

# Programming Graphics Processing Units (GPUs)

Lokmane ABBAS TURKI

lokmane.abbas\_turki@sorbonne-universite.fr

November 2018

## Plan

### Parallel architecture evolution

- From parallel to sequential
- From sequential to parallel
- Parallel efficiency laws

### CUDA, first steps

- Install CUDA and documentation
- Device query, Hello World! and Built-in variables
- Addition of two arrays: CPU vs. GPU
- Basic Monte Carlo (MC)

### Shared/registers optimization for MC

- Shared replacing global
- Registers replacing shared
- Threads/lanes communication

### Further optimizations beyond MC

- Using host memory
- Concurrency and asynchronous execution

### Real applications

- MC for Local volatility
- Various applications of Batch computing



## Plan

### Parallel architecture evolution

From parallel to sequential  
From sequential to parallel  
Parallel efficiency laws

### CUDA, first steps

Install CUDA and documentation  
Device query, Hello World! and Built-in variables  
Addition of two arrays: CPU vs. GPU  
Basic Monte Carlo (MC)

### Shared/registers optimization for MC

Shared replacing global  
Registers replacing shared  
Threads/lanes communication

### Further optimizations beyond MC

Using host memory  
Concurrency and asynchronous execution

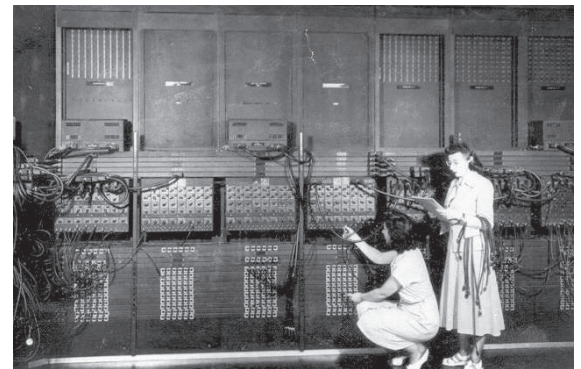
### Real applications

MC for Local volatility  
Various applications of Batch computing

## From 1946 to 70s

## ENIAC 1946

- ▶ Was the first electronic general-purpose machine
- ▶ Was a parallel machine
- ▶ Later became the first Von Neumann machine
- ▶ Was used for Monte Carlo simulation
- ▶ Although developed for ballistic research, it was first used for hydrogen bomb computations

Parallel machines  
till the 70s

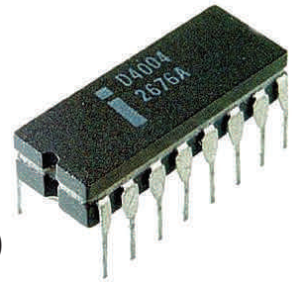
- ▶ Continued to exist as the real solution for heavy computations
- ▶ Used by specialists and dedicated essentially to military applications
- ▶ Some well known: ILLIAC IV (1971) and Cray 1 (1976)



