

ICNPG 2023

Clase 10: Números aleatorios



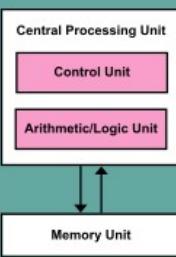
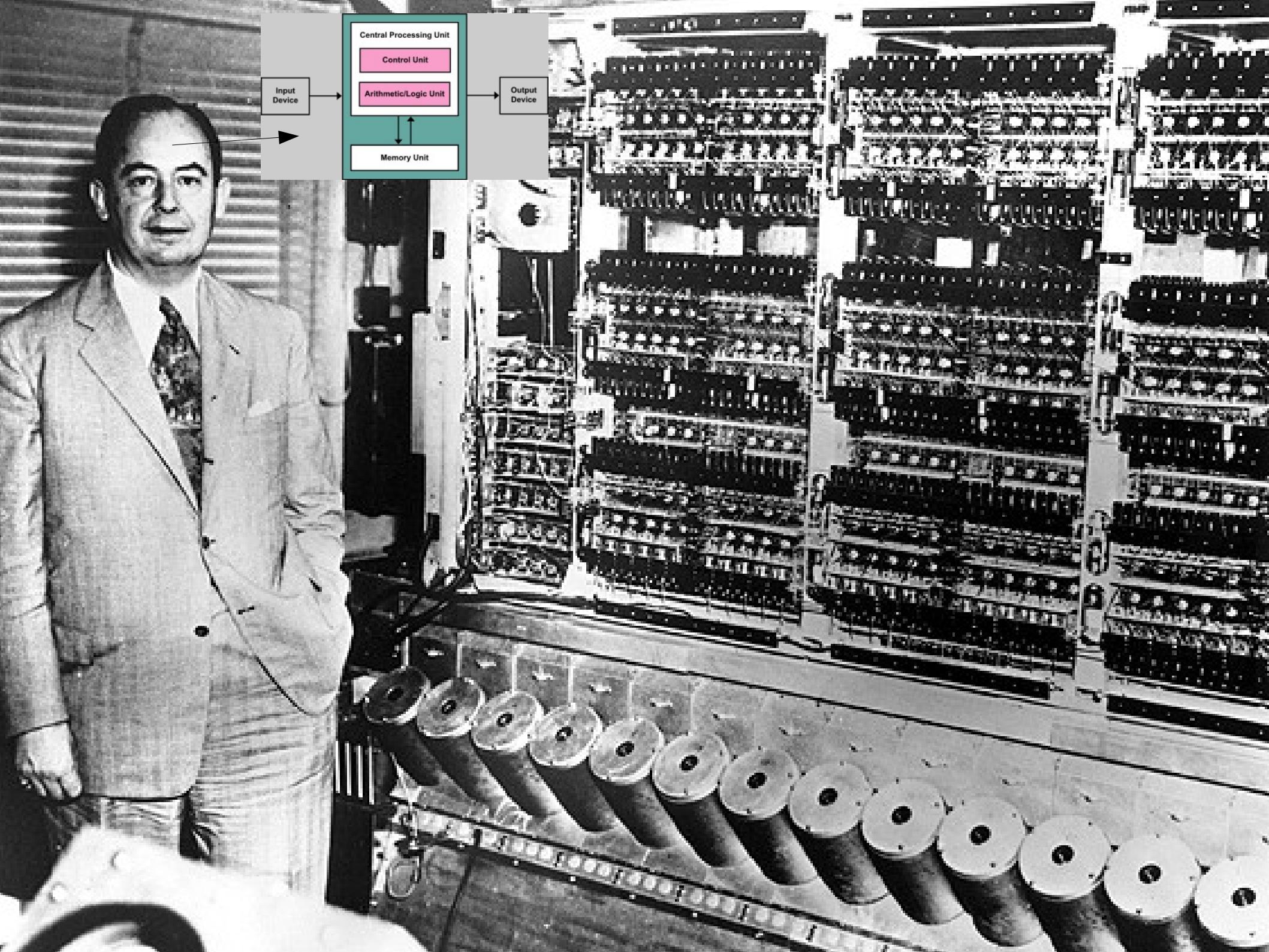
Generar números aleatorios en paralelo

"anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin"

John von Neumann



John von Neumann (registrado al nacer como Neumann János Lajos; [Budapest, Imperio austrohúngaro, 28 de diciembre de 1903](#)-[Washington, D. C., Estados Unidos, 8 de febrero de 1957](#)) fue un [matemático húngaro-estadounidense](#) que realizó contribuciones fundamentales en [física cuántica](#), [análisis funcional](#), [teoría de conjuntos](#), [teoría de juegos](#), [ciencias de la computación](#), [economía](#), [análisis numérico](#), [cibernética](#), [hidrodinámica](#), [estadística](#) y muchos otros campos. Es considerado como uno de los más importantes matemáticos de la historia moderna.



Input Device

Output Device

Aplicaciones

- **Métodos de Monte Carlo:**
 - Se basan en el “*muestreo aleatorio*” para obtener resultados numéricos estadísticos, cuando otros métodos se vuelven muy muy difíciles de usar.
 - **Ciencias Naturales:** Cálculo de propiedades de sistemas interactuantes con muchos grados de libertad (Física, Química, Biología, etc, etc)
 - **Matemática:** optimización (*simulated annealing*), integración multidimensional, muestreo de distribuciones de probabilidad complicadas, finanzas, inteligencia artificial, un montón de etc.
- **Ecuaciones diferenciales estocásticas:**
 - **Ciencias Naturales:** Sistemas con ruido, típicamente térmico, movimiento browniano de partículas mesoscópicas, ruido en circuitos, etc.
 - **Matemática:** Ecuaciones diferenciales estocásticas, etc.
- **Etc estocástico.**

Todos necesitan generar una gran cantidad de números aleatorios

Monte Carlo

- **Direct Monte Carlo:**
 - Modelado con números aleatorios
 - Muy simple de implementar
- **Monte Carlo integration**
 - Cálculo de integrales con números random
 - En particular, útil para integrales multidimensionales
- **Metropolis Monte Carlo**
 - Se genera una cadena de Markov que converge a números sampleados con una distribución deseada (ejemplo: Boltzmann, o la estacionaria que desee)

Integración con Monte Carlo

$$I = \int_a^b dx f(x) \approx \frac{b-a}{N} \sum_{i=1}^N f(x_i) = I'$$

$$\sigma^2 \equiv (\overline{I'^2} - \overline{I'}^2) = \frac{(b-a)^2}{N} (\overline{f^2} - \overline{f}^2)$$

Integración con Monte Carlo

Para baja dimensión conviene usar métodos tradicionales

$$I = \int_{\Omega} f(\bar{\mathbf{x}}) d\bar{\mathbf{x}} \quad V = \int_{\Omega} d\bar{\mathbf{x}}$$

$$I \approx Q_N \equiv V \frac{1}{N} \sum_{i=1}^N f(\bar{\mathbf{x}}_i) = V \langle f \rangle \quad \lim_{N \rightarrow \infty} Q_N = I$$

$$\text{Var}(f) \equiv \sigma_N^2 = \frac{1}{N-1} \sum_{i=1}^N (f(\bar{\mathbf{x}}_i) - \langle f \rangle)^2.$$

$$\text{Var}(Q_N) = \frac{V^2}{N^2} \sum_{i=1}^N \text{Var}(f) = V^2 \frac{\text{Var}(f)}{N} = V^2 \frac{\sigma_N^2}{N}$$

Tiende a 0 si Var(f) acotada...
o sea si f no tiene picos o divergencias

Este resultado NO depende del número de dimensiones de la integral.

Esa es la gran ventaja de la integración de Monte Carlo contra casi todos los métodos deterministas que dependen *exponencialmente* con la dimensión.

https://en.wikipedia.org/wiki/Monte_Carlo_integration

Calcular Pi ...

Año	Matemático o documento	Cultura	Aproximación	Error (en partes por millón)
~1900 a. C.	Papiro de Ahmes	Egipcia	$2^8/3^4 \sim 3,1605$	6016 ppm
~1600 a. C.	Tablilla de Susa	Babilónica	$25/8 = 3,125$	5282 ppm
~600 a. C.	La Biblia (Reyes I, 7,23)	Judía	3	45 070 ppm
~500 a. C.	Bandhayana	India	3,09	16 422 ppm
~250 a. C.	Arquímedes de Siracusa	Griega	entre $3\frac{10}{71}$ y $3\frac{1}{7}$ empleó $211875/67441 \sim 3,14163$	<402 ppm 13,45 ppm
~150	Claudio Ptolomeo	Greco-egipcia	$377/120 = 3,141666\dots$	23,56 ppm
263	Liu Hui	China	3,14159	0,84 ppm
263	Wang Fan	China	$157/50 = 3,14$	507 ppm
~300	Chang Hong	China	$10^{1/2} \sim 3,1623$	6584 ppm
~500	Zu Chongzhi	China	entre 3,1415926 y 3,1415929 empleó $355/113 \sim 3,1415929$	<0,078 ppm 0,085 ppm
~500	Aryabhata	India	3,1416	2,34 ppm
~600	Brahmagupta	India	$10^{1/2} \sim 3,1623$	6584 ppm
~800	Al-Juarismi	Persa	3,1416	2,34 ppm
1220	Fibonacci	Italiana	3,141818	72,73 ppm
1400	Madhava	India	3,14159265359	0,085 ppm
1424	Al-Kashi	Persa	$2\pi = 6,2831853071795865$	0,1 ppm

Calcular Pi ...

Año	Descubridor	Ordenador utilizado	Número de cifras decimales
1949	G.W. Reitwiesner y otros ¹⁵	ENIAC	2037
1954		NORAC	3092
1959	Guilloud	IBM 704	16 167
1967		CDC 6600	500 000
1973	Guillord y Bouyer ¹⁵	CDC 7600	1 001 250
1981	Miyoshi y Kanada ¹⁵	FACOM M-200	2 000 036
1982	Guilloud		2 000 050
1986	Bailey	CRAY-2	29 360 111
1986	Kanada y Tamura ¹⁵	HITAC S-810/20	67 108 839
1987	Kanada, Tamura, Kobo y otros	NEC SX-2	134 217 700
1988	Kanada y Tamura	Hitachi S-820	201 326 000
1989	Hermanos Chudnovsky	CRAY-2 y IBM-3090/VF	480 000 000
1989	Hermanos Chudnovsky	IBM 3090	1 011 196 691
1991	Hermanos Chudnovsky		2 260 000 000
1994	Hermanos Chudnovsky		4 044 000 000
1995	Kanada y Takahashi	HITAC S-3800/480	6 442 450 000
1997	Kanada y Takahashi	Hitachi SR2201	51 539 600 000
1999	Kanada y Takahashi	Hitachi SR8000	68 719 470 000
1999	Kanada y Takahashi	Hitachi SR8000	206 158 430 000
2002	Kanada y otros ¹⁵ [3]	Hitachi SR8000/MP	1 241 100 000 000
2004		Hitachi	1 351 100 000 000
2009	Daisuke Takahashi ¹⁶	T2K Tsukuba System	2 576 980 370 000
2009	Fabrice Bellard ¹⁷	Core i7 CPU, 2.93 GHz; RAM: 6GiB	2 699 999 990 000
2010	Shigeru Kondo	2 x Intel Xeon X5680, 3.33 GHz	5 000 000 000 000
2011	Shigeru Kondo		10 000 000 000 000

3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253421170679821480865
23066470938446095505822317253594081284811174502841027019385211055596446229489549303819644288109756659334461284
23378678316527120190914564856692346034861045432664821339360726024914127372458700660631558817488152092096282925
15364367892590360011330530548820466521384146951941511609433057270365759591953092186117381932611793105118548074
99627495673518857527248912279381830119491298336733624406566430860213949463952247371907021798609437027705392171
17675238467481846766940513200056812714526356082778577134275778960917363717872146844090122495343014654958537105
79689258923542019956112129021960864034418159813629774771309960518707211349999998372978049951059731732816096318
44594553469083026425223082533446850352619311881710100031378387528865875332083814206171776691473035982534904287
73115956286388235378759375195778185778053217122680661300192787661119590921642019893809525720106548586327886593
81827968230301952035301852968995773622599413891249721775283479131515574857242454150695950829533116861727855889
83817546374649393192550604009277016711390098488240128583616035637076601047101819429555961989467678374494482553
72684710404753464620804668425906949129331367702898915210475216205696602405803815019351125338243003558764024749
63914199272604269922796782354781636009341721641219924586315030286182974555706749838505494588586926995690927210
9302955321165344987202755960236480665499119881834797753566369807426542527862551818417574672890977727938000816
0161452491921732172147235014144197356854816136115735255213347574184946843852332390739414333454776241686251898
85562099219222184272550254256887671790494601653466804988627232791786085784383827967976681454100953883786360950
42251252051173929848960841284886269456042419652850222106611863067442786220391949450471237137869609563643719172
7646575739624138908658326459581339047802759009946576407895126946839835259570982582262052248940772671947826848
76990902640136394437455305068203496252451749399651431429809190659250937221696461515709858387410597885959772975
01617539284681382686838689427741559918559252459539594310499725246808459872736446958486538367362226260991246080
88439045124413654976278079771569143599770012961608944169486855534840635342207222582848864815845602850601684273
67467678895252138522549954666727823986456596116354886230577456498035593634568174324112515076069479451096596094
88797108931456691368672287489405601015033086179286809208747609178249385890097149096759852613655497818931297848
9989487226588048575640142704775513237964145152374623436454285844479526586782105114135473573952311342716610213
36231442952484937187110145765403590279934403742007310578539062198387447808478489683321445713868751943506430218
10484810053706146806749192781911979399520614196634287544406437451237181921799983910159195618146751426912397489
18649423196156794520809514655022523160388193014209376213785595663893778708303906979207734672218256259966150142
68038447734549202605414665925201497442850732518666002132434088190710486331734649651453905796268561005508106658
81635747363840525714591028970641401109712062804390397595156771577004203378699360072305587631763594218731251471
92819182618612586732157919841484882916447060957527069572209175671167229109816909152801735067127485832228718352
96572512108357915136988209144421006751033467110314126711136990865851639831501970165151168517143765761835155650
99898599823873455283316355076479185358932261854896321329330898570642046752590709154814165498594616371802709819
92448895757128289059232332609729971208443357326548938239119325974636673058360414281388303203824903758985243744
13276561809377344403070746921120191302033038019762110110044929321516084244485963766983895228684783123552658213
768572624334418930396864262431077322697802807318915441101044682325271620105265227211660396665573092547110557
34668206531098965269186205647693125705863566201855810072936065987648611791045334885034611365768675324944166803
79787718556084552965412665408530614344431858676975145661406800700237877659134401712749470420562230538994561314
70004078547332699390814546646458807972708266830634328587856983052358089330657574067954571637752542021149557615
25012622859413021647155097925923099079654737612551765675135751782966645477917450112996148903046399471329621073
51895735961458901938971311179042978285647503203198691514028708085990480109412147221317947647772622414254854540

