DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Optimización de Rendimiento en Arquitecturas de Computadoras

Análisis Comparativo de Técnicas de Aceleración

30 de Agosto de 2025 — Organización del Computador II

Grupo 1

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.

1.1.1. Ecuación de Onda en Una Dimensión

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 diferencial parcial de segundo orden describe fenómenos como:

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

1.1.2. Ecuación de Onda en Dos Dimensiones

La extensión a dos dimensiones espaciales resulta en:

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

Esta formulación bidimensional modela fenómenos como:

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

La solución numérica de estas ecuaciones mediante métodos de diferencias finitas o elementos finitos requiere algoritmos computacionalmente intensivos que se benefician significativamente de técnicas de optimización.

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:

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

y su transformada inversa:

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

Esta representación en el dominio frecuencial revela propiedades fundamentales de las señales y simplifica considerablemente el análisis de sistemas lineales.

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 trasforma el problema diferencial en uno algebraico, facilitando significativamente su resolución. Considerando la ecuación de onda unidimensional con condiciones iniciales:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} \tag{5}$$

Al aplicar la Transformada de Fourier espacial, obtenemos:

$$\frac{\partial^2 \hat{u}}{\partial t^2} = -c^2 \omega^2 \hat{u} \tag{6}$$

donde $\hat{u}(\omega, t)$ es la transformada de Fourier de $u(x, t)$ respecto a x. Esta ecuación diferencial ordinaria en el tiempo tiene solución analítica conocida:

$$\hat{u}(\omega, t) = A(\omega)e^{jc\omega t} + B(\omega)e^{-jc\omega t} \tag{7}$$

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 de Fourier.

Esta metodología demuestra la potencia computacional de la Transformada de Fourier, convirtiendo operaciones de derivación en multiplicaciones algebraicas simples. En implementaciones numéricas, la eficiencia de algoritmos FFT (Fast Fourier Transform) resulta crítica para la viabilidad computacional de estos métodos espectrales.

El marco teórico se fundamenta en los principios de arquitectura de computadoras y optimización de código aplicados específicamente a algoritmos de procesamiento de señales. Las aplicaciones similares en el campo científico e industrial incluyen bibliotecas de álgebra lineal optimizadas como BLAS [3], implementaciones optimizadas de FFT como FFTW [2], frameworks de computación paralela como OpenMP [4], y compiladores optimizantes que emplean técnicas avanzadas de análisis estático [5].

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:

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-j2\pi kn/N}, \quad k = 0, 1, \ldots, N-1 \tag{8}$$

donde $X[k]$ representa los coeficientes espectrales discretos. La transformada inversa se expresa como:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k]e^{j2\pi kn/N}, \quad n = 0, 1, \ldots, N-1 \tag{9}$$

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 y vencerás. 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:

$$X[k] = \sum_{n \text{ par}} x[n]e^{-j2\pi kn/N} + \sum_{n \text{ impar}} x[n]e^{-j2\pi kn/N} \tag{10}$$

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

$$X[k] = \sum_{r=0}^{N/2-1} x[2r]e^{-j2\pi kr/(N/2)} + e^{-j2\pi k/N} \sum_{r=0}^{N/2-1} x[2r+1]e^{-j2\pi kr/(N/2)} \tag{11}$$

Definiendo:

$$X_{\text{par}}[k] = \sum_{r=0}^{N/2-1} x[2r]e^{-j2\pi kr/(N/2)} \tag{12}$$

$$X_{\text{impar}}[k] = \sum_{r=0}^{N/2-1} x[2r+1]e^{-j2\pi kr/(N/2)} \tag{13}$$

La ecuación se simplifica a:

$$X[k] = X_{\text{par}}[k] + W_N^k \cdot X_{\text{impar}}[k] \tag{14}$$

donde $W_N^k = e^{-j2\pi k/N}$ es el factor de giro (twiddle factor).

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

$$X[k] = X_{\text{par}}[k] + W_N^k \cdot X_{\text{impar}}[k] \tag{15}$$

$$X[k + N/2] = X_{\text{par}}[k] - W_N^k \cdot X_{\text{impar}}[k] \tag{16}$$

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

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:

$$X[k, l] = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x[m, n]e^{-j2\pi(km/M+ln/N)} \tag{17}$$

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

La transformada inversa se expresa como:

$$x[m, n] = \frac{1}{MN} \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} X[k, l]e^{j2\pi(km/M+ln/N)} \tag{18}$$

1.6.1. Separabilidad de la FFT 2D

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

$$X[k, l] = \sum_{m=0}^{M-1} e^{-j2\pi km/M} \left[\sum_{n=0}^{N-1} x[m, n]e^{-j2\pi ln/N} \right] \tag{19}$$

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:

$$Y[m, l] = \sum_{n=0}^{N-1} x[m, n]e^{-j2\pi ln/N} \tag{20}$$

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

$$X[k, l] = \sum_{m=0}^{M-1} Y[m, l]e^{-j2\pi km/M} \tag{21}$$

1.6.2. Complejidad Computacional

La implementación separable de la FFT 2D tiene una complejidad computacional de:

$$O(MN \log M + MN \log N) = O(MN \log(MN)) \tag{22}$$

Para grillas cuadradas donde $M = N$, esto se simplifica a $O(N^2 \log N)$.