## From 70s to 2000

## Becoming sequential

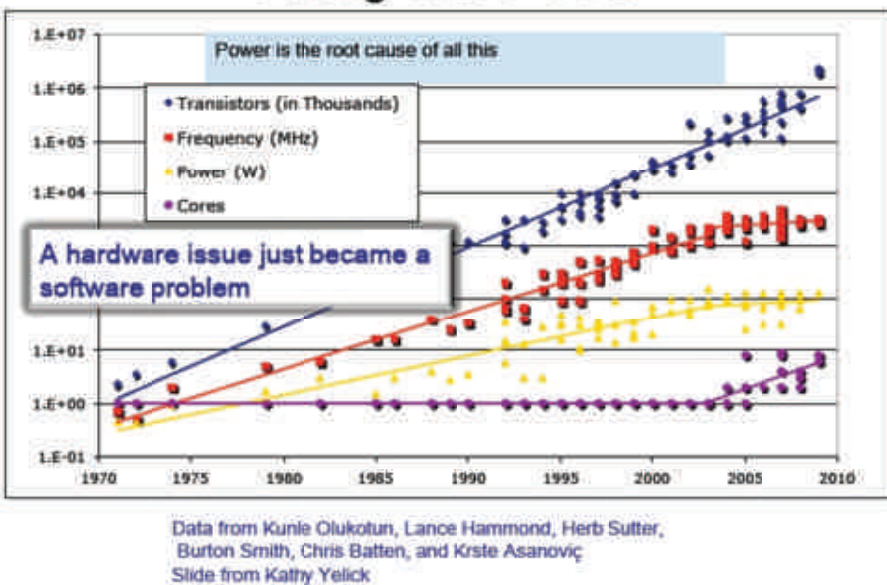
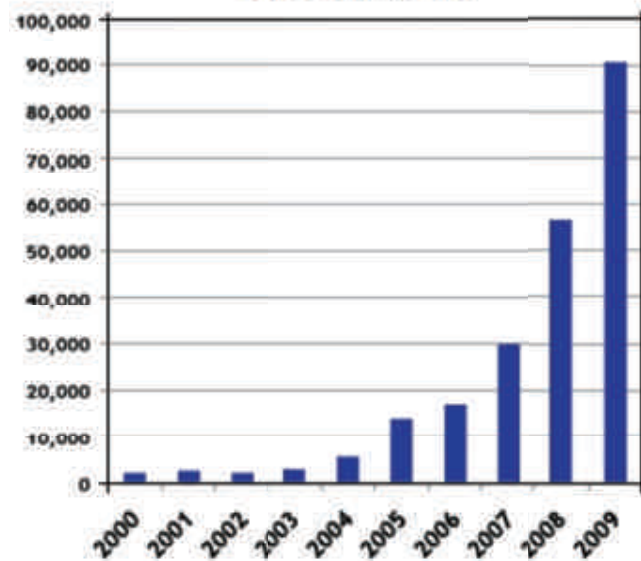
- ▶ Tradic (1954): The first transistor machine
- ▶ Intel 4004 (1971): Commercialization of the first microprocessor
- ▶ Amortizing the production costs by selling to the large public
- ▶ RAM became affordable (70s-80s) and used in microcomputers
- ▶ The memory hierarchy (Registers, cache, RAM, Hard Disc) is essential in computers
- ▶ The Moore's Law was satisfied on one core: Doubling the operating frequency each 18 months period



## Scientific simulation

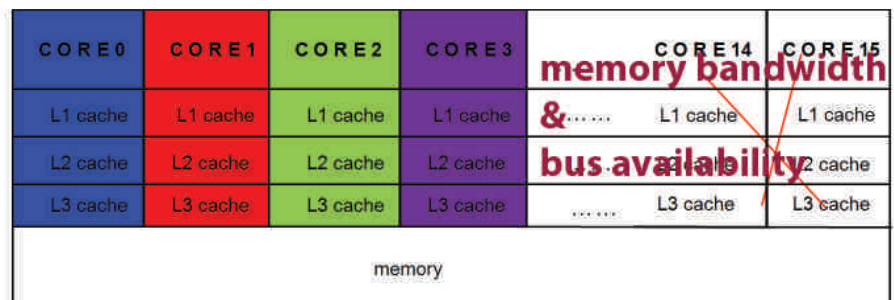
- ▶ Caltech Cosmic Cube (1981): Proposing a parallel computer at a reasonable cost
- ▶ From 1980 to 2000: Connection of serial machines to increase performance
- ▶ Difficulties due to multiplicities of platforms, inefficient inter-machine communication and insufficient documentation
- ▶ Applied mathematics expanded in the world of serial resolution of PDEs

