Programming Graphics Processing Units (GPUs)

Who? Lokmane ABBAS TURKI
lokmane.abbas_turki@sorbonne-universite.fr

When? November 2018

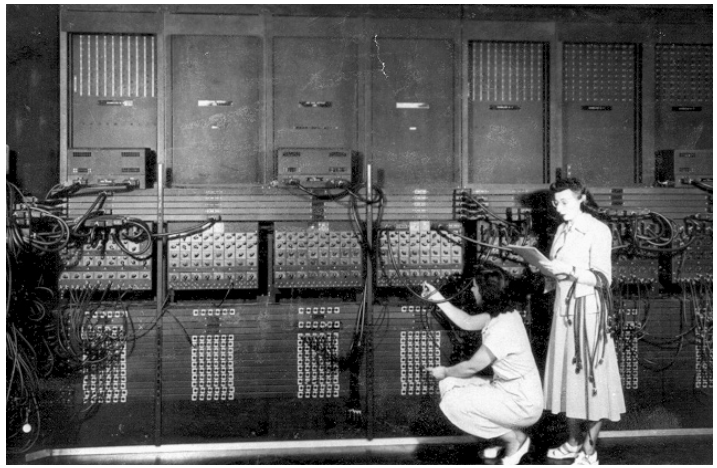
Plan

Plan

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



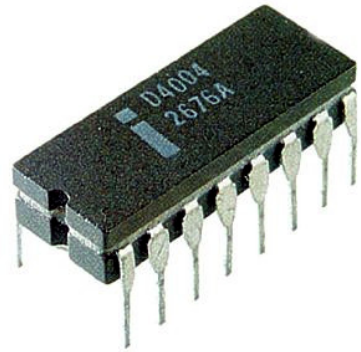
Parallel machines till the 70s

- 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

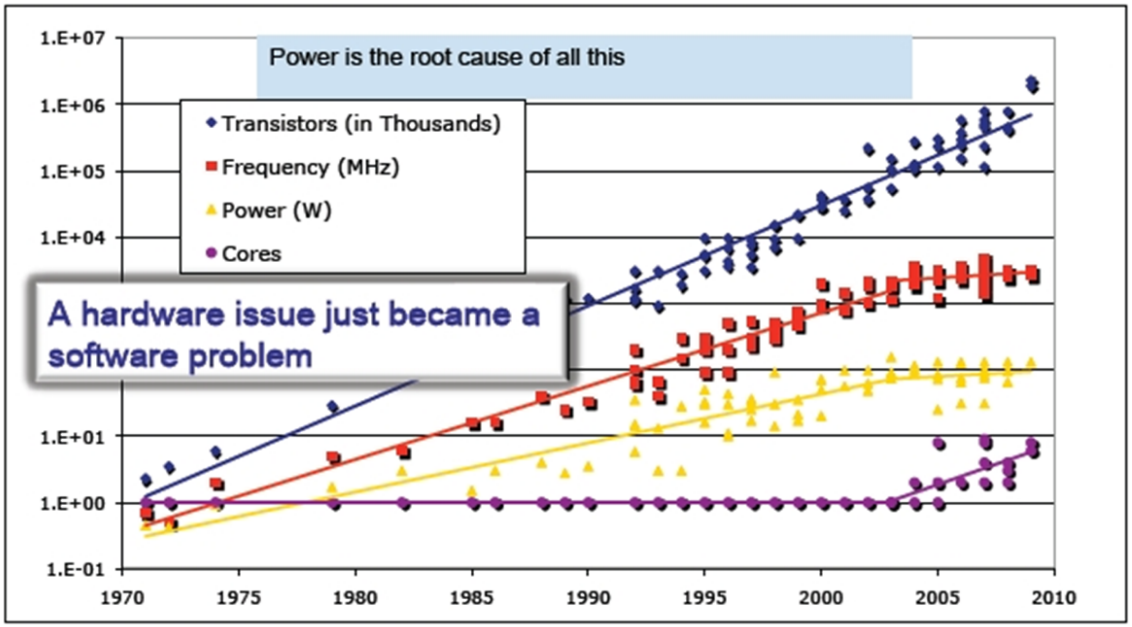


Scientific simulation

- 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

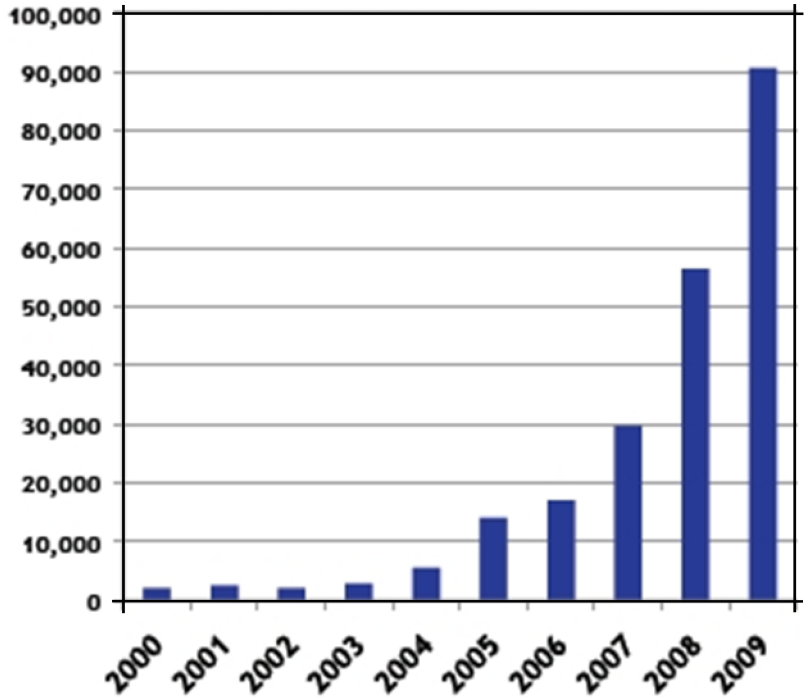
Reaching the limit

Performance Has Also Slowed, Along with Power



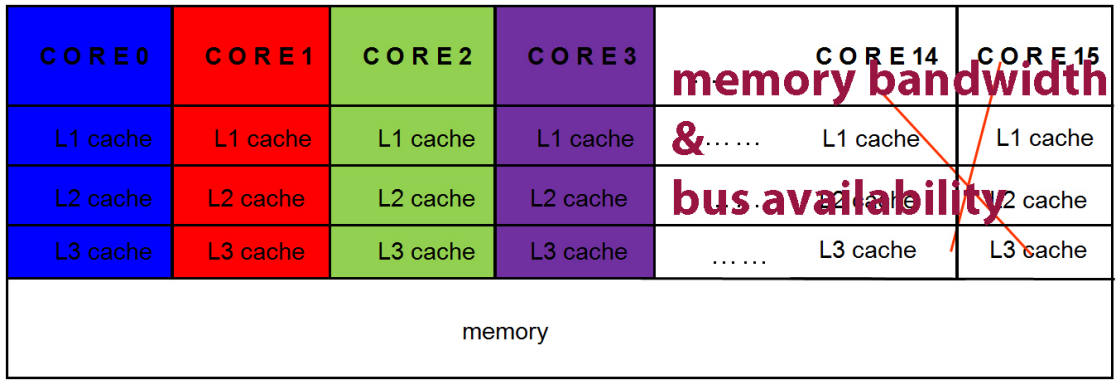
Data from Kunle Olukotun, Lance Hammond, Herb Sutter, Burton Smith, Chris Batten, and Krste Asanoviç
Slide from Kathy Yelick

