Manuale Completo: Ottimizzazione Filtri FIR per DSP Embedded

Indice

1. Principi Teorici
2. Analisi del Problema
3. Ottimizzazione 1: Decomposizione a Cascata
4. Ottimizzazione 2: Sfruttamento della Simmetria
5. Ottimizzazione 3: Coefficienti Precomputati
6. Ottimizzazione 4: Algoritmi Adattivi
7. Implementazione Pratica
8. Testing e Validazione
9. Performance Monitoring

Principi Teorici

Complessità Computazionale dei Filtri FIR

Filtro FIR Standard:

- Complessità: O(N) moltiplicazioni per campione
- Memoria: N coefficienti + N campioni delay
- Latenza: N/2 campioni (ritardo di gruppo)

Dove N = numero di taps

```
y[n] = Σ(k=0 to N-1) h[k] * x[n-k]
```

Problema della Crescita Quadratica

Per decimazioni elevate, il numero di taps cresce rapidamente:

- Decimazione 2x: ~11 taps
- Decimazione 10x: ~41 taps
- Decimazione 100x: ~91 taps
- Decimazione 150x: ~101 taps

Impatto Computazionale:

```
Operazioni/secondo = Sample_Rate × Num_Taps × Decimation_Factor
```

Esempio: 30 kSps, decimazione 100x, 91 taps = 273 MMAC/s

Analisi del Problema

Bottleneck Identificati

1. **Numero Eccessivo di Taps**
   - Formula conservativa del codice originale
   - Penalità eccessive per decimazioni dispari
2. **Calcolo Ridondante**
   - Coefficienti simmetrici calcolati due volte
   - Ricomputazione coefficienti per ogni cambio decimazione
3. **Architettura Monolitica**
   - Un singolo filtro per tutte le decimazioni
   - Non sfrutta la decomponibilità matematica

Analisi Matematica della Decomposizione

Teorema: Un filtro con decimazione D può essere decomposto in cascata:

$$H(z) = H_1(z) \times H_2(z) \times ... \times H_k(z)$$
$$\text{dove } D = D_1 \times D_2 \times ... \times D_k$$

Vantaggio: Se $N_1 + N_2 + ... + N_k < N\_originale$, si ottiene risparmio computazionale.

Ottimizzazione 1: Decomposizione a Cascata

Principi di Design

Fattorizzazione Ottimale:

1. Privilegiare fattori pari (2, 4, 8) - più efficienti
2. Evitare fattori primi grandi
3. Bilanciare numero di stadi vs taps per stadio

Algoritmo di Decomposizione

```cpp
void decomposeDecimation(uint16_t totalDecimation, vector<uint16_t>& factors) {
    uint16_t remaining = totalDecimation;

    // Priorità 1: Fattori di 2 (più efficienti)
    while (remaining % 2 == 0 && remaining > 1) {
        uint16_t factor = min(remaining, 8);  // Max 8x per stadio
        if (factor > 4) factor = 4;  // Preferisci 4x
        factors.push_back(factor);
        remaining /= factor;
    }

    // Priorità 2: Fattori di 3
    while (remaining % 3 == 0 && remaining > 1) {
        factors.push_back(3);
        remaining /= 3;
    }

    // Priorità 3: Fattori di 5
    while (remaining % 5 == 0 && remaining > 1) {
        factors.push_back(5);
        remaining /= 5;
    }

    // Resto: fattore primo o composito rimanente
    if (remaining > 1) {
        factors.push_back(remaining);
    }
}
```

Esempi di Decomposizione

Decimazione 60x:

- Monolitico: 71 taps
- Cascata: 4×3×5 = (11 + 15 + 15) = 41 taps totali (-42%)

Decimazione 100x:

- Monolitico: 91 taps
- Cascata: 4×5×5 = (11 + 15 + 15) = 41 taps totali (-55%)

Decimazione 150x:

- Monolitico: 101 taps

- Cascata: 2×3×5×5 = (7 + 15 + 15 + 15) = 52 taps totali (-48%)

