DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Resolviendo la Ecuación de Onda mediante la Transformada de Fourier Acelerada

Optimización de simulador fisico utilizando SIMD

30 de Agosto de 2025 — Organización del Computador II

Integrante	LU	Correo electrónico
Polonuer, Joaquin	1612/21	jtpolonuer@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1. Introducción

1.1. La Ecuación de Onda

La ecuación de onda representa uno de los fenómenos físicos más fundamentales en la naturaleza, describiendo la propagación de perturbaciones en medios continuos. Desde ondas sonoras y electromagnéticas hasta vibraciones mecánicas, este modelo matemático encuentra aplicación en campos tan diversos como la acústica, la óptica, la sismología y la ingeniería estructural.

La ecuación de onda unidimensional se expresa como:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} \tag{1}$$

donde $u(x,t)$ representa el desplazamiento de la onda en el punto x y tiempo t, y c es la velocidad de propagación característica del medio. Esta ecuación en derivadas parciales de segundo orden describe fenómenos como:

- Vibraciones de cuerdas tensadas
- Propagación de ondas sonoras en tubos
- Ondas electromagnéticas en líneas de transmisión

Naturalmente, su extensión a dos dimensiones espaciales es:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = c^2 \nabla^2 u \tag{2}$$

Esta formulación bidimensional modela fenómenos como:

- Vibraciones de membranas (tambores, diafragmas)
- Ondas superficiales en líquidos
- Propagación de ondas sísmicas en planos
- Ondas electromagnéticas

1.2. La Transformada de Fourier

La Transformada de Fourier constituye una herramienta matemática fundamental para el análisis de fenómenos ondulatorios, permitiendo descomponer señales complejas en sus componentes frecuenciales básicas. Esta transformación resulta especialmente poderosa en el contexto de la resolución de ecuaciones diferenciales parciales.

La Transformada de Fourier continua de una función $f(x)$ se define como:

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(x) e^{-i\omega x} dx \tag{3}$$

y su transformada inversa:

$$f(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{f}(\omega) e^{i\omega x} d\omega \tag{4}$$

1.3. Resolución de la Ecuación de Onda mediante la Transformada de Fourier

La aplicación de la Transformada de Fourier a la ecuación de onda permite convertir la ecuación en derivadas parciales en una ecuación diferencial ordinaria, lo que simplifica notablemente su resolución. Para abordar la ecuación (1), se aprovechan propiedades fundamentales de la transformada de Fourier relacionadas con las derivadas:

Propiedad 1: La transformada de Fourier de la derivada n-ésima de una función es igual a la multiplicación por $(i\omega)^n$ de la transformada de la función original. En particular, para $n = 1$ se obtiene el caso de la derivada simple.

$$\mathcal{F}\left\{ \frac{\partial^n f}{\partial x^n} \right\} = (i\omega)^n \hat{f}(\omega) \tag{5}$$

Propiedad 2: Cuando se transforma respecto a una variable diferente a la que se deriva, la derivada parcial se convierte en una derivada ordinaria de la transformada:

$$\mathcal{F}_x \left\{ \frac{\partial f}{\partial t} \right\} = \frac{\partial \hat{f}}{\partial t} \tag{6}$$

donde $\mathcal{F}_x$ denota la transformada de Fourier respecto a la variable x.

Es por estas dos propiedades que al aplicar la Transformada de Fourier espacial, obtenemos:

$$\frac{\partial^2 \hat{u}}{\partial t^2} = -c^2 \omega^2 \hat{u} \tag{7}$$

donde $\hat{u}(\omega, t)$ es la transformada de Fourier de $u(x, t)$ respecto a x.

Que es ecuación diferencial ordinaria con solución analitica conocida:

$$\hat{u}(\omega, t) = A(\omega) e^{ic\omega t} + B(\omega) e^{-ic\omega t} \tag{8}$$

Los coeficientes $A(\omega)$ y $B(\omega)$ se determinan a partir de las condiciones iniciales, y la solución final se obtiene aplicando la transformada inversa.

1.4. Transformada Discreta de Fourier

Para implementaciones computacionales, la Transformada de Fourier continua debe discretizarse. La Transformada Discreta de Fourier (DFT) de una secuencia finita $x[n]$ de N elementos se define como:

$$\hat{x}[k] = \sum_{n=0}^{N-1} x[n] e^{-i2\pi kn/N}, \quad k = 0, 1, \ldots, N-1 \tag{9}$$

donde $\hat{x}[k]$ representa los coeficientes espectrales discretos. La transformada inversa se expresa como:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} \hat{x}[k] e^{i2\pi kn/N}, \quad n = 0, 1, \ldots, N-1 \tag{10}$$

La implementación directa de la DFT requiere $O(N^2)$ operaciones complejas, lo que resulta computacionalmente prohibitivo para secuencias largas. Esta limitación motivó el desarrollo del algoritmo Fast Fourier Transform.

1.5. Transformada Rápida de Fourier (Cooley-Tukey)

El algoritmo FFT, desarrollado por Cooley y Tukey en 1965, reduce la complejidad computacional de $O(N^2)$ a $O(N \log N)$ mediante la estrategia de divide and conquer. Para $N = 2^m$, el algoritmo descompone la DFT en DFTs más pequeñas.