Average Number of Cores Per Supercomputer

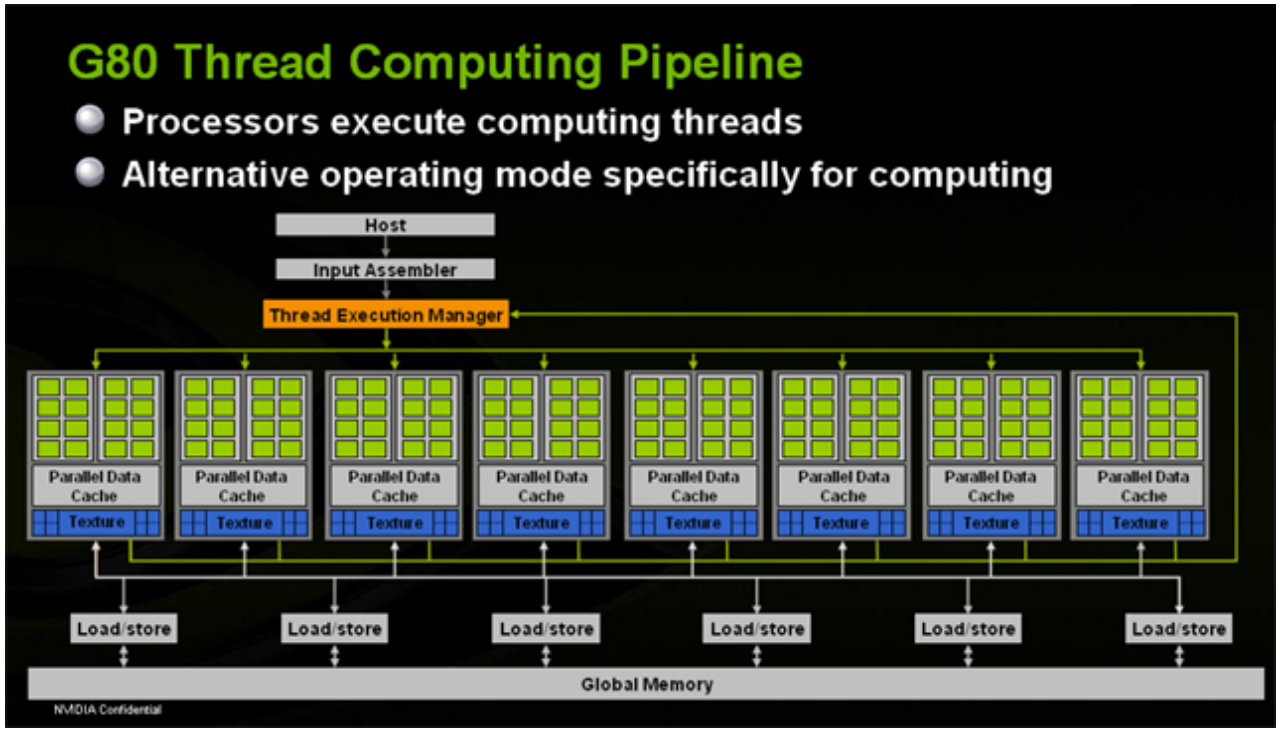


Architecture overview

Sandia National Laboratories
16 cores =? 2 cores

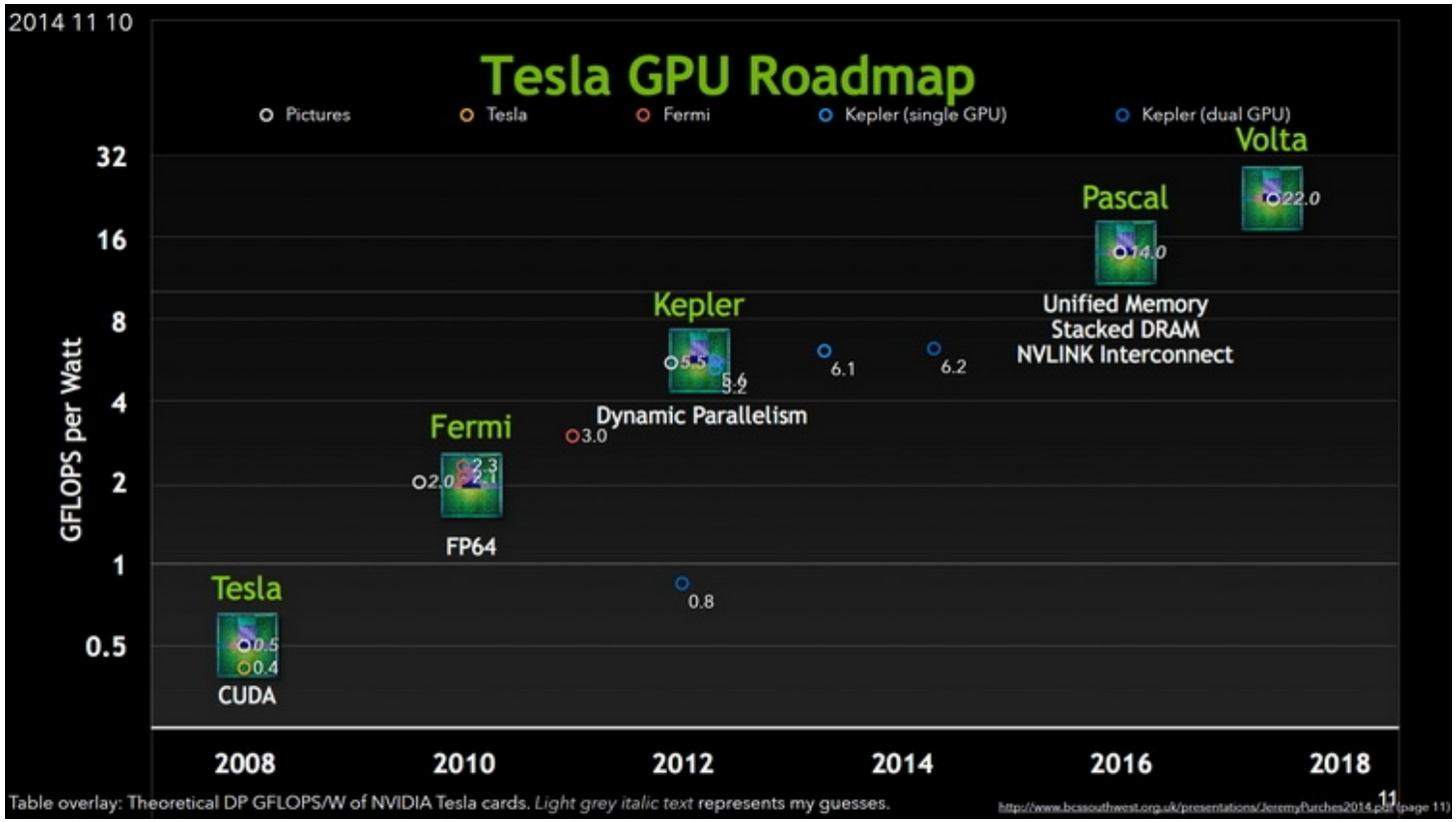


The limit architecture!
GPU (Graphic Processing Unit)



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.

