

# Simulaciones de Física Estadística usando GPGPU: tres herramientas

Alejandro. B. Kolton  
CONICET, Centro Atómico Bariloche



WHPC13

# Gracias:

- **Ezequiel Ferrero** [Grenoble]
- S. Bustingorry [Centro Atómico Bariloche].
- GPGPU@CAB [Centro Atómico Bariloche].
- Curso ICNPG 2012-2013 [Instituto Balseiro]
- NVIDIA Academic Partnership
- SNCAD

\* Estamos organizando la 3<sup>er</sup> EAGPGPU en Bariloche (2014)



# Mensaje de un inexperto entusiasta en HPC ...

- **Principiantes**: cuando diga rápidamente en la charla que algo es fácil, *realmente* lo es !
- **Expertos**: pueden ayudar muchísimo mas conociendo un poquito de la problemática y perspectiva particular de cada científico (esfuerzo mutuo de comunicación).
- **Objetivo**: intentar convencerlos, con ejemplos simples concretos, de que es relativamente fácil empezar a desarrollar códigos de alto rendimiento para innovar la investigación científica, y de que definitivamente debemos fomentar mas la sinergia interdisciplinaria en el país.

# Física Estadística Computacional con GPGPU

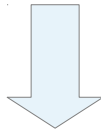
Variedad de Modelos dinámicos  
con desorden y fluctuaciones térmicas

Con aplicación a la materia condensada

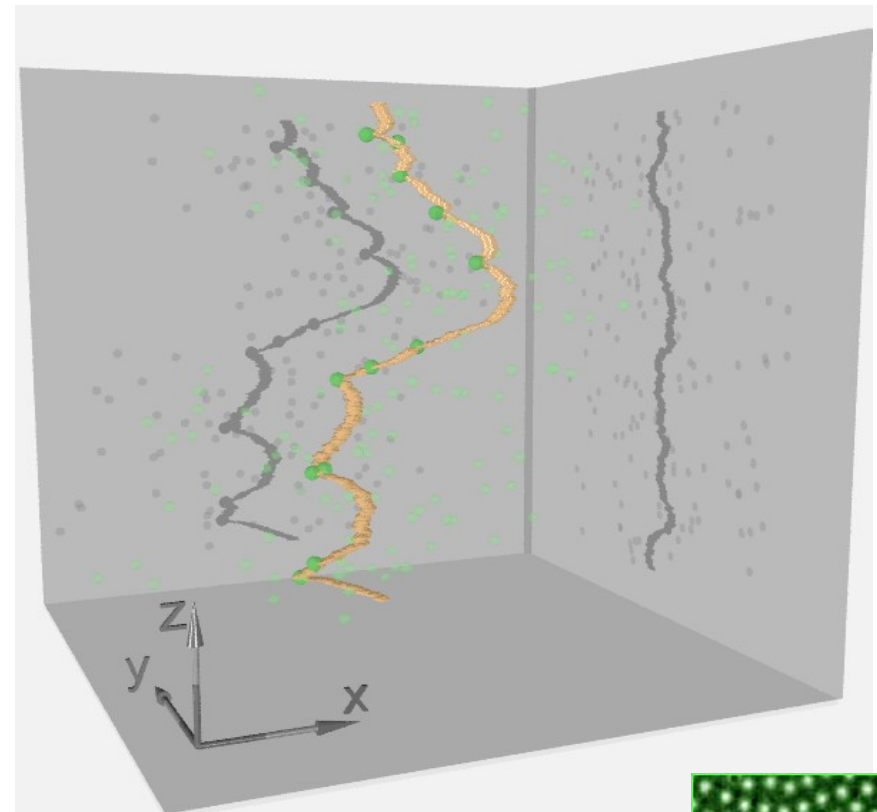
- Vórtices en superconductores.
- Paredes de dominio Magnéticas.
- Paredes de dominio ferroeléctricas.
- Líneas de contacto de mojado.
- Modelos de terremotos/avalanchas.
- “Hopping” de electrones localizados..
- Etc.

- **Dificultad:**

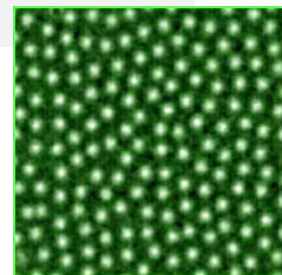
La física interesante “emerge” a gran escala: tamaño/número de partículas del sistema grande.



Necesidad de resolver numéricamente y de realizar cálculos de alto rendimiento en una variedad de modelos

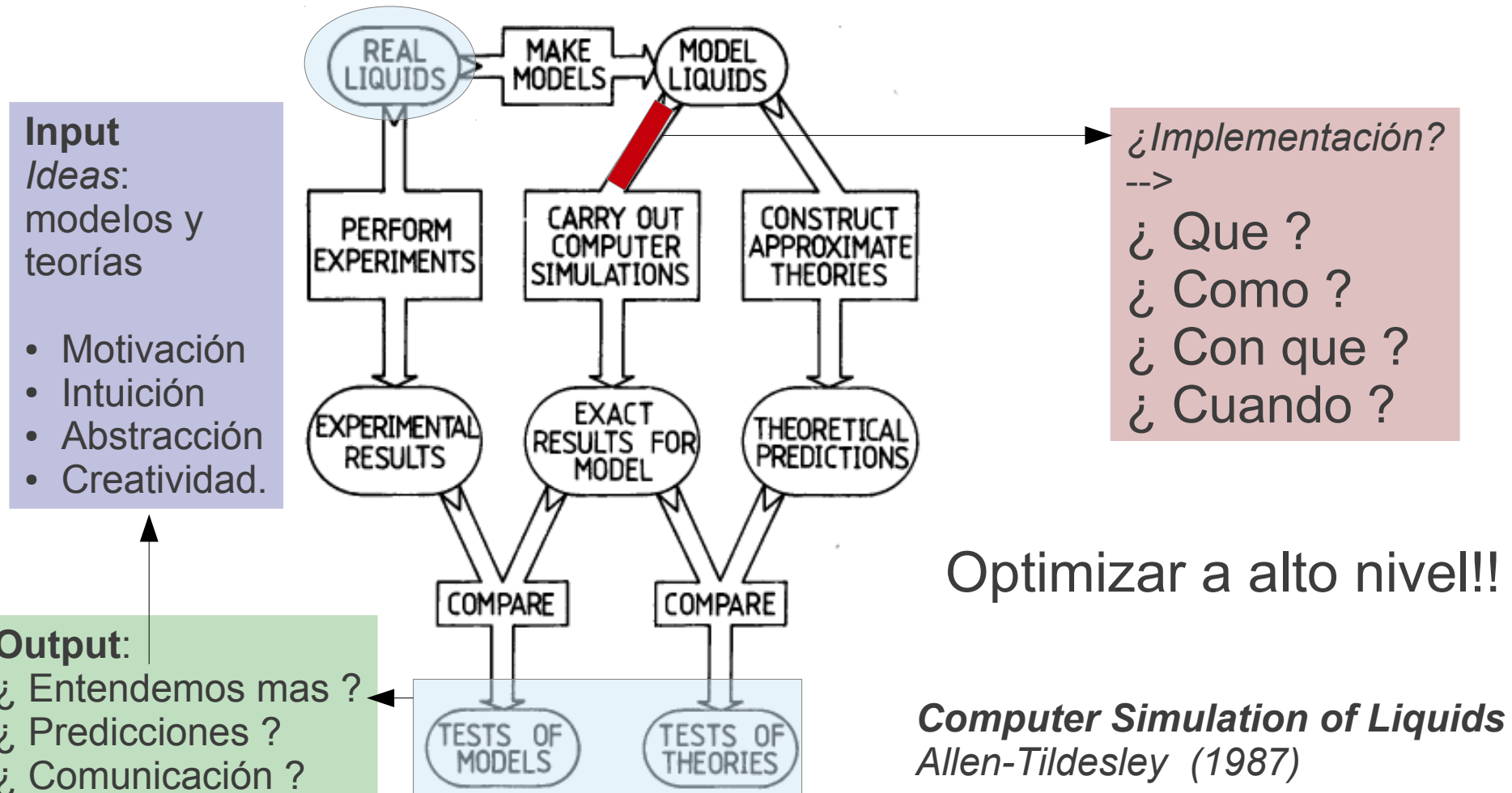


*Ej: modelar la dinámica de un vórtice en un superconductor con corriente*



# Una perspectiva

- Ciencia Básica: Preparar rápidamente un código de simulación numérica como si se tratara de un “setup experimental” para testear *ideas*...



# Una perspectiva

## Desafío:

**Escribir rápidamente un código de simulación:**

**Correcto, Robusto, y de Buen rendimiento:** Que permita obtener resultados (estadísticos) exactos y suficientemente precisos para testear las hipótesis...  
*pero no mas !*

**Alto nivel y en un lenguaje “expresivo”:** Pensar mas en “*que*” *quiero calcular*,  
*que en el “como” hacerlo...* y que sea fácil exponerlo e implementarlo.

- **GPGPU:**

- Buenos speed-ups para muchos problemas con poco esfuerzo!.
- Relativamente baratas para el poder de cálculo que brindan.
- Toda una variedad creciente de librerías activamente mantenidas, muchas de ellas gratis, útiles para aplicaciones científicas (expresivas y productivas).

*Pero antes de largarse a programar ...*

- **BUSCAR, DISCUTIR, PROBAR (95% del laburo):**  
Que herramientas de GPGPU se adecuan mejor a estas necesidades particulares ?



# Un tour sobre tres herramientas

## C++ template library



GPU y CPU multicore

## Random Numbers

**Random123:**

**a Library of Counter-Based  
Random Number Generators**

GPU y CPU multicore

## CUFFT



y su uso en dos ejemplos concretos  
de investigación científica

# Que es Thrust ?

<http://thrust.github.io/>

```
//#includes...
int main(void)
{
    // generate 32M random numbers serially
    thrust::host_vector<int> h_vec(32 << 20);
    std::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device (846M keys per second on GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```

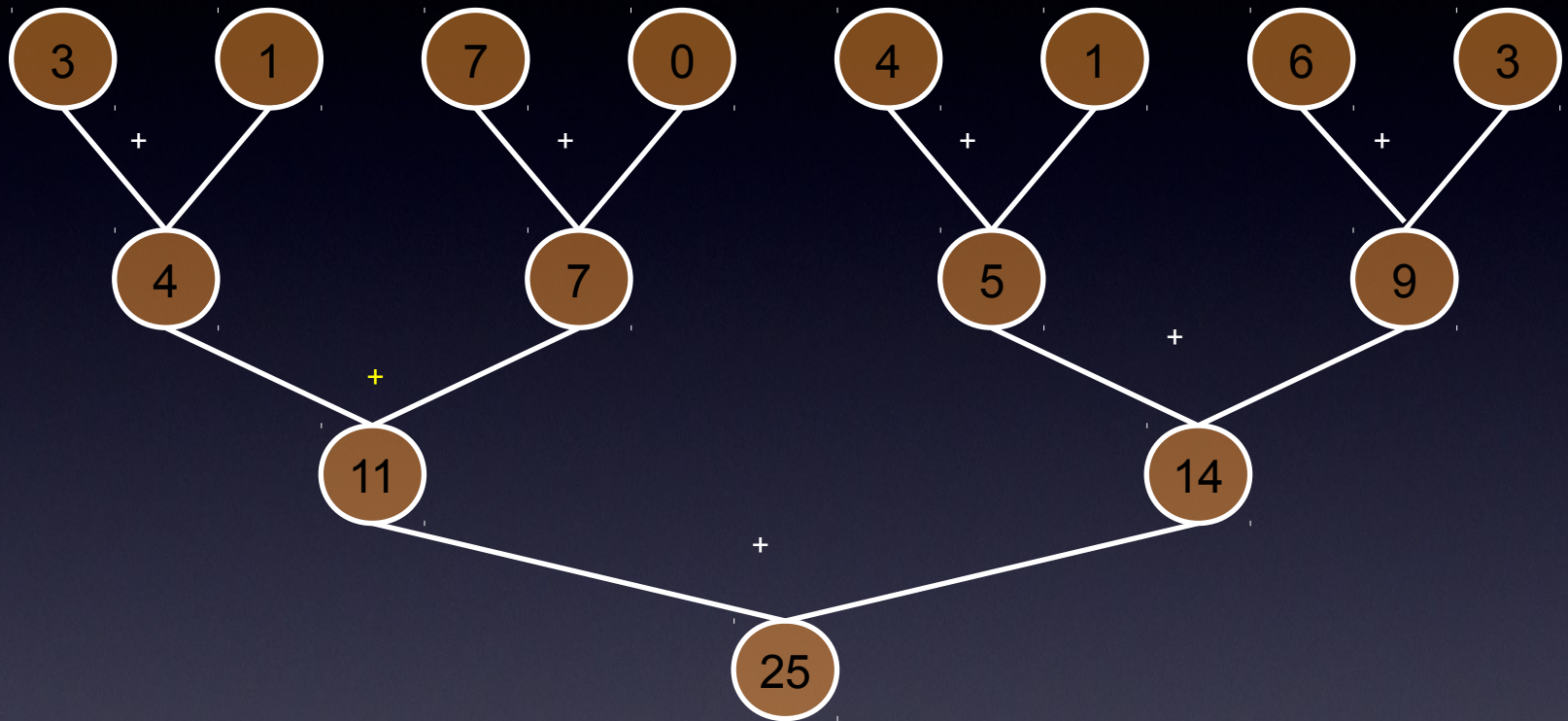
nvcc -O2 -arch=sm\_21 programa.cu -o ejecutable



# Que tipo de algoritmos tiene Thrust ?

Fundamentalmente  
*“patrones paralelos”*

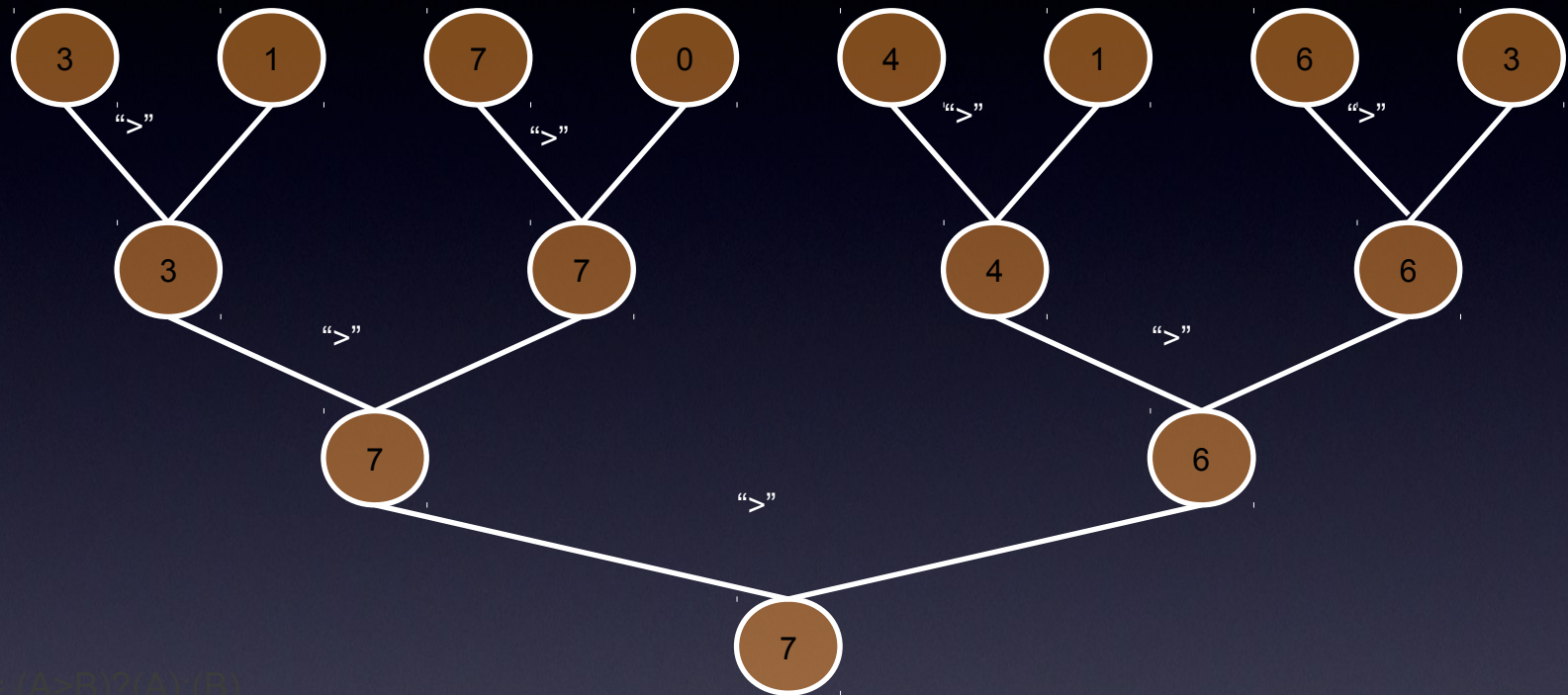
# Reducción en árbol



**Calcula la suma**

Vale para int, float, double, o cualquier tipo de dato con la operación "+" bien definida...

# Reducción

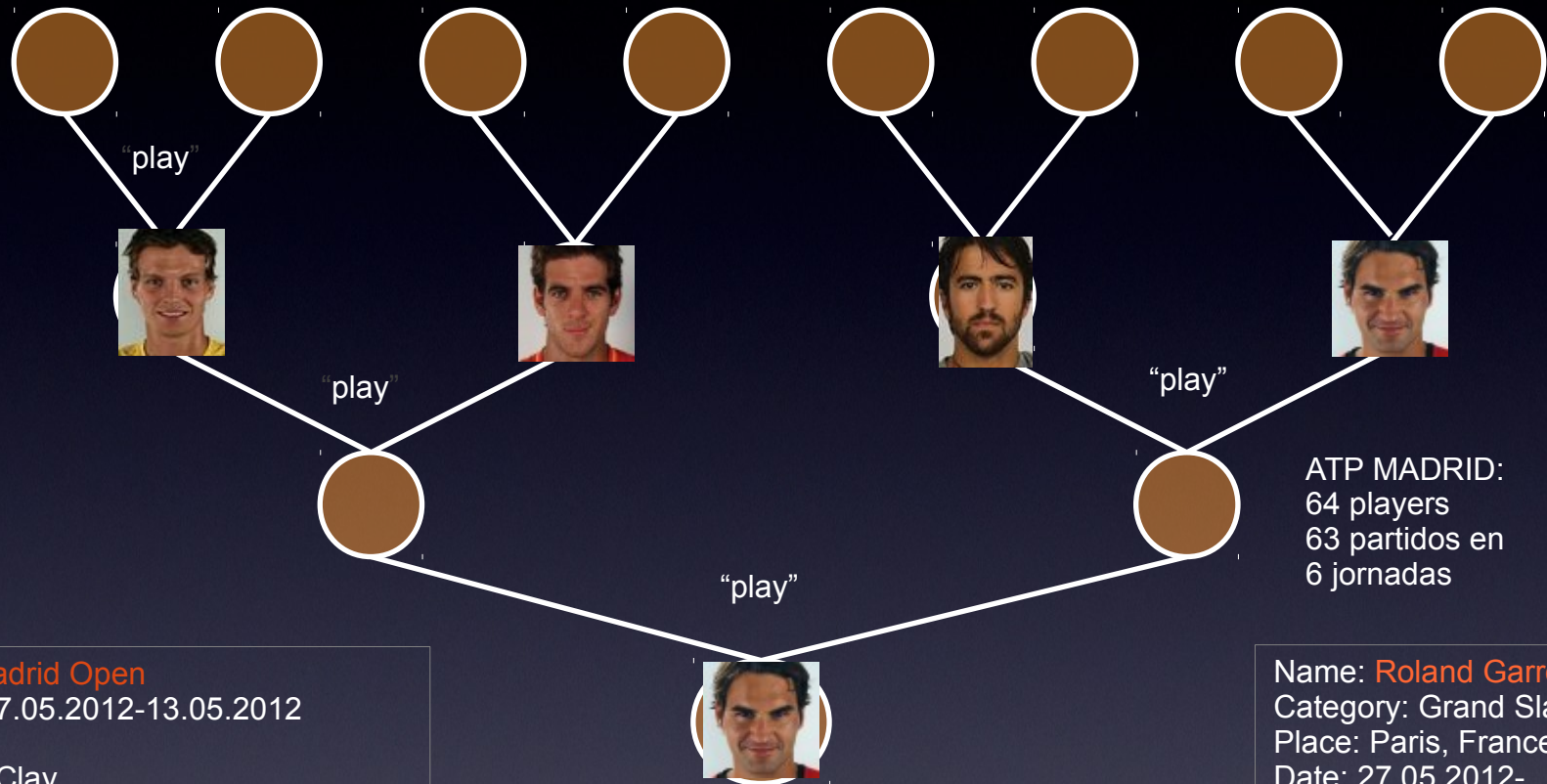


$A \gg B = (A > B) ? (A) : (B)$

**Calcula el *máximo***

Vale para int, float, double, o cualquier tipo de dato con la operación ">" bien definida...

# Reducción tenística



Mutua Madrid Open  
Spain - 07.05.2012-13.05.2012  
Draw: 64  
Surface: Clay  
Prize Money: €3,090,150  
63 partidos en 6 jornadas

ATP MADRID:  
64 players  
63 partidos en  
6 jornadas

Name: Roland Garros  
Category: Grand Slam  
Place: Paris, France  
Date: 27.05.2012-  
10.06.2012  
Draw Size: S-128 D-64

**Calcula el campeón**

log<sub>2</sub>(N) operaciones:

**Grand-Grand-Slam:**  $2^{20} = 1048576$  jugadores → solamente 20 "jornadas" !!!

# Reducción en Thrust

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include <cstdlib>

int main(void)
{
    // generate random data on the host
    thrust::host_vector<int> h_vec(100);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer to device and compute sum
    thrust::device_vector<int> d_vec = h_vec;
    int x = thrust::reduce(d_vec.begin(), d_vec.end(), 0, thrust::plus<int>());
    return 0;
}
```

