UNIVERSITÀ
DEGLI STUDI
FIRENZE

DES Decryption con C, CUDA & OpenMP

Edoardo D'Angelis
&
Mirco Ceccarelli

Introduzione

- **Obiettivo:** Decifrare una password di 8 caratteri avendo a disposizione il suo valore di hash generato tramite algoritmo DES e supponendo di conoscere il *salt*.

- **3 approcci diversi**

SEQUENZIALE: C

PARALLELO:
CUDA & OpenMP

Data Encryption Standard

- Il **DES** è un algoritmo di cifratura basato su chiave simmetrica per cifrare e decifrare dati
- Considerato insicuro perché la **chiave** utilizzata per la cifratura è di soli **56 bit**
- Facilmente vulnerabile agli attacchi
- Sostituito dall'**AES** (Advanced Encryption Standard) negli ultimi anni

Descrizione del Dataset

- Generato con uno script **Python** a partire da un file contenente 10 milioni di password
- Estratte solo le password lunghe **8 caratteri** e appartenenti al set **[a-zA-Z0-9]**
- **Dizionario finale**: 1 milione e mezzo di password

↓

Per i test scelte password uniformemente distribuite

Versione Sequenziale in C

- **Input**: il dizionario e la password cifrata da cercare
- Si scorre tutto il dizionario e ad ogni passo viene cifrata la parola corrente con la funzione ***crypt***
- Se i due hash sono uguali il ciclo viene interrotto e la password è stata trovata
- Si itera fino alla fine del dizionario
- Il tempo di esecuzione è misurato con la funzione ***clock*** della libreria ***time***

Versione Parallela in CUDA

- **Input**: il dizionario e la password cifrata da cercare
- Parole convertite in ***uint64_t*** (interi a 64 bit) per adeguarsi alla libreria des-cuda
- Allocazione e copia di memoria da ***Host*** a ***Device***
- Lancio del **Kernel**

```
// Organizzazione blocchi e grid
dim3 blockDim(blockSize);
dim3 gridDim(DICTIONARY_SIZE/blockDim.x + 1);

// Lancio kernel
kernel<<<gridDim, blockDim>>>(device_dictionary);
```

Versione Parallela in CUDA

- ***Salt*** e **password cifrata** da cercare sono salvate nella **Device Constant Memory**
- Ogni thread controlla una sola parola in base al suo indice
- Controllo per evitare ***overflow*** al contorno dell'ultimo blocco
- Misuro il tempo con ***clock***

```
__constant__ uint64_t salt;
__constant__ uint64_t encryptedPassword;

__global__ void kernel(uint64_t *dict) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < DICTIONARY_SIZE) {
    uint64_t cur_psw = dict[idx];
    uint64_t enc = full_des_encode_block(cur_psw, salt);
    if (enc == encryptedPassword) {
      // Password trovata!
      return;
    }
  }
}
```

Versione Parallela con OpenMP

- Creata la classe ***Decrypter*** con costruttore che prende in ingresso il nome del file relativo al dizionario e il *salt*
- Metodo ***decrypt*** prende in ingresso il numero di thread da usare (**#**pragma omp parallel num_threads)
- Si scorre tutto il dizionario (**#**pragma omp for)
- Ad ogni passo viene cifrata la parola corrente con la funzione ***crypt_r*** (versione **reentrant** di ***crypt***)

Versione Parallela con OpenMP

- Utilizzata la variabile booleana di tipo **volatile** per segnalare ai thread che la password è stata trovata