Implementazione Stadio Cascata

cpp

```cpp
struct CascadeStage {
    float coeffs[MAX_STAGE_TAPS];
    float delayLine[MAX_STAGE_TAPS];
    uint8_t numTaps;
    uint16_t decimationFactor;
    uint32_t sampleCounter;
    uint16_t delayIndex;

    bool process(float input, float& output) {
        // Inserisci in delay line
        delayLine[delayIndex] = input;
        delayIndex = (delayIndex + 1) % numTaps;

        // Calcola convoluzione
        float filtered = 0.0f;
        uint16_t idx = delayIndex;
        for (uint8_t i = 0; i < numTaps; i++) {
            idx = (idx == 0) ? (numTaps - 1) : (idx - 1);
            filtered += delayLine[idx] * coeffs[i];
        }

        // Decimazione
        sampleCounter++;
        if (sampleCounter >= decimationFactor) {
            sampleCounter = 0;
            output = filtered;
            return true;
        }
        return false;
    }
};
```

Calcolo Coefficienti per Stadio

Ogni stadio deve avere cutoff frequency appropriata:

cpp

```cpp
void calculateStageCoeffs(CascadeStage& stage, uint16_t stageFactor,
                          uint32_t inputSampleRate) {
    // Cutoff = 80% della nuova Nyquist frequency
    float cutoff = 0.8f * inputSampleRate / (2.0f * stageFactor);
    float omega_c = 2.0f * M_PI * cutoff / inputSampleRate;

    // Numero di taps basato su fattore di decimazione
    uint8_t taps;
    if (stageFactor == 2) taps = 11;
    else if (stageFactor <= 4) taps = 15;
    else if (stageFactor <= 8) taps = 21;
    else taps = 25;

    stage.numTaps = taps;

    // Genera coefficienti lowpass con finestra Hamming
    generateLowpassCoeffs(stage.coeffs, taps, omega_c);
}
```

Ottimizzazione 2: Sfruttamento della Simmetria

Principio Matematico

I filtri FIR a fase lineare hanno coefficienti simmetrici:

$$h[n] = h[N-1-n] \quad \text{per } n = 0,1,\ldots,N-1$$

Questo permette di riscrivere la convoluzione come:

$$y[n] = h[N/2] * x[n-N/2] + \Sigma(k=0 \text{ to } N/2-1)\ h[k] * (x[n-k] + x[n-(N-1-k)])$$

Implementazione Ottimizzata

```cpp
float processSymmetric(float input) {
    // Inserisci nuovo campione
    delayLine[delayIndex] = input;
    delayIndex = (delayIndex + 1) % numTaps;

    float output = 0.0f;
    uint16_t center = numTaps / 2;

    // Coefficiente centrale (se numTaps dispari)
    if (numTaps % 2 == 1) {
        uint16_t centerIdx = (delayIndex + center) % numTaps;
        output += delayLine[centerIdx] * coefficients[center];
    }

    // Coppie simmetriche
    uint16_t pairs = numTaps / 2;
    for (uint16_t i = 0; i < pairs; i++) {
        uint16_t idx1 = (delayIndex + i) % numTaps;
        uint16_t idx2 = (delayIndex + numTaps - 1 - i) % numTaps;

        // Una moltiplicazione, due campioni
        output += coefficients[i] * (delayLine[idx1] + delayLine[idx2]);
    }

    return output;
}
```

Analisi dei Benefici

Moltiplicazioni Risparmiate:

- Filtro normale: N moltiplicazioni
- Filtro simmetrico: N/2 + 1 moltiplicazioni (se N dispari)
- Risparmio: ~50% per N grandi

Esempio:

- 91 taps: 91 → 46 moltiplicazioni (-49%)
- 71 taps: 71 → 36 moltiplicazioni (-49%)

Ottimizzazione 3: Coefficienti Precomputati

Strategia di Precomputazione

1. **Identificare Decimazioni Comuni**
   - 2x, 3x, 4x, 5x, 8x, 10x (fattori base)
   - Calcolare coefficienti offline con precisione elevata

2. **Generazione Offline**

```matlab
% MATLAB/Octave per generazione coefficienti
function coeffs = generateOptimalFIR(decimation, taps)
    Fs = 30000;  % Sample rate
    Fc = 0.8 * Fs / (2 * decimation);  % Cutoff frequency

    % Usa Parks-McClellan per coefficienti ottimali
    coeffs = firpm(taps-1, [0 Fc Fc*1.25 Fs/2]/(Fs/2), [1 1 0 0]);
end
```