Cualquier operacion binaria asoc. y conmutativa  
(+, -, \*, max, min, etc, o user-defined)

*Generic algorithms*

... definida sobre **cualquier**  
Estructura de datos  
(int, float, etc, o user-defined),  
Viva en el HOST o en el DEVICE (GPU)



# Thrust

Thrust Content from GTC 2012: Nathan Bell

## What is Thrust?

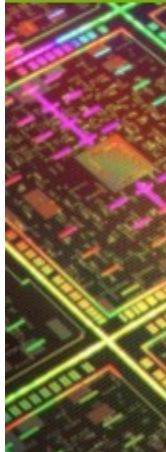
- High-Level Parallel Algorithms Library
- Parallel Analog of the C++ Standard Template Library (STL)
- Performance-Portable Abstraction Layer
- Productive way to program CUDA

Principales desarrolladores: [Jared Hoberock](#) y [Nathan Bell](#)

# Thrust

Thrust Content from GTC 2012: Nathan Bell

GPU  
TECHNOLOGY  
CONFERENCE



## Easy to Use

- Distributed with CUDA Toolkit
- Header-only library
- Architecture agnostic
- Just compile and run!

```
$ nvcc -O2 -arch=sm_20 program.cu -o program
```



# Thrust

Thrust Content from GTC 2012: Nathan Bell

## Productivity

- Containers

- `host_vector`
- `device_vector`

- Memory Management

- Allocation
- Transfers

- Algorithm Selection

- Location is implicit

```
// allocate host vector with two elements
thrust::host_vector<int> h_vec(2);

// copy host data to device memory
thrust::device_vector<int> d_vec = h_vec;

// write device values from the host
d_vec[0] = 27;
d_vec[1] = 13;

// read device values from the host
int sum = d_vec[0] + d_vec[1];

// invoke algorithm on device
thrust::sort(d_vec.begin(), d_vec.end());

// memory automatically released
```

# Thrust

## Portability

- Support for CUDA, TBB and OpenMP
  - Just recompile!

```
nvcc -DTHRUST_DEVICE_SYSTEM=THRUST_HOST_SYSTEM_OMP
```

### NVIDIA GeForce GTX 580

```
$ time ./monte_carlo
pi is approximately 3.14159

real    0m6.190s
user    0m6.052s
sys 0m0.116s
```

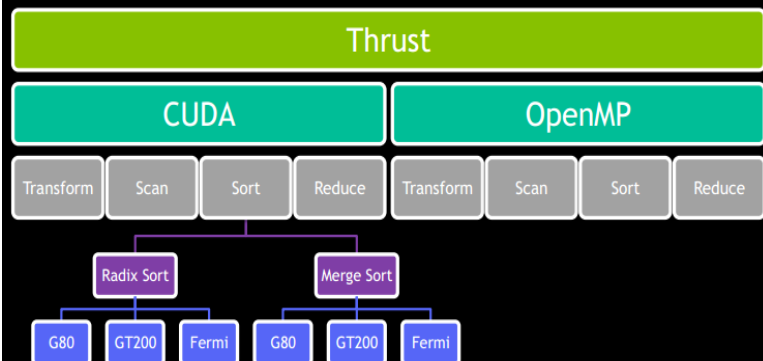
### Intel Core i7 2600K

```
$ time ./monte_carlo
pi is approximately 3.14159

real    1m26.217s
user    11m28.383s
sys 0m0.020s
```

MISMO código  
Corre en CPU  
Multicore!!

## Performance Portability



Thrust Content from GTC 2012: Nathan Bell

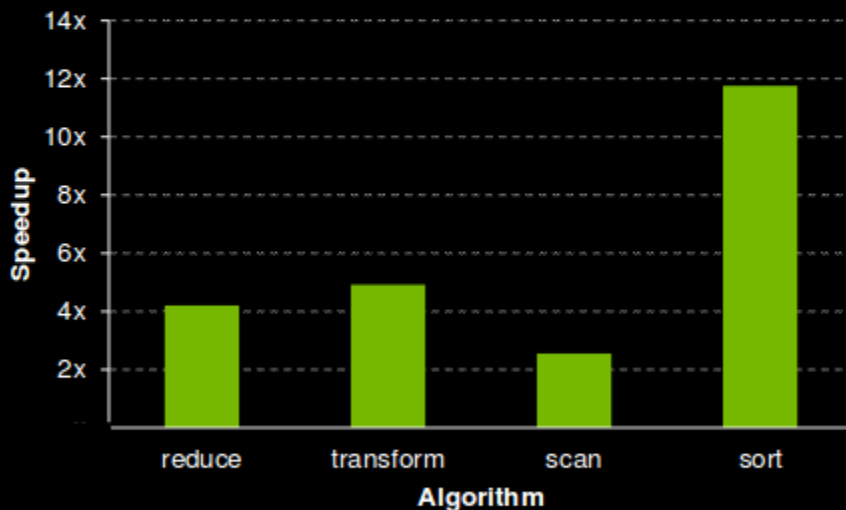
Delegamos la implementación  
de bajo nivel o “mapeo al hardware”  
a la librería

# Thrust

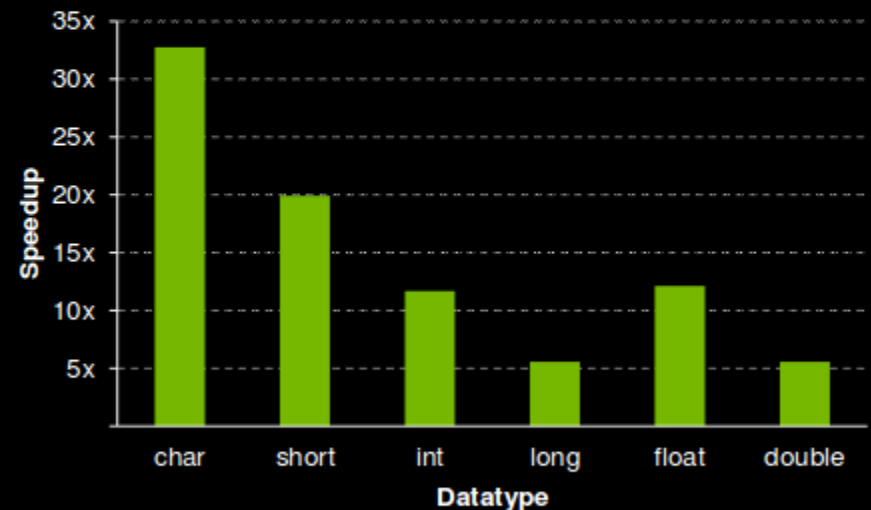
## CUDA Toolkit 5.0 Performance Report

### Thrust Performance

**Various Algorithms (32M int.)  
Speedup over TBB**



**Sort (32M samples)  
Speedup over TBB**



Performance may vary based on OS version and motherboard configuration

- Thrust 5.0 on K20X, input and output data on device
- TBB 4.1 on Intel SandyBridge E5-2687W @3.10GHz

# Thrust

Thrust Content from GTC 2012: Nathan Bell

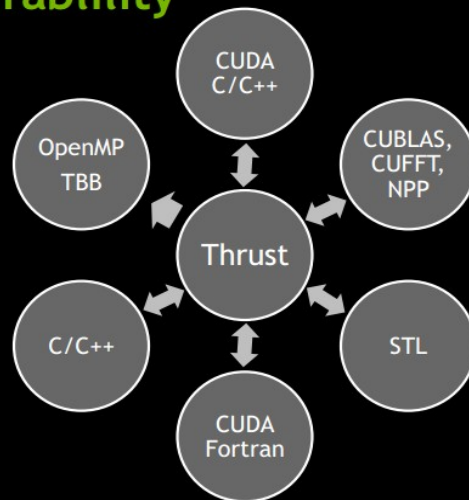
## Productivity

- Large set of algorithms
  - ~75 functions
  - ~125 variations
- Flexible
  - User-defined types
  - User-defined operators

Algorithm	Description
<code>reduce</code>	Sum of a sequence
<code>find</code>	First position of a value in a sequence
<code>mismatch</code>	First position where two sequences differ
<code>inner_product</code>	Dot product of two sequences
<code>equal</code>	Whether two sequences are equal
<code>min_element</code>	Position of the smallest value
<code>count</code>	Number of instances of a value
<code>is_sorted</code>	Whether sequence is in sorted order
<code>transform_reduce</code>	Sum of transformed sequence

La mayoría son  
“Parallel primitives”

## Interoperability



Thrust se puede usar para alivianar la escritura de gran parte del código:

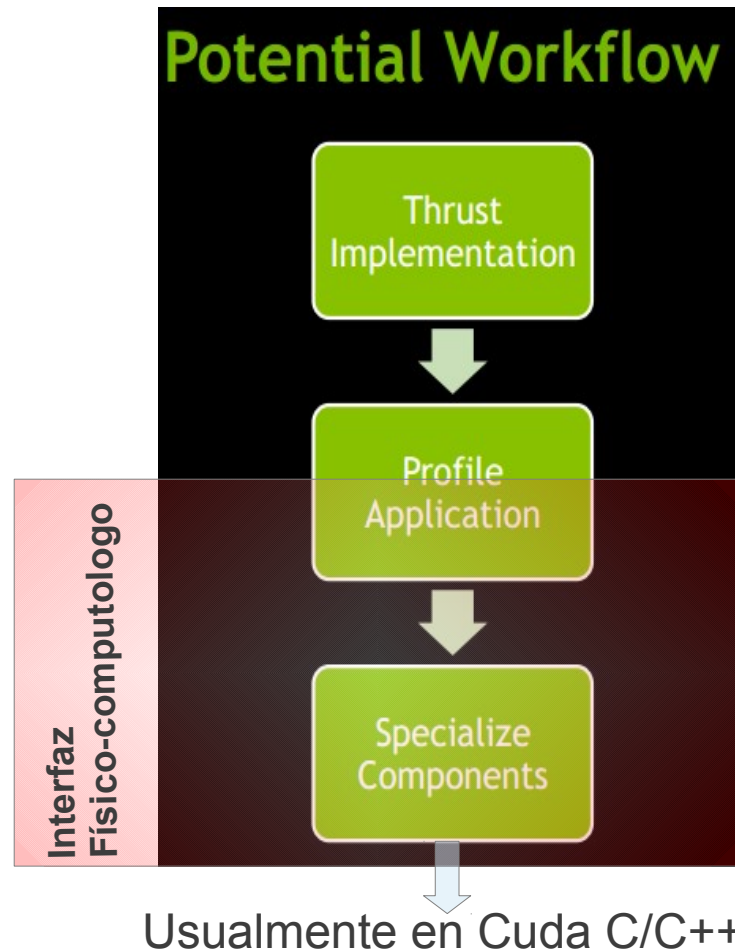
Alocacion de memoria, copias, y composición de patrones paralelas.

