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 volatiliy
- 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

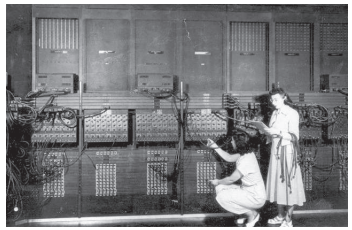
Further optimizations beyond MC
Using host memory
Concurrency and asynchronous execution

Real applications
MC for Local volatiliy
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

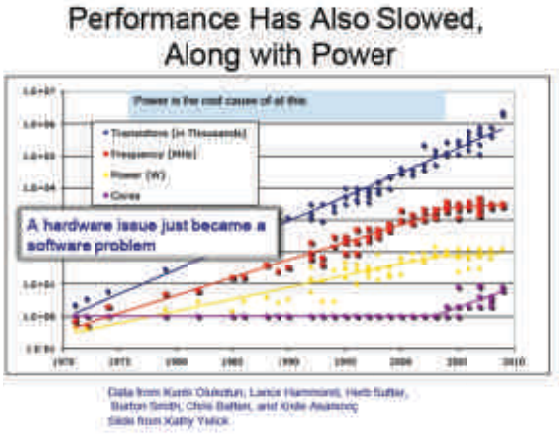
Becoming sequential

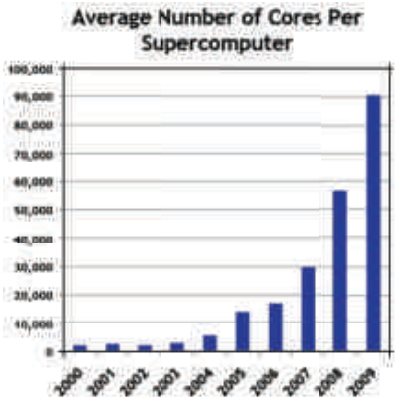
- Tradic (1954): The first transistor machine
- Intel 4004 (1971): Commercialization of the first microprocessor
- Amortizing the production costs by selling to the large public
- RAM became affordable (70s-80s) and used in microcomputers
- The memory hierarchy (Registers, cache, RAM, Hard Disc) is essential 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

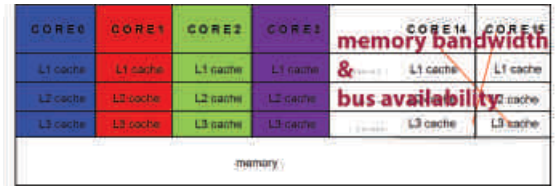
Reaching the limit



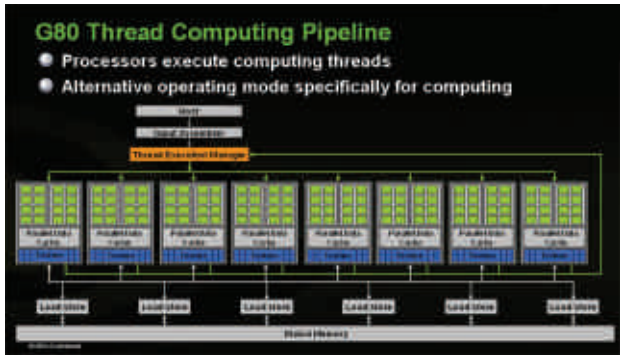


Architecture overview

Sandia National Laboratories 16 cores =? 2 cores

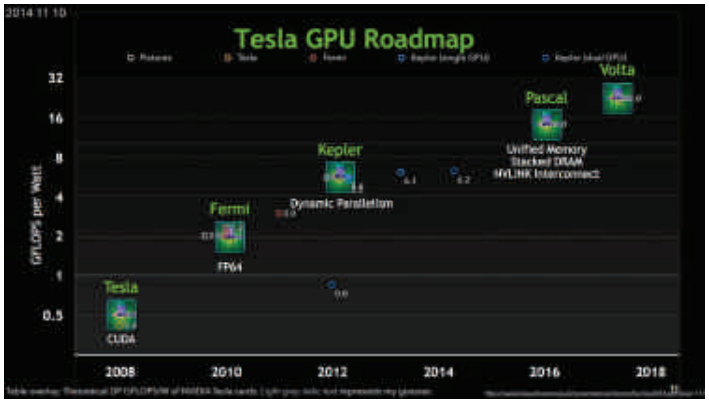


The limit architecture! GPU (Graphic Processing Unit)



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.

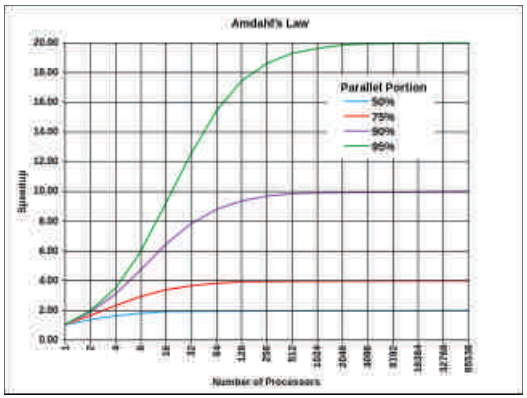
Amdahl's law

For a fixed problem

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

α: the fraction of the algorithm that is purely serial.