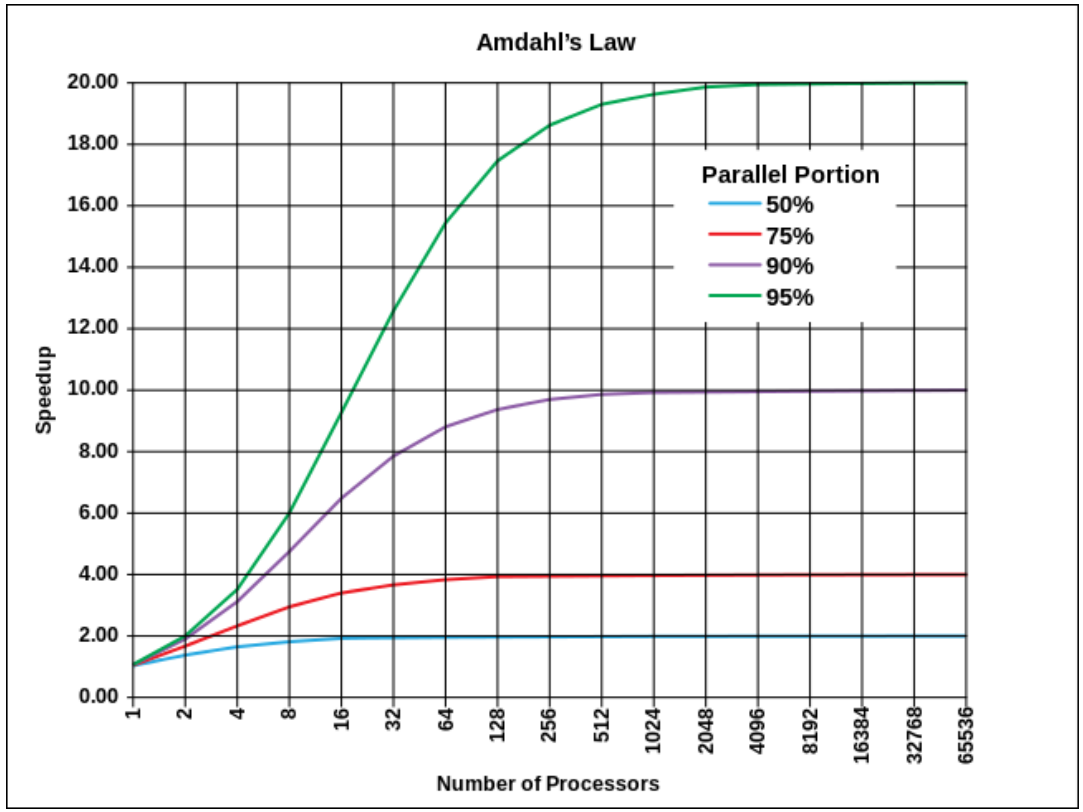
Amdahl's law

For a fixed problem

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

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

From Wikipedia



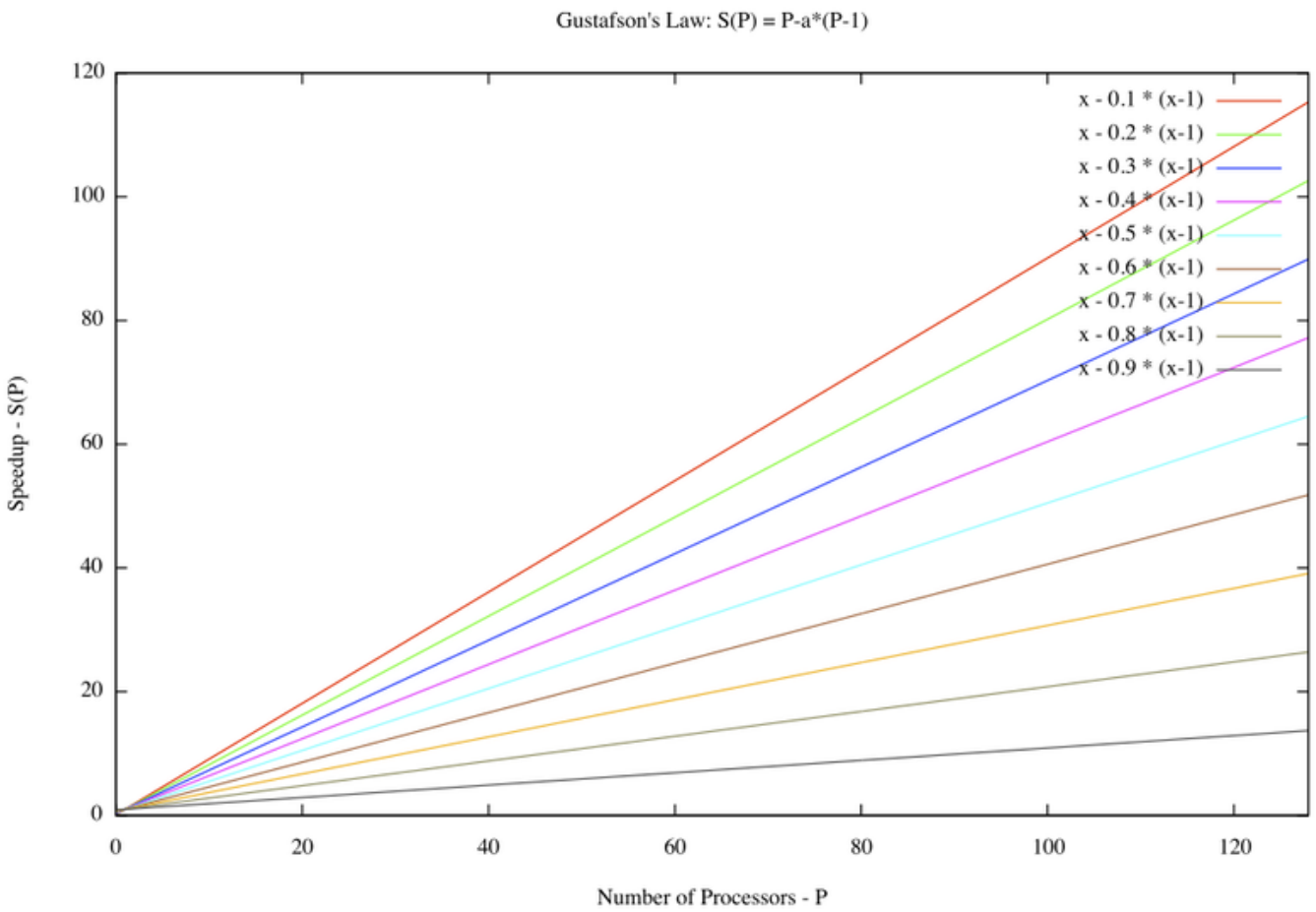
Gustafson's law

Making it bigger

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

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

From Wikipedia



Plan

PuTTY as a command console

- You should know your account name `hpcrise` and password
- `ssh ghome.metz.supelec.fr` to access to CentraleSupélec network
- `ssh term2.grid` to access to the GPU cluster
- `oarsub -p "cluster='cameron'" -q day -l nodes=1,walltime=4:00:00 -I` to get one GPU node during 4 hours
- Once finished, log out with successive `exit` or `Ctrl+d`

FileZilla Client to transfer files

Write your code on your local machine then transfer it on `ghome.metz.supelec.fr` using FileZilla Client.

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

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

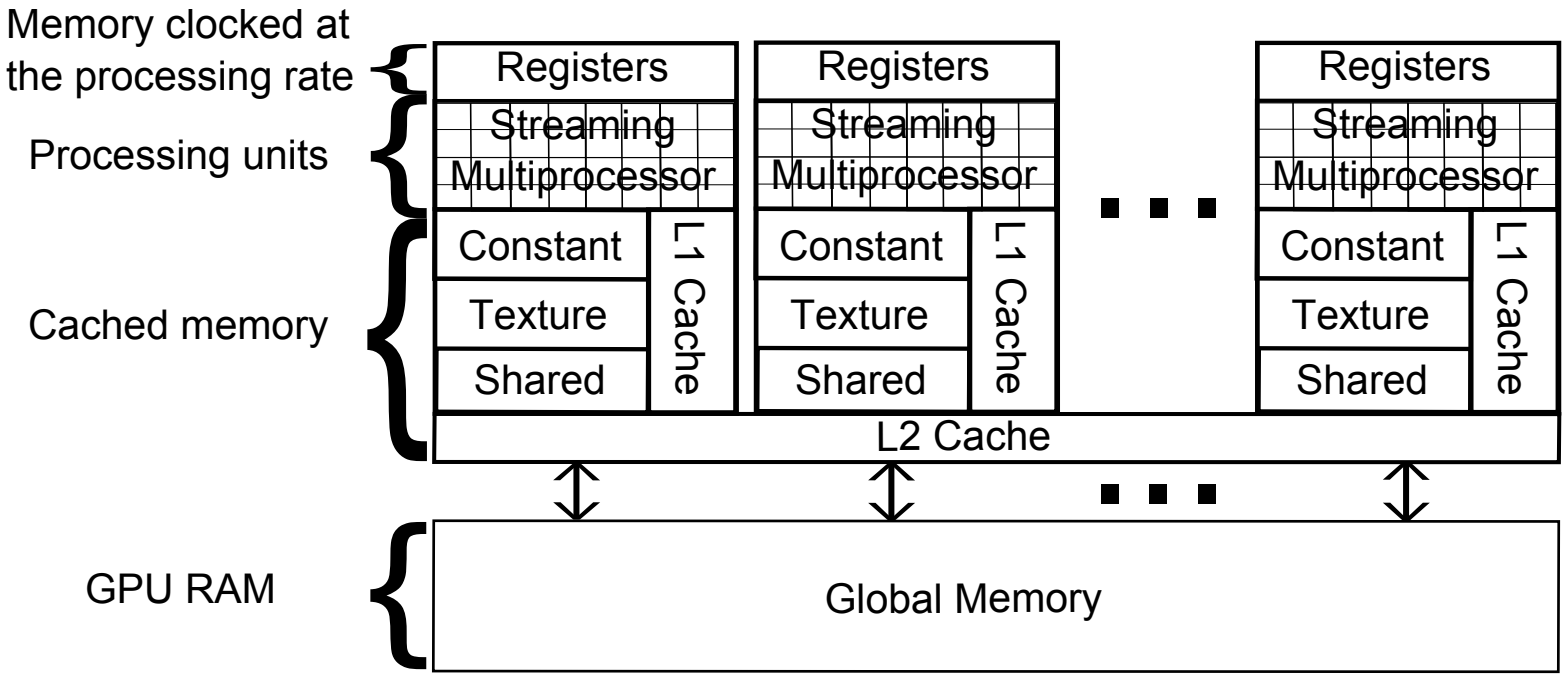
- Get the specifications of `cudaGetDeviceCount` and of `cudaGetDeviceProperties` in **CUDA_Runtime_API**
- See how they can be used from the examples of **CUDA_C_Programming_Guide**