# Limitaciones de Thrust

Review de M. Harris en:

<https://developer.nvidia.com/content/expressive-algorithmic-programming-thrust>

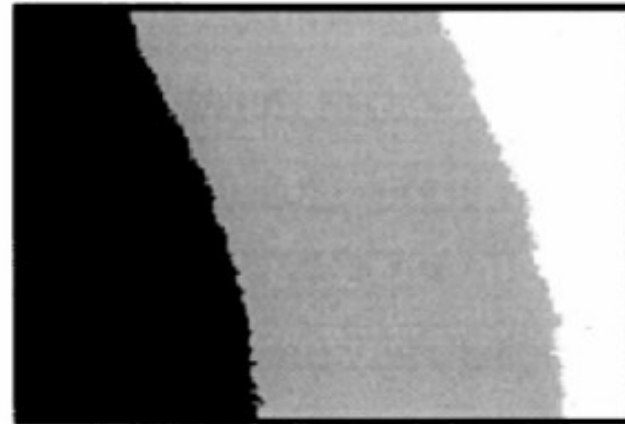
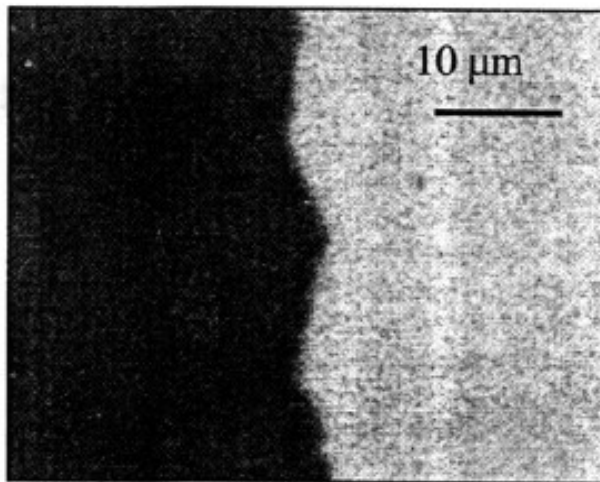
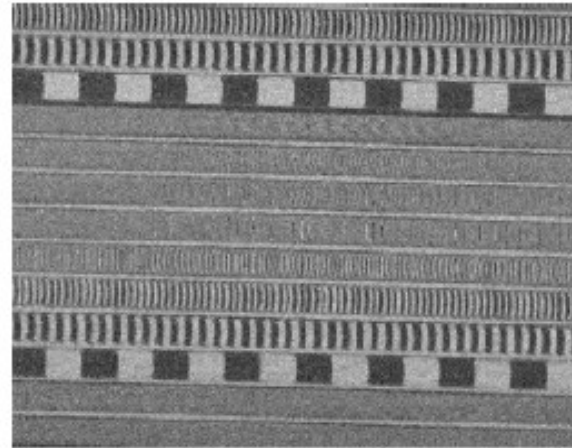
GTC 2012 Nathan Bell



# Un ejemplo concreto de aplicación científica

- **Modelo para la dinámica de una interfaz en un medio desordenado:**
  - *Matemáticamente:* PDE estocástica en función del tiempo para un campo escalar unidimensional en un potencial desordenado.
  - *Con los dedos:* una cuerda elástica que se mueve en un paisaje desordenado.

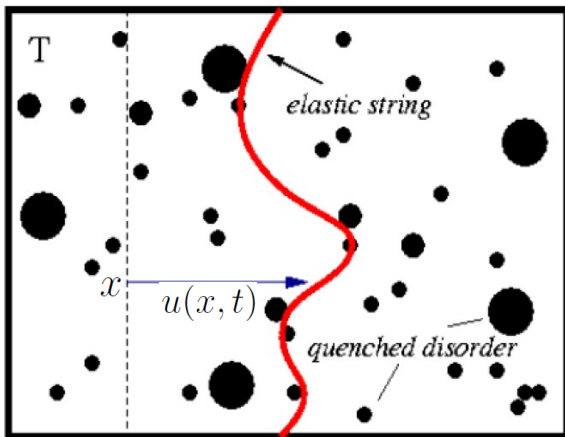
# Interfaces en medios desordenados



Magnetic domain wall in a Pt/Co/Pt film in the creep regime.  
Lemerle, Jamet, Ferre, et al (LPS Orsay).



# Un modelo mínimo para capturar la dinámica “universal” de las paredes



$$\mathcal{H}_{\text{el}}[u] \propto \int_r \sqrt{1 + |\nabla u|^2}$$

elastic limit of small distortions  $|\nabla u| \ll 1$

$$H_{\text{el}}[u] = \int_r \frac{c}{2} |\nabla u|^2$$

$$\gamma \partial_t u(x, t) = c \partial_x^2 u(x, t) + F_p(u, x) + f + \eta(x, t).$$

**Desorden congelado**

$$F_p(u, x) = -\partial_u U(u, x)$$

$$\overline{F_p(u, x) F_p(u', x')} = \Delta(u - u') \delta(x - x'),$$

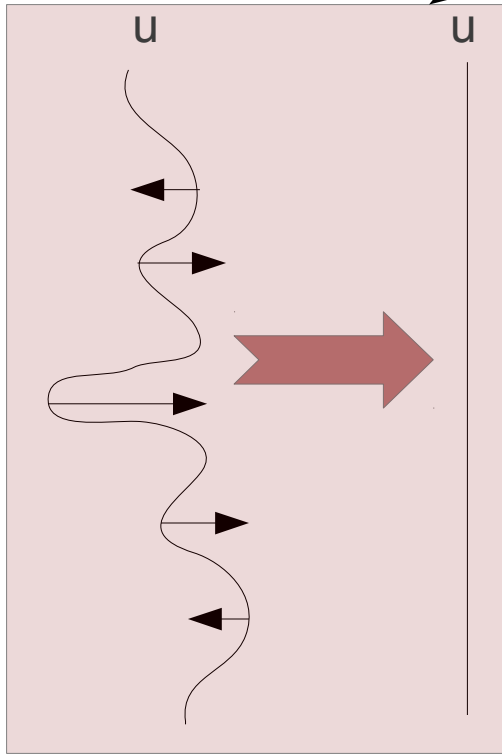
**Desorden dinámico (térmico)**

$$\langle \eta(x, t) \rangle = 0$$

$$\langle \eta(x, t) \eta(x', t') \rangle = 2k_B T \gamma \delta(x - x') \delta(t - t')$$

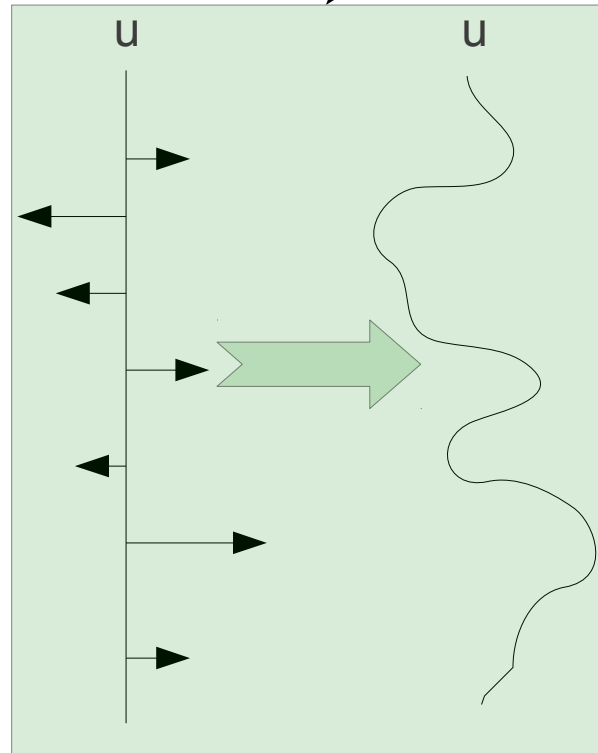
# Un modelo mínimo para capturar la dinámica “universal” de las paredes

$$\gamma \partial_t u(x, t) = \underbrace{c \partial_x^2 u(x, t)}_{\text{suaviza}} + \underbrace{F_p(u, x)}_{\text{distorsiona}} + \underbrace{f}_{\text{empuja}} + \eta(x, t)$$



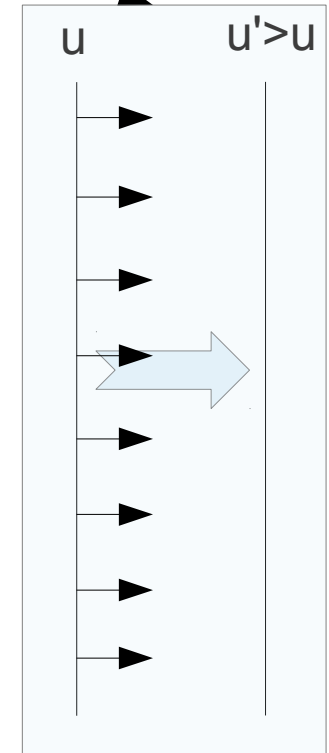
suaviza

VS



distorsiona

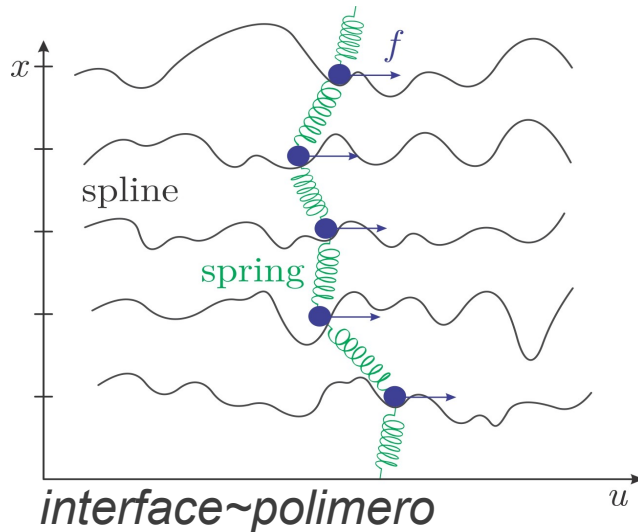
VS



empuja

# Implementación numérica del modelo

$$\gamma \partial_t u(x, t) = c \partial_x^2 u(x, t) + F_p(u, x) + f + \eta(x, t)$$



$$x = X \Delta x, \quad t = n \Delta t, \quad u \text{ continua}$$

$$F_p(u, x) \rightarrow F_p(u, X) = \text{random splines}$$

$$\eta(x, t) \rightarrow \eta(X, n) = \text{random numbers}$$

Implementación mas simple: diferencias finitas, Euler explicito

$$F_{tot}(X, n) = C[u(X-1, n) + u(X+1, n) - 2u(X, n)] \\ + F_p[u(X, n), X] + f + \sqrt{\frac{2T}{\Delta t}} R(X, n)$$

$$u(X, n+1) = u(X, n) + F_{tot}(X, n) \Delta t$$

# Implementación numérica del modelo

Implementación mas simple: explicit finite-difference

$$u(X, n + 1) = u(X, n) + F_{tot}(X, n)\Delta t$$

$$F_{tot}(X, n) = C[u(X - 1, n) + u(X + 1, n) - 2u(X, n)] \\ + F_p[u(X, n), X] + f + \sqrt{\frac{2T}{\Delta t}} R(X, n)$$

---

## ***“Casi Embarasosamente Paralelo”***

1) ***“Parallel For”***(X=0,...,L-1)

Thread “X” calcula la fuerza en X,  $F_{tot}(X) \rightarrow$  Array “Ftot” (device memory)

2) ***“Parallel For”***(X=0,...,L-1)

Thread “X” calcula  $u(X, n+1) \rightarrow$  Nuevo array “u” (device memory)

3) ***“parallel reductions”*** : calculo cantidades de interés (o **visualización** ...)

4) Vuelvo a (1), en un ***“For(n=0, 1,..., Trun)”*** de evolución temporal

# Implementación mas naive en GPU

```
// varios #include<...>
using namespace thrust;
int main()
{
```



```
// vectores que viven en la device (GPU global memory)
// declaracion y alocacion:
device_vector<REAL> u(L); // mi pared (discretizada)
device_vector<REAL> Ftot(L); // la fuerza sobre mi pared
```

```
// iteradores sobre GPU global memory: definen rangos
// necesarios para aplicar algoritmos paralelos genericos
device_vector<REAL>::iterator u_begin = u.begin();
device_vector<REAL>::iterator Ftot_begin = Ftot.begin();
```

```
// genero una evolucion temporal de la pared
for(long n=0;n<Trun;n++)
{
```

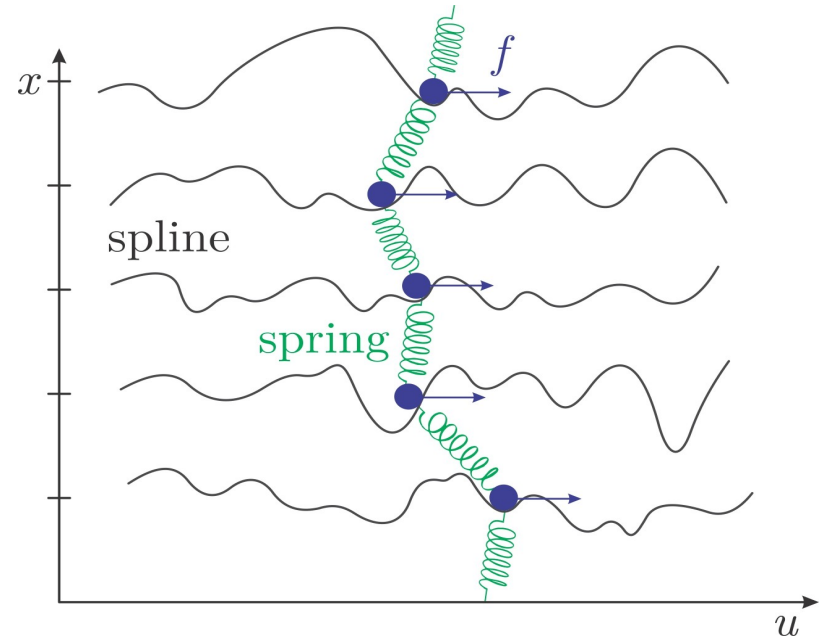
```
....
// (1) calculo paralelo Ftot(X,n) en la GPU
```

```
....
// (2) calculo paralelo u(X,n+1) en la GPU
```

```
.....
// (3) calculo paralelo propiedades en GPU
```

```
.....
// copio de mGPU -> mCPU, lo mínimo necesario!
```

```
}
....
}
```



$$F_{tot}(X, n) = C[u(X - 1, n) + u(X + 1, n) - 2u(X, n)] + F_p[u(X, n), X] + f + \sqrt{\frac{2T}{\Delta t}} R(X, n)$$

$$u(X, n + 1) = u(X, n) + F_{tot}(X, n)\Delta t$$

# Implementación en GPU

## Paso de Euler Paralelo

$$u(X, n + 1) = u(X, n) + F_{tot}(X, n)\Delta t$$

```
#define Dt ....
struct paso_de_Euler
{
    __device__
    REAL operator()(REAL u_old, REAL Ftot)
    {
        // u_old == u(X,n)
        // Ftot == Ftot(X,n)
        return (u_old + Ftot*Dt); // → u(X,n+1)
    }
}
```

Functor

“Parecido” a un kernel de cuda *pero*:

- La implementación de mas bajo nivel esta delegada (ej: en threads/blocks de Cuda).
- La interfaz es uniforme y genérica: el input/output y las operaciones pueden ser user-defined (classes).

```
int main()
{
    // vectores
    device_vector<REAL> u(L); // mi pared (discretizada)
    device_vector<REAL> Ftot(L); // la fuerza sobre mi pared

    // iteradores
    device_vector<REAL>::iterator u_begin = u.begin();
    device_vector<REAL>::iterator Ftot_begin = Ftot.begin();

    // genero una evolución temporal de la pared
    for(long n=0;n<Trun;n++)
    {
        ....
        // (1) calculo Ftot(X,n) en el device (GPU)
        ....
        // (2) avanzo un paso de Euler:
        transform(
            u_begin,u_end, Ftot_begin, ← Inputs (y su rango)
            u_begin, ← Output (in-place)
            paso_de_Euler() ← functor
        );
        ....
        // calculo propiedades o visualizo (GPU)
        ....
        // copio de memGPU -> memCPU, lo mínimo necesario!
    }
    ....
}
```