## Reaching the limit

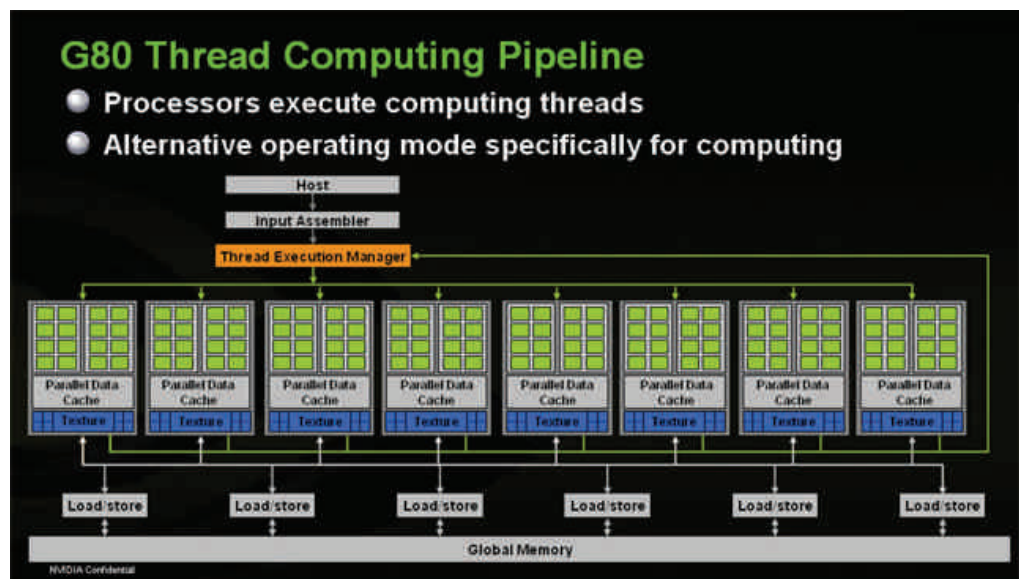
Performance Has Also Slowed,  
Along with PowerAverage Number of Cores Per  
Supercomputer

## Architecture overview

Sandia National  
Laboratories  
16 cores =? 2  
cores

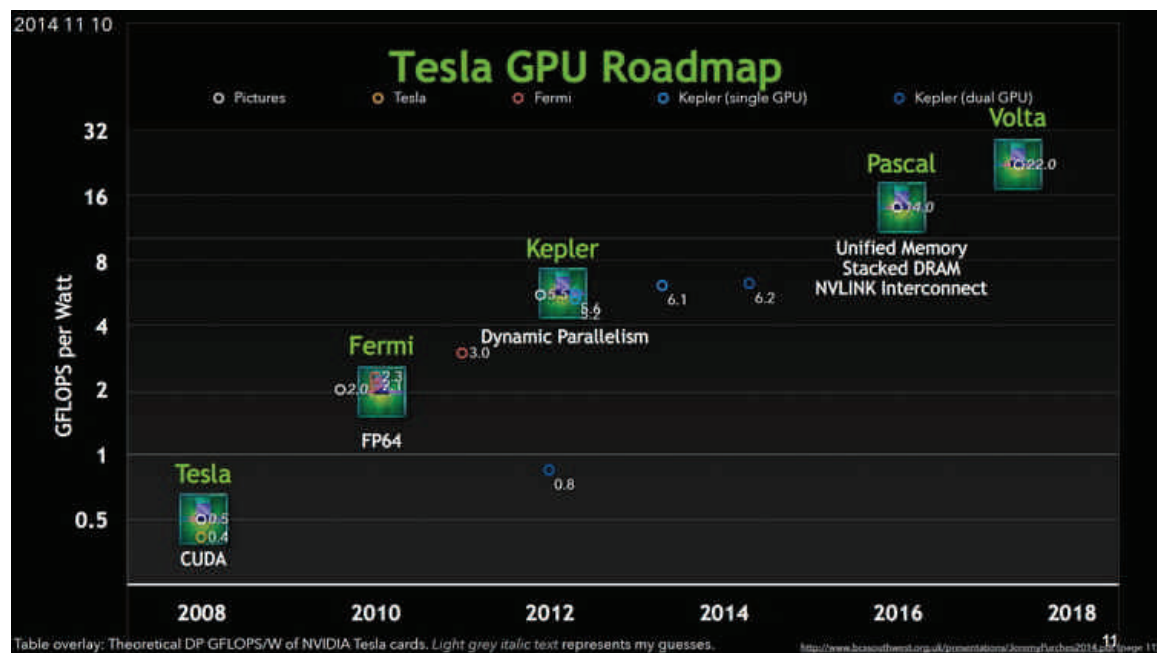


The limit  
architecture!  
GPU (Graphic  
Processing Unit)



