# Utilização de GPU na geração de tabelas arco-íris para a função hash MD5

**Alcides Mignoso e Silva**[1]

[1]Departamento de Computação - Universidade Federal de São Carlos (UFSCar)
13565-905 – São Carlos – SP – Brasil

alcidesmig@gmail.com

***Abstract.*** *This work describes the results obtained in the use of a GPU for the generation of rainbow tables for the MD5 hash function. The OpenCL architecture was been used, with the C ++ programming language (compiler version 7.4.0), in order to parallelize the application of the hash function in a previously generated word list. It was found that parallelization with the use of the GPU as the accelerator device can bring results hundreds of times better than the code executed sequentially on a CPU.*

***Resumo.*** *Este trabalho descreve os resultados obtidos na utilização de uma GPU para a geração de tabelas arco-íris para a função hash MD5. Fora utilizada a arquitetura OpenCL, com a linguagem de programação C++ (compilador versão 7.4.0), de modo a paralelizar a aplicação da função hash em uma lista de argumentos previamente gerada. Fora constatado que a paralelização com a utilização da GPU como dispositivo acelerador pode trazer resultados centenas de vezes melhor do que o código executado de forma sequencial em uma CPU.*

## 1. Introdução

Tabelas arco-íris - rainbow tables - são tabelas pré-computadas que armazenam pares valor e hash de um valor com a finalidade de reverter funções hash criptográficas. Dessa forma, tendo em mãos uma hash pode-se obter o valor associado à ela através de uma busca na tabela. Conforme o alfabeto das letras utilizadas na geração das palavras presentes na tabela vão aumentando, o seu tamanho aumenta exponencialmente, tornando a sua geração uma tarefa muito custosa, e é nesse cenário que pode-se estudar a paralelização da geração de tabelas como essa.

Para o presente estudo a função hash utilizada foi a MD5, uma função hash que foi projetada por Ronald Rivest, em meados de 1991, como sucessora da função MD4 [Fisher ]. Apesar de ter sido considerada ”criptograficamente quebrada e inadequada para uso posterior”pelo CMU Software Engineering Institute, a mesma ainda é amplamente utilizada como função hash criptográfica, utilizada para armazenar senhas e dados sensíveis em incontáveis sistemas da atualidade.

Buscou-se, portanto, desenvolver um algoritmo que pudesse gerar uma tabela arco-íris para hashes MD5 de forma paralela, ao utilizar os diversos núcleos existentes em uma GPU, para posterior comparação com o resultado obtido em uma CPU tanto de forma paralela quanto sequencial.

## 2. Desenvolvimento

Para paralelização do problema fora necessário, inicialmente, um estudo sob a versão sequencial do algoritmo.

```
std::vector<data> plain;
read_from_file(plain, file_start_read, 1e8);
int size = plain.size();
std::vector<hashed_data> hashed(size);

for (int i = 0; i < plain.size(); i++) {
    uint8_t result[16];
    md5((uint8_t *) plain[i].value, plain[i].size, result);
    hashed[i] = result;
}
```

**Figura 1. Algoritmo para geração da rainbow table de forma sequencial**

O objetivo desse trabalho não foi abordar a paralelização nas operações de leitura e escrita em disco, que não poderiam ser feitas pela GPU, e sim das operações de hash realizadas no algoritmo. Dessa forma, pode-se perceber que temos um algoritmo que faz repetições de acordo com o tamanho da lista de palavras para ele passada, aplicando a função hash para cada elemento da lista de forma sequencial. Pode-se constatar, durante uma primeira análise do algoritmo presente na Figura 1, que aparentemente não existe dependência de dados entre uma iteração e a outra, o que é excelente para a paralelização, uma vez que nem sempre pode-se controlar a ordem da execução das tarefas de forma eficiente.

Fica em aberto, então, a possibilidade de paralelizar o que acontece dentro da função MD5. A função MD5 não é uma função custosa, a sua complexidade de tempo cresce linearmente em relação a quantidade de blocos de 64 bytes existentes na mensagem, e como a nossa mensagem dificilmente ultrapassará 64 caracteres (sendo um byte para cada carácter), podemos considerar a complexidade de tempo da função para essa aplicação como constante, assim como a complexidade de espaço. Dessa forma, tendo em vista que a tarefa de executar uma função hash em uma determinada palavra poderia ser realizada rapidamente por um único núcleo de uma GPU, mesmo com a função MD5 sendo bastante extensa, é tida como boa alternativa uma paralelização dividindo as tarefas de aplicar a função nas palavras, uma por núcleo da GPU.

Ainda assim, existem mais motivos para optar por não realizar a paralelização dentro da função MD5. A divisão da função hash em pequenas partes visando a sua paralelização traria diversos problemas custosos de serem contornados. A função MD5 possui grande dependência entre os dados utilizados e gerados por ela em tempo de execução, o que causaria problemas, como por exemplo a comunicação entre tarefas visando manter a sincronização entre elas, que é um dos grandes problemas da programação paralela e certamente seria bastante custoso de resolver para esse caso. Para tanto, a alternativa mais eficiente aparenta ser a que mantém a execução da função MD5 inteiramente sequencial. Sendo assim, o problema em questão - a geração dos valores da tabela arco-íris - pode ser chamado de embaraçosamente paralelo, uma vez que o laço de repetição principal do programa pode ser quebrado e paralelizado sem nenhum esforço.