# Implementación en GPU

## Calculo de las fuerzas

$$F_{tot}(X, n) = C[u(X-1, n) + u(X+1, n) - 2u(X, n)] \\ + F_p[u(X, n), X] + f + \sqrt{\frac{2T}{\Delta t}} R(X, n)$$

```
struct fuerza
{
    long t;
    fuerza(long _t):t(_t){}; // t == "estado interno"

    __device__
    REAL operator()(tuple<int,REAL,REAL,REAL> tt)
    {
        int tid = get<0>(tt); // tid==X

        REAL um1 = get<1>(tt); // u(X-1)
        REAL u = get<2>(tt); // u(X)
        REAL up1 = get<3>(tt); // u(X+1)

        ? // ... desorden = RN(tid,[u])+RN(tid,[u]+1)(u-[u]);
        // ... ruido = RN(tid,t);

        REAL fuerza =
        (up1+um1-2*u) + desorden + ruido;

        return fuerza; // → Ftot(X)
    }
}
```

```
int main()
{
    // vectores
    device_vector<REAL> u(L); // mi pared (discretizada)
    device_vector<REAL> Ftot(L); // la fuerza sobre mi pared
    // iteradores
    device_vector<REAL>::iterator u_begin = u.begin();
    device_vector<REAL>::iterator Ftot_begin = Ftot.begin();

    // genero una evolucion temporal de la pared
    for(long n=0;n<Trun;n++)
    {
        ....
        // (1) calculo Ftot(X,n) en el device (GPU)
        transform(
            make_zip_iterator(make_tuple(
                counting_iterator<int>(0),u_begin-1,u_begin,u_begin+1)),
            make_zip_iterator(make_tuple(
                counting_iterator<int>(L),u_end-1,u_end,u_end+1)),
            Ftot_begin,
            fuerza(n)
        );
        ....
        // (2) avanzo un paso de Euler:
        ....
        // calculo propiedades (GPU)
        ....
        // copio de mGPU -> mCPU, lo mínimo necesario!
    }
    ....
}
```



# Implementación en GPU

## Calculo de las fuerzas

$$F_{tot}(X, n) = C[u(X-1, n) + u(X+1, n) - 2u(X, n)] \\ + F_p[u(X, n), X] + f + \sqrt{\frac{2T}{\Delta t}} R(X, n)$$

```
struct fuerza
{
    long t;
    fuerza(long _t):t(_t){}; // t == "estado interno"

    __device__
    REAL operator()(tuple<int,REAL,REAL,REAL> tt)
    {
        int tid = get<0>(tt); // tid==X

        REAL um1 = get<1>(tt); // u(X-1)
        REAL u = get<2>(tt); // u(X)
        REAL up1 = get<3>(tt); // u(X+1)

        // ... desorden = RN(tid,[u])+RN(tid,[u]+1)(u-[u]);
        // ... ruido = RN(tid,t);

        REAL fuerza =
            (up1+um1-2*u) + desorden + ruido;

        return fuerza; // → Ftot(X)
    }
}
```

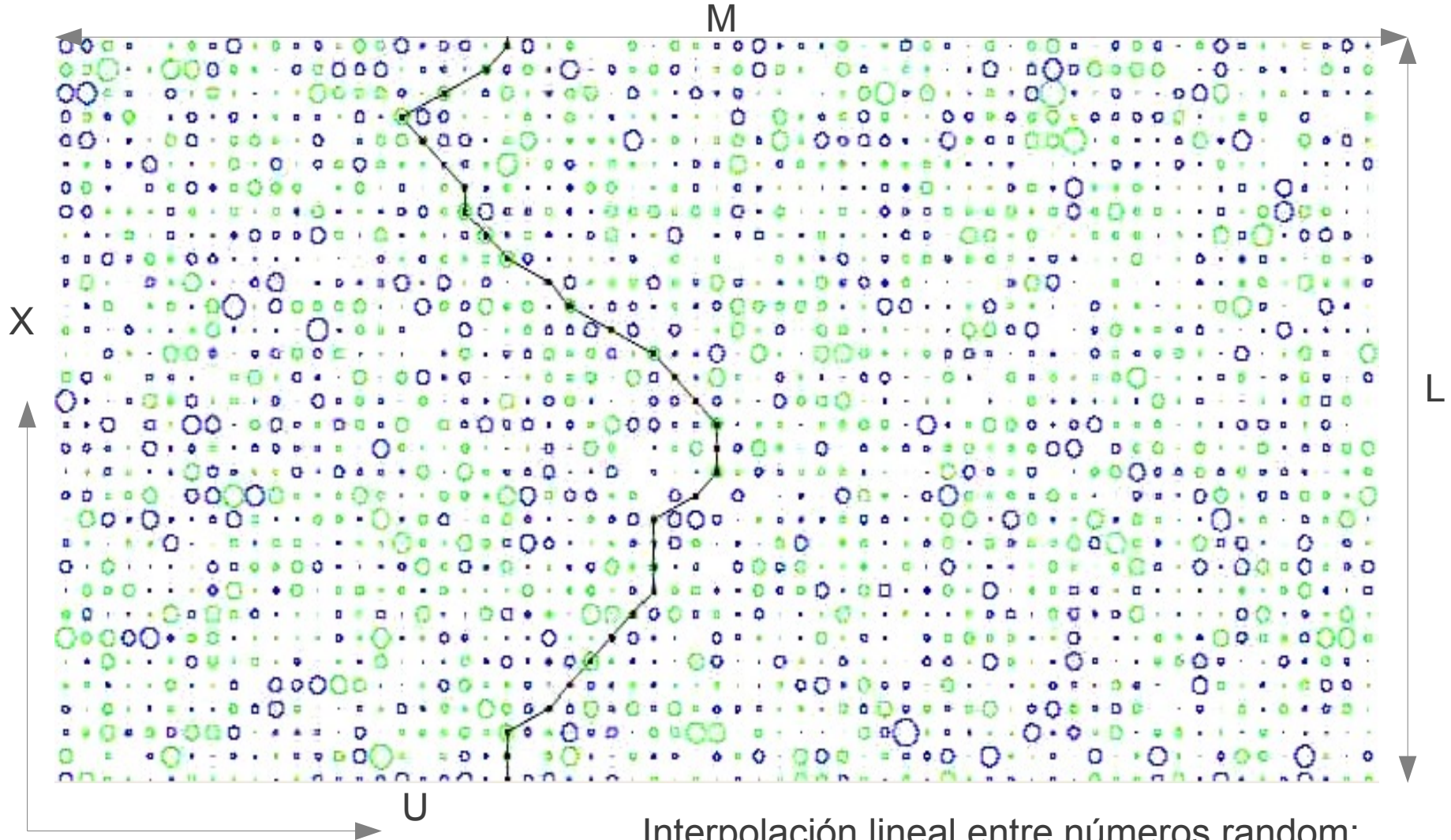
### Desorden (ruido congelado)

- Interpolacion entre números random gaussianos.
- *Secuencias de RNs descorrelacionadas solo en espacio, pero constantes en el tiempo...*

### Ruido térmico (ruido dinámico)

- Números random gaussianos.
- *Secuencias de RNs descorrelacionadas en tiempo y en espacio....*

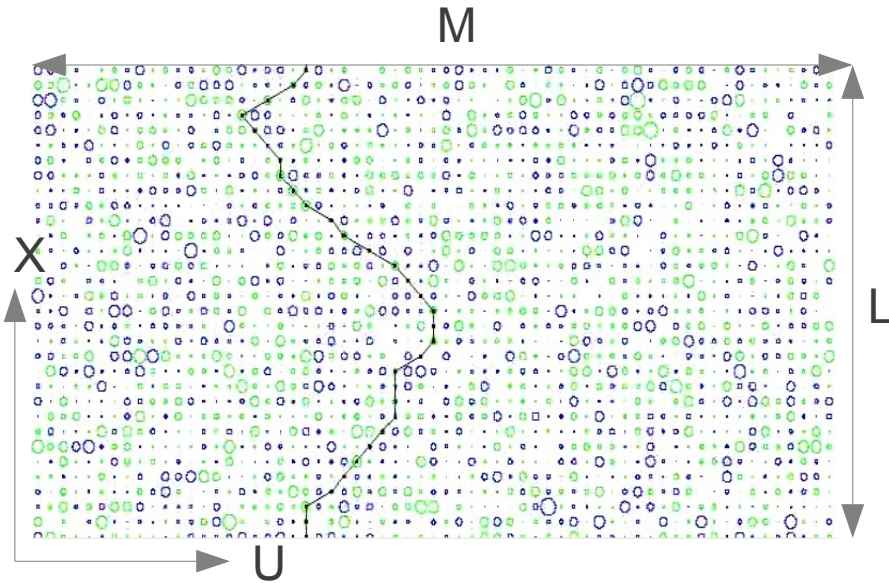
# Desorden



Interpolación lineal entre números random:

$$F_p[u(X, t), X] = f_p[U, X] + (f_p[U + 1, X] - f_p[U, X])(u - U)$$

# El problema con el Desorden



- Necesitamos simular muestras  $L \times M$ , con:  
 $L > 2^{16} = 65536$  y  $M \sim L^{5/4} \sim 2^{20}$ .
- Si guardáramos el desorden en una matriz de  $L \times M > 2^{16+20}$  floats = **256 Gb** (!!).

```
koltona@mostro:/opt/cudasdk/C/bin/linux/release$ ./deviceQuery
```


## Device 1: "Tesla C2075"

CUDA Driver Version / Runtime Version      4.2 / 4.1

CUDA Capability Major/Minor version number:    2.0

Total amount of global memory:                **5375 MBytes** (5636554752 bytes)

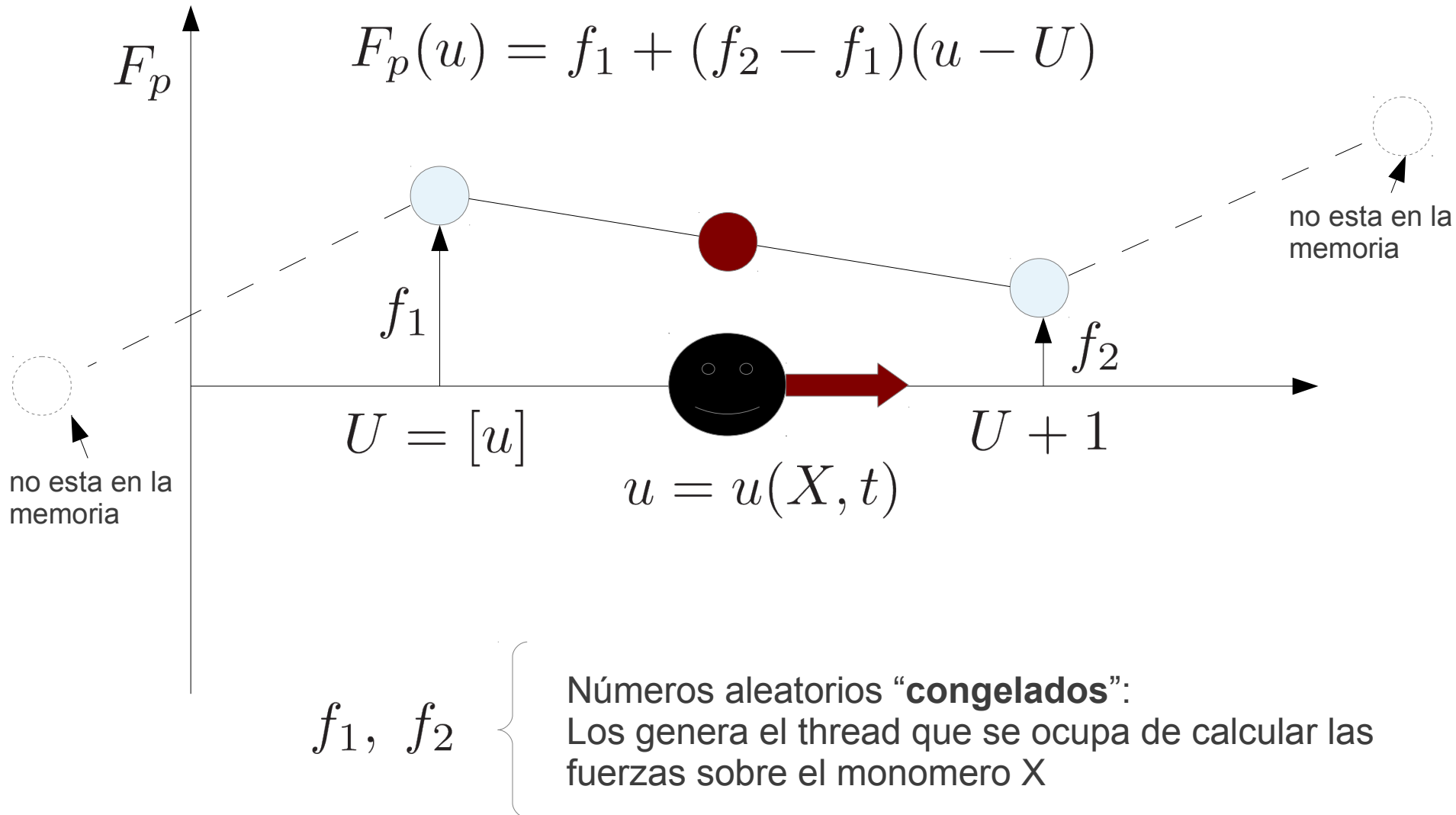
(14) Multiprocessors x (32) CUDA Cores/MP:    448 CUDA Cores

- 
- Guardar (todo) el desorden precalculado no es una opción...
  - Otras variantes son muy complicadas...
  - La mas simple → **Generar dinámicamente** usando un RNG paralelo.

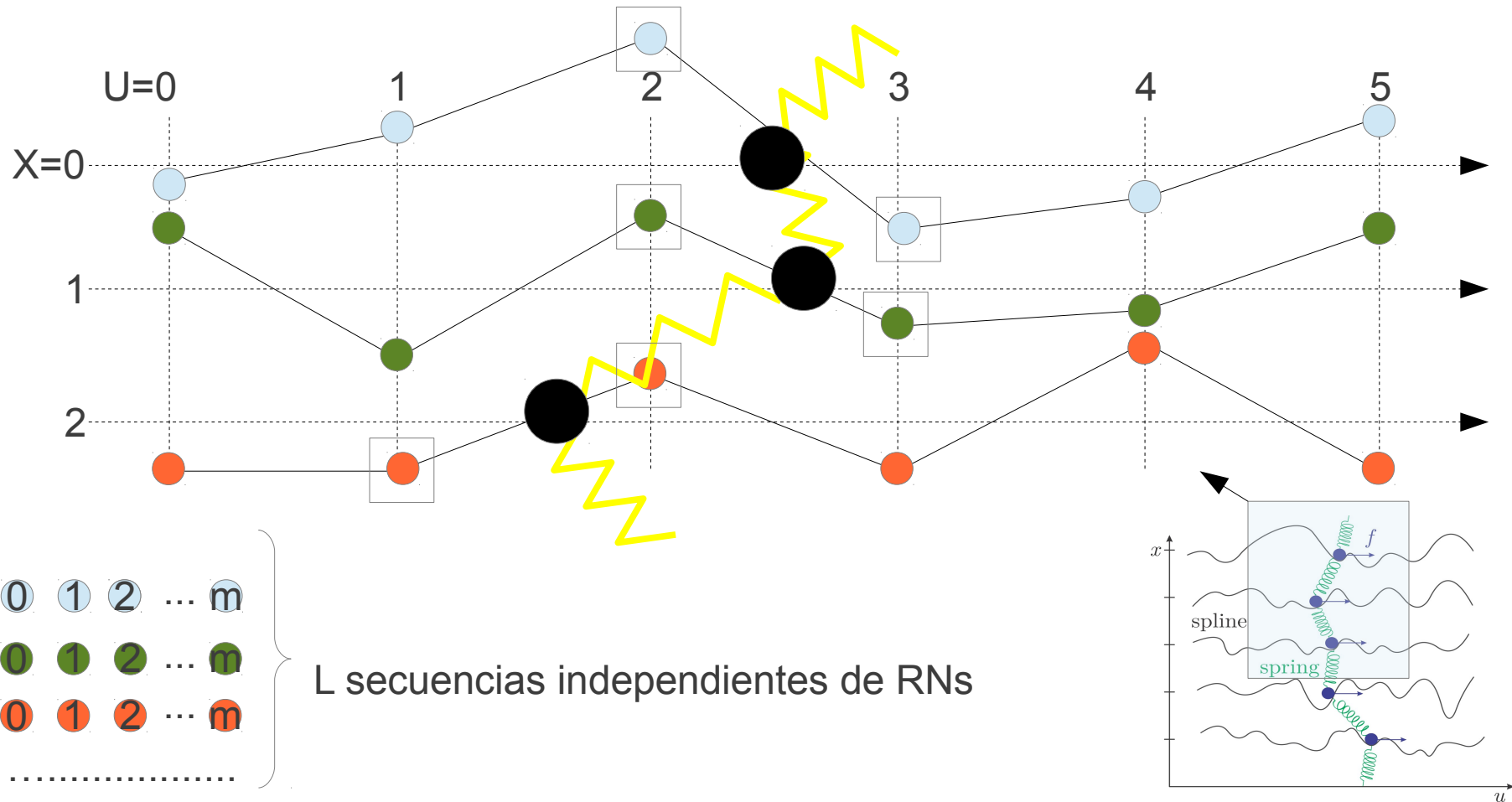
# Desorden generado dinámicamente

La mas simple fuerza continua: interpolacion lineal a trozos

$$F_p(u) = f_1 + (f_2 - f_1)(u - U)$$



# Generación dinámica del Desorden



$$F_p[u(2, t), 2] = \text{1} + \left[ \text{2} - \text{1} \right] (u(2, t) - [u(2, t)])$$

Tengo que sortear dos números aleatorios por partícula/thread...



