Multi-GPU parallelization for 3D tomographic reconstruction

Algorithm Architecture Adequacy for solving inverse problems

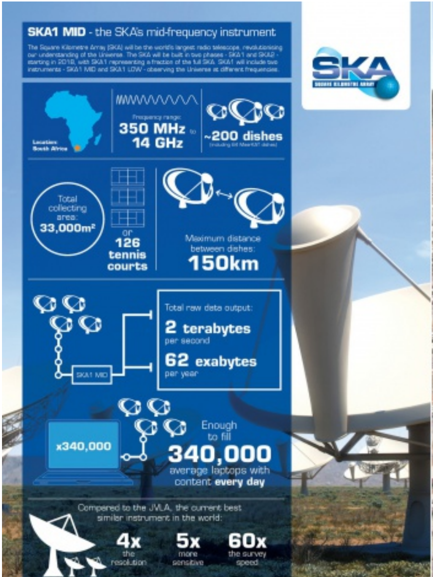
Nicolas GAC (Ass. Prof. Université Paris-sud)
Groupe Problèmes Inverses (GPI)
L2S (CentraleSupélec/CNRS/Univ Paris Sud)

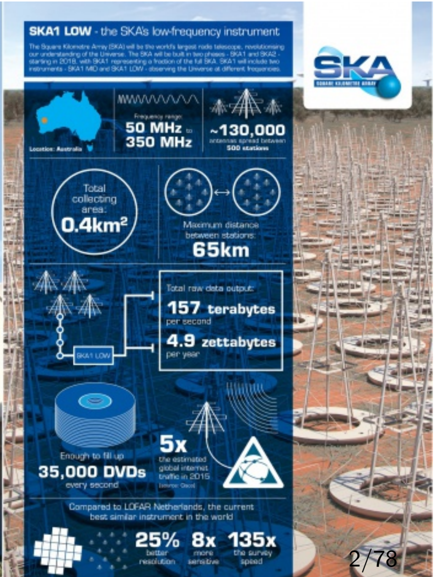
actuellement en délégation CNRS (6 mois) au laboratoire Lagrange

Travaux sur le projet SKA avec André Ferrari

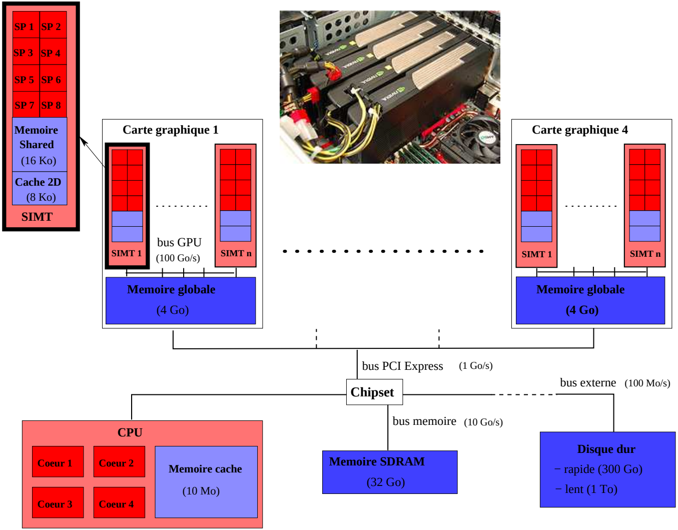
Séminaire du laboratoire Lagrange, site Valrose, 28 novembre 2017

The beauty, reconstuction algorithms for SKA





...and the beast, multi GPU servers

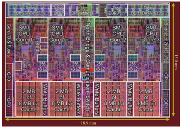


1. GPU (Graphic Processing Units) : hardware and software
   - GPU (re)designed as a many core architecure
   - Programming in CUDA
   - A toy example : acceleration of matrix multiplication
2. Solving (ill-posed) inverse Problems with big dataset
3. [Tomo3D] Parallelization on the many cores of each GPU board
4. [Tomo3D] Parallelization on the GPU boards of the server

