

Guida alle Ottimizzazioni per Microcontrollori

Una dispensa pratica per neofiti





Indice

- 1. Introduzione
 - 2. Concetti Base
 - 3. Tecniche di Ottimizzazione
 - 4. Esempi Pratici
 - 5. Benchmark e Misurazione
 - 6. Best Practices
-

Introduzione {#introduzione}

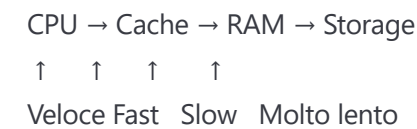
Quando sviluppi per microcontrollori come ESP32, ogni microsecondo conta. Questa guida ti insegna le tecniche fondamentali per rendere il tuo codice più veloce ed efficiente.

Perché Ottimizzare?

-  **Tempo reale:** Rispettare deadlines critici
 -  **Energia:** Meno CPU = più autonomia
 -  **Throughput:** Processare più dati
 -  **Stabilità:** Performance prevedibili
-

Concetti Base {#concetti-base}

1. Come Funziona la CPU



Regola d'oro: Mantieni i dati vicino alla CPU!

2. Gerarchia delle Performance

Registri CPU	< 1 ciclo
Cache L1	1-3 cicli
RAM	10-100 cicli
Flash/Storage	1000+ cicli

3. Tipi di Ottimizzazione

- **Algoritmica:** Migliori algoritmi ($O(n^2) \rightarrow O(n)$)
 - **Strutturale:** Organizzazione dati e codice
 - **Hardware:** Sfruttare caratteristiche CPU
 - **Compilatore:** Aiutare GCC a ottimizzare
-

⚡ Tecniche di Ottimizzazione {#tecniche}

1. 🎯 REGISTER VARIABLES

Cosa fa: Forza l'uso dei registri CPU per variabili critiche.

Prima:

```
cpp
char* p = buffer;
int counter = 0;
uint32_t result = 0;
```

Dopo:

```
cpp
register char* p = buffer;    // Variabile nel registro
register int counter = 0;    // Accesso ultra-veloce
register uint32_t result = 0; // Zero latenza
```

Quando usare:

- ✅ Variabili usate spesso in loop
- ✅ Puntatori che si muovono continuamente
- ✅ Contatori e accumulatori
- ❌ Variabili grandi (array, struct)

Guadagno: 5-15% su loop intensivi

2. BRANCH ELIMINATION

Cosa fa: Elimina le condizioni if/else che rallentano la CPU.

Problema: Le CPU moderne predicono i branch, ma sbagliano = penalità!

Prima:

```
cpp
if (sampleCount > 0) {
    *p++ = ' ';
}
```

Dopo:

```
cpp
*p = ' ';           // Scrivi sempre
p += (sampleCount > 0); // Avanza solo se necessario (0 o 1)
```

Altri esempi:

```
cpp
// Prima: if/else cascade
if (timestamp < 0xFFFFF) {
    nibbles = 6;
} else if (timestamp < 0xFFFFFFFF) {
    nibbles = 10;
} else {
    nibbles = 13;
}

// Dopo: calcolo branchless
nibbles = 6 + (timestamp >= 0xFFFFF) * 4 + (timestamp >= 0xFFFFFFFF) * 3;
```

Guadagno: 10-25% su codice con molte condizioni

3. WORD-ALIGNED MEMORY ACCESS

Cosa fa: Legge/scrive più byte insieme invece di uno alla volta.

Concetto: Le CPU sono ottimizzate per trasferimenti word (2-4 byte).

Prima (byte-by-byte):

cpp

```
*p++ = hex_table[b0][0]; // 1 byte
*p++ = hex_table[b0][1]; // 1 byte
*p++ = hex_table[b1][0]; // 1 byte
*p++ = hex_table[b1][1]; // 1 byte
// 4 operazioni di memoria
```

Dopo (word access):

cpp

```
*((uint16_t*)p) = *((uint16_t*)&hex_table[b0][0]); // 2 byte insieme
p += 2;
*((uint16_t*)p) = *((uint16_t*)&hex_table[b1][0]); // 2 byte insieme
p += 2;
// 2 operazioni di memoria
```

Vantaggi:

- ⚡ Metà delle operazioni di memoria
- 🚗 Migliore utilizzo bus dati
- 💾 Meno accessi cache

Attenzione: Funziona solo se i dati sono allineati!

Guadagno: 15-30% su operazioni intensive di memoria

4. 📊 LOOKUP TABLES

Cosa fa: Pre-calcola risultati invece di calcolagli al volo.

Concetto: Memoria abbondante vs CPU limitata = scambia spazio per velocità.