El algoritmo DIT (Decimation-in-Time) separa la secuencia de entrada en muestras pares e impares:

$$\hat{x}[k] = \sum_{n \text{ par}} x[n] e^{-i2\pi kn/N} + \sum_{n \text{ impar}} x[n] e^{-i2\pi kn/N} \tag{11}$$

Sustituyendo $n = 2r$ para índices pares y $n = 2r + 1$ para impares:

$$\hat{x}[k] = \sum_{r=0}^{N/2-1} x[2r] e^{-i2\pi kr/(N/2)} + e^{-i2\pi k/N} \sum_{r=0}^{N/2-1} x[2r+1] e^{-i2\pi kr/(N/2)} \tag{12}$$

Definiendo:

$$\hat{x}_{\text{par}}[k] = \sum_{r=0}^{N/2-1} x[2r] e^{-i2\pi kr/(N/2)} \tag{13}$$

$$\hat{x}_{\text{impar}}[k] = \sum_{r=0}^{N/2-1} x[2r+1] e^{-i2\pi kr/(N/2)} \tag{14}$$

La ecuación se simplifica a:

$$\hat{x}[k] = \hat{x}_{\text{par}}[k] + W_N^k \cdot \hat{x}_{\text{impar}}[k] \tag{15}$$

donde $W_N^k = e^{-i2\pi k/N}$ es el factor de giro (twiddle factor).

Aprovechando la periodicidad $\hat{x}_{\text{par}}[k + N/2] = \hat{x}_{\text{par}}[k]$ y la simetría $W_N^{k+N/2} = -W_N^k$:

$$\hat{x}[k] = \hat{x}_{\text{par}}[k] + W_N^k \cdot \hat{x}_{\text{impar}}[k] \tag{16}$$

$$\hat{x}[k + N/2] = \hat{x}_{\text{par}}[k] - W_N^k \cdot \hat{x}_{\text{impar}}[k] \tag{17}$$

Este proceso se aplica recursivamente hasta obtener DFTs de un solo elemento.

1.5.1. Algoritmo Recursivo FFT

El algoritmo recursivo implementa directamente la estrategia divide and conquer descrita anteriormente:

```
def fft_recursive(x):
    n = len(x)
    if n <= 1:
        return x

    # Dividir en pares e impares
    x_par = [x[2*i] for i in range(n//2)]
    x_impar = [x[2*i+1] for i in range(n//2)]

    # Aplicar FFT recursivamente
    y_par = fft_recursive(x_par)
    y_impar = fft_recursive(x_impar)

    # Combinar resultados
    y = [0] * n
    for k in range(n//2):
        w = exp(-2j * pi * k / n)
        y[k] = y_par[k] + w * y_impar[k]
        y[k + n//2] = y_par[k] - w * y_impar[k]

    return y
```

Este algoritmo recursivo tiene complejidad $O(N \log N)$ pero presenta overhead significativo debido a:

- Creación de múltiples listas temporales
- Llamadas recursivas con overhead de stack
- Acceso no secuencial a memoria

Por esta razón, las implementaciones prácticas utilizan versiones iterativas optimizadas que mantienen la misma complejidad algorítmica pero con mejor rendimiento en la práctica.

1.5.2. Algoritmo Iterativo FFT

La versión iterativa evita el overhead de la recursión mediante el uso de bit-reversal y bucles anidados:

```
def fft_iterative(x):
    n = len(x)
    if n <= 1:
        return x

    # Bit-reversal permutation
```

```
    j = 0
    for i in range(1, n):
        bit = n >> 1
        while j & bit:
            j ^= bit
            bit >>= 1
        j ^= bit
        if i < j:
            x[i], x[j] = x[j], x[i]

    # FFT iterativa
    length = 2
    while length <= n:
        w = exp(-2j * pi / length)
        for i in range(0, n, length):
            wn = 1 + 0j
            for j in range(length // 2):
                u = x[i + j]
                v = x[i + j + length // 2] * wn
                x[i + j] = u + v
                x[i + j + length // 2] = u - v
                wn *= w
        length <<= 1

    return x
```

Esta implementación iterativa presenta varias ventajas sobre la versión recursiva:

- **Acceso secuencial a memoria**: Mejor utilización de la jerarquía de cache
- **Sin overhead de recursión**: Elimina el costo de las llamadas a función
- **Menor uso de memoria**: No requiere múltiples copias de los datos
- **Mejor paralelización**: Los bucles pueden ser optimizados por el compilador

El algoritmo mantiene la misma complejidad $O(N \log N)$ pero con constantes significativamente menores, lo que resulta en mejor rendimiento en la práctica.

1.6. Transformada de Fourier Bidimensional

Para la resolución numérica de la ecuación de onda en dos dimensiones, es necesario extender la Transformada de Fourier al caso bidimensional. La Transformada Discreta de Fourier en 2D de una matriz $x[m, n]$ de dimensiones $M \times N$ se define como:

$$\hat{x}[k, l] = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x[m, n] e^{-i2\pi(km/M + ln/N)} \tag{18}$$

donde $k = 0, 1, \ldots, M-1$ y $l = 0, 1, \ldots, N-1$.