Dessa forma, utilizando da arquitetura OpenCL, pôde-se paralelizar a execução das funções hash escrevendo um kernel que executasse a função MD5 em um valor re-

cebido de um buffer de entrada e gravasse o valor hash em um buffer de saída. Como modelo, foi utilizada uma implementação da função MD5 já existente [MD5 ], que teve que ser fortemente adaptada para que o kernel se tornasse viável de ser utilizado. A arquitetura OpenCL provê funções previamente existentes que fazem o escalonamento das tarefas para os núcleos da GPU, e é recomendado, pelos desenvolvedores, que parâmetros como ”work group size”não sejam alterados na mão, para que o tamanho mais adequado possa ser utilizado pela plataforma. Sendo assim, bastou escrever o kernel, como pode ser observado na Figura 2, e enviar as tarefas para o dispositivo usando as chamadas de funções OpenCL, como pode ser observado na Figura 3.

```
typedef struct {
    uint8_t value[16];
} hashed_data;
typedef struct {
    char value[SIZE_RAW_TEXT];
    size_t size;
} data;

__kernel void md5_hash(__global data * in, __global hashed_data * out) {
    int i = get_global_id(0);
    data aux = in[i];

    uint8_t result[16];
    md5((uint8_t *) &aux.value, in[i].size, result);
    int j = 0;
    for (j = 0; j < 16; j++) {
        out[i].value[j] = result[j];
    }
}
```

**Figura 2. Base do kernel do código OpenCL**

```
std::vector<data> plain;
read_from_file(plain, file_start_read, 1e8);
int size = plain.size();
std::vector<hashed_data> hashed(size); // Host memory for store hashed PINs

/* Create buffers */
cl::Buffer d_plain = cl::Buffer(context, plain.begin(), plain.end(), true);
cl::Buffer d_hash = cl::Buffer(context, CL_MEM_WRITE_ONLY, sizeof(hashed_data) * size);


/* Enqueue tasks */
md5_hash(
    cl::EnqueueArgs(
        queue,
        cl::NDRange(size)),
    d_plain, d_hash);

queue.finish();

/* Copy from device to cpu */
cl::copy(queue, d_hash, hashed.begin(), hashed.end());
```

**Figura 3. Base do código OpenCL**

## 3. Resultados e discussões

Para os testes de benchmark foram utilizados uma máquina com o processador Intel® Core™ i7-5500U (4M de cache, até 3.00 GHz), onde foi testado o código sequencial e o código paralelizado (OpenCL) utilizando essa CPU como acelerador, e uma máquina Intel® Core™ i3-7100 3.90GHz com uma placa de video GTX 1050 Ti de 4GB de RAM, onde foi testado o código paralelizado (OpenCL) utilizando a placa de video como dispositivo acelerador. Ainda nos testes, foi utilizado um arquivo de entrada gerado em tempo de execução composto com todas as combinações possíveis de até 4 ou até 5 dígitos de um alfabeto pré-definido composto por todas as letras do alfabeto de forma minúscula e maiúscula e todos os números de 0 a 9. Nos tempos contabilizados para cada teste não estão contabilizados os tempos de leitura e escrita em disco, somente o tempo de execução das funções hashes e transferência de dados para e do dispositivo em questão. Cada instância do problema foi rodada 8 vezes em cada dispositivo, e a margem de erro dos testes está destacada nos gráficos.

