



Arbitragis Trading Cuda in Finance

Le parallélisme en Finance

- A. HPC & Finance
- B. Programmation multi threadée
- C. Cuda

II. Cuda par l'exemple

- A. Black Scholes
- B. Correlation
- C. Monte Carlo
- D. Annexes



Tuan Nguyen

Tel: + 33 1 47 61 02 34

tuan.nguyen@arbitragis.com



A. Le parallélisme en finance

Pricing d'options vanille

Meilleure précision sur les grecques (delta, gamma, vega)

Multitude d'analyse de risque scénario en temps-réel,

Pricing d'options exotiques

Monte Carlo à haute dimension

Meilleure précision sur les grecques croisées

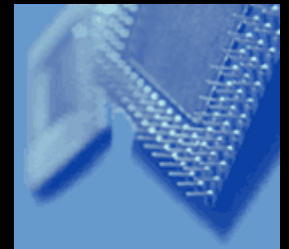
Calculs de Value At Risk plus rapides

Les technologies employées

MPI, OpenMP

FPGA

Une nouvelle technologie : CUDA.





B. Définitions

Difficultés de la programmation multi-threadée

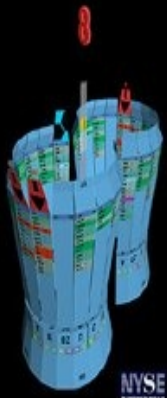
La synchronisation

L'accès concurrentiel des threads à la mémoire

Une façon différente de penser la programmation

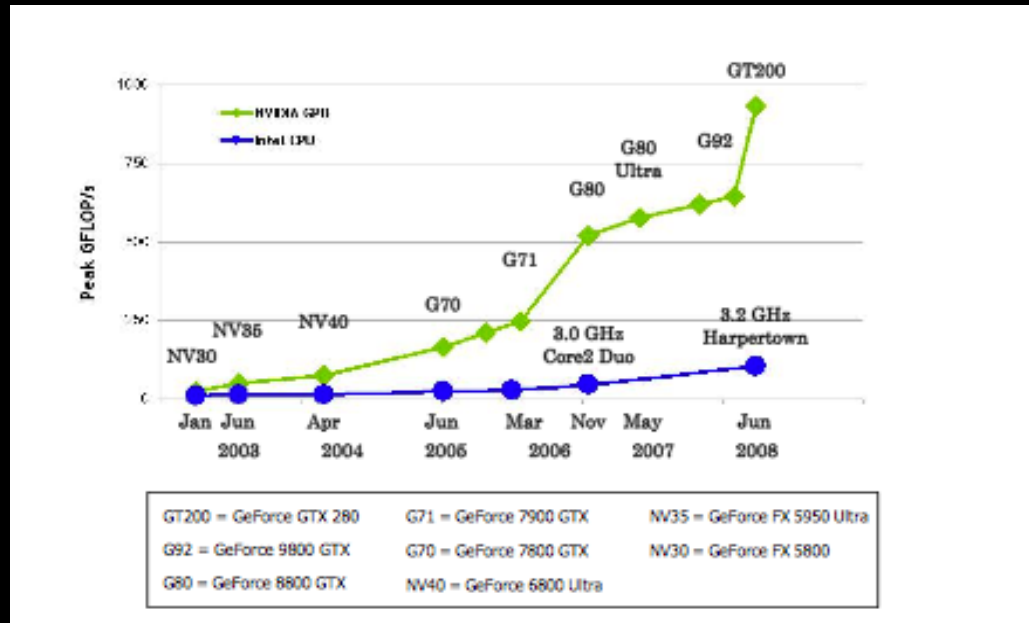
Une résolution du problème :

Cuda : permettre à chaque thread de traiter le même problème avec des données différentes.