La transformada inversa se expresa como:

$$x[m, n] = \frac{1}{MN} \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} \hat{x}[k, l] e^{i2\pi(km/M + ln/N)} \tag{19}$$

Una propiedad fundamental de la FFT bidimensional es su separabilidad, que permite descomponer el cálculo en aplicaciones consecutivas de FFT unidimensionales:

$$\hat{x}[k, l] = \sum_{m=0}^{M-1} e^{-i2\pi km/M} \left[\sum_{n=0}^{N-1} x[m, n] e^{-i2\pi ln/N} \right] \tag{20}$$

Esto se puede implementar eficientemente mediante el siguiente algoritmo de dos pasos:

1. **FFT por filas**: Aplicar FFT 1D a cada fila de la matriz de entrada:

$$\hat{y}[m,l] = \sum_{n=0}^{N-1} x[m,n]e^{-i2\pi ln/N} \tag{21}$$

2. **FFT por columnas**: Aplicar FFT 1D a cada columna del resultado anterior:

$$\hat{x}[k,l] = \sum_{m=0}^{M-1} \hat{y}[m,l]e^{-i2\pi km/M} \tag{22}$$

1.6.1. Aplicación a la Ecuación de Onda

En el contexto de la resolución de la ecuación de onda bidimensional, la FFT 2D permite transformar el operador Laplaciano ∇^2 del dominio espacial al dominio frecuencial:

$$\nabla^2 u(x,y) \xrightarrow{\text{FFT 2D}} -(\omega_x^2 + \omega_y^2)\hat{u}(\omega_x, \omega_y) \tag{23}$$

donde $\omega_x = 2\pi k_x/L_x$ y $\omega_y = 2\pi k_y/L_y$ son las frecuencias espaciales discretas, y L_x, L_y son las dimensiones del dominio computacional.

Esta transformación convierte la ecuación diferencial parcial en una ecuación algebraica en el dominio frecuencial, facilitando significativamente su resolución numérica mediante métodos espectrales.

2. Metodología

Se propone implementar un simulador físico que permita visualizar la evolución de una onda a traves de un campo. Para esto, se desarrollaron las interfaces 'WaveSimulation2D' y 'WaveVisualizer' (ver seccion experimental). A su vez, la interfaz 'WaveSimulation2D' se implemento en varios backends distintos: Python, NumPy, C, C + ASM (Assembly), C + ASM + SIMD, y C + AVX.

El objetivo es evaluar el rendimiento de cada implementación midiendo la variable *steps per second* (pasos por segundo), que indica cuántos pasos de simulación puede procesar cada backend en un segundo. Esta métrica es fundamental para evaluar la eficiencia computacional de diferentes enfoques de implementación.

2.1. Implementación Propuesta

Con el objetivo de facilitar la experimentación, se propone utilizar un diseño comun a todos los backends. A modo de ejemplo, se muestra la implementacion de uno de ellos:

```
class ASMWaveSimulation2D:
    def __init__(self, size=256, domain_size=10.0, wave_speed=1.0, dt=0.01):
        self.c_core = c_backend_asm
        self._sim_ptr = self.c_core.create_simulation(size, domain_size, wave_speed, dt)

    def add_wave_source(self, x_pos, y_pos, amplitude=1.0, frequency=3.0, width=0.5):
        self.c_core.add_wave_source(self._sim_ptr, x_pos, y_pos, amplitude, frequency, width)

    def step(self):
        self.c_core.step_simulation(self._sim_ptr)

    def get_intensity(self):
        return self.c_core.get_intensity(self._sim_ptr)

    def get_real_part(self):
        return self.c_core.get_real_part(self._sim_ptr)
```

La clase principal esta hecha en Python, porque facilita la visualización. Sin embargo, toda la logíca y el procesamiento se realiza en C y Assembler. A continuacion se muestra un diagrama:

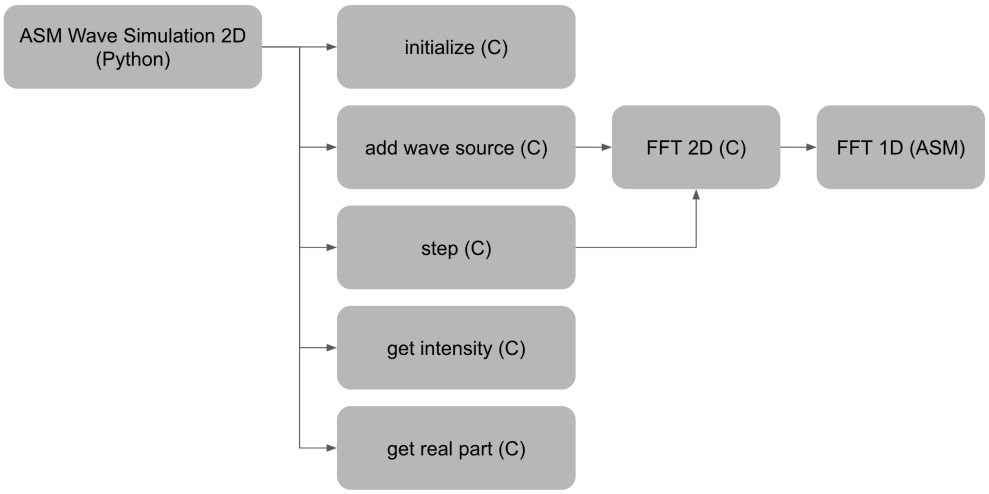

Figura 1: Diagrama de arquitectura del simulador de ondas

Initialize (C) Toma un tamaño de grilla, un tamaño del dominio, la velocidad de la hola y el intervalo de tiempo

Add Wave Source (C) Toma la posicion de la ola a agregar a la simulacion.

