Paralelización en CUDA: Transformada de Fourier Discreta y Continua

Jesús Losada Arauzo

23 de mayo de 2025

Resumen

En este informe se analiza la paralelización mediante CUDA de la Transformada de Fourier Discreta (DFT) y versiones continuas con integración numérica. El objetivo es medir el rendimiento frente a la versión secuencial utilizando diferentes técnicas de integración: suma de rectángulos, trapecios y método de Simpson. Se emplea memoria unificada y eventos CUDA para una gestión sencilla de datos y tiempos.

Paralelización en CUDA

El programa implementa varios *kernels* CUDA, uno para cada versión de la transformada:

- **DFT**: versión discreta clásica.
- **CFT**: versión continua básica, usando suma directa.
- **CFT_Trapecio** y **CFT_Simpson**: integración continua por métodos numéricos.

Se utiliza una distribución fija de hilos: `BLOCK_SIZE = 256` y `NUM_BLOCKS = 10`. Cada hilo se encarga de calcular una componente del vector de Fourier (una frecuencia). Internamente, el hilo hace un bucle que suma o integra la función con el método correspondiente.

Distribución del trabajo

Cada hilo CUDA calcula un índice global:

```
i = blockIdx.x * blockDim.x + threadIdx.x
```

Con ese índice se decide si se procesa la componente i del vector de salida. De este modo, la carga de trabajo se distribuye automáticamente entre los hilos, permitiendo ejecutar en paralelo todos los valores de la transformada.

Medición de tiempo

El tiempo se mide mediante `cudaEventRecord`, lo cual permite medir con precisión únicamente la ejecución del kernel (sin contar asignaciones ni E/S). El resultado se almacena en archivos para su posterior graficado.

Resultados obtenidos

Las siguientes gráficas muestran los tiempos medidos en milisegundos para distintos tamaños de muestra. Se comparan versiones secuenciales y paralelas.

- Para muestras pequeñas, CUDA tiene cierta penalización inicial.
- A partir de cierto tamaño, la versión paralela supera ampliamente a la secuencial.

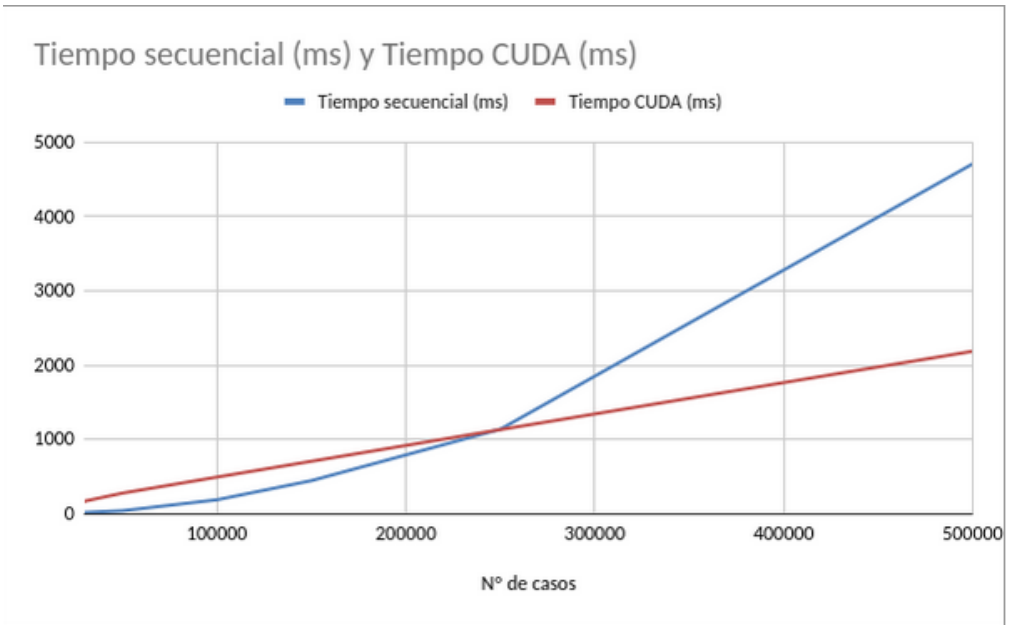

Figura 1: Tiempo de ejecución - DFT (Transformada Discreta)