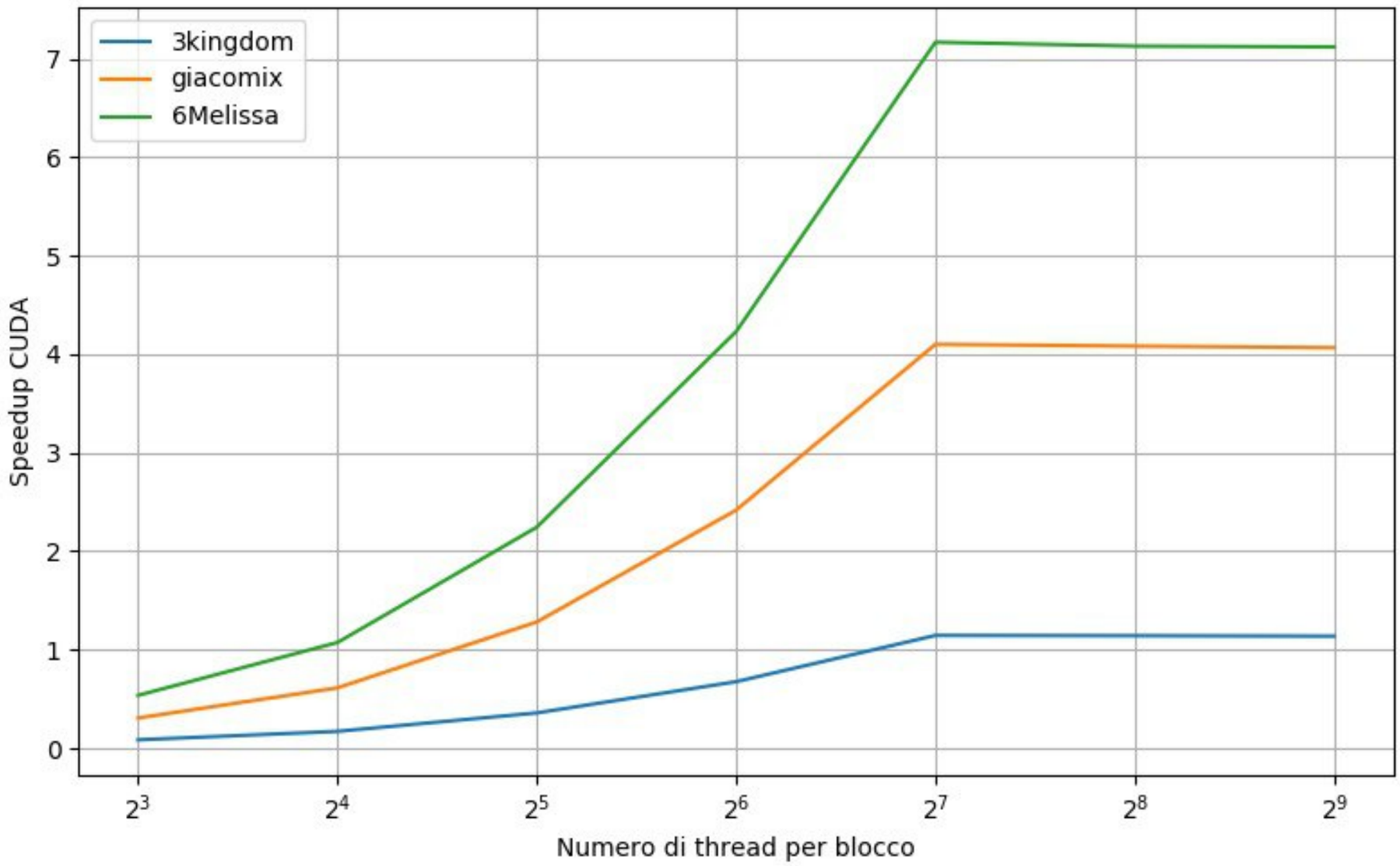
- Misuro il tempo con la funzione ***now()*** della classe ***steady_clock*** della libreria ***chrono***

```
double Decrypter::decrypt(int threads) {

  volatile bool found = false;
  auto start = chrono::steady_clock::now();

#pragma omp parallel num_threads(threads)
    {
  struct crypt_data data;
  data.initialized = 0;

#pragma omp for
  for (int i = 0; i < dict.size(); i++) {
    if (found) continue;
    char *current = crypt_r(dict[i], salt, &data);
    if (strcmp(current, encrypted) == 0) {
       // Password trovata!
       found = true;
    }
  }
    }

  if (found) {
    auto end = chrono::steady_clock::now();
    std::chrono::duration<double> s = end - start;
    return s.count();
  } else {
    return 0;
  }
}
```

Speedup in CUDA

- **Thread per blocco:** 8, 16, 32, 64, 128, 256, 512
- **Blocchi per grid:** $\frac{dictionary\_size}{block\_size} + 1$

