ICNPG 2023

Transformadas de Fourier

¿ Que es la CUFFT ?

The **FFT** is a divide-and-conquer algorithm for efficiently computing discrete Fourier transforms of complex or real-valued data sets. It is one of the most important and widely used numerical algorithms in computational physics and general signal processing. The CUFFT library provides a simple interface for computing FFTs on an NVIDIA GPU, which allows users to quickly leverage the floating-point power and parallelism of the GPU in a highly optimized and tested FFT library.

¿ Que es la CUFFT ?

This document describes cuFFT, the NVIDIA® CUDA™ Fast Fourier Transform (FFT) product. It consists of two separate libraries: cuFFT and cuFFTW. The cuFFT library is designed to provide high performance on NVIDIA GPUs. The cuFFTW library is provided as a porting tool to enable users of FFTW to start using NVIDIA GPUs with a minimum amount of effort.

The cuFFT product supports a wide range of FFT inputs and options efficiently on NVIDIA GPUs. This version of the cuFFT library supports the following features:

- Algorithms highly optimized for input sizes that can be written in the form $2^a \times 3^b \times 5^c \times 7^d$. In general the smaller the prime factor, the better the performance, i.e., powers of two are fastest.
- An $O(n \log n)$ algorithm for every input data size
- Half-precision (16-bit floating point), single-precision (32-bit floating point) and double-precision (64-bit floating point). Transforms of lower precision have higher performance.
- Complex and real-valued input and output. Real valued input or output require less computations and data than complex values and often have faster time to solution. Types supported are:
  - C2C - Complex input to complex output
  - R2C - Real input to complex output
  - C2R - Symmetric complex input to real output
- 1D, 2D and 3D transforms
- Execution of multiple 1D, 2D and 3D transforms simultaneously. These batched transforms have higher performance than single transforms.
- In-place and out-of-place transforms
- Arbitrary intra- and inter-dimension element strides (strided layout)

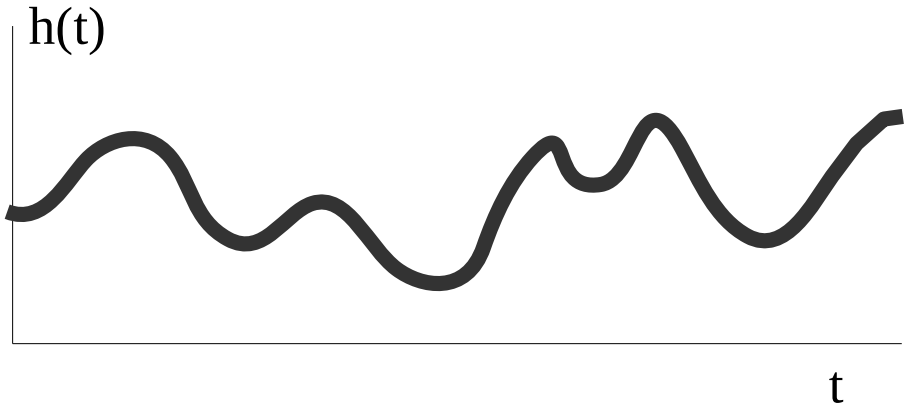
Por ej esto es necesario hacerlo para sacar el espectro de un audio a tiempo real

Si tenemos una señal muy larga y queremos hacer transformadas de a pedazos, podemos decirle como input. Esto es más eficiente que mandar en serie transformadas de señales pequeñas

modifica o no el array de entrada

Se pueden acceder a los datos de forma no continua, "pegando saltos" sin tener que reordenarlos

Repaso de Transformada de Fourier



"Descomposición en senos y cosenos"

$$H(\omega) = \int_{-\infty}^{\infty} h(t) e^{i\omega t} dt$$

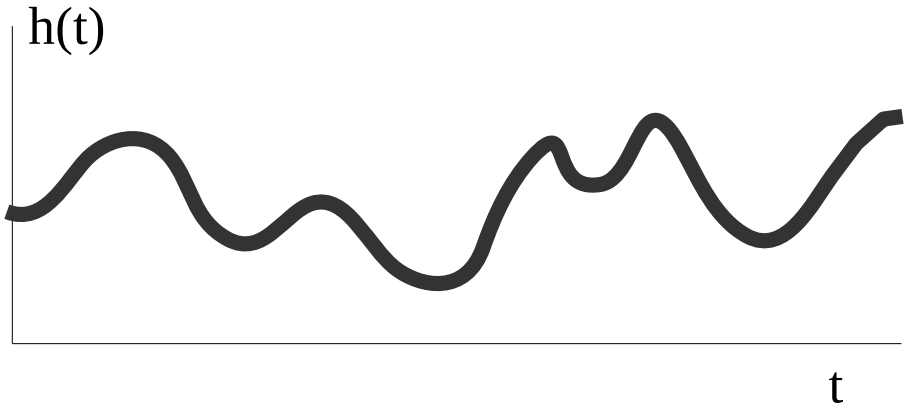
$$h(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} H(\omega) e^{-i\omega t} d\omega$$

Ver Por ej:
Press etal, Numerical Recipes

Simetrías

If . . .	then . . .
$h(t)$ is real	$H(-f) = [H(f)]^*$
$h(t)$ is imaginary	$H(-f) = -[H(f)]^*$
$h(t)$ is even	$H(-f) = H(f)$ [i.e., $H(f)$ is even]
$h(t)$ is odd	$H(-f) = -H(f)$ [i.e., $H(f)$ is odd]
$h(t)$ is real and even	$H(f)$ is real and even
$h(t)$ is real and odd	$H(f)$ is imaginary and odd
$h(t)$ is imaginary and even	$H(f)$ is imaginary and even
$h(t)$ is imaginary and odd	$H(f)$ is real and odd

Repaso de Transformada de Fourier



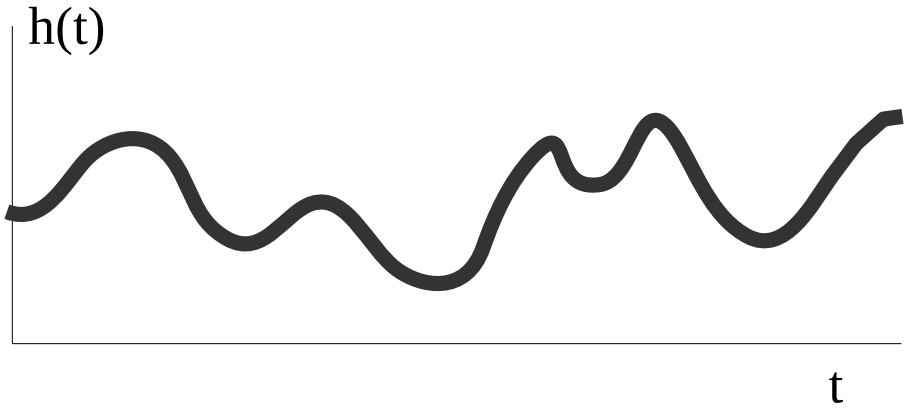
$$H(\omega) = \int_{-\infty}^{\infty} h(t) e^{i\omega t} dt$$

$$h(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} H(\omega) e^{-i\omega t} d\omega$$

Esto es útil si tenemos que reescalear nuestro problema

$h(at) \Longleftrightarrow \frac{1}{\lvert a \rvert} H(\frac{f}{a})$	time scaling
$\frac{1}{\lvert b \rvert} h(\frac{t}{b}) \Longleftrightarrow H(bf)$	frequency scaling
$h(t - t_0) \Longleftrightarrow H(f)\, e^{2\pi i f t_0}$	time shifting
$h(t)\, e^{-2\pi i f_0 t} \Longleftrightarrow H(f - f_0)$	frequency shifting

Repaso de Transformada de Fourier



"Descomposición en senos y cosenos"

$$H(\omega) = \int_{-\infty}^{\infty} h(t) e^{i\omega t} dt$$

$$h(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} H(\omega) e^{-i\omega t} d\omega$$

Porque es tan importante la transformada de Fourier en física computacional ?

Convolution theorem

$$g * h \equiv \int_{-\infty}^{\infty} g(\tau) h(t-\tau)\, d\tau$$

Operador...

Interacción...

$$g * h \iff G(f) H(f)$$

Multiplicación "element-wise"

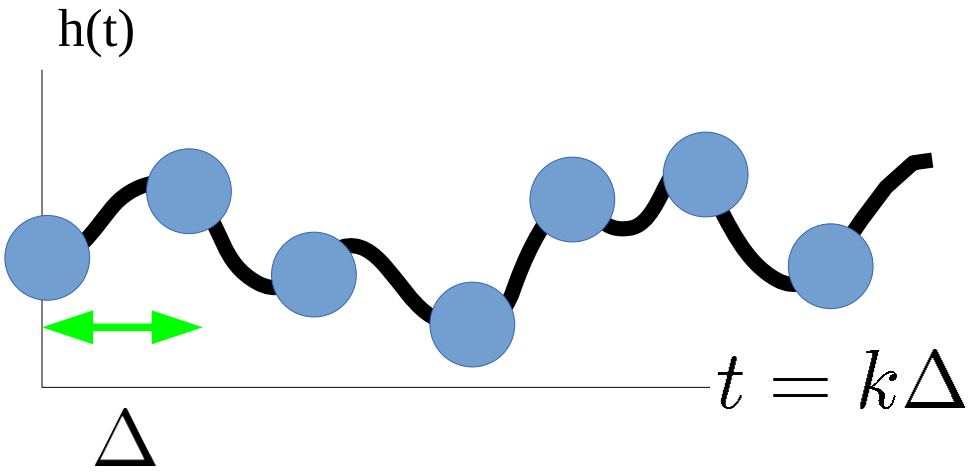
Correlation theorem

$$\mathrm{Corr}(g,h) \equiv \int_{-\infty}^{\infty} g(\tau + t) h(\tau)\, d\tau$$

$$\mathrm{Corr}(g,h) \iff G(f) H^*(f)$$

Transformada de Fourier discreta

$$h_k \equiv h(t_k), \qquad t_k \equiv k\Delta, \qquad k = 0, 1, 2, \ldots, N-1$$



Sampling theorem

$$H(f_n) = \int_{-\infty}^{\infty} h(t) e^{2\pi i f_n t}\, dt \approx \sum_{k=0}^{N-1} h_k\, e^{2\pi i f_n t_k}\, \Delta = \Delta \sum_{k=0}^{N-1} h_k\, e^{2\pi i k n/N}$$

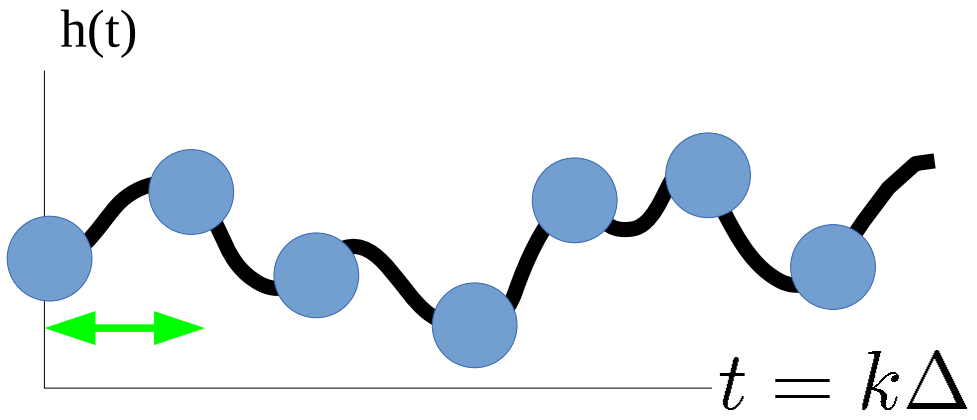
$$f_n \equiv \frac{n}{N\Delta}, \qquad n = -\frac{N}{2}, \ldots, \frac{N}{2}$$

Transformada discreta

$$H_n \equiv \sum_{k=0}^{N-1} h_k\, e^{2\pi i k n/N}$$

Transformada de Fourier discreta

$$h_k \equiv h(t_k), \qquad t_k \equiv k\Delta, \qquad k = 0, 1, 2, \ldots, N-1$$



Transformada discreta

$$H_n \equiv \sum_{k=0}^{N-1} h_k \, e^{2\pi i k n / N}$$

Anti-Transformada

$$h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n \, e^{-2\pi i k n / N}$$