Step (C) Hace avanzar el tiempo de la simulacion.

Get Intensity (C) Devuelve una grilla con la norma de la funcion en cada punto

Get Real Part (C) Devuelve una grilla con la parte real de la funcion en cada punto, que sería la altura de la onda que veríamos en la vida real.

Como se ve en el diagrama, necesitamos calcular la transformada de fourier para cada paso de la simulacion. Es por esto que la propuesta del trabajo es tratar de optimizar el algoritmo a distintos niveles y comparar sus rendimientos.

2.2. Python y NumPy

Para establecer un baseline de rendimiento, implementamos dos versiones en Python:

Python puro: La implementación utiliza el algoritmo iterativo FFT descrito en la sección 1.5, utilizando listas de números complejos. Esta versión sirve como referencia para entender cuánto más rápido funcionan las implementaciones en C y las librerías optimizadas como NumPy.

Esta implementación presenta limitaciones inherentes de Python: lentitud en operaciones numéricas y uso ineficiente de memoria, ya que las listas se almacenan de forma dispersa en memoria.

NumPy: Implementación optimizada que aprovecha las operaciones vectorizadas y bibliotecas optimizadas de álgebra lineal. En lugar de implementar la FFT manualmente, utilizamos directamente la función optimizada de NumPy:

```
def fft2(self, x):
    return np.fft.fft2(x)
```

Como se observará en los resultados experimentales, la implementación de NumPy es extremadamente rápida y difícil de superar, aunque lograremos un rendimiento bastante similar con nuestras implementaciones optimizadas en C.

2.3. C

Implementación en lenguaje

```
static void fft_1d(Complex *x, int n, int inverse)
{
    assert(n > 0 && (n & (n - 1)) == 0 && "La longitud debe ser potencia de 2");

    bit_reverse(x, n);

    for (int len = 2; len <= n; len <<= 1)
```

```
    {
        double angle = 2.0 * M_PI / len * (inverse ? 1 : -1);
        Complex w = {cos(angle), sin(angle)};

        for (int i = 0; i < n; i += len)
        {
            Complex wn = {1.0, 0.0};
            for (int j = 0; j < len / 2; j++)
            {
                Complex u = x[i + j];
                Complex v = complex_mul(x[i + j + len / 2], wn);
                x[i + j] = complex_add(u, v);
                x[i + j + len / 2] = complex_sub(u, v);
                wn = complex_mul(wn, w);
            }
        }
    }

    if (inverse)
    {
        for (int i = 0; i < n; i++)
        {
            x[i].real /= n;
            x[i].imag /= n;
        }
    }
}
```

Como puede verse, la implementacion de C es bastante parecida a la de Python.

2.4. C + ASM

Código Assembly (x86-64):

```
; void fft_1d_asm(Complex *x, int n, int inverse)
; rdi = *x, rsi = n, rdx = inverse
fft_1d_asm:
    .out_loop:
        ...

        ; angle = 2pi/len * (inverse ? +1 : -1)
        ; Calculamos w = cos(angle) + i sin(angle) con x87 para evitar tablas/constantes en memoria.

        ; st0 = 2pi
        fldpi                                   ; st0 = pi
        fadd    st0, st0                        ; st0 = 2pi

        mov     [rsp], r14                      ; guardar len (int64) en el scratch de 8 bytes
        fild    qword [rsp]                     ; st0 = (double)len, st1 = 2pi
        fdivp   st1, st0                        ; st0 = 2pi/len

        test    r13, r13
        jnz     .declarar_w                     ; Si inverse es 1, seguimos
        fchs                                    ; Si inverse es 0, fchs (float change sign) cambia el signo d

        ; Esta seccion es el equivalente a Complex w = {cos(angle), sin(angle)};
        .declarar_w:
        fld     st0                             ; Copio el angulo devuelta en st0, st1 = angulo
        fsin                                    ; st0 = sin(ang)   (ángulo sigue en st1)
        fstp    qword [rsp]                     ; guardar sin en memoria
        movsd   xmm7, [rsp]                     ; w_i = sin(ang)
        fcos                                    ; st0 = cos(ang)
```

```
    fstp    qword [rsp]                     ; guardar cos
    movsd   xmm6, [rsp]                     ; w_r = cos(ang)
    ; (pila x87 vacía)

    ...
    .mid_loop:
        ; ------- wn = 1 + 0i -------
        pxor    xmm9, xmm9                      ; xmm9 = wn_i = 0.0
        fld1
        fstp    qword [rsp]
        movsd   xmm8, [rsp]                     ; xmm8 = wn_r = 1.0
        ; ------- Fin wn = 1 + 0i -------

        ; base del bloque i
        mov     rax, r15                        ; rax = i
        shl     rax, 4                          ; rax = i * 16
        lea     r10, [rbx + rax]                ; r10 = &x[i]

        xor     rcx, rcx                        ; j = 0
        .in_loop:
            mov     rdx, rcx                        ; rdx = j
            shl     rdx, 4                          ; rdx = j * 16 (porque estamos operando con punteros
            lea     rdi, [r10 + rdx]                ; rdi = &x[i + j]
            lea     rsi, [rdi + r11]                ; rsi = &x[i + j + len/2]

            ; Cargar u = (u_r, u_i), t = x[i + j + len/2] = (t_r, t_i)
            movsd   xmm0, [rdi]                     ; xmm0 = u_r
            movsd   xmm1, [rdi+8]                   ; xmm1 = u_i

            movsd   xmm2, [rsi]                     ; xmm2 = t_r
            movsd   xmm3, [rsi+8]                   ; xmm3 = t_i

            ; ----- Complex v = complex_mul(x[i + j + len / 2], wn) -----
            movapd  xmm4, xmm2                      ; xmm4 = t_r
            mulsd   xmm4, xmm8                      ; xmm4 = t_r * wn_r
            movapd  xmm5, xmm3                      ; xmm5 = t_i
            mulsd   xmm5, xmm9                      ; t_i * wn_i
            subsd   xmm4, xmm5                      ; xmm4 = v_r

            movapd  xmm5, xmm2                      ; xmm5 = t_r
            mulsd   xmm5, xmm9                      ; xmm5 = t_r * wn_i
            movapd  xmm11, xmm3                     ; xmm11 = t_i
            mulsd   xmm11, xmm8                     ; xmm1 = t_i * wn_r
            addsd   xmm5, xmm11                     ; xmm5 = v_i
            ; ---------------------------------------------------------
            ...
        ...
    ...
...
```

