

```
or object to mirror  
mirror_mod.mirror_object  
operation == "MIRROR_X":  
mirror_mod.use_x = True  
mirror_mod.use_y = False  
mirror_mod.use_z = False  
operation == "MIRROR_Y":  
mirror_mod.use_x = False  
mirror_mod.use_y = True  
mirror_mod.use_z = False  
operation == "MIRROR_Z":  
mirror_mod.use_x = False  
mirror_mod.use_y = False  
mirror_mod.use_z = True
```

```
#selection at the end -add  
obj.select= 1  
modifier_ob.select=1  
context.scene.objects.active  
("Selected" + str(modifier_ob.name))  
mirror_ob.select= 0
```

OPTIMIZACIÓN DE CÓDIGO EN C++

```
Y mirror_mod.mirror_x  
object.mirror_mod.mirror_x  
error X"
```

LIBRERÍAS ESPECIALIZADAS EN PROCESAMIENTO DE SEÑALES

Excelente manejo de arrays

- **Eigen:** Es una biblioteca de C++ para álgebra lineal, matrices y vectores, soluciones numéricas y más. Es conocida por su rendimiento y es ampliamente utilizada en aplicaciones que requieren operaciones matemáticas intensivas. [Eigen](#)
- **Armadillo:** Se basa en LAPACK (la librería utilizada por numpy)
 - [Armadillo: C++ library for linear algebra & scientific computing \(sourceforge.net\)](#)
- **Librosa:** Original de python, existe implementación en CPP ([GitHub - ewan-xu/LibrosaCpp: LibrosaCpp is a c++ implementation of librosa to compute short-time fourier transform coefficients, mel spectrogram or mfcc](#))
- **FFTW:** Transformada Rápida de Fourier (FFT) en una o más dimensiones, de tamaño arbitrario y de datos reales o complejos. [FFTW Home Page](#)
- **Scipy:** Original de python no tiene equivalente en C++



TIPS PARA OPTIMIZAR CÓDIGO

Disculpe por la confusión. Aquí hay algunas técnicas para optimizar manualmente la velocidad y el tamaño del código en Visual Studio Code:

1. **Reduzca la complejidad del código:** La complejidad del código puede afectar negativamente la velocidad y el tamaño del código. Por lo tanto, es importante reducir la complejidad del código tanto como sea posible. Puede hacerlo dividiendo el código en funciones más pequeñas y reutilizables, utilizando estructuras de datos más eficientes y eliminando el código redundante.
2. **Optimice los bucles:** Los bucles son una parte importante del código, pero también pueden ser una fuente de ineficiencia. Para optimizar los bucles, puede intentar reducir el número de iteraciones, utilizar operaciones más eficientes y evitar la creación de objetos innecesarios.
3. **Utilice variables locales:** Las variables locales son más rápidas que las variables globales. Por lo tanto, es importante utilizar variables locales siempre que sea posible.
4. **Evite las operaciones costosas:** Algunas operaciones son más costosas que otras. Por ejemplo, las operaciones de cadena son más costosas que las operaciones numéricas. Por lo tanto, es importante evitar las operaciones costosas siempre que sea posible.
5. **Utilice la memoria de manera eficiente:** La memoria es un recurso limitado. Por lo tanto, es importante utilizar la memoria de manera eficiente. Puede hacerlo utilizando estructuras de datos más eficientes y liberando la memoria no utilizada.



BUCLES FOR BASADOS EN CONTADOR Y EN RANGO

Bucle for tradicional basado en contador

```
for (int i = 0; i < N; i++) {  
    // Procesamiento de la señal  
}
```

Bucle basado en rango

El bucle for basado en rango puede ser más eficiente cuando se itera sobre contenedores, ya que no requiere el acceso al índice.

Además, el bucle for basado en rango puede hacer que el código sea más legible y menos propenso a errores, especialmente cuando se trabaja con contenedores complejos.

```
#include <iostream>  
#include <list>  
  
int main() {  
    // Crear una lista de enteros  
    std::list<int> lista = {1, 2, 3, 4, 5};  
  
    // Iterar sobre la lista con un bucle for basado en rangos  
    for (const int& valor : lista) {  
        std::cout << valor << " ";  
    }  
  
    std::cout << std::endl;  
    return 0;  
}
```



