizado um cálculo: faz-se a soma do número de bits setados como 1 no bitmap que sejam anteriores ao caractere de entrada pretendido. Neste caso, o bitmap tem 2 valores setados como 1, em ‘A’ (66 em decimal ASCII) na coluna 3/posição 2 e em ‘D’ (68 em decimal ASCII) que na Figura 2.21 está implícito na coluna3/posição 4, ou seja, o único setado como 1 antes de ‘D’ é ‘B’. A transição é calculada da seguinte forma: nóAtual = *offset* + *i,* onde *offset* é o *offset* do primeiro nó e *i* é o valor da soma. Como *i* é igual a 1 e o offset de ‘A’ é igual a 3, o segundo filho pode ser encontrado na posição 3+1=4, que corresponde ao nó 4 da Figura 2.21.

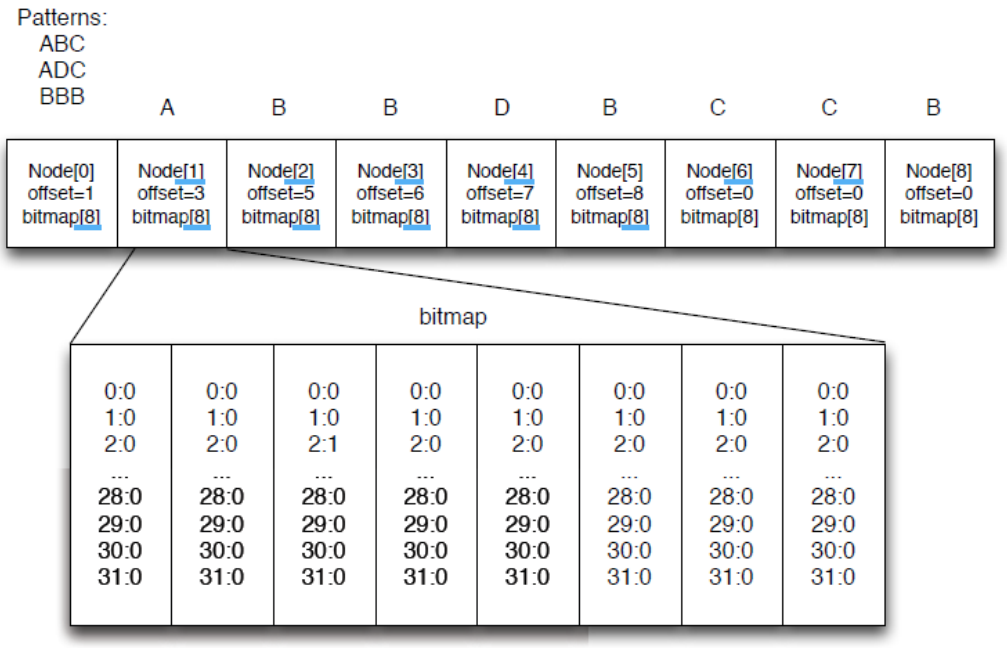


Figura 2.21: Ilustração do bitmap correspondente ao primeiro nó de um DFA (BELLEKENS ,2014).

2.5. Considerações Finais do Capítulo

Este capítulo deu uma visão geral sobre o funcionamento do Snort, das GPUs e de como andam as pesquisas sobre a paralelização dos algoritmos de comparação de *strings* através de GPUs, com foco no algoritmo Aho-Corasick. Com isso, nota-se que as GPUs têm sido uma ferramenta eficaz na aceleração da comparação de *strings* e que elas têm uma importante hierarquia de memória que pode ser explorada a fim de aumentar ainda mais o desempenho da busca dos padrões nos pacotes de redes. Por fim, algumas técnicas de otimizações e compactações podem ser necessárias devido ao tamanho limitado de suas memórias.

3. Metodologia da Pesquisa

Com base nos pressupostos teóricos, alguns dos principais algoritmos de comparação de *strings* foram paralelizados, de modo que fosse possível avaliar diversos cenários, comparando os algoritmos paralelizados entre si e entre suas versões seriais, a fim de verificar aquele que obteve o melhor desempenho para implementá-lo no IDS Snort. Utilizou-se a plataforma de computação paralela CUDA da Nvidia como recurso necessário para a paralelização dos algoritmos.

Através do processo de pesquisa da Figura 3.1 foi possível estudar, analisar, discutir e identificar a melhor forma de buscar padrões em um texto de forma paralela.

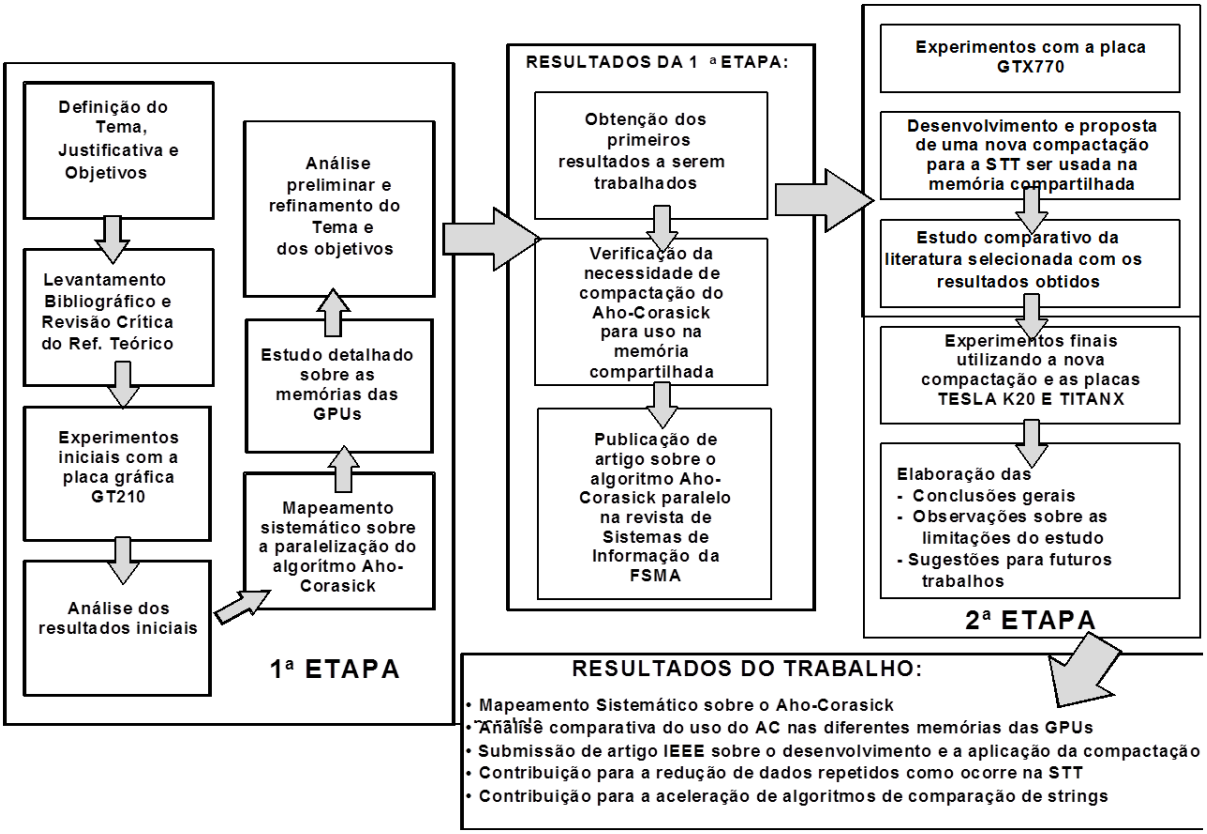


Figura 3.1: Etapas da pesquisa.

3.1. Materiais e Métodos

Para a implementação do proposto na Metodologia, foi necessária a utilização dos seguintes recursos:

- Hardware e Software do Sistema
  - Características da máquina de teste **GT210**:

Sistema Operacional Windows 8;

Processador Intel Pentium Dual Core E5700 de 3.0 GHz;

Memória RAM: 4 GB;

GPU: GEFORCE 210;

CUDA Cores: 16;

Clock da GPU: 589 MHz;

Memória Global: 1 GB;

Taxa de Transferência da memória (*Bandwidth*): 8 GB/s.

- Características da máquina de teste **Tesla K20**:

  Sistema Operacional RedHat 6.4;

  Processador Intel Xeon Ten-Core E5-2660v2 de 2.2 GHz;

  Memória RAM: 64 GB;

  GPU: NVIDIA TESLA K20;

  CUDA Cores: 2496;

  Clock da GPU: 706 MHz;

  Memória Global: 5 GB.

  Taxa de transferência da memória (*Bandwidth*): 208 GB/s

- Características da máquina de teste **Titanx**:

  Sistema Operacional Linux 3.19.0-80-generic #88~14.04.1-Ubuntu;

  Processador Intel Core i5 1,2 GHz 6ª geração;

  Memória RAM: 8 GB;

  GPU: NVIDIA TITAN X (Pascal);

  CUDA Cores: 3584;

  Clock da GPU: 1417 MHz;

  Memória Global: 12 GB;

  Taxa de transferência da memória (Bandwidth): 480 GB/s;

- Características da máquina de teste **CPU**:

  Sistema Operacional Windows 8.1 de 64 bits;

  Processador Intel Core i3-4005U 1,70 GHz;

  Memória RAM: 4 GB;

  GPU: Não possui;

Neste trabalho, foi realizada a paralelização dos algortimos Naive (Força-Bruta) e Aho-Corasick, sendo que a análise de desempenho dos mesmos pode ser vista mais adiante nas seções 4 e 5.

Inicialmente foi necessário decidir em usar o algoritmo Naive em sua versão pura, ou seja, sem nenhuma técnica que acelere seu processamento, ou o Naive Melhorado. Após essa decisão, começaram os experimentos de comparação entre o Naive e o Aho-Corasick paralelos. O texto usado nesses experimentos iniciais para buscar os padrões pode ser visto no anexo C. Esse texto foi replicado dez vezes se tornando um texto de 40960 bytes. Já os padrões utilizados foram: {“Brazil”, “it”, “home”, “primeiro”, “today”, “in”, “and”, “the”}.

Para usufruirmos da computação paralela possibilitada pela GPU, seguimos a abordagem de *chuncks*, dividindo o texto de entrada em fragmentos que são processados em paralelo, como TRAN, N. et al. (2012) fizeram em seu trabalho. Ainda baseado no trabalho deles, também exploramos a hierarquia de memórias disponível nas GPUs criando três abordagens diferentes para o Naive e três para o Aho-Corasick. Adicionalmente, exploramos a ocupância dos programas nos núcleos da GPU a fim de verificar o impacto que esta métrica pode proporcionar ao desempenho dos algoritmos paralelos.

Com a finalização dos experimentos iniciais na placa GT210, verificou-se que o algoritmo Naive teve um desempenho inferior ao algoritmo Aho-Corasick e, por isso, os experimentos finais voltaram-se para o aperfeiçoamento do Aho-Corasick paralelo. Passou-se a utilizar padrões de ataques reais, retirados de dezenove regras do Snort, além de pacotes de redes como texto de entrada. Neste ponto, notou-se que não seria possível executar a STT como uma matriz bidimensional na memória compartilhada das GPUs TITAN X e TESLA K20 devido à baixa capacidade de armazenamento dessa memória. Por conseguinte, foi necessário diminuir o tamanho da STT, sendo aplicada uma nova técnica de compactação, a qual pode ser vista em detalhes na Seção 3.4.