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**:

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

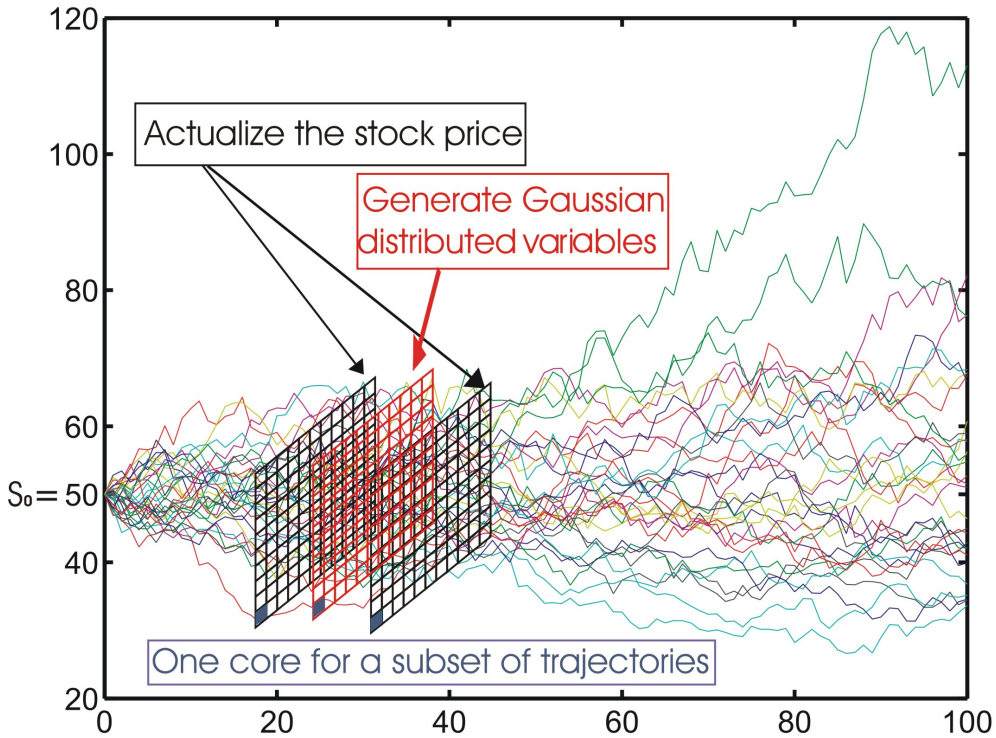
- Denoting $\epsilon_n = E(X) - \dfrac{X_1 + X_2 + ... + X_n}{n}$
- **Central limit theorem**:

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

- 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

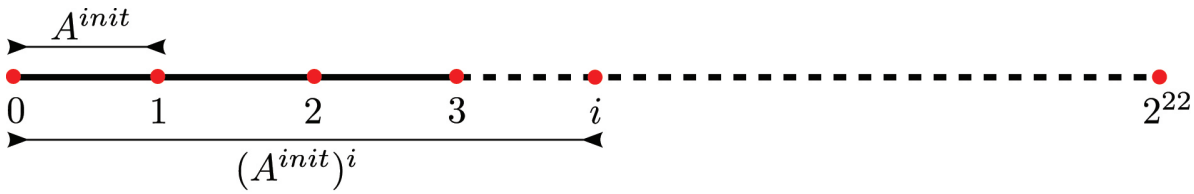
Without loss of generality:

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

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 above S 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 $S_0 = x_0$

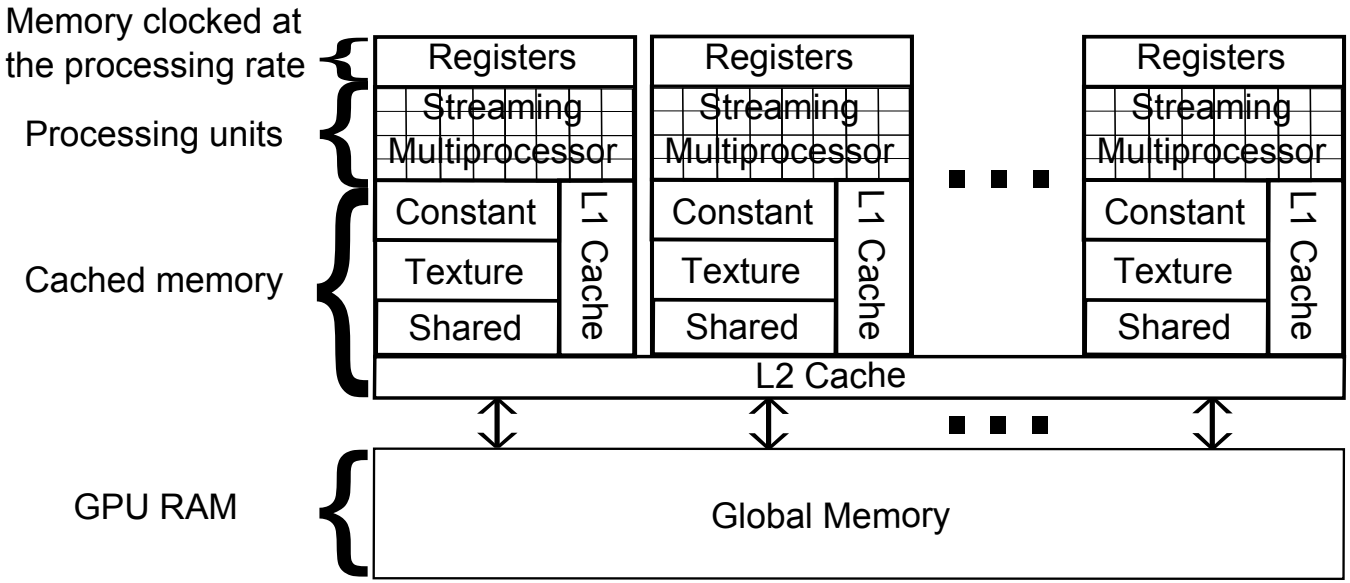
- σ is the volatility
- G is independent from S_s and has a standard random distribution
- x_0 is the initial spot price of S at time 0