OPTIMIZACIÓN DE BUCLES

- Para optimizar los bucles en C++, se pueden aplicar varias técnicas. Aquí hay algunas de ellas:
 - **Expansión de bucles:** Esta técnica implica desenrollar el bucle para reducir la cantidad de iteraciones necesarias. Esto puede mejorar el rendimiento al reducir la cantidad de saltos de bucle y la cantidad de veces que se ejecuta el código de comprobación de bucle.
 - **Reducción de fuerza:** Esta técnica implica reemplazar una operación costosa con una operación menos costosa. Por ejemplo, se puede reemplazar una multiplicación por una suma o una resta.
 - **Eliminación de subexpresiones comunes:** Esta técnica implica eliminar subexpresiones comunes del código para reducir la cantidad de cálculos necesarios. Por ejemplo, si una subexpresión se calcula varias veces en un bucle, se puede calcular una vez fuera del bucle y almacenar el resultado en una variable.
 - **Optimización de la memoria caché:** Esta técnica implica organizar el código para que los datos se almacenen en la memoria caché de manera más eficiente. Por ejemplo, se pueden reorganizar las estructuras de datos para que los datos que se utilizan juntos se almacenen juntos en la memoria caché.
 - **Vectorización:** Esta técnica implica utilizar instrucciones vectoriales para procesar varios elementos de datos a la vez. Esto puede mejorar el rendimiento al reducir la cantidad de ciclos de reloj necesarios para procesar los datos.



IMPLEMENTACIÓN BÁSICA DE UN FILTRO FIR

```
// Aplicar el filtro FIR con doble bucle
for (size_t n = 0; n < tamanoX; ++n) {
    for (size_t m = 0; m < tamanoH; ++m) {
        y[n + m] += x[n] * h[m];
    }
}
```

Este código implementa un filtro FIR con doble bucle que utiliza una señal de entrada x y una respuesta al impulso h para producir una señal de salida y . El primer bucle se utiliza para recorrer la señal de entrada, mientras que el segundo bucle se utiliza para recorrer la respuesta al impulso. El resultado final se almacena en la señal de salida y .



LOOP UNROLLING

```
for (int i = 0; i < N; i++) {  
    // Procesamiento de la señal  
}
```

```
for (int i = 0; i < N; i += 4) {  
    // Procesamiento de la señal para i  
    // Procesamiento de la señal para i+1  
    // Procesamiento de la señal para i+2  
    // Procesamiento de la señal para i+3  
}
```

En lugar de iterar de uno en uno, procesamos varios elementos cada vez:

- ❏ Reduce el número de veces que se evalúa la condición del bucle
- ❏ Permite un mejor uso de la cache del procesador
- ❏ Proporciona más oportunidades al procesador para realizar optimizaciones



PIPELINE SOFTWARE

```
// Supongamos que tenemos una función que procesa un elemento de datos
void procesarDato(int& dato) {
    // Realizar operaciones sobre 'dato'
}

// Y otra función que prepara el siguiente dato
int prepararSiguienteDato(int indice) {
    // Preparar y retornar el siguiente dato basado en 'indice'
    return indice * 2; // Ejemplo simple
}

// Usando un pipeline en un bucle for
for (int i = 0; i < N; i++) {
    int dato = prepararSiguienteDato(i);
    procesarDato(dato);
}
```

Se puede crear un pipeline descomponiendo el bucle en tareas independientes que puedan ser paralelizadas



SOBRECARGA (OVERHEAD) POR LLAMADAS A FUNCIÓN

Razones

1. **Paso de parámetros:** Cada vez que se llama a una función, los parámetros deben ser pasados a ella, lo cual puede implicar copiar valores o referencias, dependiendo de cómo se haya definido la función.
2. **Creación y destrucción del marco de pila:** Al entrar y salir de una función, se crea y destruye un marco de pila (stack frame) para almacenar las variables locales y la dirección de retorno.
3. **Saltos en el flujo de ejecución:** Las llamadas a funciones implican saltos en el flujo de ejecución del programa, lo que puede afectar la predicción de saltos y la eficiencia del pipeline del procesador.