4. Paralelização Inicial e Compactação do Aho-Corasick

Nesta seção é descrita a implementação dos algoritmos desenvolvidos no trabalho, incluindo os detalhes sobre suas estruturas. De forma geral, os algoritmos Força-Bruta e Aho-Corasick foram paralelizados através da plataforma de computação paralela CUDA da Nvidia. Ao final, o Aho-Corasick paralelo foi compactado a fim de possibilitar sua execução na memória compartilhada com o intuito de se obter *speedups* cada vez maiores.

4.1. Comunicação CPU - GPU

A Figura 4.1 ilustra como a CPU se comunica com a GPU e quais variáveis são enviadas de uma plataforma para outra. O texto de entrada, o qual possui os pacotes de redes onde os padrões de ataques são buscados, fica armazenado em um arquivo de texto no formato de uma matriz.

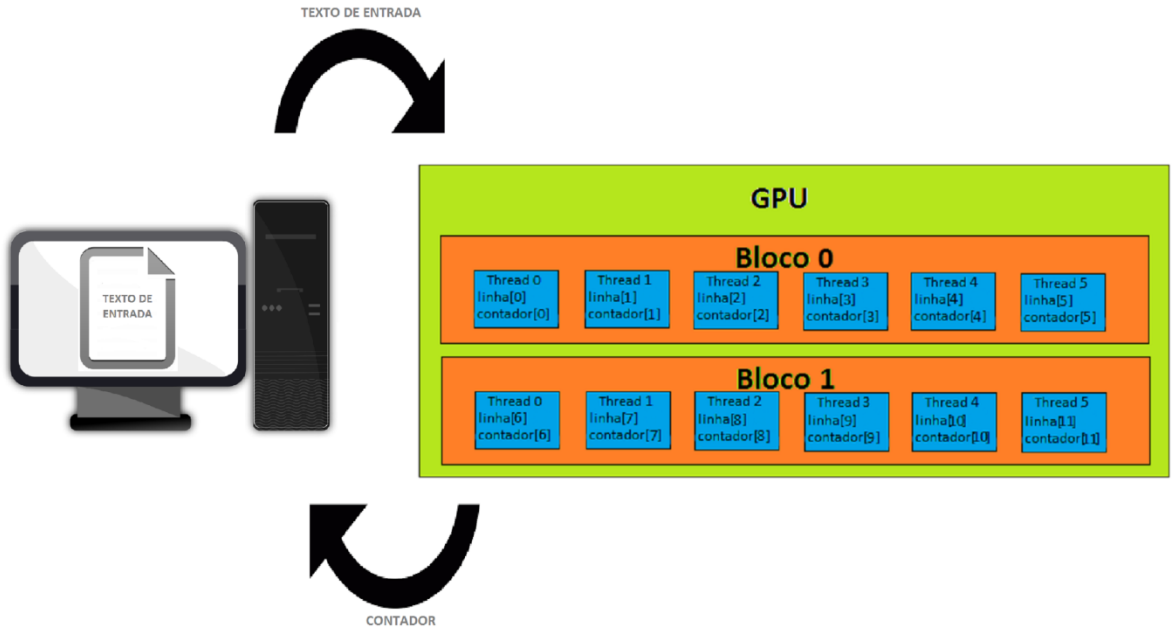


Figura 4.1: Ilustração do processamentos dos softwares.

O texto em matriz é enviado da CPU para a GPU através da função *main*. Cada *thread* da GPU processa uma linha da matriz e conta a quantidade de padrões encontrados. Ao final do processamento, o contador é enviado para o host.

A função *main* também é responsável pela chamada ao *kernel* de comparação de *strings* na GPU e pelo cálculo do tempo de execução médio desse *kernel* através do *event* do CUDA. Ao final, o tempo médio e a quantidade de padrões encontrados no texto são impressos na tela. A taxa de transferência, medida em Megacaracteres por segundo (Mcps),

é calculada através da quantidade de caracteres total do texto de entrada dividida pelo tempo de execução do algoritmo.

O fluxograma completo da função *main* nos testes iniciais pode ser visto na Figura 4.2.

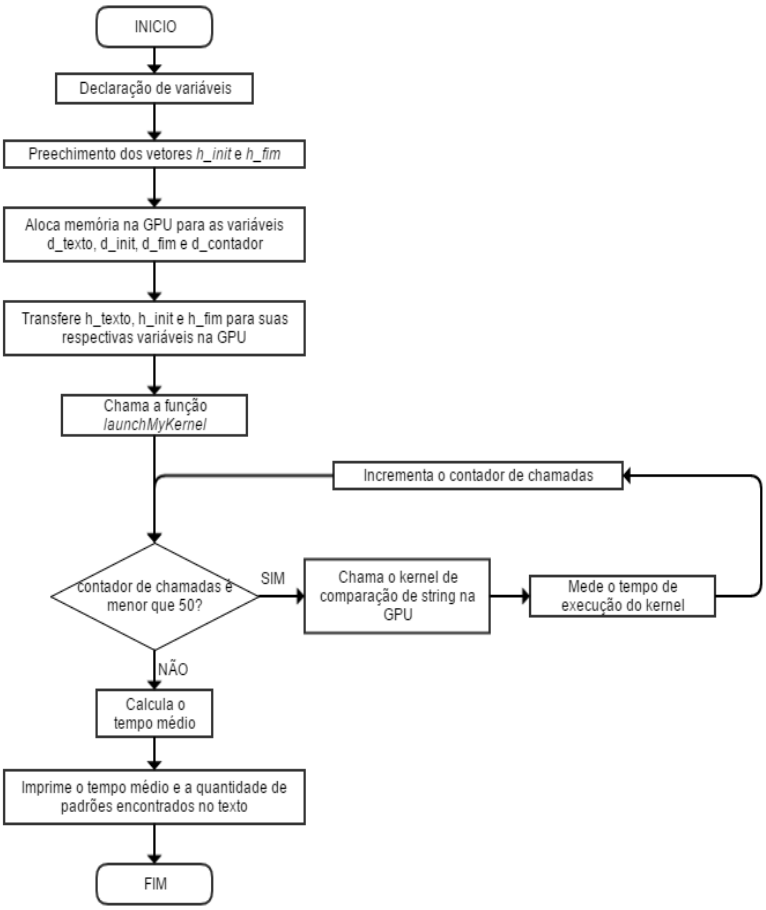


Figura 4.2: Fluxograma da função main de todos os algoritmos desenvolvidos nos testes iniciais.

A função "launchMyKernel" é disponibilizada no site da Nvidia (Harrys). Ela foi implementada e adaptada com duas finalidades bem definidas:

1) Atribuir os valores do tamanho do *grid* e do bloco. Essa atribuição pode ocorrer diretamente atribuindo valores desejados às variáveis *gridSize* e *blockSize* ou ser calculada através da função *cudaOccupancyMaxPotentialBlockSize*, que calcula dinamicamente tamanhos de blocos que atinjam uma ocupância máxima teórica, e, após isso, calcula-se o tamanho do grid com a fórmula:

$$gridSize = (arrayCount + blockSize - 1)/blockSize. \qquad \textbf{Eq. 3.1}$$

2) Medir a ocupância teórica com a função *cudaOccupancyMaxActiveBlocksPerMultiprocessor*. Essa função prediz a ocupância (dada pela quantidade de blocos ativos máximos) em termos do tamanho do bloco e uso de memória compartilhada de um kernel. Para calcular a ocupância em termos de porcentagem, podemos utilizar a fórmula seguinte:

$$Ocupância = \frac{\left(\frac{maxActiveBlock * blockSize}{props.warpSize}\right)}{\left(\frac{props.maxThreadsPerMultiProcessor}{props.warpSize}\right)} \qquad \textbf{Eq. 3.2}$$

Onde as variáveis *props.warpSize* e *props.maxThreadsPerMultiProcessor* podem ser obtidas através da biblioteca *cudaGetDeviceProperties* da Nvidia.

4.2. Naive Puro e Melhorado

Inicialmente, implementamos uma versão paralela dos algoritmos Naive Puro e Naive Melhorado (DAN, 1997) a fim de definir qual deles seria usado na comparação com o Aho-Corasick paralelo. A Figura 4.3 mostra uma implementação mista, utilizando a linguagem portugol adaptada para nosso problema, onde as linhas destacadas pertencem exclusivamente à versão Naive Melhorado paralelo. As linhas não marcadas estão presentes nos dois algoritmos.

```
1  ALGORITMO NaiveParallel
2  DECLARE idx, j, skip NUMERICO
3  DECLARE pattern[][] LITERAL
4
5  idx <- threadIdx.x + blockIdx.x * blockDim.x
6  d_contador[idx] <- 0
7  j <- 1
8  skip <- 0
9
10 para i <- 1 ate m faca
11      j <- 1
12      skip <- 0
13      se d_texto[i] = pattern[idx][0]
14      entao enquanto pattern[idx] <> "\0" faca
15           se d_texto[i+j] = pattern[idx][0]
16           entao
17                skip <- j - 1
18
19           se d_texto[i+j] <> pattern[idx][j]
20           entao
21                i <- i + skip
22                break
23                senao
24                     se d_texto[i+j] <- pattern[idx][j]
25                     entao
26                          se pattern[idx][j+1] = "\0"
27                          entao
28                               d_contador[idx] <- d_contador[idx] + 1
29                               skip <- 0
30
31           j <- j + 1
32
33 FIM_ALGORITMO
```

Figura 4.3: Pseudocódigo base para os experimentos com o Naive.

Essa implementação compara o texto completo[2] com padrões paralelizados, por exemplo: o núcleo 1 da GPU percorre o texto comparando com o padrão 1, o núcleo 2 percorre o mesmo texto comparando com o padrão 2, e assim sucessivamente até o núcleo

[2] Vale ressaltar que nos experimentos de comparação entre o Naive e o Aho-Corasick paralelos explicados mais adiante o texto não é percorrido completamente, ele é dividido em *chunks*, o que faz aumentar o desempenho do algoritmo, diminuindo o tempo de execução de 68,75 ms para 5,69 ms. Porém nos experimentos de comparação entre o Naive Puro e Melhorado todos os *threads* percorreram a mesma sequência na memoria global.

8, uma vez que temos 8 padrões.

O algoritmo é iniciado declarando as variáveis *idx*, que recebe o identificador do *thread* do bloco e é a responsável pela paralelização do algoritmo, *j* que é um contador responsável por percorrer o padrão quando sua letra inicial é encontrada no texto, e *skip* que é usado para pular comparações exclusivamente no Naive Melhorado. A matriz de caracteres *pattern* armazenou os padrões {“Brazil”, “it”, “home”, “primeiro”, “today”, “in”, “and”, “the”} para simular o caso médio e “abxwwwyabxwwwz” repetidamente em todas as suas oito linhas para simular o pior caso. *Pattern* é acessada de forma paralela no *loop* da linha 10. Já o *d_contador* é uma variável da GPU que armazena a quantidade de padrões encontrados no texto em cada núcleo de comparação.