From Wikipedia



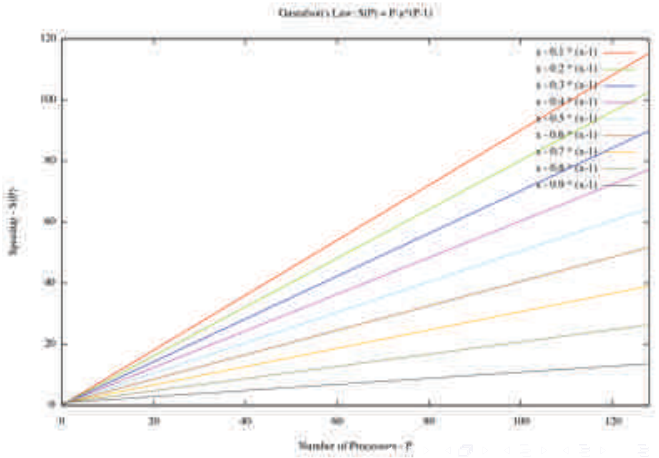
Gustafson's law

Making it bigger

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

α: 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 volatiliy
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

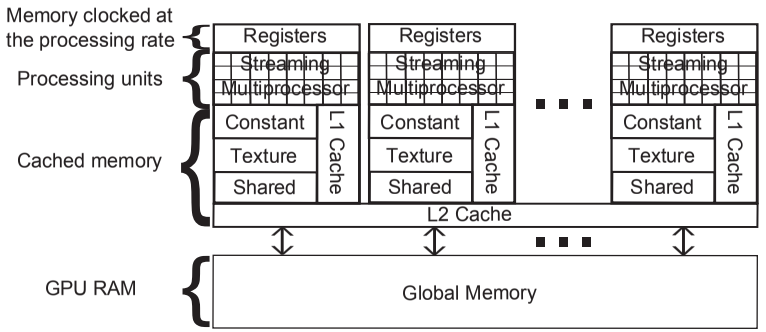
- Get the specifications of `cudaGetDeviceCount` and of `cudaGetDeviceProperties` in **CUDA_Runtime_API**
- See how they can be used from the examples of **CUDA_C_Programming_Guide**

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: Excutes threads
- Streaming multiprocessor: Executes blocks

Built-in variables

Known within functions excuted 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 *numBlocks* $\times$ *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 *numBlocks* $\times$ *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**:

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

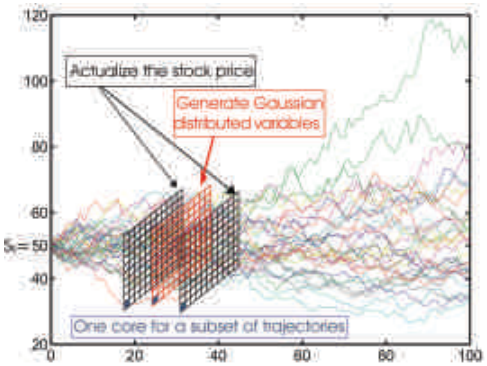
- Denoting $\epsilon_n = E(X) - \frac{X_1 + X_2 + ... + X_n}{n}$
- **Central limit theorem**:

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

- There is a 95% chance of having:

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

European path-dependant pricing



Iterating if Path-Dependant Contracts

For each time step:

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

General Form of linear RNGs

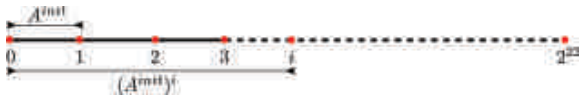
Without loss of generality:

$$X_n = (AX_{n-1} + C)\ mod(m) = (A\ :\ C) \begin{pmatrix} X_{n-1} \\ \dots \\ 1 \end{pmatrix}\ mod(m) \qquad (3)$$

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 $\sim 2^{185}$

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



Parallel-RNG from Parameterization of RNGs

* Same parallelization as SPRNG Prime Modulus LCG.
* The same RNG with different parameters "a":

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

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
- $0 < T_1 < ... < T_M = T$ is a predetermined schedule
- The barrier B should be crossed I_T times $\in \{P_1, ..., P_2\} \subset \{0, ..., M\}$
- 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 $S_0 = x_0$

- σ 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

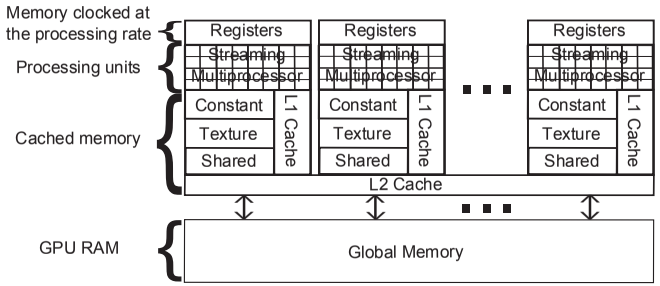
Complete `MC.cu`

After memory allocation and free, uncomment the kernel call then fill it with the appropriate call of `BoxMuller_d` and of `BS_d`.