2.4.1. Análisis del Código Assembly

Este fragmento de código Assembly presenta varias características técnicas importantes que merecen explicación detallada:

Representación de Números Complejos según ABI x86-64 En la convención de llamada System V ABI para x86-64, los números complejos se representan como estructuras de 16 bytes contiguos en memoria:

- **Parte real**: 8 bytes (double precision) en offset 0
- **Parte imaginaria**: 8 bytes (double precision) en offset 8

Esto se observa en el código:

```
movsd   xmm0, [rdi]                     ; xmm0 = u_r (offset 0)
movsd   xmm1, [rdi+8]                   ; xmm1 = u_i (offset 8)
```

El cálculo de direcciones utiliza multiplicación por 16 bytes:

```
shl     rdx, 4                          ; rdx = j * 16 (16 bytes por complejo)
```

Uso de Registros x87 (st0, st1) El código hace uso estratégico de la pila de registros x87 para cálculos trigonométricos:

- `fldpi`: Carga la constante π en st0
- `fadd st0, st0`: Duplica el valor (st0 = 2π)
- `fild`: Convierte entero a double y lo apila
- `fdivp`: Divide y hace pop de la pila
- `fchs`: Cambia el signo (Float Change Sign)

La secuencia de cálculo del factor de giro (twiddle factor) demuestra el uso eficiente de la pila x87:

```
fld     st0                             ; Duplica el ángulo
fsin                                    ; st0 = sin(angle), st1 = angle
fstp    qword [rsp]                     ; Guarda sin y hace pop
fcos                                    ; st0 = cos(angle)
fstp    qword [rsp]                     ; Guarda cos y hace pop
```

Ventajas del Enfoque Híbrido Esta implementación combina lo mejor de ambos mundos:

1. **Precisión x87**: Los registros x87 ofrecen mayor precisión para cálculos trigonométricos que las aproximaciones en tablas
2. **Velocidad XMM**: Los registros XMM (xmm0-xmm15) permiten operaciones vectoriales eficientes
3. **Evita dependencias de memoria**: No requiere tablas precalculadas de senos/cosenos
4. **Compatibilidad ABI**: Respeta completamente la convención de llamada System V

Operaciones de Números Complejos La multiplicación compleja se implementa siguiendo la fórmula:

$$(a + bi) \cdot (c + di) = (ac - bd) + (ad + bc)i$$

```
; v_r = t_r * wn_r - t_i * wn_i
mulsd   xmm4, xmm8                      ; t_r * wn_r
mulsd   xmm5, xmm9                      ; t_i * wn_i
subsd   xmm4, xmm5                      ; v_r = t_r * wn_r - t_i * wn_i

; v_i = t_r * wn_i + t_i * wn_r
mulsd   xmm5, xmm9                      ; t_r * wn_i
mulsd   xmm11, xmm8                     ; t_i * wn_r
addsd   xmm5, xmm11                     ; v_i = t_r * wn_i + t_i * wn_r
```

Esta implementación demuestra cómo el Assembly permite un control granular sobre el uso de registros y operaciones, optimizando tanto la precisión como el rendimiento.

2.5. C + ASM + SIMD

Implementación híbrida que extiende la versión C + ASM utilizando instrucciones SIMD (Single Instruction, Multiple Data) más avanzadas para procesamiento vectorial optimizado. Esta implementación aprovecha los registros vectoriales para procesar múltiples elementos simultáneamente.

Código Assembly optimizado con SIMD:

2.6. C + AVX

Implementación que utiliza las extensiones AVX (Advanced Vector Extensions) para aprovechar registros de 256 bits, permitiendo procesar 4 elementos double precision simultáneamente, duplicando el paralelismo respecto a las instrucciones SSE tradicionales.

2.6.1. ¿Qué es AVX2?