Na linha 10 do algoritmo se inicia um *loop* que percorre completamente o texto de tamanho “m” que fica armazenado na memória global da GPU. Caso a primeira letra de *pattern* seja encontrada no texto, *pattern* é comparado com o texto, caractere a caractere, no *loop* que se inicia na linha 14, até não existir uma correspondência ou até que seu caractere especial final (‘\0’) seja encontrado. Neste último caso o padrão foi percorrido por completo, o que indica que uma correspondência foi encontrada, havendo o incremento de *d_contador*.

Após os testes iniciais, no pior caso o Naive Melhorado paralelo conseguiu um desempenho 2,63% maior que o Naive Puro paralelo, porém no caso médio a versão Pura superou a versão Melhorada em 22,16%. Isso ocorreu porque nenhum dos padrões do caso médio possui a característica necessária para o salto ocorrer, que é ter o primeiro caractere no meio do seu conteúdo. Sendo assim, no Naive Melhorado foi adicionada mais complexidade (linhas de códigos) que não está sendo aproveitada. Por essas razões, o Naive Puro foi escolhido para ser comparado com o algoritmo Aho-Corasick.

Nota-se que o algoritmo desenvolvido tem indireções e condicionais, o que faz os *threads* dessincronizarem. Os algoritmos podem se tornar mais otimizados para a GPU através de melhoria no código-fonte, porém, como os dois tem essa característica de indireções e condicionais, acreditamos que a comparação entre os dois algoritmos paralelos ainda seja válida.

Para os experimentos de comparação com o Aho-Corasick paralelo, foram criadas três abordagens do Naive Puro paralelo utilizando de forma diferente as memórias da GPU. Em cada abordagem, variou-se a configuração do *grid* e bloco de modo a obter uma melhor ocupância. Os padrões utilizados foram mais uma vez {“Brazil”, “it”, “home”, “primeiro”, “today”, “in”, “and”, “the”}.

Na primeira abordagem os padrões se mantiveram no registrador e o texto da memória

global foi acessado por partes (*chunks*). Para fazer isso foi necessário fazer algumas mudanças no algoritmo da Figura 4.3, como mostra o trecho de código da Figura 4.4.

```
8  para n <- 1 ate 8 faca
9       para i <- d_init[idx] ate d_fim[idx] faca
10           j <- 1
11           se d_texto[i] = pattern[n][0]
12           entao enquanto pattern[n] <> "\0" faca
```

Figura 4.4: Trecho do pseudocódigo da abordagem 1 do Naive paralelo.

Podemos notar que o *loop* da linha 8 foi adicionado para que em vez de um núcleo acessar apenas um padrão, ele acesse todos os padrões e o compare com o *chunk*. E o *loop* que percorre o texto (linha 9), em vez de iniciar da primeira posição do texto até a última, inicia da primeira posição do *chunk* até a última do mesmo.

Na segunda abordagem os padrões foram enviados para uma matriz de caracteres na memória compartilhada. O *chunk* se manteve na memória global como na abordagem 1.

Na terceira abordagem os padrões voltaram a ser declarados no registrador e o *chunk* foi enviado da memória global para o vetor *text* no registrador, como mostra a Figura 4.5.

```
8  DECLARE len NUMERICO
9  DECLARE text[] LITERAL
10 len <- 0
11 para i <- d_init[idx] ate d_fim[idx] faca
12      text[len] <- d_texto[i]
13      len <- len + 1
```

Figura 4.5: Trecho do pseudocódigo da abordagem 3 do Naive paralelo.

O gráfico da Figura 4.6 mostra os resultados obtidos com as três abordagens do Naive. A segunda abordagem, com uma ocupância de 54%, obteve o melhor desempenho entre todos os testes, tendo um tempo de execução de 5,69 ms com taxa de transferência de 7,20 MBps.

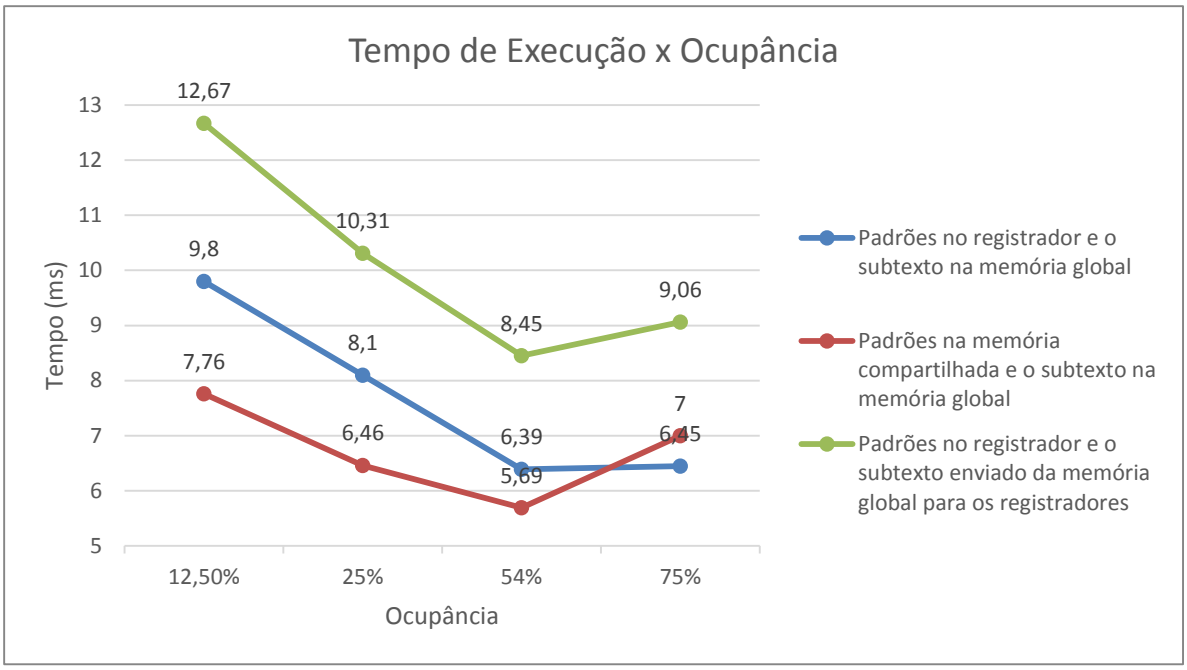


Figura 4.6: Resultados experimentais das abordagens do Naive.

A abordagem com os registradores obteve os piores resultados porque provavelmente houve a utilização de vetor nos registradores da GPU. Como foram utilizados vetores para armazenar os padrões, esses dados foram devem ter sido levados automaticamente de volta para a memória global (mais especificamente na memória local). Somado a isso, o fato de ter que levar o *chunk* da memória global para o registrador através de um *loop* que passa caractere a caractere ocasionou a perda de desempenho nessa abordagem.

4.3. Aho-Corasick Paralelo

A primeira abordagem é representada aqui na linguagem portugol adaptada para nosso problema. Todos os estados e a variável falha são representados por números inteiros. Nessa primeira abordagem o *chunk* ficou na memória global, como podemos ver pelo índice do contador *i* na linha 9 da Figura 4.7. Ao final do algoritmo, ao invés de imprimirmos a saída da função *output* como no algoritmo original, incrementamos o *d_contador* caso um padrão tenha sido encontrado (linhas 19 e 20) para podermos comparar com as abordagens do Naive.

```
1  ALGORITMO ahoCorasickParallel
2  DECLARE idx, est_atual, est_proximo, i NUMERICO
3
4  idx <- threadIdx.x + blockIdx.x * blockDim.x
5  d_contador[idx] <- 0
6  est_atual <- est0
7  est_proximo <- -1
8
9  para i <- d_init[idx] ate d_fim[idx] faca
10      est_proximo <- gotoAC(est_atual, d_texto[i])
11
12      se est_proximo = falha
13      entao
14          est_proximo <- failAC(est_atual)
15          i <- i - 1
16
17      est_atual <- est_proximo
18
19      se outputAC(est_atual) <> ""
20      entao d_contador[idx] <- d_contador[idx] + 1
21
22 fim_algoritmo
```

Figura 4.7: Pseudocódigo da primeira abordagem do Aho-Corasick paralelo.

Na segunda abordagem o *chunk* foi enviado da memória global para o vetor *text* no registrador, como na terceira abordagem do Naive (Ver Figura 4.5).

O experimento com as duas abordagens do Aho-Corasick paralelo mostrou que a abordagem onde o *chunk* é acessado diretamente da memória global teve um melhor desempenho se comparado com a segunda abordagem, independentemente da ocupância. Mais uma vez a GPU não fez o armazenamento dos dados (neste caso, o texto de entrada) nos registradores, colocando-os de volta na memória global, o que ocasionou a perda de desempenho desta abordagem.

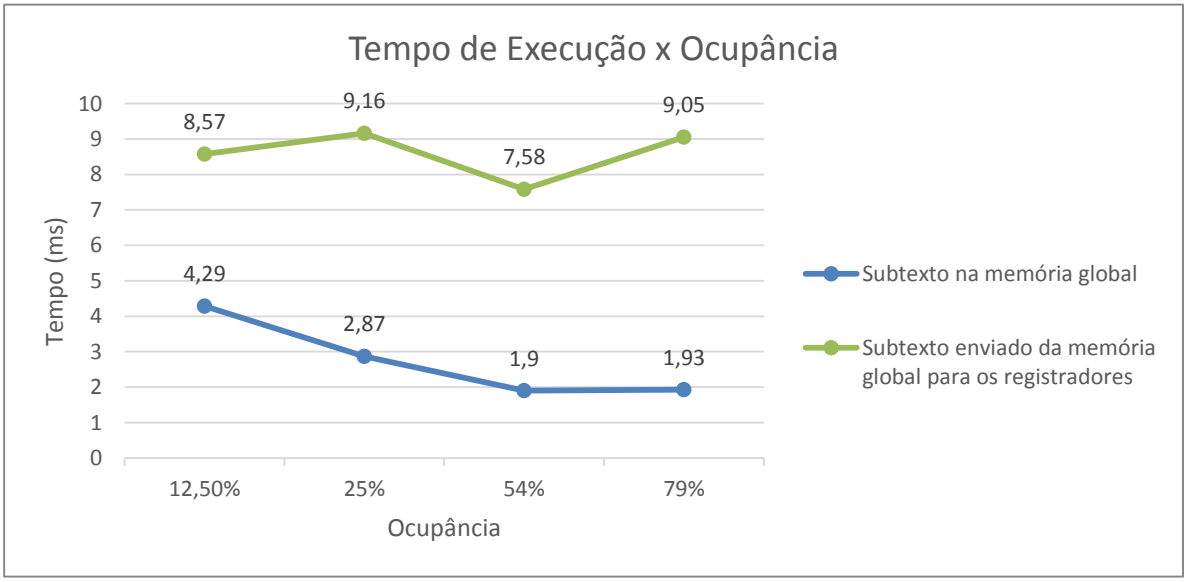


Figura 4.8: Resultados experimentais das abordagens do Aho-Corasick.