## Efficiency and programmability

Computational capabilities



Programming languages

- OpenCL: Low level language, can be implemented on all cards.
- CUDA: Less low level than OpenCL, dedicated to Nvidia cards.
- OpenACC (came from OpenHMPP): A directives language, its use does not require to rewrite the CPU code.



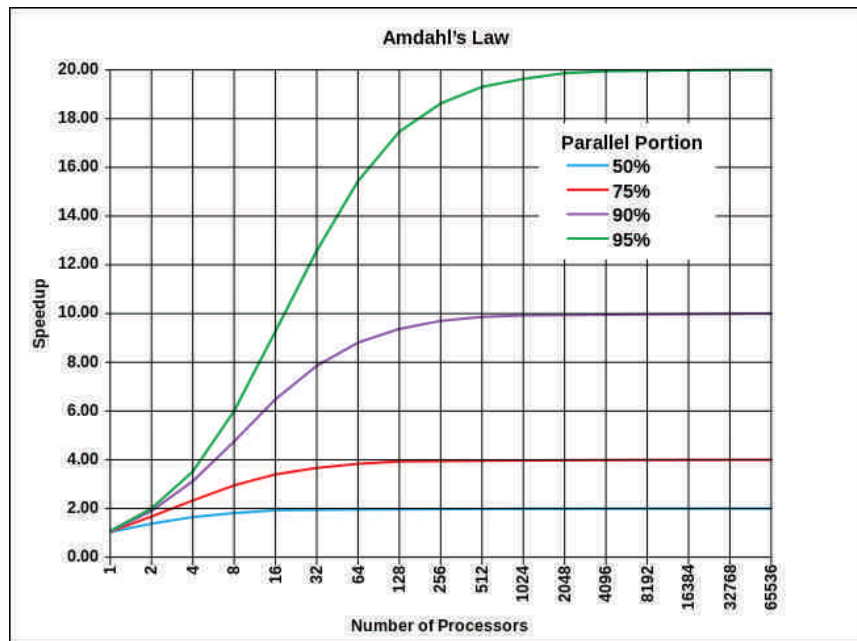
## Amdahl's law

For a fixed problem

$$T(P) = T(1) \left( \alpha + \frac{1 - \alpha}{P} \right), \quad S(P) = \frac{T(1)}{T(P)} = \frac{1}{\alpha + \frac{1 - \alpha}{P}}, \quad (1)$$

$\alpha$ : the fraction of the algorithm that is purely serial.

