EP2: Criptografia e Hashing em GPUs usando CUDA

Victor O. F. Faria (8515919), Gustavo Covas (7995052), Mauricio Luiz Abreu Cardoso (6796479)

MAC 5742-0219 Introdução à Programação Concorrente, Paralela e Distribuída

1. Introdução

Utilizamos o conteúdo da lista de vídeos de tutoriais de *CUDA* disponível em `https://www.youtube.com/watch?v=m0nhePeHwFs&list=PLKK11Ligqititws0ZOoGk3SW-TZCar4dK` para auxiliar no entendimento do funciomanento básico de *CUDA*.

Escolhemos os algoritmos que aparentemente seriam os mais fáceis de paralelizarmos e que aparentavam menos complexidade de serem entendidos. Um deles a princípio fora o *arcfour*, contudo apesar de bastante simples, sua paralelização foi possível e descontinuamos o trabalho sobre ele porque suas iterações dependiam fortemente dos estados determinados por iterações anteriores. Assim sendo, ficamos com os algortimos: *rot13*, *md2* e *base64*, cujas implementações seguem detalhadas abaixo.

2. Algoritmos escolhidos

2.1. ROT-13

Escolhemos esse algoritmo devido à sua simplicidade. Como o algoritmo é trivialmente paralelizável, na nossa implementação em *CUDA* cada thread codifica um único caracter da string. Contudo, apesar de simples, o algoritmo é um bom exemplo para ilustrar o funcionamento básico de *CUDA* - inclusive acreditamos que possa ser um bom exemplo a se trabalhar em aula.

2.2. MD2

Para implementar a versão em *CUDA*, decidimos simplificar a struct `MD2_CTX` para facilitar a alocação de memória na GPU utilizando o `cudaMalloc` (Não conseguimos alocar um vetor de structs). Sendo assim, os arrays de `BYTE`s `ctx->data`, `ctx->state` e `ctx->checksum` foram concatenados em um único array de `BYTE`s, e os acessos aos dados antes armazenados por aqueles arrays é feito utilizando offsets calculados para o array único. Ao final desse processo, percebemos que o algoritmo MD2 não é paralelizável de forma fácil, pois o cálculo do digest de um pedaço da mensagem depende especificamente dos `BYTE`s de estado determinados pelo cálculo das partes anteriores. Assim, entregamos uma versão sequencial que roda na GPU, utilizando apenas um único thread. Provavelmente seu desempenho é inferior a uma implementação usando a GPU.

2.3. Base64

Aqui temos um algoritmo cuja paralelização em *CUDA* foi interessante. Entretanto, antes de continuarmos devemos informar que não finalizamos completamente sua implementação - acabamos ficando travados após a codificação da função de encode e acabamos por fim descobrindo que o algoritmo original também não correspondia à saida presente no código e assim o abandonamos. Contudo houve o aprendizado de uma paralelização não trivial, como descrito abaixo.

Fora alguns casos especiais de borda, os quais foram delegados trivialmente a Threads específicas, o trecho principal de código dividia a saida codificada em pedaços de 76 caracteres (break line) por convenção. Neste momento tivemos que pensar como decidir quais das threads deveriam inserir as quebras de linhas e como essas quebras de linha impactariam onde as demais threads deveriam inserir os caracteres de saída no buffer. Por exemplo, dada uma entrada qualquer cuja saída possua 760 caracteres (sem quebras de linha), não seria possível dividir trivialmente em 10 threads e cada uma escrevendo em uma posição fixa no buffer de saida.

A solução para tanto foi ao invés de contabilizar as quebras de linha iterativamente como na versão orignal, matematizar o problema e derivarmos onde seriam as quebras de linha através de aritmética básica e a partir daí manter um contador para cada thread dizendo quantas quebras de linha já foram colocadas no buffer antes dessa thread escrever. Assim sendo, por exemplo a thread que cuida de escrever os caracteres 77 a 80 da sequência de saída, os escreveria nas posições 78 a 81 do buffer pois $\left\lfloor \frac{77}{76} \right\rfloor = 1$ e analogamente para os demais.