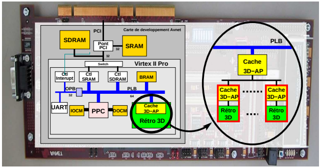
Calcul haute performance

High Performance Computing (HPC)

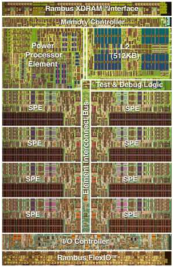
- Parallélisation sur machines multi-processeurs
  ➲ Efficace sur machine à mémoire distribuée
- Noeuds de calculs performants
  ➲ processeurs multi-core, many-core ou FPGA/ASIC



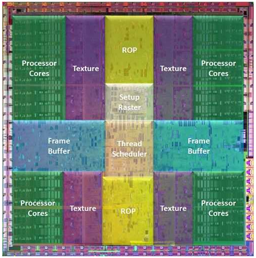
Intel Nehalem (4 coeurs)



SoPC (prototypage)



IBM Cell (8+1 coeurs)



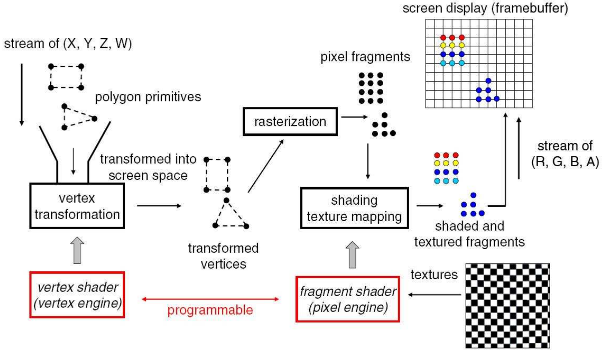
Nvidia GTX 200 (240 coeurs)

GPU : Graphic Processing Unit

Evolution vers une architecture *many core*

- A l'origine, architecture dédiée pour le rendu de volume
  ➲ Pipeline graphique (prog. en OpenGL/Cg)
- Depuis 2006, architecture adaptée à la parallélisation de divers calculs scientifiques
  ➲ CUDA : Common Unified Device Architecure (prog. en C)