From Wikipedia



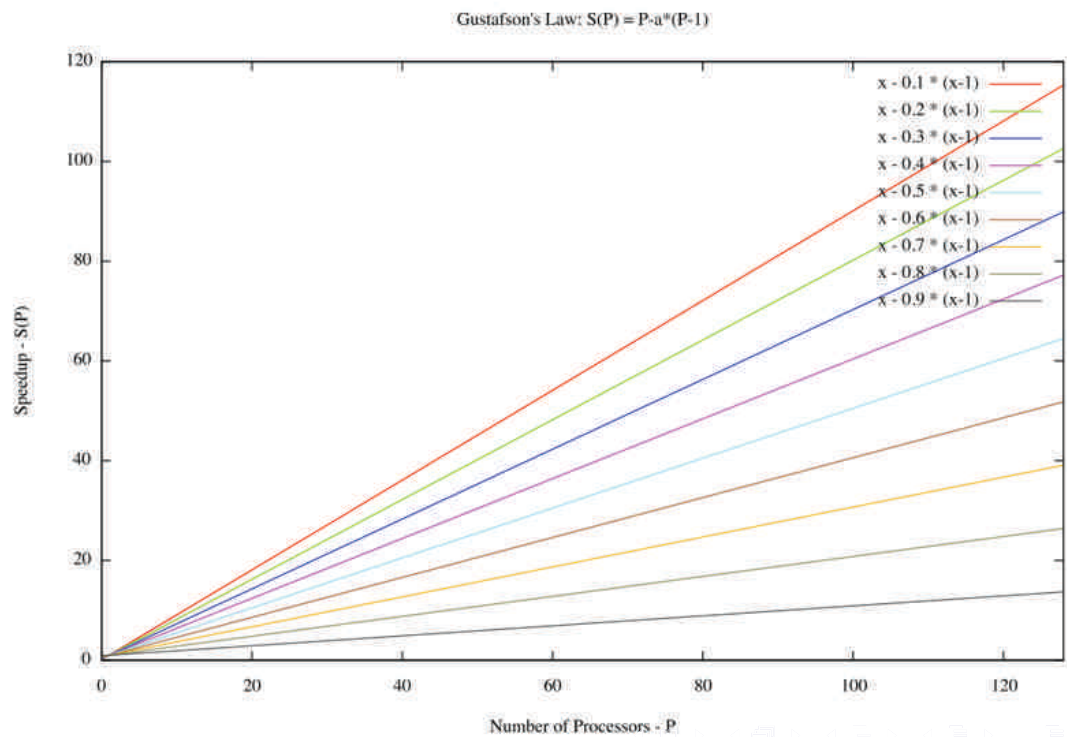
## Gustafson's law

Making it bigger

$$T(1) = (\alpha + [1 - \alpha]P)T(P), \quad S(P) = \frac{T(1)}{T(P)} = P - \alpha(P - 1), \quad (2)$$

$\alpha$ : the fraction of the algorithm that is purely serial.

From Wikipedia



## Plan

### Parallel architecture evolution

From parallel to sequential  
From sequential to parallel  
Parallel efficiency laws

### CUDA, first steps

Install CUDA and documentation  
Device query, Hello World! and Built-in variables  
Addition of two arrays: CPU vs. GPU  
Basic Monte Carlo (MC)

### Shared/registers optimization for MC

Shared replacing global  
Registers replacing shared  
Threads/lanes communication

### Further optimizations beyond MC

Using host memory  
Concurrency and asynchronous execution

### Real applications

MC for Local volatility  
Various applications of Batch computing

## Install CUDA on Linux machines

- ▶ gcc/g++ should be already available on your machine
- ▶ Install CUDA: <https://developer.nvidia.com/cuda-downloads>
- ▶ Disabling Secure Boot on UEFI (BIOS)
- ▶ Add /usr/local/cuda/bin to PATH and /usr/local/cuda/lib64 to LD\_LIBRARY\_PATH

## Install CUDA on Windows machines

- ▶ Install Visual Studio 2017 Community with C/C++ tools: <https://visualstudio.microsoft.com/fr/downloads/>
- ▶ Install CUDA: <https://developer.nvidia.com/cuda-downloads>
- ▶ Disabling Secure Boot on UEFI (BIOS)
- ▶ Add the address of cl compiler to Path
- ▶ Perform register changes explained at 7:40 in the video <https://www.youtube.com/watch?v=8NtHDkUoN98>

**Important!** **Very often** use the documentation provided by NVIDIA, in particular:

- ▶ **CUDA\_C\_Programming\_Guide**: Necessary document for the CUDA language handling and global understanding of the hardware architecture of the GPU
- ▶ **CUDA\_Runtime\_API**: Document describing the CUDA functions that allow to program the GPU

## Compilation + Execution

- ▶ Compile `DevQuery.cu` using `nvcc DevQuery.cu -arch=sm_50 -o DQ`
- ▶ Execute DQ using `./DQ` on Linux machines and using `DQ` on Windows machines

