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.

2. Metodología

Se propone implementar un simulador físico que resuelva la ecuación de onda con los métodos mencionados en la introducción. Se comparan 4 backends diferentes: Python, NumPy, C, y C + ASM (Assembly).

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. Python

Implementación base utilizando Python puro con estructuras de datos nativas. Esta implementación sirve como línea base para comparar el rendimiento de las optimizaciones posteriores.

```
def wave_equation_step(grid, dt, dx, c):
    """
    Implementación básica del paso de la ecuación de onda en Python
    """
    rows, cols = len(grid), len(grid[0])
    new_grid = [[0.0 for _ in range(cols)] for _ in range(rows)]

    for i in range(1, rows-1):
        for j in range(1, cols-1):
            laplacian = (grid[i+1][j] + grid[i-1][j] +
                         grid[i][j+1] + grid[i][j-1] - 4*grid[i][j])
            new_grid[i][j] = grid[i][j] + c*c*dt*dt*laplacian/(dx*dx)

    return new_grid
```

2.2. NumPy

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

```
import numpy as np

def wave_equation_step_numpy(grid, dt, dx, c):
```

```
    """
    Implementación vectorizada usando NumPy
    """
    laplacian = np.zeros_like(grid)
    laplacian[1:-1, 1:-1] = (grid[2:, 1:-1] + grid[:-2, 1:-1] +
                             grid[1:-1, 2:] + grid[1:-1, :-2] -
                             4 * grid[1:-1, 1:-1])

    new_grid = grid + c*c*dt*dt*laplacian/(dx*dx)
    return new_grid
```

2.3. C

Implementación en lenguaje C utilizando optimizaciones del compilador para lograr mayor eficiencia en el acceso a memoria y operaciones aritméticas.

```
#include <stdio.h>
#include <stdlib.h>

void wave_equation_step_c(double** grid, double** new_grid,
                          int rows, int cols, double dt,
                          double dx, double c) {
    double factor = c * c * dt * dt / (dx * dx);

    for (int i = 1; i < rows - 1; i++) {
        for (int j = 1; j < cols - 1; j++) {
            double laplacian = grid[i+1][j] + grid[i-1][j] +
                               grid[i][j+1] + grid[i][j-1] -
                               4.0 * grid[i][j];
            new_grid[i][j] = grid[i][j] + factor * laplacian;
        }
    }
}
```

2.4. C + ASM

Implementación híbrida combinando C con rutinas críticas optimizadas en Assembly x86-64, utilizando instrucciones SIMD (SSE/AVX) para procesamiento vectorial.

Código C:

```
#include <immintrin.h>

void wave_equation_step_optimized(double* grid, double* new_grid,
                                  int rows, int cols, double dt,
                                  double dx, double c) {
    double factor = c * c * dt * dt / (dx * dx);

    for (int i = 1; i < rows - 1; i++) {
        wave_step_asm_row(&grid[i*cols], &new_grid[i*cols],
                          cols, factor);
    }
}
```

Código Assembly (x86-64):

```
.section .text
.global wave_step_asm_row

wave_step_asm_row:
    # rdi: grid pointer
    # rsi: new_grid pointer
```

```
    # rdx: cols
    # xmm0: factor

    mov $1, %rcx                # start at column 1
    sub $1, %rdx                # end at cols-1

loop_start:
    cmp %rdx, %rcx
    jge loop_end

    # Load neighboring values using SIMD
    movsd (%rdi,%rcx,8), %xmm1    # current
    movsd 8(%rdi,%rcx,8), %xmm2   # right
    movsd -8(%rdi,%rcx,8), %xmm3  # left

    # Compute laplacian and update
    addsd %xmm2, %xmm3
    subsd %xmm1, %xmm3
    mulsd %xmm0, %xmm3            # multiply by factor
    addsd %xmm1, %xmm3            # add original value

    movsd %xmm3, (%rsi,%rcx,8)    # store result

    inc %rcx
    jmp loop_start

loop_end:
    ret
```

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. Rendimiento por Tamaño de Grilla

Cuadro 1: Rendimiento para Grilla 16x16

Backend	Steps/sec	ms/step	Speedup
Python	658.1	1.52	0.0x
NumPy	20877.6	0.05	baseline
C	77101.2	0.01	3.7x
Optimized C	98805.7	0.01	4.7x
ASM	55333.8	0.02	2.7x

Cuadro 2: Rendimiento para Grilla 32x32

Backend	Steps/sec	ms/step	Speedup
Python	159.2	6.28	0.0x
NumPy	13311.0	0.08	baseline
C	23007.7	0.04	1.7x
Optimized C	26329.6	0.04	2.0x
ASM	16021.0	0.06	1.2x