Se podría poner de distinta forma... con un 1/sqrt(N) en ambos lados

Transformada de Fourier discreta

Transformada discreta 1D

$$H_n \equiv \sum_{k=0}^{N-1} h_k \, e^{2\pi i k n / N}$$

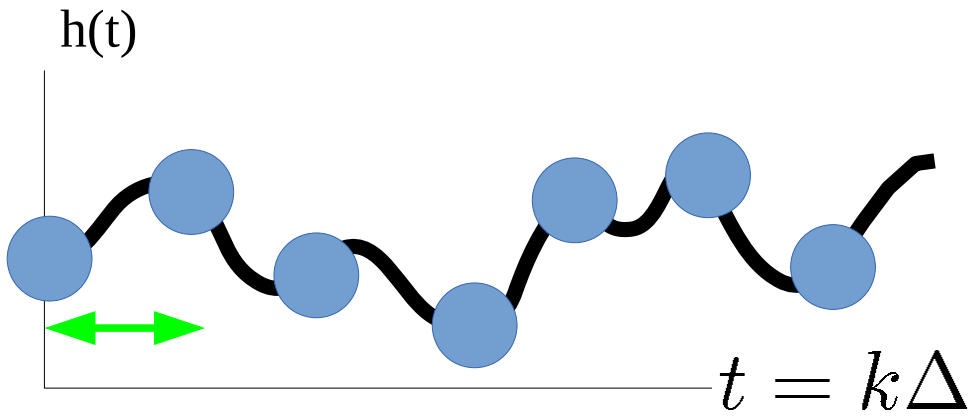
Transformada discreta 2D

$$H(n_1, n_2) \equiv \sum_{k_2=0}^{N_2-1} \sum_{k_1=0}^{N_1-1} \exp(2\pi i k_2 n_2 / N_2) \, \exp(2\pi i k_1 n_1 / N_1) \, h(k_1, k_2)$$

Transformada discreta 3D, etc

Transformada de Fourier discreta

$$h_k \equiv h(t_k), \qquad t_k \equiv k\Delta, \qquad k = 0, 1, 2, \ldots, N-1$$



Transformada discreta

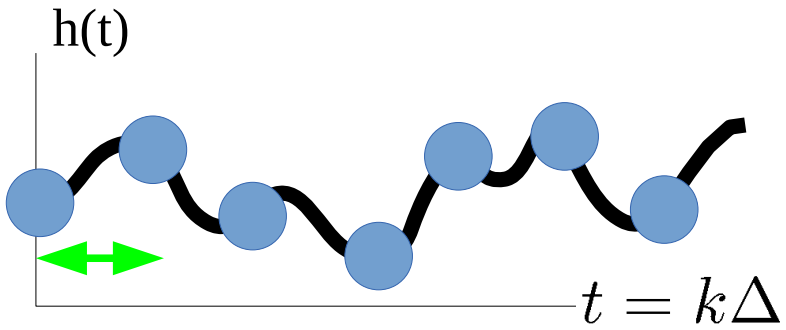
$$H_n \equiv \sum_{k=0}^{N-1} h_k \, e^{2\pi i k n/N}$$

Anti-Transformada

$$h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n \, e^{-2\pi i k n/N}$$

Transformada de Fourier discreta

$$h_k \equiv h(t_k), \qquad t_k \equiv k\Delta, \qquad k = 0, 1, 2, \ldots, N-1$$



Transformada discreta

$$H_n \equiv \sum_{k=0}^{N-1} h_k \, e^{2\pi i k n / N}$$

Anti-Transformada

$$h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n \, e^{-2\pi i k n / N}$$

Como calcularlas eficientemente?

La transformación es una transformación lineal

Algoritmos Secuenciales

- Naïve approach [hasta ~1960]: O(N*N)
- Algoritmos de *Fast Fourier transform (FFT)* [Cooley–Tukey o Gauss?] : O(N*log2 N) → recursivo

Fast Fourier Transforms

Está en C y C++

La librería pública de FFT mas popular:

"La mas rápida del oeste"

Our benchmarks, performed on on a variety of platforms, show that FFTW's performance is typically superior to that of other publicly available FFT software, and is even competitive with vendor-tuned codes. In contrast to vendor-tuned codes, however, FFTW's performance is portable: the same program will perform well on most architectures without modification. Hence the name, "FFTW," which stands for the somewhat whimsical title of "Fastest Fourier Transform in the West."

Anda en Intel, AMD, ...

Otra librería... pero paga...

Vendor-tuned codes:

MKL: Math Kernel Library

Intel's Math Kernel Library (MKL) is a library of optimized, math routines for science, engineering, and financial applications. Core math functions include BLAS, LAPACK, ScaLAPACK, Sparse Solvers**, Fast Fourier Transforms**, and Vector Math.

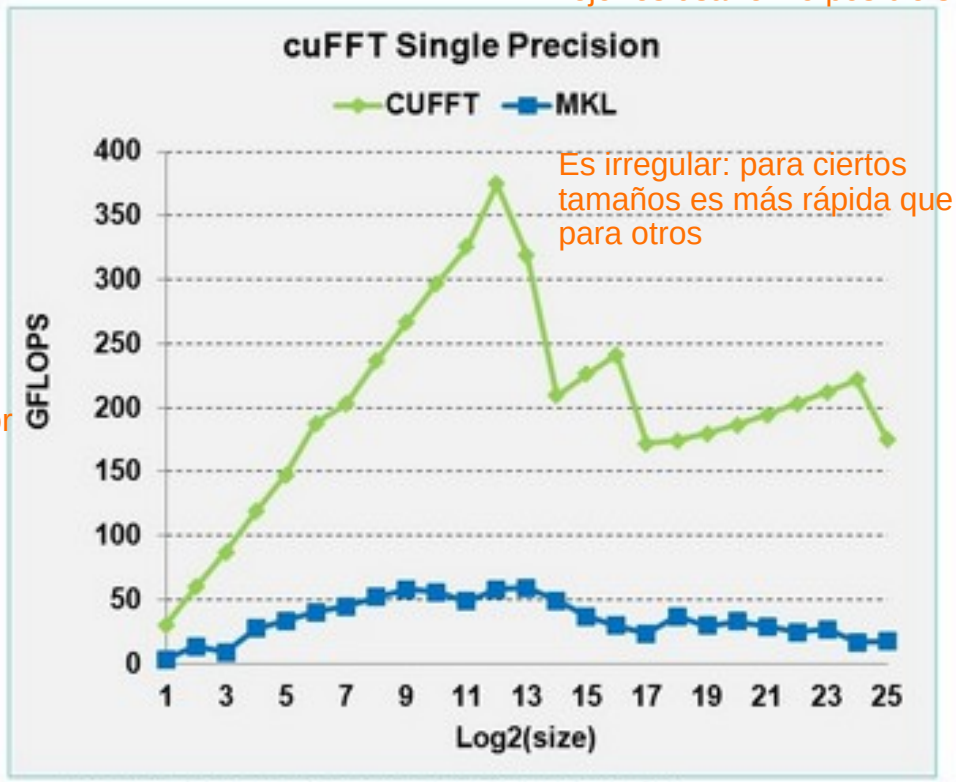
The library supports Intel and compatible processors and is available for Windows, Linux and Mac OS X operating systems.

CUFFT

FFTS UP TO 10X FASTER THAN MKL

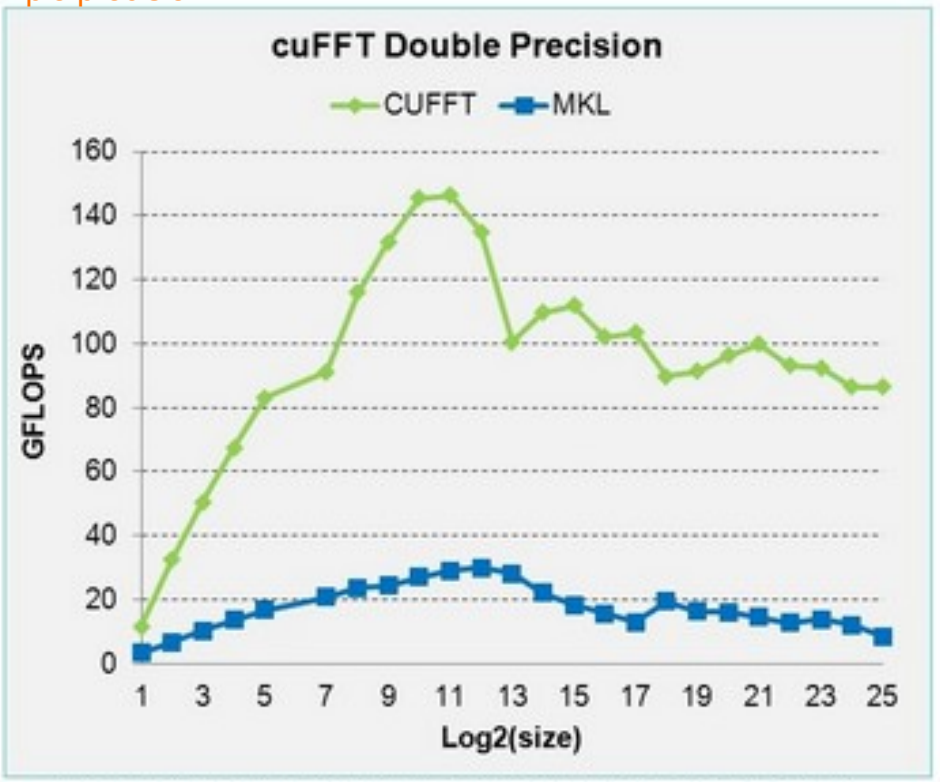
1D used in audio processing and as a foundation for 2D&3D FFTs

Cambia la velocidad entre simple y doble precisión, lo mejor es usar en lo posible simple precisión



Es irregular: para ciertos tamaños es más rápida que para otros

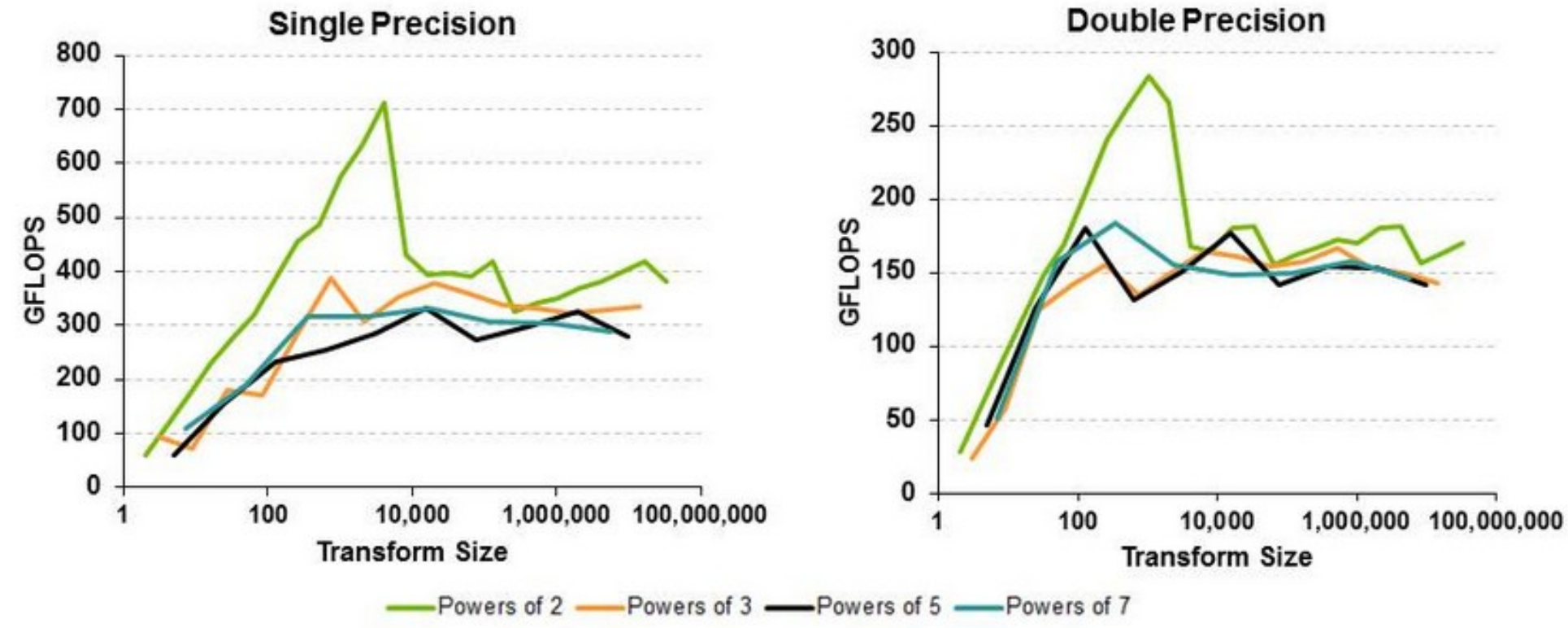
Cuántas operaciones puede hacer por segundo