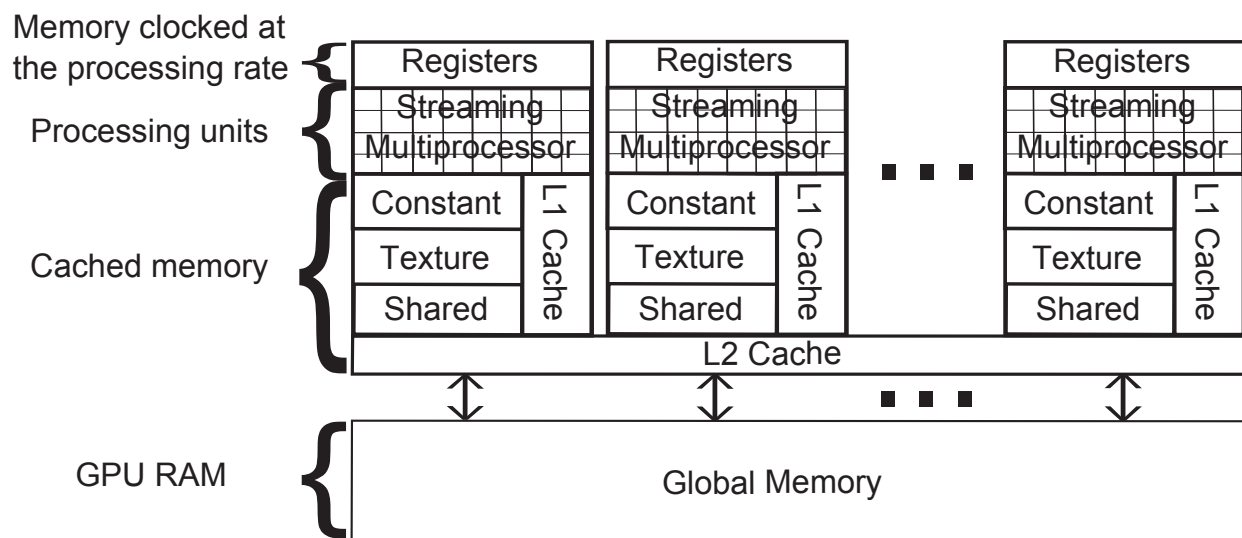
## Use documentation

- ▶ Get the specifications of `cudaGetDeviceCount` and of `cudaGetDeviceProperties` in **CUDA\_Runtime\_API**
- ▶ See how they can be used from the examples of **CUDA\_C\_Programming\_Guide**

**First code** Write in the `main` function of `DevQuery.cu` the appropriate code that

- ▶ Displays the number of available GPUs
- ▶ Give the properties of GPUs
- ▶ How could you catch execution errors using `testCUDA`?

## GPU architecture



### Hardware software equivalence

- ▶ Streaming processor: Executes threads
- ▶ Streaming multiprocessor: Executes blocks

### Built-in variables

Known within functions executed on GPU: threadIdx.x, blockDim.x, blockIdx.x, gridDim.x

## Always with DevQuery.cu

### Hello World!

- ▶ See in CUDA documentation how `cudaDeviceSynchronize` and `printf` work
- ▶ `Printf Hello World!` in `empty_k`
- ▶ Use `cudaDeviceSynchronize` just after `empty_k` call in the main function
- ▶ Call `empty_k` with `<<<1, 1>>>`, then with `<<<1, 8>>>` and with `<<<8, 1>>>`

### Built-in variables

- ▶ Instead of Hello World!, display the built-in variables
- ▶ Execute with `<<<1, 1>>>`, `<<<1, 32>>>`, `<<<1, 33>>>`, `<<<32, 1>>>` and with `<<<8, 4>>>`
- ▶ Propose a linear combination of `threadIdx.x` and `blockIdx.x` that provides successive different values

## Function declaration and calling

### Standard C functions

The same as for C or C++ programming

### Kernel functions

- ▶ Called by the CPU and executed on the GPU
- ▶ Declared as `__global__ void myKernel (...) { ...; }`
- ▶ Called standardly by  
`myKernel<<<numBlocks, threadsPerBlock>>>(...);`  
 where  
   numBlocks should take into account the number of multiprocessors  
   threadsPerBlock should take into account the warp size
- ▶ Dynamic parallelism: kernels can be called within kernels by the GPU and executed on the GPU