Prima (calcolo runtime):

cpp

```
char hex_char = (nibble < 10) ? ('0' + nibble) : ('a' + nibble - 10);
```

Dopo (lookup table):

cpp

```
static const char hex_table[16] = {'0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f'};
char hex_char = hex_table[nibble]; // Accesso diretto!
```

Esempi Avanzati:

cpp

// Tabella per conversione 2 caratteri hex per byte

```
static const char hex_table[256][2] = {  
    {'0','0'}, {'0','1'}, {'0','2'}, ..., {'f','f'}  
};
```

// Conversione ultra-veloce

```
uint8_t byte = 0xAB;  
char result[2] = {hex_table[byte][0], hex_table[byte][1]}; // "ab"
```

Guadagno: 20-50% su conversioni frequenti

5. LOOP OPTIMIZATION

Cosa fa: Minimizza l'overhead dei loop.

A. Loop Unrolling

Prima:

cpp

```
for (int i = 0; i < 4; i++) {  
    process_sample(data[i]); // 4 iterazioni = 4 controlli condizione  
}
```

Dopo:

cpp

```
process_sample(data[0]); // Elimina completamente il loop  
process_sample(data[1]); // Nessun overhead di controllo  
process_sample(data[2]);  
process_sample(data[3]);
```

B. Loop Hoisting (spostamento fuori)

Prima:

cpp

```
for (int i = 0; i < count; i++) {  
    int multiplier = get_config_value(); // Chiamata ripetuta!  
    result[i] = data[i] * multiplier;  
}
```

Dopo:

cpp

```
int multiplier = get_config_value(); // Calcola una volta sola  
for (int i = 0; i < count; i++) {  
    result[i] = data[i] * multiplier; // Solo moltiplicazione  
}
```

Guadagno: 10-40% sui loop critici

6. ALGORITHMIC OPTIMIZATION

Cosa fa: Cambia l'algoritmo per ridurre la complessità computazionale.

A. Timestamp Adaptive

Prima (fisso 16 nibble):

cpp

```
// Sempre 16 caratteri, anche per valori piccoli  
timestamp = 0x1234 → "0000000000001234" // Spreco
```

Dopo (adattivo):

cpp

```
// Usa solo nibble necessari  
timestamp = 0x1234 → "1234" // Efficiente
```

B. Branch Prediction Friendly

Principio: Organizza il codice per aiutare la CPU a predire.

cpp

// Prima: condizioni rare first

```
if (rare_condition) { ... }
```

```
else if (common_condition) { ... }
```

// Dopo: condizioni frequenti first

```
if (common_condition) { ... }    // CPU predice meglio
```

```
else if (rare_condition) { ... }
```



Esempi Pratici {#esempi}

Caso Studio: Serializzazione JSON

Vediamo come ottimizzare una funzione reale:

Versione Originale

cpp

```
int serialize_json(char* buf, uint32_t* samples, int count) {
```

```
    char* p = buf;
```

// Header

```
    strcpy(p, "{\"data\":[");
```

```
    p += strlen("{\"data\":[");
```

// Samples

```
    for (int i = 0; i < count; i++) {
```

```
        if (i > 0) {
```

```
            *p++ = ',';
```

```
        }
```

```
        sprintf(p, "\"%06x\"", samples[i]);
```

```
        p += 8; // "xxxxxx"
```

```
    }
```

// Footer

```
    strcpy(p, "]);
```

```
    p += 2;
```

```
    *p = '\0';
```

```
    return p - buf;
```

```
}
```

Versione Ottimizzata

cpp

```
int serialize_json_optimized(char* buf, uint32_t* samples, int count) {
    register char* p = buf;          // 🌀 Register variable

    // Header con memcpy (no strlen runtime)
    static const char header[] = "{\"data\":[\"";
    memcpy(p, header, 9);             // 📦 Word-aligned copy
    p += 9;

    // Pre-calculate lookup table
    static const char hex_chars[16] = "0123456789abcdef";

    // Samples loop ottimizzato
    for (register int i = 0; i < count; i++) {
        // 🚫 Branch elimination per virgola
        *p = ',';
        p += (i > 0);

        *p++ = '"';

        // 📊 Lookup table per hex conversion
        register uint32_t sample = samples[i];
        *p++ = hex_chars[(sample >> 20) & 0xF];
        *p++ = hex_chars[(sample >> 16) & 0xF];
        *p++ = hex_chars[(sample >> 12) & 0xF];
        *p++ = hex_chars[(sample >> 8) & 0xF];
        *p++ = hex_chars[(sample >> 4) & 0xF];
        *p++ = hex_chars[sample & 0xF];

        *p++ = '"';
    }

    // Footer
    *p++ = ']';
    *p++ = ' ';
    *p = '\\0';

    return p - buf;
}
```

Risultato: ~60% più veloce!