C. Cuda : Présentation

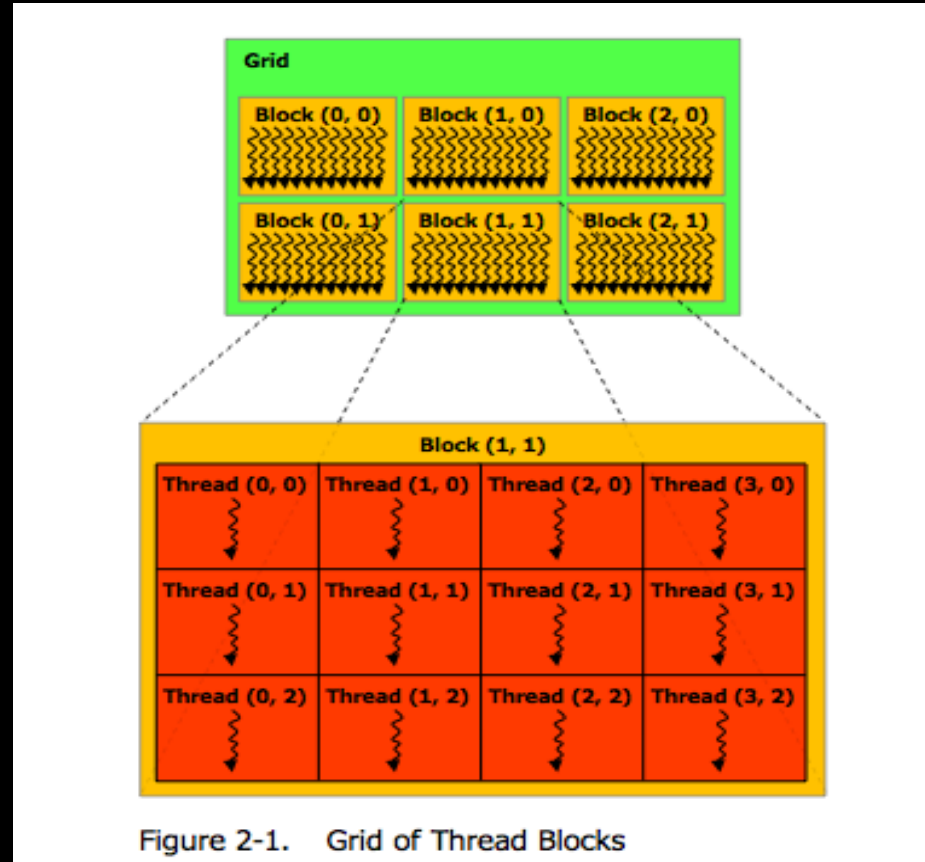
Une puissance de calcul hors du commun



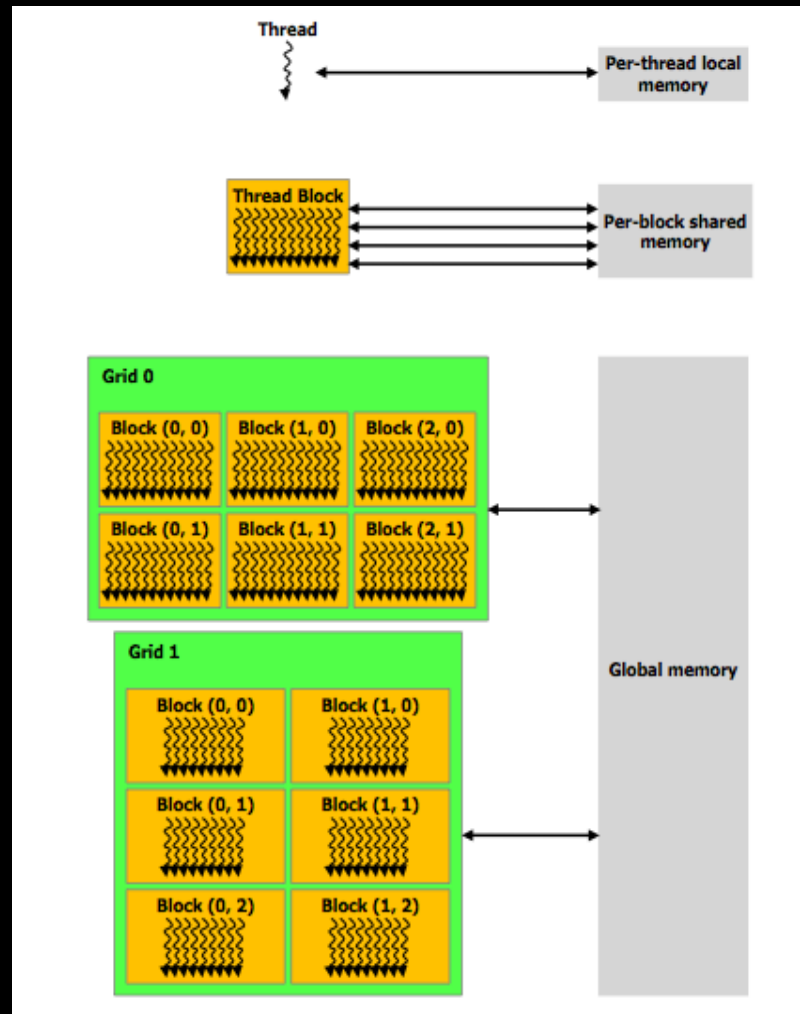
Matériel fourni par Arbitragis et Nvidia à l'ECP :
70% de la puissance de calcul de Météo France
pour 0.10% du coût.