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

Shared: Second fastest memory



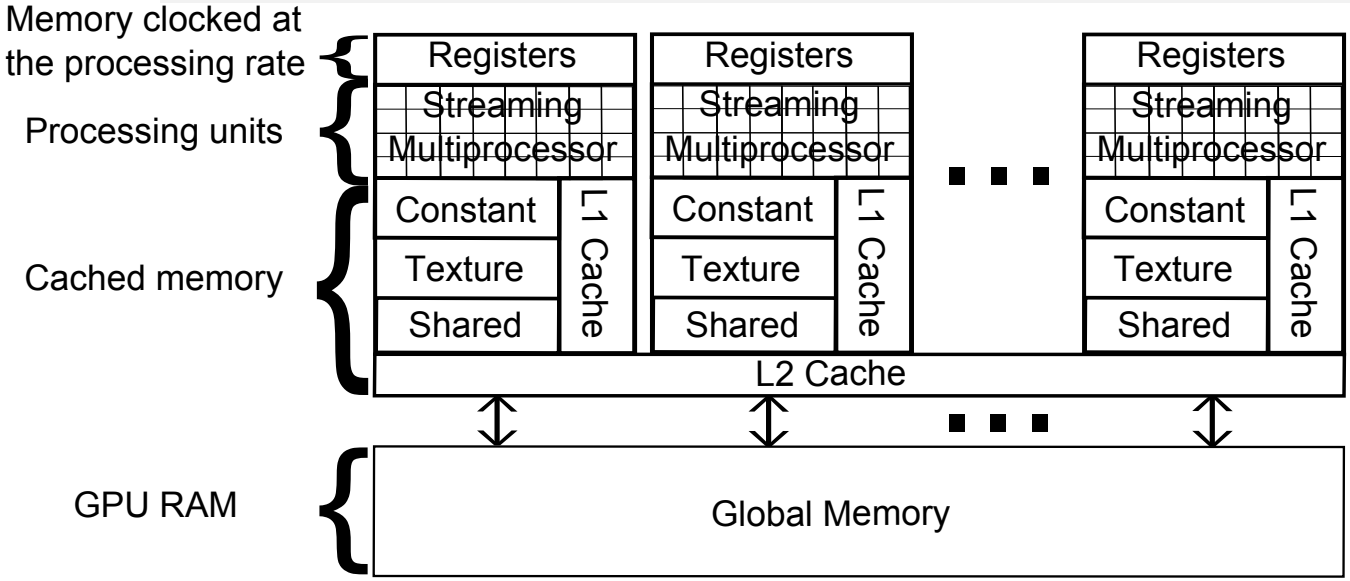
Shared memory

- 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 For the Black & Scholes process, replace the global array by static shared array

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`

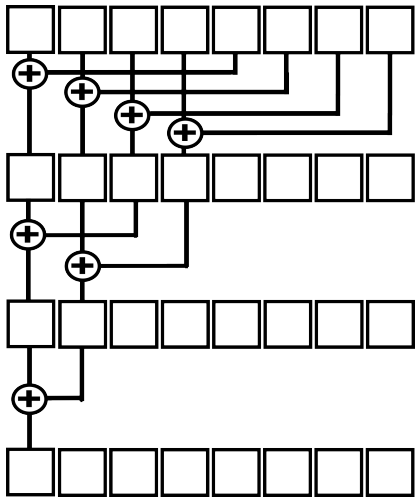
- Except for the Black & Scholes process, replace as much as possible the use of global arrays by registers
- For the Black & Scholes process, replace the static allocation of the shared memory by a dynamic one

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?

Exercise

Using

```
sum_k<<<NB, NTPB>>>(float *In, float *Out, int d)
```

we compute the sum `Out` of elements of a floating point array `In` of size d. In the following, when needed, we assume only static shared memory allocation, for instance:

```
__shared__ float InSh[64];
```

1. Define the kernel `sum_k` using the maximum number of threads to sum on `In` when we execute
   a) `sum_k<<<1, 1>>>(In, Out, 32);`
   b) `sum_k<<<1, 2>>>(In, Out, 32);`
   c) `sum_k<<<1, 4>>>(In, Out, 32);`
   d) `sum_k<<<1, 64>>>(In, Out, 128);`
2. Using question 1.d, define `sum_k` that sums on `In` when we execute `sum_k<<<2, 64>>>(In, Out, 256);`
3. Compare solution 2. to a sum that involves only atomicAdd.
4. From now on, we want to compute various sums stored in the same array pointed by `Out`. We sum on various arrays of the same size d stored in a concatenated way in `In`. Explain how the indices
   `int tidx = threadIdx.x%d; int Qt = (threadIdx.x-tidx)/d;`
   `int gbx = Qt + blockIdx.x*(blockDim.x/d);` can be used in `sum_k`.
5. Write `sum_k<<<40, 512>>>(In, Out, 256);` that batch computes various sums.

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

Plan

Standard host allocation vs. locked memory vs. mapped memory

Standard host allocation

- Memory space virtually limited by the machine's RAM
- The OS controls the memory space
- The data transfer is the slowest it can get

Locked allocation

- Memory space much smaller then the machine's RAM
- The OS loses some control on the memory space
- The data transfer is faster then standard allocation

Mapped allocation

- Memory space much smaller then the machine's RAM
- The OS loses some control on the memory space
- Double pointers: One used by CPU and the other one used by GPU.
- The **implicit** data transfer is the fastest it can get

How to implement?

Locked memory

- Make sure that the memory space to be locked is not too big
- Replace `malloc` by `cudaHostAlloc` and `free` by `cudaFreeHost`

Mapped allocation

- Make sure that the memory space to be locked is not too big
- Before calling any other function (at the beginning of main), one has to execute `cudaSetDeviceFlags(cudaDeviceMapHost)`
- The CPU allocation has to be done using `cudaHostAlloc` with `cudaHostAllocMapped` as an option.
- The GPU gets a pointer with `cudaHostGetDevicePointer`
- Make sure that the GPU had finished working on mapped memory thanks to `cudaDeviceSynchronize`.

Do this

- Write a code that compares the standard transfer to the locked memory transfer
- Compare the GPU use of mapped memory to the standard solution and to the locked memory solution

Using different streams of type cudaStream_t

For concurrency

- Create streams with `cudaStreamCreate`
- Execute concurrent kernels with different streams. For example:

```
myKernel1<<<..., ..., ..., stream1>>>(...);
myKernel2<<<..., ..., ..., stream2>>>(...);
```

- Make sure that streams finished processing with `cudaStreamSynchronize`
- Destroy streams with `cudaStreamDestroy`

For asynchronous execution

- Use `cudaHostAlloc` for allocation on the host memory
- Create streams with `cudaStreamCreate`
- The transfer has to be done using `cudaMemcpyAsync` involving a different stream from the one used in the kernel call
- Destroy streams with `cudaStreamDestroy`

In `StreamT.cu`

- Function `withoutStream` allows comparison between using a big array or small frames
- Define a function `withStream` that operates on frames using streams

Plan

Local volatility from implied volatility

First array: r_g

$$dS_t = S_t r_g(t)dt + S_t \sigma_{loc}(S_t, t)dW_t, \quad S_0 = x_0.$$

- S is the stock price process where x_0 is the spot price
- W is a Brownian motion with $W_0 = 0$
- r_g is the risk-free rate, assumed piecewise constant
- $\sigma_{loc}(x, t)$ is a local volatility function: $\mathbb{R}_+^* \times \mathbb{R}_+ \to \mathbb{R}_+^*$

Let k, q with $(x, t) \in]K_g[k], K_g[k+1]] \times]T_g[q], T_g[q+1]]$

$$\bar{\sigma}(x, t) = \sum_{i=0}^{3} \sum_{j=0}^{3} C_g(k, q, i, j) l^i u^j$$

$$\text{where } l = \frac{t - T_g[q]}{T_g[q+1] - T_g[q]} \text{ and } u = \frac{x - K_g[k]}{K_g[k+1] - K_g[k]}$$