1.6.3. 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:

```python
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:

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.

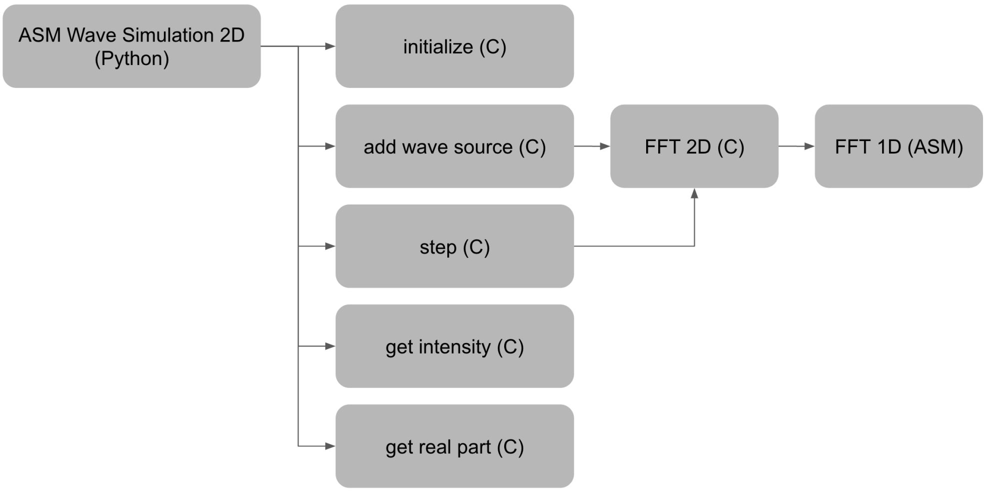

Figura 1: Diagrama de arquitectura del simulador de ondas

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

Comenzamos implementando la FFT unidimensional en python puro, utilizando listas de numeros complejos. Esta implementación sirve como un ˙baseline˙ y nos permite entender que tanto mas rapido funciona C y las librerias como numpy, a su vez que facilita el entendimiento del codigo.

Esto tiene varios problemas, como que Python es lento de por si y ademas que su uso de memoria no es optimo, porque cada lista se guarda desperdigada en cualquier lado.

```
def _fft_1d(self, x: list[complex]) -> list[complex]:
n = len(x)
if n <= 1:
    return x[:]

assert n & (n - 1) == 0, f"La longitud {n} debe ser potencia de 2"

# Bit-reversal
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
length = 2
while length <= n:
    w = cmath.exp(-2j * math.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
```

2.3. NumPy

Implementación optimizada utilizando NumPy como estado del arte para computación científica en Python. Aprovecha las operaciones vectorizadas y bibliotecas optimizadas de álgebra lineal.

En este caso, no le pedimos a numpy que calcule la transformada unidimensional, sino que simplemente podemos usar np.fft.fft2

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

Como veremos a continuación, la implementacion de numpy es extremadamente rapida y dificil de vencer, pero podemos lograr una performance bastante similar.

2.4. 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.

2.5. 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
            ; ------------------------------------------------------------
            ...
        ...
    ...
...
```

2.6. 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:

2.7. 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.

Código C:

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**: macOS 24.6.0
- **Compilador**: GCC con optimizaciones -O3
- **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: 20 (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.