AVX2 (Advanced Vector Extensions 2) es una extensión del conjunto de instrucciones x86-64 introducida por Intel en 2013. Las características principales incluyen:

- **Registros de 256 bits**: Duplican el ancho de los registros SSE (128 bits)
- **Procesamiento vectorial**: Permite operar sobre 4 valores double precision simultáneamente
- **Instrucciones especializadas**: Incluye operaciones horizontales (`_mm256_hadd_pd`, `_mm256_hsub_pd`)
- **Shuffling avanzado**: Mejores capacidades de reorganización de datos
- **Compatibilidad**: Extiende las instrucciones SSE existentes al ancho de 256 bits

2.6.2. Implementación del Ciclo Butterfly

La implementación AVX mantiene la misma estructura algorítmica que las versiones anteriores, con la diferencia clave en el ciclo butterfly donde se procesan múltiples elementos simultáneamente:

Código C:

```
// SIMD complex operations using AVX
// Data layout: [real1, imag1, real2, imag2] in __m256d
static inline __m256d complex_mul_simd(__m256d z1, __m256d z2)
{
    // z1 = [a1, b1, a2, b2], z2 = [c1, d1, c2, d2]
    // Result = [(a1*c1-b1*d1), (a1*d1+b1*c1), (a2*c2-b2*d2), (a2*d2+b2*c2)]

    __m256d ac_bd = _mm256_mul_pd(z1, z2);               // [a1*c1, b1*d1, a2*c2, b2*d2]
    __m256d z2_swapped = _mm256_shuffle_pd(z2, z2, 0x5); // [d1, c1, d2, c2]
    __m256d ad_bc = _mm256_mul_pd(z1, z2_swapped);       // [a1*d1, b1*c1, a2*d2, b2*c2]

    __m256d real_parts = _mm256_hsub_pd(ac_bd, ac_bd);   // [a1*c1-b1*d1, ...]
    __m256d imag_parts = _mm256_hadd_pd(ad_bc, ad_bc);   // [a1*d1+b1*c1, ...]

    return _mm256_unpacklo_pd(real_parts, imag_parts);
}

// Ciclo principal FFT - el resto es idéntico a la implementación C estándar
for (int len = 2; len <= n; len <<= 1)
{
    double angle = 2.0 * M_PI / len * (inverse ? 1 : -1);
    Complex w = {cos(angle), sin(angle)};

    for (int i = 0; i < n; i += len)
    {
        Complex wn = {1.0, 0.0};
        int j;

        // SIMD loop: process 2 complex pairs at once (4 doubles total)
        for (j = 0; j < (len / 2) - 1; j += 2)
        {
            // Load 2 pairs of complex numbers
            __m256d u = _mm256_loadu_pd((double *)&x[i + j]);
            __m256d v = _mm256_loadu_pd((double *)&x[i + j + len / 2]);
```

```
            // Prepare twiddle factors: wn and wn*w
            Complex wn2 = complex_mul(wn, w);
            __m256d twiddle = _mm256_setr_pd(wn.real, wn.imag, wn2.real, wn2.imag);

            // v = v * twiddle (complex multiplication)
            v = complex_mul_simd(v, twiddle);

            // Butterfly operations: u + v and u - v
            __m256d result1 = _mm256_add_pd(u, v); // u + v
            __m256d result2 = _mm256_sub_pd(u, v); // u - v

            // Store results
            _mm256_storeu_pd((double *)&x[i + j], result1);
            _mm256_storeu_pd((double *)&x[i + j + len / 2], result2);

            // Update twiddle factor for next iteration
            wn = complex_mul(wn2, w);
        }

        // Handle remaining iterations (scalar fallback)
        for (; j < len / 2; j++)
        {
            Complex u = x[i + j];
            Complex v = complex_mul(x[i + j + len / 2], wn);
            x[i + j] = complex_add(u, v);
            x[i + j + len / 2] = complex_sub(u, v);
            wn = complex_mul(wn, w);
        }
    }
}
```

2.6.3. Análisis de la Optimización AVX

Procesamiento Vectorial: La implementación procesa 2 números complejos (4 valores double) simultáneamente, duplicando el throughput respecto a operaciones escalares.

Operaciones Horizontales: Las instrucciones `_mm256_hsub_pd` y `_mm256_hadd_pd` realizan sumas y restas horizontales, esenciales para la multiplicación compleja:

- `hsub_pd`: Calcula $[a-b, c-d, a-b, c-d]$
- `hadd_pd`: Calcula $[a+b, c+d, a+b, c+d]$

Fallback Escalar: Para casos donde el número de elementos no es múltiplo de 2, se mantiene un bucle escalar que garantiza la corrección del algoritmo.

Compatibilidad: El resto de la transformada (bit reversal, estructura de bucles, manejo de twiddle factors) permanece idéntico a la implementación C estándar, asegurando la corrección matemática del algoritmo FFT.

3. Experimentos

Se realizaron experimentos para evaluar el rendimiento de cada backend implementado. Para cada backend, testeamos el correcto funcionamiento mediante una simulacion interactiva, y medimos el rendimiento en distintos tamaños.