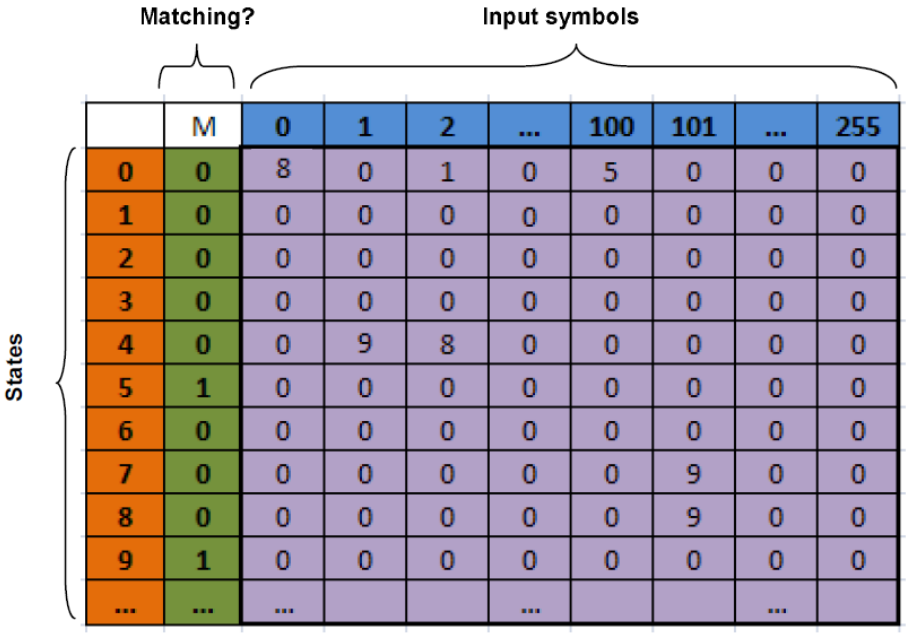
Como era de se esperar, o Aho-Corasick paralelo superou o Naive paralelo em relação ao tempo de execução da comparação dos padrões, tendo seu melhor resultado quando alcançou a ocupância de 54% na abordagem da memória global, executando a comparação dos padrões em um tempo de 1,90 ms com taxa de transferência de 21,56 Mcps. Enquanto

isso o melhor resultado do Naive foi na abordagem de memória compartilhada, com uma ocupância de 54%, com uma taxa de transferência de 7,20 Mcps.

Devemos observar com esses resultados iniciais que o Naive tenderá a ser mais lento à medida que o número de padrões crescerem, uma vez que, da forma como foi implementado, cada padrão adicionado resultará em mais repetições no *loop* onde ocorre a comparação de padrões. Já o Aho-Corasick tende a manter o tempo de execução inalterado mesmo com um número maior de padrões, como foi mostrado por TRAN, N. et al. (2012).

4.4. Proposta e uso de uma nova técnica de compactação da STT

A STT da Figura 2.14 foi modificada (conforme a Figura 4.9) para possibilitar a explicação de como ocorre a compactação de forma completa.



	M	0	1	2	...	100	101	...	255
0	0	8	0	1	0	5	0	0	0
1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	9	8	0	0	0	0	0
5	1	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	9	0	0
8	0	0	0	0	0	0	9	0	0
9	1	0	0	0	0	0	0	0	0
...	...	...			...			...	

Figura 4.9: Tabela de Transição de Estados (Adaptada de TRAN et al., 2013).

Em vez da tabela, passam a existir três vetores, conforme a descrição que segue juntamente com a Figura 4.10:

a) Vetor de Índices (VI): O tamanho de VI será igual à quantidade de estados, sendo que o índice *i* de VI corresponde ao estado *i* da máquina AC. O valor armazenado, ou seja, VI[*i*], significa o índice inicial no vetor VE correspondente ao estado *i*. 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.

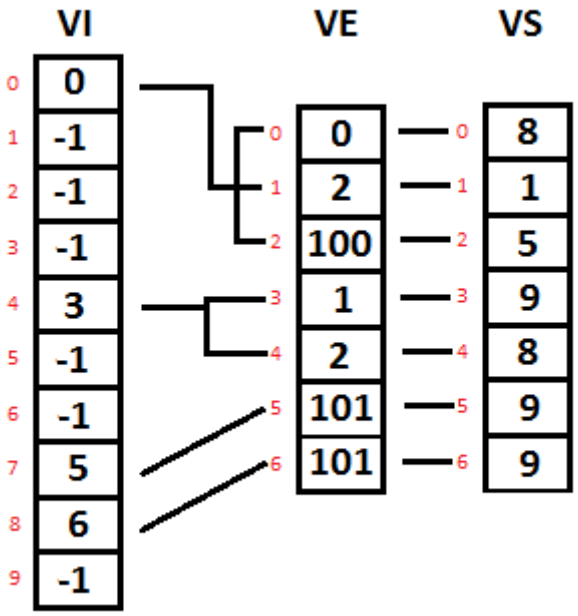


Figura 4.10: STT compactada.

Para exemplificar como a transição de estado ocorre nos vetores de compactação, suponha que entre na máquina AC o caractere correspondente ao valor inteiro 2 e o estado atual seja 4. VI[4] indica que os caracteres de entrada que podem levar a máquina a um estado válido se inicia 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 obtem-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 2 é diferente de 1. Avançando para o índice 4, encontra-se o caractere dois. 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 2 a entrada fosse 200, não haveria uma saída correspondente em VS, esse caso indica que ocorreu uma falha na função *goto* e o vetor de falhas deverá ser consultado para o estado 4.

Supondo que a tabela tivesse apenas esses 10 estados, 256 caracteres de entradas possíveis (conforme a tabela *ASCII*) e somente os valores de saída que aparecem na Figura 4.9, a compactação consegue reduzir de 2560 para 24 dados os quais devem ser armazenados em uma das memórias da GPU.

Porém, como o Snort possui milhares de regras, uma máquina mais próxima do real pode chegar a ter milhares de estados, fazendo com que a compactação seja ainda mais

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.

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

1) Declare a variável *contador_de_entradas* e atribua zero a essa variável;
2) Inicie o *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;

Fazendo uma breve comparação com a técnica de compactação bit-mapeamento apresentada na seção 2.4.1, para cada nó que tenha pelo menos um filho, existe a necessidade de se criar um bitmap de 256 bits no bit-mapeamento.

Os dados do gráfico da Figura 4.11 foram gerados usando-se a função *goto* criada a partir das regras *free* do Snort, as quais possuíam mais de 5.000 palavras-chave *content*. Ela mostra quantos nós filhos cada nó tem. A máquina de estado gerada teve um total de 48317 estados sendo que o estado com mais nós filhos foi o estado 0 que possui 88 nós filhos. Pode-se ver no gráfico da Figura 4.11 que mais de 1500 nós (precisamente 43686 nós) possuem apenas 1 nó filho, 960 nós possuem 2 filhos, 224 nós possuem 3 filhos, e assim sucessivamente.

Figura 4.11: Quantidade de nós filhos em uma máquina real.

Fazendo uma breve comparação com a técnica de compactação bitmapeamento apresentada na Seção 2.4.1, a compactação do bitmapeamento só será mais eficaz do que a compactação desse trabalho quando um nó tiver mais de 10 filhos, já que, nesse caso, a compactação desse trabalho necessitará de mais de 256 bits para representar todos os caracteres que levam o estado a um dos seus filhos em VE (8 bits para cada filho) juntamente com todos os seus filhos em VS (16 bits para cada filho), enquanto que no bitmapeamento cada nó terá um bitmap de 256 bits associado, independentemente da quantidade de filhos. A Figura 4.11 mostra que é mais comum um nó ter menos de 10 filhos, sendo que mais de 93% dos nós estão nessa situação. Por isso a compactação sugerida nesta dissertação de mestrado mostra-se mais eficaz em relação ao uso da memória, como pode ser visto na Tabela 4.1.

Tabela 4.1: Redução da quantidade de dados através da compactação.

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

A STT gerada obteve 48317 estados. A quantidade de KB para cada situação foi obtida através dos seguintes cálculos:

1) STT comum: (48317 x 256 x 16) / 8000 = 24738. Onde 256 é a quantidade de caracteres da tabela ASCII, 16 é o tamanho do short int em bits e a divisão por 8000 é para transformar de bits para KBytes.
2) Bitmapeamento: (((48317 - 3225)x256) + 48317x16) / 8000. O cálculo do lado esquerdo da soma é a quantidade de bits que os bitmaps consomem, onde 3225 é a

quantidade de nós que não tem filho (não necessitando de um bitmap), 256 é o tamanho do bitmap. A multiplicação do lado direito da soma é a quantidade de bits que cada nó consome, onde 16 é o tamanho do short int.

3) Nova compactação: Neste caso, todos os nós ocupam um short int de 16 bits. Além disso, cada transição válida ocupa um char de 8 bits, por exemplo, se a STT tem 960 nós que possuem apenas 2 filhos, esses filhos ocuparão 960x8x2 bits. O cálculo completo é dado da seguinte forma: ((48317x16) + ((43686x8) + (960x8x2) + (224x8x3) + (82x8x4) + (51x8x5) + (32x8x6) + (25x8x7) + (18x8x8) + (13x8x9) + (10x8x10) + (9x8x11) + (8x8x12) + (6x8x13) + (4x8x14) + (2x8x15) + (6x8x16) + (2x8x17) + (2x8x18) + (2x8x19) + (2x8x21) + (1x8x24) + (1x8x26))) / 8000.

O código utilizado para gerar a STT a partir das palavras-chaves do Snort pode ser visto no anexo B na linguagem C.

Os primeiros ensaios com a compactação foram realizados em um host online com placa GEFORCE GTX 770 de 1536 cores CUDA onde posteriormente este host ficou indisponível e os ensaios finais passaram a ser realizados nas placas TITANX e TESLA K20. O gráfico da Figura 4.12 mostrou que se o texto de entrada tivesse menos de 4MB seria mais eficiente usar a STT completa na memória de textura ou na global. A partir de 4MB o desempenho de ambas as abordagens começariam a se igualar e tornariam-se praticamente iguais. Se o texto de entrada tivesse um tamanho acima de 7MB a STT compactada na memória compartilhada mostrava ser mais eficiente em todos os ensaios na GTX 770. Portanto, se um texto fosse maior que 7 MB seria mais indicado usar a memória compartilhada.