C. Définitions Cuda

Threads, Blocks & Grids



C. Cuda : Accès à la mémoire





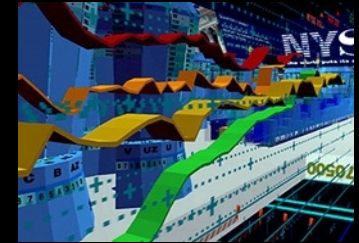
C. Cuda : le hardware

Exemple : Tesla D870

Multiprocesseurs

Processeurs

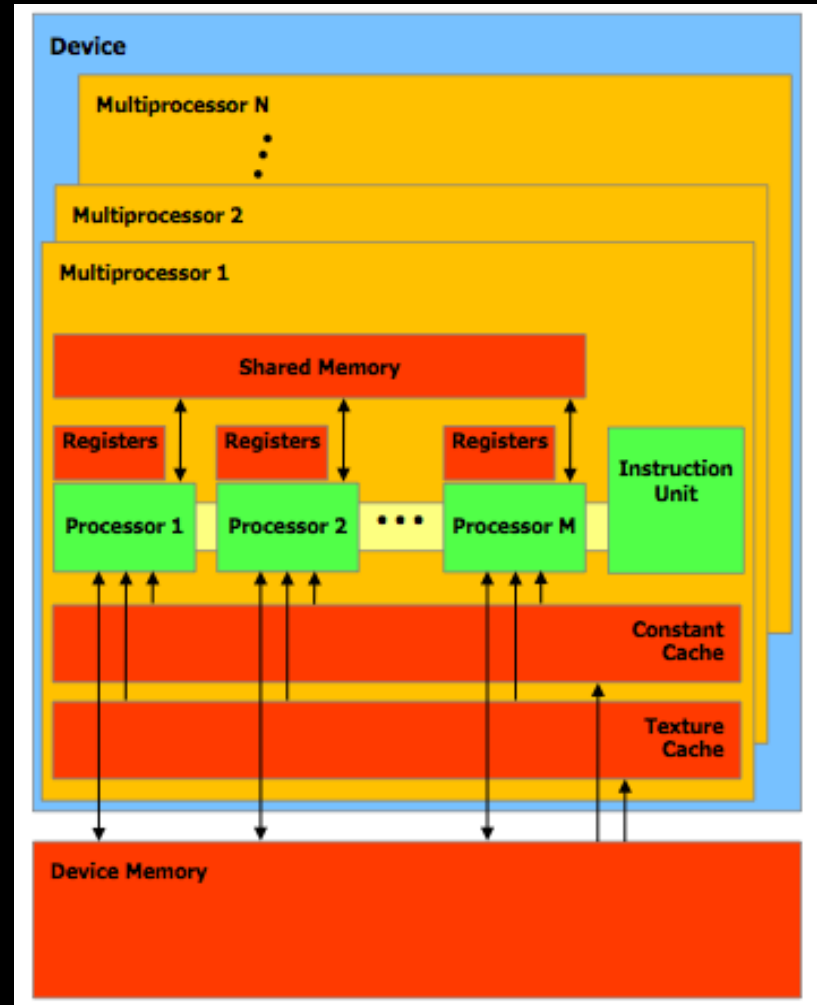
Warp (nombre de threads pouvant être traité par un processeur)





C. Cuda : Multiprocesseurs, Processeurs et Blocks

Tesla D870 : 2 GPU C870
(total de 4 GPU)
16 Multiprocesseurs / GPU
8 Processeurs /
Multiprocesseur
Total / GPU : 128
Processeurs ou "cores"
Fréquence / core : 1.35 Ghz.
Fréquence cumulée : 690
Ghz vs 3.4 Ghz pour Intel.
Mémoire : 3 Go.



C. Cuda : Déclaration des fonctions

Function Type Qualifiers

`__device__`

The **`__device__`** qualifier declares a function that is:

- Executed on the device
- Callable from the device only.