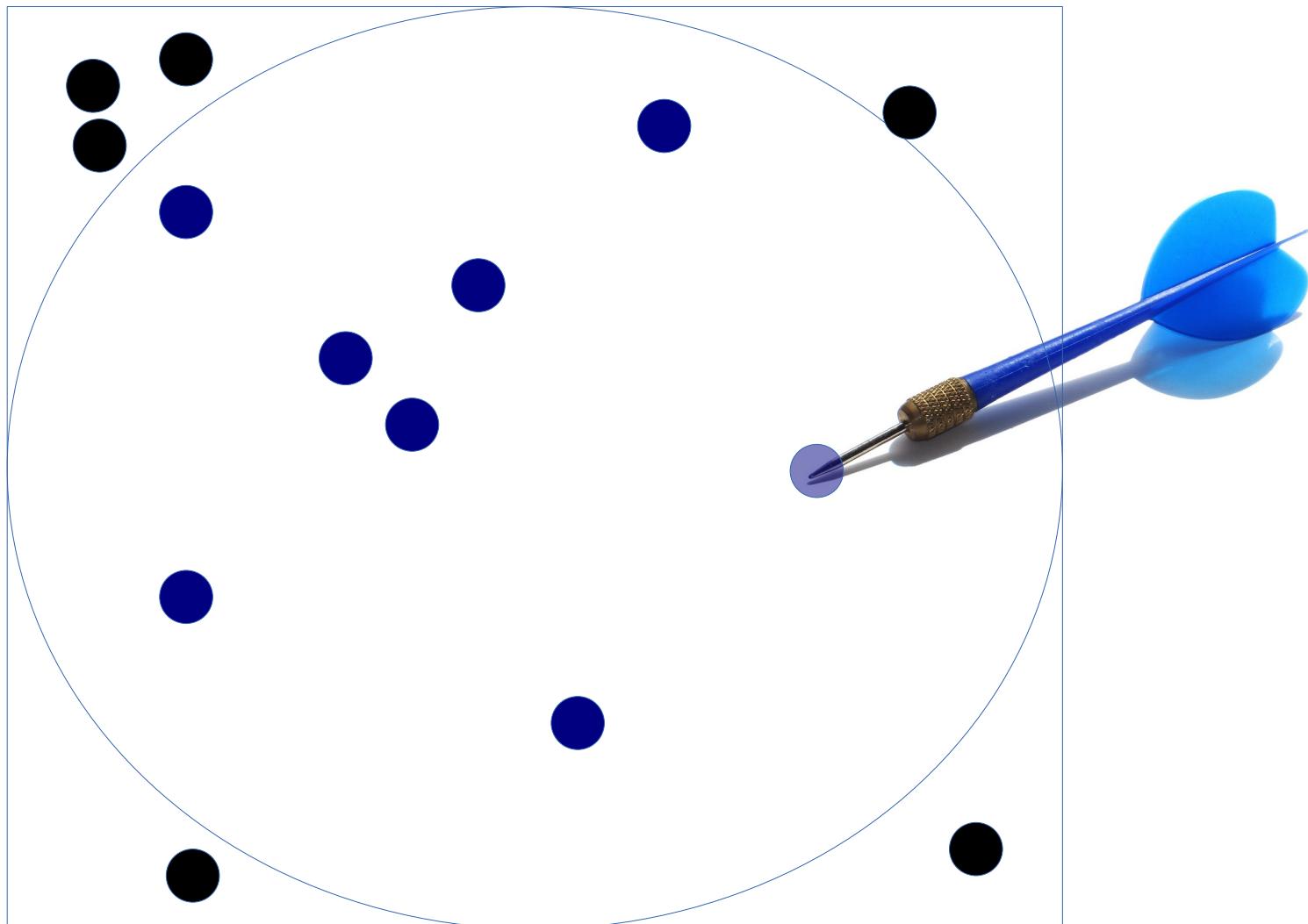
Calculando

$$\pi$$

usando números
aleatorios?

Calculando Pi usando números aleatorios

Ejemplo básico de generador de nros aleatorios

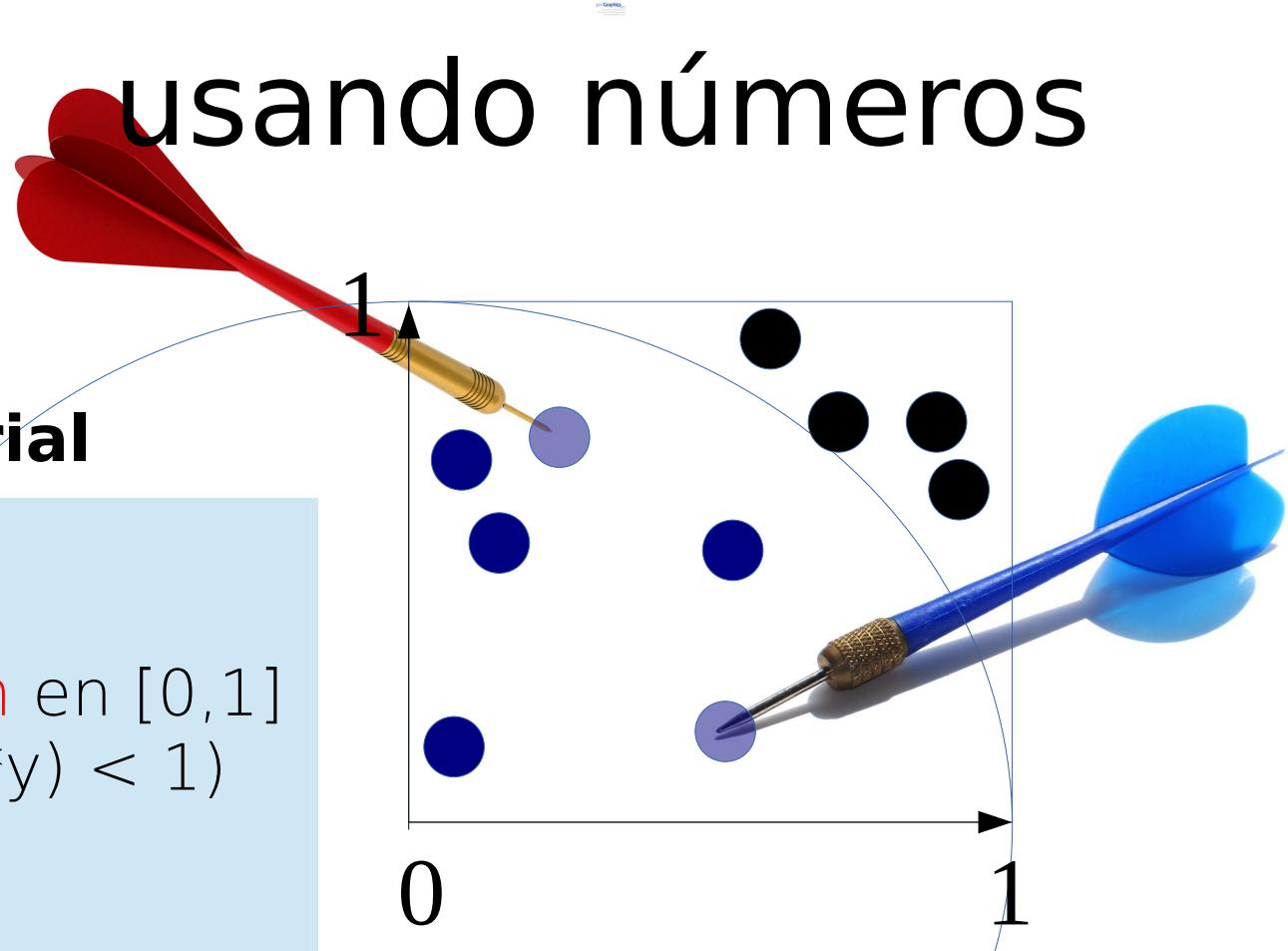


$$4 \text{ Azules}/(\text{Negros} + \text{Azules}) \rightarrow ?$$

Calculando π usando números aleatorios

Versión Serial

```
while(Totales < N)
{
    x, y números random en [0,1]
    Azules += ((x*x + y*y) < 1)
    Totales++;
}
Pi = 4*Azules/Totales
```



¿ Que cuidados hay que tener con los RNs ?

Hay que cuidar que la distribución sea uniforme y estén descorrelacionados entre sí. Por ejemplo, que la sucesión de nros aleatorios no sea similar a una trayectoria. Hay una serie de test que se puede hacer a los generadores para ver que cumplan esa condición

¿ Versión Paralela ?

Se tiran muchos RN al mismo tiempo o muchos generadores de RN y luego hacemos el promedio de los resultados parciales

¿ Porque conviene ?

¿ Que cuidados hay que tener ?

$4 \text{ Azules}/(\text{Negros} + \text{Azules}) \rightarrow \pi$

Generar números (pseudo)aleatorios

- Típicamente **necesitamos muchísimos RN**, y queremos garantizar que sean de “buena calidad”, que sean lo más “random” posible.
- Varios de los algoritmos desarrollados para lograr buenos RN son **intrínsecamente secuenciales**; o sea, malos para GPUs

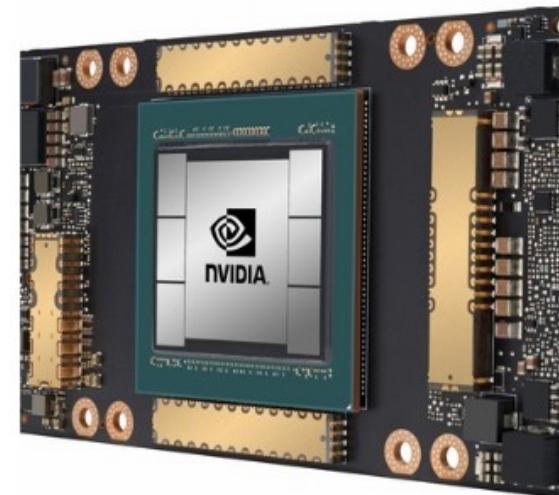
$$x_{n+1} = f(x_n) \qquad \qquad x_0 = \text{“seed”}$$

- Por lo tanto es/fue necesario revisar el asunto desde la raíz.

¿Cómo implementar el uso de RNGs en GPUs ?

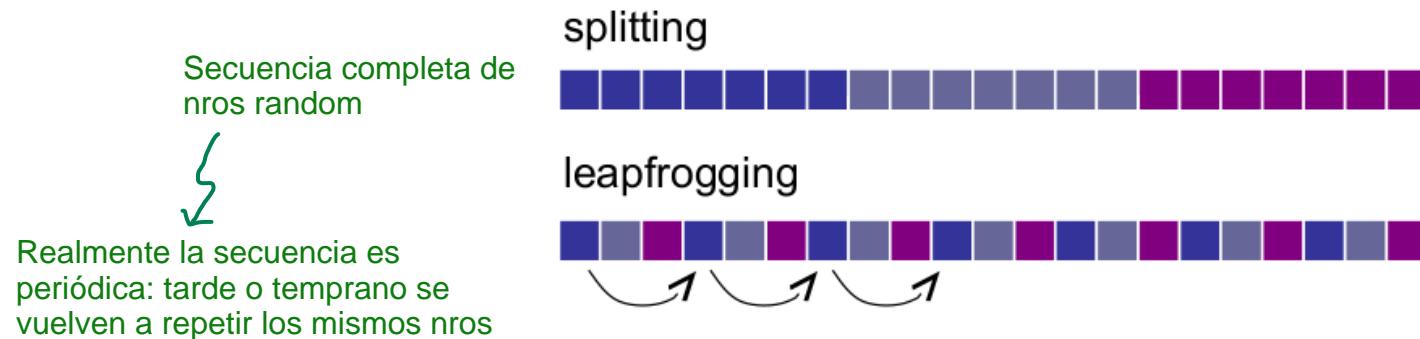
- NO es una buena estrategia: utilizar un generador "clásico" centralizado en CPU y distribuir RN a distintos threads GPU.
Esto mataría la performance
- Cada thread debe generar RN, con una instancia distinta de RNG.
- Se busca en lo posible :
 - a) preparar cientos de miles de RNGs independientes que brinden secuencias descorrelacionadas.
 - b) minimizar el uso y transferencia de memoria relacionados a la producción de números aleatorios.

```
97540038 89058924 11203660 67110393 77180775 31157173 48605971  
95061909 02022164 77739874 60984146 61119174 53359697 88138466  
39966630 19951776 16391156 68348039 64379956 15695047 09287812  
77226413 51166939 76904118 63621137 75978018 29045161 49974612  
81376873 35452092 01990879 19154686 82330997 28955302 95237399  
43476489 23762475 35131499 89077156 06201281 10501824 33154167  
19711389 51111229 32502896 81587097 79880877 33104567 43560122  
85081683 16387085 45406746 91688455 39721974 22863449 56098514  
46911949 90543945 22590287 39370906 58369744 42443874 51263312  
13244456 48155008 42027649 36680044 79378289 43686380 55118412  
45224219 15140367 76410153 85598940 01270982 62095985 72861374  
96056059 55507069 44225551 59745610 91267949 00841967 13989135  
35485028 36796450 21058269 11742350 90294671 67111700 71896354  
90323639 06852575 56194161 34194684 89126309 44456416 35370866  
45942911 11826307 52171943 76551463 12087611 05666438 97948548  
81896875 94827442 56369439 51699019 81150789 15709381 33632281  
42903681 49946891 82756006 74687826 24772707 59051979 57986317
```



Estrategias para la generación de RN en paralelo

- ➊ División de la secuencia de un RNG con un largo período en *subsecuencias* no superpuestas, producidas por **diferentes threads**.



- ➋ Utilización de un único RNG de período muy grande y **semillado diferente para cada thread**, con superposiciones de secuencias descorrelacionadas.
- ➌ Preparación de **generadores independientes** para cada thread (de una misma 'clase' de RNGs) utilizando distintos lags, multiplicadores, shifts, etc.

Limitaciones y opciones

- Necesidad de **minimizar transferencia de memoria**, permitir que el estado del RNG permanezca en registros o en shared.

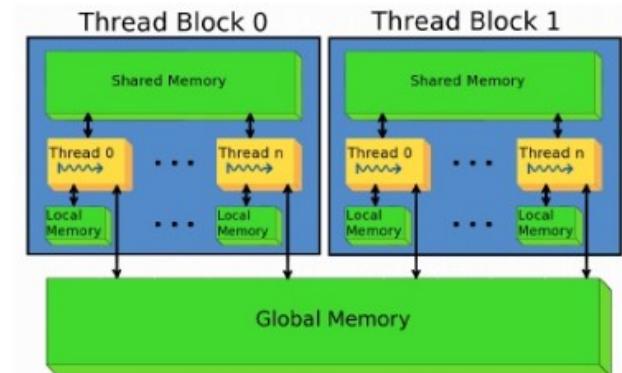
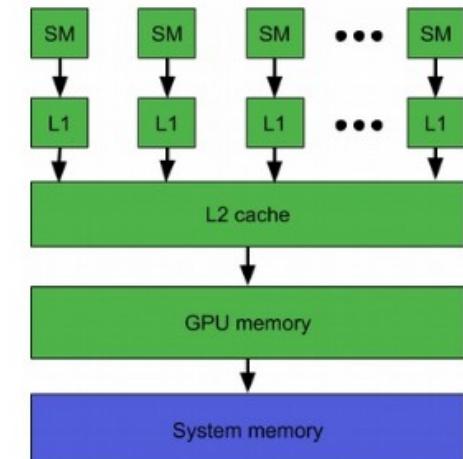
- *Not the case* para RNGs muy usados en CPUs: Mersenne Twister (MT19937) tiene un estado de 624 palabras (~20kB), que debe compararse con los 48kB de shared memory disponibles en un SM de Fermi.