Para minimizar el sobrecoste:

1. **Uso de funciones inline:** Las funciones `inline` sugieren al compilador que inserte el cuerpo de la función en el punto de llamada, eliminando el overhead de la llamada a función. Sin embargo, el uso excesivo de funciones `inline` puede aumentar el tamaño del código ejecutable.
2. **Paso de parámetros por referencia:** En lugar de pasar parámetros por valor, lo cual implica una copia, se pueden pasar por referencia para evitar la sobrecarga de la copia.
3. **Optimizaciones del compilador:** Los compiladores modernos realizan optimizaciones que pueden reducir o eliminar el overhead de las llamadas a funciones, especialmente si el cuerpo de la función es pequeño y se llama frecuentemente.
4. Si el cuerpo de la función es simple y se utiliza dentro de un bucle, considera **realizar las operaciones directamente en el bucle en lugar de llamar a una función.**




EFICIENCIA DE BUCLES FOR BASADO EN CONTADORES

- **Acceso secuencial:** Los elementos se acceden en un orden secuencial, lo que es beneficioso para la localidad espacial y puede resultar en menos fallos de caché.
- **Predicción de saltos:** Los procesadores modernos son muy buenos en predecir los saltos que ocurren en bucles con contadores, lo que puede mejorar el rendimiento al reducir los ciclos de CPU desperdiciados.
- **Vectorización:** Los compiladores pueden optimizar bucles con contadores para utilizar instrucciones SIMD (Single Instruction, Multiple Data), procesando múltiples elementos del arreglo en paralelo.





DIRECTIVAS (PRAGMAS)



Indicaciones al
compilador
para optimizar
el código en
bucles

DESENROLLADO DE BUCLES

Incrementar el cuerpo del bucle para reducir la cantidad de iteraciones y, por lo tanto, la sobrecarga de la comprobación de la condición del bucle y el incremento de la variable de control.

```
// Sin pragma unroll  
for (int i = 0; i < 4; ++i) {  
    // Código del bucle  
}
```

```
// Con pragma unroll  
#pragma unroll  
for (int i = 0; i < 4; ++i) {  
    // Código del bucle  
}
```



COLAPSO DE BUCLES (OPENMP)

es una directiva de OpenMP que permite combinar múltiples bucles anidados en un solo bucle paralelo con un rango de iteración más grande.

Combina bucles anidados en un solo bucle con un rango de iteración más grande para reducir la sobrecarga de la gestión de múltiples bucles anidados.

los bucles for que iteran sobre i y j se colapsan en un solo bucle paralelo, permitiendo que las iteraciones se distribuyan entre los diferentes hilos de ejecución disponibles. Esto puede ser especialmente útil cuando se trabaja con matrices grandes o conjuntos de datos que se benefician de la paralelización.

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < N; ++i) {
    for (int j = 0; j < M; ++j) {
        // Código que trabaja con elementos [i][j]
    }
}
```



OPTIMIZE

La sintaxis es

```
#pragma optimize( "[optimization-list]", {on | off} )
```

```
#pragma optimize("símbolo de optimización", on)
// Código a optimizar aquí
#pragma optimize("", on)
```

/O1: Optimiza para el tamaño del código (Minimizar tamaño).

/O2: Optimiza para la velocidad (Maximizar velocidad).

/Ox: Usa la máxima optimización (excepto /Og).

/Ot: Favorece la velocidad sobre el tamaño del código.

/Os: Favorece el tamaño sobre la velocidad del código.

/Oy: Omite la información del marco de pila (frame pointer).

/Ob: Expande o no las llamadas inline.

/Oi: Genera instrucciones intrínsecas.

/Og: Usa optimizaciones globales.

/Oa: Asume no aliasing en las funciones del programa.

/Ow: Asume no aliasing dentro de una función.

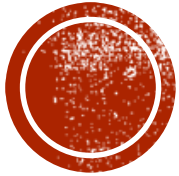
/Oy-: No omite el marco de pila (frame pointer).





OTROS PRAGMAS

- **#pragma loop(hint_parallel(n)):** Indica al compilador que el bucle puede ejecutarse en paralelo con 'n' hilos.
- **#pragma ivdep:** Informa al compilador de que las iteraciones del bucle son independientes y pueden vectorizarse sin preocuparse por las dependencias de memoria.
- **#pragma vector always:** Fuerza la vectorización del bucle, incluso si el compilador determina que puede no ser beneficioso.
- **#pragma vector novect:** Impide la vectorización del bucle.
- **#pragma vector aligned:** Asegura que los datos están alineados para la vectorización.
- **#pragma omp parallel for:** Utiliza OpenMP para paralelizar el bucle.



INTRINSICS

[Intrinsics X86](#)

[Intrinsics AMD](#)



SUMA SATURADA

- **`_mm_adds_epi8`**: Suma saturada de enteros de 8 bits.
- **`_mm_adds_epu16`**: Suma saturada de enteros de 16 bits.
- **`_mm_adds_epu8`**: Suma saturada de enteros de 8 bits sin signo.
- **`_mm_adds_epu16`**: Suma saturada de enteros de 16 bits sin signo.



PRODUCTO ESCALAR

- **_mm_dp_ps: Producto escalar de números de punto flotante de precisión simple.**
- **_mm256_dp_ps: Producto escalar de números de punto flotante de precisión simple para vectores de 256 bits.**



OPERACIONES DE BUCLE Y VECTORIZACIÓN

- **`_mm_prefetch`**: Prefetch de datos en la caché antes de que sean necesarios.
- **`_mm_stream_load_si128`**: Carga un vector entero de 128 bits utilizando una operación de streaming no temporal.
- **`_mm256_load_si256`**: Carga un vector entero de 256 bits desde la memoria.
- **`_mm256_store_si256`**: Almacena un vector entero de 256 bits en la memoria. **`_mm_dp_ps`**: Producto escalar de números de punto flotante de precisión simple.
- **`_mm256_dp_ps`**: Producto escalar de números de punto flotante de precisión simple para vectores de 256 bits.



MANIPULACIÓN DE BITS

- **`_bit_scan_forward`**: Escanea el bit menos significativo establecido.
- **`_mm_extract_epi32`**: Extrae un entero de 32 bits de un vector de enteros de 128 bits.

EJEMPLO DE FILTRO FIR CON INTRINSICS