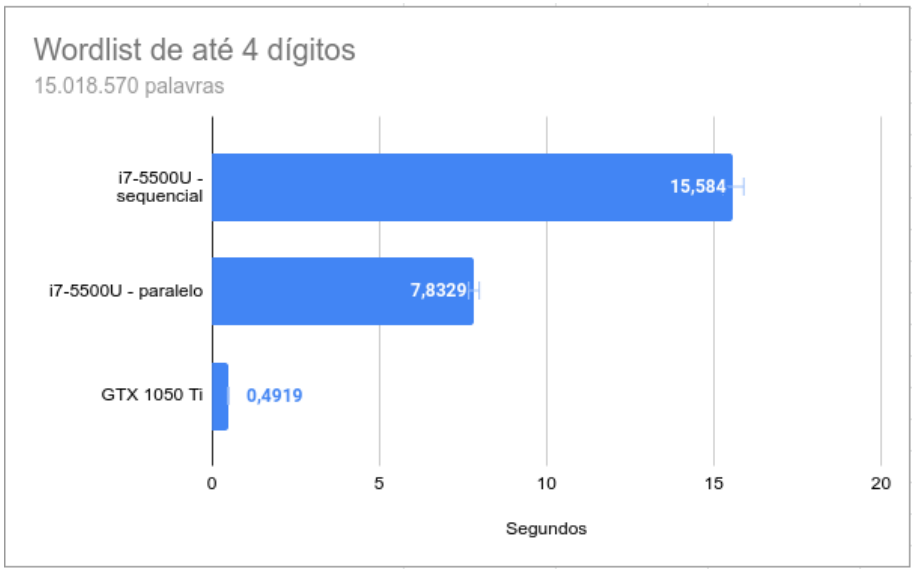


**Figura 4. Benchmark para até 4 dígitos**

Apesar de não ser tão justa a comparação (por serem dispositivos completamente diferentes), pode-se perceber, tanto na Figura 4 quanto na Figura 5 que o mesmo código é executado absurdamente mais rápido na GPU de forma paralelizada do que na CPU, tanto na forma sequencial (mais de 103 vezes) quanto na paralela (mais de 43 vezes). Isso se dá, majoritariamente, pelo fato da GPU em questão ter 768 unidades de processamento enquanto a CPU em questão possuí 4. Apesar dos cores da GPU sofrerem pelas limitações impostas pela arquitetura (SIMT + SIMD), é evidente que para esse exemplo, onde a mesma sequência de instruções é aplicada várias vezes em múltiplos dados, ela se sobressaí em relação a CPU em questão e muito provavelmente em relação a maioria das CPUs existentes no mercado de computadores pessoais.

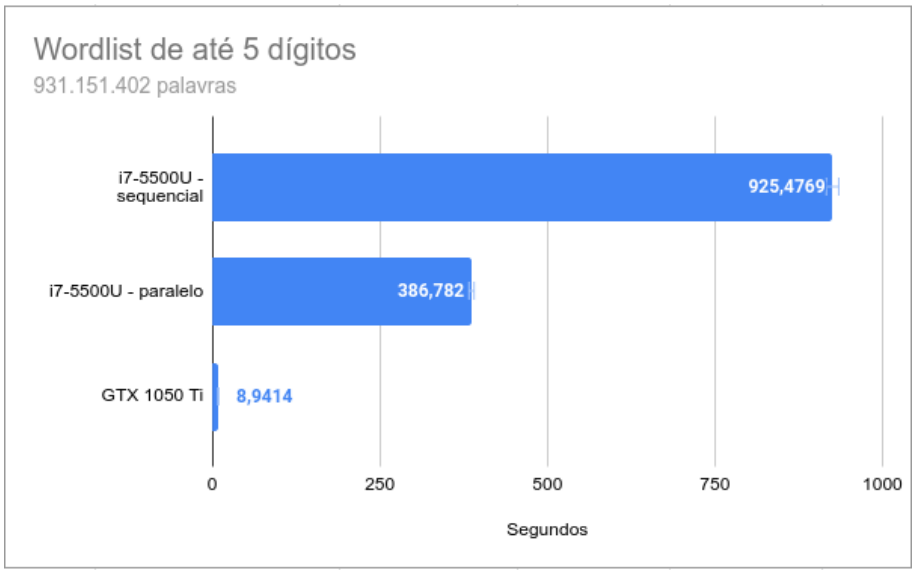


**Figura 5. Benchmark para até 5 dígitos**

Vale ressaltar que a GPU utilizada para os testes foi lançada em 2016, e na época não era a mais potente de sua linha. Dessa forma, resultados ainda mais gritantes podem ser alcançados utilizando dispositivos com maior capacidade de processamento. Além disso, para arquivos cujo tamanho não cabem inteiramente na memória (como por exemplo o da Figura 5), ocorrem leituras em partes de 1e8 bytes de forma sequencial. Dessa forma, caso mais GPUs estivessem disponíveis esse processamento poderia ser paralelizado enviando subsets de dados diferentes para cada diferente GPU.

## 4. Conclusão

Através dos resultados obtidos, pode-se concluir que a utilização de GPUs para tarefas como aplicação de funções hashes em um grande conjunto de dados, assim como teorizado, é uma excelente alternativa. Também pode-se perceber a facilidade para geração de tabelas arco-íris para alfabetos simples, logicamente caso símbolos e caracteres especiais fossem utilizados no alfabeto a quantidade de operações aumentaria exponencialmente, mas isso ainda poderia ser contornado acrescentando mais dispositivos aceleradores, como GPUs, ou até mesmo fazer uma implementação para FPGAs, que traria suas vantagens e desvantagens.

Por fim, fica clara a necessidade do estudo aprofundado de programação paralela por profissionais da computação, uma vez que cada dia mais a complexidade dos problemas vem crescendo e a utilização de algoritmos paralelos para resolvê-los é uma das alternativas mais promissoras da atualidade.

O código desenvolvido para o projeto pode ser encontrado em: https://github.com/alcidesmig/md5-rainbow-table-gen-opencl.

## Referências

Fisher, T. *MD5 history*. https://www.lifewire.com/what-is-md5-2625937.

MD5. *pod32g MD5 implementation*. https://github.com/pod32g/MD5.