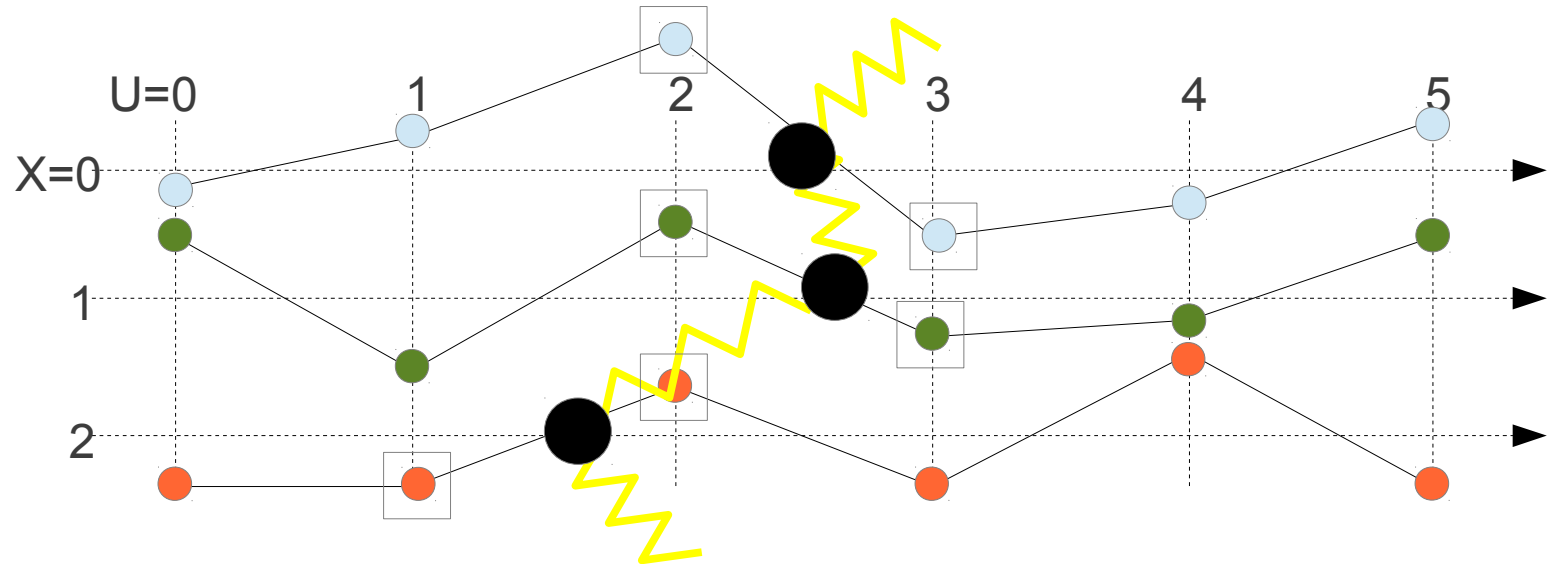
# Random number generators for massively parallel simulations on GPU

M. Manssen, M. Weigel, A. K. Hartmann

generator	bits/thread	failures in TestU01			Ising test	perf. $\times 10^9/s$
		SmallCrush	Crush	BigCrush		
LCG32	32	12	—	—	failed	58
LCG32, random	32	3	14	—	passed	58
LCG64	64	None	6	—	failed	46
LCG64, random	64	None	2	8	passed	46
MWC	64 + 32	1	29	—	passed	44
Fibonacci, $r = 521$	$\geq 80$	None	2	—	failed	23
Fibonacci, $r = 1279$	$\geq 80$	None	(1)	2	passed	23
XORWOW (cuRAND)	192	None	None	1/3	failed	19
MTGP (cuRAND)	$\geq 44$	None	2	2	—	18
XORShift/Weyl	32	None	None	None	passed	18
Philox4x32_7	(128)	None	None	None	passed	41
Philox4x32_10	(128)	None	None	None	passed	30

*Cual elegimos?*

# El problema con generar dinámicamente el Desorden congelado



$$F_p[u(2, t), 2] = \textcircled{1} + \left[ \textcircled{2} - \textcircled{1} \right] (u(2, t) - [u(2, t)])$$

Si la cuerda se mueve para adelante, genero siempre nuevos números aleatorios (lo usual). Esto es aceptable para las fluctuaciones térmicas, que nunca se repiten, pero...

*Que pasa si por una fluctuación térmica o una fuerza externa la cuerda va para atrás?*

*El desorden tiene que ser el mismo!!! →*

***Necesidad de recuperar un numero aleatorio ya generado, pero sin guardar nada en memoria global .... SE PUEDE ?***



El clima podrá cambiar, pero el paisaje cuando voy  
tiene que ser el mismo que cuando vuelvo!



# Random number generators for massively parallel simulations on GPU

M. Manssen, M. Weigel, A. K. Hartmann

*Regeneración sencilla de números ya generados?*

Counter-based RNGs

generator	bits/thread	failures in TestU01			Ising test	perf. $\times 10^9/s$
		SmallCrush	Crush	BigCrush		
LCG32	32	12	—	—	failed	58
LCG32, random	32	3	14	—	passed	58
LCG64	64	None	6	—	failed	46
LCG64, random	64	None	2	8	passed	46
MWC	64 + 3	29	—	—	passed	44
Fibonacci, $r = 521$	$\geq 0$	None	2	—	failed	23
Fibonacci, $r = 1279$	$\geq 80$	None	(1)	2	passed	23
XORWOW (cuRAND)	192	None	None	1/3	failed	19
MTGP (cuRAND)	$\geq 44$	None	2	2	—	18
XORShift/Weyl	32	None	None	None	passed	18
Philox4x32_7	(128)	None	None	None	passed	41
Philox4x32_10	(128)	None	None	None	passed	30

*Cual elegimos?*

# Algunas herramientas

C++ template library

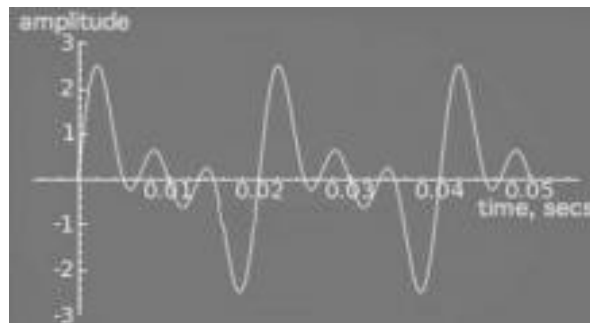


Random Numbers

**Random123:**

**a Library of Counter-Based  
Random Number Generators**

CUFFT



# Counter-Based RNGs



## Parallel Random Numbers: As Easy as 1, 2, 3

John K. Salmon, Mark A. Moraes, Ron O. Dror, David E. Shaw

“.. We demonstrate that independent, **keyed transformations of counters** produce a large alternative class of PRNGs with **excellent statistical properties**... are ideally suited to modern multicore CPUs, GPUs, clusters, and special-purpose hardware because they vectorize and parallelize well, and **require little or no memory for state**. ... All our PRNGs pass **rigorous statistical tests** and produce at least  **$2^{64}$  unique parallel streams of random numbers, each with period  $2^{128}$**  or more. In addition to essentially unlimited parallel scalability, our PRNGs offer excellent single-chip performance: **Philox** is faster than the CURAND library on a single NVIDIA GPU.



# Counter-Based RNGs

## Overview

counter-based RNGs are stateless functions whose arguments are a counter, and a key and whose return value is the same type as the counter.

**value = CBRNGname(counter, key)**

The returned value is a deterministic function of the key and counter, i.e. a unique (counter, key) tuple will always produce the same result. The result is highly sensitive to small changes in the inputs, so that the sequence of values produced by simply incrementing the counter (or key) is effectively indistinguishable from a sequence of samples of a uniformly distributed random variable.

Exactamente lo que necesitamos!

$$F_p[u(X, t), X] = \underbrace{f_p[U, X]}_{\text{CBRNG}(U, X)} + ((f_p[U + 1, X] - f_p[U, X]))(u - U)$$

$\text{CBRNG}(U, X) \qquad \text{CBRNG}(U, X)$

Counter = U = (int)u  
Key = X = thread/particle ID

# Counter-Based RNGs

Exactamente lo que necesitamos!

- Buena calidad, state-less
- **Fácil de usar**
- **RNs recuperables!**

DESORDEN

$$F_p[u(X, t), X] = \underbrace{f_p[U, X]}_{\text{CBRNG}(U, X)} + (\underbrace{f_p[U + 1, X]}_{\text{CBRNG}(U+1, X)} - f_p[U, X])(u - U)$$

Counter = U = (int)u  
Key = X = thread/particle ID

TEMPERATURA

$$R(X, t) = \text{CBRNG}'(t, X)$$

Counter = n (tiempo)  
Key = X = thread/particle ID

Implementación mas simple: explicit finite-difference

$$F_{tot}(X, n) = C[u(X - 1, n) + u(X + 1, n) - 2u(X, n)] \\ + F_p[u(X, n), X] + f + \sqrt{\frac{2T}{\Delta t}} R(X, n)$$

$$u(X, n + 1) = u(X, n) + F_{tot}(X, n)\Delta t$$



# Implementación en GPU

## Cálculo de las fuerzas

$$F_{tot}(X, n) = C[u(X-1, n) + u(X+1, n) - 2u(X, n)] \\ + F_p[u(X, n), X] + f + \sqrt{\frac{2T}{\Delta t}} R(X, n)$$

```
struct fuerza
{
    long t;
    fuerza(long _t):t(_t){}; // t == "estado interno"

    __device__
    REAL operator()(tuple<int,REAL,REAL,REAL> tt)
    {
        int tid = get<0>(tt); // tid==X

        REAL um1 = get<1>(tt); // u(X-1)
        REAL u    = get<2>(tt); // u(X)
        REAL up1  = get<3>(tt); // u(X+1)
        int U = (int)u;

        desorden = CBRNG(U,tid)+CBRNG(U+1,tid)(u-U);
        ruido     = CBRNG(t,tid+L);

        REAL fuerza =
        (up1+um1-2*u) + desorden + ruido;

        return fuerza; // → Ftot(X)
    }
}
```

```
int main()
{
    // vectores
    device_vector<REAL> u(L); // mi pared (discretizada)
    device_vector<REAL> Ftot(L); // la fuerza sobre mi pared
    // iteradores
    device_vector<REAL>::iterator u_begin = u.begin();
    device_vector<REAL>::iterator Ftot_begin = Ftot.begin();

    // genero una evolución temporal de la pared
    for(long n=0;n<Ttot;n++)
    {
        ....
        // (1) calculo Ftot(X,T) en el device (GPU)
        transform(
            make_zip_iterator(make_tuple(
                counting_iterator<int>(0),u_begin-1,u_begin,u_begin+1)),
            make_zip_iterator(make_tuple(
                counting_iterator<int>(L),u_end-1,u_end,u_end+1)),
            force_begin,
            fuerza(n)
        );
        ....
        // (2) avanzo un paso de Euler:
        ....
        // calculo propiedades o visualizo (GPU)
        ....
        // copio de mGPU -> mCPU, lo mínimo necesario!
    }
    ....
}
```



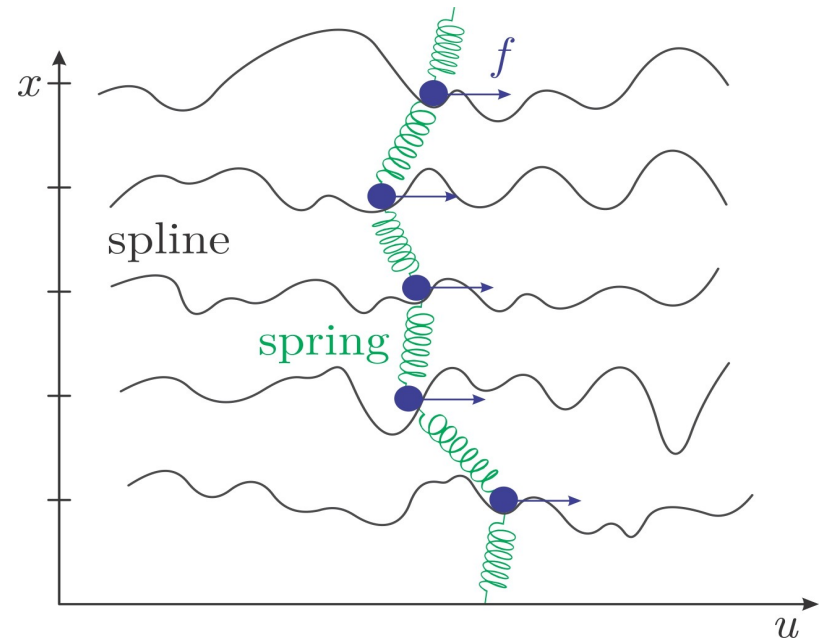
# Implementación en GPU

```
// varios #include<...>
using namespace thrust;
int main()
{
    // vectores que viven en la device (GPU global memory)
    // declaracion y alocacion:
    device_vector<REAL> u(L); // mi pared (discretizada)
    device_vector<REAL> Ftot(L); // la fuerza sobre mi pared

    // iteradores sobre GPU global memory: definen rangos
    // necesarios para aplicar algoritmos paralelos genéricos
    device_vector<REAL>::iterator u_begin = u.begin();
    device_vector<REAL>::iterator Ftot_begin = Ftot.begin();

    // genero una evolución temporal de la pared
    for(long n=0;n<Trun;n++)
    {
        ....
        // (1) calculo Ftot(X,n) en la GPU
        ....
        // (2) calculo u(X,n+1) en la GPU
        ....
        // (3) calculo propiedades (GPU)
        ....
        // copio de mGPU -> mCPU, lo mínimo necesario!
    }
    ....
}
```

LISTO!



Implementación simple  
pero rapidísima !

# Implementación en GPU

```
// varios #include<...>
using namespace thrust;
int main()
{

    // vectores que viven en la device (GPU global memory)
    // declaracion y alocacion:
    device_vector<REAL> u(L); // mi pared (discretizada)
    device_vector<REAL> Ftot(L); // la fuerza sobre mi pared

    // iteradores sobre GPU global memory: definen rangos
    // necesarios para aplicar algoritmos paralelos genéricos
    device_vector<REAL>::iterator u_begin = u.begin();
    device_vector<REAL>::iterator Ftot_begin = Ftot.begin();

    // genero una evolución temporal de la pared
    for(long n=0;n<Trun;n++)
    {
        ....
        // (1) calculo Ftot(X,n) en la GPU
        ....
        // (2) calculo u(X,n+1) en la GPU
        ....
        // (3) calculo propiedades (GPU)
        ....
        // copio de mGPU -> mCPU, lo mínimo necesario!
    }
    ....
}
```

LISTO!