Este es el mejor (o uno de los mejores) RNG. Pasa todos los tests

- Dos estrategias posibles:

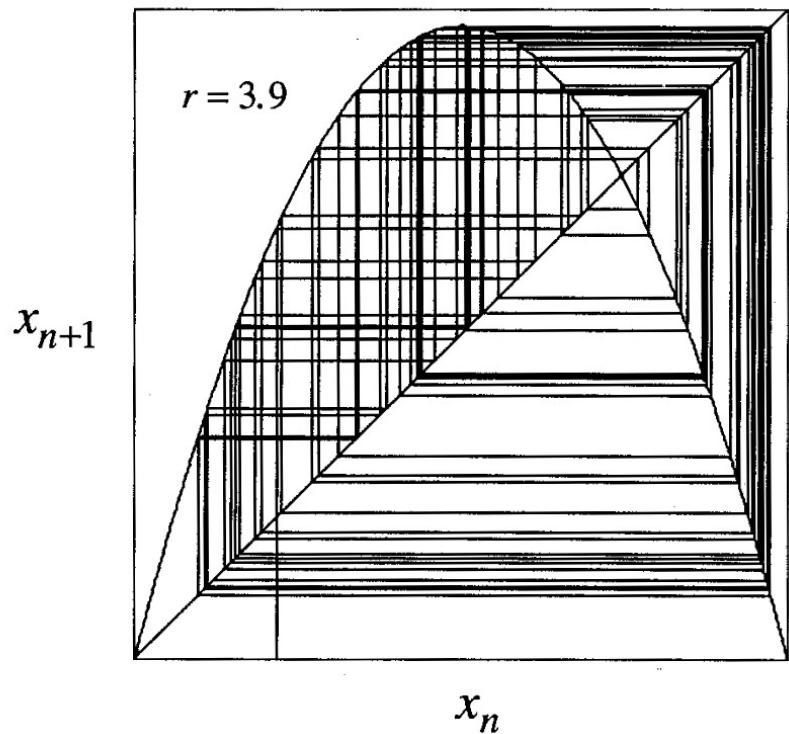
- Buscar generadores con **estado pequeño** y de buena calidad.
- Compartir el **estado grande** de un RNG de gran período entre los threads de **un mismo bloque** y que estos colaboren para **producir varios RN en una llamada vectorizada**.

Hay generadores malos que generan nros rapidísimos y generadores buenos que generan nros lentísimo



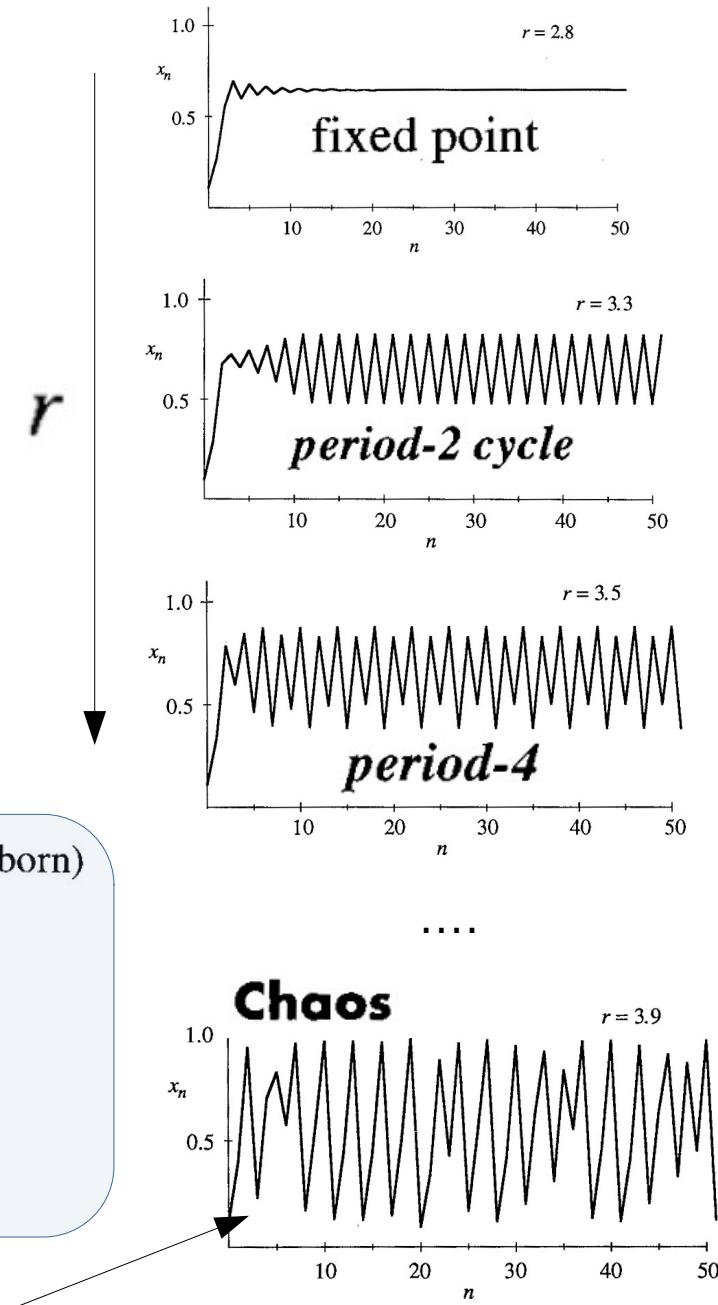
Mapeo logístico

$$x_{n+1} = rx_n(1 - x_n), \quad 0 \leq r \leq 4$$



Si bien genera nros random, es demasiado simple

$r_1 = 3$	(period 2 is born)
$r_2 = 3.449\dots$	4
$r_3 = 3.54409\dots$	8
$r_4 = 3.5644\dots$	16
$r_5 = 3.568759\dots$	32
\vdots	\vdots
$r_\infty = 3.569946\dots$	∞



Generador de números pseudo-aleatorios para ciertos r

Sí se usa:

Linear Congruential Generators (LCG)

$$x_{n+1} = ax_n + c \pmod{m}.$$

Variante	Período	Estado
32-bits ($m=2^{32}$) $a = 1664525; c = 1013904223$	2^{32}	(x_n) 32 bits
64-bits ($m=2^{64}$) $a = 2862933555777941757; c = 1442695040888963407$	2^{64}	(x_n) 64 bits

Para obtener un número en $[0,1]$ hacemos simplemente $u_n = x_n / m$

{Ventaja para implementar en paralelo: fácilmente se salta en la secuencia

$$x_{n+t} = a_t x_n + c_t \pmod{m}, \quad a_t = a^t \pmod{m}, \quad c_t = \sum_{i=1}^t a^i c \pmod{m}.$$

- Eligiendo t igual al número de threads, todos pueden generar números de la misma secuencia global (período de la sub-secuencia: p / t).
- Otra opción, sin garantía de independencia de las sub-secuencias, es largar distintas semillas x_0 (RN sorteados previamente) en cada hilo: “LGC-random”

Multiply With Carry (MWC)

$$\begin{aligned}x_{n+1} &= ax_n + c_n \pmod{m}, \\c_{n+1} &= \lfloor (ax_n + c_n)/m \rfloor.\end{aligned}$$

A handwritten multiplication example on lined paper. It shows 325 being multiplied by 6. The multiplication is set up as follows:

$$\begin{array}{r} 325 \\ \times 6 \\ \hline 1950 \end{array}$$

Variante a los LCG manteniendo el tamaño del estado. c_n en el paso $(n+1)$ es el "carry" de la iteración previa.

Variante	Período	Estado
32-bits ($m=2^{32}$)	$am-2 (\sim 2^{64})$	(x_n, c_n) 64 bits

- Para alcanzar un buen período se necesita que $(am-1)$ y $(am-2)/2$ sean primos. Se dice que $am-1$ es un *safeprime*.
- Eligiendo distintos a que cumplan esa condición podemos producir un gran número de secuencias independientes del RNG de período y calidad comparables.
- ~ 150000 secuencias independientes de período mayor a $1,797 \times 10^{19}$ con $a=4294967118$
- Se carga un array de *safeprimes* y cada thread usa una entrada distinta

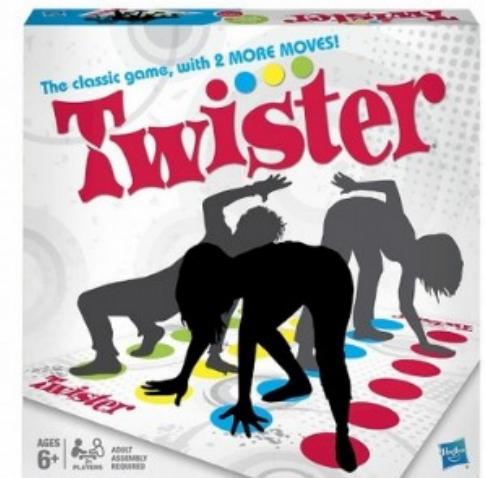
Este es el mejor que se puede usar en CPU

Mersenne Twister (MT)

- Basado en una variante del concepto del Lagged Fibonacci a una forma de feedback generalizado de shift de registros, con recursión:

$$x_n = (x_{n-N}|x_{n-N+1})A \oplus x_{n-N+M}.$$

- $N=[k/32]$, donde 2^k-1 es el "primo Mersenne"; $1 < M < N$ es el intervalo adicional o tamaño del estado y $(x_{n-N}|x_{n-N+1})$ denota una concatenación de cadenas de bits; la matriz A de 32×32 define la **operación de twist**.
- Finalmente, antes del output, se agrega una transformación de "tempering" $u_n = \lfloor x_n T \rfloor$,



Variante	Período	Estado
MT19937	$2^{19937}-1$	624x32 bits
MT607 (CUDA SDK)	$2^{607}-1$	19x32 bits
MTGP (cuRAND 4.1) vectorizado en 256 threads	$2^{11213}-1$	351x32 bits /256

RNGs para aplicaciones GPGPU

Las bibliotecas (que vimos) usan algunos de estos

LCG Son muy rápidos y de estado pequeño. Usar el LCG64 (período aceptable) semillando at random, a conciencia.

MWC Rápido y de período aceptable, estado pequeño. Permite secuencias independientes.

Fibonacci Necesidad de compartir estado entre threads de un mismo bloque y subdividir secuencia. 2x más lento que MWC. Usar el de estado más grande ($r=1279$).

MT No recomendable para GPGPU. De muy buena calidad y período astronómico, pero no es tan rápido y su estado es demasiado grande.

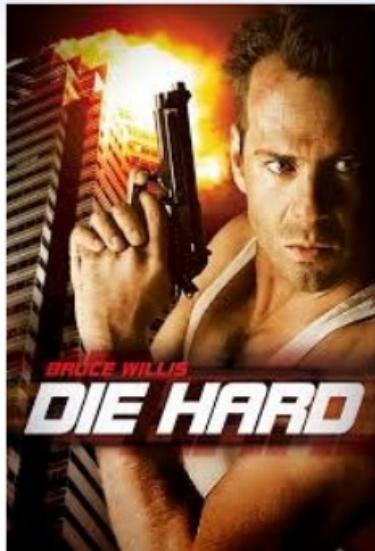
XORWOW Muy buen período. Lento. Falla el test de Ising

XorShift / Weyl Colaboración entre threads de un mismo bloque. Muy buen período y calidad. Lento.

Philox-4x Buen período. Buena calidad. No guarda estado. Uso de memoria justo. Rápido. Permite muchísimas secuencias independientes.



Vamos a usar este



Tests de randomicidad



- Muy malos RNGs han sido incorporados a librerías standard en los comienzos de las simulaciones en computadoras....y aún hoy.
- Surgen colecciones ("baterías") de tests para valorar propiedades estadísticas de secuencias de números pseudo-aleatorios.
- La suite **DIEHARD** de Marsaglia fue durante muchos años el standard en el área. Luego se reemplazó por la suite **TestU01** de L'Ecuyer con las baterías **SmallCrush**, **Crush** y **BigCrush**.
- Aún cuando un RNG pasa todos los test **no hay garantía** de que no se produzcan desviaciones sistemáticas en una simulación de Monte Carlo.

Random number generators for massively parallel simulations on GPU

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

Cuántos bits se usan por hilos.
Cuanto menos mejor por la poca
capacidad de memoria de los hilos

Cuántos nros se
generan por segundo

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, $r = 521$	≥ 80	None	2	—	failed	23
Fibonacci, $r = 1279$	≥ 80	None	(1)	2	passed	23
XORWOW (cuRAND)	192	None	None	1/3	failed	19
MTGP (cuRAND)	≥ 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?

cuRAND

Random Number Generation on NVIDIA GPUs

[DOWNLOAD >](#) [DOCUMENTATION >](#) [SAMPLES >](#) [SUPPORT >](#) [FEEDBACK >](#)

The NVIDIA CUDA Random Number Generation library [cuRAND] delivers high performance GPU-accelerated random number generation [RNG]. The cuRAND library delivers high quality random numbers 8x faster using hundreds of processor cores available in NVIDIA GPUs. The cuRAND library is included in both the [NVIDIA HPC SDK](#) and the [CUDA Toolkit](#).

Explore what's new in the latest release...