3. **Archiviazione Efficiente**

```cpp
// Lookup table per coefficienti comuni
struct PrecomputedFilter {
    uint8_t decimation;
    uint8_t numTaps;
    const float* coefficients;
};

const float COEFFS_2X_11TAPS[] = {
    -0.0123f, 0.0234f, -0.0456f, 0.0891f, -0.1789f,
    0.6234f, -0.1789f, 0.0891f, -0.0456f, 0.0234f, -0.0123f
};

const PrecomputedFilter FILTER_TABLE[] = {
    {2, 11, COEFFS_2X_11TAPS},
    {3, 15, COEFFS_3X_15TAPS},
    {4, 15, COEFFS_4X_15TAPS},
    {5, 21, COEFFS_5X_21TAPS},
    // ...
};
```

Lookup e Fallback

cpp

```cpp
bool loadPrecomputedCoeffs(uint16_t decimation, uint16_t& taps, float* coeffs) {
    for (const auto& filter : FILTER_TABLE) {
        if (filter.decimation == decimation) {
            taps = filter.numTaps;
            memcpy(coeffs, filter.coefficients, taps * sizeof(float));
            return true;
        }
    }
    return false;  // Usa calcolo runtime
}
```

Benefici

1. **Tempo di Setup:** Eliminazione completa calcolo coefficienti
2. **Qualità:** Coefficienti ottimizzati con algoritmi avanzati
3. **Consistenza:** Risultati identici e ripetibili
4. **Memoria:** Coefficienti in ROM/Flash (non RAM)

Ottimizzazione 4: Algoritmi Adattivi

Selezione Dinamica dell'Algoritmo

```cpp
enum FilterMode {
    PASSTHROUGH,      // Decimation = 1
    PRECOMPUTED,      // Coefficienti in lookup table
    SYMMETRIC_SINGLE, // Filtro singolo con simmetria
    CASCADE,          // Cascata di stadi
    POLYPHASE         // Implementazione polifase
};

FilterMode selectOptimalMode(uint16_t decimation, uint16_t& estimatedTaps) {
    if (decimation == 1) return PASSTHROUGH;

    // Controlla se esistono coefficienti precomputati
    if (hasPrecomputedCoeffs(decimation)) {
        estimatedTaps = getPrecomputedTaps(decimation);
        return PRECOMPUTED;
    }

    // Calcola taps necessari per filtro singolo
    uint16_t singleTaps = calculateMinimumTaps(decimation);

    // Calcola taps per implementazione cascata
    vector<uint16_t> factors;
    decomposeDecimation(decimation, factors);
    uint16_t cascadeTaps = calculateCascadeTaps(factors);

    // Scegli implementazione più efficiente
    if (cascadeTaps < singleTaps * 0.7f) {
        estimatedTaps = cascadeTaps;
        return CASCADE;
    } else if (singleTaps > 25) {
        estimatedTaps = singleTaps;
        return SYMMETRIC_SINGLE;
    } else {
        estimatedTaps = singleTaps;
        return PRECOMPUTED;  // Fallback
    }
}
```

Calcolo Adattivo dei Taps

Formula intelligente che bilancia qualità vs performance:

cpp

```cpp
uint16_t calculateAdaptiveTaps(uint16_t decimation) {
    // Base empirica con correzioni
    uint16_t baseTaps;

    if (decimation <= 2) {
        baseTaps = 9;   // Ridotto da 11
    } else if (decimation <= 5) {
        baseTaps = 13 + (decimation - 2) * 2;  // Crescita più lenta
    } else if (decimation <= 15) {
        baseTaps = 19 + (decimation - 5) * 1.5f;
    } else if (decimation <= 50) {
        baseTaps = 34 + (decimation - 15) * 0.8f;
    } else {
        baseTaps = 62 + (decimation - 50) * 0.4f;
    }

    // Penalità ridotta per decimazioni dispari
    if (decimation > 2 && decimation % 2 == 1) {
        uint16_t penalty = 4;  // Ridotto da 10-20
        if (decimation > 15) penalty = 2;
        baseTaps += penalty;
    }

    // Assicura numero dispari
    if (baseTaps % 2 == 0) baseTaps++;

    return min(baseTaps, MAX_TAPS);
}
```