Implementación simple  
pero rápida

Mismo código: 2 ejecutables  
para distintas devices

- **PARALELIZADO EN GPU**
- ✓ nvcc -o **cuerdaCUDA** cuerda.cu

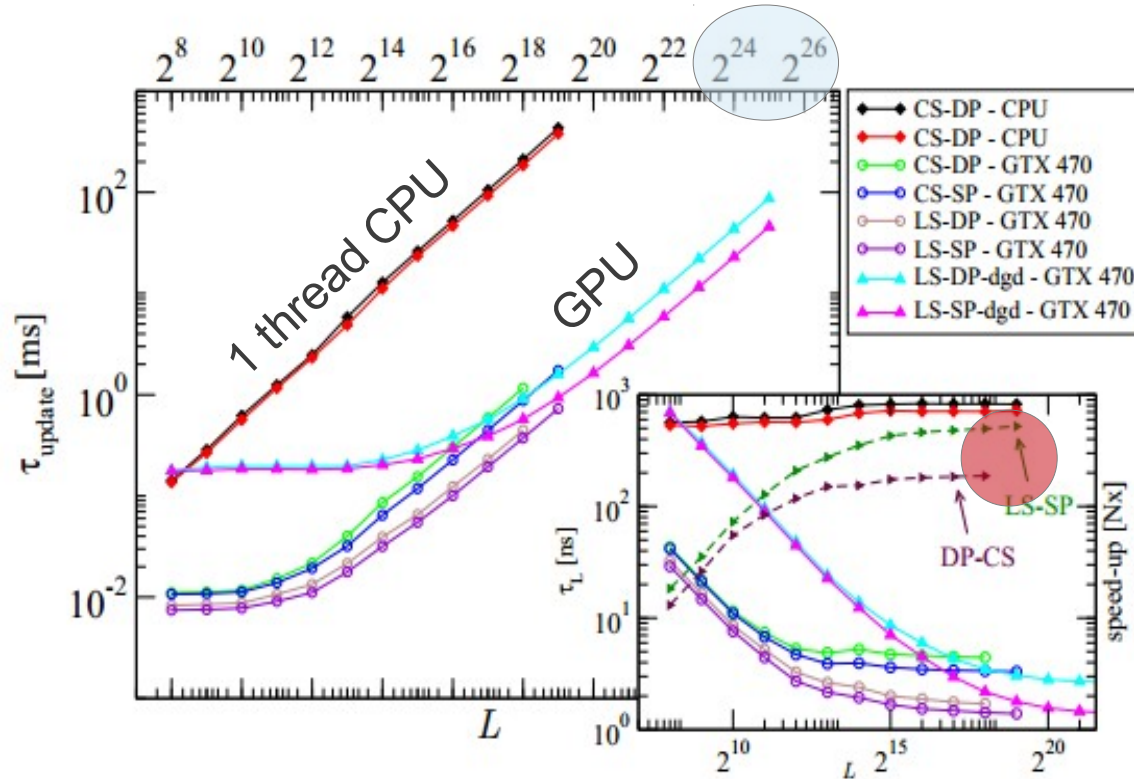
- **PARALELIZADO EN CPU-MULTITHREAD**
- ✓ cp cuerda.cu cuerda.cpp
- ✓ g++ -o **cuerdaOMP** cuerda.cpp -O2 -fopenmp -DTHRUST\_DEVICE\_BACKEND=THRUST\_DEVICE\_BACKEND\_OMP -lgomp

*En la última versión de thrust  
"Direct System Access":*

- *System specific vectors*
- *Execution policies.*

# Resultados

## Performance



- Sistemas mucho mas grandes (**33554432** partículas) → Exponentes críticos mucho mas precisos.

- **Nueva física:** Descubrimos “correcciones de scaling” → explican una controversia.

- **CPU:** AMD Phenom(tm) II X4 955 Processor @3.2GHz
- **GPU:** NVIDIA GTX 470.

# Resultados

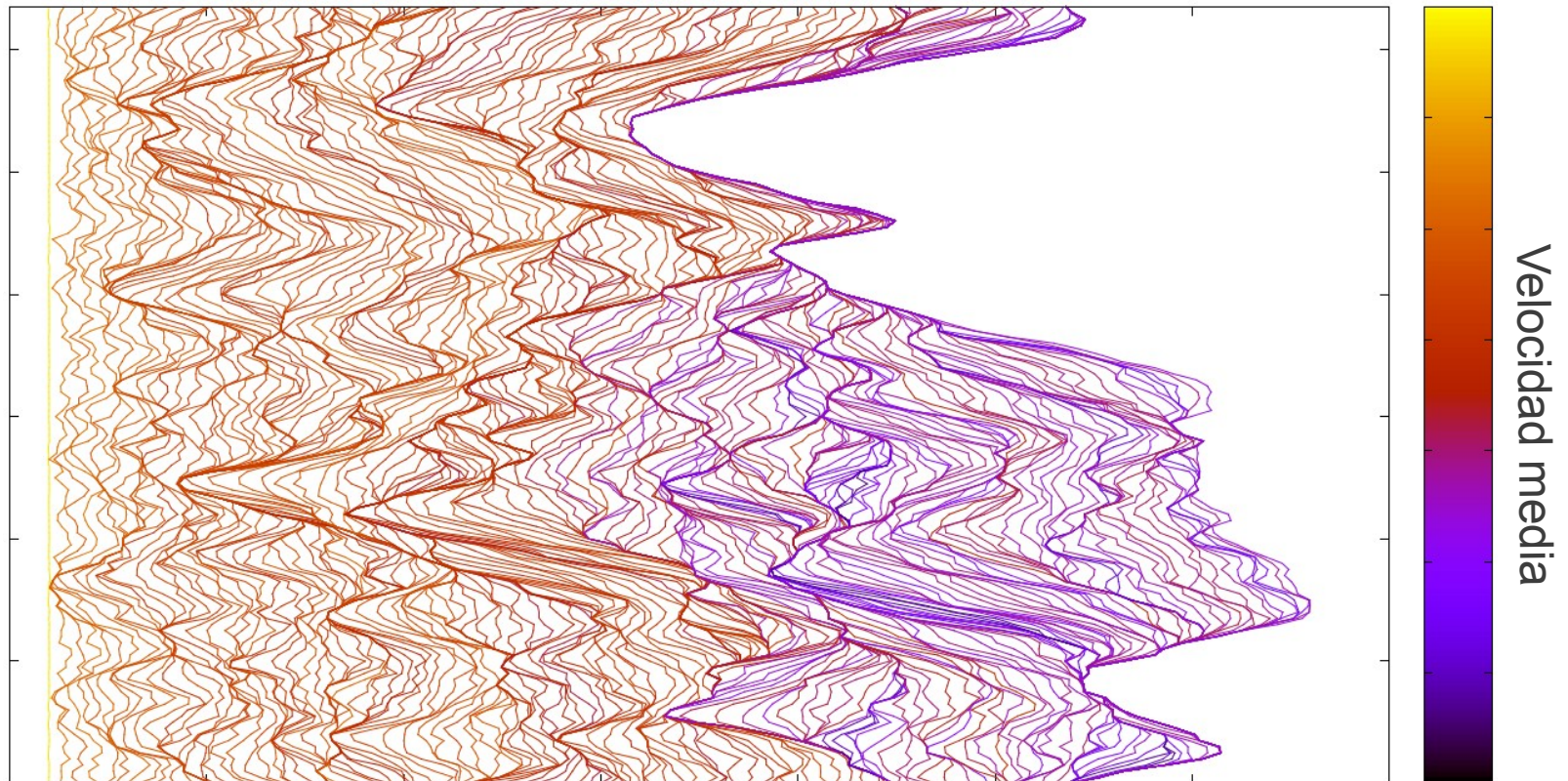
PHYSICAL REVIEW E **87**, 032122 (2013)

## Nonsteady relaxation and critical exponents at the depinning transition

E. E. Ferrero, S. Bustingorry, and A. B. Kolton

*CONICET, Centro Atómico Bariloche, 8400 San Carlos de Bariloche, Río Negro, Argentina*

(Received 28 November 2012; revised manuscript received 4 February 2013; published 11 March 2013)





# Algunas herramientas

C++ template library

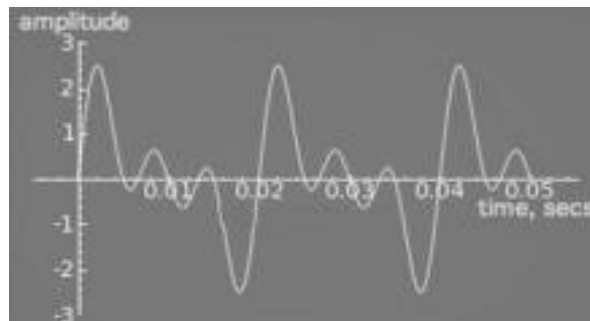
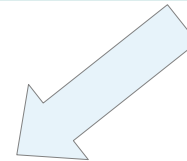


Random Numbers

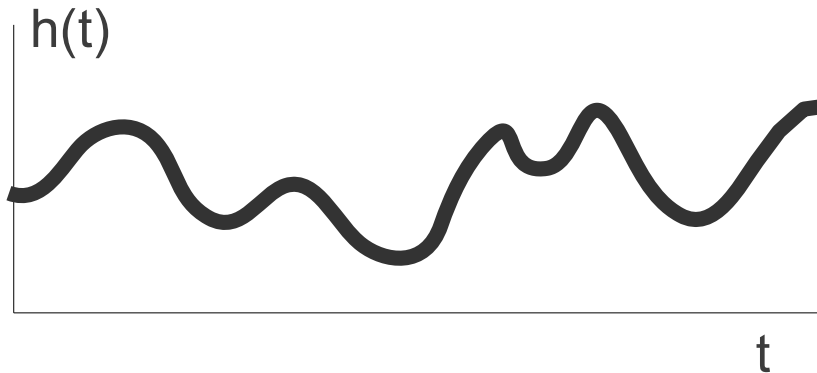
**Random123:**

**a Library of Counter-Based  
Random Number Generators**

CUFFT



# 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

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

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

Operador...

Interacción...

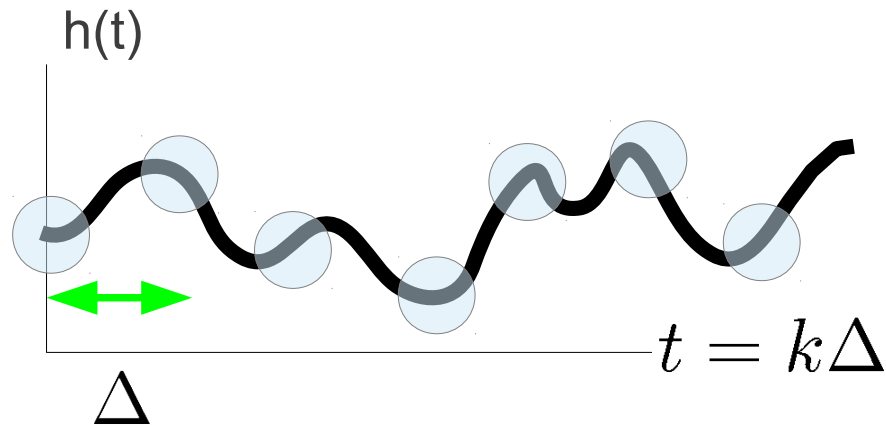
## Correlation theorem

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

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

# Transformada de Fourier Discreta

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



## Sampling theorem

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

$$f_n \equiv \frac{n}{N\Delta}, \quad n = -\frac{N}{2}, \dots, \frac{N}{2}$$

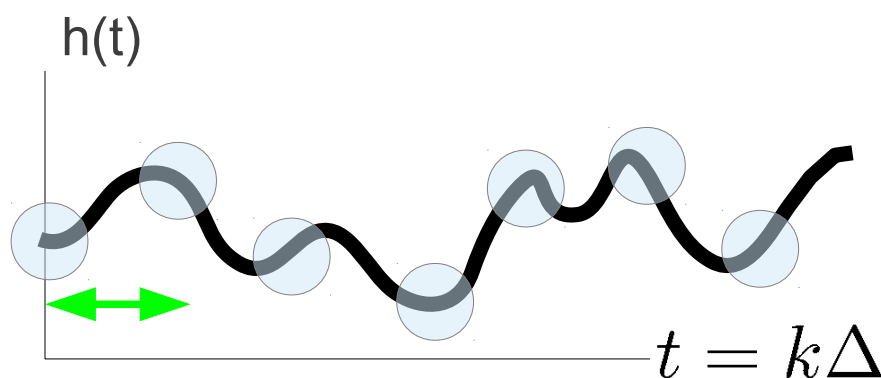
Transformada  
discreta

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



# Transformada de Fourier Discreta

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



Transformada  
discreta

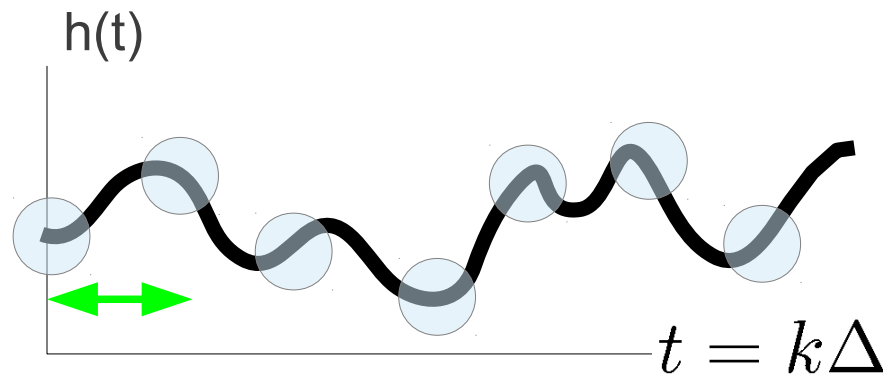
$$H_n \equiv \sum_{k=0}^{N-1} h_k e^{2\pi i k n / N}$$

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

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



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

Transformada discreta 1D

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

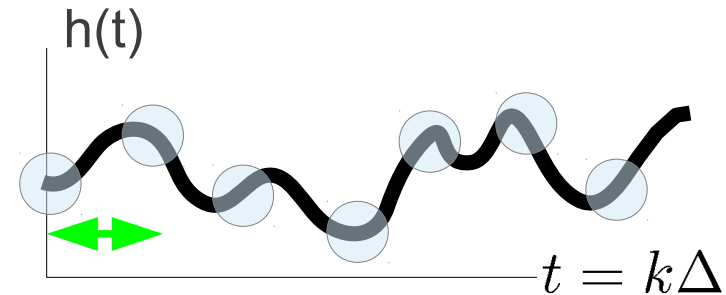
Transformada discreta 2D

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

Transformada discreta 3D, etc

# Transformada de Fourier Discreta

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



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?

Algoritmos  
Secuenciales

{ Naïve approach [hasta ~1960]:  $O(N^2)$   
Algoritmos de *Fast Fourier transform* (FFT)  
[Cooley–Tukey o Gauss?] :  $O(N \log_2 N)$  → recursivo

# Fast Fourier transforms

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."

---

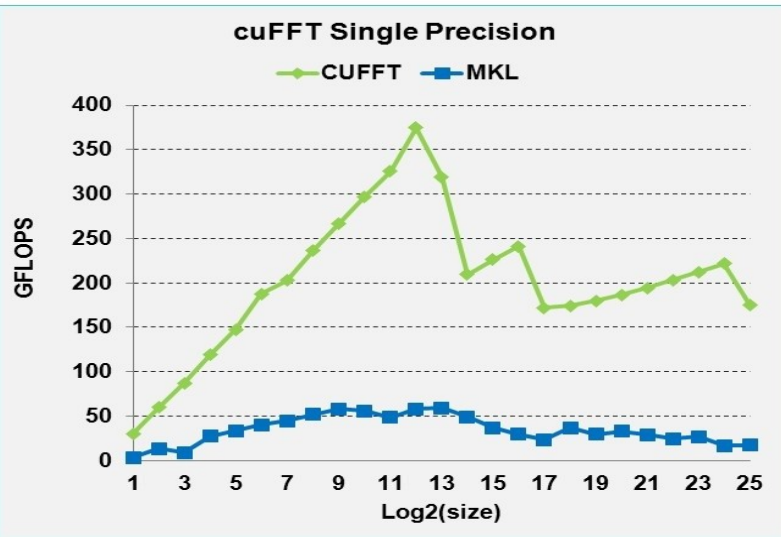
Vendor-tuned codes:

## MKL: Math Kernel Library

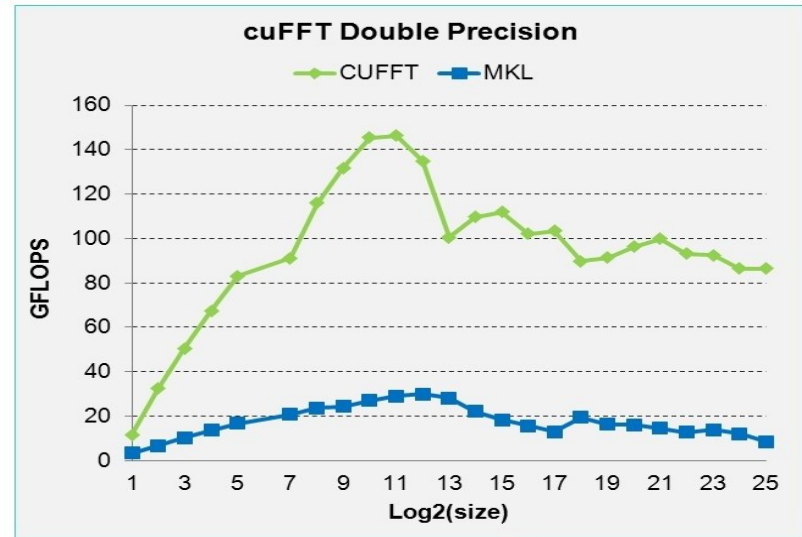
Intel's Math Kernel Library (MKL) is a library of optimized, math routines for science, engineering, and financial applications. Core math functions include BLAS, LAPACK, ScaLAPACK, Sparse Solvers, **Fast Fourier Transforms**, and Vector Math.