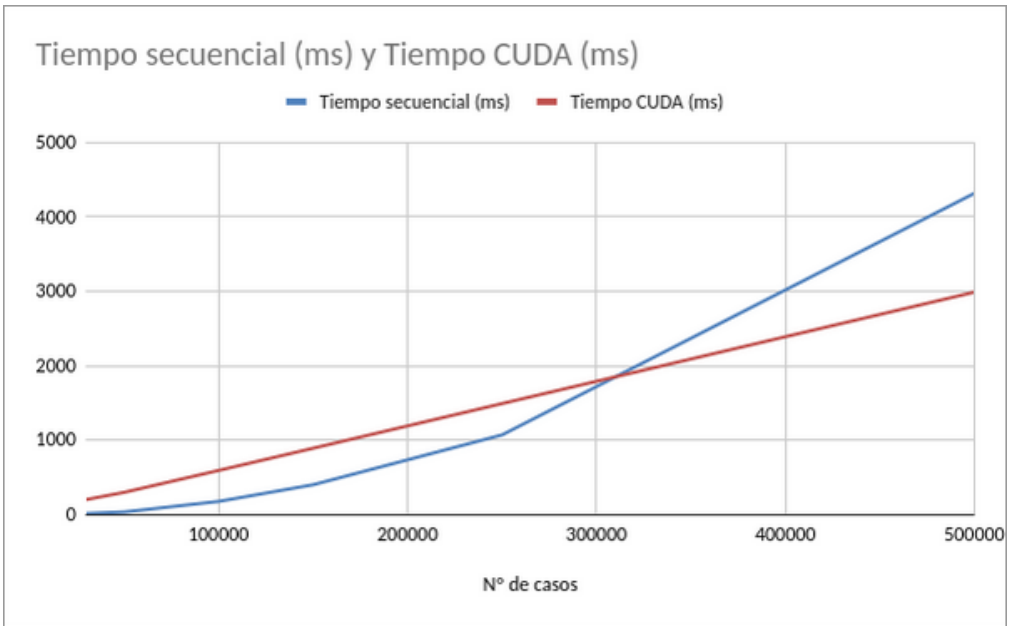

Figura 2: Tiempo de ejecución - Suma de Rectángulos

Código CUDA resumido

```
__global__ void DFT(cuDoubleComplex *Fourier, const double *muestras, int N){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N){
        cuDoubleComplex sum = make_cuDoubleComplex(0.0, 0.0);
        for (int j = 0; j < N; j++){
            double angle = -2.0 * PI * i * j / N;
            cuDoubleComplex term = make_cuDoubleComplex(muestras[j]*cos(angle),
                                                        muestras[j]*sin(angle));
            sum = cuCadd(sum, term);
        }
        Fourier[i] = sum;
    }
}

__global__ void CFT(cuDoubleComplex *Fourier, const double *muestras, const int
    ↪ TAM_VECTOR_MUESTRAS, const double paso_temporal){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < TAM_VECTOR_MUESTRAS){
        Fourier[i] = make_cuDoubleComplex(0.0,0.0);
        cuDoubleComplex sum = make_cuDoubleComplex(0.0,0.0);
        double omega = 2.0*PI*i/(T_MAX-T_MIN);//La w de la formula que es el omega
        //Aqui ya entra en juego tanto el intervalo del tiempo como los valores
            ↪ que dan la funcion, es decir el tiempo esta entre -1 y 1
        //y los valores de la funcion estn en mis muestras
```