- Measured on sizes that are exactly powers-of-2
- cuFFT 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz
- Performance may vary based on OS version and motherboard configuration

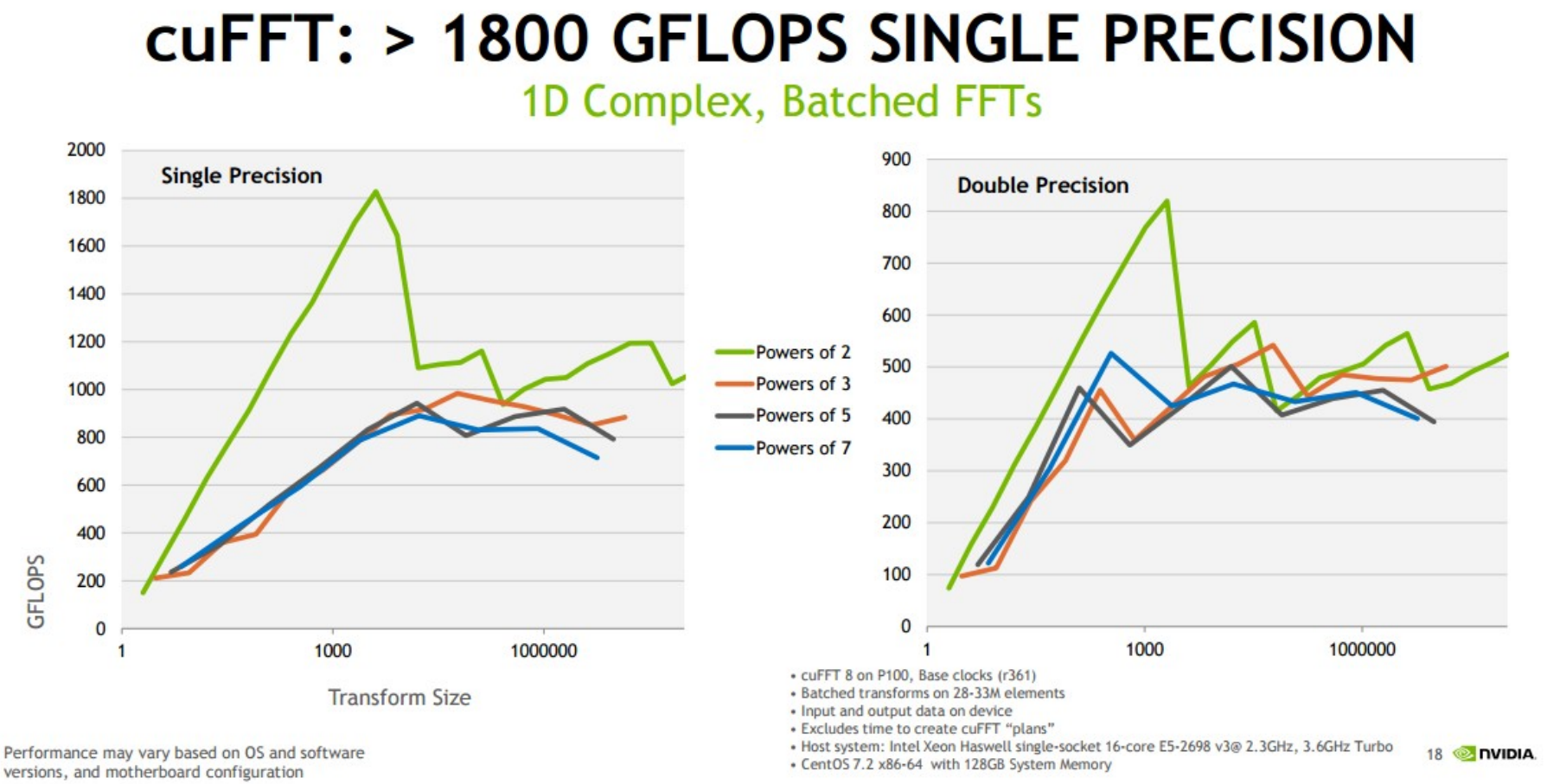
CUFFT



Performance may vary based on OS version and motherboard configuration

• cuFFT 6.0 on K40c, ECC ON, 28M-33M elements, input and output data on device

De 700 (K40) a 1800 (P100) GFLOPS



CUFFT Samples

simpleCUFFT - Simple CUFFT

Example of using CUFFT. In this example, CUFFT is used to compute the 1D-convolution of some signal with some filter by transforming both into frequency domain, multiplying them together, and transforming the signal back to time domain.

simpleCUFFT_2d_MGPU - SimpleCUFFT_2d_MGPU

Example of using CUFFT. In this example, CUFFT is used to compute the 1D-convolution of some signal with some filter by transforming both into frequency domain, multiplying them together, and transforming the signal back to time domain on Multiple GPU.

simpleCUFFT_callback - Simple CUFFT Callbacks

Example of using CUFFT. In this example, CUFFT is used to compute the 1D-convolution of some signal with some filter by transforming both into frequency domain, multiplying them together, and transforming the signal back to time domain. The difference between this example and the Simple CUFFT example is that the multiplication step is done by the CUFFT kernel with a user-supplied CUFFT callback routine, rather than by a separate kernel call.

simpleCUFFT_MGPU - Simple CUFFT_MGPU

Example of using CUFFT. In this example, CUFFT is used to compute the 1D-convolution of some signal with some filter by transforming both into frequency domain, multiplying them together, and transforming the signal back to time domain on Multiple GPU.

Ejemplo de CUFFT

```
3D Complex-to-Complex Transforms

#define NX 64
#define NY 64
#define NZ 128

cufftHandle plan;
cufftComplex *data1, *data2;
cudaMalloc((void**)&data1, sizeof(cufftComplex)*NX*NY*NZ);
cudaMalloc((void**)&data2, sizeof(cufftComplex)*NX*NY*NZ);
/* Create a 3D FFT plan. */
cufftPlan3d(&plan, NX, NY, NZ, CUFFT_C2C);

/* Transform the first signal in place. */
cufftExecC2C(plan, data1, data1, CUFFT_FORWARD);

/* Transform the second signal using the same plan. */
cufftExecC2C(plan, data2, data2, CUFFT_FORWARD);

/* Destroy the cuFFT plan. */
cufftDestroy(plan);
cudaFree(data1); cudaFree(data2);
```

Este plan informa a la librería qué queremos hacer y la librería se encarga de buscar el mejor algoritmo

Se le informa qué tipo de transformada se va a hacer

Se está asumiendo que los datos están ordenados continuamente (que no hay stride)

No hace falta un plan por transformada, pero tiene que ser el mismo tipo de datos

Ejemplo de CUFFT

```
Batched transforms

#define NX 256
#define BATCH 10
#define RANK 1
...
{
    cufftHandle plan;
    cufftComplex *data;
    ...
    cudaMalloc((void**)&data,sizeof(cufftComplex)*NX*BATCH);

    cufftPlanMany(&plan, RANK, NX, &iembed, istride, idist,
    &oembed, ostride, odist, CUFFT_C2C, BATCH);
    ...
    cufftExecC2C(plan, data, data, CUFFT_FORWARD);
    cudaDeviceSynchronize();
    ...
    cufftDestroy(plan);
    cudaFree(data);
}
```

CUFFT y Thrust

```
#include <cufft.h>
typedef cufftDoubleReal REAL;
typedef cufftDoubleComplex COMPLEX;
// includes de thrust

int main(){
        // declaro/aloco arrays input, y su transformada, output
        thrust::device_vector<REAL> input(N);
        thrust::device_vector<COMPLEX> output(N/2+1);
        // llenar input[] con alguna señal ...

        // cast a punteros normales a memoria del device
        REAL * d_input_raw = thrust::raw_pointer_cast(&input[0]);
         COMPLEX * d_output_raw = thrust::raw_pointer_cast(&output[0]);

        cufftHandle planfft; // declara un plan ...
        cufftPlan1d(&planfft,N,CUFFT_R2C,1); // lo inicializa ...
        cufftExecR2C(planfft, d_input_raw, d_output_raw);  // y lo ejecuta ...

        // copio la transformada del device al host, y la imprimo
        thrust::host_vector<COMPLEX> output_host = output;
        for(i=0; i < N/2+1; i++)  cout << output_host[i].x << " " << output_host[i].y << endl;
}
```

Debería tener como output N elementos PERO como son complejos, realmente tengo el doble de datos. El "+1" está por las dudas si N es impar. Además, todas las frecuencias negativas están consideradas dadas las propiedades de simetría de la transformada de fourier

Ejemplo 1: Transformada 1d

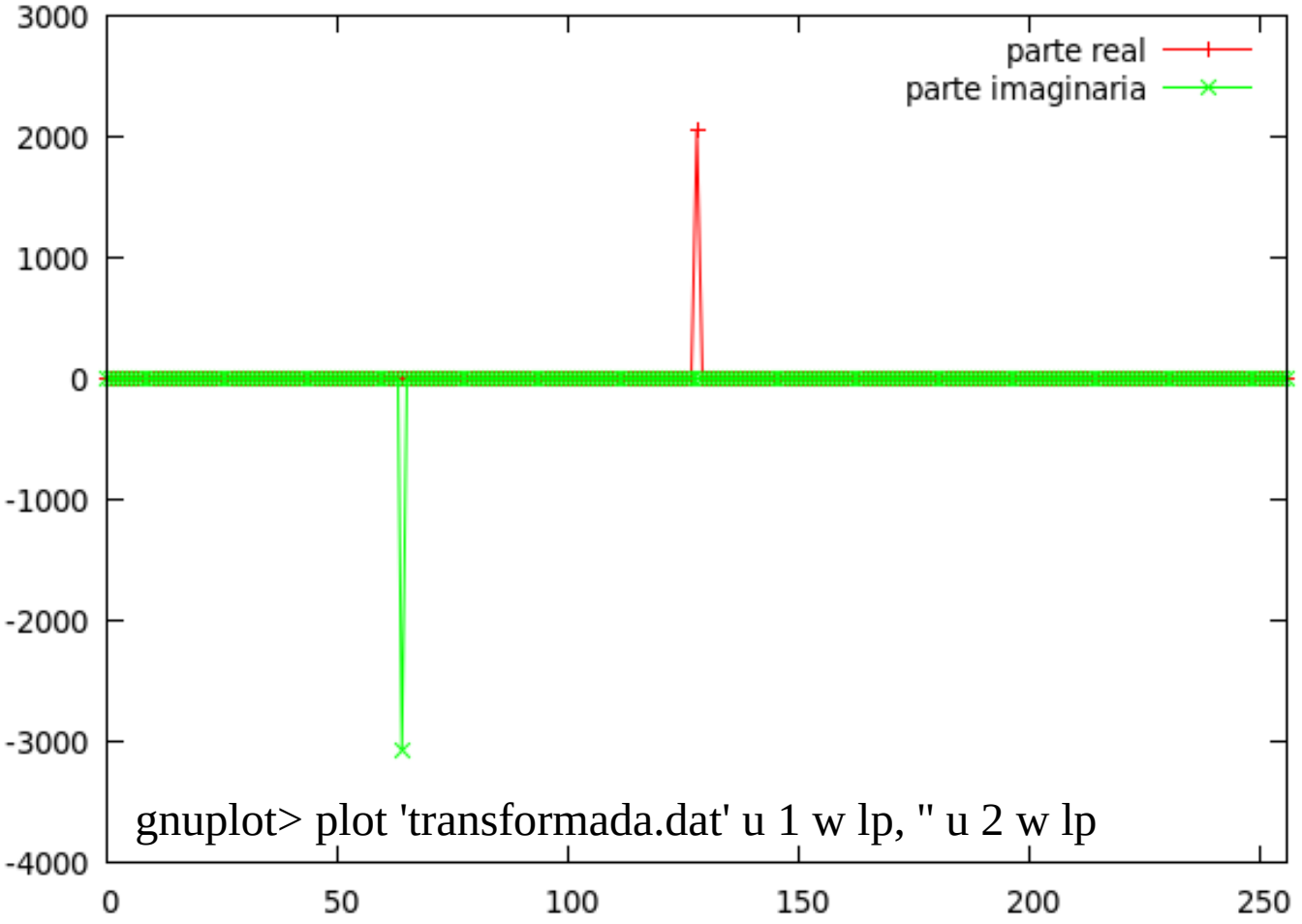
- /share/apps/icnpg/clases/clase11/Simple_cufft_fftw_thrust
- **simple_cufft.cu**: transformada en CUDA
- Compilar y Correr simple_cufft:
  - **¿Es correcta la transformada?**

simple_cufft.cu

Ejemplo 1: Transformada 1d

```
#define N 512
#define A1 4
#define A2 6
#define T1 N/4
#define T2 N/8
```