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 volatiliy
Various applications of Batch computing

Shared: Second fastest memory



Shared memory

- Cached memory visible to all threads of the same block
- Has a lifetime of a kernel
- Static allocation of arrays: `__shared__ float A[100];`
- Dynamic allocation of arrays: `extern __shared__ float A[];`
  kernel call: `myKernel<<<..., ..., 100*sizeof(float)>>>(...);`

In `MemComp` Replace the use of `aGlob` by a static shared array

In `MC_k` Replace as much as possible the use of global arrays by static shared arrays

Bullet option on an average of Black & Scholes

Price at $t = 0$

$$F_0 = e^{-r_0 T} E\left((S_T - K)_+ 1_{\{I_T \in [P_1, P_2]\}}\right) \text{ with } I_t = \sum_{T_i \leq t} 1_{\{S_{T_i} < B\}}$$

- K, T are respectively the contract's strike and maturity
- $0 < T_1 < ... < T_M = T$ is a predetermined schedule
- The barrier B should be crossed I_T times $\in \{P_1, ..., P_2\} \subset \{0, ..., M\}$
- r_0 risk-free rate

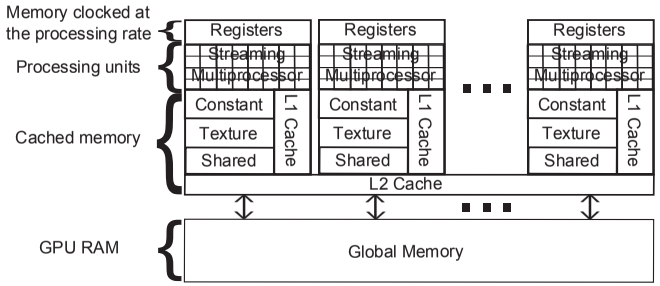
Multi-dimensional model

For $0 \leq s < t \leq T$, $S_t = 1/D \sum_{i=1}^{D} S_t^i$
$S_t^i \equiv S_s^i \exp\left((r_0 - \sigma^2/2)(t-s) + \sigma\sqrt{t-s} G_i\right)$ and $S_0^i = x_i$

- σ is the volatility
- Each G_i is independent from $(S_s^1, ..., S_s^D, G_1, ..., G_{i-1}, G_{i+1}, ..., G_D)$ and has a standard random distribution
- x_i is the initial spot price of S^i at time 0

Complete `MC.cu` Adapt the code for this multi-dimensional setting using dynamic shared arrays and test it with $D = 4$.

Registers: Fastest memory



Registers

- Divided homogeneously between threads of the same block
- Have a lifetime of a kernel
- Cannot be used for arrays

In `MemComp` Replace the use of `aGlob` by a register variable

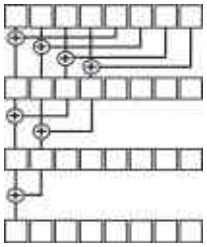
In `MC_k` Replace as much as possible the use of global arrays (shared arrays) by registers

Some facts

- Threads can access to any memory space of the shared memory of their block
- The synchronization barrier `__syncthreads();` ensures that threads of the same block wait for all other threads of the same block.
- Threads of different blocks cannot exchange values within the same kernel

Dot product exercise

- Store the product result in the shared memory then perform a reduction using the following scheme with a `__syncthreads();` at each step



- How `atomicAdd` operates? Use it for the reduction through blocks
- What happens when `atomicAdd` is used for the whole sum?

Some facts

- Threads in the same warp, called lanes, can access to any register associated to their warp
- Functions prefixed with `__shfl` are needed for this communication
- Lanes do not need synchronization

Syntax example

```
float __shfl_down(float var, unsigned int delta, int width);
```

- `var` is the value to be communicated
- `delta` is a lane translation index
- `width` is the number of involved threads $\leq$warpSize=32

Test this

```
__global__ void kernel(int *A){
    int idx = threadIdx.x + blockIdx.x*blockDim.x;
    int lane = threadIdx.x%warpSize;
    int loc;
    loc = A[idx];
    if(blockIdx.x<1 && lane==idx){
        printf("%d , %i, %i \n", lane, loc,
                __shfl_down(loc, 1, warpSize));
    }
}
```

Now, make the appropriate changes to `MC_k`

Benefits

- Makes the shared memory available for other tasks
- Registers are faster
- Does not have to synchronize between threads