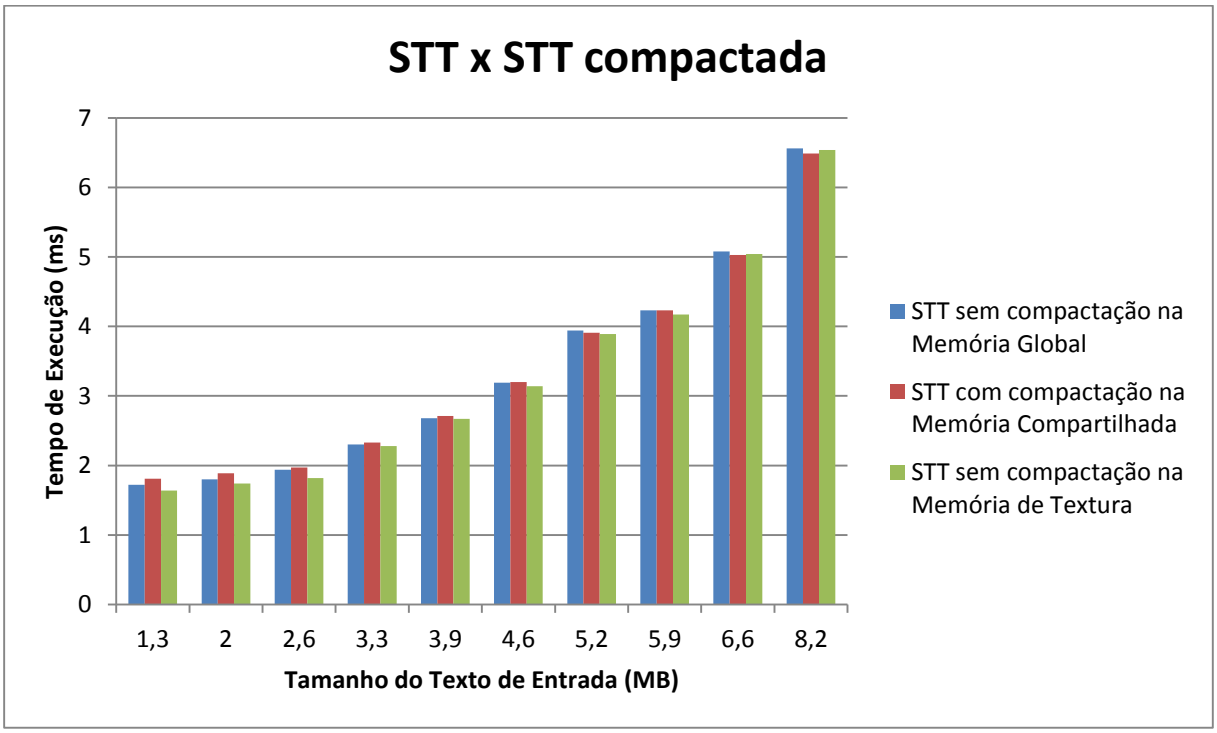


Figura 4.12: Curva de crescimento dos tempos de execução nas versões normal e compactada do Aho-Corasick paralelo.

Como 7 MB de texto era uma quantidade pequena de dados e que não era boa para avaliar o desempenho da GPU, textos de tamanhos maiores também foram testados. Nestes casos a STT compactada também superou a versão normal, como ocorreu quando o texto de entrada possuía 250MB e foi obtido 197 ms de tempo de execução, ao passo que a STT normal foi executada em 199 ms. Porém esse resultado não teve muito impacto positivo e algumas melhorias foram feitas no algoritmo compactado. Alguns *if's* desnecessários foram retirados. O VE que era declarado como um vetor de números inteiros passou a ser um vetor de short int. Mas a maior mudança esteve relacionada à placa GPU utilizada, pois esta teve uma grande influência no ganho da versão compactada. Ao deixar de usar placas menos potentes em termos cores CUDA e mais antigas como as placas GT210 e GTX 770 e passar a usar placas mais pontentes como a TITANX e TESLA K20 os ganhos também aumentaram.

5. Resultados GPU x CPU

As subseções seguintes têm por finalidade apresentar os resultados obtidos nos experimentos controlados.

Basicamente foram usados dois tipos de textos de entrada nos experimentos para simular os pacotes de redes colhidos pelo Snort durante seu processamento. Um texto sintético (pacotes sintéticos) que é composto por 1000 caracteres de um texto em inglês, o qual pode ser visto no Anexo D. 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.

Já o outro texto (pacotes reais) foi formado pelos caracteres dos pacotes obtidos na rede da Universidade Federal de Sergipe, com o auxílio do software Wireshark. O arquivo com os pacotes superou o tamanho de 100 MB, porém apenas parte desse arquivo foi utilizado já que as GPUs eram acessadas remotamente e ficava inviável enviar um arquivo desse tamanho toda vez que precisasse fazer modificação nele. A parte utilizada desse arquivo possuía 1000000 de caracteres, tendo um 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*.

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.

Além disso, quatro abordagens foram adotadas para fazer estes ensaios finais:

1) Memória Global: Nesta abordagem os dados do texto de entrada e a máquina AC foram colocados na memória global da GPU;
2) Memória de Textura: Os dados do texto de entrada foram colocados na memória global e a máquina AC foi colocada na memória de textura da GPU;

3) Memória Compartilhada: Os dados do texto de entrada foram colocados na memória global e a máquina AC foi colocada na memória compartilhada da GPU utilizando a nova compactação criada neste trabalho;
4) Serial: Versão serial idêntica à versão de memória global, porém para ser executada na CPU.

5.1. Análise com Pacotes Sintéticos

5.1.1. TESTE 1 – Bloco de Tamanho 1024

Inicialmente foi verificado que o tamanho do grid e do bloco podem influenciar no tempo de execução dos algoritmos paralelos, sendo que o passo inicial foi fixar a melhor configuração desses dois componentes dentre alguns valores aleatórios.

Um texto sintético de aproximadamente 492 KB foi utilizado na execução do algoritmo. Os tamanhos dos blocos foram variados com valores menores ou igual a 1024, já que o tamanho máximo do bloco é 1024 para as placas de testes utilizadas neste trabalho. Os tamanhos dos grids foram variações proporcionais aos tamanhos dos blocos.

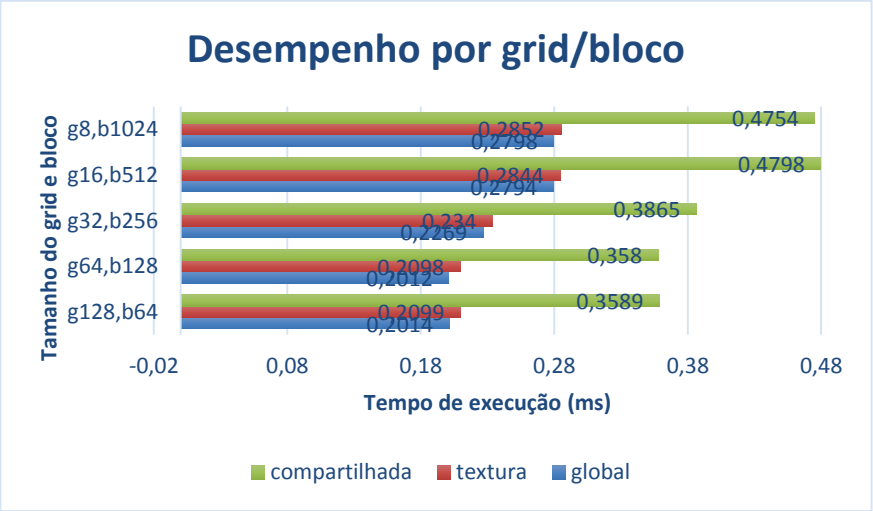


Figura 5.1: Variação do tamanho do grid e bloco da GPU.

Pode-se ver na Figura 5.1 que o menor tempo de execução foi obtido quando o grid foi dividido em 64 blocos e o bloco foi dividido em 128 threads.

Após a fixação da melhor configuração de grids e blocos, o tamanho do texto de entrada foi variado a fim de ver os comportamentos dos algoritmos. Para medir o tempo de execução nas versões paralelas, o tempo de execução do algoritmo foi somado ao tempo que a GPU levou para transferir os dados do *host* para a GPU e ao tempo de transferência dos dados da GPU para o *host*. Os resultados podem ser vistos na Figura 5.2.

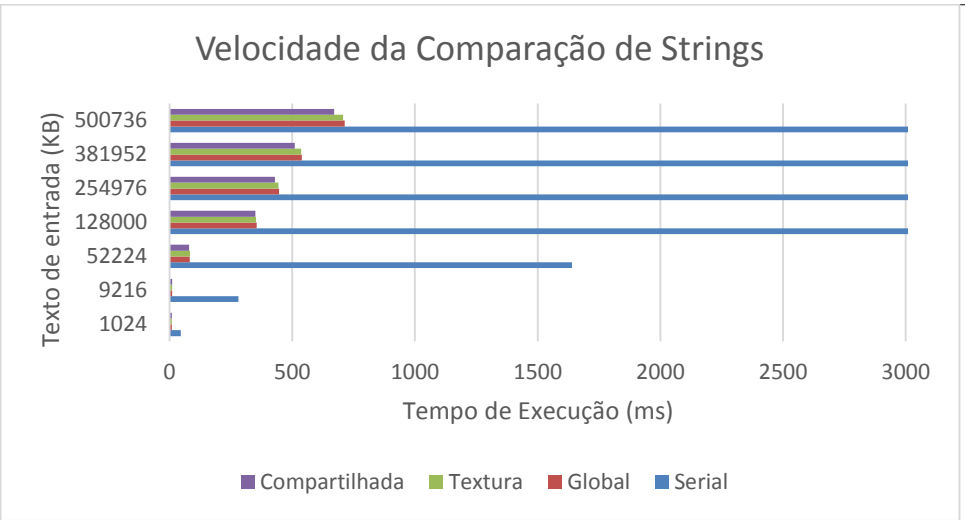


Figura 5.2: Tempo de execução das quarto abordagens.

Pelo gráfico da Figura 5.2 nota-se que quanto maior for o texto de entrada maior será o ganho da memória compartilhada em relação às demais abordagens. O ganho começa a aparecer entre 52 e 128 MB e a partir daí vai aumentando. Com 500 MB a memória compartilhada obteve ganhos de desempenho de 5,46% e 6,46% em relação às versões de memória de textura e memória global, respectivamente. Além disso, a versão compactada foi 22 vezes mais rápida que a versão serial.

Tabela 5.1: Taxa de transferência (Mcps).

Tamanho Entrada/Abordagem	1 MB	9,2 MB	52 MB	128 MB	255 MB	382 MB	500 MB
Serial	21,74	32,74	31,71	35,93	34,21	32,82	32,89
Global	500,00	2839,51	1416,51	1479,94	1442,06	1417,65	1428,41
Textura	537,63	2893,08	1414,20	1487,16	1440,68	1416,55	1428,16
Compartilhada	502,51	2839,51	1510,31	1532,57	1564,90	1565,45	1575,30

5.1.2. TESTE 2 – Blocos de Tamanho 100 e 1000

Neste teste, ao invés de se usar um texto de entrada com tamanho múltiplo de 1024, foi utilizado um texto múltiplo de 1000 para que a GPU trabalhasse em cima de um texto mais simétrico. Com isso os resultados melhoraram em relação ao TESTE 1.

Um texto sintético de 1 GB foi utilizado na execução do algoritmo. Os tamanhos dos blocos foram variados com valores menores ou iguais a 1000, já que o tamanho máximo do bloco é 1024 para as placas de testes utilizadas neste trabalho. Os tamanhos dos grids foram variações proporcionais aos tamanhos dos blocos desde que a multiplicação do tamanho do grid com o tamanho do bloco fosse exatamente 1000000, pois essa é uma das formas para o algoritmo executar corretamente, ou seja, encontrar o número correto de padrões.