### device functions

- ▶ Called by the GPU and executed on the GPU
- ▶ Declared as  
   `__device__ void myDivFun (...) { ...; }`  
   `__device__ float myDivFun (...) { ...; }`
- ▶ Called simply by `myDivFun(...)` but only within other device functions or kernels



**On CPU** We want to add two large arrays of integers and put the result in a third one.

- ▶ Create a new file `.cu`, include `stdio.h` and `timer.h` in the top
- ▶ In the main function, allocate three arrays `a`, `b`, `c` using `malloc`
- ▶ Assign some values to `a` and `b`
- ▶ Using functions defined in `timer.h`, compute the execution time of adding `a` and `b`
- ▶ Free the CPU memory using `free`

**On GPU** We keep the same CPU code. For given values of *numBlocks* and *threadsPerBlock*, we want to perform an addition of  $\text{numBlocks} \times \text{threadsPerBlock}$  integers

- ▶ Allocate `aGPU`, `bGPU`, `cGPU` on the GPU using `cudaMalloc`
- ▶ Transfer the values of `a`, `b` to `aGPU`, `bGPU` using `cudaMemcpy`
- ▶ Write the kernel that adds `aGPU` to `bGPU` and return the result in `cGPU`
- ▶ Copy `cGPU` to `c`
- ▶ Compute the execution time
- ▶ Propose a trick to deal with sizes different from  $\text{numBlocks} \times \text{threadsPerBlock}$

## Compute the execution time on GPU with

```
float TimeVar;
cudaEvent_t start, stop;
testCUDA(cudaEventCreate(&start));
testCUDA(cudaEventCreate(&stop));
testCUDA(cudaEventRecord(start, 0));

/*****

To compute the execution time of this part of the code

*****/

testCUDA(cudaEventRecord(stop, 0));
testCUDA(cudaEventSynchronize(stop));
testCUDA(cudaEventElapsedTime(&TimeVar, start, stop));
testCUDA(cudaEventDestroy(start));
testCUDA(cudaEventDestroy(stop));
```

**Pricing European**  $F_t = e^{-r(T-t)} E(f(S_s, t \leq s < T) | \mathfrak{F}_t), t \in [0, T)$  where

- ▶  $f$  is the contract's payoff
- ▶  $T$  is the contract's maturity
- ▶  $r$  is the risk-free interest rate

$X = f(S_{s, t \leq s < T})$  We want to simulate  $F(0, y) = E(X)$  using a family  $\{X_i\}_{i \leq n}$  of i.i.d  $\sim X$

- ▶ **Strong law of large numbers:**

$$P \left( \lim_{n \rightarrow +\infty} \frac{X_1 + X_2 + \dots + X_n}{n} = E(X) \right) = 1$$

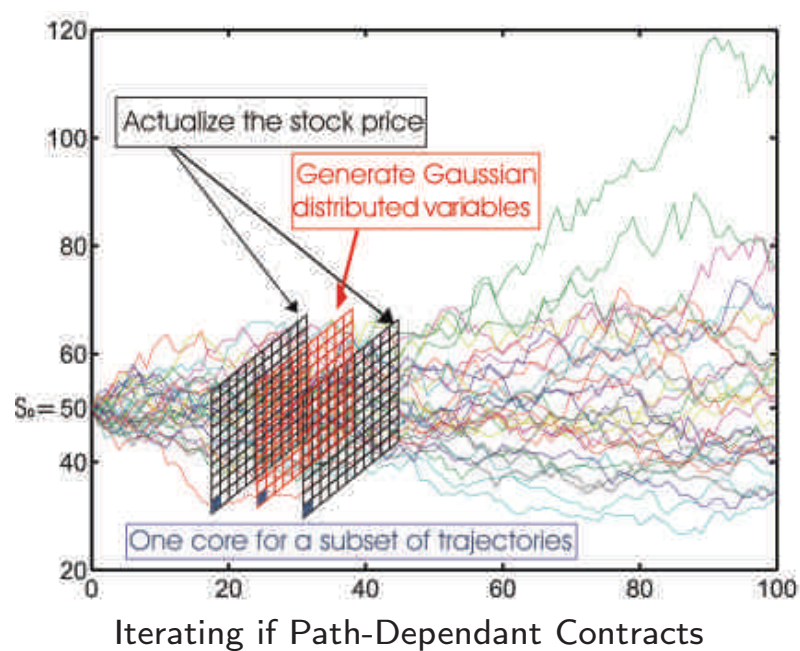
- ▶ Denoting  $\epsilon_n = E(X) - \frac{X_1 + X_2 + \dots + X_n}{n}$
- ▶ **Central limit theorem:**