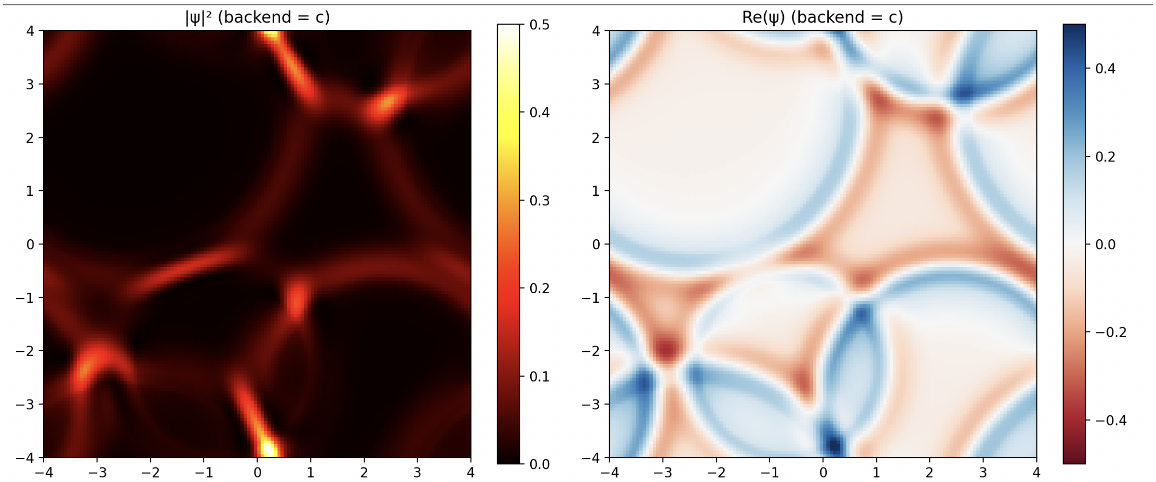

Figura 2: Visualización interactiva del simulador de ondas 2D

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	628.8	154.8	37.8	9.2	2.2	0.5	0.1
ASM	56.375.1	16.487.0	4.289.3	1.062.5	244.3	57.4	11.7
C	75.234.2	20.784.5	5.161.3	1.191.6	261.5	61.0	14.1
ASM+SIMD	53.601.3	16.677.2	4.547.7	1.144.5	263.3	62.9	14.3
C+AVX	**90.785.8**	**23.217.8**	**5.752.3**	**1.353.4**	**296.7**	68.0	15.7
NumPy	20.237.9	12.219.4	5.633.7	1.690.5	392.7	**81.7**	**16.6**

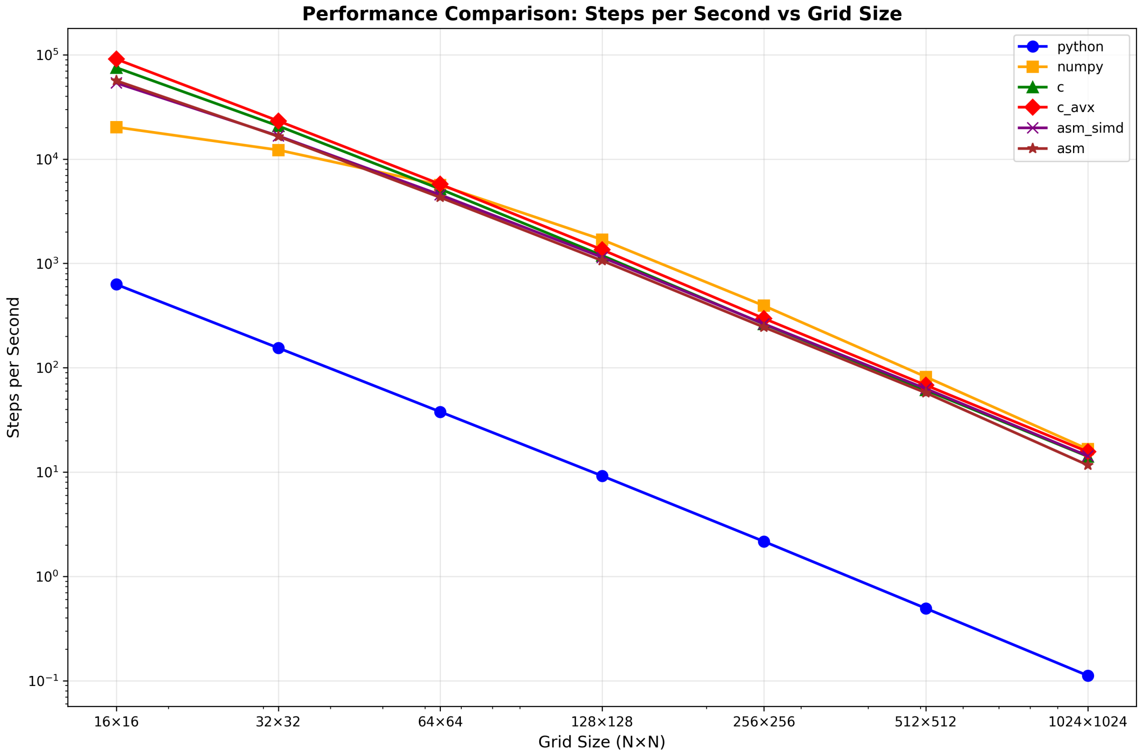

Figura 3: Comparación visual del rendimiento entre implementaciones de FFT y solver de ecuación de onda

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 y la Figura 3, 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 90,785.8 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).

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.