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

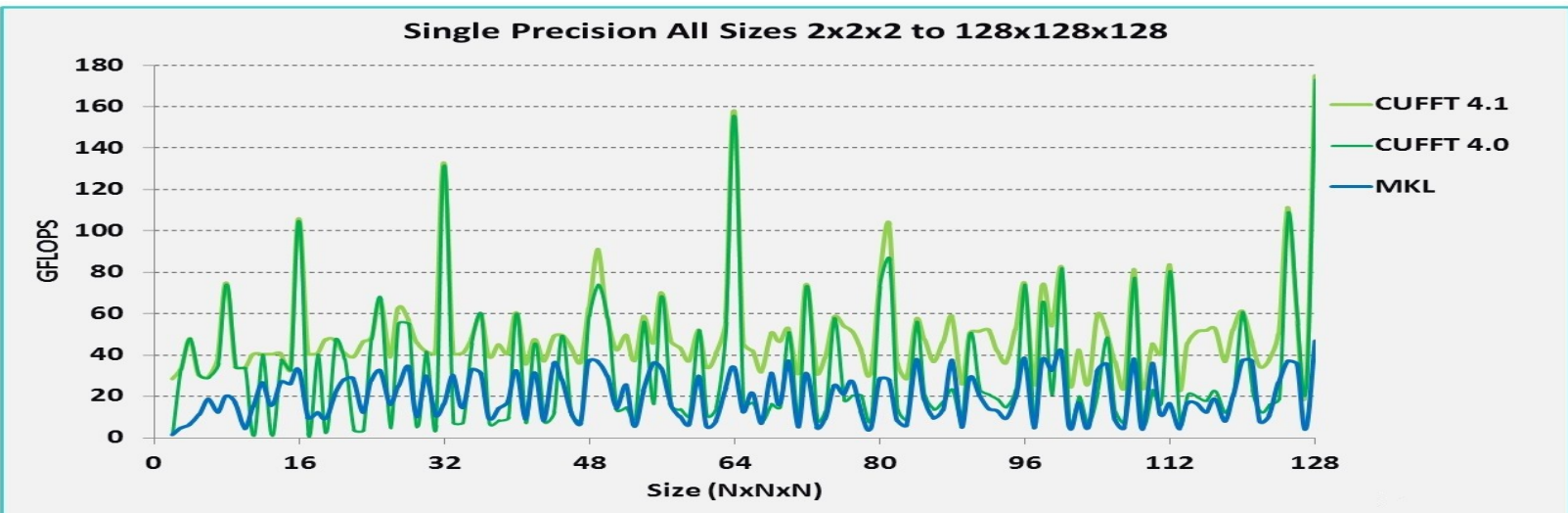
# Cuda FFT: CUFFT



- 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 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

- Performance may vary based on OS ver. and motherboard config.

# Interoperabilidad CUFFT - 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]);

    // crea un plan
    cufftHandle planfft;
    cufftPlan1d(&planfft, N, CUFFT_R2C, 1);

    // ejecuta el plan
    cufftExecR2C(planfft, d_input_raw, d_output_raw);

    // copio la transformada del device al host
    thrust::host_vector<REAL> output_host = output;