$$\frac{\sqrt{n}}{\sigma} \epsilon_n \rightarrow G \sim \mathcal{N}(0, 1)$$

- ▶ There is a 95% chance of having:

$$|\epsilon_n| \leq 1.96 \frac{\sigma}{\sqrt{n}}$$

## European path-dependant pricing



For each time step:

- 1 Random number generation (if parallelized)
- 2 Stock price actualization
- 3 Compute the payoff

## General Form of linear RNGs

Without loss of generality:

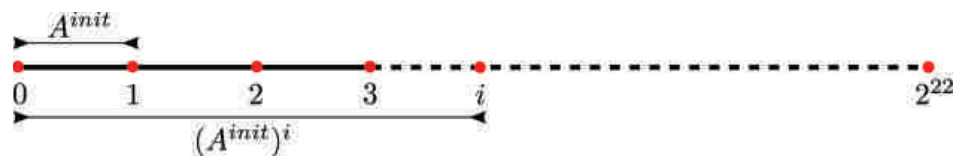
$$X_n = (AX_{n-1} + C) \bmod(m) = (A : C) \begin{pmatrix} X_{n-1} \\ \vdots \\ 1 \end{pmatrix} \bmod(m) \quad (3)$$

## Parallel-RNG from Period Splitting of One RNG

Pierre L'Ecuyer proposed a very efficient RNG (1996) which is a CMRG on 32 bits: Combination of two MRG with  $lag = 3$  for each MRG.

- \* Very long period  $\sim 2^{185}$

$$x_n = (a_1 x_{n-1} + a_2 x_{n-2} + a_3 x_{n-3}) \bmod(m)$$



## Parallel-RNG from Parameterization of RNGs

Same parallelization as SPRNG Prime Modulus LCG.

- \* The same RNG with different parameters "a":

$$x_n = ax_{n-1} + c \bmod(m)$$

## Bullet option in Black & Scholes model

- Price at  $t = 0$   $F_0 = e^{-r_0 T} E \left( (S_T - K)_+ 1_{\{I_T \in [P_1, P_2]\}} \right)$  with  $I_t = \sum_{T_i \leq t} 1_{\{S_{T_i} < B\}}$
- ▶  $K, T$  are respectively the contract's strike and maturity
  - ▶  $0 < T_1 < \dots < T_M = T$  is a predetermined schedule
  - ▶ The barrier  $B$  should be crossed  $I_T$  times  $\in \{P_1, \dots, P_2\} \subset \{0, \dots, M\}$
  - ▶  $r_0$  risk-free rate

- Black & Scholes model For  $0 \leq s < t \leq T$ ,  $S_t \equiv S_s \exp \left( (r_0 - \sigma^2/2)(t-s) + \sigma\sqrt{t-s}G \right)$  and  $S_0 = x_0$
- ▶  $\sigma$  is the volatility
  - ▶  $G$  is independent from  $S_s$  and has a standard random distribution
  - ▶  $x_0$  is the initial spot price of  $S$  at time 0

- Complete MC.cu After memory allocation and free, uncomment the kernel call then fill it with the appropriate call of BoxMuller\_d and of BS\_d.