Implementazione Pratica

Architettura del Sistema Ottimizzato

cpp

```cpp
class OptimizedAdaptiveFIR {
private:
    FilterMode currentMode;

    // Implementazioni specializzate
    SingleStageFilter singleStage;
    CascadeFilter cascadeFilter;

    // Statistiche performance
    uint32_t cyclesPerSample;
    uint32_t maxCycles;

public:
    bool processSample(uint32_t input, uint32_t& output) {
        uint32_t startCycles = getCycles();

        bool hasOutput = false;
        switch (currentMode) {
            case PASSTHROUGH:
                output = input;
                hasOutput = true;
                break;

            case PRECOMPUTED:
            case SYMMETRIC_SINGLE:
                hasOutput = singleStage.process(input, output);
                break;

            case CASCADE:
                hasOutput = cascadeFilter.process(input, output);
                break;
        }

        // Monitoring performance
        uint32_t cycles = getCycles() - startCycles;
        updatePerformanceStats(cycles);

        return hasOutput;
    }
};
```

Sistema di Configurazione Automatico

cpp

```cpp
void autoConfigureFilter(uint16_t decimation) {
    printf("Auto-configuring for decimation %dx...\n", decimation);

    // Analizza opzioni disponibili
    FilterMode mode = selectOptimalMode(decimation, estimatedTaps);

    switch (mode) {
        case CASCADE: {
            vector<uint16_t> factors;
            decomposeDecimation(decimation, factors);

            printf("Selected CASCADE: ");
            for (auto f : factors) printf("%dx ", f);
            printf("(total %d taps)\n", estimatedTaps);

            setupCascadeFilter(factors);
            break;
        }

        case SYMMETRIC_SINGLE:
            printf("Selected SYMMETRIC: %d taps\n", estimatedTaps);
            setupSingleFilter(decimation, true);
            break;

        case PRECOMPUTED:
            printf("Selected PRECOMPUTED: %d taps\n", estimatedTaps);
            loadPrecomputedFilter(decimation);
            break;
    }

    currentMode = mode;
}
```

Testing e Validazione

Test di Correttezza

1. **Risposta in Frequenza**

cpp

```cpp
void testFrequencyResponse() {
    for (float freq = 100; freq < 15000; freq += 100) {
        float response = getFrequencyResponse(freq);
        float expectedResponse = calculateIdealResponse(freq);

        float error = abs(response - expectedResponse);
        assert(error < 0.05f);  // 5% tolerance
    }
}
```

2. **Test di Aliasing**

cpp

```cpp
void testAntiAliasing(uint16_t decimation) {
    float nyquist = SAMPLE_RATE / (2.0f * decimation);

    // Test frequenze oltre Nyquist
    for (float freq = nyquist * 1.1f; freq < nyquist * 3.0f; freq += 100) {
        float response = getFrequencyResponse(freq);
        assert(response < -40.0f);  // Attenuazione minima 40dB
    }
}
```

Test di Performance

cpp

```cpp
struct PerformanceTest {
    uint16_t decimation;
    FilterMode mode;
    uint32_t avgCycles;
    uint32_t maxCycles;
    uint16_t taps;
    float throughputMSps;
};

void benchmarkAllModes() {
    vector<uint16_t> testDecimations = {2, 3, 5, 10, 15, 30, 50, 100, 150};

    for (auto dec : testDecimations) {
        for (auto mode : {SYMMETRIC_SINGLE, CASCADE}) {
            PerformanceTest result = runPerformanceTest(dec, mode);
            printf("Dec %3d | Mode %d | Taps %3d | Cycles %5d | MSPS %.1f\n",
                   result.decimation, result.mode, result.taps,
                   result.avgCycles, result.throughputMSps);
        }
    }
}
```

Validazione Audio

cpp

```cpp
void validateAudioQuality() {
    // Genera segnale test multi-tone
    vector<float> testSignal = generateMultiTone({440, 1000, 5000, 8000});

    // Processa con filtro ottimizzato
    vector<float> filteredSignal;
    for (auto sample : testSignal) {
        uint32_t output;
        if (processSample(floatToInput(sample), output)) {
            filteredSignal.push_back(inputToFloat(output));
        }
    }

    // Analizza THD+N, SNR, dinamica
    AudioMetrics metrics = analyzeAudioQuality(filteredSignal);

    assert(metrics.thd_n < -80.0f);   // THD+N < -80dB
    assert(metrics.snr > 100.0f);     // SNR > 100dB
    assert(metrics.dynamic > 120.0f); // Range dinamico > 120dB
}
```

