Compactação do Algoritmo de Comparação de Strings do Snort

Implementação CUDA do Algoritmo Aho-Corasick

Thiago Carvalho

30 de Novembro de 2025

Sumário

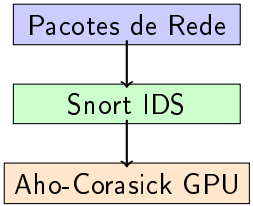
Contexto e Motivação

O Problema:

- Snort IDS precisa comparar pacotes de rede com milhares de padrões
- Comparação de strings é o gargalo (70% do tempo)
- Redes modernas: 10-100 Gbps

A Solução:

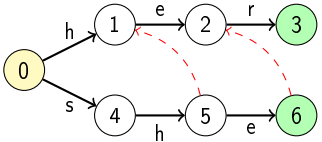
- Algoritmo Aho-Corasick (busca múltiplos padrões em O(n+m))
- Aceleração via GPU (CUDA)
- Compactação da tabela de estados (STT)



Algoritmo Aho-Corasick

Estruturas principais:

1. **Função GOTO**: Transições entre estados
2. **Função FAILURE**: Fallback quando não há transição
3. **Função OUTPUT**: Padrões encontrados em cada estado



Exemplo: padrões "her"e "she"
Linhas pontilhadas = failure links

Complexidade:

- Construção: $O(m)$ onde m = soma dos padrões
- Busca: $O(n + z)$ onde n = texto, z = matches

Problema: State Transition Table (STT)

STT Tradicional:

- Matriz $N_{estados} \times 256$ (todos os caracteres ASCII)
- A maioria das entradas é -1 (sem transição)
- Para 2830 estados: $2830 \times 256 \times 4$ bytes = **2.9 MB**

Estado	a	b	c	...	h	...	z	...
0	-1	-1	-1	...	1	...	-1	...
1	-1	-1	-1	...	-1	...	-1	...
...	...	...	...	...	...	...	...	...

→ 99% das entradas são -1 (desperdício de memória!)

Solução: STT Compactada

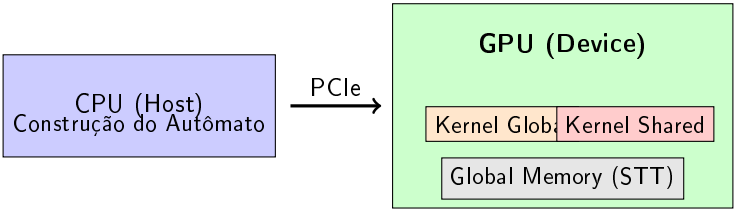
Estrutura compactada com 5 vetores:

- **VI** (Vetor de Índices): Início das entradas de cada estado
- **NE** (Número de Entradas): Quantidade de transições por estado
- **VE** (Vetor de Entradas): Caracteres de entrada
- **VS** (Vetor de Estados): Estados destino
- **est0**: Lookup direto para estado 0

Resultado:

Métrica	Valor
Estados	2830
Entradas válidas	2829
STT Original	2830 KB
STT Compactada	36 KB
Compressão	**98.7%**

Arquitetura da Solução



Dois kernels implementados:

1. **GPU Global**: STT compactada em memória global
2. **GPU Shared Híbrido**: Cache de est0/failure/output em shared memory

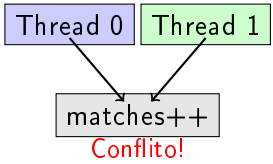
Condições de Corrida

Problema: Múltiplas threads podem encontrar o mesmo padrão simultaneamente

Pontos de Condição de Corrida:

1. **Contagem de matches**: Threads incrementando contador global
2. **Bordas de segmentos**: Padrões que cruzam limites entre threads
3. **Acesso à STT**: Leitura concorrente (não é problema - read-only)

Exemplo de Race Condition:



Tratamento das Condições de Corrida

Solução 1: Acumulação Local + atomicAdd

- Cada thread mantém contador LOCAL de matches
- Apenas no final: `atomicAdd(&total_matches, local_count)`
- Reduz contenção de 1 atomic/match para 1 atomic/thread

Solução 2: Técnica de Overlap

- Cada thread processa 64 caracteres EXTRAS antes do seu segmento
- Só conta matches DENTRO do seu segmento real
- Evita duplicação e perda de matches nas bordas