Avant CUDA : pipeline graphique



Vertex Shader

Transformation géométrique

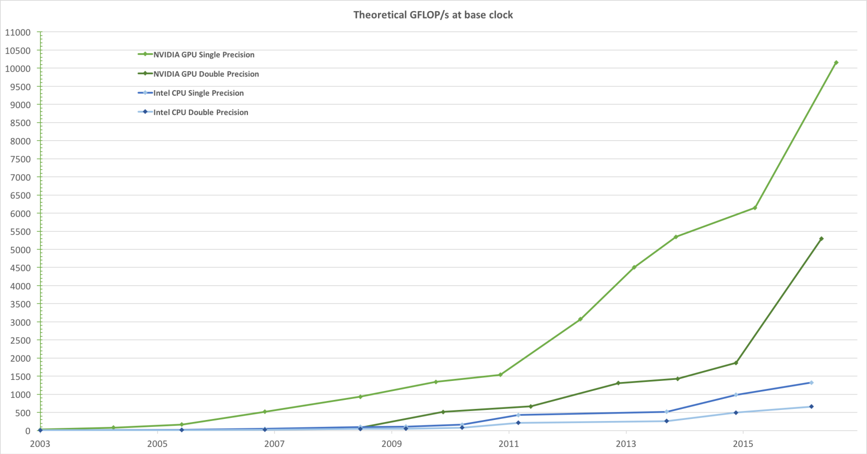
Rasterization

Polygon $->$ Fragments

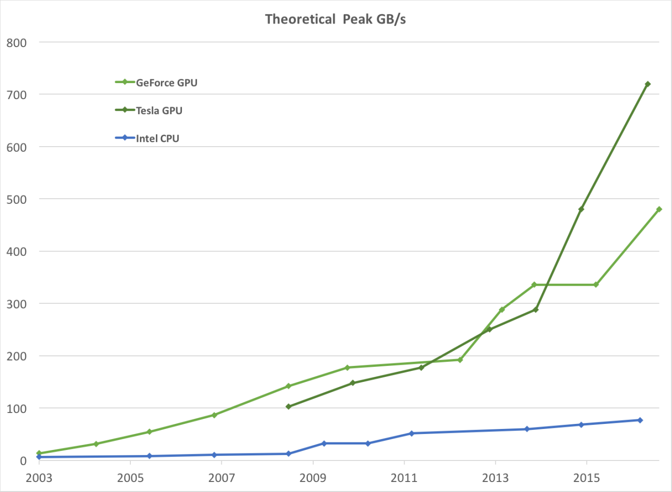
Fragment Shader

Calcul sur les Pixels

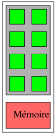
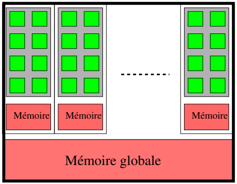
Puissance de calcul



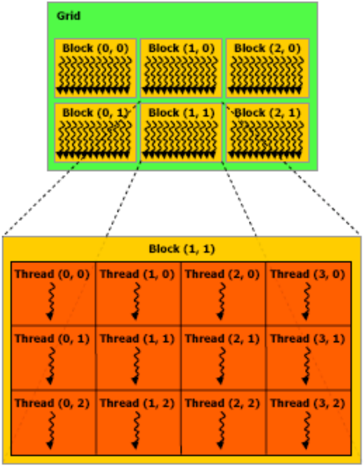
Débit mémoire



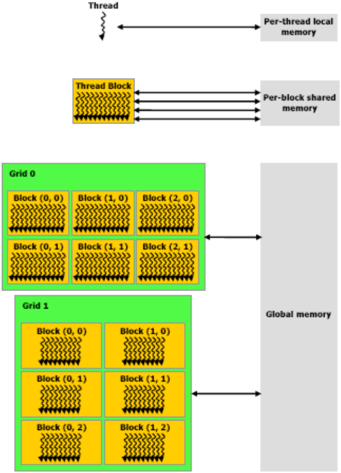
Découpage en threads

	Matériel	Logiciel	Exécution	
	un Streaming Processor (SP)	**un thread**	séquentielle	**(a)**
	un Streaming MultiProcessor (SM)	**un bloc de threads** (plusieurs warps)	parallèle (SIMT)	**(b)**
	une carte GPU (device)	**une grille de threads** (kernel)	parallèle (MIMT)	**(c)**

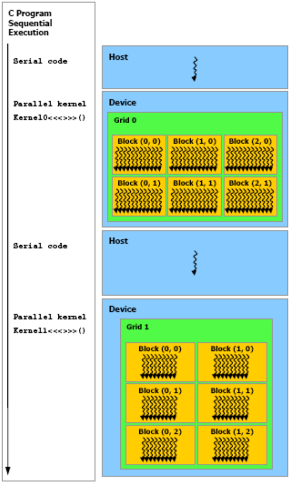
Un **id** par thread et un **id** par bloc de threads