Benchmark e Misurazione {#benchmark}

Come Misurare le Performance

cpp

```
void benchmark_function() {
    const int iterations = 1000;
    char buffer[1024];
    uint32_t samples[100];

    // Inizializza dati test
    for (int i = 0; i < 100; i++) {
        samples[i] = 0x123456 + i;
    }

    // Test versione originale
    uint32_t start = esp_timer_get_time();
    for (int i = 0; i < iterations; i++) {
        serialize_json(buffer, samples, 100);
    }
    uint32_t time_original = esp_timer_get_time() - start;

    // Test versione ottimizzata
    start = esp_timer_get_time();
    for (int i = 0; i < iterations; i++) {
        serialize_json_optimized(buffer, samples, 100);
    }
    uint32_t time_optimized = esp_timer_get_time() - start;

    // Report
    Serial.printf("Originale:  %d µs/call\n", time_original / iterations);
    Serial.printf("Ottimizzata: %d µs/call\n", time_optimized / iterations);
    Serial.printf("Speedup:    %.2fx\n", (float)time_original / time_optimized);
    Serial.printf("Risparmio:   %d µs/call\n", (time_original - time_optimized) / iterations);
}
```

Metriche Importanti

- **Latenza:** Tempo per singola operazione
- **Throughput:** Operazioni per secondo
- **Jitter:** Variabilità nei tempi
- **CPU usage:** % utilizzo processore

✅ Best Practices {#best-practices}

📋 Checklist Ottimizzazioni

🔍 Analisi Prima

- ☐ Profile il codice (trova i bottleneck reali)
- ☐ Misura baseline performance
- ☐ Identifica hot paths (codice eseguito spesso)
- ☐ Verifica memory access patterns

Applica Ottimizzazioni

- ☐ Register variables per loop critici
- ☐ Branch elimination dove possibile
- ☐ Word-aligned memory access
- ☐ Lookup tables per calcoli ripetuti
- ☐ Loop unrolling per loop brevi
- ☐ Pre-calcolo di costanti

Test e Validazione





- ☐ Benchmark prima/dopo
- ☐ Test funzionalità (no regressioni)
- ☐ Stress test per stabilità
- ☐ Verifica su hardware target

Errori Comuni da Evitare

1. **Over-optimization:** Non ottimizzare codice che non è bottleneck
2. **Premature optimization:** Prima fai funzionare, poi ottimizza
3. **Sacrificare readability:** Codice illeggibile = bug futuri
4. **Ignorare il compilatore:** GCC è già molto bravo
5. **Non misurare:** "Sento che è più veloce" \neq realmente più veloce

Quando Ottimizzare

Priorità di ottimizzazione:

- | | |
|--|---|
| 1. Algoritmi ($O(n^2) \rightarrow O(n)$) |  Massimo impatto |
| 2. Hot paths (codice eseguito spesso) |  Alto impatto |
| 3. Memory access patterns |  Medio impatto |
| 4. Micro-ottimizzazioni |  Basso impatto |

Strumenti Utili

Profiling

cpp

```
#define PROFILE_START(name) uint32_t start_##name = esp_timer_get_time()
#define PROFILE_END(name) Serial.printf(#name ": %d μs\n", esp_timer_get_time() - start_##name)

// Uso:
PROFILE_START(json_serialization);
serialize_json(buffer, data, count);
PROFILE_END(json_serialization);
```

Memory Analysis

cpp

```
void print_memory_info() {
    Serial.printf("Free heap: %d bytes\n", esp_get_free_heap_size());
    Serial.printf("Min free heap: %d bytes\n", esp_get_minimum_free_heap_size());
    Serial.printf("Stack high water mark: %d\n", uxTaskGetStackHighWaterMark(NULL));
}
```

🎓 Riepilogo per Neofiti





Le 5 Regole d'Oro

1. 🎯 **Misura prima di ottimizzare**
"Non puoi migliorare ciò che non misuri"
2. 🔥 **Ottimizza i hot paths**
80% del tempo è speso nel 20% del codice
3. 💾 **Mantieni i dati vicini alla CPU**
Registri > Cache > RAM > Storage
4. 🚫 **Elimina le decisioni dal loop**
Pre-calcola tutto quello che puoi
5. 📦 **Pensa in termini di parole, non byte**
Le CPU lavorano meglio con word-aligned data

Risultati Tipici

- **Register variables:** +5-15%
- **Branch elimination:** +10-25%
- **Word-aligned access:** +15-30%
- **Lookup tables:** +20-50%
- **Algorithm changes:** +50-300%

Per Approfondire

-  "Computer Systems: A Programmer's Perspective"
-  ARM Cortex-M optimization guides
-  ESP-IDF performance guides
-  Compiler optimization flags (-O2, -O3)

Questa dispensa fornisce le basi per ottimizzazioni efficaci su microcontrollori. Ricorda: prima fai funzionare il codice, poi rendilo veloce! 🚀