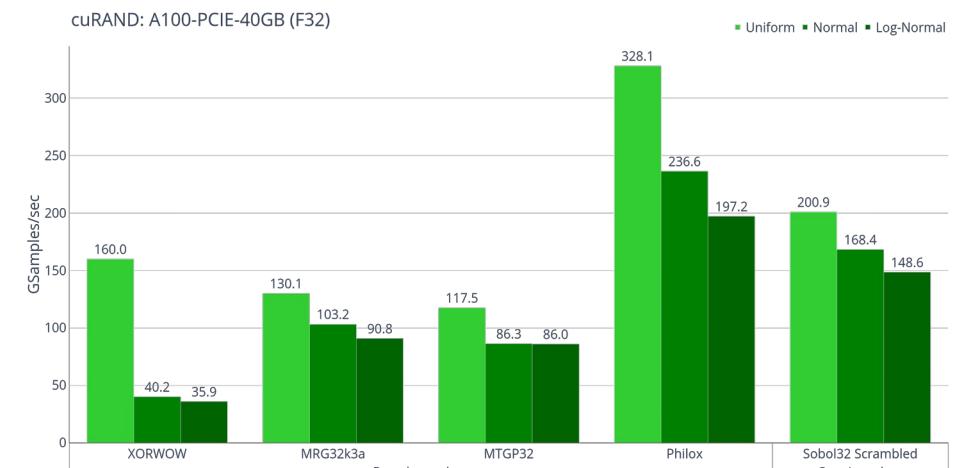
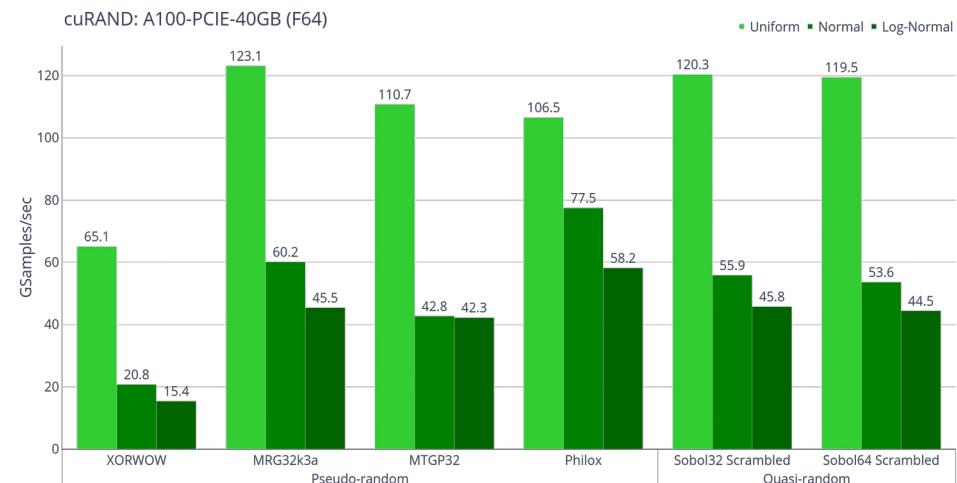
Review the latest [CUDA performance report](#) to learn how much you could accelerate your code.

cuRAND Performance

cuRAND also provides two flexible interfaces, allowing you to generate random numbers in bulk from host code running on the CPU or from within your CUDA functions/kernels running on the GPU. A variety of RNG algorithms and distribution options means you can select the best solution for your needs.

cuRAND Key Features

- **Flexible usage model**
 - Host API for generating random numbers in bulk on the GPU
 - Inline implementation allows use inside GPU functions/kernels, or in your host code
- **Four high-quality RNG algorithms**
 - MRG32k3a
 - MTGP Mersenne Twister
 - XORWOW pseudo-random generation
 - Sobol' quasi-random number generators, including support for scrambled and 64-bit RNG
- **Multiple RNG distribution options**
 - Uniform distribution
 - Normal distribution
 - Log-normal distribution
 - Single-precision or double-precision
 - Poisson distribution



- CURAND_RNG_PSEUDO_DEFAULT
- CURAND_RNG_PSEUDO_XORWOW
- CURAND_RNG_PSEUDO_MRG32K3A
- CURAND_RNG_PSEUDO_MTGP32
- CURAND_RNG_PSEUDO_MT19937
- CURAND_RNG_PSEUDO_PHILOX4_32_10
- CURAND_RNG_QUASI_DEFAULT
- CURAND_RNG_QUASI_SOBOLO32
- CURAND_RNG_QUASI_SCRAMBLED_SOBOLO32
- CURAND_RNG_QUASI_SOBOLO64
- CURAND_RNG_QUASI_SCRAMBLED_SOBOLO64

cuRAND

The API reference guide for cuRAND, the CUDA random number generation library.

Hay 2 formas de generar nros aleatorios.
cuRAND provee las 2:

Table of Contents

- [Introduction](#)
- [1. Compatibility and Versioning](#)
- [2. Host API Overview](#)
 - [2.1. Generator Types](#)
 - [2.2. Generator Options](#)
 - [2.2.1. Seed](#)
 - [2.2.2. Offset](#)
 - [2.2.3. Order](#)
 - [2.3. Return Values](#)
 - [2.4. Generation Functions](#)
 - [2.5. Host API Example](#)
 - [2.6. Static Library support](#)
 - [2.7. Performance Notes](#)
- [3. Device API Overview](#)
 - [3.1. Pseudorandom Sequences](#)
 - [3.1.1. Bit Generation with XORWOW and MRG32k3a generators](#)
 - [3.1.2. Bit Generation with the MTGP32 generator](#)
 - [3.1.3. Bit Generation with Philox_4x32_10 generator](#)
 - [3.1.4. Distributions](#)
 - [3.2. Quasirandom Sequences](#)
 - [3.3. Skip-Ahead](#)
 - [3.4. Device API for discrete distributions](#)
 - [3.5. Performance Notes](#)
 - [3.6. Device API Examples](#)
 - [3.7. Thrust and cuRAND Example](#)
 - [3.8. Poisson API Example](#)
- [4. Testing](#)
- [5. Modules](#)
 - [5.1. Host API](#)
 - [5.2. Device API](#)
- [A. Bibliography](#)
- [B. Acknowledgements](#)

- Se trabaja como con cualquier biblioteca para CPU.
- El resultado es un array de números random.
- El trabajo se hace en host o en device.
- Cuando se hace en el device, el resultado queda en la memoria del device.

Se llaman funciones que están disponibles desde el host. Se mandan a los hilos los nros ya generados ((o bien se generan en el device))

Cada hilo genera sus propios hilos

No todas las librerías permiten usar la Device API. Ej: cupy

- Se usan funciones de device para generar secuencias de números random para ser consumidos directamente dentro de kernels.
- Más rápido, menos memoria.
- Util cuando al kernel o los functors lo escribimos nosotros.

Ejemplos

- Host API: se llama desde el host

Genera el array random en device



curandCreateGenerator

```
#ifndef CPU
float *devData;

/* Allocate n floats on device */
CUDA_CALL(cudaMalloc((void **)&devData, n*sizeof(float)));

/* Create pseudo-random number generator */
CURAND_CALL(curandCreateGenerator(&gen,
    CURAND_RNG_PSEUDO_DEFAULT));

/* Set seed */
CURAND_CALL(curandSetPseudoRandomGeneratorSeed(gen,
    1234ULL));

/* Generate n floats on device */
CURAND_CALL(curandGenerateUniform(gen, devData, n));

/* Copy device memory to host */
CUDA_CALL(cudaMemcpy(hostData, devData, n * sizeof(float),
    cudaMemcpyDeviceToHost));
#else
```

Genera el array random en host



curandCreateGeneratorHost

```
/* Create pseudo-random number generator */
CURAND_CALL(curandCreateGeneratorHost(&gen,
    CURAND_RNG_PSEUDO_DEFAULT));

/* Set seed */
CURAND_CALL(curandSetPseudoRandomGeneratorSeed(gen,
    1234ULL));

/* Generate n floats on device */
CURAND_CALL(curandGenerateUniform(gen, hostData, n));
```

Se podría haber
usado un array de
thrust

π cuda

Calculando Pi usando números aleatorios



¿Versión Paralela? → M tiradores tiran N dardos

¿Porque conviene?

¿Que cuidados hay que tener?

→ Debería ser como si un solo tirador tirara $N \times M$

Genero y guardo todas las tiradas y cuento

- Host API curand + thrust

Problema: genero un montón de datos

Predicate functor

```
struct dentroCuadradoUnidad
{
    host__device_
    bool operator()(thrust::tuple<float, float> tup)
    {
        float x=thrust::get<0>(tup);
        float y=thrust::get<1>(tup);
        return (x*x+y*y)<1;
    }
};
```

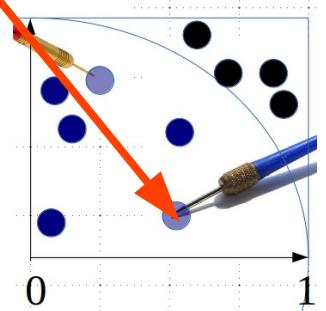
```
thrust::device_vector<float> devData(n);

/* Create pseudo-random number generator */
CURAND_CALL(curandCreateGenerator(&gen,
    CURAND_RNG_PSEUDO_DEFAULT));

/* Set seed */
CURAND_CALL(curandSetPseudoRandomGeneratorSeed(gen,
    1234ULL));

/* Generate n floats on device */
float *devData_raw=thrust::raw_pointer_cast(&devData[0]);
CURAND_CALL(curandGenerateUniform(gen, devData_raw, n));

// usamos la primera mitad como x, la segunda como y
int adentro = thrust::count_if(
    thrust::make_zip_iterator((thrust::make_tuple(devData.begin(),devData.begin()+n/2)),
    thrust::make_zip_iterator((thrust::make_tuple(devData.begin() +n/2,devData.begin() +n)),
    dentroCuadradoUnidad()
);
```



La mitad son x, la otra mitad son y, en $[0,1]$.

countif: Cuenta los que caen en el cuadrante del círculo.

Genero y guardo todas las tiradas y cuento

- Host API de cupy (no hay device API)

Random sampling (`cupy.random`)

Differences between `cupy.random` and `numpy.random`:

- Most functions under `cupy.random` support the `dtype` option, which do not exist in the corresponding NumPy APIs. This option enables generation of float32 values directly without any space overhead.
- `cupy.random.default_rng()` uses XORWOW bit generator by default.
- Random states cannot be serialized. See the description below for details.
- CuPy does not guarantee that the same number generator is used across major versions. This means that numbers generated by `cupy.random` by new major version may not be the same as the previous one, even if the same seed and distribution are used.

numpy

```
def piCPU(n):
    ## en CPU
    data = np.random.uniform(-0.5,0.5,(n,2))

    inside = len(
        np.argwhere(
            np.linalg.norm(data, axis=1) < 0.5
        )
    )
    print(inside/n*4)
```

cupy

```
def piGPU(n):
    ## en GPU
    dataGPU = cp.random.uniform(-0.5,0.5,(n,2))

    inside = len(
        cp.argwhere(
            cp.linalg.norm(dataGPU, axis=1) < 0.5
        )
    )
    print(inside/n*4)
```

Más eficiente: Cuento mientras voy tirando

Semilla0, Semilla1, Semilla2, Semilla3, Semilla4, Semilla5

Transformación

Pseudocódigo

```
Semilla0
while(Totales < N)
{
    x, y números random en [0,1]
    Azules += ((x*x + y*y) < 1)
    Totales++;
}
Pi[0] = 4*Azules/Totales
```

```
Semilla1
while(Totales < N)
{
    x, y números random en [0,1]
    Azules += ((x*x + y*y) < 1)
    Totales++;
}
Pi[0] = 4*Azules/Totales
```

```
Semilla2
while(Totales < N)
{
    x, y números random en [0,1]
    Azules += ((x*x + y*y) < 1)
    Totales++;
}
Pi[0] = 4*Azules/Totales
```

```
Semilla3
while(Totales < N)
{
    x, y números random en [0,1]
    Azules += ((x*x + y*y) < 1)
    Totales++;
}
Pi[0] = 4*Azules/Totales
```

```
Semilla4
while(Totales < N)
{
    x, y números random en [0,1]
    Azules += ((x*x + y*y) < 1)
    Totales++;
}
Pi[0] = 4*Azules/Totales
```

```
Semilla5
while(Totales < N)
{
    x, y números random en [0,1]
    Azules += ((x*x + y*y) < 1)
    Totales++;
}
Pi[0] = 4*Azules/Totales
```

numero random = curand(tid)

Lo vamos a semillar con el nro de hilo

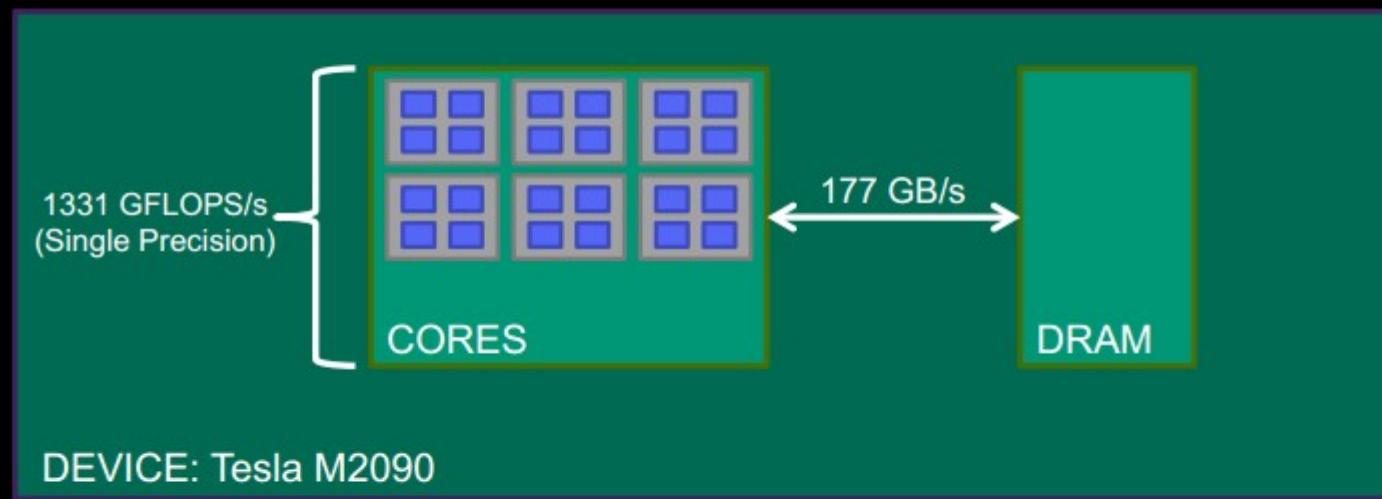
Reducción

Reducción: $Pi = (Pi[0] + Pi[1] + Pi[2] + Pi[3] + Pi[4] + Pi[5])/6$

Buenas prácticas



Simplified View of a GPU



Necesitamos poder generar dentro de kernels

Necesitamos un “device API”

Semilla0

```
while(Totales < N)
{
    x, y números random en [0,1]
    Azules += ((x*x + y*y) < 1)
    Totales ++
}
Pi[0] = 4*Azules/Totales
```

Semilla1

```
while(Totales < N)
{
    x, y números random en [0,1]
    Azules += ((x*x + y*y) < 1)
    Totales ++
}
Pi[0] = 4*Azules/Totales
```

Semilla2

```
while(Totales < N)
{
    x, y números random en [0,1]
    Azules += ((x*x + y*y) < 1)
    Totales ++
}
Pi[0] = 4*Azules/Totales
```

Semilla3

```
while(Totales < N)
{
    x, y números random en [0,1]
    Azules += ((x*x + y*y) < 1)
    Totales ++
}
Pi[0] = 4*Azules/Totales
```

Semilla4

```
while(Totales < N)
{
    x, y números random en [0,1]
    Azules += ((x*x + y*y) < 1)
    Totales ++
}
Pi[0] = 4*Azules/Totales
```

Semilla5

```
while(Totales < N)
{
    x, y números random en [0,1]
    Azules += ((x*x + y*y) < 1)
    Totales ++
}
Pi[0] = 4*Azules/Totales
```

Si N (dardos) y M (tiradores) son $N \gg 1$ y $M \gg 1$

• Como elegimos las M semillas?

- Como se que cada tirador tira una buena secuencia? (tiros uniformes y descorrelacionados)
- Como se que la secuencia de cada tirador esta descorrelacionada con los demás?