Na placa Tesla K20 o bloco foi inicialmente fixado em 10 e o grid em 100000. Esse

teste foi executado em menos de 1 ms, porém a quantidade de padrões encontrados se mostrava instável, onde às vezes se encontrava a quantidade correta de padrões de ataques e outras vezes a quantidade era negativa. Por este motivo o valor de bloco igual a 10 foi descartado. Com o bloco fixo em 100 e o grid em 10000 foi obtido o menor tempo de execução, sendo que a versão compartilhada foi executada em apenas 539 ms, enquanto que as versões de textura e global foram executadas em 680 e 685 ms, respectivamente. Para um bloco de 1000 e um grid de 1000, as versões compartilhada, global e textura foram executadas em 671, 681 e 686 ms. Por apresentar o menor tempo de execução, a configuração de bloco igual a 100 foi a escolhida.

Após a fixação do bloco em 100, o tamanho do texto de entrada foi variado a fim de ver o comportamento dos algoritmos. Para medir o tempo de execução nas versões paralelas, o tempo de execução do algoritmo foi somado ao tempo que a GPU levou para transferir os dados do *host* para a GPU e ao tempo de transferência dos dados da GPU para o *host*. Os resultados podem ser vistos na Figura 5.3.

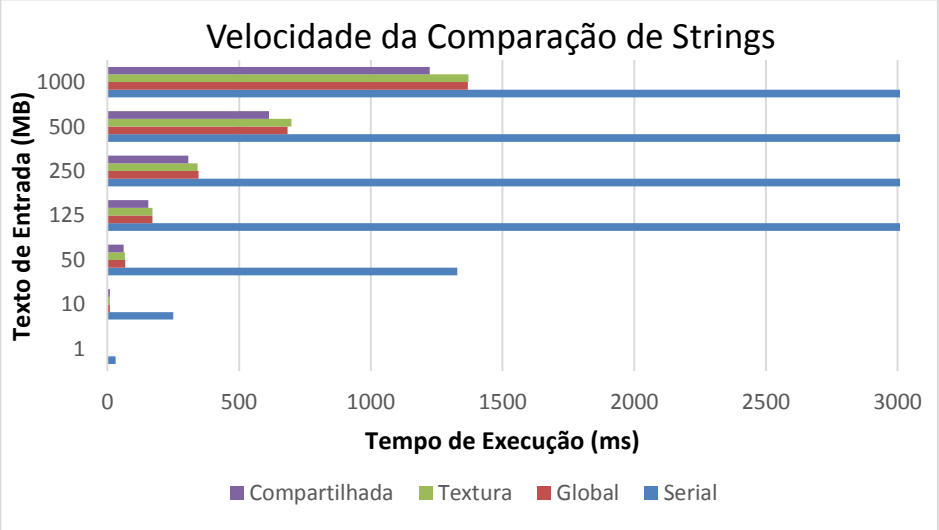


Figura 5.3: Tempo de execução das quatro metodologias na placa Tesla K20.

Os tempos de execução em milissegundos da Figura 5.3, as taxas de tranferências e o percentual de consumo da GPU por parte do kernel podem ser vistos na Tabela 5.2.

Tabela 5.2: Resultados experimentais com dados sintéticos - 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

Pode-se ver que a GPU executou de forma mais rápida que a CPU em todos os ensaios. Nota-se também que entre 10 MB e 50 MB de texto de entrada a abordagem de memória compartilhada já consegue superar as outras duas versões paralelas. O gráfico da Figura 5.4 mostra 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) foi igual 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 próximo a 15% em relação às outras duas abordagens.

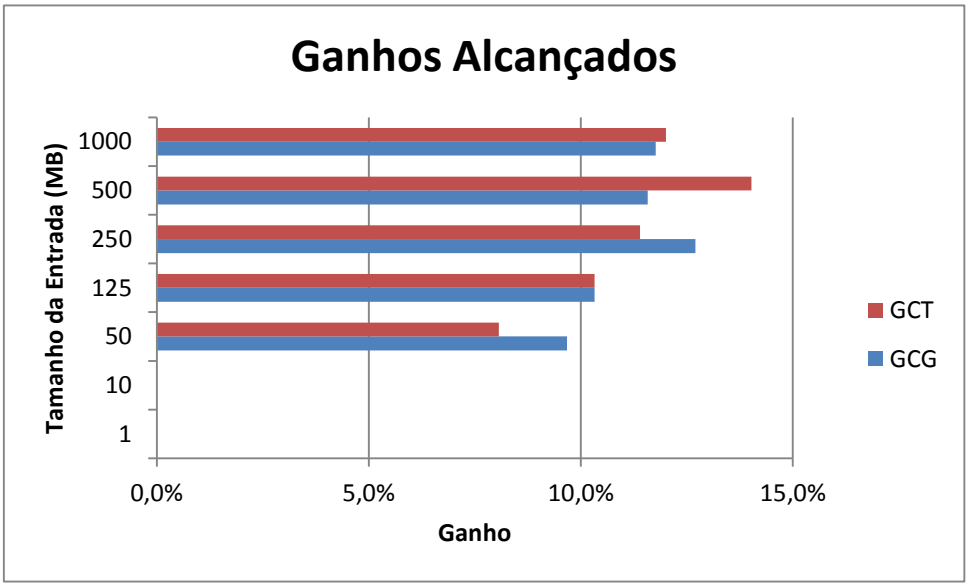


Figura 5.4: Ganhos obtidos com a abordagem compartilhada utilizando dados sintéticos - Tesla K20.

Já na placa TITANX o desempenho da abordagem compactada se mostrou ainda mais eficiente. Nesta placa o algoritmo que apresentou o melhor desempenho com um texto de entrada de 1GB foi a versão compactada com um bloco de 1000 e um grid de 1000.

Partindo do bloco igual a 1000, o gráfico abaixo foi obtido variando-se o tamanho da entrada e o tamanho do grid de forma proporcional ao tamanho da entrada.

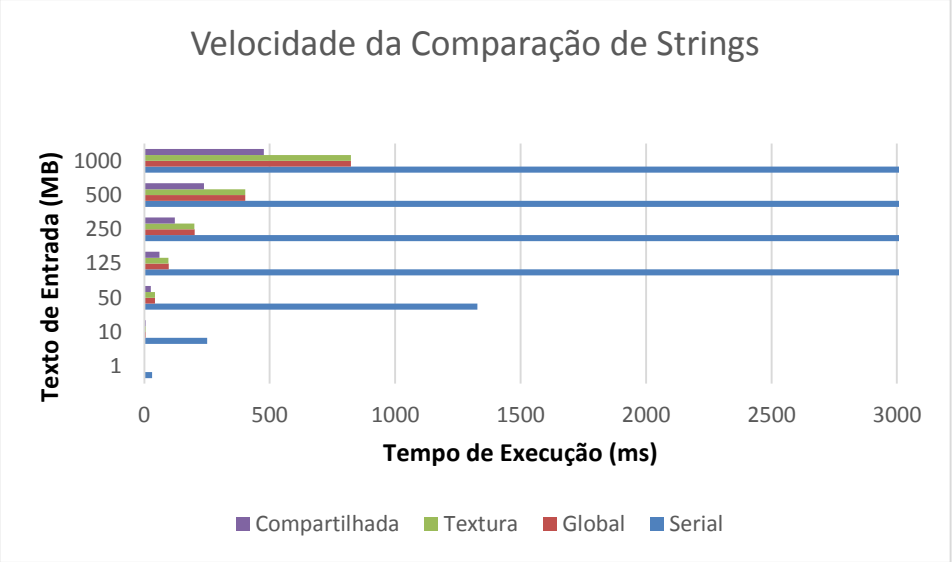


Figura 5.5: Tempo de execução das quatro metodologias na placa Titanx.

Diferentemente da placa Tesla K20, na placa Titanx 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, como pode ser visto na Figura 5.6. Além disso, de acordo com a Tabela 5.3, a versão compactada foi 56 vezes mais rápida que a versão serial em termos de Tempo de Execução quando o tamanho do texto de entrada foi 1 GB.

Tabela 5.3: Resultados experimentais 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

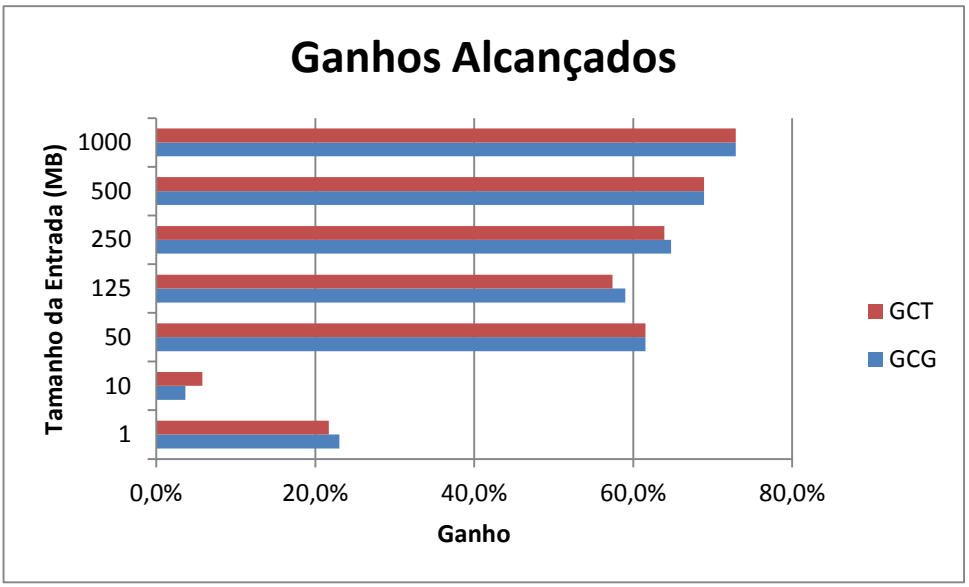


Figura 5.6: Ganhos obtidos com a abordagem compartilhada utilizando dados sintéticos - Titanx.

5.2. Análise com Pacotes Reais

Nestes ensaios pacotes de redes obtidos através da rede da UFS foram utilizados como o texto de entrada. A configuração do grid e do bloco utilizada nos testes sintéticos foi também utilizada nestes testes com pacotes reais.