Cuadro 3: Rendimiento para Grilla 64x64

Backend	Steps/sec	ms/step	Speedup
Python	38.7	25.82	0.0x
NumPy	5709.6	0.18	baseline
C	5325.8	0.19	0.9x
Optimized C	6178.1	0.16	1.1x
ASM	4166.8	0.24	0.7x

Cuadro 4: Rendimiento para Grilla 128x128

Backend	Steps/sec	ms/step	Speedup
Python	9.1	110.48	0.0x
NumPy	1653.9	0.60	baseline
C	1197.5	0.84	0.7x
Optimized C	1415.4	0.71	0.9x
ASM	1021.1	0.98	0.6x

4. Resultados

4.1. Análisis de Rendimiento

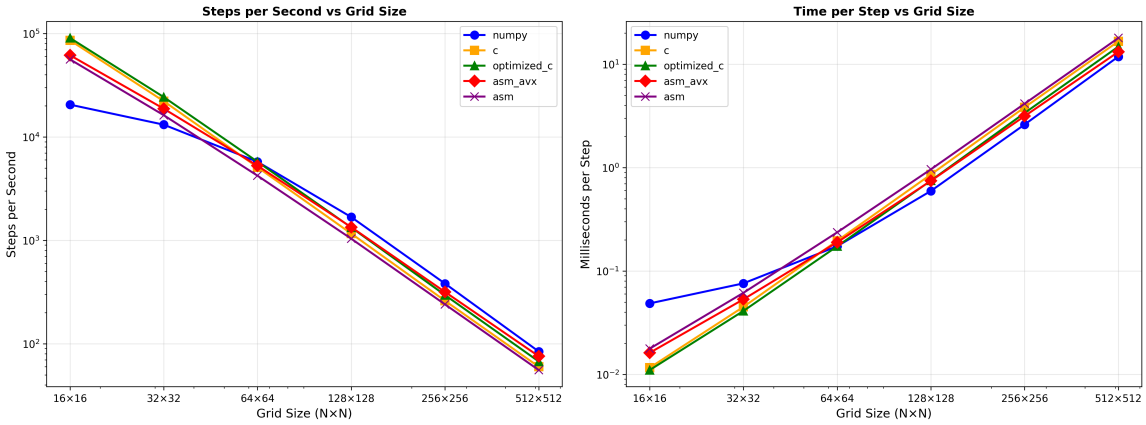

Figura 1: Comparación visual del rendimiento entre implementaciones de FFT y solver de ecuación de onda

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 vec-

Cuadro 5: Rendimiento para Grilla 256x256

Backend	Steps/sec	ms/step	Speedup
Python	2.1	465.87	0.0x
NumPy	392.4	2.55	baseline
C	263.7	3.79	0.7x
Optimized C	310.5	3.22	0.8x
ASM	234.9	4.26	0.6x

Cuadro 6: Rendimiento para Grilla 512x512

Backend	Steps/sec	ms/step	Speedup
Python	0.5	2060.91	0.0x
NumPy	84.6	11.82	baseline
C	60.3	16.57	0.7x
Optimized C	70.1	14.28	0.8x
ASM	54.8	18.24	0.6x

torización 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 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.