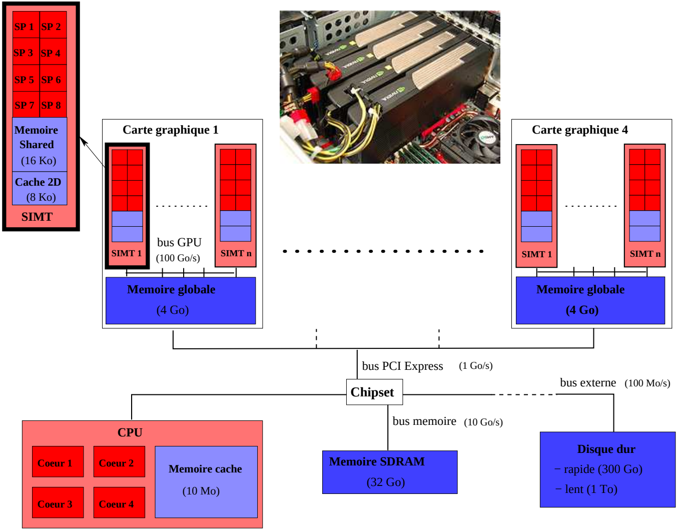
Hiérarchie mémoire



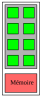
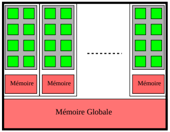
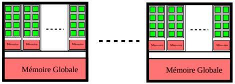
PC hote et carte graphique



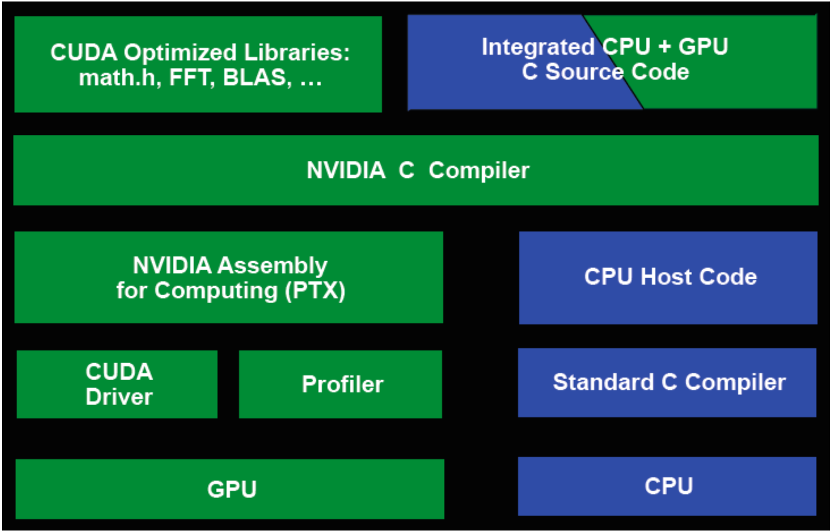
Supercalculateur personnel



Découpage en threads et en **grilles (kernels)**

	Matériel	**Logiciel**	**Exécution**	
	un Stream Processor (SP)	un thread	séquentielle	**(a)**
	un processeur SIMT	un bloc de threads	parallèle (SIMT)	**(b)**
	une carte GPU (device)	une grille de threads (kernel)	parallèle (MIMD) mémoire centralisée	**(c)**
	PC multi−carte	threads du PC hote via librairie pthread (un thread CPU = un kernel GPU)	parallèle (MIMD) mémoire distribuée	**(d)**

Flot de développement logiciel



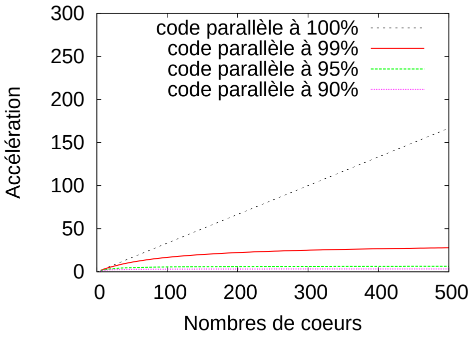
Programmation GPU

❶ Parallélisation de l'algorithme

➲ nourrir en threads (plus ou moins indépendants) le GPU

n coeurs (1 Ghz) vs 1 coeur (3 Ghz)

taux de parallélisation	accélération GTX 200 (240 coeurs)
100 %	80
99 %	24
95 %	6
90 %	3