Fourth array: C_g

$$C_g(k, q, i, j) = C_g[k * (n_t - 1) * 16 + q * 16 + i * 4 + j]$$

with $(k, q, i, j) \in \{0, ..., n_k - 1\} \times \{0, ..., n_t - 1\} \times \{0, ..., 3\} \times \{0, ..., 3\}$ and n_k, n_t are the size of K_g and T_g

Approximated local volatility

$$\sigma_{loc}^2(x, t) = Dupire\left(\bar{\sigma}, \frac{\partial \bar{\sigma}}{\partial t}, \frac{\partial \bar{\sigma}}{\partial x}, \frac{\partial^2 \bar{\sigma}}{\partial x^2}\right)$$

Pricing bullet option with Monte Carlo (MC)

Price for $t \in [0, T)$

$$F_t = e^{-\int_t^T r_g(u)du} E\left((S_T - K)_+ 1_{\{I_T \in [P_1, P_2]\}} | \mathfrak{F}_t\right) \text{ with } I_t = \sum_{T_i \leq t} 1_{\{S_{T_i} < B\}}$$

- 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 above S I_T times with $\{P_1, ..., P_2\} \subset \{0, ..., M\}$

(S_t, I_t) is Markov

$$F(t, x, j) = e^{-\int_t^T r_g(u)du} E(X | S_t = x, I_t = j), \; X = (S_T - K)_+ 1_{\{I_T \in [P_1, P_2]\}}$$

Discritization set

$0 = t_0 < t_1 < ... < t_{N_t} = T$ finer than $0 < T_1 < ... < T_M < T$ with $\delta t = \sqrt{t_{k+1} - t_k}$

For each t_k, $k = 0, ..., N_t - 1$:

1. Random number generation (RNG) of independent Normal variables G_i
2. Stock price actualization $S^i_{t_{k+1}} = S^i_{t_k}(1 + r_g(t_k)\delta t \delta t + \sigma_{loc}(S^i_{t_k}, t_k)\delta t G_i)$
3. If $t_{k+1} = T_l$ with $l \in \{1, ..., M\}$, $I^i_{T_l} = I^i_{T_{l-1}} + (S^i_{T_l} < B)$

At t_{N_t}

Compute the payoff X^i then average

PDE, $u(t,x,j)$ with respect to x, Black & Scholes PDE

The price

$$F(t,x,j) = e^{-\int_t^T r_g(u)du} E(X|S_t = x, I_t = j),\ X = (S_T - K)_+ 1_{\{I_T \in [P_1, P_2]\}}$$

For $u(t,x,j) = e^{\int_t^T r_g(u)du} F(t, e^x, j)$, we equivalently solve u PDE given by

$$\frac{1}{2}\sigma_{loc}^2(x,t)\frac{\partial^2 u}{\partial x^2}(t,x,j) + \mu(x,t)\frac{\partial u}{\partial x}(t,x,j) = -\frac{\partial u}{\partial t}(t,x,j) \tag{6}$$

$$\mu(x,t) = r_g(t) - \frac{\sigma_{loc}^2(x,t)}{2},\ (x,t,j) \in]0, \max(K_g)] \times [T_k, T_{k+1}[\times [0, P_2]$$
with $T_0 = 0$ and $T_{M+1} = T$

Final and boundary conditions

$u(T,x,j) = \max(e^x - K, 0)$ for any (x,j) and heuristically
$u(t, \log[\min(Kg)], j) = \texttt{pmin} = 0$ and
$u(t, \log[\max(Kg)], j) = \texttt{pmax} = \max(Kg) - K$

Crank-Nicolson scheme

Denoting $u_{k,i,j} = u(t_k, x_i, j)$, $\sigma = \sigma_{loc}(x_i, t_k)$ and $\mu = \mu(x_i, t_k)$

$$q_u u_{k,i+1} + q_m u_{k,i} + q_d u_{k,i-1} = p_u u_{k+1,i+1} + p_m u_{k+1,i} + p_d u_{k+1,i-1}$$

$$q_u = -\frac{\sigma^2\Delta t}{4\Delta x^2} - \frac{\mu\Delta t}{4\Delta x},\quad q_m = 1 + \frac{\sigma^2\Delta t}{2\Delta x^2},\quad q_d = -\frac{\sigma^2\Delta t}{4\Delta x^2} + \frac{\mu\Delta t}{4\Delta x}$$

$$p_u = \frac{\sigma^2\Delta t}{4\Delta x^2} + \frac{\mu\Delta t}{4\Delta x},\quad p_m = 1 - \frac{\sigma^2\Delta t}{2\Delta x^2},\quad p_d = \frac{\sigma^2\Delta t}{4\Delta x^2} - \frac{\mu\Delta t}{4\Delta x}$$

PDE, $u(t,x,j)$ with respect to j, barrier condition

$$u_t(x,j) = \mathbb{E}\left[(S_T - K)_+ 1_{\left\{\sum_{i=1}^{M} 1_{\{S_{T_i}<B\}} \in [P_1,P_2]\right\}} \middle| S_t = x, \sum_{T_i \le t} 1_{\{S_{T_i}<B\}} = j\right]$$