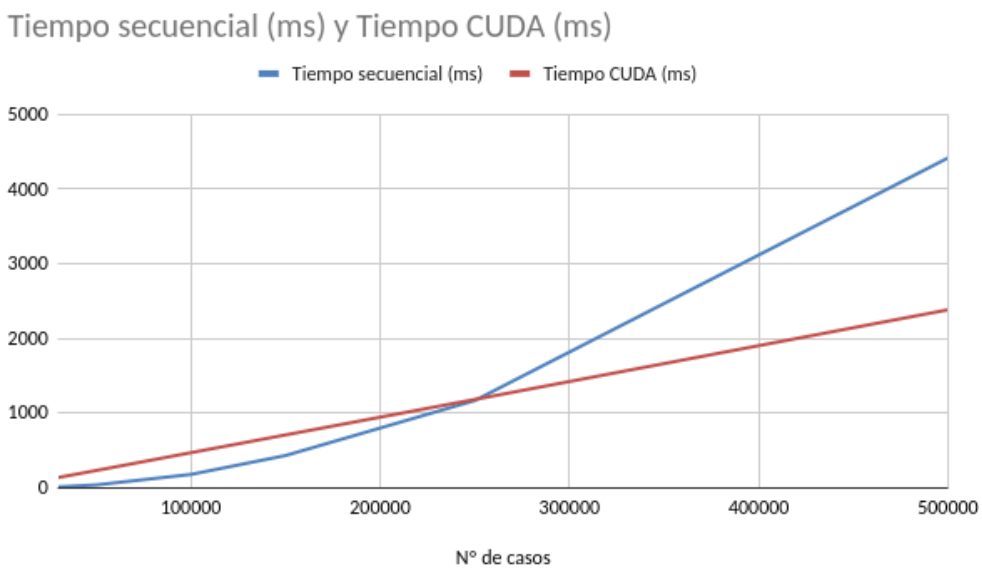

Figura 3: Tiempo de ejecución - Método del Trapecio

```
//Vamos desde el minimo hasta el maximo pero con nuestro paso temporal
    ↪ para tomar fourier lo ms preciso posible
for (double j = T_MIN;j<T_MAX; j= j+paso_temporal){
    //printf("Estoy en el segundo FOR");
    int indice = (int)((j - T_MIN)/paso_temporal);
    if (indice <= 0 ){
        // Si es menor o igual a 0, suponemos que coge el primer elemento
        cuDoubleComplex expo = cuCexp(make_cuDoubleComplex(0.0, -omega * j)
            ↪ );
        cuDoubleComplex temp = make_cuDoubleComplex(muestras[0], 0.0);
        cuDoubleComplex prod = cuCmul(temp, expo);
        sum = cuCadd(sum, cuCmulReal(prod, paso_temporal));
    }
    else if (indice >= TAM_VECTOR_MUESTRAS - 1) {
        // Si es mayor o igual al nmero de elementos, cogemos el ltimo
        cuDoubleComplex expo = cuCexp(make_cuDoubleComplex(0.0, -omega * j)
            ↪ );
        cuDoubleComplex temp = make_cuDoubleComplex(muestras[
            ↪ TAM_VECTOR_MUESTRAS - 1], 0.0);
        cuDoubleComplex prod = cuCmul(temp, expo);
        sum = cuCadd(sum, cuCmulReal(prod, paso_temporal));
    } else {
        // Si no es vlido y cogemos el valor calculado
        cuDoubleComplex expo = cuCexp(make_cuDoubleComplex(0.0, -omega * j)
            ↪ ); // Usamos la funcin cuCexp para la exponencial
        cuDoubleComplex temp = make_cuDoubleComplex(muestras[indice], 0.0);
            ↪  // Tomamos la muestra correspondiente
```