```
void fir_filter(float* input, float* coeffs, float* output, int num_samples, int
num_coeffs) {
    __m256 coeff_vector, input_vector, result_vector, sum_vector;

    // Cargar los coeficientes del filtro en un vector de 256 bits (8 floats)
    coeff_vector = _mm256_loadu_ps(coeffs);

    for (int i = 0; i < num_samples; i++) {
        sum_vector = _mm256_setzero_ps(); // Inicializar la suma a cero

        for (int j = 0; j < num_coeffs; j += 8) {
            // Cargar 8 muestras de entrada en un vector de 256 bits
            input_vector = _mm256_loadu_ps(&input[i + j]);

            // Realizar el producto escalar entre las muestras de entrada y los
coeficientes
            result_vector = _mm256_dp_ps(input_vector, coeff_vector, 0xf1);

            // Sumar los resultados parciales
            sum_vector = _mm256_add_ps(sum_vector, result_vector);
        }

        // Almacenar el resultado en el buffer de salida
        output[i] = _mm256_cvtss_f32(sum_vector); // Convertir el primer elemento del
vector a float
    }
}
```

_mm256_loadu_ps: Carga un vector de 256 bits con números de punto flotante de precisión simple desde la memoria en un registro YMM sin requerir que la dirección de memoria esté alineada1.

_mm256_setzero_ps: Inicializa un registro YMM con todos los elementos a cero. Es útil para establecer el punto de partida para acumular sumas en operaciones vectoriales1.

_mm256_dp_ps: Realiza el producto escalar de dos vectores de 256 bits de números de punto flotante de precisión simple y suma los pares de productos. El resultado se almacena en todas las posiciones del vector resultante o en una posición específica dependiendo del inmediato de control1.

_mm256_add_ps: Suma dos vectores de 256 bits de números de punto flotante de precisión simple y almacena el resultado en un registro YMM1.

_mm256_cvtss_f32: Convierte el primer elemento de un vector de 256 bits de números de punto flotante de precisión simple a un escalar de punto flotante de precisión simple (float). Se utiliza para extraer el resultado de una operación vectorial y almacenarlo en una variable de tipo float1.



AÑADIMOS RESTRICT

```
void fir_filter(float* __restrict input, float* __restrict  
coeffs, float* __restrict output, int num_samples, int  
num_coefs) {
```

es una indicación para el compilador de que los objetos apuntados por esos punteros no serán accedidos a través de otros punteros (es decir, no tienen alias). Esto permite al compilador realizar optimizaciones más agresivas al asumir que no hay dependencias de datos a través de esos punteros



APROVECHANDO EL ANCHO DE BANDA

En C++, puedes aprovechar el ancho de banda y realizar operaciones en paralelo utilizando las extensiones SIMD (Single Instruction, Multiple Data) que proporcionan los procesadores modernos. SIMD permite realizar la misma operación en múltiples datos simultáneamente, lo que puede ser muy eficiente cuando se trabaja con grandes cantidades de datos.

Para hacer esto en C++, puedes utilizar intrínsecos específicos del procesador o bibliotecas de vectorización que abstraen estas operaciones a un nivel más alto. Los intrínsecos son funciones especiales que mapean directamente a instrucciones de máquina SIMD, como SSE (Streaming SIMD Extensions), AVX (Advanced Vector Extensions) y NEON para ARM.



APROVECHANDO EL ANCHO DE BANDA