// la transformada de este array
Input[i] = 2 A1 cos(2*M_PI*i*T1/N) + 2 A2 sin(2*M_PI*i*T2/N);
// **¿es esto?** :



Ejemplo 1: Transformada 1d

simple_cufft.cu

```
#include <cufft.h>
```

Ejemplo 1: Transformada 1d

simple_cufft.cu

```
#ifdef DOUBLE_PRECISION
typedef cufftDoubleReal REAL;
typedef cufftDoubleComplex COMPLEX;
#else
typedef cufftReal REAL;
typedef cufftComplex COMPLEX;
#endif
```

Ejemplo 1: Transformada 1d

simple_cufft.cu

Estructuras de datos que vamos a necesitar

```
// Un container de thrust para guardar el input real en GPU
thrust::device_vector<REAL> D_input(N);

// toma el raw_pointer del array de input, para pasarselo a CUFFT luego
REAL *d_input = thrust::raw_pointer_cast(&D_input[0]);

// Un container de thrust para guardar el ouput complejo en GPU = transformada del input
int Ncomp=N/2+1;
thrust::device_vector<COMPLEX> D_output(Ncomp);

// toma el raw_pointer del array de output, para pasarselo a CUFFT luego
COMPLEX *d_output = thrust::raw_pointer_cast(&D_output[0]);
```

Ejemplo 1: Transformada 1d

simple_cufft.cu

```
/* Parametros de la senial */
#define A1 4
#define A2 6
#define T1 N/4
#define T2 N/8
struct FillSignal
{
	__device__ __host__
	REAL operator()(unsigned tid)
	{
		// empiece con esta funcion, luego ponga cualquiera que quiera transformar...
		return A1*2.0*cosf(2*M_PI*tid*T1/(float)N) + A2*2.0*sinf(2*M_PI*tid*T2/(float)N);
	}
};
```

Llenado de la señal en el device

```
// thrust::device_vector<REAL> D_input(N);

thrust::transform(thrust::make_counting_iterator(0), thrust::make_counting_iterator(N),
D_input.begin(),FillSignal());
```

Ejemplo 1: Transformada 1d

simple_cufft.cu

Creación de planes

```
// crea el plan de transformada de cuFFT

#ifdef DOUBLE_PRECISION

cufftHandle plan_d2z;
CUFFT_SAFE_CALL(cufftPlan1d(&plan_d2z,N,CUFFT_D2Z,1));

#else

cufftHandle plan_r2c;
CUFFT_SAFE_CALL(cufftPlan1d(&plan_r2c,N,CUFFT_R2C,1));

#endif
```

Ejemplo 1: Transformada 1d

simple_cufft.cu

Creación de planes

```
// crea el plan de transformada de cuFFT

#ifdef DOUBLE_PRECISION

cufftHandle plan_d2z;
CUFFT_SAFE_CALL(cufftPlan1d(&plan_d2z,N,CUFFT_D2Z,1));

#else

cufftHandle plan_r2c;
CUFFT_SAFE_CALL(cufftPlan1d(&plan_r2c,N,CUFFT_R2C,1));

#endif
```

Ejemplo 1: Transformada 1d

simple_cufft.cu

Ejecución de la transformada

```
/* ---- Start ---- */
        // un timer para GPU
        gpu_timer tfft;
        tfft.tic();

        //Transforma Fourier ejecutando el plan
        #ifdef DOUBLE_PRECISION
        CUFFT_SAFE_CALL(cufftExecD2Z(plan_d2z, d_input, d_output));
        #else
        CUFFT_SAFE_CALL(cufftExecR2C(plan_r2c, d_input, d_output));
        #endif
        tfft.tac();
        /* ---- Stop ---- */
```

Ejemplo 1: Transformada 1d

simple_cufft.cu

Copia al host de la transformada, parte real e imaginaria

```
	// declara un vector para copiar/guardar la transformada en el host:
	thrust::host_vector<COMPLEX> H_output=D_output;

#ifdef IMPRIMIR
	ofstream transformada_out("transformada.dat");
	for(int j = 0 ; j < Ncomp ; j++){
		transformada_out << COMPLEX(H_output[j]).x << " " << COMPLEX(H_output[j]).y << endl;
	}
#endif
```

Ejemplo 1: Transformada 1d

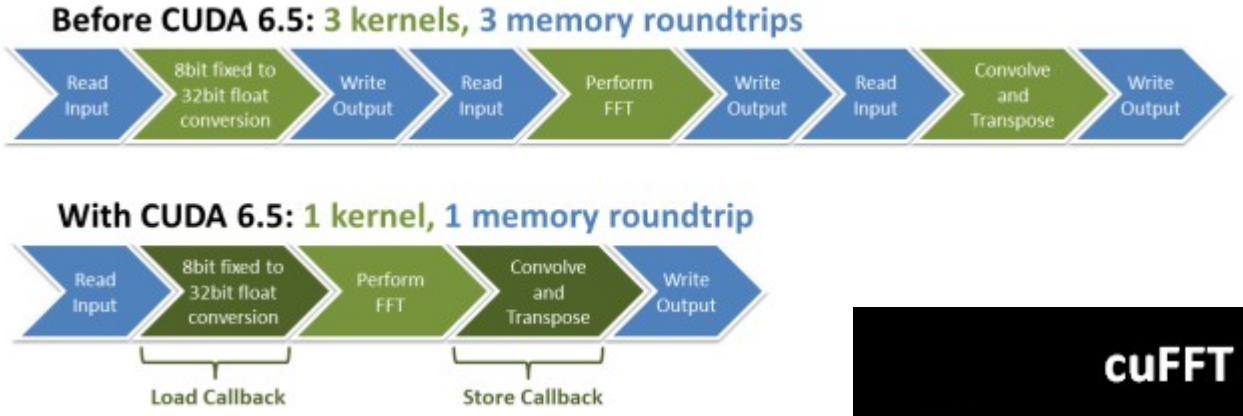
- /share/apps/icnpg/clases/clase11/Simple_cufft_fftw_thrust
- **simple_cufft.cu**: transformada en CUDA
- Compilar y Correr simple_cufft:
  - **¿Es correcta la transformada?**
- Completar template de **simple_cufft.cu:**
  - **Implemente la transformada inversa de la transformada obtenida.**
- **simple_fftw_thrust_solucion.cpp**: transformadas serial y openmp
- Comparar performances.

Guía del usuario

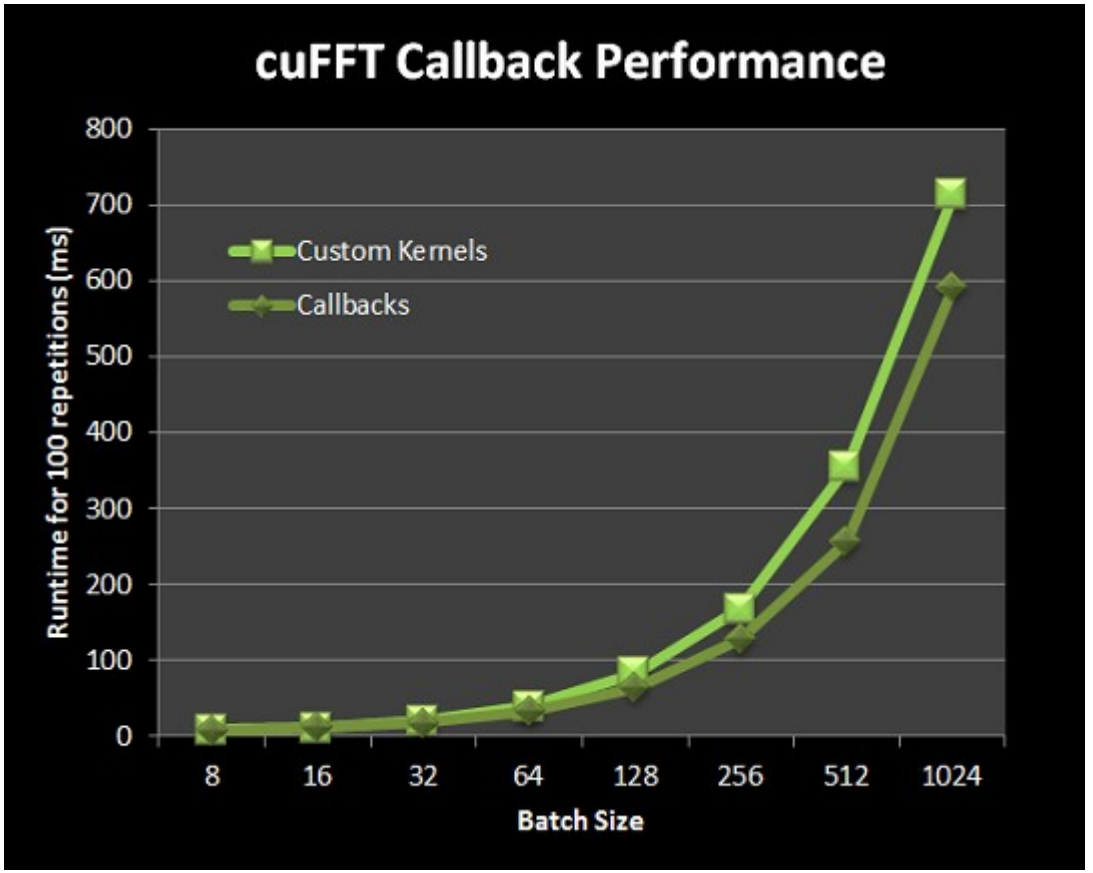
Mas adelante

Nuevo: útil para optimizar
- kernel fusion -

Callbacks



- https://devblogs.nvidia.com/cuda-pro-tip-use-cufft-callbacks-custom-data-processing/



Guía del usuario

Chapter 7. FFTW INTERFACE TO CUFFT

NVIDIA provides FFTW3 interfaces to the cuFFT library. This allows applications using FFTW to use NVIDIA GPUs with minimal modifications to program source code. To use the interface first do the following two steps

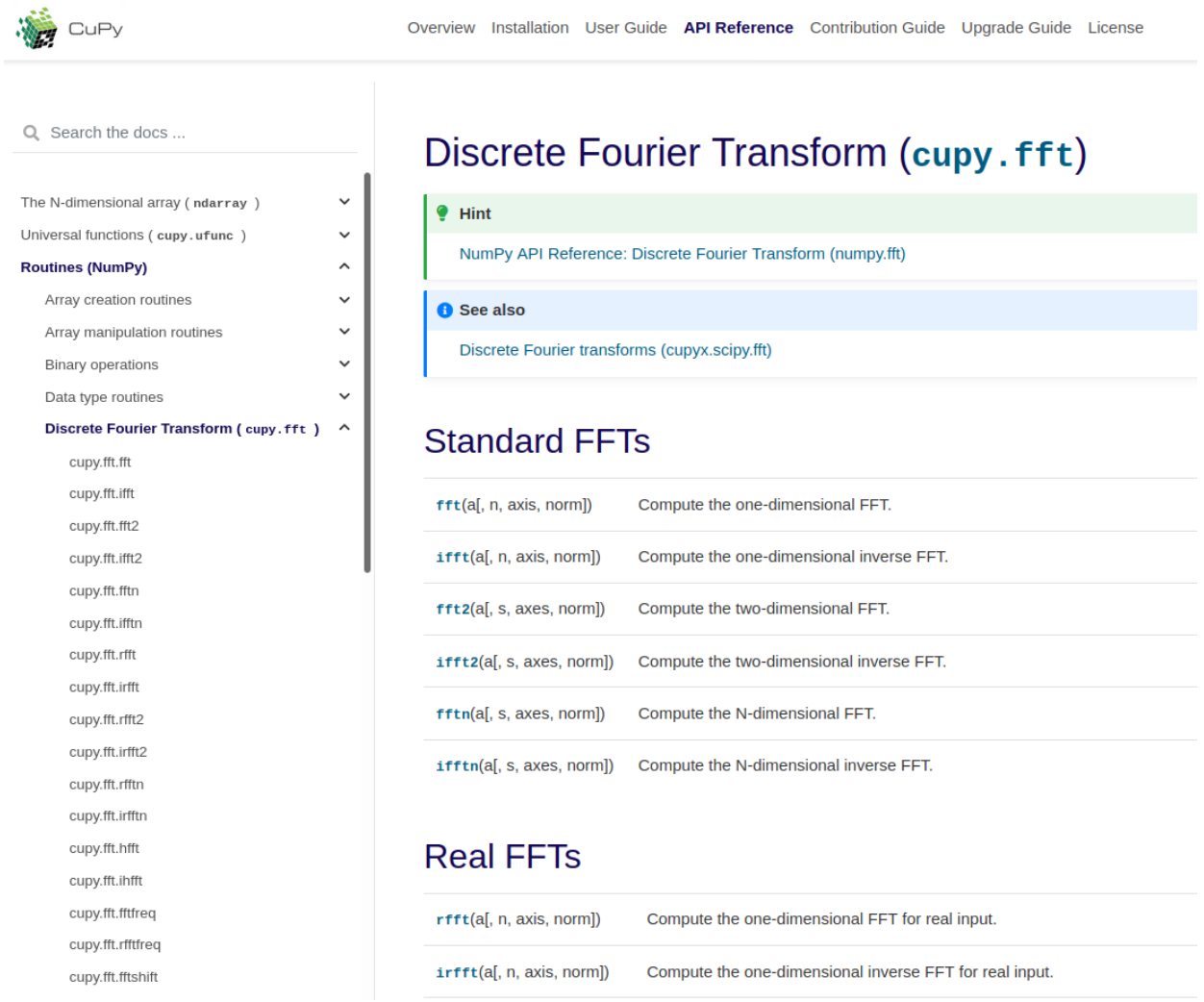
- It is recommended that you replace the include file **`fftw3.h`** with **`cufftw.h`**
- Instead of linking with the double/single precision libraries such as **`fftw3/fftw3f`** libraries, link with both the cuFFT and cuFFTW libraries
- Ensure the search path includes the directory containing **`cuda_runtime_api.h`**

After an application is working using the FFTW3 interface, users may want to modify their code to move data to and from the GPU and use the routines documented in the FFTW Conversion Guide for the best performance.