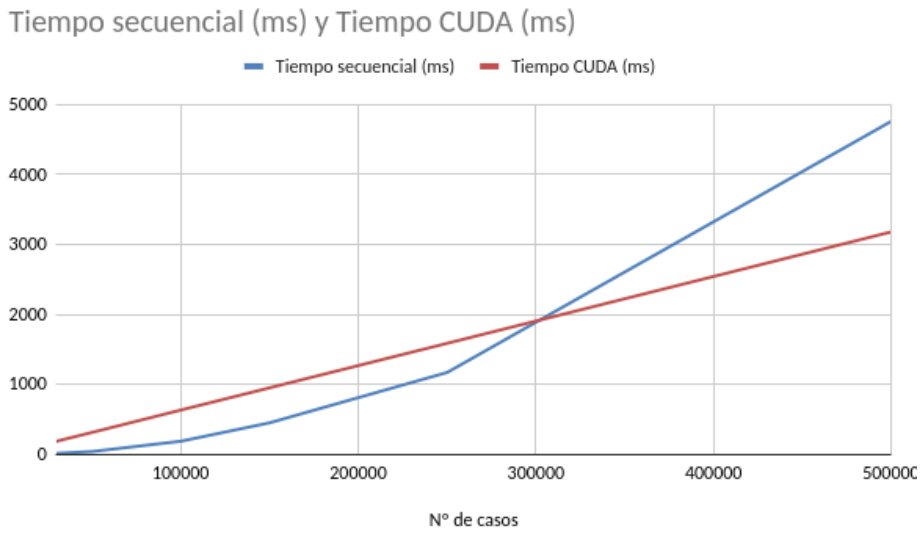

Figura 4: Tiempo de ejecución - Método de Simpson

```
                cuDoubleComplex prod = cuCmul(temp, expo); // Multiplicamos la
                    ↪ muestra por la exponencial
                sum = cuCadd(sum, cuCmulReal(prod, paso_temporal)); // Acumulamos
                    ↪ el resultado, aplicando el paso temporal
            }
        }
        Fourier[i] = sum;
    }
}

__global__ void CFT_Simpson(cuDoubleComplex *Fourier, const double *muestras,
    ↪ const int TAM_VECTOR_MUESTRAS, const double paso_temporal){

        int i = blockIdx.x * blockDim.x + threadIdx.x;
        if (i < TAM_VECTOR_MUESTRAS){
            Fourier[i] = make_cuDoubleComplex(0.0,0.0);
            cuDoubleComplex sum = make_cuDoubleComplex(0.0,0.0);
            double omega = 2.0*PI*i/(T_MAX-T_MIN);//La w de la formula que es el
                ↪ omega

            for (double j = T_MIN;j<T_MAX - paso_temporal; j += paso_temporal){
                int indice1 = (int)((j - T_MIN)/paso_temporal);
                int indice2 = indice1 + 1;

                if (indice2 >= TAM_VECTOR_MUESTRAS) indice2 = TAM_VECTOR_MUESTRAS -
                    ↪  1;

                double x_medio = j + paso_temporal / 2.0;
```

```
            int indice_medio = (int)((x_medio - T_MIN) / paso_temporal);

            if (indice_medio >= TAM_VECTOR_MUESTRAS) indice_medio =
                ↪ TAM_VECTOR_MUESTRAS - 1;

            double simpson = (muestras[indice1] + 4.0*muestras[indice_medio] +
                ↪ muestras[indice2])/6.0;

            //Aqui la parte nueva adems de distribuirlo para cada
            cuDoubleComplex prod = cuCexp(make_cuDoubleComplex(0.0, -omega * j)
                ↪ );
            cuDoubleComplex temp = make_cuDoubleComplex(simpson, 0.0);
            cuDoubleComplex prod2 = cuCmul(temp, prod);
            sum = cuCadd(sum,cuCmulReal(prod2,paso_temporal));


        }

        Fourier[i] = sum;

    }


}

__global__ void CFT_Trapecio(cuDoubleComplex *Fourier, const double *muestras,
    ↪ const int TAM_VECTOR_MUESTRAS, const double paso_temporal){

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < TAM_VECTOR_MUESTRAS){
        Fourier[i] = make_cuDoubleComplex(0.0,0.0);
        cuDoubleComplex sum = make_cuDoubleComplex(0.0,0.0);
        double omega = 2.0*PI*i/(T_MAX-T_MIN);//La w de la formula que es el omega

        for(double j = T_MIN; j < T_MAX - paso_temporal; j += paso_temporal){
            int indice1 = (int)((j - T_MIN) / paso_temporal);
            int indice2 = indice1 + 1;

            if (indice2 >= TAM_VECTOR_MUESTRAS) indice2 = TAM_VECTOR_MUESTRAS - 1;

            double promedio = (muestras[indice1] + muestras[indice2])/2.0;

            //Aqui la parte nueva adems de distribuirlo para cada
            cuDoubleComplex prod = cuCexp(make_cuDoubleComplex(0.0, -omega * j));
            cuDoubleComplex temp = make_cuDoubleComplex(promedio, 0.0);
            cuDoubleComplex prod2 = cuCmul(temp, prod);
            sum = cuCadd(sum,cuCmulReal(prod2,paso_temporal));
```

```
        }

        Fourier[i] = sum;

    }


}
```