Ejemplos

- Device API+thrust

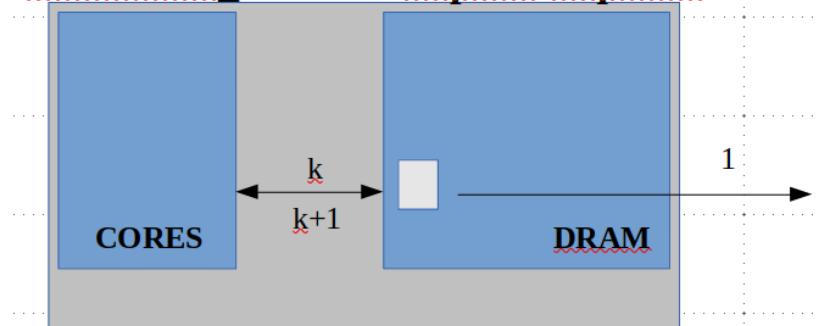
```
int main(void)
{
    // use 30K independent seeds
    int M = 30000;

    float estimate = thrust::transform_reduce(
        thrust::counting_iterator<int>(0),
        thrust::counting_iterator<int>(M),
        estimate_pi(),
        0.0f,
        thrust::plus<float>());
    estimate /= M;

    std::cout << std::setprecision(6);
    std::cout << "pi is approximately ";
    std::cout << estimate << std::endl;
    return 0;
}
```

Valor inicial y
operación
de reducción

Transform_reduce + implicit sequence



La secuencia no se genera porque es implícita

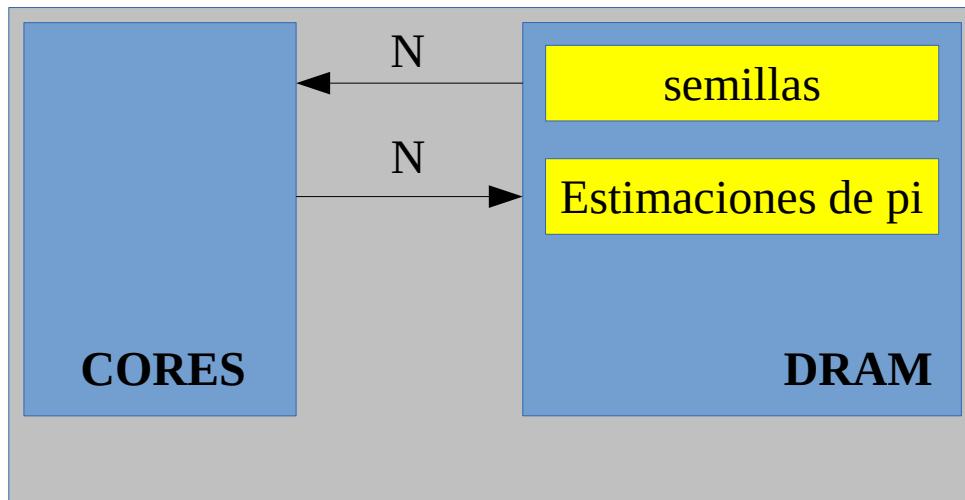
}

Una secuencia (implícita para ahorrar...)
de semillas 0,1,2,..., M-1

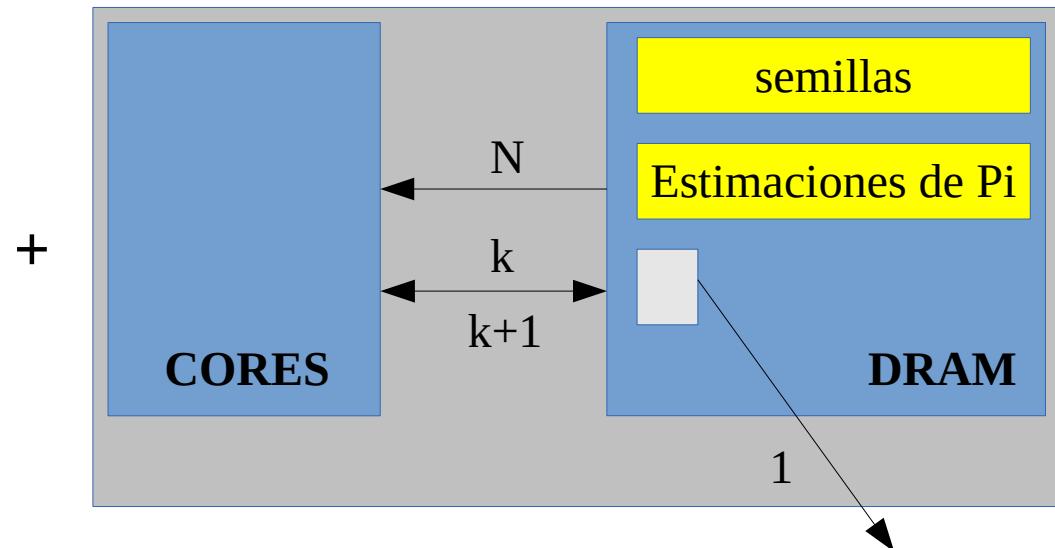
Que se genera con cada semilla
Una estimación parcial de Pi
Definido como **functor**

Optimización: kernel-fusion + implicit sequences

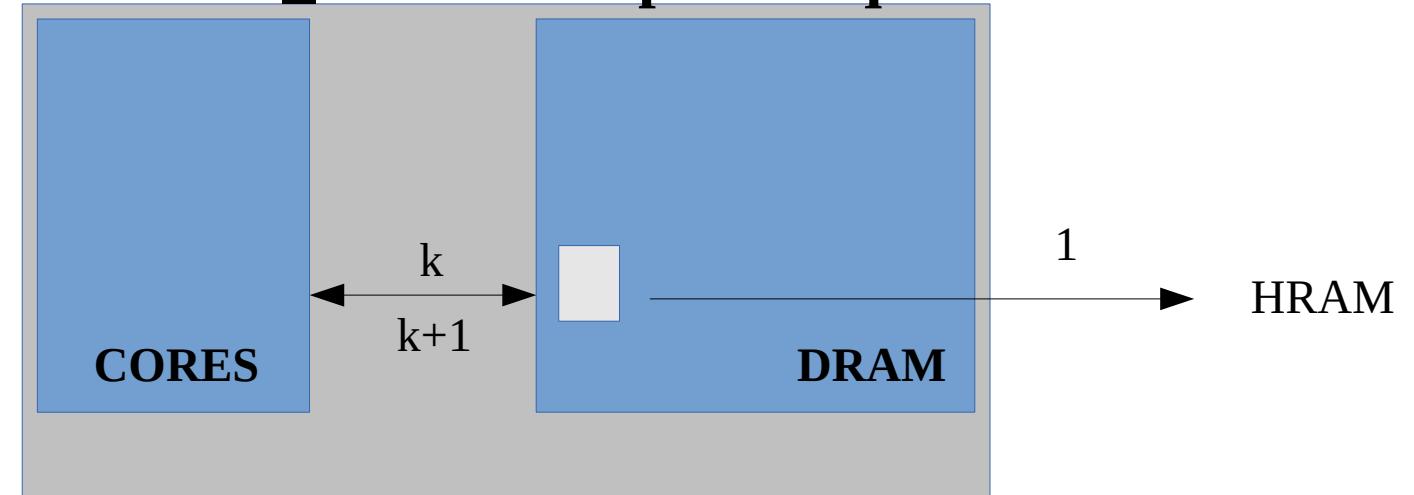
transform



reduce



Transform_reduce + implicit sequence

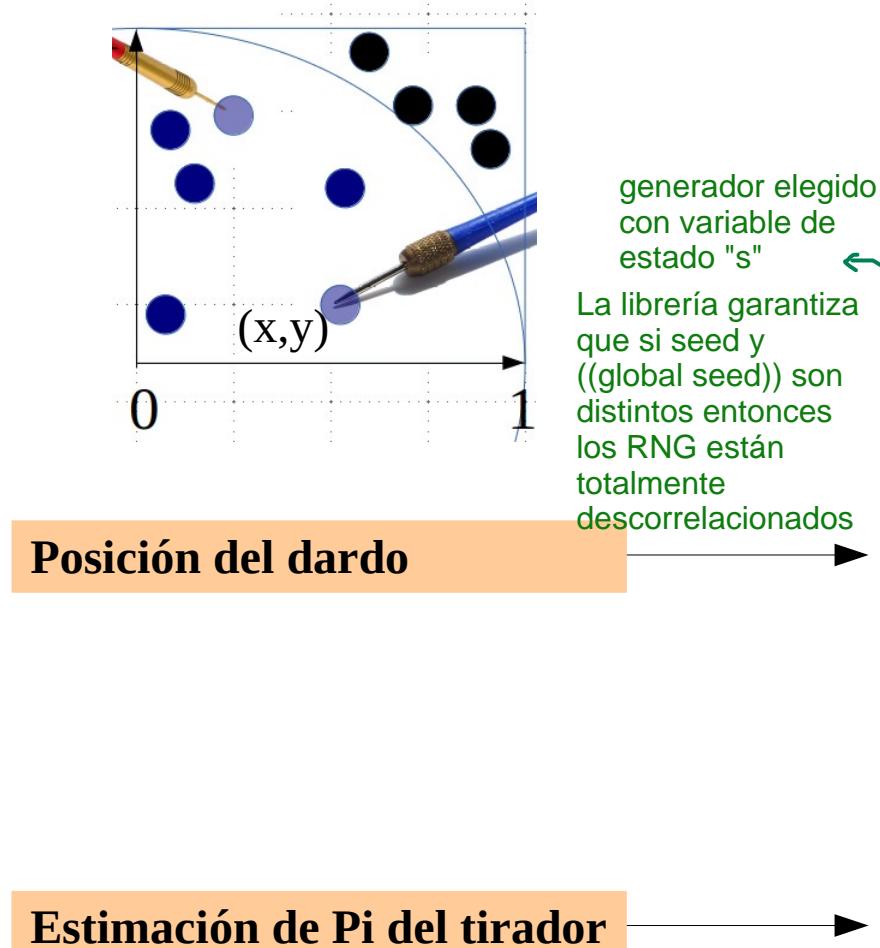


Adaptado de suma
de cuadrados de

Julien Demouth, Nvidia

Ejemplos

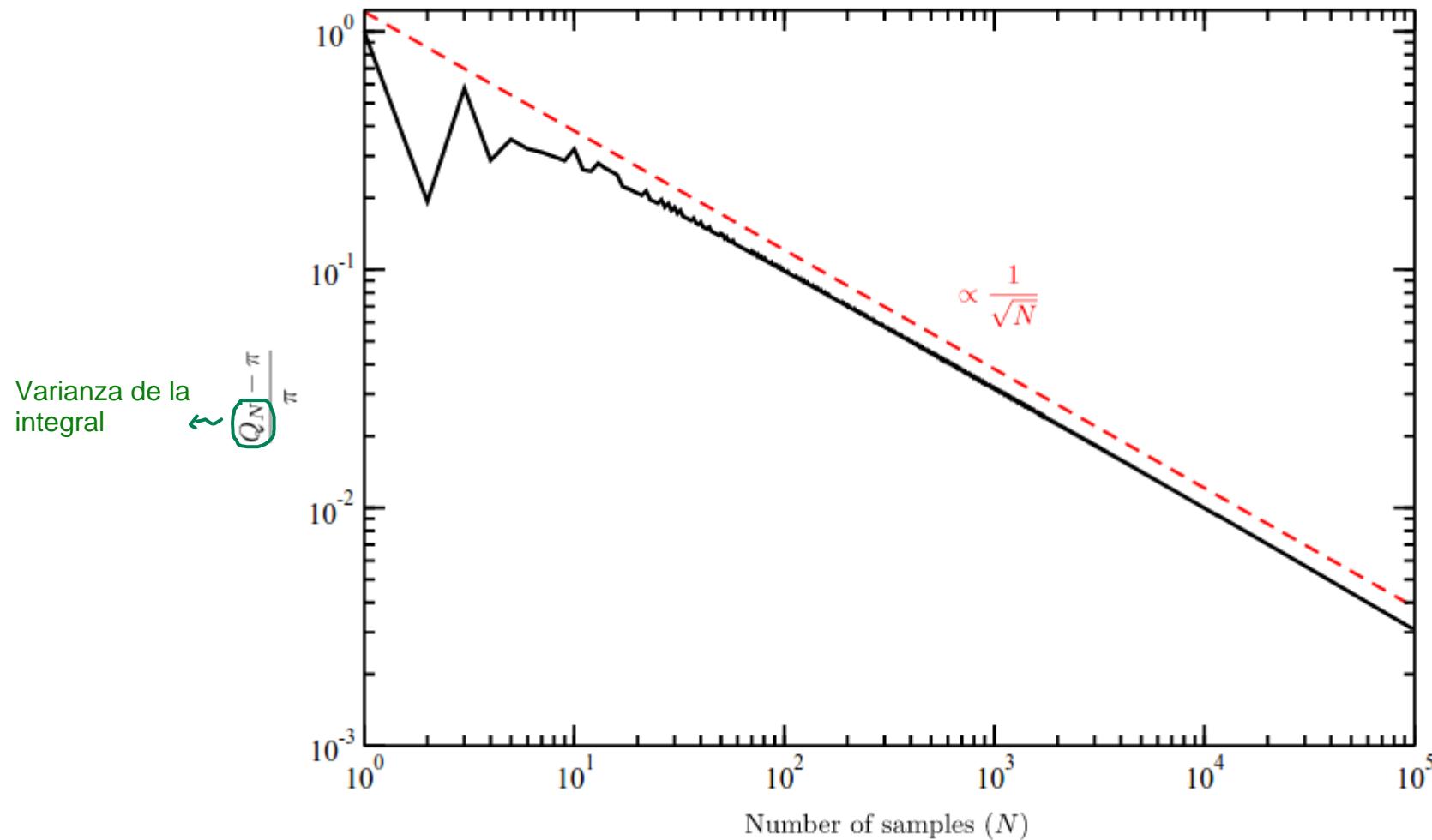
- Device API+thrust



Functor: cada hilo su estimación de Pi

```
struct estimate_pi :  
| public thrust::unary_function<unsigned int, float>  
{  
| | _device_  
| | float operator()(unsigned int thread_id)  
| | {  
| | | float sum = 0;  
| | | unsigned int N = 10000; // samples per thread  
| | | unsigned int seed = thread_id; nro del hilo!  
| | | curandStatePhilox4_32_10_t s;  
| | |  
| | | // seed a random number generator  
| | | curand_init(GLOBALSEED,seed, 0, &s);  
| | |  
| | | // take N samples in a quarter circle  
| | | for(unsigned int i = 0; i < N; ++i)  
| | | {  
| | | | // draw a sample from the unit square  
| | | | float x = curand_uniform(&s);  
| | | | float y = curand_uniform(&s);  
| | | |  
| | | | // measure distance from the origin  
| | | | float dist2 = (x*x + y*y);  
| | | |  
| | | | // add 1.0f if (u0,u1) is inside the quarter circle  
| | | | if(dist2 <= 1.0f)  
| | | | | sum += 1.0f;  
| | | }  
| | |  
| | | // multiply by 4/N to get the area of the whole circle  
| | | sum *= 4.0f/N;  
| | |  
| | | return sum;  
| | }  
};
```