Solução 3: STT Read-Only

- Autômato construído na CPU, copiado para GPU
- GPU apenas lê a STT (sem escrita = sem race condition)
- Pode usar `__ldg()` para cache otimizado

Kernel de Memória Global

Características:

- Block size: 128 threads
- Grid size: 512 blocos
- STT completa em memória global
- Aproveita cache L2 da GPU (32 MB na RTX 4060 Ti)

Divisão do trabalho:

- Cada thread processa um segmento do texto
- Overlap de 64 caracteres entre segmentos (para padrões nas bordas)
- Acumulação local + atomicAdd no final

Kernel Híbrido de Shared Memory

Problema: Autômato com 2830 estados não cabe na shared memory (48 KB)

Solução: Abordagem híbrida

Em Shared Memory (9 KB):

- `est0[256]`: Lookup estado 0
- `failure_cache[1024]`
- `output_cache[1024]`

Em Global Memory:

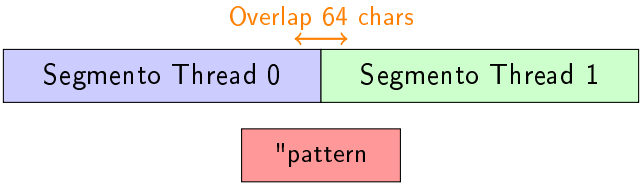
- VI, NE, VE, VS
- Failure completo
- Output completo

Configuração:

- Block size: 1024 threads (melhor ocupância)
- Estado 0 é 50% dos acessos $\rightarrow$ est0 em shared acelera muito

Técnica de Overlap

Problema: Padrões que cruzam bordas entre segmentos



Solução implementada:

- Cada thread começa a processar 64 caracteres antes do seu segmento
- Só conta matches dentro do seu segmento real
- Garante que nenhum padrão seja perdido nas bordas

Ambiente de Testes

CPU:

- AMD Ryzen 5 5500
- 6 cores / 12 threads
- Arquitetura Zen 3

Memória:

- 32 GB DDR4
- Disco: SSD M.2 1 TB (65.1 GB livres)

Software: CUDA 12.x, GCC, Linux (WSL2)

GPU:

- NVIDIA GeForce RTX 4060 Ti
- 34 SMs, 4352 CUDA Cores
- 8 GB GDDR6
- 48 KB Shared Memory/bloco
- 32 MB Cache L2
- Driver: 581.80

Dataset e Metodologia

Dataset:

- 495 padrões tipo Snort IDS
- Categorias:
  - SQL Injection (63)
  - XSS (59)
  - Buffer Overflow (52)
  - Command Injection (47)
  - Path Traversal (43)
  - Protocol Attacks (40)
- Autômato: 2830 estados

Instâncias de Teste:

- 1 KB, 10 KB, 100 KB
- 1 MB, 10 MB
- 50 MB, 100 MB
- 500 MB, 1 GB

Metodologia:

- 5 iterações por experimento
- Média dos tempos de kernel
- Validação: matches GPU = CPU

Resultados: Textos Pequenos (menor ou igual a 10 MB)

Tamanho	Serial	GPU Global	GPU Shared	Melhor
1 KB	0.01 ms	0.15 ms	0.14 ms	Serial
100 KB	0.80 ms	0.22 ms (4x)	0.50 ms (2x)	Global
1 MB	8.01 ms	0.37 ms (22x)	2.76 ms (3x)	Global
10 MB	81 ms	1.46 ms (56x)	4.08 ms (20x)	Global

Análise:

- GPU Global vence para textos pequenos
- Cache L2 de 32 MB mascara latência da memória global
- Overhead do cache híbrido não compensa