Referencias

- CUFFT Library User Guide NVIDIA.
- CUDA Libraries SDK Code Samples (simpleCUFFT, PDEs, etc).
- FFTW home page: http://www.fftw.org/ [interface parecida para CPU + openmp, mpi].
- Numerical Recipes in C [algoritmo FFT explicado].
- Libros de Numerical PDEs.... [como discretizar las PDEs?]

Ejemplo 2: Transformada 1d en cupy



https://docs.cupy.dev/en/stable/reference/fft.html

Ejemplo 3: convolución 2d en CUDA C++

- /share/apps/icnpg/clases/Clases_cufft/Convolucion2d_cufft
- Convoluciona o filtra un arreglo bidimensional usando cufft:
  - Transformo filtro (analiticamente)
  - Transformo imagen (cufft)
  - Multiplico las transformadas complejas
  - Antitransformo el producto y normalizo debidamente…
- ¿Funciona?
- ¿Para qué sirve?

Ejemplo 4: operadores diferenciales 2d

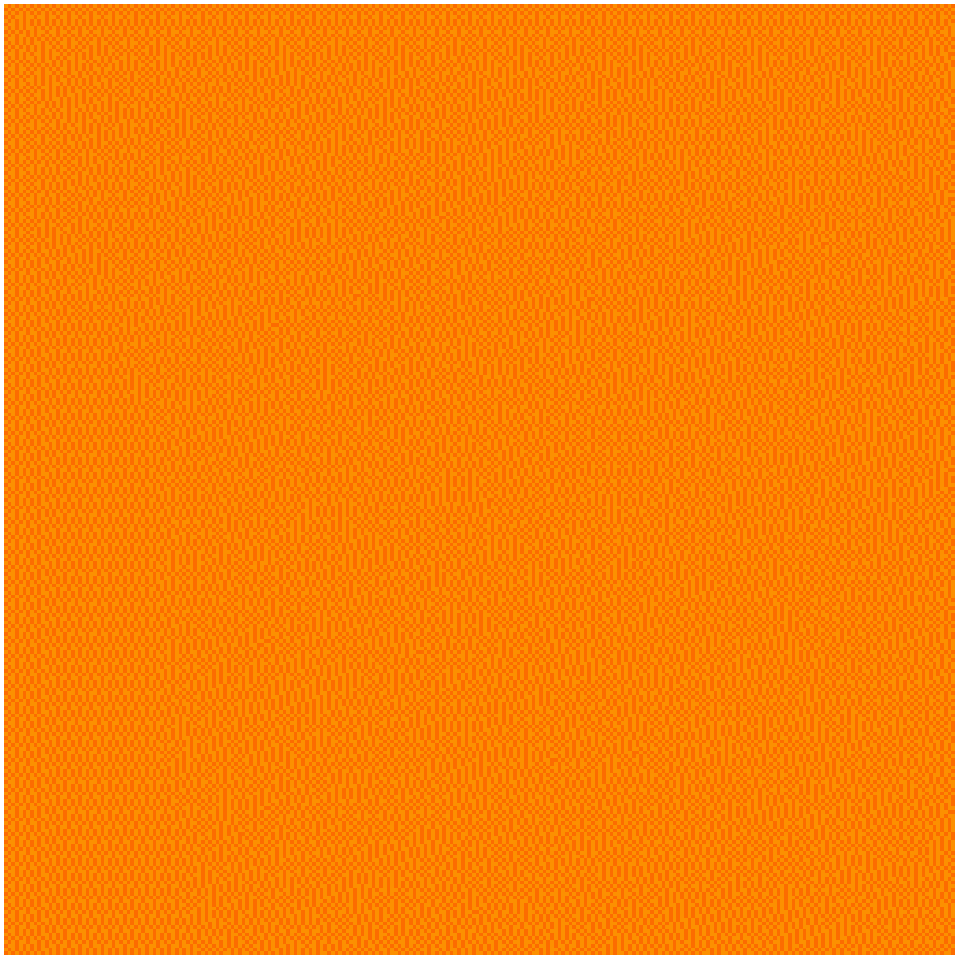
Ecuación “del calor” en dos dimensiones con fuentes arbitrarias en el espacio de Fourier

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

Ecuación de difusión con fuentes arbitrarias

Una configuración de fuentes divertida

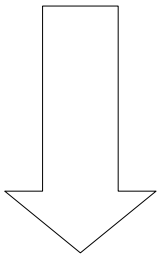
“Placa Metálica” [Condiciones de Contorno Periódicas]



En las diapositivas esto era una animación

Ecuación “del calor” en dos dimensiones con fuentes arbitrarias en el espacio de Fourier

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



Transformando Fourier

Modos desacoplados

$$\frac{\partial}{\partial t}\tilde{u}(q_x,q_y,t) = -c\left(q_x^2+q_y^2\right)\tilde{u}(q_x,q_y,t) + \tilde{\rho}(q_x,q_y,t)$$

$$\tilde{u}(q_x,q_y,t=0) = 0 \rightarrow$$

Solución analítica!

$$\tilde{u}(q_x,q_y,t) = \int_0^t dt'\; e^{-c(q_x^2+q_y^2)(t-t')}\tilde{\rho}(q_x,q_y,t')$$

Ecuación de difusión con fuentes arbitrarias

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

Diferencias finitas espaciales

$$\begin{cases} \delta_y^2 u_{i,l} = u_{j,l+1} - 2u_{j,l} + u_{j,l-1} \\ \delta_x^2 u_{i,l} = u_{j+1,l} - 2u_{j,l} + u_{j-1,l} \end{cases}$$

Esquema explícito (sin fuente)

$$u_{j,l}(t+\Delta t) = u_{j,l}(t) + \frac{c\Delta t}{\Delta^2}(\delta_x^2 u_{j,l}(t) + \delta_y^2 u_{j,l}(t)), \quad \frac{c\Delta t}{\Delta^2} \leq \frac{1}{2}$$

Esquema implícito [Crank-Nicholson]

$$u_{j,l}(t+\Delta t) = u_{j,l}(t) +$$
$$\frac{c\Delta t}{2\Delta^2}(\delta_x^2 u_{j,l}(t) + \delta_x^2 u_{j,l}(t+\Delta t) + \delta_y^2 u_{j,l}(t) + \delta_y^2 u_{j,l}(t+\Delta t))$$

Ecuación de difusión con fuentes arbitrarias

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

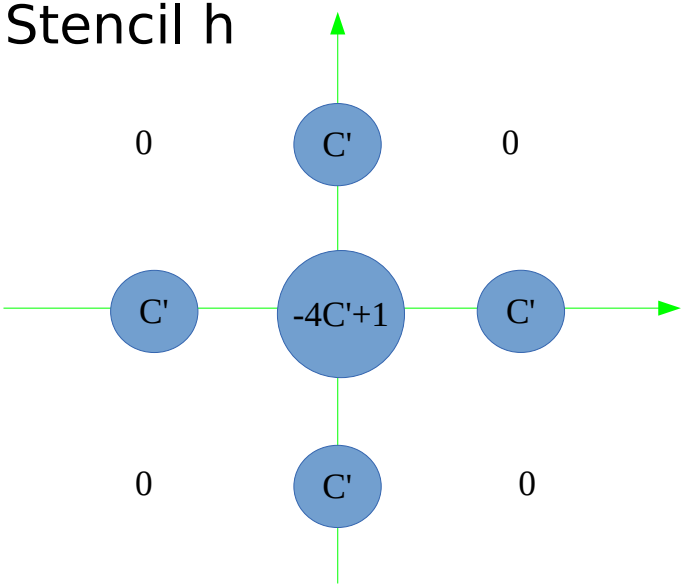
Diferencias finitas

$$u_{i,j}(t+\Delta t) \approx u_{i,j}(t) +$$

$$\frac{c\Delta t}{\Delta^2}\left(u_{i+1,j}(t) + u_{i-1,j}(t) + u_{i,j+1}(t) + u_{i,j-1}(t) - 4u_{i,j}(t)\right) + \Delta t\rho_{i,j}(t)$$

$$u_{i,j}(t+\Delta t) \approx \underbrace{\sum_{a,b} u_{i+a,j+b}h_{a,b}}_{\text{Convolucion en 2D}} + \delta t\rho_{i,j}(t)$$

Stencil h



Se puede resolver en el espacio real (ej: con CUSP),
Pero lo haremos en espacio de Fourier (cuFFT)

$$g * h \Longleftrightarrow G(f)H(f)$$

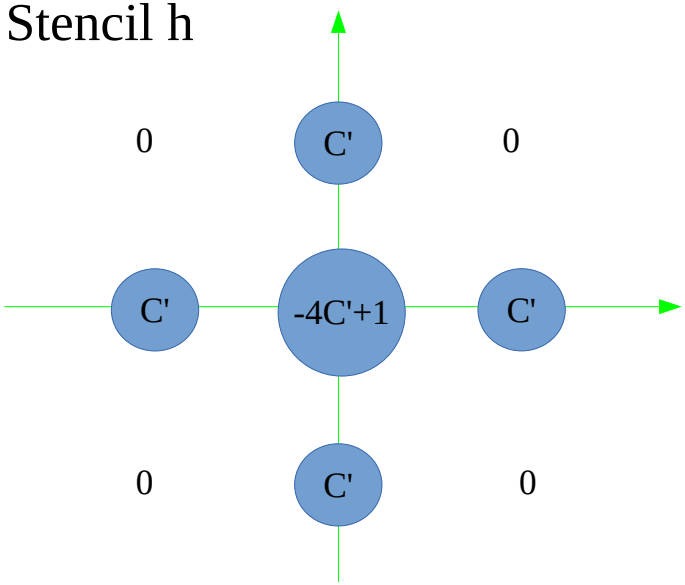
Ecuación de difusión con fuentes arbitrarias

$$u_{i,j}(t+\Delta t) \approx \sum_{a,b} u_{i+a,j+b} h_{a,b} + \delta t \rho_{i,j}(t)$$

$$g * h \Longleftrightarrow G(f)H(f)$$

Todo desacoplado :-)

Stencil h



$$\tilde{u}_{n,m}(t+\Delta t) \approx \tilde{u}_{n,m}(t)\tilde{h}_{n,m} + \delta t \tilde{\rho}_{n,m}(t)$$

$$\tilde{h}_{n,m} = \sum_{a,b} e^{2\pi i(na/L_x + mb/L_y)} h_{a,b}$$

$$\tilde{h}_{n,m} = (-4C' + 1) + 2C'[\cos(2\pi(n/L_x)) + \cos(2\pi(m/L_y))]$$

Aplicar el stencil o hacer la convolución 2D en el espacio real
se reduce a multiplicar por un campo escalar constante la transformada

Ecuación de difusión con fuentes arbitrarias

Esto se calcula una sola vez al principio

$$\tilde{h}_{n,m} = (-4C' + 1) + 2C'[\cos(2\pi(n/L_x)) + \cos(2\pi(m/L_y))]$$

Loop temporal

```
for(t=0;t<TRUN;t++)
{
```

$$\tilde{u}_{n,m}(t + \Delta t) \approx \tilde{u}_{n,m}(t)\tilde{h}_{n,m} + \delta t \tilde{\rho}_{n,m}(t), \quad \forall\, n, m$$

```
}
```

Euler Step
Trivialmente
Paralelizable

Ecuación de difusión con fuentes arbitrarias

$$\tilde{u}_{n,m}(t+\Delta t) \approx \tilde{u}_{n,m}(t)\tilde{h}_{n,m} + \delta t \tilde{\rho}_{n,m}(t), \quad \forall\, n, m$$

Euler Step
Trivialmente
Paralelizable

```
// Paso de Euler en el espacio de Fourier
__global__ void euler_step
(cufftComplex *a, const cufftComplex *b,
const float *qqmodule, int nx, int ny)
{
// idx and idy, the location of the element in the original NxN array
   int idx = blockIdx.x*blockDim.x+threadIdx.x;
   int idy = blockIdx.y*blockDim.y+threadIdx.y;
   if ( idx < nx && idy < ny){
          int index = idx*ny + idy;
          const float fac=-(qqmodule[index]);
          a[index].x = (1.f+C_BETA*fac*DT)*a[index].x + DT*b[index].x;
          a[index].y = (1.f+C_BETA*fac*DT)*a[index].y + DT*b[index].y;
   }
}
```

Conviene trabajar
Con grillas 2D!

