

## Łamanie MD5

Obliczenia równoległe na kartach graficznych CUDA

Rafał Leja (UWr)

29 stycznia 2026

# Plan

- ① Tło: MD5 i problem do rozwiązania.
- ② Implementacja: CPU vs GPU.
- ③ Benchmarki.
- ④ Podsumowanie.

# MD5

Funkcja skrótu MD5 (Message-Digest Algorithm 5)

$h : 0, 1^* \rightarrow 0, 1^{128}$  128-bitowy skrót (hash) z dowolnej długości wejścia. 128-bitowy stan wewnętrzny

# MD5 do przechowywania haseł

- W latach 1990 - 2000 standard do haszowania haseł
- Główna zaleta - szybkość obliczeń
- Około 2004 odkryto, że MD5 jest podatne na kolizje.
- W rezultacie, wiele systemów zaczęło migrować do bezpieczniejszych algorytmów, takich jak SHA-1\* i SHA-256.
- \*SHA-1 również okazało się podatne na kolizje.

# MD5 do przechowywania haseł

W 2019 ponad ćwierć CMS-ów używało MD5 do haszowania haseł użytkowników.

# MD5 - padding

- Dane wejściowe są dzielone na bloki 512-bitowe
- Jeśli ostatni blok jest krótszy niż 512 bitów, stosowany jest padding
  - Dodanie bitu '1' na końcu danych
  - Dodanie bitów '0' aż do osiągnięcia długości 448 bitów
  - Dodanie 64-bitowej reprezentacji długości oryginalnych danych

## MD5 - inicjalizacja

- MD5 używa czterech 32-bitowych słów jako stanu wewnętrznego:
  - A = 0x67452301
  - B = 0xEFCDAB89
  - C = 0x98BADCFE
  - D = 0x10325476
- Te wartości są inicjalizowane na początku procesu haszowania.

# MD5 - funkcje pomocnicze

MD5 używa czterech nieliniowych funkcji bitowych:

- $F(X, Y, Z) = (X \text{ AND } Y) \text{ OR } (\text{NOT } X \text{ AND } Z)$
- $G(X, Y, Z) = (X \text{ AND } Z) \text{ OR } (Y \text{ AND NOT } Z)$
- $H(X, Y, Z) = X \text{ XOR } Y \text{ XOR } Z$
- $I(X, Y, Z) = Y \text{ XOR } (X \text{ OR NOT } Z)$

# MD5 - główna pętla

- Dla każdego 512-bitowego bloku danych:
  - Podziel blok na szesnaście 32-bitowych słów  $M[0..15]$
  - Wykonaj 64 rundy operacji, podzielone na cztery fazy po 16 rund każda
  - W każdej rundzie użyj jednej z funkcji F, G, H, I oraz odpowiedniego słowa  $M[i]$  i stałej  $K[i]$

# MD5 - aktualizacja stanu

- Po przetworzeniu każdego bloku, zaktualizuj stan wewnętrzny:
  - $A = A + AA$
  - $B = B + BB$
  - $C = C + CC$
  - $D = D + DD$
- Gdzie AA, BB, CC, DD to wartości stanu przed przetworzeniem bieżącego bloku.

# MD5 - wynik końcowy

- Po przetworzeniu wszystkich bloków, wynikowy hash to konkatenacja A, B, C, D
- Wynik jest reprezentowany jako 32-znakowy ciąg szesnastkowy.

# Implementacja - MD5

```
uint32_t F, g;
for (int i = 0; i < 64; i++) {
    if (result[0] != 0)
        | return;
    switch (i / 16) {
    case 0:
        | F = (b & c) | (~b & d);
        | g = i;
        | break;
    case 1:
        | F = (d & b) | (~d & c);
        | g = (5 * i + 1) % 16;
        | break;
    case 2:
        | F = b ^ c ^ d;
        | g = (3 * i + 5) % 16;
        | break;
    case 3:
        | F = c ^ (b | ~d);
        | g = (7 * i) % 16;
        | break;
    }

    F = F + a + K_d[i] + M[g];
    a = d;
    d = c;
    c = b;
    b = b + ((F << S_d[i]) | (F >> (32 - S_d[i])));
}
```

Rysunek: Fragment implementacji MD5

# implementacja - generowanie haseł

```
__device__ void int_to_passwd(unsigned long long idx, int len, int base,
                           const char *all_chars, uint32_t *M) {
    char passwd[64] = {[0]=0};
    for (int i = len - 1; i >= 0; i--) {
        passwd[i] = all_chars[idx % base];
        idx /= base;
    }

    for (int i = 0; i < len; i++) {
        M[i / 4] |= (uint8_t)passwd[i] << ((i % 4) * 8);
    }
}
```