Programmation GPU

❶ Parallélisation de l'algorithme

➲ nourrir en threads (plus ou moins indépendants) le GPU

❷ Implémentation GPU

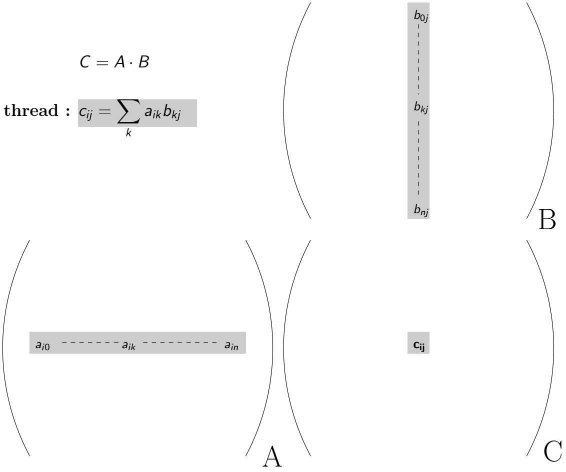
Selon l'intensité arithmétique du code (puissance de calcul exploitée / débit des données), l'execution sera soit *memory bound* soit *computation bound* (ex : calcul X^k [**?**])

➲ optimisation du code portera alors soit sur les **accès mémoire** ou soit sur la **complexité arithmétique**

Parallélisation du calcul matriciel

$$C = A \cdot B$$

un thread : $c_{ij} = \sum_k a_{ik} b_{kj}$

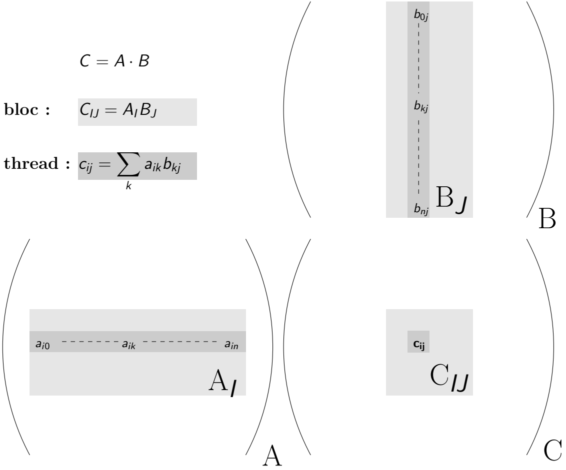


Découpage en blocs de threads

$$C = A \cdot B$$

un bloc : $C_{IJ} = A_I B_J$

un thread : $c_{ij} = \sum_k a_{ik} b_{kj}$



kernel = code des threads executés sur le GPU

```
__global__ void matrixMul_kernel( float* C, float* A, float* B,int matrix_size) {

float C_sum;
int i_first,j_first;
int i,j;

i_first=blockIdx.x*BLOCK_SIZE;
j_first=blockIdx.y*BLOCK_SIZE;

i=i_first+threadIdx.x;
j=j_first+threadIdx.y;

for (k = 0; k < matrix_size; k++)
   C_sum += A[i][k] * B[k][j];

C[i][j] = C_sum;

}
```

Lancement du kernel depuis le PC hôte

```
#define BLOCK_SIZE 16

void matrixMul_host(int N) {

...

//setup execution parameters
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(N /BLOCK_SIZE , N /BLOCK_SIZE );

//execute the kernel
matrixMul_kernel<<< grid, threads >>>(C_device, A_device, B_device, N);

...

}
```