$\tilde{u}_{n,m}(t+\Delta t)$ $\tilde{h}_{n,m}$ $\tilde{u}_{n,m}(t)$ $\tilde{\rho}_{n,m}(t)$

Ecuación de difusión con fuentes arbitrarias

for(t=0;t<TRUN;t++)
{

$\tilde{u}_{n,m}(t) \rightarrow u_{i,j}(t)$

Para dibujar (cada tanto) el campo en espacio real → **CUFFT (inv)** → memcpy

...

$\rho_{i,j}(t) \rightarrow \tilde{\rho}_{n,m}(t)$

Las fuentes son dadas instantaneamente en el espacio real, pero las necesitamos en Fourier → **CUFFT**

...

$$\tilde{u}_{n,m}(t+\Delta t) \approx \tilde{u}_{n,m}(t)\tilde{h}_{n,m} + \delta t \tilde{\rho}_{n,m}(t), \quad \forall\, n,m$$

Kernel Paso de Euler

}

Ecuación de difusión con fuentes arbitrarias

```
for(t=0;t<TRUN;t++)
{
```

$$\tilde{u}_{n,m}(t) \rightarrow u_{i,j}(t)$$

Para dibujar (cada tanto) el campo en espacio real → **CUFFT** → memcpy

...

$$\rho_{i,j}(t) \rightarrow \tilde{\rho}_{n,m}(t)$$

Las fuentes son dadas instantaneamente en el espacio real, pero las necesitamos en Fourier → **CUFFT**

...

$$\tilde{u}_{n,m}(t+\Delta t) \approx \tilde{u}_{n,m}(t)\tilde{h}_{n,m} + \delta t \tilde{\rho}_{n,m}(t), \quad \forall\, n, m$$

```
}
```

Kernel Paso de Euler

Hands-on:
Ecuación de difusión con fuentes arbitrarias

Loop temporal principal

main.cu

```
gpu_timer cronometro;
cronometro.tic();
for(int i=0;i<T_RUN+1; i++){

        /* Get data each T_DATA steps*/
        if(i%T_DATA==0) {
                /* Take the scalar field back from Fourier space and normalize */
                T.AntitransformFromFourierSpace();
                T.Normalize();

                T.CpyDeviceToHost(); // To be used in a moment for printing.

                /* Print frame for visualization */
                sprintf(filename, "frame%d.ppm", 100000000+i);
                ifstream file(filename);
                T.PrintPicture(filename,i);
        }
        {

                float phase=2.f*M_PI*i*4.f/T_RUN;
                float x0=0.2*sin(phase);float y0=0.2*cos(phase); float intensidad=0.1;
                T.InitParticular(1, LX,LY,4,x0,y0,0.1,intensidad); // perfil de fuentes instantaneo
                T.TransformForceToFourierSpace(); // transforma perfil de fuentes para Euler Step
        }

        /* Integration step in Fourier Space*/
        T.EulerStep();
}
cronometro.tac();
```

En este problema:
Solo para hacer imágenes

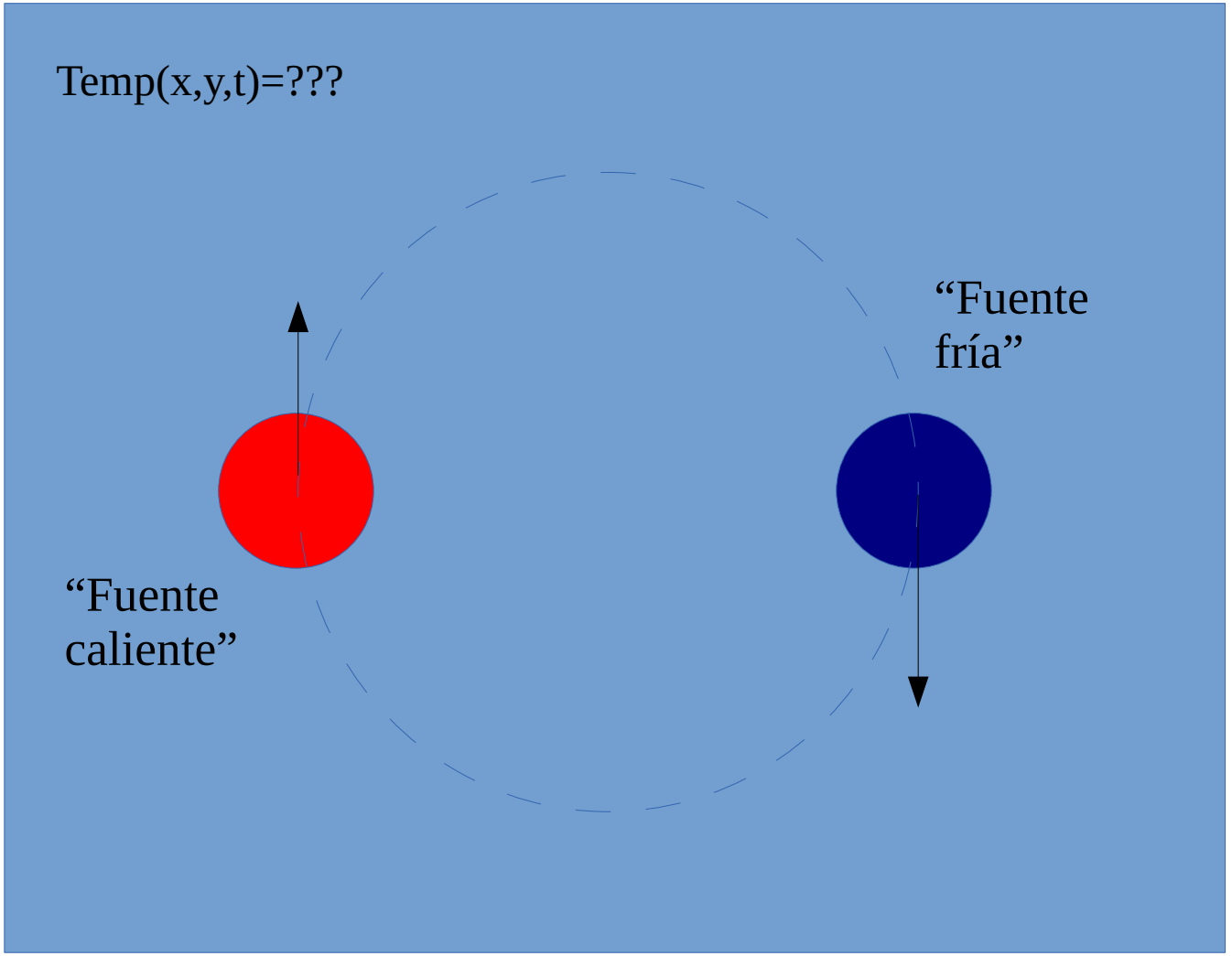
Controla fuentes
en espacio real...

Un paso de tiempo en el
Espacio de Fourier

Hands-on:
Ecuación de difusión con fuentes arbitrarias

Una configuración de fuentes divertida

"Placa Metálica" [Condiciones de Contorno Periódicas]



Hands-on!
Ecuación de difusión con fuentes arbitrarias

- /share/apps/icnpg/clases/Clases_cufft/Heat/
- make
- qsub jobGPU

- *Escribe snapshots.ppm → película.*

Referencias

- CUFFT Library User Guide NVIDIA.
- CUDA Libraries SDK Code Samples (simpleCUFFT, PDEs, etc).
- FFTW home page: http://www.fftw.org/ [interface parecida para CPU + openmp, mpi].
- Numerical Recipes in C [algoritmo FFT explicado].
- Libros de Numerical PDEs.... [como discretizar las PDEs?]

Un ejemplo concreto de aplicación científica

- **Formación de patrones de magnetización en films ferromagnéticos:**

  Matemáticamente: Ecuación diferencial parcial no local en función del tiempo para un campo escalar bidimensional.

Un modelo simple para la magnetización en films ferromagnéticos 2D

$$\phi(\mathbf{r},t) = \phi(x,y,t)$$

Magnetización coarse-grained (“soft spin”),
o parámetro de orden bidimensional

Landau approach: modelar la energía libre en función del parámetro de orden

$$H_l = \alpha_0 \int d\mathbf{r} \left(-\frac{\phi(\mathbf{r})^2}{2} + \frac{\phi(\mathbf{r})^4}{4} \right) - h_0 \int d\mathbf{r}\phi(\mathbf{r})$$

Dos mínimos locales, y un campo
Que propicia uno de los dos

$$H_{rig} = \beta_0 \int d\mathbf{r} \frac{|\nabla\phi(\mathbf{r})|^2}{2}$$

Un precio energético por “doblar”
La magnetización

$$H_{dip} = \gamma_0 \int d\mathbf{r}d\mathbf{r}'\phi(\mathbf{r})\phi(\mathbf{r}')G(\mathbf{r},\mathbf{r}')$$

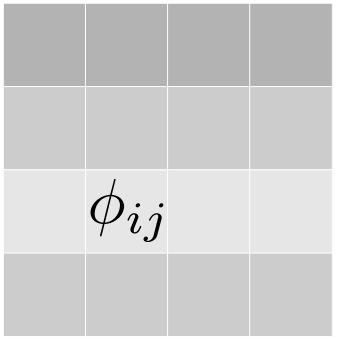
Interacciones dipolares de largo alcance
$G(\mathbf{r},\mathbf{r}') \sim 1/|\mathbf{r}-\mathbf{r}'|^3$

$$\frac{\partial\phi(\mathbf{r})}{\partial t} = -\lambda \frac{\delta(H_l + H_{rig} + H_{dip})}{\delta\phi(\mathbf{r})} = -\lambda \left(\alpha_0(-\phi+\phi^3) - h_0 - \beta_0\nabla^2\phi + \gamma_0 \int d\mathbf{r}' G(|\mathbf{r}-\mathbf{r}'|)\phi(\mathbf{r}') \right)$$

Numerical simulations of two-dimensional magnetic domain patterns E. A. Jagla, Phys. Rev. E 70, 046204 (2004)

Método Naive

$$\frac{\partial \phi(\mathbf{r})}{\partial t} = -\lambda \left(\alpha_0(-\phi + \phi^3) - h_0 - \beta_0 \nabla^2 \phi + \gamma_0 \int d\mathbf{r}' G(|\mathbf{r} - \mathbf{r}'|) \phi(\mathbf{r}') \right)$$



Discretización en espacio real

$$\phi_{ij}^{\text{new}} = \phi_{ij} - \lambda \delta t \left(\alpha(-\phi_{ij} + \phi_{ij}^3) - h_0 - \beta(\phi_{i+1\,j} + \phi_{i-1\,j} + \phi_{i,j+1} + \phi_{i\,j-1} - 4\phi_{ij}) + \gamma \sum_{kl} G_{ij,kl} \phi_{kl} \right)$$

Naive: O(N*N) ⟶ $\sim N$ operaciones

$\sim N^2$ operaciones

G es densa...

G es densa ... pero tiene tanta información?

