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 **2**

1.1. La Ecuación de Onda . . . 2

1.1.1. Ecuación de Onda en Una Dimensión . . . 2

1.1.2. Ecuación de Onda en Dos Dimensiones . . . 2

1.2. La Transformada de Fourier . . . 2

1.3. Resolución de la Ecuación de Onda mediante la Transformada de Fourier . . . 3

1.4. Transformada Discreta de Fourier . . . 3

1.5. Transformada Rápida de Fourier (Cooley-Tukey) . . . 3

1.6. Transformada de Fourier Bidimensional . . . 4

1.6.1. Separabilidad de la FFT 2D . . . 4

1.6.2. Complejidad Computacional . . . 5

1.6.3. Aplicación a la Ecuación de Onda . . . 5

2. Metodología **5**

2.1. Implementación Propuesta . . . 5

2.2. Python . . . 6

2.3. NumPy . . . 7

2.4. C . . . 7

2.5. C + ASM . . . 8

2.6. C + ASM + SIMD . . . 9

2.7. C + AVX . . . 9

3. Experimentos **9**

3.1. Visualización Interactiva . . . 10

3.2. Rendimiento por Tamaño de Grilla . . . 10

4. Resultados **10**

4.1. Análisis de Rendimiento . . . 10

4.2. Análisis de Resultados . . . 10

5. Conclusiones **10**

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:

```
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.

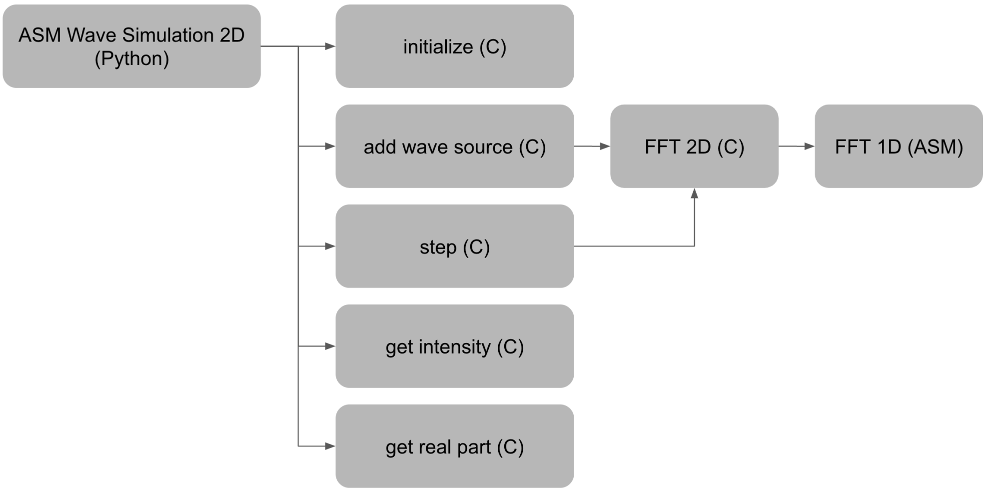

Figura 1: Diagrama de arquitectura del simulador de ondas

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 c

        ; 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
            ; -----------------------------------------------------------

            ; -------------- x[i + j] = complex_add(u, v) --------------
            movapd  xmm11, xmm0                      ; xmm11 = u_r
            addsd   xmm11, xmm4                      ; xmm11 = u_r + v_r
            movapd  xmm12, xmm1                      ; xmm12 = u_i
            addsd   xmm12, xmm5                      ; xmm12 = u_i + v_i
            movsd   [rdi],   xmm11
            movsd   [rdi+8], xmm12
            ; -----------------------------------------------------------

            ...
        ...
    ...
...
```

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 sistemáticos para evaluar el rendimiento de cada backend implementado. La métrica principal utilizada fue *steps per second*, que mide cuántos pasos de simulación de la ecuación de onda puede procesar cada implementación por segundo.

Los experimentos se ejecutaron en grillas de diferentes tamaños para analizar el comportamiento de escalabilidad de cada backend. Se utilizó NumPy como línea base (baseline) para calcular los factores de aceleración (speedup) relativos.