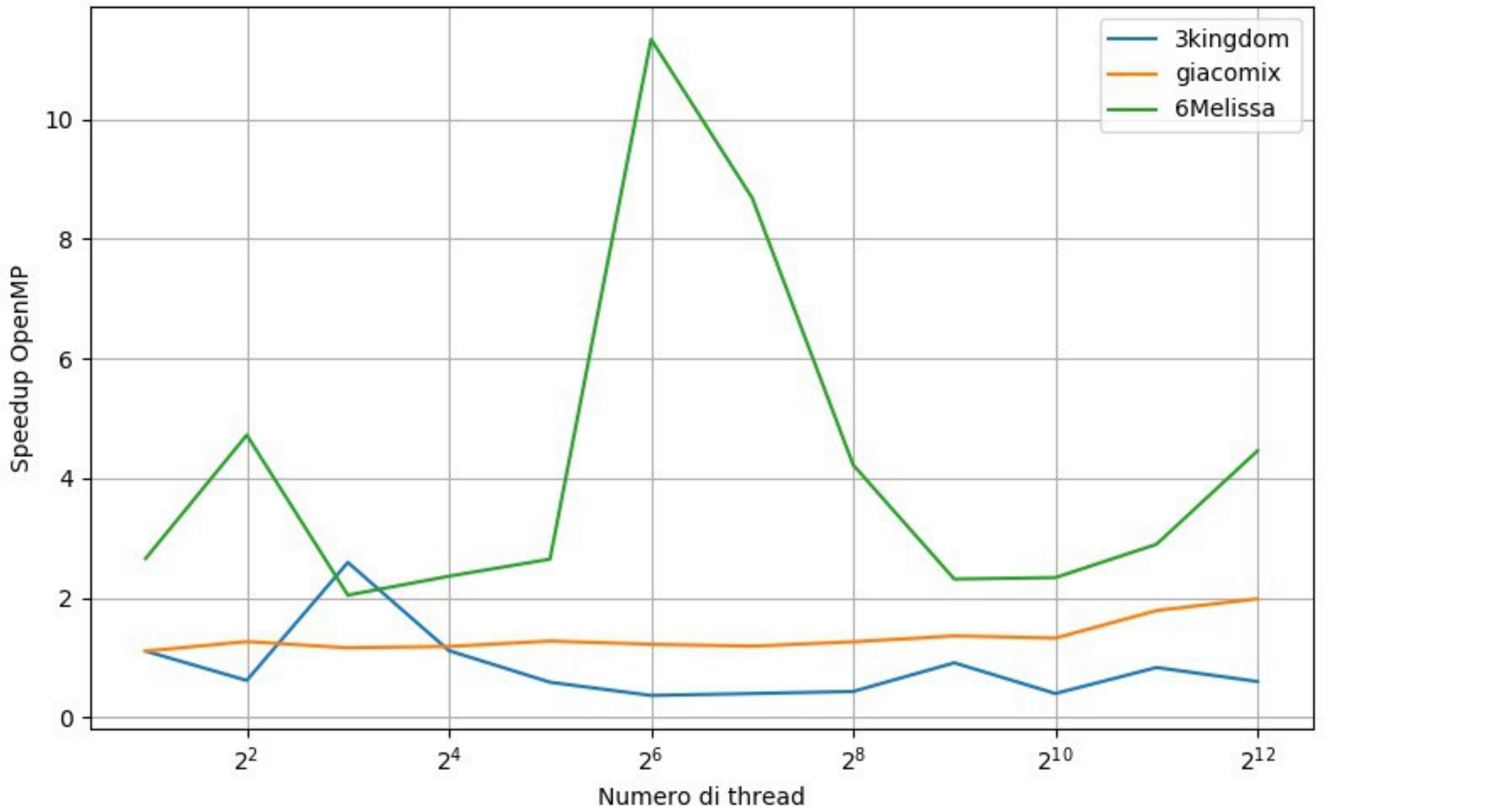
$$Speedup = \frac{tempo\_sequenziale}{tempo\_parallelo}$$