Gestion de la mémoire GPU via le PC hôte

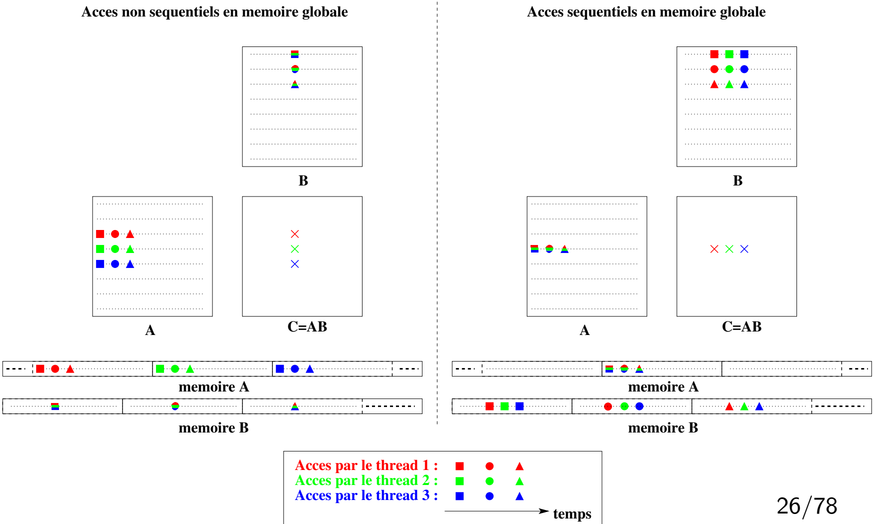
```
#define BLOCK_SIZE 16

void matrixMul_host(int N) {

// allocate host memory int mem_size=N^2*sizeof(float);
float* A_host = (float*) malloc(mem_size);
float* B_host = (float*) malloc(mem_size);
float* C_host = (float*) malloc(mem_size);

// allocate device memory
float* A_device,B_device,C_device;
cudaMalloc((void**) &A_device, mem_size);
cudaMalloc((void**) &B_device, mem_size);
cudaMalloc((void**) &C_device, mem_size);

// copy host memory to device cudaMemcpy(A_device, A_host, mem_size,cudaMemcpyHostToDevice);
cudaMemcpy(B_device, B_host, mem_size,cudaMemcpyHostToDevice);

//setup execution parameters
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(N /BLOCK_SIZE , N /BLOCK_SIZE );

// execute the kernel
matrixMul_kernel<<< grid, threads >>>(C_device, A_device, B_device, N);

// copy result from device to host
cudaMemcpy(C_host, C_device, mem_size,cudaMemcpyDeviceToHost) ;

}
```

Temps GPU

Matrices de taille 1024·1024

	Processeur	Temps d'exécution	Transfert mémoire
C non optimisé	Xeon Quad core 2.7 Ghz	9.35 s	
Cuda	Testla C1060 240 PE @1,3 Ghz	1.35 s **(*6,9)**	$< 1\%$

Accès séquentiels à la mémoire globale



Temps GPU avec accès mémoire sequentiels

Matrices de taille 1024·1024

	Processeur	Temps d'exécution	Transfert mémoire
C non optimisé	Xeon Quad core 2.7 Ghz	9.35 s	
Cuda	Testla C1060 240 PE @1,3 Ghz	1.35 s **(*6,9)**	$< 1\%$
Cuda acces seq.	Testla C1060 240 PE @1,3 Ghz	124 m s **(*10,9)**	5%