`__global__`

The **`__global__`** qualifier declares a function as being a kernel. Such a function is:

- Executed on the device,
- Callable from the host only.

`__host__`

The **`__host__`** qualifier declares a function that is:

- Executed on the host,
- Callable from the host only.

It is equivalent to declare a function with only the **`__host__`** qualifier or to declare it without any of the **`__host__`**, **`__device__`**, or **`__global__`** qualifier; in either case the function is compiled for the host only.

However, the **`__host__`** qualifier can also be used in combination with the **`__device__`** qualifier, in which case the function is compiled for both the host and the device.



C. Cuda : Déclaration des variables

Variable Type Qualifiers

__device__

The **__device__** qualifier declares a variable that resides on the device.

At most one of the other type qualifiers defined in the next three sections may be used together with **__device__** to further specify which memory space the variable belongs to. If none of them is present, the variable:

- Resides in global memory space,
- Has the lifetime of an application,
- Is accessible from all the threads within the grid and from the host through the runtime library.

__constant__

The **__constant__** qualifier, optionally used together with **__device__**, declares a variable that:

- Resides in constant memory space,
- Has the lifetime of an application,
- Is accessible from all the threads within the grid and from the host through the runtime library.



C. Cuda : Déclaration des variables

`__shared__`

The `__shared__` qualifier, optionally used together with `__device__`, declares a variable that:

- ❑ Resides in the shared memory space of a thread block,
- ❑ Has the lifetime of the block,
- ❑ Is only accessible from all the threads within the block.

Only after the execution of a `__syncthreads ()` (Section 4.4.2) are writes to shared variables guaranteed to be visible by other threads. Unless the variable is declared as volatile, the compiler is free to optimize the reads and writes to shared memory as long as the previous statement is met.



C. Cuda : Appel d'une fonction du kernel

Une fonction du kernel est une fonction exécutée sur la GPU

```
__global__ void Func(float* parameter);  
must be called like this:  
Func<<< Dg, Db, Ns >>>(parameter);
```

Dg est de type dim3 (Dg.x * Dg.y) : Dimension et taille de chaque grid en nombre de blocks

Db est de type dim3 (Db.x * Db.y * Db.z) : Dimension et taille de chaque block en nombre de threads