Speedup OpenMP

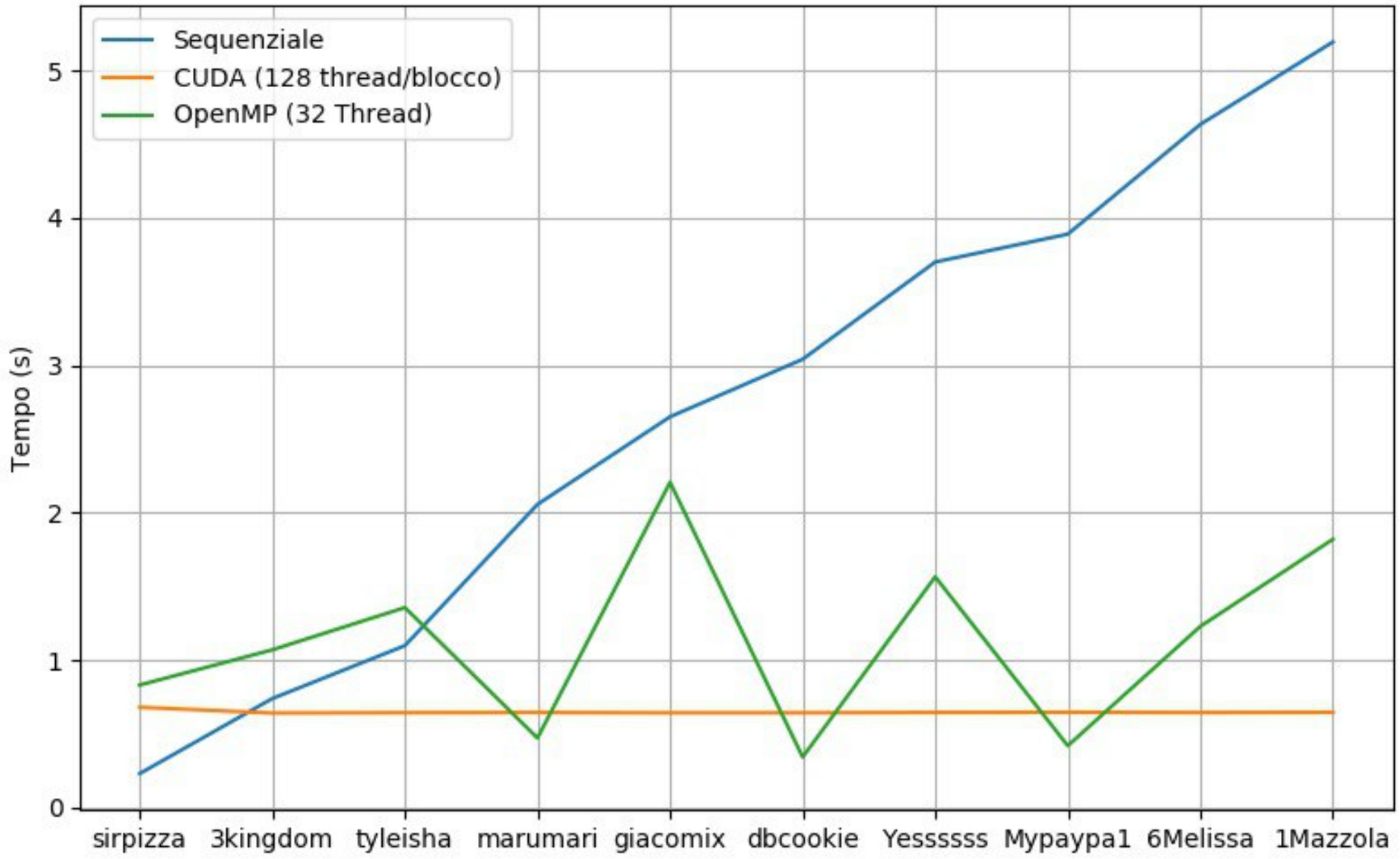
- **Thread utilizzati:** 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096

$$Speedup = \frac{tempo\_sequenziale}{tempo\_parallelo}$$



Confronto sui Tempi di Esecuzione

- **CUDA**: scelti 128 thread per blocco
- **OpenMP**: scelti 32 thread



Conclusione

- **Approccio parallelo** quasi sempre più **efficiente**, ma richiede una scelta appropriata del numero di thread da utilizzare per ottenere lo **speedup** migliore

- **CUDA:** più macchinoso da utilizzare, richiede una o più GPU, ma offre tempo di esecuzione costante

- **OpenMP:** facile da implementare ma il tempo di esecuzione dipende dalla divisione interna in ***chunk.***