$t \in [T_M, T[$

$$u_t(x,j) = \mathbb{E}[(S_T - K)_+ | S_t = x]$$

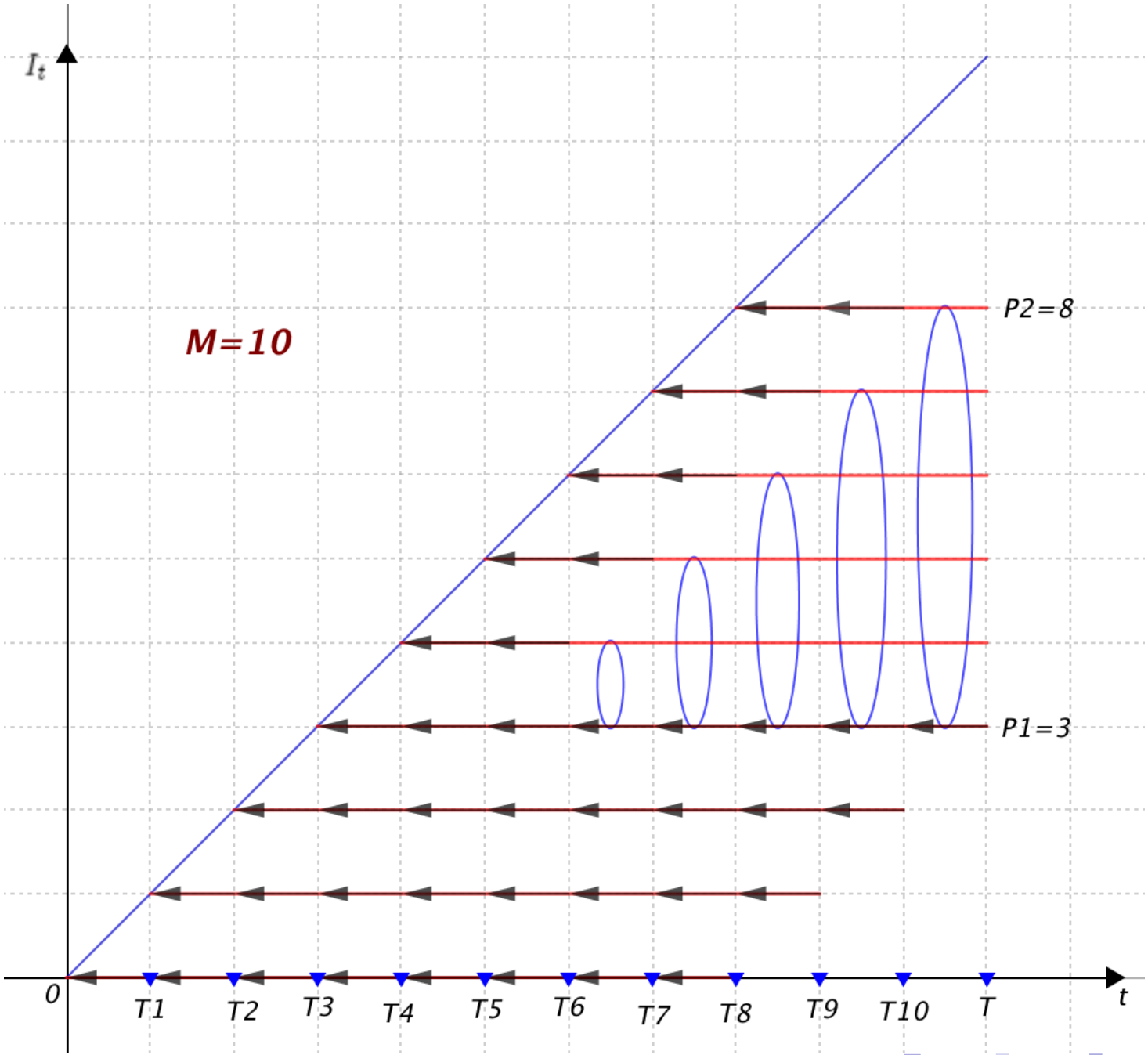
$t \in [T_{M-1}, T_M[$

$$\begin{aligned} u_t(x,j) &= \mathbb{E}[(S_T - K)_+ 1_{\{S_{T_M} \ge B\}} \mid S_t = x] 1_{\{j=P_2\}} \\ &+ \mathbb{E}[(S_T - K)_+ 1_{\{S_{T_M} < B\}} \mid S_t = x] 1_{\{j=P_1-1\}} \\ &+ \mathbb{E}[(S_T - K)_+ | S_t = x] 1_{\{j \in [P_1, P_2-1]\}} \end{aligned}$$

$[T_{M-k-1}, T_{M-k}[$
$k = M-1, ..., 1$
$(T_0 = 0)$

$$\begin{aligned} u_t(x,j) &= \mathbb{E}[u_{T_{M-k}}(S_{T_{M-k}}, P_2) 1_{\{S_{T_{M-k}} \ge B\}} \mid S_t = x] 1_{\{j=P_2\}} \\ &+ \mathbb{E}[u_{T_{M-k}}(S_{T_{M-k}}, p_k^1) 1_{\{S_{T_{M-k}} < B\}} \mid S_t = x] 1_{\{j=p_k^1-1\}} \\ &+ \mathbb{E}\left[\begin{matrix} u_{T_{M-k}}(S_{T_{M-k}}, j) 1_{\{S_{T_{M-k}} \ge B\}} \\ + u_{T_{M-k}}(S_{T_{M-k}}, j+1) 1_{\{S_{T_{M-k}} < B\}} \end{matrix} \middle| S_t = x\right] 1_{\{j \in [p_k^1, P_2-1]\}} \end{aligned}$$

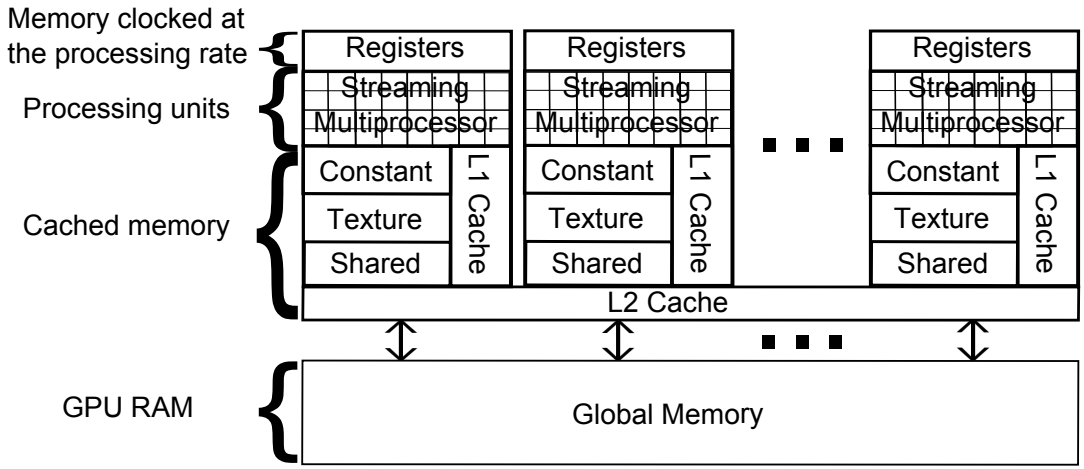
with $p_k^1 = \max(P_1 - k, 0)$


I_t
M=10
P2=8
P1=3
0
T1
T2
T3
T4
T5
T6
T7
T8
T9
T10
T
t

Probability and statistics are generally local

Local dependencies

- Local means easy
- Local explains precisely global
- Local can be parallelized on GPUs



Machine learning

- Neural network: Deep learning has a very large number of publications.
- Random Forest also successfully adapted to GPUs:
  L. Breiman, Random Forests. Machine Learning, 45, 5–32, 2001
- ...
- Some libraries use floating operations, others use integers:
  M. Muja and D. G. Lowe, Fast matching of binary features. IEEE Transaction on Pattern Analysis and Machine Intelligence, 2014, DOI: 10.1109/CRV.2012.60.

Thomas method

Symmetric tridiagonal

$$\begin{pmatrix} d_1 & c_1 & & & & \\ a_2 & d_2 & c_2 & & 0 & \\ & a_3 & d_3 & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & \ddots & \ddots & c_{n-1} \\ & & & & a_n & d_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \tag{7}$$

Forward phase

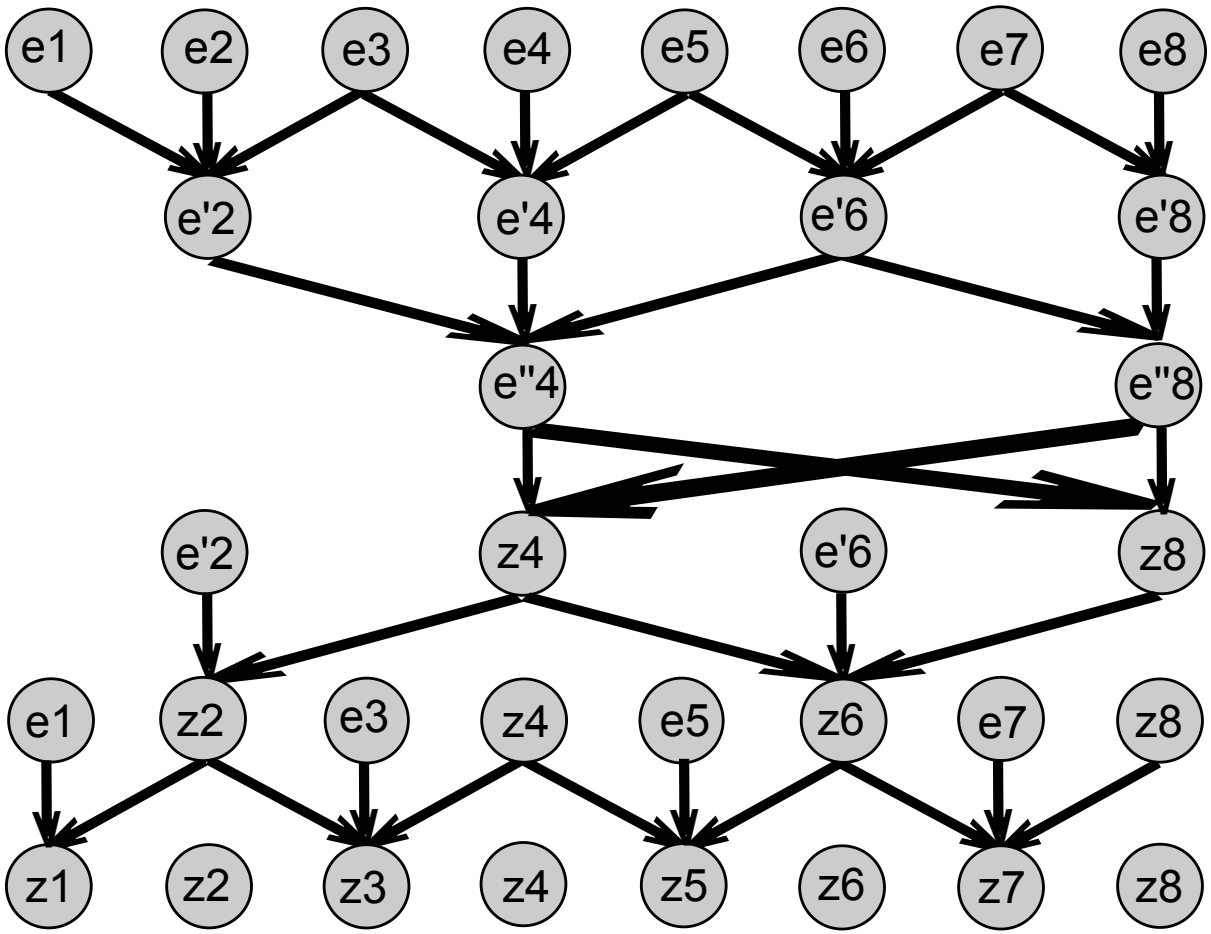
$$c'_1 = \frac{c_1}{d_1},\ y'_1 = \frac{y_1}{d_1},\ c'_i = \frac{c_i}{d_i - a_i c'_{i-1}},\ y'_i = \frac{y_i - a_i y'_{i-1}}{d_i - a_i c'_{i-1}} \text{ when } i = 2, ..., n.$$