Ns est la taille allouée par block



C. Cuda : Variables prédéfinies

gridDim : dimensions de la grid (x, y, z) en nombre de blocks

blockIdx : numéro du block dans la grid

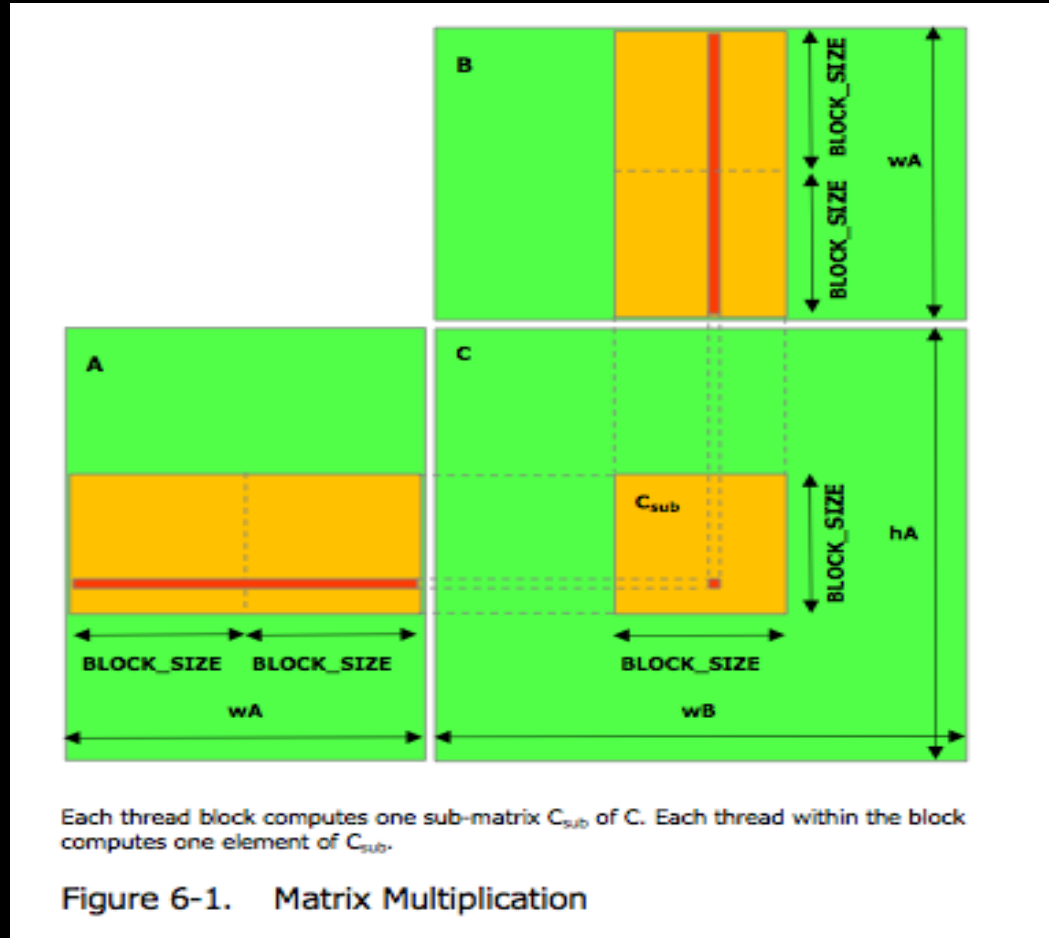
blockDim : dimensions du block

threadIdx : id du thread dans le block (x, y, z)

warpSize : taille du warp en nombre de threads



II. Cuda par l'exemple : Multiplication de matrice





A. Multiplication de Matrices

Pp 73-74 du Cuda Programming Guide 2.0



II. Example of application in finance : Black Scholes

- A. Correlation parallélisée
- B. Black Scholes parallélisé
- C. Monte Carlo parallélisé
- D. Annexes



A. Corrélation parallélisée

CUBLAS – librairie d'operations algebriques
(BLAS = Basic Linear Algebra Subprograms)

$$\rho_{X,Y} = \frac{\frac{1}{N-1} \sum_{j=1}^N \left(X_j - \frac{1}{N} \sum_{i=1}^N X_i \right) \left(Y_j - \frac{1}{N} \sum_{i=1}^N Y_i \right)}{\sigma_X \sigma_Y}$$

Exemple: code source



B. Exemples : Black Scholes parallélisé





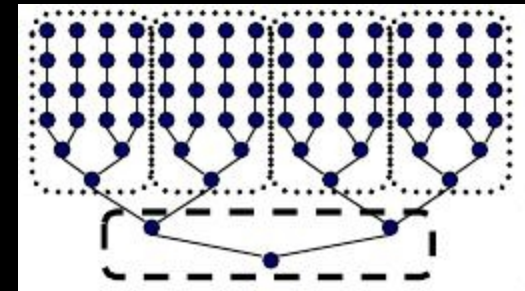
C. Monte Carlo

$$V_{mean}(S, T) = \frac{1}{N} \sum_{i=1}^N V_i(S, T)$$



Comment diviser le calcul sur GPU?

- une option par block
- une trajectoire par thread du block



Paramètres de l'option: T, R, V

```
const float MuByT = (R - 0.5 * V*V) * T;
```

```
const float VBySqrtT = V * sqrt(T);
```

```
__device__ inline float endCallValue(float S, float X, float r, float MuByT, float VBySqrtT)
```

```
{
```

```
float callValue = S * __expf(MuByT + VBySqrtT * r) - X;
```

```
return (callValue > 0) ? callValue : 0;
```