G es densa ... pero tiene un montón de simetrías!

$$G_{ij,kl} = G_{kl,ij}, \; G_{(i+p)(j+q),(k+p)(l+q)} = G_{ij,kl}$$

Transformación de Fourier

$\phi(\mathbf{r}) = \phi(x, y)$

Magnetización coarse-grained (“soft spin”),
o parámetro de orden bidimensional

$$\frac{\partial \phi(\mathbf{r})}{\partial t} = -\lambda \left(\alpha_0(-\phi + \phi^3) - h_0 - \beta_0 \nabla^2 \phi + \gamma_0 \int d\mathbf{r}' G(|\mathbf{r} - \mathbf{r}'|)\phi(\mathbf{r}') \right)$$

- **Dificultad:** ya no solo a primeros vecinos, sino todos los puntos del espacio 2D están acoplados

Pero, en espacio de Fourier:

$$\frac{\partial \phi_{\mathbf{k}}}{\partial t} = -\lambda[\alpha_0(-\phi + \phi^3)|_{\mathbf{k}} - h_0 \delta(\mathbf{k}) + (\beta_0 k^2 + \gamma_0 G_{\mathbf{k}})\phi_{\mathbf{k}}]$$

Si no fuera por este termino, en el espacio de Fourier
quedarían Lx * Ly ecuaciones desacopladas!!.

- Mas simple ir al espacio de Fourier para la interacción de largo alcance y volver al espacio real para calcular termino cubico.
- También es eficiente si N=Lx Ly es grande: **N^2 vs 2 N log N**

Solución numérica: método pseudo-spectral

$$\phi = \phi(x, y)$$

Magnetización coarse-grained (“soft spin”), o parámetro de orden bidimensional

$$\frac{\partial \phi_{\mathbf{k}}}{\partial t} = -\lambda[\alpha_0(-\phi + \phi^3)|_{\mathbf{k}} - h_0\delta(\mathbf{k}) + (\beta_0 k^2 + \gamma_0 G_{\mathbf{k}})\phi_{\mathbf{k}}]$$

Semi-implicit Euler step:

estabilidad

$$\frac{\phi_{\mathbf{k}}^{t+\delta t} - \phi_{\mathbf{k}}^{t}}{\delta t} = [\alpha(\phi - \phi^3)|_{\mathbf{k}}^{t} + h_0\delta(\mathbf{k}) - \gamma G_{\mathbf{k}}\phi_{\mathbf{k}}^{t} - \beta k^2 \phi_{\mathbf{k}}^{t+\delta t}]$$

$$\Rightarrow \quad \phi_{\mathbf{k}}^{t+\delta t} = \frac{\phi_{\mathbf{k}}^{t} + \delta t[\alpha(\phi - \phi^3)|_{\mathbf{k}}^{t} + h_0\delta(\mathbf{k}) - \gamma G_{\mathbf{k}}\phi_{\mathbf{k}}^{t}]}{1 + \beta k^2 \delta t}$$

Algoritmo

Update:

$$\phi_{\mathbf{k}}^{t+\delta t} = \frac{\phi_{\mathbf{k}}^{t} + \delta t[\alpha(\phi - \phi^3)|_{\mathbf{k}}^{t} + h_0\delta(\mathbf{k}) - \gamma G_{\mathbf{k}}\phi_{\mathbf{k}}^{t}]}{1 + \beta k^2 \delta t}$$

(1) Calcular en espacio real:
(2) Transformadas:
(3) Hacer paso de Euler.
(4) Antitransformada:
(5) Volver a (1)

Kernel trivial

$[\phi^3]_{\mathbf{r}}^{t}$

Cufft 2d

$\phi^t \rightarrow \phi_{\mathbf{k}}^{t}$

$[\phi^3]^t \rightarrow [\phi^3]_{\mathbf{k}}^{t}$

$\phi_{\mathbf{k}}^{t} \rightarrow \phi_{\mathbf{k}}^{t+\delta t}$

Kernel trivial

Cufft 2d

$\phi_{\mathbf{k}}^{t+\delta t} \rightarrow \phi_{\mathbf{r}}^{t+\delta t}$

Implementación

Lineal pero No-local

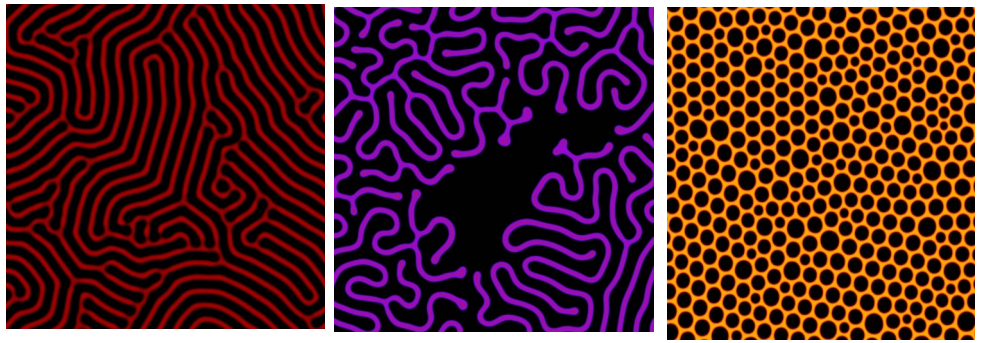
$$\frac{\partial \phi(\mathbf{r})}{\partial t} = -\lambda \left(\alpha(-\phi + \phi^3) - h_0 - \beta \nabla^2 \phi + \gamma \int d\mathbf{r}' G(|\mathbf{r} - \mathbf{r}'|) \phi(\mathbf{r}') \right)$$

Discretización + Transformada de Fourier

$$\phi_{\mathbf{k}}^{t+\delta t} = \frac{\phi_{\mathbf{k}}^{t} + \delta t [\alpha(\phi - \phi^3)|_{\mathbf{k}}^{t} + h_0 \delta(\mathbf{k}) - \gamma G_{\mathbf{k}} \phi_{\mathbf{k}}^{t}]}{1 + \beta k^2 \delta t}$$

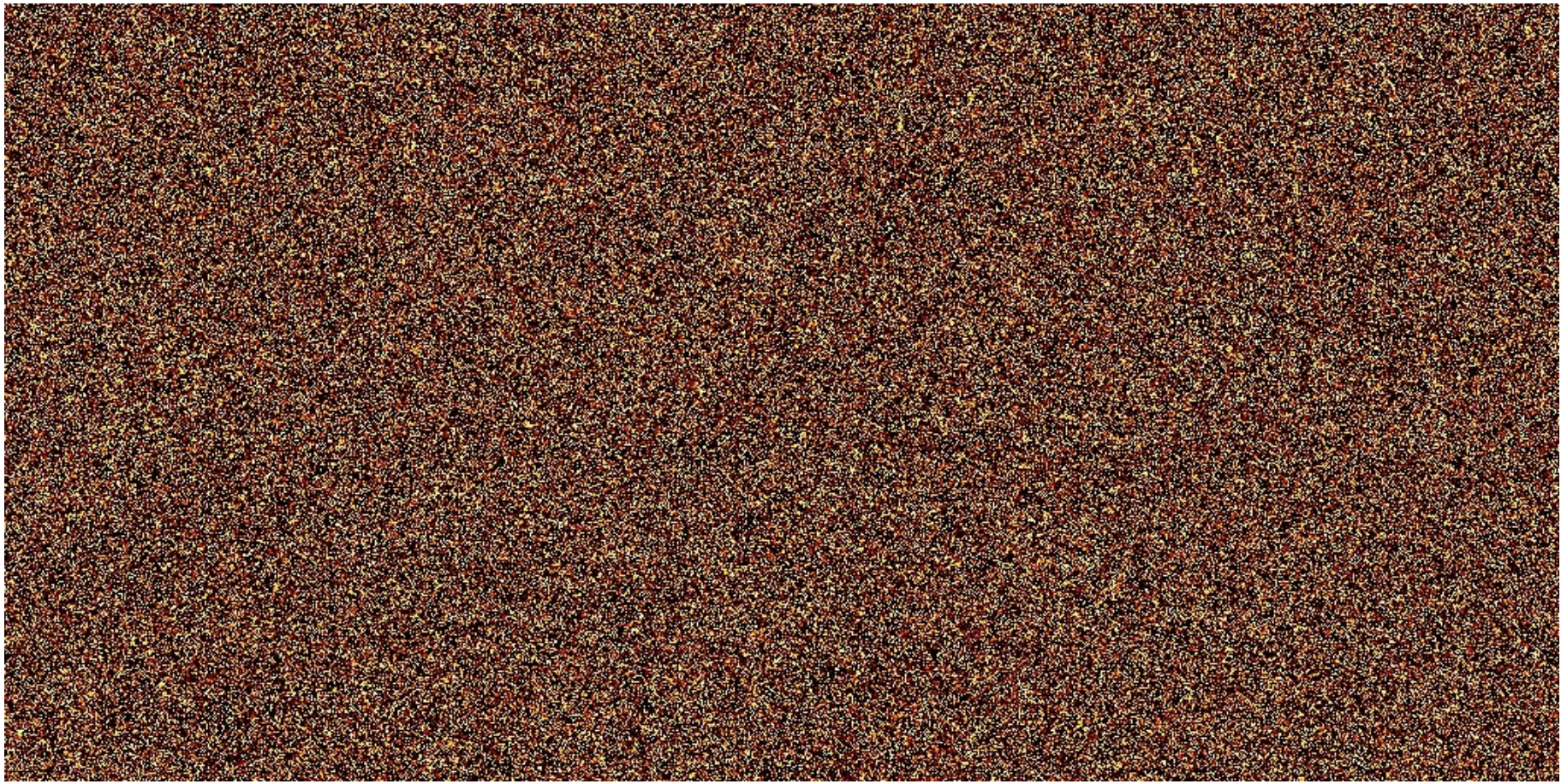
Local y lineal

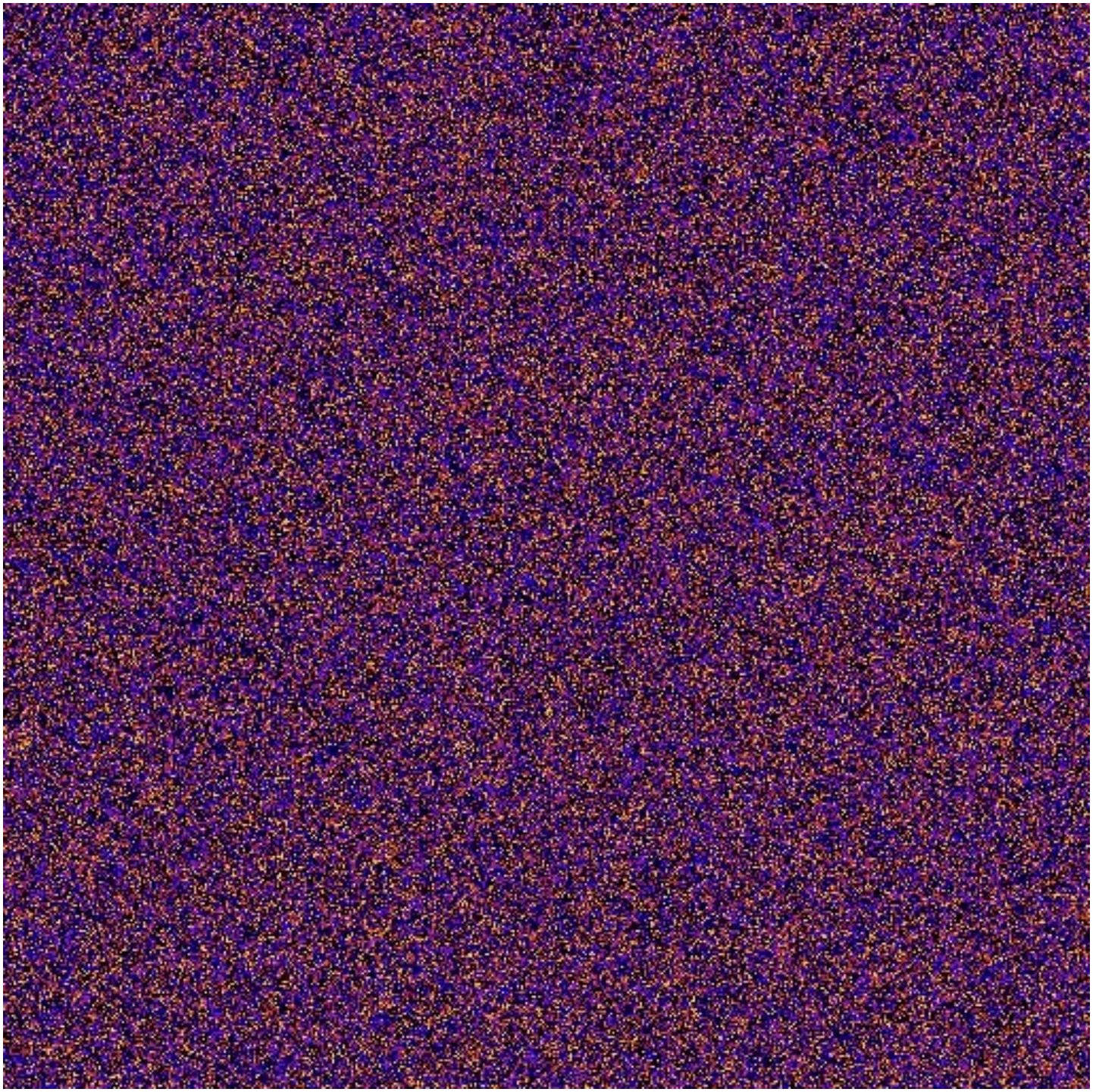
Requiere transformar/antitransformar matrices en cada paso de tiempo....