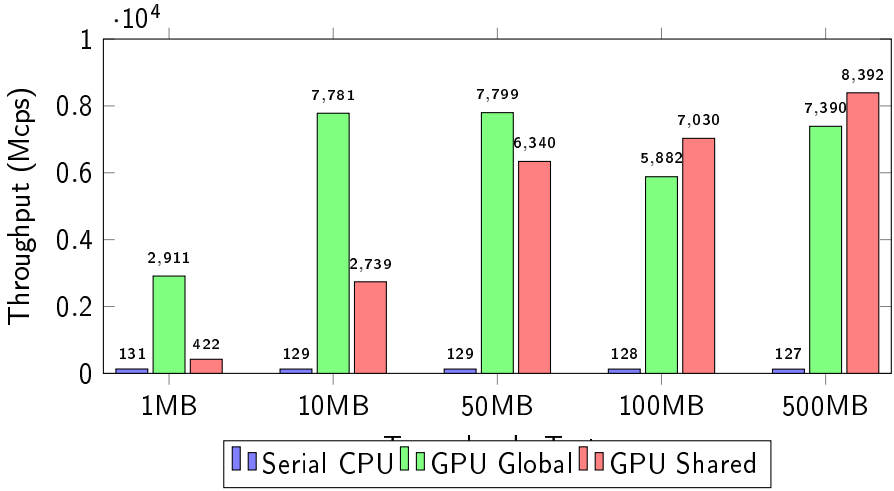
Resultados: Textos Grandes (maior ou igual a 50 MB)

Tamanho	GPU Global	GPU Shared	Ganho Shared	Speedup vs CPU
50 MB	6.87 ms	8.51 ms	-24%	59x
100 MB	18.65 ms	14.96 ms	**+20%**	55x
500 MB	83.59 ms	62.49 ms	**+25%**	66x
1 GB	150 ms	120 ms	**+25%**	71x

Análise:

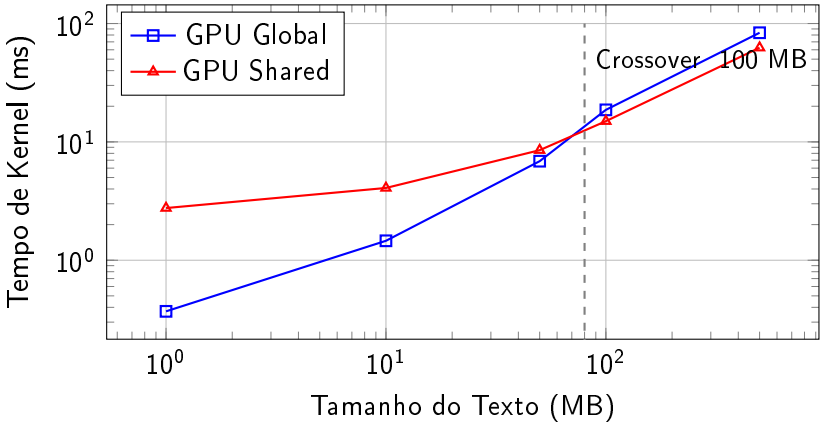
- Shared Memory vence para textos maiores que 100 MB
- Ganho de até 25% no tempo de kernel
- Speedup máximo: 66x vs CPU serial

Throughput Comparativo



Throughput máximo: 8.4 Gcps (GPU Shared, 500 MB)

Crossover Point



Crossover point: 100 MB (acima disso, Shared Memory é melhor)

Principais Contribuições

1. **Compactação eficiente da STT**
   - Redução de 98.7% no uso de memória
   - 2830 KB $\rightarrow$ 36 KB
2. **Kernel híbrido para autômatos grandes**
   - Suporta qualquer número de estados
   - Cache inteligente em shared memory
3. **Speedup significativo**
   - Até 66x vs CPU serial
   - 25% mais rápido que memória global para textos grandes
4. **Técnica de overlap**
   - Garante correção nos limites entre segmentos
   - 100% de precisão nos matches

Trabalhos Futuros

- **Texture Memory**: Usar texture cache para acessos à STT
- **Constant Memory**: Armazenar est0 em constant memory (broadcast)
- **Multi-GPU**: Distribuir texto entre múltiplas GPUs
- **Streams CUDA**: Overlap de transferência e computação
- **Integração com Snort**: Plug-in para uso em produção

Resumo

Métrica	Valor
Padrões Snort	495
Estados do Autômato	2830
Compressão STT	98.7%
Speedup Máximo	66x
Throughput Máximo	8.4 Gcps
Ganho Shared vs Global	+25% (textos maiores que 100 MB)

A implementação demonstra que a compactação da STT combinada com uso inteligente de shared memory pode acelerar significativamente a detecção de intrusões baseada em Aho-Corasick.

Obrigado!

Perguntas?

Thiago Carvalho
TN741 - Computação de Alto Desempenho
UFRRJ - 2025