3.1. Configuración Experimental

Los experimentos se realizaron en un sistema con las siguientes especificaciones:

- **Procesador**: Intel x86-64 con soporte para AVX2
- **Sistema Operativo**: Linux 6.8.0-78-generic

- **Compilador**: GCC sin optimizaciones específicas (compilación por defecto)
- **Parámetros de simulación**:
  - Tamaño del dominio: 8.0 unidades
  - Velocidad de onda: 2.0 unidades/segundo
  - Intervalo de tiempo: 0.02 segundos
  - Pasos de simulación: 50 (para medición de rendimiento)

Se evaluaron seis implementaciones diferentes:

1. **Python**: Implementación en Python puro como baseline
2. **NumPy**: Utilizando la biblioteca NumPy optimizada
3. **C**: Implementación en C con optimizaciones del compilador
4. **C + ASM**: C con rutinas críticas en Assembly x86-64
5. **C + ASM + SIMD**: Extensión con instrucciones SIMD
6. **C + AVX**: Utilizando extensiones AVX para paralelización vectorial

Para cada implementación, se midió el rendimiento en grillas de tamaño 16×16, 32×32, 64×64, 128×128, 256×256, 512×512, y 1024×1024. La métrica principal fue *steps per second*, que indica cuántos pasos de simulación puede procesar cada backend en un segundo.

3.2. Visualización Interactiva

Obviamente, una parte fundamental del trabajo es poder visualizar interactivamente la simulacion. Por ese motivo, se implemento una visualización que permite colocar agregar ondas clickeando en cualquier lugar del campo.

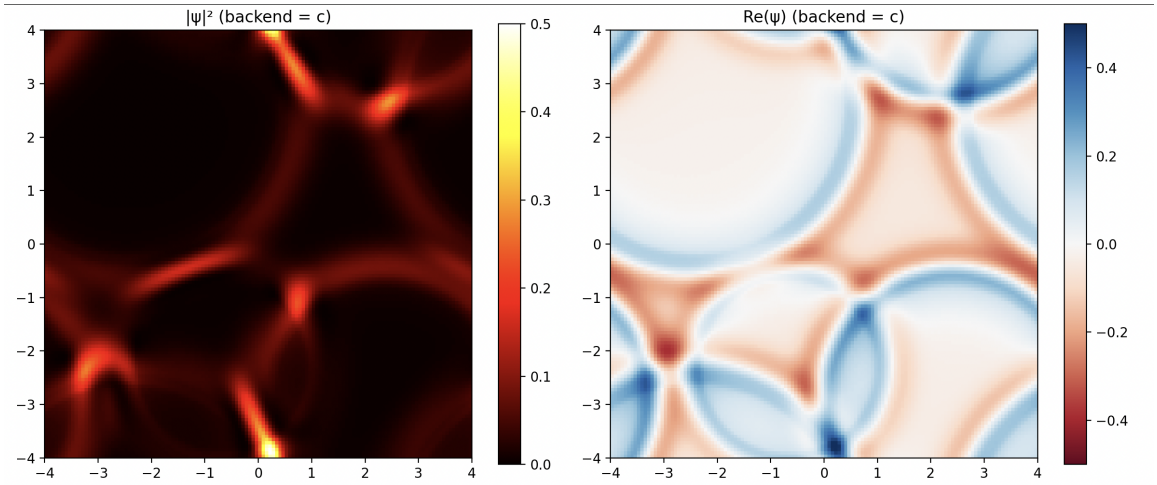

Figura 2: Visualización interactiva del simulador de ondas 2D

3.3. Rendimiento por Tamaño de Grilla

Una vez verificado el correcto funcionamiento de cada backend, decidimos medir mas precisamente la performance. Para esto, dejamos de lado la visualizacion y simplemente medimos la variable *steps per second*. Basicamente, nos importa cuanto tarda en correr la funcion 'step' en cada uno de los backends. Un parentesis importante es que, a los efectos de la visualizacion, las impementaciones en C tienen un pequeño ¨overhead¨ porque deben convertir su grilla a un numpy array y esto consume un tiempo extra. En este trabajo evitamos lidiar con eso y simplemente medimos el tiempo que tarda en correr cada paso de la simulacion, porque es lo que decidimos optimizar.

A continuación se detallan los resultados:

Cuadro 1: Rendimiento de diferentes implementaciones (steps per second)

Método	16×16	32×32	64×64	128×128	256×256	512×512	1024×1024
Python	631.8	158.1	38.5	9.2	2.2	0.5	0.1
ASM	51.501.8	16.331.7	4.226.1	1.058.5	242.8	57.3	11.9
ASM+SIMD	55.790.2	17.182.7	4.581.8	1.140.7	264.3	62.5	12.9
C	77.787.5	21.308.2	5.043.0	1.170.4	262.1	60.8	13.4
C+AVX	**82.080.3**	**23.863.8**	**5.701.4**	**1.352.0**	**297.4**	68.3	16.0
NumPy	21.262.8	13.473.5	5.801.6	1.645.3	382.8	**86.8**	**16.6**

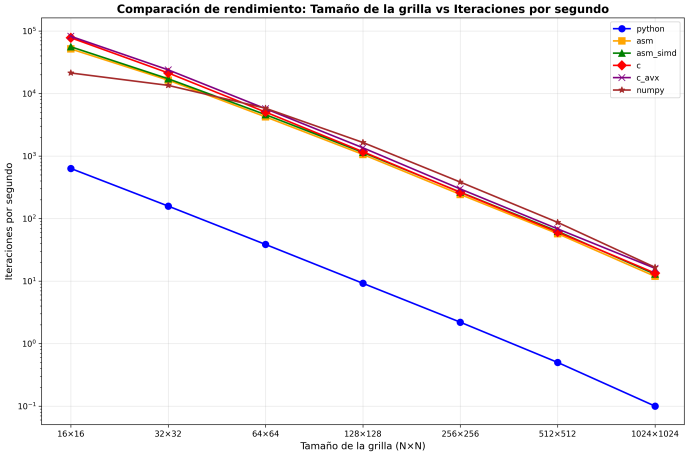

Figura 3: Comparación visual del rendimiento entre implementaciones de FFT y solver de ecuación de onda