Backward phase

$$x_n = y'_n,\ x_i = y'_i - c'_i x_{i+1} \text{ when } i = n-1, ..., 1. \tag{9}$$

From CR: Shared occupation $4n$ and complexity $O(nlog_2(n))$

CR when $n = 8$



Step 1: Forward reduction to a 4-unknown system involving z2, z4, z6 and z8

Step 2: Forward reduction to a 2-unknown system involving z4 and z8

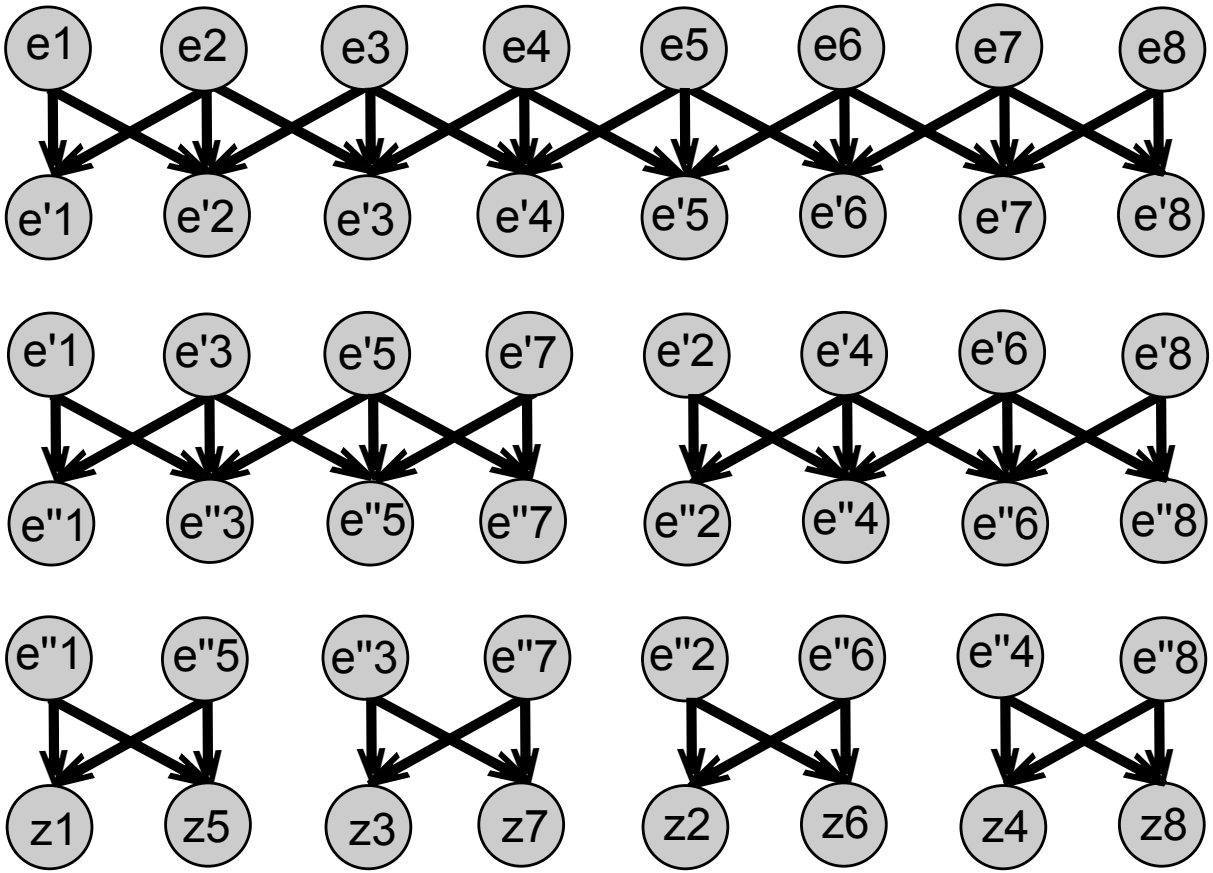
Step 3: Solve 2-unknown system

Step 4: Backward substitution to solve the rest 2 unknowns

Step 5: Backward substitution to solve the rest 4 unknowns

To a new version of PCR: Shared occupation $5n$ and complexity $O(nlog_2(n))$

PCR when $n = 8$



Step 1: Reduced to 2 systems of 4 unknowns

Step 2: Reduced to 4 systems of 2 unknowns

Step 3: Solve

$$\begin{pmatrix} d_1 & c_1 & & & & & \\ a_2 & d_2 & c_2 & & & & \\ & a_3 & d_3 & c_3 & & & \\ & & a_4 & d_4 & c_4 & & \\ & & & a_5 & d_5 & c_5 & \\ & & & & a_6 & d_6 & c_6 \\ & & & & & a_7 & d_7 \end{pmatrix}$$

$$\xrightarrow{(R)} \begin{pmatrix} d'_1 & 0 & c'_1 & & & & \\ 0 & d'_2 & 0 & c'_2 & & & \\ a'_3 & 0 & d'_3 & 0 & c'_3 & & \\ & a'_4 & 0 & d'_4 & 0 & c'_4 & \\ & & a'_5 & 0 & d'_5 & 0 & c'_5 \\ & & & a'_6 & 0 & d'_6 & 0 \\ & & & & a'_7 & 0 & d'_7 \end{pmatrix}$$

$$\xrightarrow{(P)} \begin{pmatrix} d'_1 & c'_1 & & & & & \\ a'_3 & d'_3 & c'_3 & & & & \\ & a'_5 & d'_5 & c'_5 & & & \\ & & a'_7 & d'_7 & 0 & & \\ & & & 0 & d'_2 & c'_2 & \\ & & & & a'_4 & d'_4 & c'_4 \\ & & & & & a'_6 & d'_6 \end{pmatrix}$$

$$\xrightarrow{(R)} \begin{pmatrix} d''_1 & 0 & c''_1 & & & & \\ 0 & d''_3 & 0 & c''_3 & & & \\ a''_5 & 0 & d''_5 & 0 & & & \\ & a''_7 & 0 & d''_7 & 0 & & \\ & & & 0 & d''_2 & 0 & c''_2 \\ & & & & 0 & d''_4 & 0 \\ & & & & a''_6 & 0 & d''_6 \end{pmatrix}$$

$$\xrightarrow{(P)} \begin{pmatrix} d''_1 & c''_1 & & & & & \\ a''_5 & d''_5 & 0 & & & & \\ & 0 & d''_3 & c''_3 & & & \\ & & a''_7 & d''_7 & 0 & & \\ & & & 0 & d''_2 & c''_2 & \\ & & & & a''_6 & d''_6 & 0 \\ & & & & & 0 & d''_4 \end{pmatrix}$$