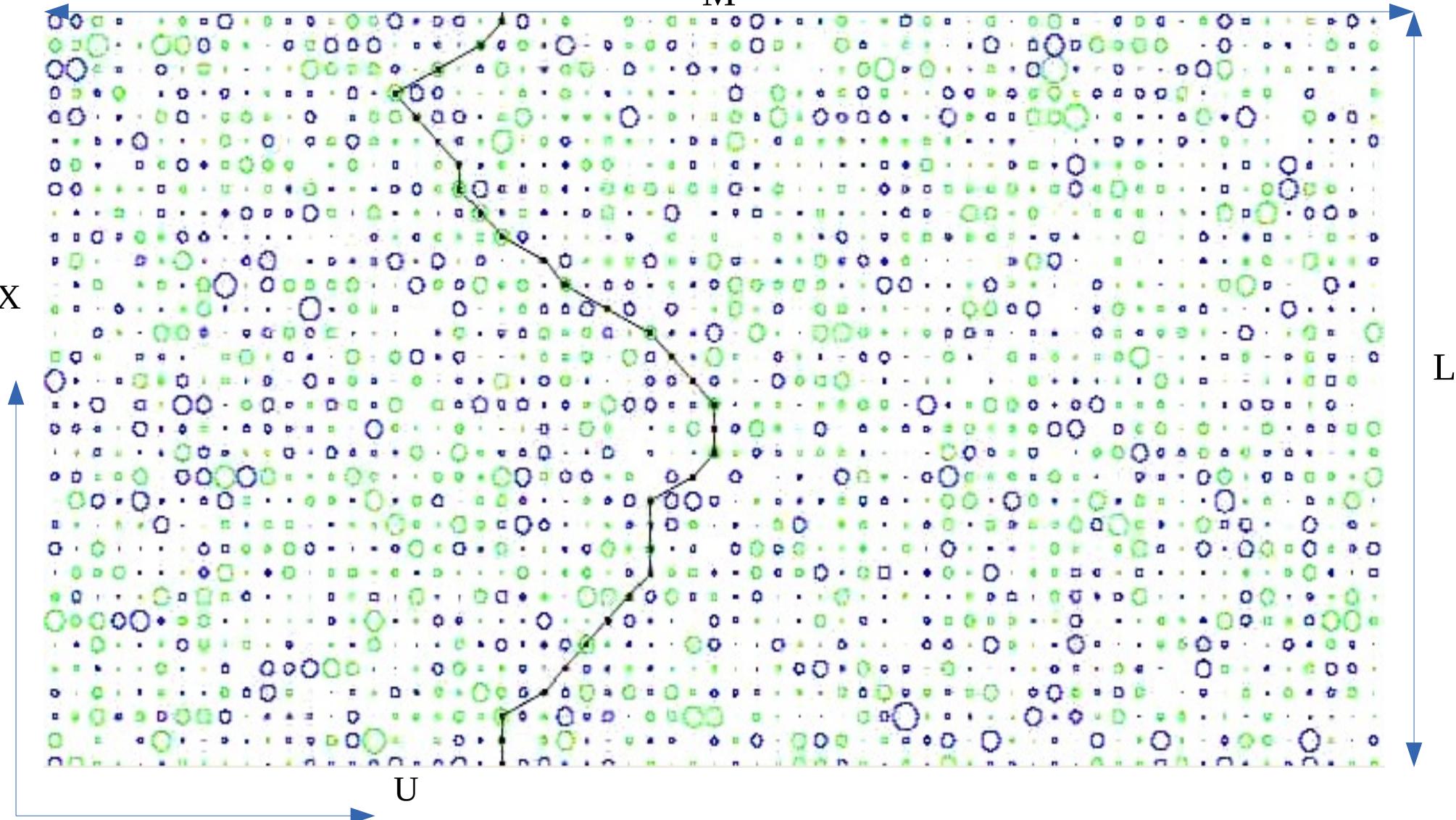
¿ Converge a Pi ?



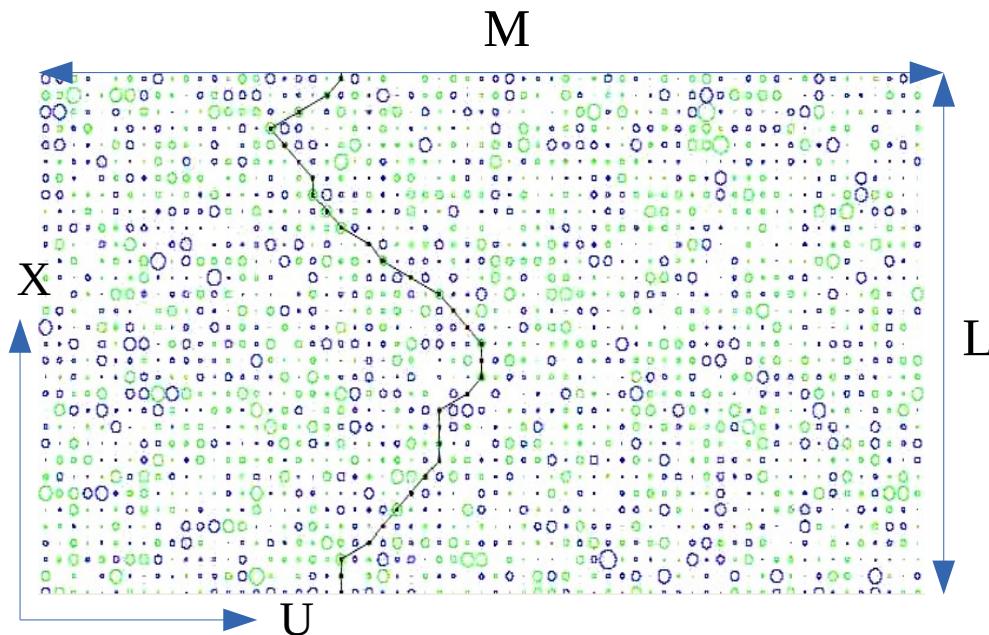
¿Por qué se recomienda Philox?

Desorden

Cuerda en una malla sobre la que hay pozos y montañas fijas M



El problema con el Desorden



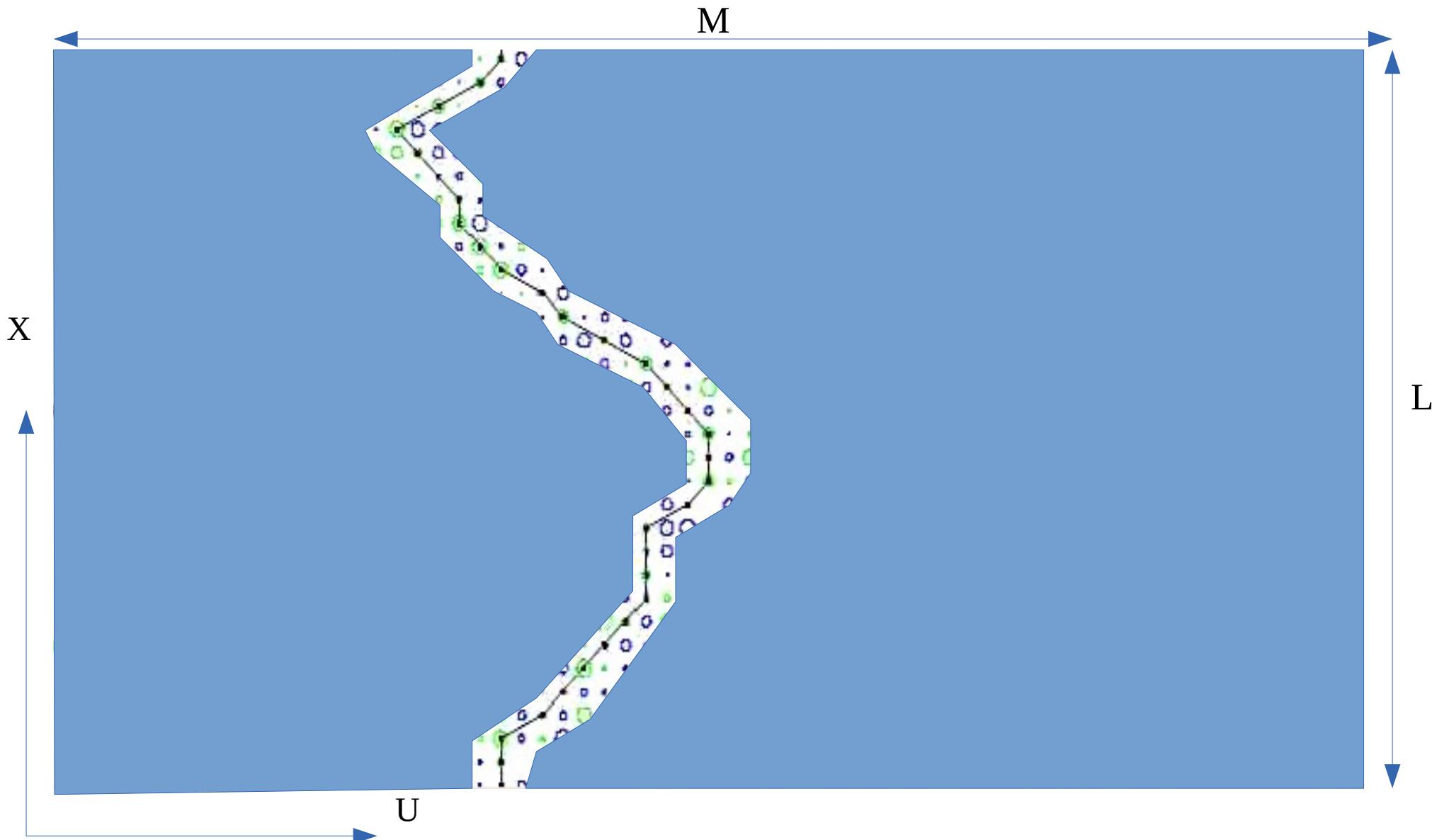
- Necesitamos simular muestras $L \times M$, con: $L > 2^{16} = 65536$ y $M \sim L^{5/4} \sim 2^{20}$.
- Si guardaramos el desorden en una matriz de $L \times M > 2^{16+20}$ floats = **256 Gb** (!!).

```
[koltona@compute-0-5 deviceQuery]$ ./deviceQuery
```

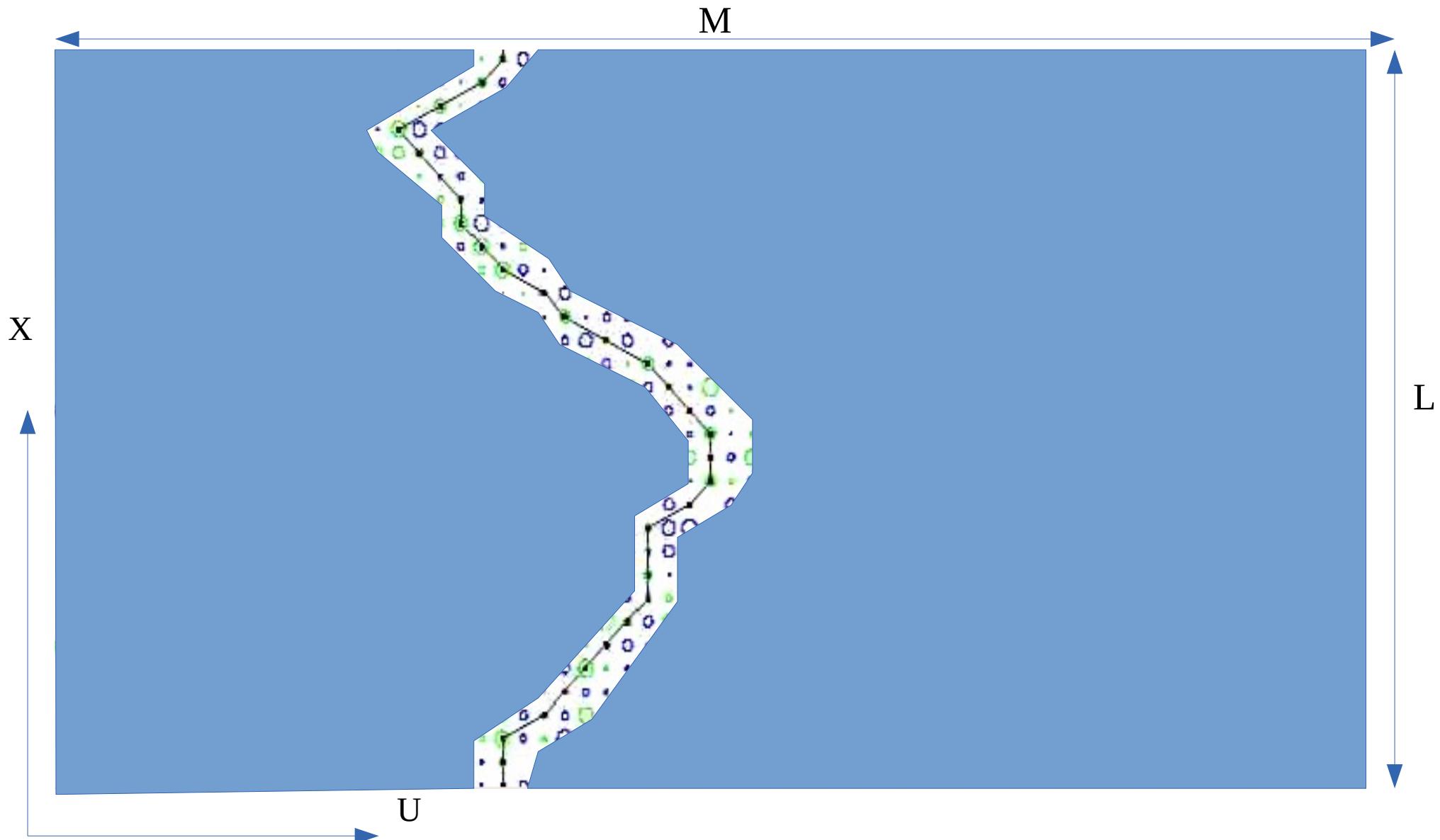
```
Device 0: "GeForce GTX TITAN Black"
CUDA Driver Version / Runtime Version      7.5 / 6.5
CUDA Capability Major/Minor version number: 3.5
Total amount of global memory:      6143 MBytes (6441730048 bytes)
(15) Multiprocessors, (192) CUDA Cores/MP:   2880 CUDA Cores
```

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

Desorden generado “dinámicamente”



¿Y si la cuerda retrocede?



Necesitamos un RNG que me de los mismos datos si "vuelvo hacia atrás"

Problema

- Generador de números aleatorio de buena calidad ...
- *Que me permita recuperar números que ya genero, rápidamente ...*
- Generador serial y paralelo ...

Problema

- Generador de números aleatorio de buena calidad ...
- *Que me permita recuperar números que ya genero, rápidamente ...*
- Generador serial y paralelo ... interface portable.

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, $r = 521$	≥ 80	None	2	—	failed	23
Fibonacci, $r = 1279$	≥ 80	None	(1)	2	passed	23
XORWOW (cuRAND)	192	None	None	1/3	failed	19
MTGP (cuRAND)	≥ 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?

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. ×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, $r = 521$	≥ 80	None	2	—	failed	23
Fibonacci, $r = 1279$	≥ 80	None	(1)	2	passed	23
XORWOW (cuRAND)	192	None	None	1/3	failed	19
MTGP (cuRAND)	≥ 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?