Rysunek: Fragment implementacji generowania haseł

# Implementacja - CUDA

```
__global__ void md5_passwd_gpu(const char *all_chars, int base, int len,
                               unsigned long long offset,
                               struct md5_state *target,
                               unsigned long long *result) {
    unsigned long long idx = blockIdx.x * blockDim.x + threadIdx.x + offset;
    uint32_t M[16] = {[0]=0};
    int_to_passwd(idx, len, base, all_chars, M);
    for (int i = len; i < 64; i++) {
        if (i == len) {
            | M[i / 4] |= 0x80 << ((i % 4) * 8);
        } else if (i >= 56) {
            uint64_t bit_len = len * 8;
            M[14] = (uint32_t)(bit_len & 0xFFFFFFFF);
            M[15] = (uint32_t)((bit_len >> 32) & 0xFFFFFFFF);
            break;
        }
    }

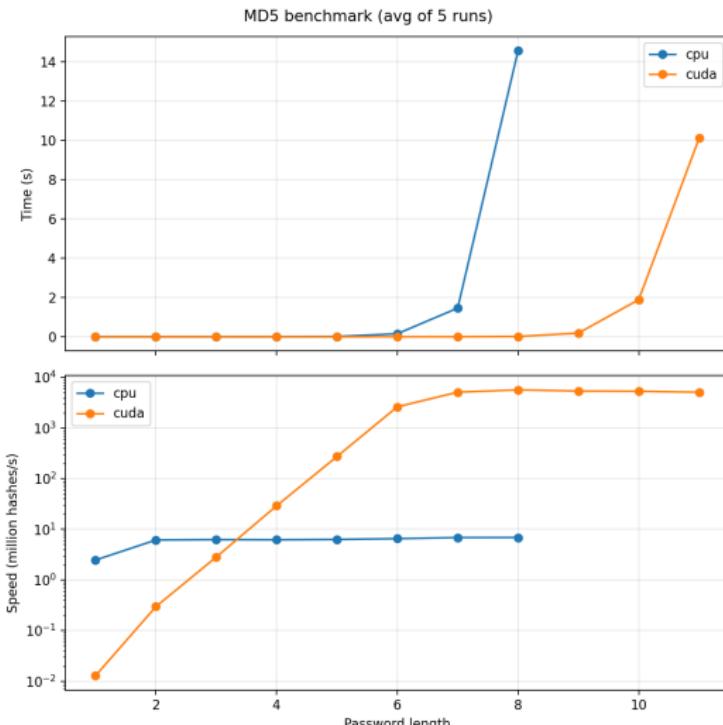
    if (result[0] != 0)
        return;

    uint32_t a = 0x67452301;
    uint32_t b = 0xefcdab89;
    uint32_t c = 0x98badcfe;
    uint32_t d = 0x10325476;
    md5_round(o_a: &a, o_b: &b, o_c: &c, o_d: &d, M, result);

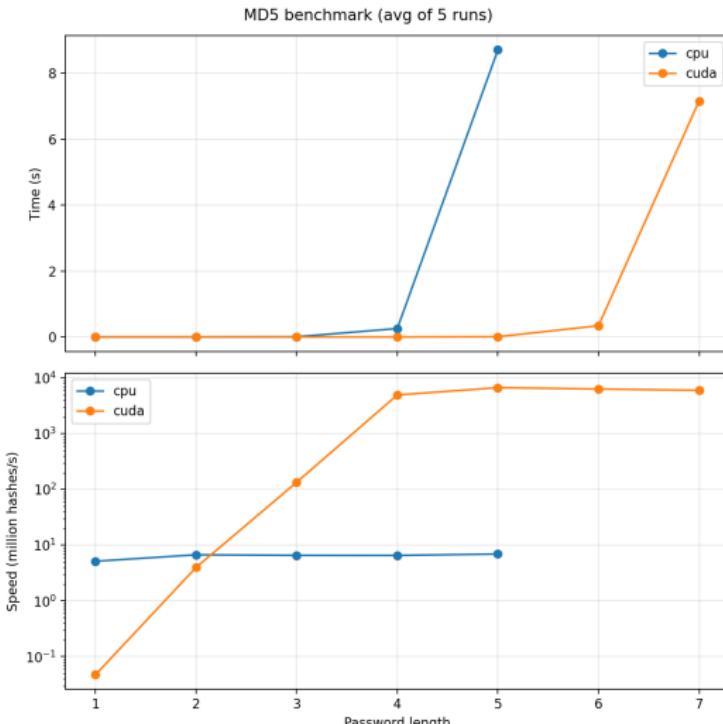
    if (target->a == a && target->b == b && target->c == c &&
        target->d == d) {
        atomicExch(address: &result[0], val: idx + 1);
    }
}
```

Rysunek: Fragment implementacji jądra CUDA

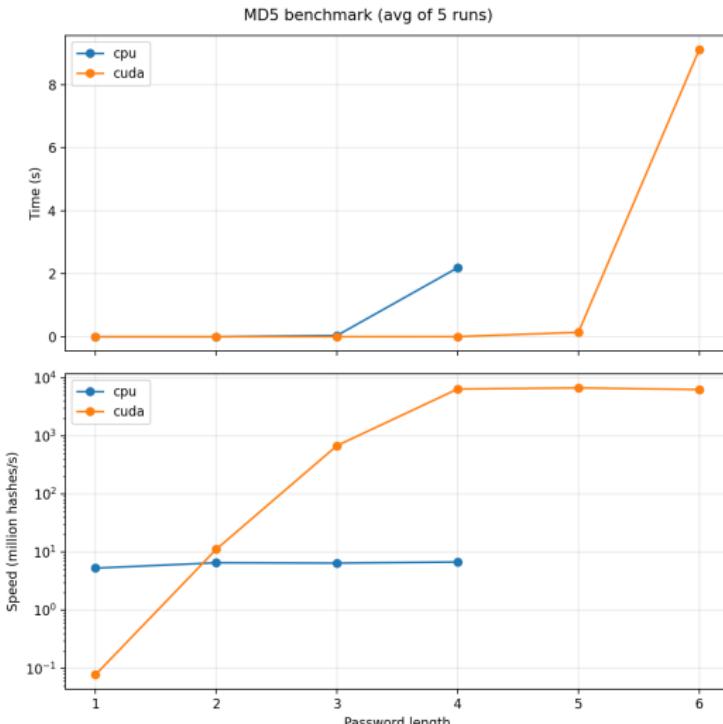
# Wydajność - porównanie CPU i GPU (cyfry)



# cyfry + małe litery



# cyfry + małe i duże litery



# cyfry + litery + znaki specjalne