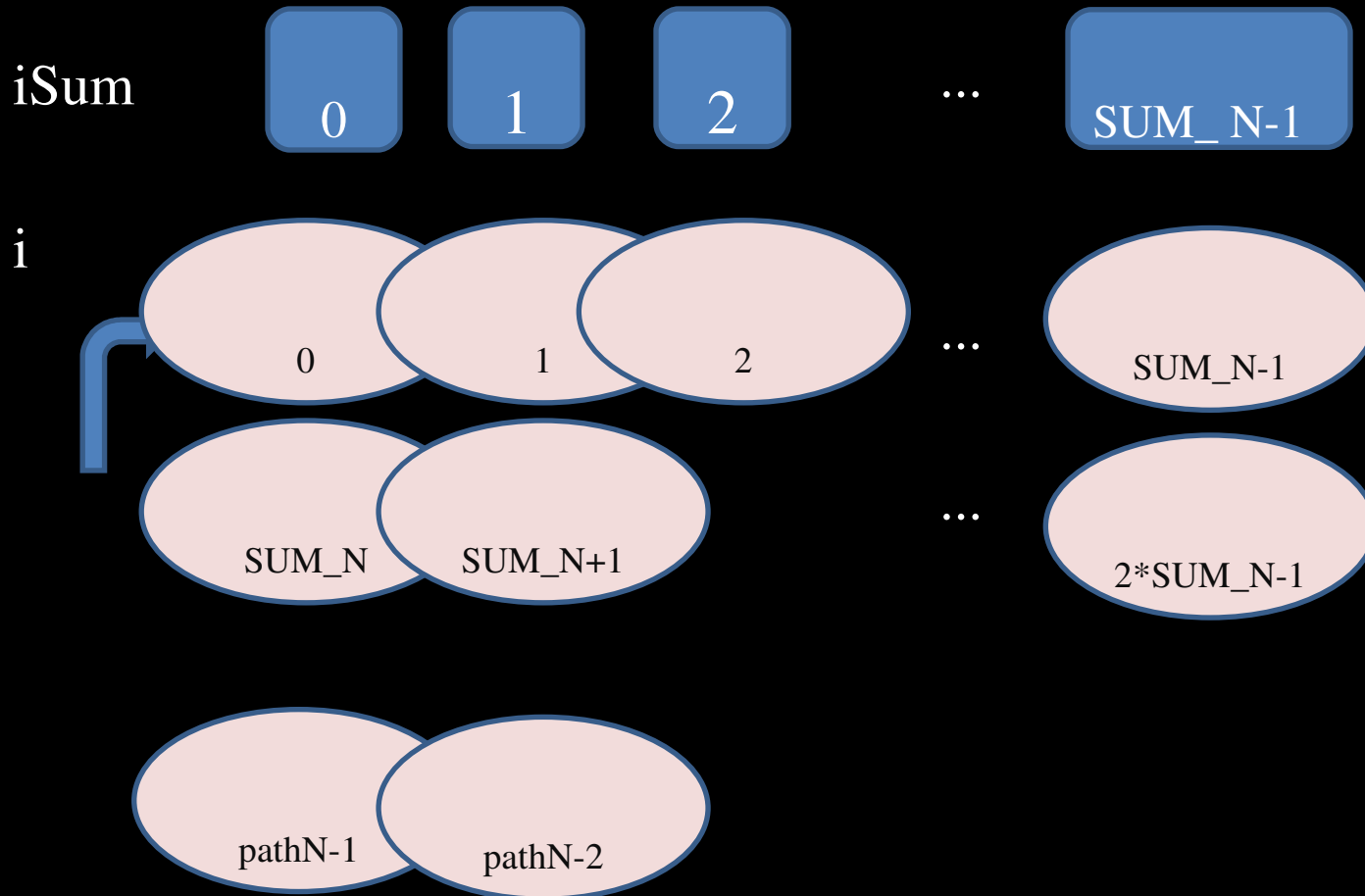
```
}
```

$$S_T = S_0 e^{\left(R - \frac{v^2}{2}\right)T + v\sqrt{T}N(0,1)}$$

$$V_{call}(S, T) = \max(S_T - X, 0)$$



C. Monte Carlo







D. Annexes

Sites internet :

Articles et codes sur la finance et Cuda :

<http://www.arbitragis-research.com>

http://www.nvidia.com/object/cuda_get.html (le sdk)

Bibliographie :

Excellent livre sur le parallélisme :

Patterns for Parallel Programming (Software Patterns Series) (Hardcover)

by Timothy G. Mattson (Author), Beverly A. Sanders (Author), Berna L. Massingill (Author)

Article intéressant sur un design pattern inventé par Google récemment (pattern map reduce)

<http://labs.google.com/papers/mapreduce.html>



Micro FAQ

1. Version à utiliser : télécharger la version 2.0 de Cuda (cf annexes)
2. Pour compiler un code Cuda existant

```
cd ~/mypath/NVIDIA_CUDA_SDK/  
make
```

Pour exécuter le projet BlackScholes :

```
cd bin/linux/release  
./BlackScholes
```

3. Pour créer un nouveau projet :
- ```
cd ~/mypath/NVIDIA_CUDA_SDK/
cp -R template monSuperProjet
cd monSuperProjet
```

Et modifiez vos fichiers.