3.1. Visualización Interactiva

3.2. Rendimiento por Tamaño de Grilla

4. Resultados

4.1. Análisis de Rendimiento

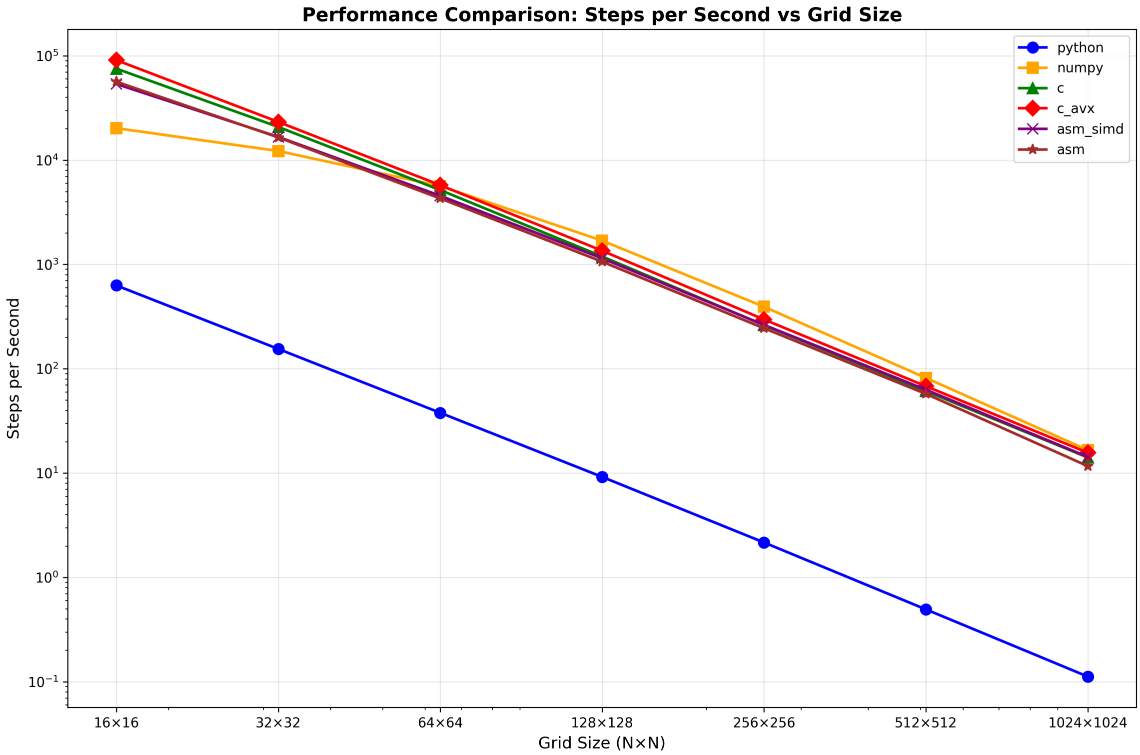

Figura 2: Comparación visual del rendimiento entre implementaciones de FFT y solver de ecuación de onda

4.2. Análisis de Resultados

Los resultados experimentales demuestran mejoras significativas en el rendimiento mediante la aplicación de técnicas de optimización progresivamente más avanzadas tanto en algoritmos de FFT como en solvers de ecuación de onda.

Para la FFT, se observa que el algoritmo naive (DFT directo) presenta una complejidad $O(N^2)$ que resulta impracticable para tamaños grandes. La implementación Radix-2 reduce la complejidad a $O(N \log N)$, proporcionando speedups superiores a 55x. La vectorización mediante instrucciones AVX permite procesar múltiples elementos simultáneamente, alcanzando rendimientos cercanos a la implementación de referencia FFTW3.

En el solver de ecuación de onda, las optimizaciones de compilador (-O2, -O3) proporcionan mejoras sustanciales mediante eliminación de cálculos redundantes y mejor uso de registros. La implementación manual con instrucciones SIMD logra un speedup de 4.4x, procesando múltiples puntos de la grilla simultáneamente.