    // hago algo con la transformada, por ej., la imprimo:
    for(i=0; i < N/2+1; i++)
        cout << output_host[i].x << " " << output_host[i].y << endl;
}
```



# 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  
 $G(\mathbf{r}, \mathbf{r}') \sim 1/|\mathbf{r} - \mathbf{r}'|^3$

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

# 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  $L_x * L_y$  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  $N=L_x L_y$  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:

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

estabilidad



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

# 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)

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

$$\begin{cases} \phi^t \rightarrow \phi_{\mathbf{k}}^t \\ [\phi^3]^t \rightarrow [\phi^3]_{\mathbf{k}}^t \end{cases}$$

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

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

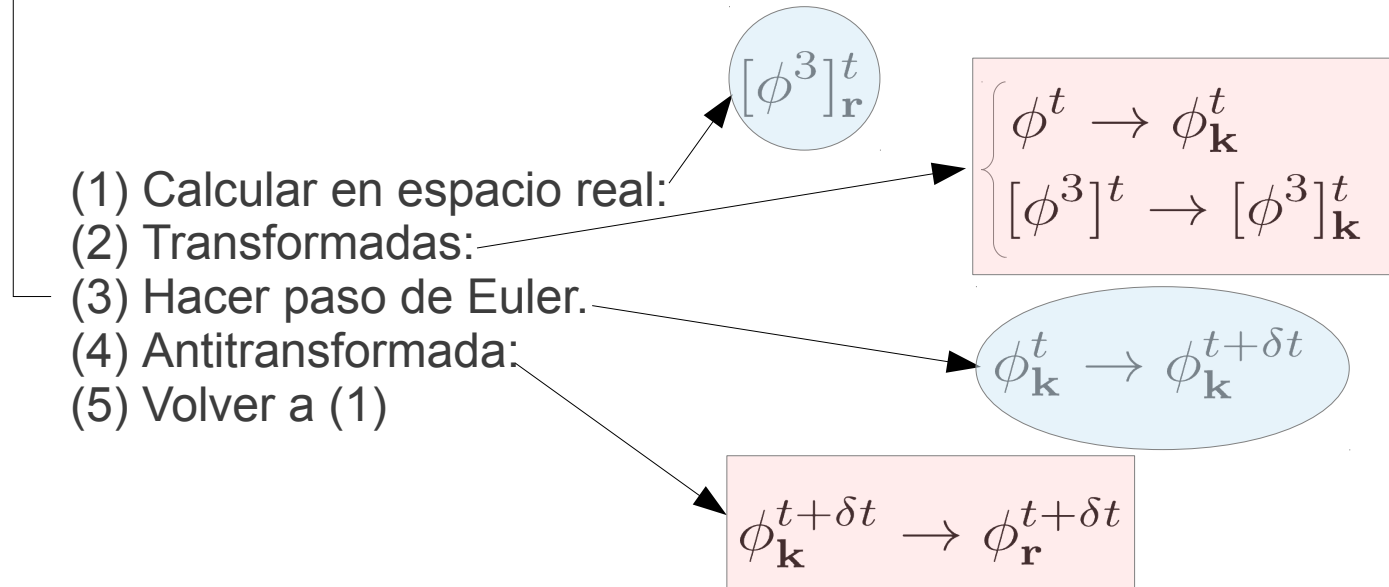
Trivialmente  
paralelizable

No trivialmente  
paralelizable (FFT)

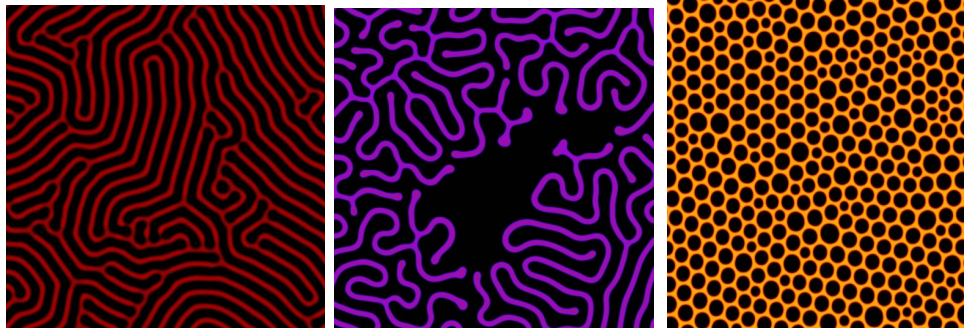
# Implementación

Update:

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

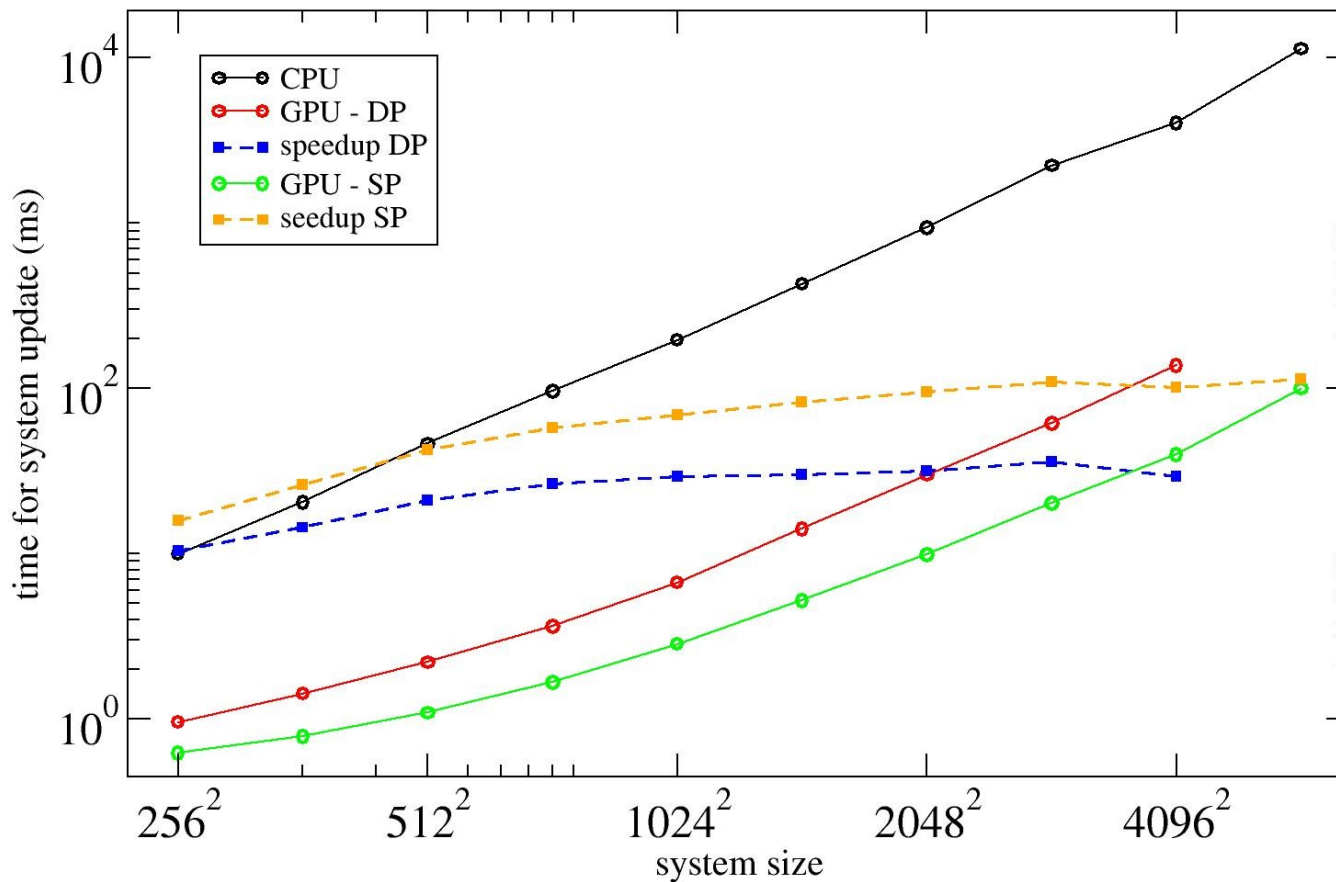


Galeria:



# 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

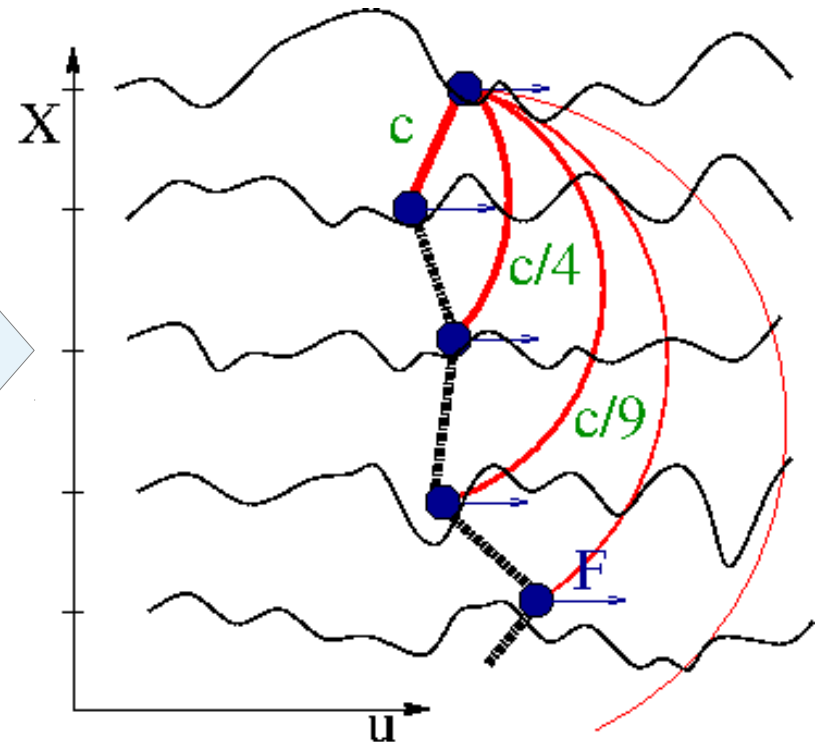
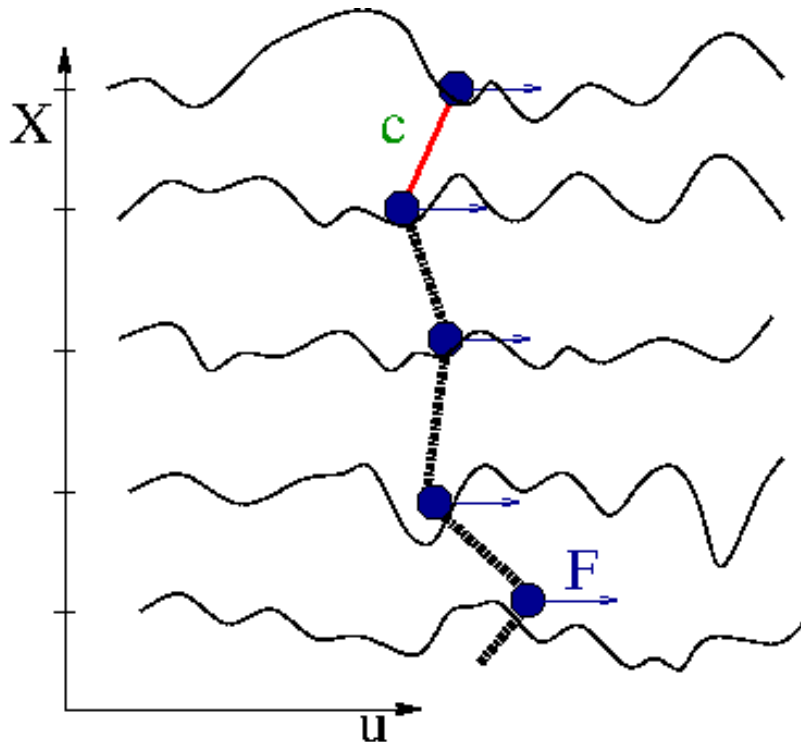


# La cuerda revisited...

$$\gamma \partial_t u(x, t) = \underbrace{c \partial_x^2 u(x, t)} + F_p(u, x) + f + \eta(x, t)$$

De local a no local

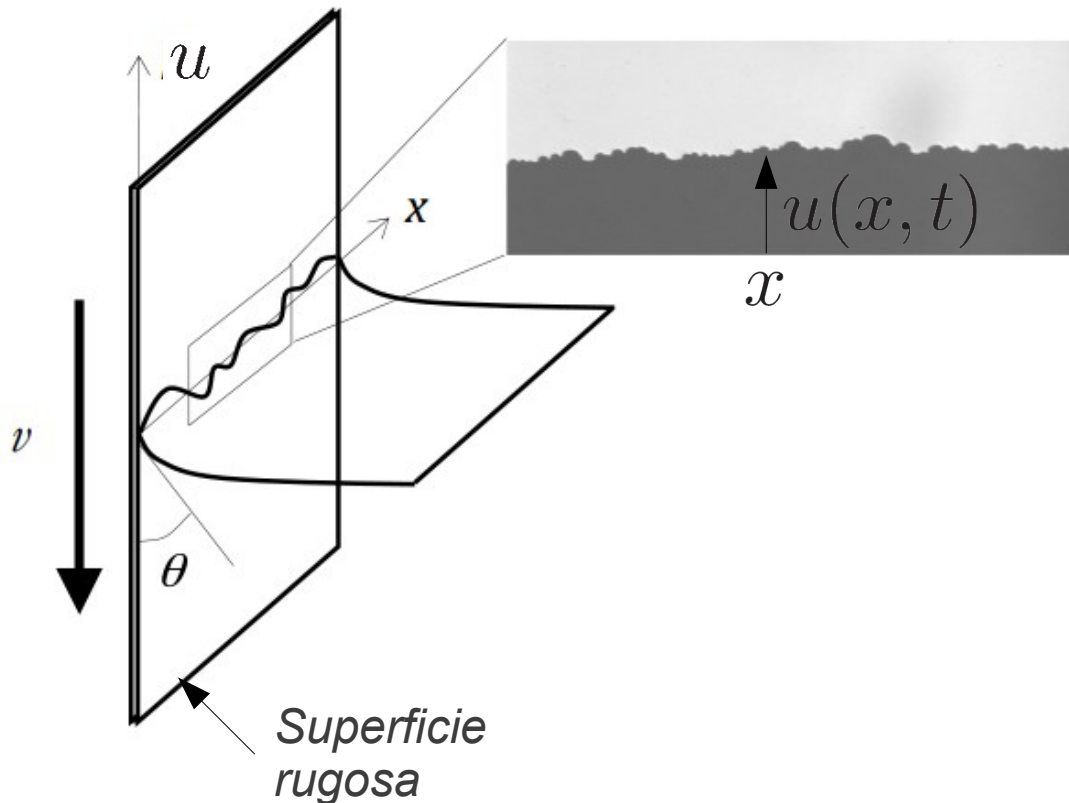
$$\gamma \partial_t u(x, t) = c \int \frac{u(x', t) - u(x, t)}{|x - x'|^2} + F_p(u, x) + f + \eta(x, t)$$



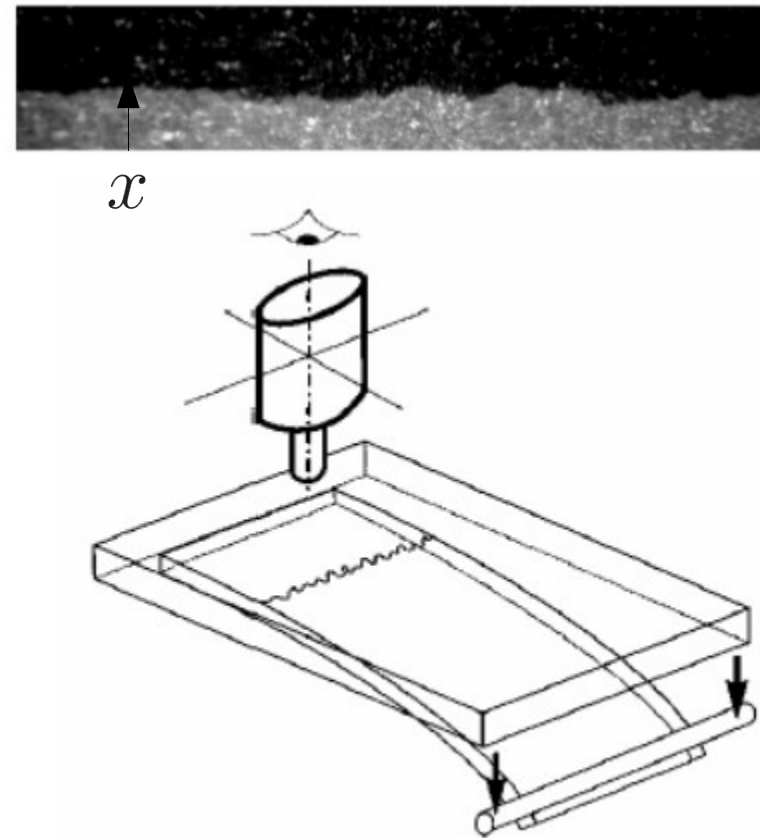
# Interacciones no locales en la cuerda

$$\gamma \partial_t u(x, t) = c \int \frac{u(x', t) - u(x, t)}{|x - x'|^2} + F_p(u, x) + f + \eta(x, t)$$

## Lineas de contacto de mojado



## Fracturas



# Interacciones no locales en la cuerda I

$$\gamma \partial_t u(X, t) = c \sum_{X'} u(X', t) G(|X - X'|) + F_p(u, X) + f + \eta(X, t)$$

Y si lo escribimos en forma matricial?. Por ejemplo, si  $L=4$  con PBC tenemos:

$$\overleftrightarrow{G} = \begin{pmatrix} 0 & G(1) & G(2) & G(1) \\ G(1) & 0 & G(1) & G(2) \\ G(2) & G(1) & 0 & G(1) \\ G(1) & G(2) & G(1) & 0 \end{pmatrix} \quad \vec{u}(t) = \begin{pmatrix} u(0, t) \\ u(1, t) \\ u(2, t) \\ u(3, t) \end{pmatrix}$$

$$\Rightarrow \gamma \partial_t \vec{u}(t) = c \overleftrightarrow{G} \vec{u} + \vec{F}_p + \vec{f} + \vec{\eta}(t) \quad O(L^2)$$

La matriz  $G$  es **densa**  $\rightarrow$  producto matrix-vector son  $L^2$  operaciones.  
Las operaciones matriciales se pueden paralelizar por ejemplo con CUBLAS o **MAGMA**.

***Pero estamos desaprovechando todas las simetrías de  $G$  !!!  
Hay otra solución que va como  $O(L \log L)$ ....***

# Interacciones no locales en la cuerda II

Avanzo en el espacio Real, pero calculo la interacción en Fourier

$$\gamma \partial_t u(X, t) = c \underbrace{\sum_{X \neq X'} u(X', t) G(X - X')}_{\text{Fourier transform}} + F_p(u, X) + f + \eta(X, t)$$

$$N^{-1} \sum_q e^{i2\pi q X/L} \boxed{u_q(t) G_q}$$

$$u(X, t) \rightarrow u_q(t)$$

- (1) Transformar Fourier
- (2) Multiplicar transformadas
- (3) Antitransformar producto
- (4) Hacer paso de Euler en Real
- (5) Volver a (1)

- (1), (3) **cuFFT**.
- (2), (4) trivialmente paralelo.

$$O(L \log L)$$

# Interacciones no locales en la cuerda III

$$\gamma \partial_t u(X, t) = c \sum_{X'} u(X', t) G(X - X') + F_p(u, X) + f + \eta(X, t)$$

*Convolución*

Avanzo en FOURIER  
Calculo desorden en  
El espacio REAL

- Acopla las partículas
- Acopla los modos

$$\gamma \partial_t u_q(t) = cu_q(t)G_q + \sum_X^L e^{i2\pi qX/L} F_p(u(X, t), X) + f\delta_{q,0} + \eta_q(t)$$

- (1) Calcular en espacio real
- (2) Transformar
- (3) Hacer paso de Euler en Fourier
- (4) Antitransformar
- (5) Volver a (1)

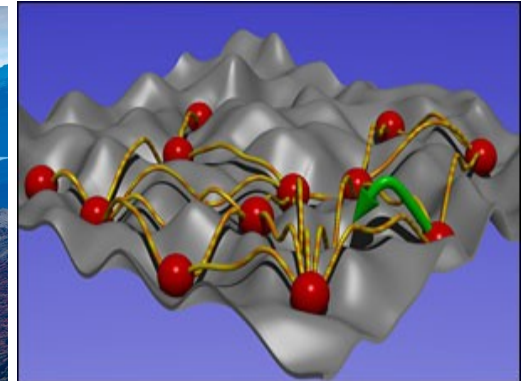
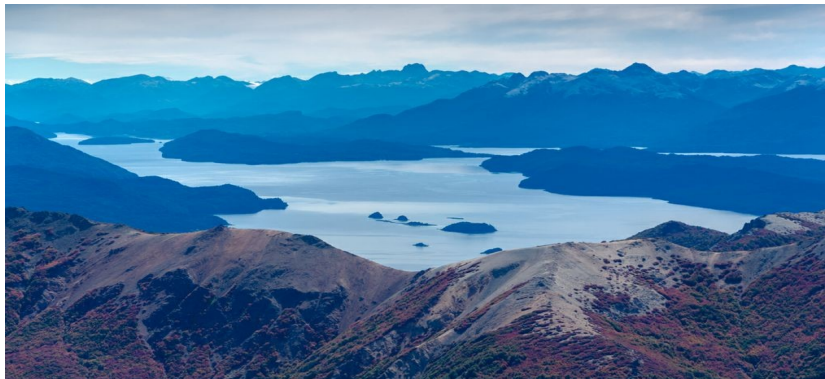
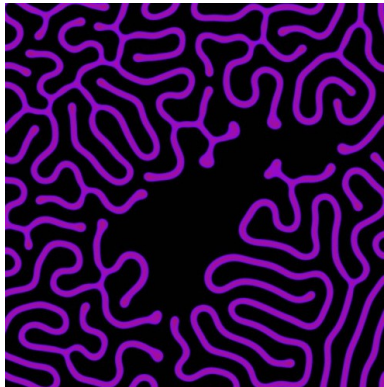
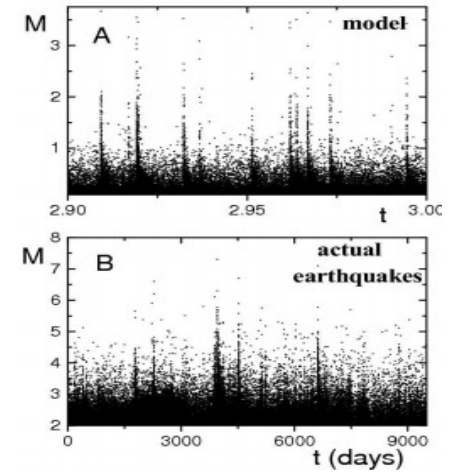
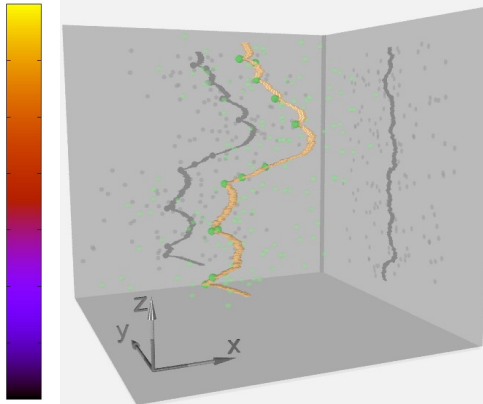
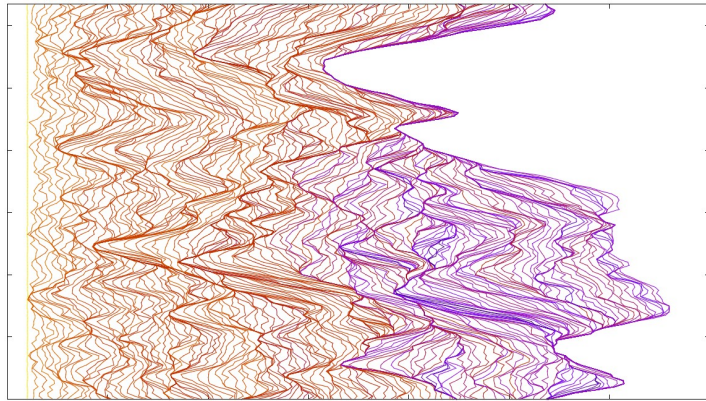
(1),(3) trivialmente paralelizable  
(2), (4) cuFFT.

$$u_q(t) \rightarrow u(X, t)$$

$O(L \log L)$

# Física Computacional con GPGPU

Con aplicación a la materia condensada



**Ofrezco temas de maestría/doctorado/postdoc**

Variedad de problemas físicamente interesantes y casi-embarazosamente paralelos) y no tanto) para ofrecer!!.

<http://fisica.cab.cnea.gov.ar/solidos/personales/kolton/>



# Manos a la obra...

- (1) **Un solo fuente:** `qew_minimal.cu`
- (2) **Compilar:** `make TAMANIO=16384`
- (3) **Dos ejecutables:** `qew_CUDA`, `qew_OMP`
- (4) **Ejecutar:** `sbatch ./cuda_job.sh`
- (5) **Mirar cola:** `squeue`
- (6) **Ir al nodo que corre:** `ssh mendietaxx`
- (7) **Mirar el nodo :** `htop` → `nvidia-smi` → `exit`
- (8) **Mirar el output:** donde esta el cuello de botella?
- (9) **OMP:** `sbatch ./omp_job` → `ssh mendietaxx` → `htop` → `exit`
- (10) **GPU vs CPU:** comparar outputs!

***Experimentar!!***

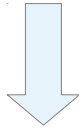
# Manos a la obra...

- **TODO** para principiantes:

## REDUCE

```
#include <thrust/reduce.h>
...
int data[6] = {1, 0, 2, 2, 1, 3};
int result = thrust::reduce(data, data + 6);

// result == 9
```



$$v(t) = \sum_{X=0}^{L-1} F_{tot}(X, t) / L$$

$$u_{cm}(t) = \sum_{X=0}^{L-1} u(X, t) / L$$

## TRANSFORM\_REDUCE

```
...
int data[6] = {-1, 0, -2, -2, 1, -3};
int result = thrust::transform_reduce(data, data + 6,
                                     absolute_value<int>(),
                                     0,
                                     thrust::maximum<int>());

// result == 3
```



$$w^2(t) = \sum_{X=0}^{L-1} [u(X, t) - u_{cm}(t)]^2 / L$$

- **FIXME** para agregar fluctuaciones térmicas

# Manos a la obra...

- **TODO** para los avanzados :-)
- Optimización:  
profiling → cuellos de botella, interpretación ?
- Visualización (placa conectada a monitor): OpenGL?
- *Criticas, comentarios, sugerencias, bienvenidas!!!.*

# Manos a la obra...

- (1) **Un solo fuente:** `simple_cufft_thrust.cu`
- (2) **Compilar:** `make TAMANIO=4194304`
- (3) **Ejecutables:** `simple_cufft` (`simple_fftw`, `simple_fftw_threads`).
- (4) **Ejecutar:** `sbatch ./cuda_job.sh`
- (5) **Resolver los TODO**
- (6) **Analizar el output, discutir...**

*Experimentar!!*

# Ejercicio

```
for(i=0;i<N;i++) d3[i] = d1[i]*d2[i];
```

# En PyCUDA

```
for(i=0;i<N;i++) d3[i] = d1[i]*d2[i];
```

- In [40]: n=10000000
- In [41]: b\_cpu=np.random.randn(n).astype('float32')
- In [42]: a\_cpu=np.random.randn(n).astype('float32')
- In [43]: b\_gpu=curand(n)
- In [44]: a\_gpu=curand(n)
- In [45]: %timeit a\_gpu\*b\_gpu  
1000 loops, best of 3: 1.57 ms per loop



# En Thrust?

```
for(i=0;i<N;i++) d3[i] = d1[i]*d2[i];
```

- `cp -a /home/akolton/ejemplo_thrust .`
- Editar template: `main.cu`
- Mirar Makefile: que hace?.
- `make`
- A explorar:
  - 1) Editar `cuda_job.sh` → `sbatch ./cuda_job.sh;`
  - 2) Editar `omp_job.sh` → `sbatch ./omp_job.sh;`

# Peformance

```
for(i=0;i<N;i++) d3[i] = d1[i]*d2[i];
```