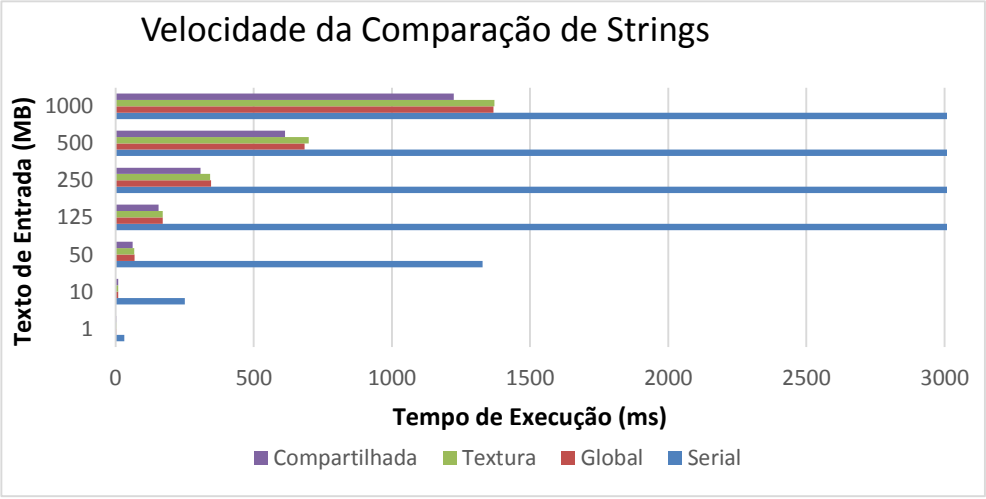


Figura 5.7: Tempo de execução das quatro metodologias na placa Tesla K20 – dados reais.

Tabela 5.4: Resultados experimentais 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

Pode-se ver pela Tabela 5.4 que a GPU executou de forma mais rápida que a CPU em todos os ensaios. Nota-se também que a versão de memória compartilhada foi mais rápida que a versão de memória global em todos os ensaios e mais rápida que a versão de memória de textura quando o texto de entrada teve tamanho entre 200 e 500 MB, como pode ser visto no gráfico da Figura 5.8.

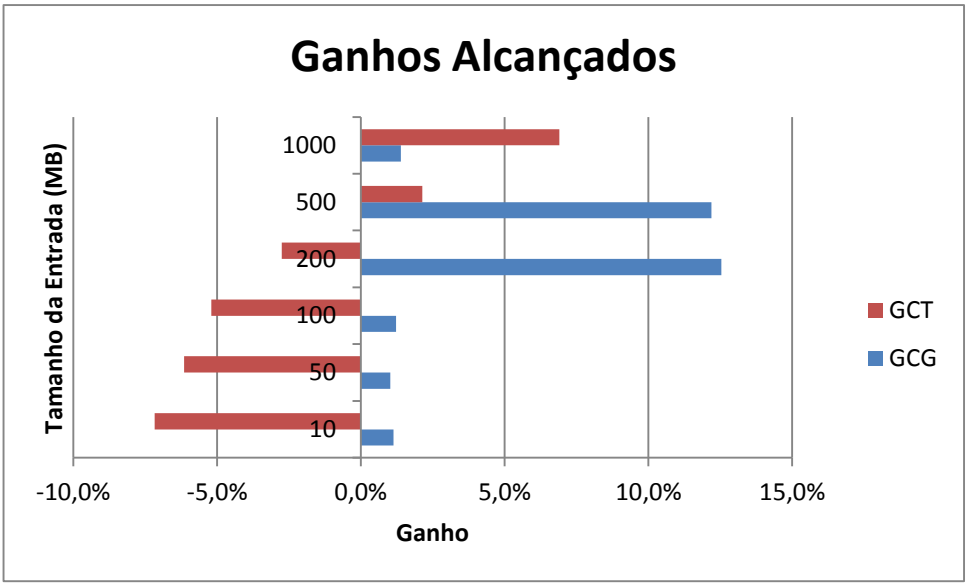


Figura 5.8: Ganhos obtidos com a abordagem compartilhada utilizando dados reais - Tesla K20.

Diferentemente da placa Tesla K20, na placa Titanx o ganho da versão compartilhada em relação às outras abordagens já começa a aparecer a partir de 10 MB e a partir daí vai aumentando. Com 500 MB de texto de entrada, a memória compartilhada obteve um ganho de desempenho de 12,1% em relação à memória de textura e 14,9% em relação à memória global, como pode ser visto na Figura 5.10. Além disso, de acordo com a Tabela 5.5, a versão compactada foi 48 vezes mais rápida que a versão serial quando o tamanho do texto de entrada foi 500 MB.

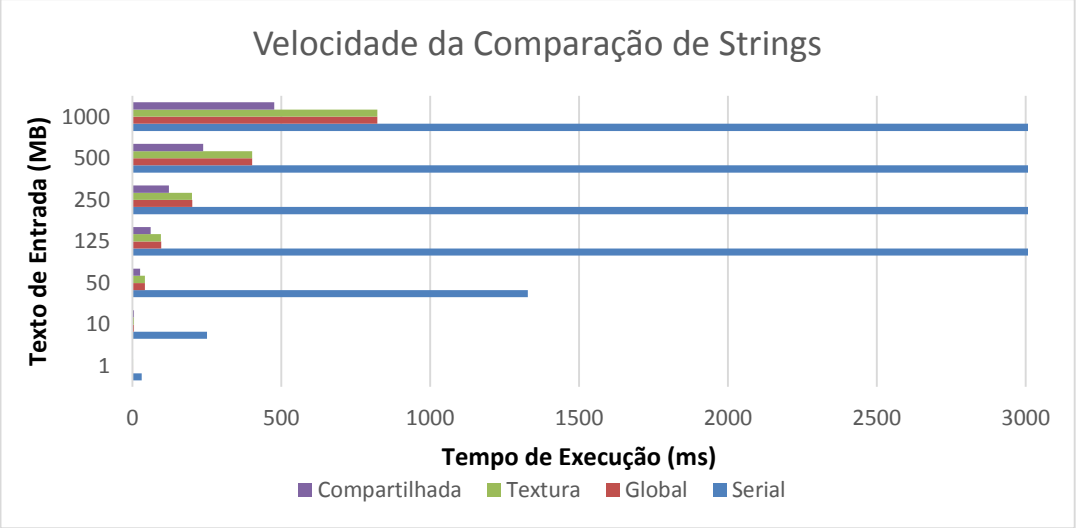


Figura 5.9: Tempo de execução das quatro metodologias na placa Titanx – dados reais.

Tabela 5.5: Resultados experimentais 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

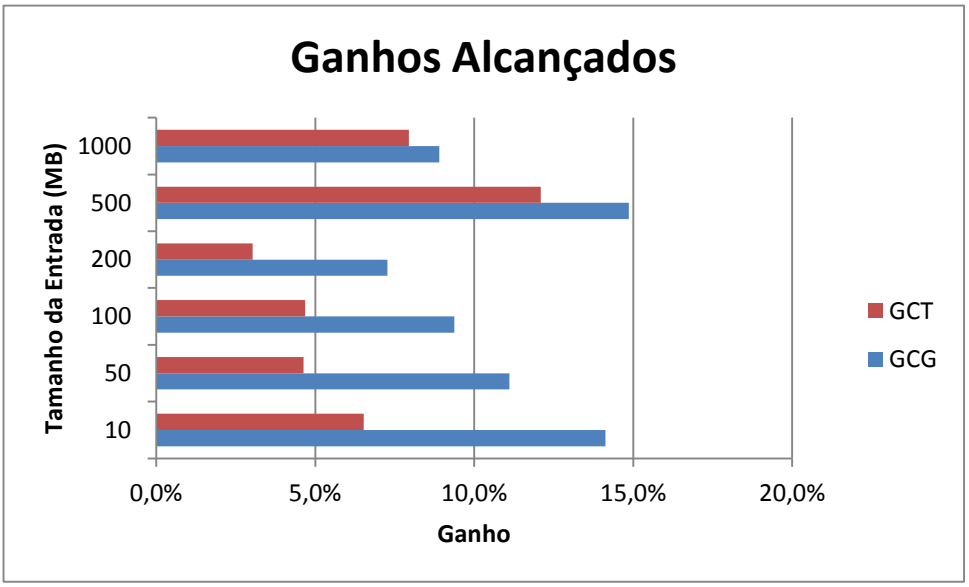


Figura 5.10: Ganhos obtidos com a abordagem compartilhada utilizando dados reais - Titanx.

Fazendo uma breve análise dos resultados obtidos, nota-se que os ganhos obtidos devido ao fato da STT ter sido levada para a memória compartilhada foram satisfatórios,

uma vez que foram alcançados *speedups* maiores do que quando a STT é implementada em outras memórias da GPU, principalmente quando o texto de entrada tem tamanho superior a 200 MB. Mostramos também a importância de não só se variar o uso das memórias, mas também a configuração do grid e do bloco (por conseguinte a ocupância), visto que o mesmo algoritmo saiu de uma execução de 671 ms para 539 ms só pelo fato de ter saído de um bloco e grid iguais a 1000 para um bloco igual a 100 e um grid igual a 10000, como ocorreu na placa TESLA K20 com dados sintéticos. Porém deve ser observado também que devido ao tamanho do *warp* ser de 32 threads há uma tendência que configurações múltiplas de 32 possam funcionar melhor do que estas configurações múltiplas de 100.

Com estes resultados observa-se também que a CPU não foi mais rápida que a GPU mesmo com um texto de entrada inferior a 40 MB como tinha sido sugerido por THAMBAWITA, D. et al. (2014). Notou-se que é difícil determinar um ponto onde a GPU supere a CPU. Isso depende muito dos hardwares que se está utilizando, da forma que os algoritmos foram implementados, das configurações de grids e blocos das GPUs, entre outros fatores.

6. Conclusão

Esta dissertação de mestrado abordou a paralelização de algoritmos de comparação de *strings*, uma solução que tem se mostrado bastante eficiente quando aplicada em softwares que necessitam processar uma grande quantidade de dados como é o caso dos IDSs.

Foi mostrado que o processamento de dados em softwares de segurança da informação como o IDS Snort utilizando os algoritmos de comparação de *strings* seriais não é feito com velocidade suficiente para não haver perda de dados ou sobrecarga da CPU. E para piorar a situação, a quantidade de ataques de redes e a velocidade dos *links* de internet aumentam cada vez mais, aumentando, assim, a quantidade de dados a serem processados.

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, os algoritmos Força-Bruta e Aho-Corasick foram paralelizados a fim de se obter uma análise de desempenho comparativa entre os algoritmos seriais e paralelos.

Foram extraídos 18 ataques das regras *free* do Snort a fim de construir a STT do algoritmo Aho-Corasick, o qual mostrou ser mais rápido que o algoritmo Força Bruta. A STT gerada teve tamanho maior que a capacidade total da memória compartilhada e por isso houve a necessidade de compactá-la. A nova compactação criada neste trabalho retira todos os valores desnecessários da STT, possibilitando que ela seja armazenada na memória compartilhada da GPU.

Os testes foram feitos em duas GPUs e uma CPU, utilizando as memórias global, de textura ou compartilhada das GPUs em cada abordagem para ver qual das memórias executava a comparação de *strings* de forma mais rápida. Além disso, foram utilizados dois tipos de dados. No primeiro tipo, dados sintéticos, os dados eram mais uniformes, situação positiva para a GPU devido à facilidade de sincronismo das *threads*, tornando o processamento das abordagens mais veloz. Com este tipo de dado, as duas placas tiveram ganhos de velocidades consideráveis podendo chegar a um ganho de 73% com a placa TITANX na abordagem de memória compartilhada (que usa a compactação) se comparada com as abordagens que não usam a memória compartilhada. Se comparado com a CPU, o ganho da versão compartilhada é maior ainda, podendo ser 56 vezes mais rápido. No segundo tipo, dados reais, os dados tinham uma aleatoriedade maior visto que foram colhidos a partir dos pacotes de uma rede real. Nele, as abordagens tiveram ganhos menores 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.