```
#include <emmintrin.h> // Incluir para funciones intrínsecas de Intel SSE2

void sum_arrays(const char* array1, const char* array2, char* sum, int size) {
    for (int i = 0; i < size; i += 16) { // Procesar 16 enteros de 8 bits a la vez
        __m128i v1 = _mm_loadu_si128((__m128i*)&array1[i]);
        __m128i v2 = _mm_loadu_si128((__m128i*)&array2[i]);
        __m128i v3 = _mm_adds_epi8(v1, v2); // Suma saturada de enteros de 8 bits
        _mm_storeu_si128((__m128i*)&sum[i], v3);
    }
}
```

`_mm_loadu_si128` carga 128 bits de datos desde la memoria en un registro XMM,

`_mm_adds_epi8` realiza una suma saturada de dos vectores de enteros de 8 bits,

`_mm_storeu_si128` almacena el resultado de nuevo en la memoria.



MAC

existen optimizaciones para la operación MAC (Multiply-Accumulate) en C++. Estas optimizaciones suelen venir en forma de funciones intrínsecas que permiten realizar múltiples operaciones de multiplicación y acumulación en una sola instrucción, aprovechando las capacidades SIMD (Single Instruction, Multiple Data) del procesador

En arquitecturas X86-64, por ejemplo, puedes utilizar las extensiones SIMD como SSE, AVX o AVX-512, que ofrecen intrínsecos específicos para operaciones MAC. Estos intrínsecos pueden realizar multiplicaciones y sumas de vectores de datos en paralelo, lo que puede resultar en un rendimiento significativamente mejorado para aplicaciones que requieren un gran número de estas operaciones, como el procesamiento de señales digitales o la computación científica.



MAC

```
#include <immintrin.h> // Incluir para funciones intrínsecas de Intel AVX2

void mac_operation(const float* a, const float* b, float* c, int size) {
    for (int i = 0; i < size; i += 8) { // Procesar 8 floats a la vez
        __m256 va = _mm256_loadu_ps(&a[i]);
        __m256 vb = _mm256_loadu_ps(&b[i]);
        __m256 vc = _mm256_loadu_ps(&c[i]);

        // Realizar la operación MAC
        __m256 result = _mm256_fmadd_ps(va, vb, vc); // va * vb + vc

        _mm256_storeu_ps(&c[i], result);
    }
}
```

`_mm256_fmadd_ps` es una función intrínseca que realiza la operación Fused Multiply-Add (FMA), que es una forma de operación MAC.

Multiplica cada elemento del primer vector (va) con el elemento correspondiente del segundo vector (vb) y suma el resultado al elemento correspondiente del tercer vector (vc).



COMPARATIVA DE SUMA Y SUMA ACUMULADA

`_mm256_add_ps` es una función intrínseca que realiza la suma de dos vectores de 256 bits de números de punto flotante de precisión simple1. Es una operación simple que solo realiza la adición.

`_mm256_fmadd_ps` es una función intrínseca que realiza una operación de Multiply-Add Fused (FMA), es decir, multiplica dos vectores de números de punto flotante y suma el resultado a un tercer vector, todo en una sola instrucción1.

Esta operación es más compleja que una simple suma, ya que combina la multiplicación y la suma, pero puede ser más eficiente en términos de rendimiento porque reduce la latencia y el número de instrucciones necesarias para realizar ambas operaciones.



Implementación	Descripción	Ciclos de reloj (estimados)
1. Doble bucle tradicional	Utiliza bucles anidados sin optimizaciones específicas del hardware.	Más alto
2. Intrinsics y restrict	Utiliza intrínsecos y punteros restrict para optimizar la memoria y las operaciones.	Medio-Alto
3. Intrinsics para 4x16 bits y restrict	Utiliza intrínsecos para operaciones paralelas de 16 bits y punteros restrict.	Medio
4. Opción 3 con punteros restrict	Similar a la opción 3, pero con énfasis en la no-aliasing de punteros.	Medio
5. Intrinsics para 4x16 bits con MAC y restrict	Utiliza intrínsecos para operaciones paralelas de 16 bits y una operación MAC al final, con punteros restrict.	Más bajo

COMPARATIVA DIFERENTES TÉCNICAS DE OPTIMIZACIÓN