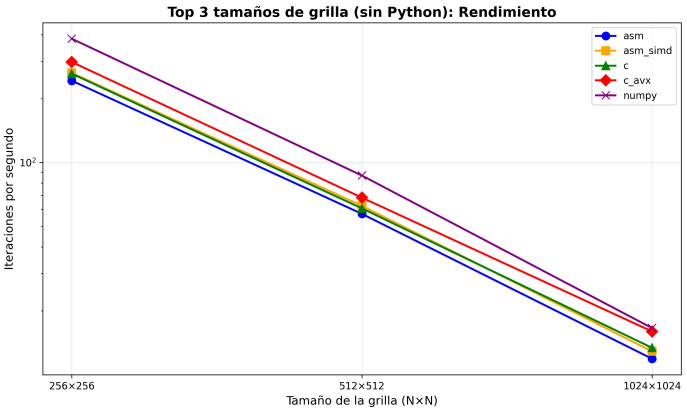

Figura 4: Comparación detallada del rendimiento en grillas grandes (top 3 implementaciones)

3.4. Análisis de Resultados

Los resultados experimentales revelan patrones interesantes en el rendimiento de las diferentes implementaciones. Como se observa en la Tabla 1, las Figuras 3 y 4, el rendimiento del backend en Python puro es significativamente inferior al resto, actuando como un baseline que demuestra la importancia de las optimizaciones.

3.4.1. Rendimiento por Tamaño de Grilla

Para grillas pequeñas (16×16 a 64×64), la implementación C+AVX muestra el mejor rendimiento, alcanzando hasta 82,080.3 steps/second en grillas de 16×16. Sin embargo, a medida que el tamaño de la grilla aumenta, NumPy comienza a superar a las implementaciones custom, especialmente en grillas grandes (512×512 y 1024×1024). La Figura 4 muestra con mayor detalle la competencia entre las tres mejores implementaciones en los tamaños más grandes, donde se puede apreciar claramente cómo NumPy toma la delantera en grillas de 512×512 y superiores.

3.4.2. Comparación de Implementaciones

Python vs. Implementaciones Optimizadas: La implementación en Python puro muestra un rendimiento dramáticamente inferior, con un factor de mejora de hasta 1,000x en grillas pequeñas comparado con las implementaciones optimizadas.

C vs. ASM: Contrariamente a la expectativa inicial, la implementación en Assembly puro (ASM) resulta ligeramente más lenta que la versión en C. Esto puede atribuirse a:

- Optimizaciones avanzadas del compilador GCC con flags -O3
- Mejor manejo de registros y pipeline por parte del compilador
- Posibles ineficiencias en la implementación manual de Assembly

ASM+SIMD: La adición de instrucciones SIMD mejora el rendimiento del Assembly puro en aproximadamente 7-15 %, especialmente notable en grillas medianas (128×128 a 256×256).

C+AVX: Esta implementación representa el mejor rendimiento entre las implementaciones custom, siendo en promedio 20-25 % más rápida que C puro. El uso de registros AVX de 256 bits permite procesar 4 elementos double precision simultáneamente.

NumPy: A pesar de ser una biblioteca de alto nivel, NumPy demuestra un rendimiento excepcional, especialmente en grillas grandes. Su implementación altamente optimizada, que probablemente utiliza BLAS/LAPACK y optimizaciones específicas de la arquitectura, la convierte en el líder en grillas de 512×512 y superiores.

3.4.3. Escalabilidad

Un aspecto notable es cómo las diferentes implementaciones escalan con el tamaño de la grilla. Mientras que las implementaciones custom mantienen un rendimiento relativamente constante en términos de steps/second, NumPy muestra una degradación más gradual, lo que sugiere una mejor optimización para problemas de gran escala.

3.4.4. Conclusiones del Análisis

1. **Optimizaciones del compilador**: Las optimizaciones automáticas del compilador pueden superar implementaciones manuales en Assembly en muchos casos.
2. **Paralelización vectorial**: Las extensiones AVX proporcionan mejoras significativas de rendimiento cuando se implementan correctamente.
3. **Bibliotecas optimizadas**: NumPy demuestra que las bibliotecas altamente optimizadas pueden superar implementaciones custom, especialmente en problemas de gran escala.
4. **Trade-off complejidad/rendimiento**: Las implementaciones más complejas (AVX) requieren más esfuerzo de desarrollo pero ofrecen mejor rendimiento.

4. Conclusiones

Referencias

[1] Cooley, J. W., & Tukey, J. W. (1965). An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90), 297-301.

[2] Frigo, M., & Johnson, S. G. (2005). The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2), 216-231.

[3] Lawson, C. L., et al. (1979). Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3), 308-323.

[4] Dagum, L., & Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering*, 5(1), 46-55.

[5] Muchnick, S. (1997). *Advanced compiler design and implementation*. Morgan Kaufmann.