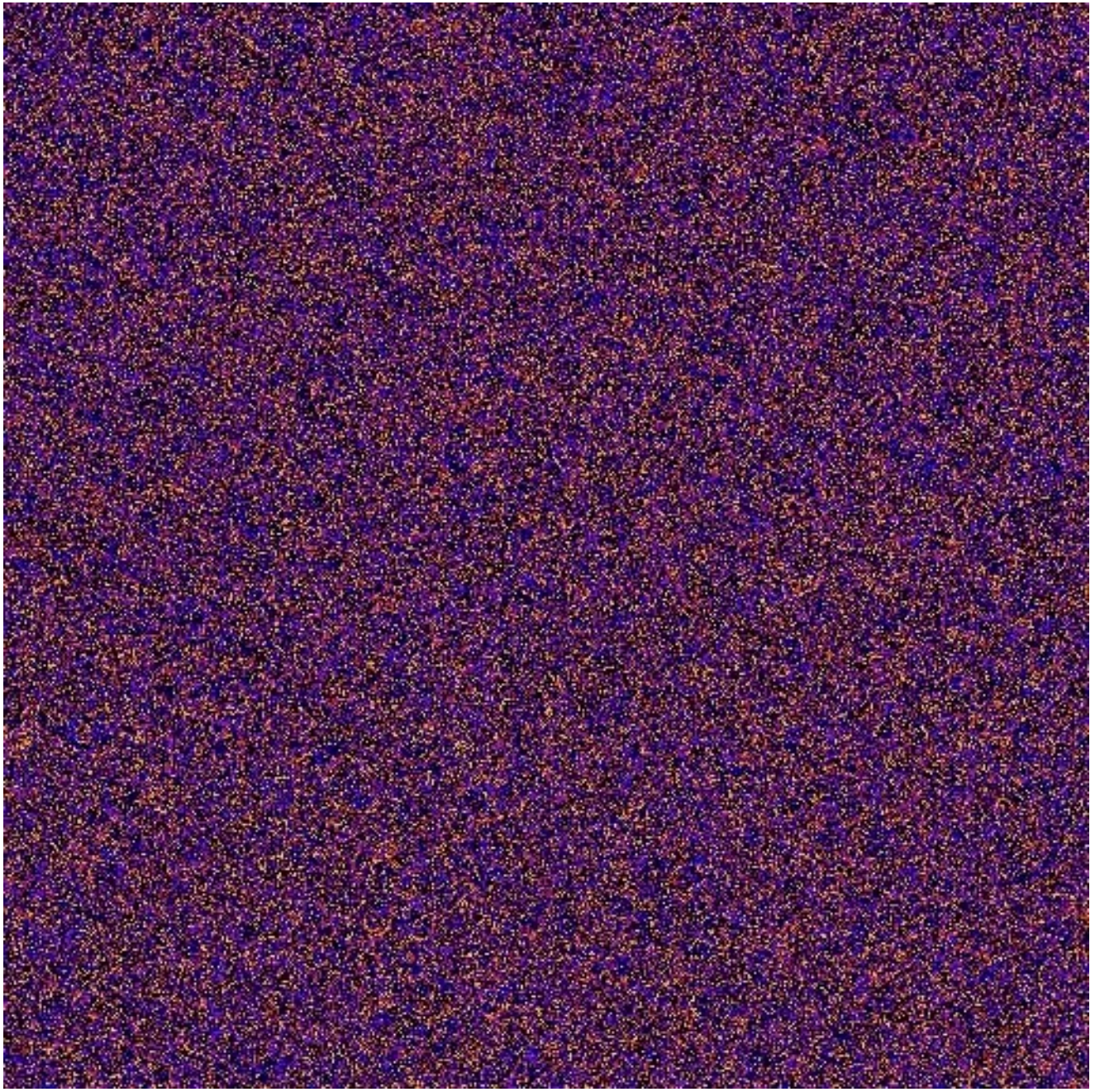


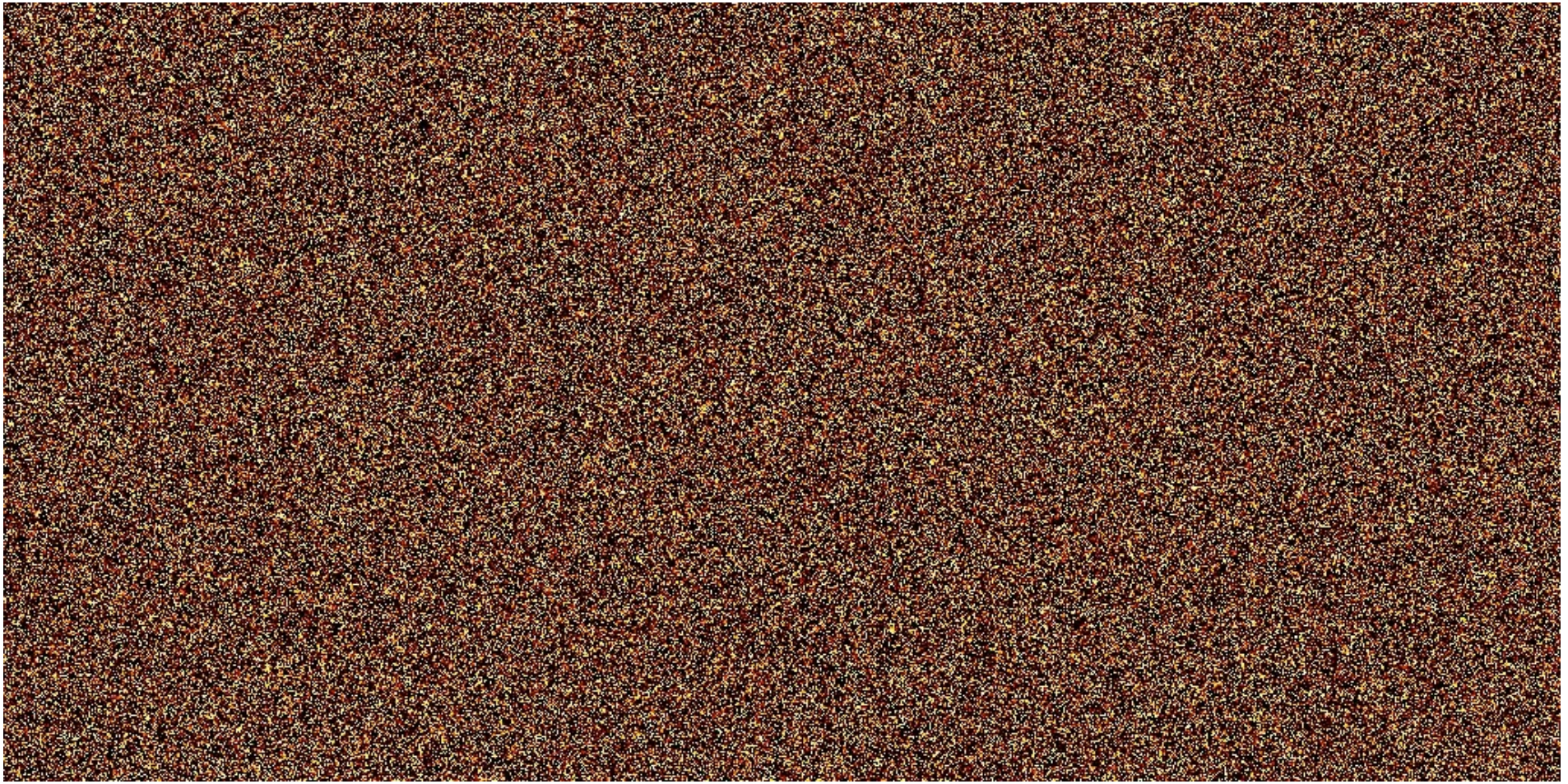
Naive:
O(N*N)

PseudoEspectral:
O(N log(N))



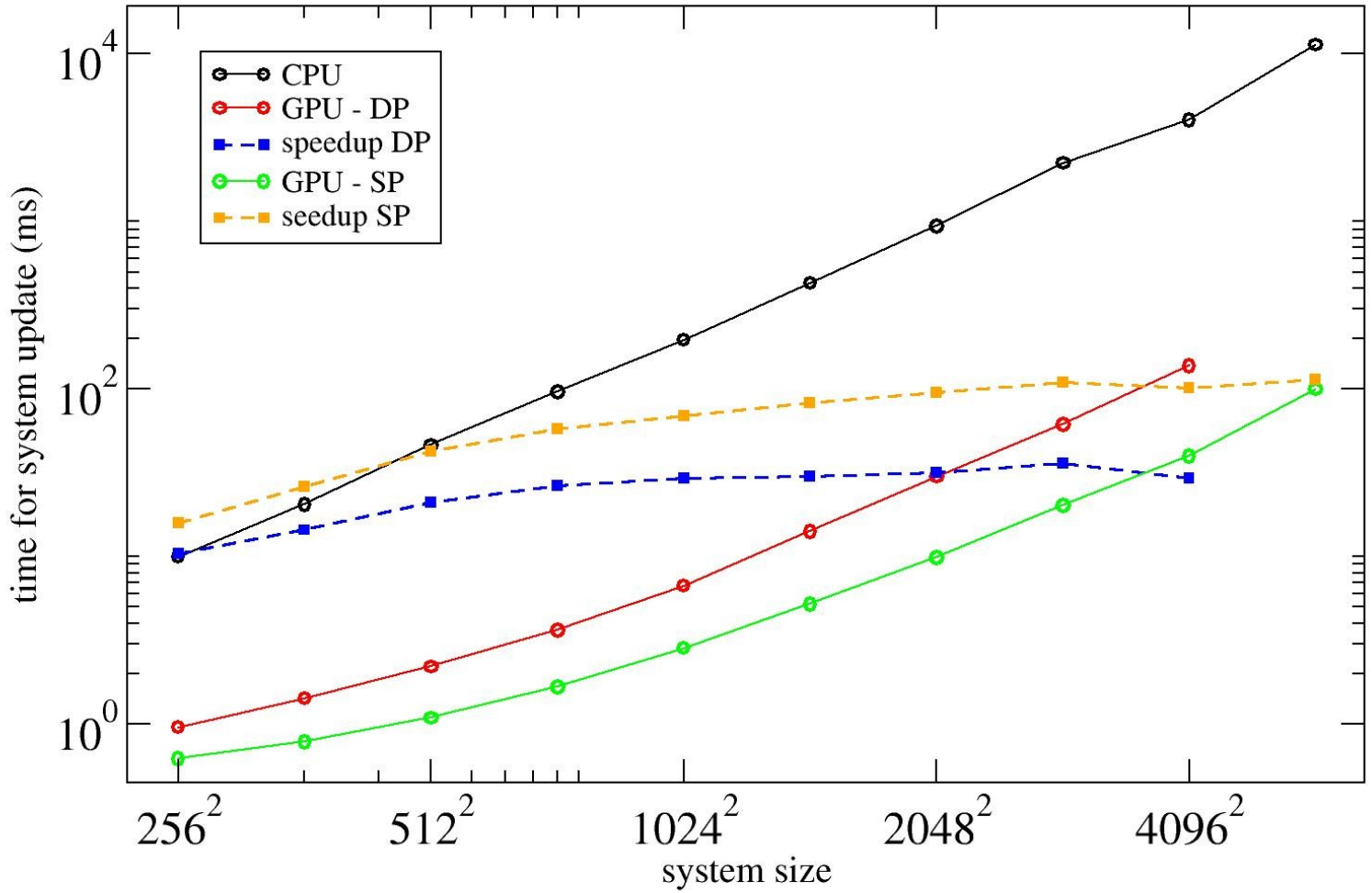






Performance

$$\phi_{\mathbf{k}}^{t+\delta t} = \frac{\phi_{\mathbf{k}}^{t} + \delta t[\alpha(\phi - \phi^3)|_{\mathbf{k}}^{t} + h_0\delta(\mathbf{k}) - \gamma G_{\mathbf{k}}\phi_{\mathbf{k}}^{t}]}{1 + \beta k^2 \delta t}$$



~50 x

GTX 470 vs 1 thread AMD Phenom II