Sendo assim, considerando os hardwares utilizados neste trabalho e sabendo que o componente de Detecção gasta aproximadamente 76% de todo o tempo de execução do Snort, se, por exemplo, o Snort capturar pacotes por 100 minutos, ele levará 76 minutos para buscar os padrões nesses pacotes de forma serial. Paralelizando, ele terá uma redução de 48 vezes, ou seja, levará pouco mais de 1 minuto para analisar os pacotes de forma completa. Então, pode-se concluir através da base teórica, que mostrou o esforço da comunidade de computação de alto para paralelizar o algoritmo AC presente no Snort, e através dos experimentos realizados neste trabalho, que a paralelização do módulo de detecção do IDS poderá ocasionar um ganho de desempenho que viabilizará seu uso em redes de alta velocidade.

Além de ter conseguido obter resultados positivos nos ensaios realizados com a versão compactada, outras observações feitas neste trabalho também são de grande valor, como o impacto causado pela variação da configuração do grid e bloco e a possível necessidade de mudança dessa configuração de acordo com a placa utilizada. Além disso, a compactação criada neste trabalho não fica restrita ao problema do Aho-Corasick. Qualquer problema que precise diminuir a quantidade de dados, sendo que existe uma grande repetição de dados (semelhante aos zeros da STT) pode ser resolvido com esta compactação. Por fim, notou-se que esta abordagem de memória compartilhada pode ser uma tendência para um futuro próximo já que as pesquisas até o momento não tem adotado essa metodologia, mas as placas atuais mostraram que ela funciona melhor que as metodologias antigas.

A compactação criada neste trabalho mostrou-se bastante eficiente, porém pode melhorar ainda mais. O algoritmo que busca os padrões no Aho-Corasick usando a STT compactada possui um *loop* de ordem N para cada estado da STT e alguns *if*'s, comandos que dessincronizam as *threads* da GPU e tornam o processamento mais lento. Como trabalho futuro, pode-se medir o efeito das partes do código do kernel no Aho-Corasick compactado através de medidores inseridos no próprio código-fonte e coletar informações sobre indireções. Posteriormente, pode-se tentar reduzir a ordem de complexidade do *loop* e retirar alguns *if*'s que por ventura sejam desnecessários. Além disso, pode-se retirar os conflitos de banco na abordagem de memória compartilhada para aumentar o desempenho. Outra sugestão de trabalho futuro é fazer mais testes variando a configuração de grids e blocos, pois esta variação mostrou ter bastante influência nos tempos de execuções obtidos.

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. Efficient string matching: an aid to bibliographic search, Communications of the ACM, v.18 n.6, p.333-340, Jun. 1975.

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.

CHENG, J.; GROSSMAN, M.; MCKERCHER, T. Professional CUDA C Programming. Wrox, Sep. 2014. 528 p.

DAN, G. Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, 1997. 534 p.

HARRIS, M. CUDA Pro Tip: Occupancy API Simplifies Launch Configuration. Disponível em: < http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-occupancy-api-simplifies-launch-configuration >. Acesso em: 15/12/2015.

HUANG, N.; HUNG, H.; LAI, S. et al. A GPU-Based Multiple-Pattern Matching Algorithm for Network Intrusion Detection Systems. The 22nd International Conference on Advanced Information Networking and Applications, pp 62-67, Mar. 2008.

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

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 >

JIANG, H.; SALAMATIAN, K.; MATHY, L. Scalable high-performance parallel design for Network Intrusion Detection Systems on many-core processors. The Symposium on Architectures for Networking and Communications Systems, pp 137-146, Oct. 2013.

JÚNIOR, J. B. S.; ORDONEZ, E. D. M.; NUNES, M. S. N. Acelerando a comparação de strings do IDS Snort através da programação paralela: um mapeamento sistemático da literatura sobre o algoritmo Aho-Corasick paralelizado. Revista de Sistemas de Informação da FSMA. 2016.

JUNTA, P. H. G.; PEREIRA, V. F. IDS SNORT: Análise de desempenho no Windows, Linux e IDE Nvidia. São Cristóvão, Universidade Federal de Sergipe. 2015.

KOUZINOPOULOS, C. S.; MARGARITIS, K. G. String Matching on a multicore GPU using CUDA. The 13th Panhellenic Conference on Informatics, pp 14-18, Sep. 2008.

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.

LIN, J.; ADJEROH, D.; JIANG, Y. A Faster Quick Search Algorithm. Algorithms 2014, 7, pp 253-275, Jun. 2014.

OKE, P.; VAIDYA, A. Optimization of Parallel Aho-Corasick Multipattern Matching Algorithm on GPU. International Journal of Innovative Research in Computer and Communication Engineering. Vol. 3, pp 5191-5200, Jun. 2015. Acessado em: 21/10/2015. Disponível em: < http://www.ijircce.com/upload/2015/june/34_Optimization.pdf >

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

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.

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.

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.

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.

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.

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.

WALTON, J. NVIDIA's GeForce 700M Family: Full Details and Specs. Disponível em: < http://www.anandtech.com/show/6873/nvidias-geforce-700m-family-full-details-and-specs/2 >. Acesso em: 22/01/2016.

ANEXO A

```
__device__ int gotoAC(int est_atual, char caractere) //estarei considerando estado como um simples int
{
        int est_proximo=0;
        switch(est_atual) {
                case est0: if (caractere == 'a') est_proximo = est27; else if (caractere == 'h') est_proximo = est9; • • •
                case est1: if (caractere == 'r') est_proximo = est2; else est_proximo = falha; break;
                case est2: if (caractere == 'a') est_proximo = est3; else est_proximo = falha; break;
:                                                                                                  :
                case est29: est_proximo = falha; break;
                case est30: if (caractere == 'e') est_proximo = est31; else est_proximo = falha; break;
                case est31: est_proximo = falha; break;
        }
  return est_proximo; //retorna o próximo estado
}
```

Figura 6.1: Trecho do código da função goto.

```
__device__ int failAC(int est_atual)
{
        int est_proximo=0;
        switch(est_atual) {
                case est1: est_proximo = est0; break;
                case est2: est_proximo = est0; break;
:                                              :
                case est30: est_proximo = est9; break;
                case est31: est_proximo = est0; break;
        }
  return est_proximo; //retorna o próximo estado
}
```

Figura 6.2: Trecho do código da função de falha.

```
__device__ char* outputAC(int est_atual)
{
        char* palavra = "";

        switch(est_atual) {
                case est0: palavra = ""; break;
                case est1: palavra = ""; break;
:                                        :
                case est31: palavra = "the"; break;
        }
  return palavra;
}
```

Figura 6.3: Trecho do código da função output.

ANEXO B – Códigos-Fontes

Todos os códigos-fontes desenvolvidos no trabalho estão disponíveis no link abaixo:
https://gist.github.com/anonymous/aec7d8dffdb3afd540d8675ab921d201

ANEXO C – Texto utilizado para buscar padrões

Brazil is the largest country in South America and the fifth largest nation in the world. It forms an enormous triangle on the eastern side of the continent with a 4,500-mile (7,400-kilometer) coastline along the Atlantic Ocean. It has borders with every South American country except Chile and Ecuador. The Brazilian landscape is very varied. It is most well known for its dense forests, including the Amazon, the world's largest jungle, in the north. But there are also dry grasslands (called pampas), rugged hills, pine forests, sprawling wetlands, immense plateaus, and a long coastal plain. Northern Brazil is dominated by the Amazon River and the jungles that surround it. The Amazon is not one river but a network of many hundreds of waterways. Its total length stretches 4,250 miles (6,840 kilometers), making it the longest river on Earth. Thousands of species live in the river, including the infamous piranha and the boto, or pink river dolphin. Northern Brazil is dominated by the Amazon River and the jungles that surround it. The Amazon is not one river but a network of many hundreds of waterways. Its total length stretches 4,250 miles (6,840 kilometers), making it the longest river on Earth. Thousands of species live in the river, including the infamous piranha and the boto, or pink river dolphin. Brazil has the greatest variety of animals of any country in the world. It is home to 600 mammal species, 1,500 fish species, 1,600 bird species, and an amazing 100,000 different types of insects. Brazil's jungles are home to most of its animal life, but many unique species also live in the pampas and semidesert regions. In the central-western part of Brazil sits a flat, swampy area called the Pantanal. This patchwork of flooded lagoons and small islands is the world's largest wetland. Here live giant anacondas, huge guinea pig relatives called capybaras, and fierce South American alligators called caimans. For thousands of years, people have been exploiting the jungles of Brazil. But since Europeans arrived about five centuries ago, forest destruction has been rampant. Most of Brazil's Atlantic rain forest is now gone, and huge tracts of the Amazon are disappearing every year. The government has established many national parks and refuges, but they only cover about 7 percent of the country. Most Brazilians are descended from three ethnic groups: Amerindians, European settlers (mainly from Portugal), and Africans. Starting in the 19th century, waves of immigrants from Europe, the Middle East, and even Japan added to this mix. This diversity of cultures has created a rich religious, musical, and culinary culture. Brazilians are soccer crazy, and their country has produced some of the best players. The most famous of all is Edson Arantes do Nascimento, better known as Pelé. Brazil has won the World Cup soccer finals five times, more than any other nation, and is hosting the tournament this year. Brazil is a federal republic with a president, a National Congress, and a judiciary. From 1888 until recently, the country struggled with democracy. But in 1985, the military government was peacefully removed, and by 1995, Brazil's politics and economy had become fairly stable. Brazil has many different soils and climates, so it can produce a great variety of crops. Its agricultural exports include sugarcane, latex, coffee, cocoa beans, cotton, soybeans, rice, and tropical fruits. Brazil is also South America's most industrial nation, producing chemicals, steel, aircraft, and cars. Until recently, scientists thought Brazil was first settled by Asians about 10,000 years ago. But new evidence shows there were people living there at least 32,000 years ago. Some experts think they may have arrived from islands in the Pacific Ocean. Brazil was added to the map of the world during the great European explorations in the late 15th century led by Portugal and Spain. When Europeans first reached the coast of Brazil, the country was home to about 30 million indigenous people, or Amerindians. Today, only about 300,000 remain, living primarily in Bra.

ANEXO D

Amanda is USERLOGINst cybercopin Authentication unsuccessful/cybercopnpass -iss@issworld.Login failed for user 'sa'ngle on the eastpass -cklause continent pass -saint0-mile (7,400-kilometer) coastline along the Atlantic Ocean. It has borders with every South American country except Chile and Ecuador. The Brazilian landscape is very varied. It is most well known for its dense forests, including the Amazon, the world's largest jungle, in the north. But there are also dry grasslands (called pampas), rugged hills, pine forests, sprawling wetlands, immense plateaus, and a long coastal plain. Northern Brazil is dominated by the Amazon River and the jungles that surround it. The Amazon is not one river but a network of many hundreds of waterways. Its total length stretches 4,250 miles (6,840 kilometers), making it the longest river on Earth. Thousands of species live in the river, including the infamous piranha and the boto, or pink river dolphin. Northern Brazil is dominated by the Amazon Ri