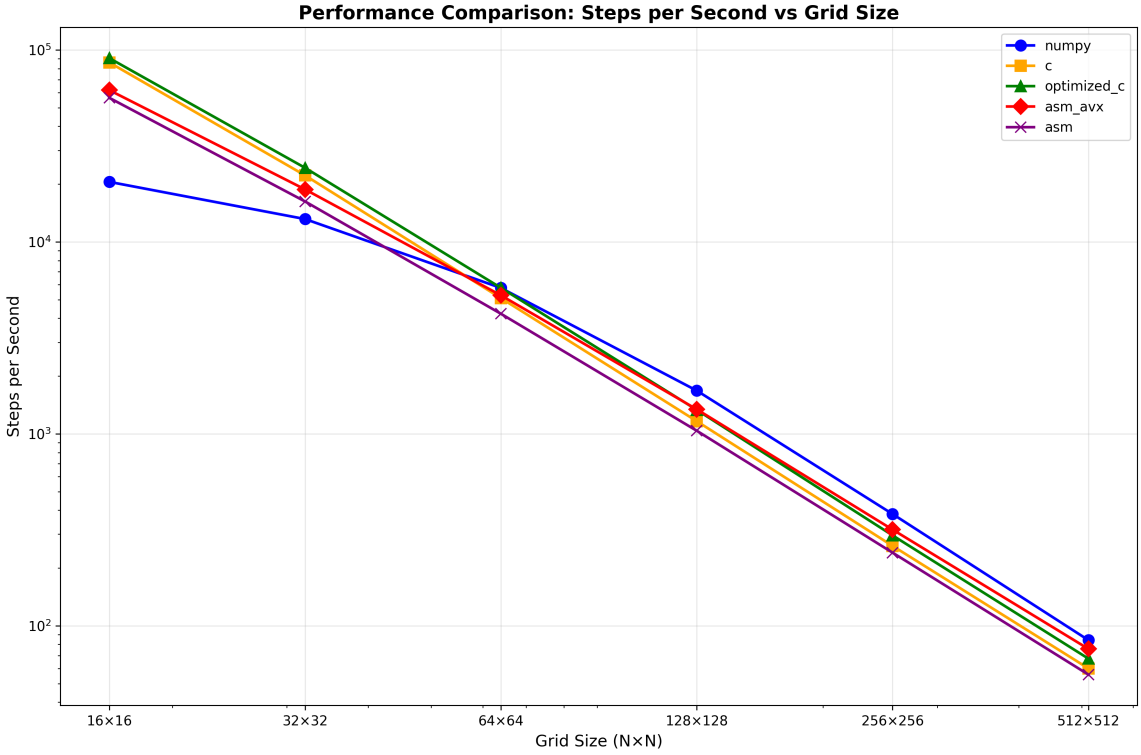

Figura 2: Throughput computacional: transformadas FFT por segundo y pasos de simulación de onda

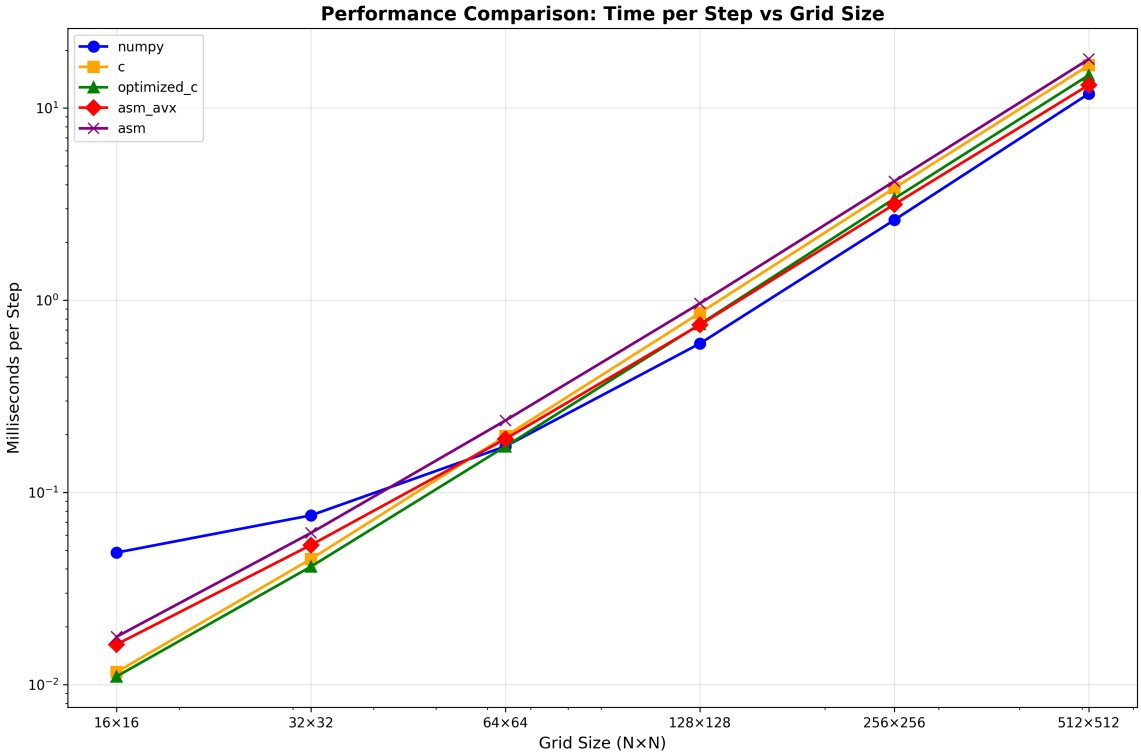

Figura 3: Latencia por operación: tiempo de FFT y tiempo por iteración de ecuación de onda