Performance Monitoring

Metriche in Tempo Reale

cpp

```cpp
class PerformanceMonitor {
private:
    uint32_t totalCycles;
    uint32_t sampleCount;
    uint32_t peakCycles;
    float avgCycles;

public:
    void recordSample(uint32_t cycles) {
        totalCycles += cycles;
        sampleCount++;
        peakCycles = max(peakCycles, cycles);

        // Update running average
        avgCycles = (float)totalCycles / sampleCount;
    }

    void printStats() {
        printf("Performance Stats:\n");
        printf("  Avg cycles/sample: %.1f\n", avgCycles);
        printf("  Peak cycles: %lu\n", peakCycles);
        printf("  CPU usage: %.1f%%\n",
               100.0f * avgCycles / CYCLES_PER_SAMPLE_BUDGET);
    }
};
```

Ottimizzazione Dinamica

cpp

```cpp
void adaptiveOptimization() {
    static uint32_t lastOptimization = 0;
    uint32_t currentTime = getSystemTime();

    // Ottimizza ogni 10 secondi
    if (currentTime - lastOptimization > 10000) {
        PerformanceStats stats = monitor.getStats();

        if (stats.avgCycles > CYCLES_BUDGET * 0.8f) {
            // Performance critica: riduci qualità
            printf("High CPU load detected, reducing filter quality\n");
            reduceFilterComplexity();
        } else if (stats.avgCycles < CYCLES_BUDGET * 0.5f) {
            // Margine disponibile: aumenta qualità
            printf("CPU headroom available, improving filter quality\n");
            increaseFilterQuality();
        }

        lastOptimization = currentTime;
    }
}
```

Profilazione Dettagliata

cpp

```cpp
#ifdef ENABLE_PROFILING
#define PROFILE_START(name) uint32_t start_##name = getCycles()
#define PROFILE_END(name) profileData[#name] += getCycles() - start_##name

void profileFilterOperations() {
    PROFILE_START(input_conversion);
    float floatInput = convertInputToFloat(inputSample);
    PROFILE_END(input_conversion);

    PROFILE_START(fir_convolution);
    float filtered = doConvolution(floatInput);
    PROFILE_END(fir_convolution);

    PROFILE_START(decimation_check);
    bool hasOutput = checkDecimation();
    PROFILE_END(decimation_check);

    PROFILE_START(output_conversion);
    uint32_t output = convertFloatToOutput(filtered);
    PROFILE_END(output_conversion);
}
#endif
```

Conclusioni e Best Practices

Raccomandazioni di Implementazione

1. **Inizia con Profilazione:** Misura prima di ottimizzare
2. **Implementa Gradualmente:** Una ottimizzazione alla volta
3. **Mantieni Backward Compatibility:** API esistente deve funzionare
4. **Testing Rigoroso:** Ogni modifica deve passare tutti i test
5. **Documentazione:** Spiega chiaramente ogni ottimizzazione

Tabella Riassuntiva Ottimizzazioni

Ottimizzazione	Beneficio	Complessità	Quando Usare
Cascata	40-70% meno cicli	Media	Decimazioni >30x
Simmetria	50% meno moltiplicazioni	Bassa	Filtri >25 taps
Precomputati	Zero calcolo coefficienti	Bassa	Decimazioni comuni
Adattivo	20-40% meno taps	Media	Tutte le situazioni

Roadmap di Sviluppo

Fase 1 (Immediate):

- Implementa algoritmo taps adattivo
- Aggiungi coefficienti precomputati per 2x, 3x, 5x, 10x

Fase 2 (Breve Termine):

- Implementa ottimizzazione simmetria
- Sistema di profilazione base

Fase 3 (Medio Termine):

- Decomposizione a cascata completa
- Selezione automatica algoritmo

Fase 4 (Lungo Termine):

- Ottimizzazione dinamica
- Implementazione SIMD/DSP hardware

Questo manuale fornisce tutti gli strumenti teorici e pratici per implementare efficacemente le ottimizzazioni del filtro FIR, mantenendo alta qualità audio e riducendo significativamente il carico computazionale.