Random number generators for massively parallel simulations on GPU

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

Regeneración sencilla de números ya generados?

generator	bits/thread	failures in TestU01			Ising test	perf. ×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, $r = 511$	≥ 80	None	2	—	failed	23
Fibonacci, $r = 1279$	≥ 80	None	(1)	2	passed	23
XORWOW (cuRAND)	192	None	None	1/3	failed	19
MTGP (cuRAND)	≥ 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

complicado recuperar números

Counter-based RNGs



- The Random123 library is a collection of counter-based random number generators ([CBRNGs](#)) for **CPUs (C and C++) and GPUs (CUDA and OpenCL)**, as described in [Parallel Random Numbers: As Easy as 1, 2, 3](#), Salmon, Moraes, Dror & Shaw, **SC11, Seattle , Washington, USA, 2011**, ACM
- They are intended for use in **statistical applications and Monte Carlo simulation** and have passed all of the rigorous SmallCrush, Crush and BigCrush tests in the [extensive TestU01 suite](#)of statistical tests for random number generators.
- They are **not suitable for use in cryptography or security** even though they are constructed using principles drawn from cryptography.

<http://www.thesalmons.org/john/random123/releases/1.06/docs/>

Counter-based RNGs

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

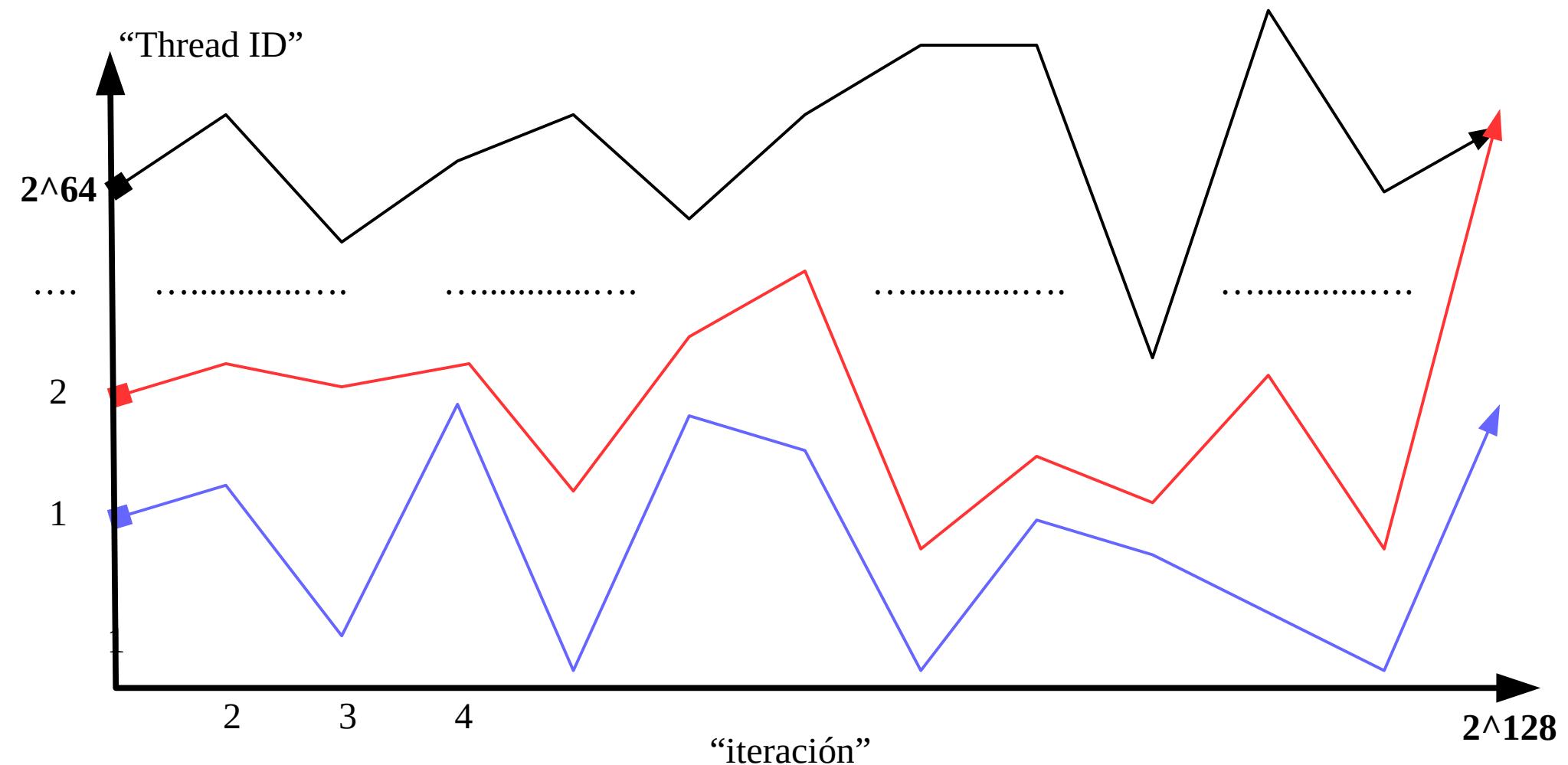
John K. Salmon, Mark A. Moraes, Ron O. Dror, David E. Shaw

".. 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

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

“.. We demonstrate that... **2^{64} unique parallel streams of random numbers, each with period 2^{128} or more...**”.



Overview Random123



- Unlike conventional RNGs, **counter-based RNGs are stateless functions** (or function classes i.e. functors) whose arguments are a counter, and a key and whose return value is the same type as the counter.

Es un
mapeo de
2 nros a
un nro
random

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

Podemos usar key = thread id y counter como la iteración que estamos haciendo

- The returned value is a **deterministic function of the key and counter**.
- The result is **highly sensitive to small changes in the inputs**.
- **CBRNGs are as fast as, or faster** than conventional RNGs, much **easier to parallelize**, use **minimal memory/cache** resources, and require very **little code**.
- The CBRNGs in the Random123 library work with counters and keys consisting of N words, where words have a width of W bits, encapsulated in r123arrayNxW structs, or equivalently, for C++, in the ArrayNxW typedefs in the r123 namespace.

Calculando Pi usando números aleatorios

Semilla0, Semilla1, Semilla2, Semilla3, Semilla4, Semilla5

Transformación

Semilla0

```
while(Totales < N)
{
    x, y números random en [0,1]
    Azules += ((x*x + y*y) < 1)
    Totales++;
}
Pi[0] = 4*Azules/Totales
```

Semilla1

```
while(Totales < N)
{
    x, y números random en [0,1]
    Azules += ((x*x + y*y) < 1)
    Totales++;
}
Pi[0] = 4*Azules/Totales
```

Semilla2

```
while(Totales < N)
{
    x, y números random en [0,1]
    Azules += ((x*x + y*y) < 1)
    Totales++;
}
Pi[0] = 4*Azules/Totales
```

Semilla3

```
while(Totales < N)
{
    x, y números random en [0,1]
    Azules += ((x*x + y*y) < 1)
    Totales++;
}
Pi[0] = 4*Azules/Totales
```

Semilla4

```
while(Totales < N)
{
    x, y números random en [0,1]
    Azules += ((x*x + y*y) < 1)
    Totales++;
}
Pi[0] = 4*Azules/Totales
```

Semilla5

```
while(Totales < N)
{
    x, y números random en [0,1]
    Azules += ((x*x + y*y) < 1)
    Totales++;
}
Pi[0] = 4*Azules/Totales
```

numero random = CBRNGname(Totales, tid)

Reducción

Reducción: $Pi = (Pi[0] + Pi[1] + Pi[2] + Pi[3] + Pi[4] + Pi[5])/6$

Thrust transform_reduce → Pi

```
struct estimate_pi{...}
using namespace thrust;
int main(void)
{
//uso 30K tiradores independientes, y cada uno tira 10K dardos
    int M = 30000;

    float estimate;
    estimate = transform_reduce(
        counting_iterator<int>(0),
        counting_iterator<int>(M),
        estimate_pi(),
        0.0f,
        thrust::plus<float>()
    );
    estimate /= M; // 300K dardos tirados...(comparar tiempos con CPU)

    std::cout << "pi es aproximadamente " << estimate << std::endl;

    return 0;
}
```

No se está usando cuRAND y por lo tanto la interfaz es distinta

En Clases_aleatorios
usar make para compilar y luego decir qué se quiere compilar, por ejemplo,
make cuda_Pi

Pi.cu

Valor inicial y operación de reducción

Que se genera con cada semilla

Una secuencia (implícita) de semillas 0,1,2,..., M-1

Functor de Thrust

Pi.cu

```
struct estimate_pi{
    __device__
    float operator()(unsigned int thread_id){

        RNG philox; RNG::ctr_type c={{}}, k={{}}, r;
        k[0]=thread_id; → Una “key” distinta
        // Para cada tirador

        float sum = 0;
        unsigned int N = 10000; // muestras por thread

        for(unsigned int i = 0; i < N; ++i){
            c[0]=i; → Un “counter” distinto
            r = philox(c, k); → Para cada dardo del tirador
            // Para cada dardo

            float x=(u01_closed_closed_32_53(r[0]));
            float y=(u01_closed_closed_32_53(r[1]));
            float dist = sqrtf(x*x + y*y); // distancia al origen

            if(dist <= 1.0f)
                sum += 1.0f;
        }
        sum *= 4.0f;
        return sum / N; → Estimación de Pi del tirador
    }
};
```

Metropolis Monte Carlo

Monte Carlo

Necesidad de realizar un muestreo pesado en el espacio de las configuraciones



Proceso estocástico markoviano $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \dots$

Sea $P(\mathbf{x}, t)$ la probabilidad asociada con el estado $\mathbf{x} \equiv \{s_i\}$ al tiempo t .

Consideremos la probabilidad de transición $W_{\mathbf{x} \rightarrow \mathbf{x}'}$ del estado \mathbf{x} al estado \mathbf{x}'

Ecuación maestra
$$\frac{d}{dt}P(\mathbf{x}, t) = - \sum_{x'} W_{\mathbf{x} \rightarrow \mathbf{x}'} P(\mathbf{x}, t) + \sum_{x'} W_{\mathbf{x}' \rightarrow \mathbf{x}} P(\mathbf{x}', t)$$

Si pedimos que $P(\mathbf{x}, t) \rightarrow P_{eq}(\mathbf{x})$ debe ocurrir $\frac{d}{dt}P(\mathbf{x}, t) = 0$

es decir,
$$\sum_{x'} W_{\mathbf{x} \rightarrow \mathbf{x}'} P(\mathbf{x}) = \sum_{x'} W_{\mathbf{x}' \rightarrow \mathbf{x}} P(\mathbf{x}')$$
 condición de **balance global**

ó (más fuerte)
$$W_{\mathbf{x} \rightarrow \mathbf{x}'} P(\mathbf{x}) = W_{\mathbf{x}' \rightarrow \mathbf{x}} P(\mathbf{x}')$$
 condición de **balance detallado**

Monte Carlo (cont.)

balance detallado:

$$\frac{W_{\mathbf{x} \rightarrow \mathbf{x}'}}{W_{\mathbf{x}' \rightarrow \mathbf{x}}} = \frac{P(\mathbf{x}')}{P(\mathbf{x})}$$

distribución de Boltzmann:

$$P(\mathbf{x}) \equiv P_{eq}(\mathbf{x}) \equiv e^{-\frac{\mathcal{H}(\mathbf{x})}{k_B T}}$$

$$\frac{W_{\mathbf{x} \rightarrow \mathbf{x}'}}{W_{\mathbf{x}' \rightarrow \mathbf{x}}} = e^{-\frac{\mathcal{H}(\mathbf{x}') - \mathcal{H}(\mathbf{x})}{k_B T}}$$

Metropolis

$$W_{\mathbf{x} \rightarrow \mathbf{x}'} = \frac{1}{\tau} \min[1, \exp(-\Delta \mathcal{H}/k_B T)]$$

Glauber

$$W_{\mathbf{x} \rightarrow \mathbf{x}'} = \frac{1}{2\tau} [1 - \tanh(\Delta \mathcal{H}/2k_B T)]$$

Algoritmo de Metropolis

- 1) partimos de una configuración \mathbf{x}
- 2) elegimos al azar una nueva configuración \mathbf{x}'
(que difiere de \mathbf{x} en el estado de un spin flip*)
- 3) calculamos ΔH
- 4) si $\Delta H \leq 0$, aceptamos la nueva configuración
- 5) si $\Delta H > 0$ la aceptamos con probabilidad $\exp(-\Delta H/k_B T)$
- 6) repitimos desde 2)

N repeticiones = 1 MCS

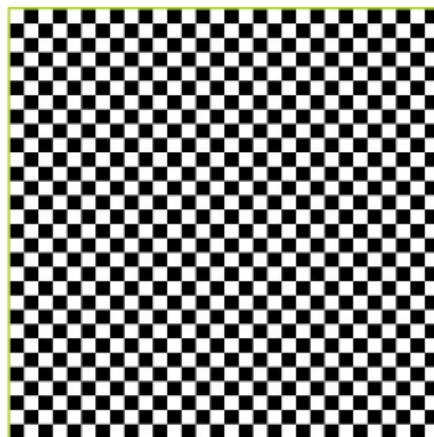
TBC in hands-on...