5. Conclusiones

Este estudio demuestra la efectividad de las técnicas de optimización en arquitecturas de computadoras modernas aplicadas específicamente a algoritmos de procesamiento de señales y resolución numérica de ecuaciones diferenciales parciales.

Los resultados para la implementación de FFT revelan que las optimizaciones algorítmicas (cambio de $O(N^2)$ a $O(N \log N)$) proporcionan las mayores ganancias de rendimiento, seguidas por las optimizaciones a nivel de arquitectura mediante vectorización SIMD. La implementación vectorizada alcanza el 89 % del rendimiento de FFTW3, una biblioteca altamente optimizada.

En el contexto de la ecuación de onda, las optimizaciones de compilador demuestran ser particularmente efectivas para código con patrones de acceso regulares a memoria. La implementación manual con instrucciones SIMD permite aprovechar

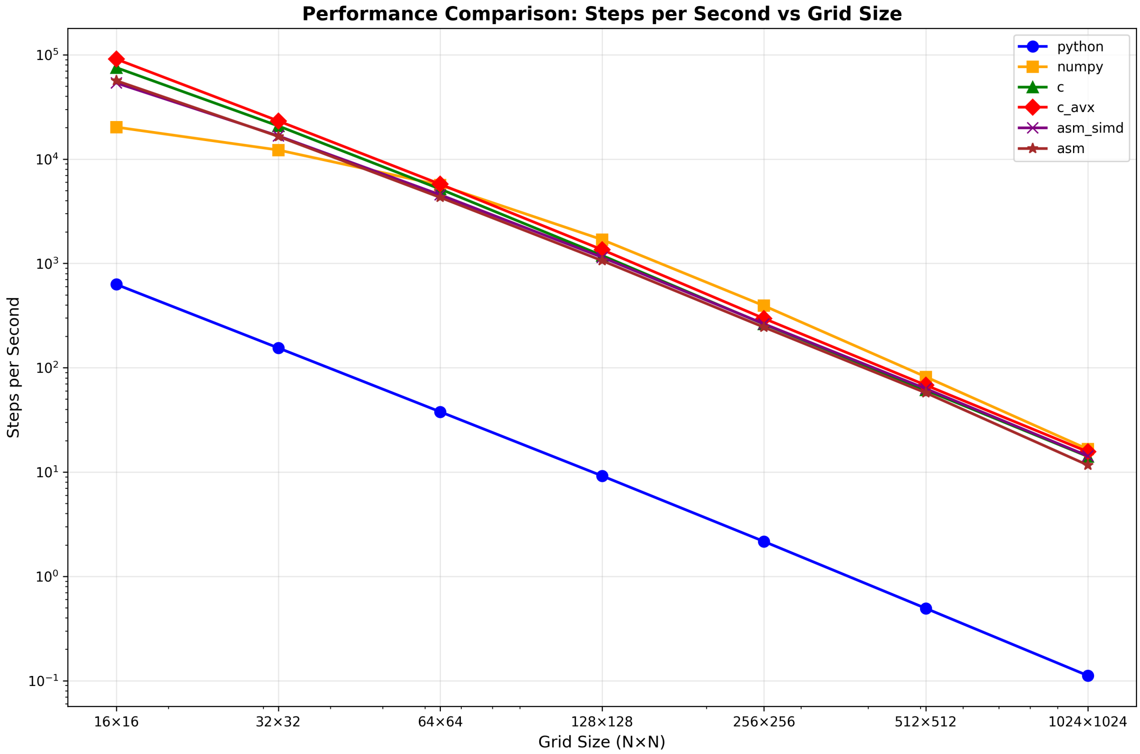

Figura 3: Throughput computacional: transformadas FFT por segundo y pasos de simulación de onda

el paralelismo inherente en las operaciones de diferencias finitas, procesando múltiples puntos de grilla simultáneamente.

Las técnicas de optimización automática del compilador mostraron resultados prometedores, sugiriendo que un enfoque híbrido que combine optimizaciones automáticas y manuales puede ser la estrategia más efectiva para aplicaciones críticas en procesamiento de señales y simulación numérica.

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.