*Esquema clásico: elegir el sitio al azar
También se usa: orden secuencial 'typewriter'

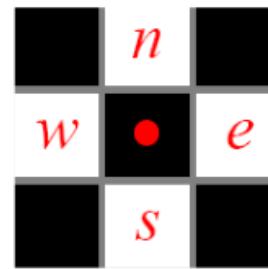
Estrategia de Paralelización

(interacción a primeros vecinos)

Esquema del "tablero de ajedrez"



Red cuadrada de $N=L \times L$ spins,
interacciones a primeros vecinos
+ c.c. periódicas



Dado un spin en un casillero negro,
todos sus vecinos estarán en casilleros
blancos, y viceversa

Dos subredes de spins no interactuantes: blancas/negras

En nuestro código CUDA un MCS será:

```
// white update, read from black
updateCUDA<<<dimGrid, dimBlock>>>(temp, WHITE, white, black);
CUT_CHECK_ERROR("Kernel updateCUDA(WHITE) execution failed");

// black update, read from white
updateCUDA<<<dimGrid, dimBlock>>>(temp, BLACK, black, white);
CUT_CHECK_ERROR("Kernel updateCUDA(BLACK) execution failed");
```

Actualización "not-random", NO cumple **balance detallado** pero SÍ balance global

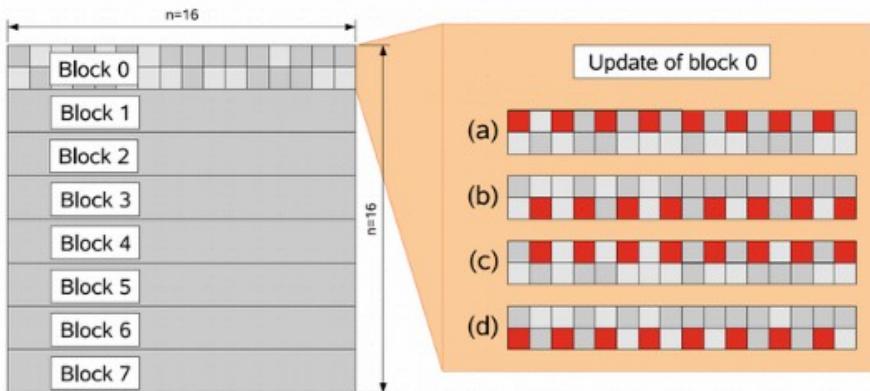
Implementaciones del *checkerboard*: Ejemplo 1



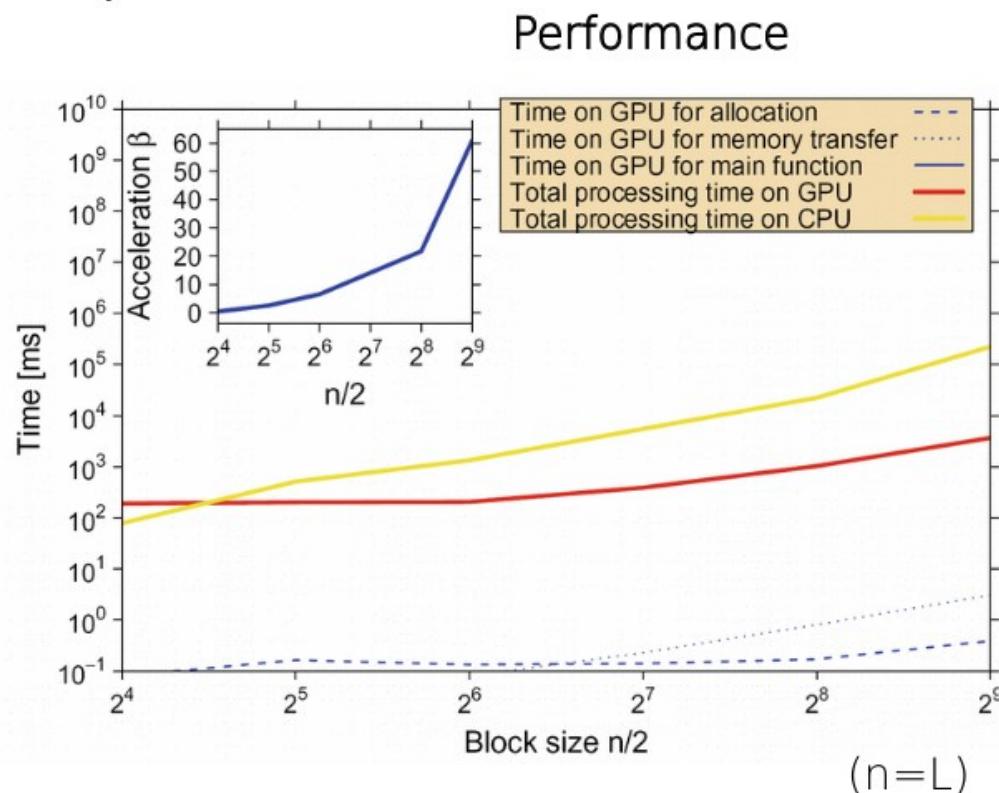
T. Preis, P. Virnau, W. Paul, J.J. Schneider
Johannes Gutenberg University of Mainz, Germany

J. Comput. Phys. 228, 4468 (2009)

Implementación Ising 2D:



e.g.: paralelización para un sistema de 16x16 spins



(n=L)

- Características:**
- primer trabajo en IsingGPGPU (JCP 'most read' en 2009).
 - LCG-32 inicializado *at random*.
 - todos los updates se hacen en memoria global.
 - transfieren array de semillas a shared antes de cada kernel.
 - limitación de tamaño $L \leq 1024$ para la GT200, ($L \leq 2048$ para Fermi).

Aceleración máxima: 60x (con GTX 280)

<https://arxiv.org/pdf/1906.06297.pdf>

lattice size	flip/ns
$(1 \times 2048)^2$	231.09
$(2 \times 2048)^2$	318.95
$(4 \times 2048)^2$	379.27
$(8 \times 2048)^2$	411.65
$(16 \times 2048)^2$	420.44
$(32 \times 2048)^2$	420.77
$(64 \times 2048)^2$	418.23
$(123 \times 2048)^2$	417.53
1 TPUv3 core in [7]	12.91
32 TPUv3 cores in [7]	336.01
FPGA (1024^2) [8]	614.40

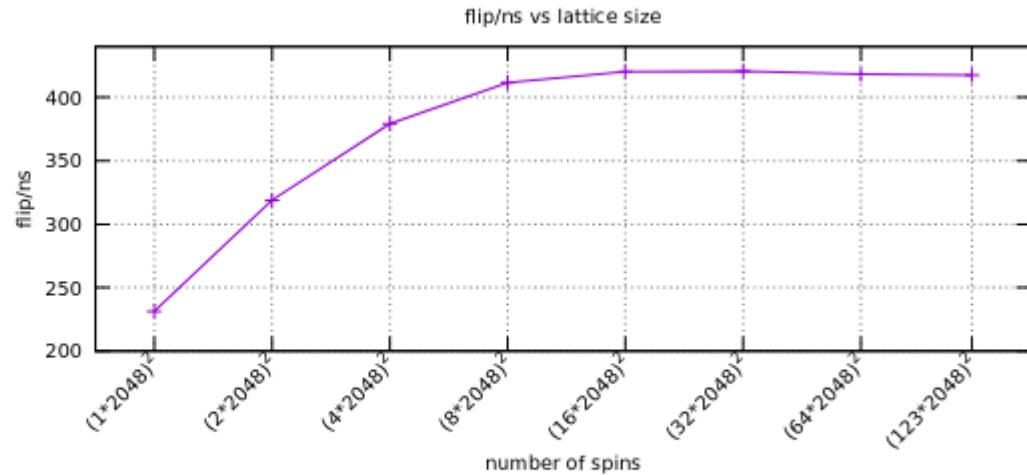


Table 2: Flips per nanosecond obtained by the optimized multi-spin code on a single Tesla V100-SXM card with different lattice sizes, requiring an amount of memory ranging from 2MB to 30GB. For comparison purposes, the table also reports the best timings with 1 and 32 TPUv3 cores from [7], and with 1 FPGA from [8].

lattice size	Basic (Python)	Basic (CUDA C)	Tensor Core	TPU (on TPUv3 core) [7]
$(20 \times 128)^2$	15.179	48.147	31.010	8.1920
$(40 \times 128)^2$	40.984	59.606	35.356	9.3623
$(80 \times 128)^2$	42.887	64.578	38.726	12.336
$(160 \times 128)^2$	43.594	66.382	39.152	12.827
$(320 \times 128)^2$	43.768	66.787	39.208	12.906
$(640 \times 128)^2$	43.535	66.954	38.749	12.878

Table 1: Single-GPU performance comparison between basic, tensor core, and TPU implementations. Results reported in flips per nanosecond on the same lattices used in [7].

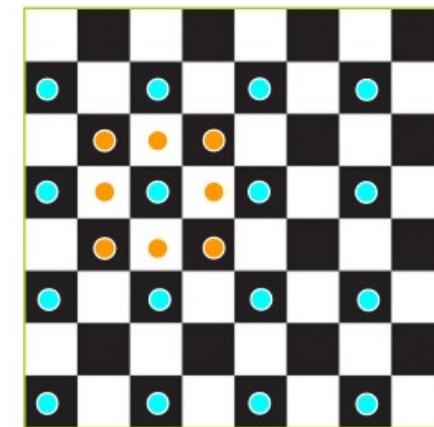
Extensiones a otros sistemas

- Interacciones a **2^{dos} o 3^{ros} vecinos** (o extensión a **d>2**)

El 'chequerboard' (ampliado) seguiría siendo útil

- Interacciones de **largo alcance**

En el **espacio real** no se puede hacer más que **parallelizar las sumas** en el cálculo de campos locales



- Interacciones competitivas J_{ij} (**spin glass**)

La extensión es directa

$$\mathcal{H} = - \sum_{\langle i,j \rangle} J_{ij} s_i s_j - \sum_i h_i s_i$$

- Spins continuos (tipo Heisenberg)
speed-up (en SP) mejor que en Ising

$$\mathcal{H} = -J \sum_{\langle i,j \rangle} \mathbf{s}_i \cdot \mathbf{s}_j, \quad |\mathbf{s}_i| = 1$$



Computer Physics Communications
Volume 196, November 2015, Pages 290-303



Highly optimized simulations on single- and multi-GPU systems of the 3D Ising spin glass model

M. Lulli ^a M. Bernaschi ^b, G. Parisi ^{a, c}

<https://doi.org/10.1016/j.cpc.2015.06.019>

Order, Disorder and Criticality, pp. 271-340 (2018)

Chapter 6: Monte Carlo Methods for Massively Parallel Computers

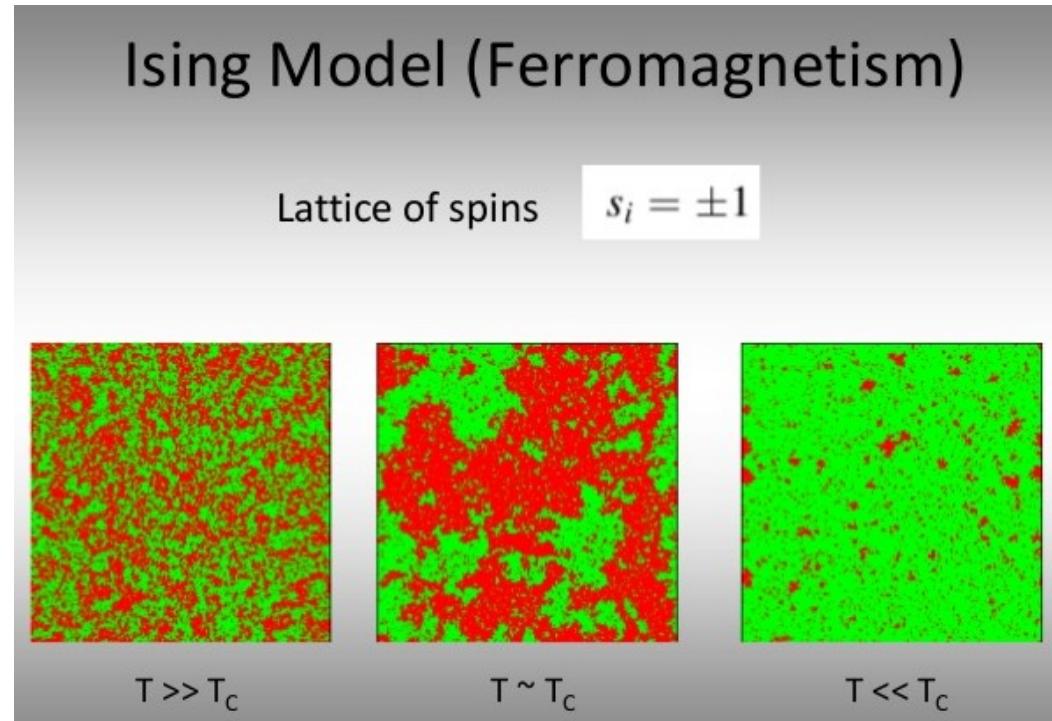
Martin Weigel

https://doi.org/10.1142/9789813232105_0006

Posibles proyectos de fin de curso!

Modelo de Ising en 2d en la red cuadrada

- `$ cp -r /share/apps/icnpg/clases/clase10 .`
- `$ cd clase10/MinIsing`
- `$ make`
- `$ qsub jobGPU`
- `$ qsub jobGPU_omp`
- *Comparar*



https://en.wikipedia.org/wiki/Square-lattice_Ising_model

Modelo de Ising en 2d

```
thrust::device_vector<int> M(N);
...
// loop de pasos de MonteCarlo
for(int nt=0;nt<nrun;nt++)
{
    //ojo!: lento, GPU->CPU->disco de mucha info! (hacer cada muchos pasos)
    if((nt+1)%tsnap==0) print_campo_de_magnetizacion(M, L, fout);

    // imprime magnetizacion por sitio (usando parallel reduction)
    mout << thrust::reduce(M.begin(),M.end())*1.0/N << std::endl;

    for(int color=0;color<2;color++){ // "checkerboard decomposition"
        // update de sitios de color "color" usando transform_if paralelo
        thrust::transform_if(
            thrust::make_counting_iterator(0),thrust::make_counting_iterator(N), //rango
            M.begin(), // output
            metropolis(Mraw,T,L,nt,globalseed), // operacion
            ficha(color, L) // predicado
        );
    }
}
```

Preguntas thrust:

- `thrust::make_counting_iterator(...)` ???
- `thrust::transform_if(...)` ???

```
struct ficha
{
    bool color;
    int L;
    ficha(bool _color, int _L):color(_color),L(_L){};

    __device__ __host__
    bool operator()(int n){
        return ((n%L+int(n/L))%2==color); // true si n es
        color
    };
};
```

Modelo de Ising en 2d

```
struct metropolis
{
    int L; float T; int *Mptr; int t; int seed;
    metropolis(int *_Mptr, float _T, int _L, int _t, int _seed):
        Mptr(_Mptr), T(_T), L(_L), t(_t), seed(_seed) {}

    __device__ int operator()(int n)
    {
        int nx=n%L; int ny=int(n/L);

        int local_field=
            Mptr[(nx-1+L)%L + ny*L] + Mptr[(nx+1+L)%L + ny*L] +
            Mptr[nx+((ny+1+L)%L)*L] + Mptr[nx+((ny-1+L)%L)*L];

        // contribucion de nuestro spin sin flipar a la energia
        float ene0=-Mptr[n]*local_field;

        // contribucion a la energia de nuestro spin flipado
        //int enel=M[n]*vecinos;
        float enel=Mptr[n]*local_field;

        // metropolis: aceptar flipido solo si r < exp(-(enel-ene0)/temp)
        float p=exp(-(enel-ene0)/T);

        // numero random entre [0,1] uniforme
        //float r=float(rand())/RAND_MAX;->philox
        float rn = uniform(n, seed, t);

        // metropolis update segun regla de aceptancia
        return (rn<p)?(-Mptr[n]):(Mptr[n]);
    }
};
```

Cada thread se encarga de hacer el metropolis update de un sitio, calculando el campo local debido a los vecinos.

¡Crucial tener buenos números!

Conclusiones

- Usar paralelización (GPU o CPU) para simulaciones de Monte Carlo en sistemas de spins con interacciones de corto alcance no es una opción, **es un deber**.
- Las características del **RNG** a utilizar y su **conveniencia**, deben ser testeadas en cada caso.
 - La técnica generalizada del **chequerboard**, combinada con un **esquema de frames** según la cantidad de RNGs independientes, es "*the way to go*".
- **RNGs recomendados**: LCG64-random (velocidad), MWC (velocidad + independencia de secuencias, limitadas), Philox (versatilidad)

Ejemplos de la clase

- **Curand:** como usar device y host API en CUDA.
- **SimplePi:** como usar Random123 philox, en host y device, portable CUDA, CPP, OMP.
- **Mini-Ising:** Monte Carlo del modelo de Ising2d red cuadrada
- **Numba y cupy.** ICNPG_Curand.ipynb