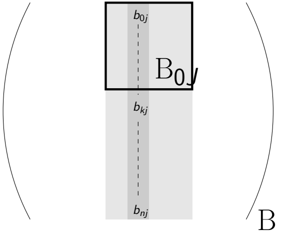
Optimisation des accès mémoire

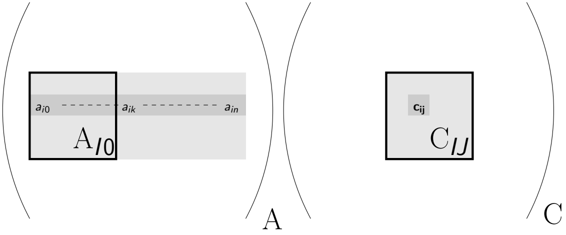
$$C = A \cdot B$$

un bloc : $C_{IJ} = A_I B_J$

un thread : $c_{ij} = \sum_k a_{ik} b_{kj}$

en mémoire shared





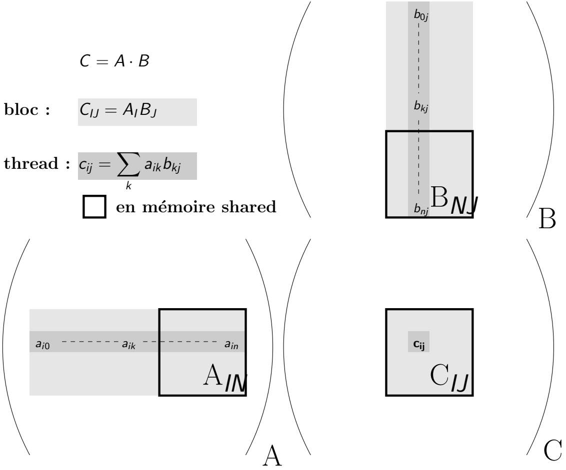
Optimisation des accès mémoire

$$C = A \cdot B$$

un bloc : $C_{IJ} = A_I B_J$

un thread : $c_{ij} = \sum_k a_{ik} b_{kj}$

$\square$ **en mémoire shared**



Variable type qualifiers

__device__

- en mémoire globale
- durée de vie de l'application
- accessible par tous les threads de la grille et par le hôte via la librairie runtime

__constant__

- en mémoire globale (accès via cache constante)
- durée de vie de l'application
- accessible par tous les threads de la grille et par le hôte via la librairie runtime

__shared__

- en mémoire shared (locale à un coeur SIMT)
- durée de vie du bloc de threads
- **seulement accessible par les threads d'un même bloc**

Temps GPU optimisé

Matrices de taille 1024·1024

	Processeur	Temps d'exécution	Transfert mémoire
C non optimisé	Xeon Quad core 2.7 Ghz	9.35 s	
Cuda	Testla C1060 240 PE @1,3 Ghz	1.35 s **(*6,9)**	$<$ 1%
Cuda acces seq.	Testla C1060 240 PE @1,3 Ghz	124 m s **(*10,9)**	5%
Cuda shared mem.	Testla C1060 240 PE @1,3 Ghz	17,5 m s **(*7,1)**	34%

Librairie CUBLAS : CUda Basic Linear Algebra Subprograms

```
#include <cublas.h>
#include <cutil.h>

int main(void) {
float alpha = 1.0f, beta = 0.0f;
int N = 1024;
int mem_size = 1024*1024*sizeof(float);

// Allocate host memory
float* A_host = (float*) malloc(mem_size);
float* B_host = (float*) malloc(mem_size);
float* C_host = (float*) malloc(mem_size);

cublasInit();

//Allocate device memory
float* A_device,B_device,C_device;
cublasAlloc(N*N, sizeof(float), (void **)&A_device);
cublasAlloc(N*N, sizeof(float), (void **)&B_device);
cublasAlloc(N*N, sizeof(float), (void **)&C_device);

// copy host memory to device
cublasSetMatrix(N,N, sizeof(float), A_host, N, A_device, N);
cublasSetMatrix(N,N, sizeof(float), B_host, N, B_device, N);

//Calcul matriciel sur le GPU
cublasSgemm('n', 'n', N, N, N, alpha, A_device, N,B_device, N, beta, C_device, N);

//Récupération du résultat sur le PC hôte
cublasGetMatrix(N,N, sizeof(float), C_device,N, C_host, N);
}
```

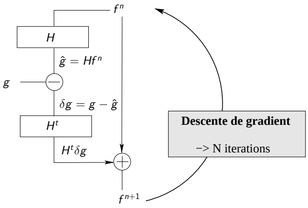
Temps CUBLAS

Matrices de taille 1024·1024

	Processeur	Temps d'exécution	Transfert mémoire
C non optimisé	Xeon Quad core 2.7 Ghz	9.35 s	
Cuda	Testla C1060 240 PE @1,3 Ghz	1.35 s **(*6,9)**	$<1\%$
Cuda acces seq.	Testla C1060 240 PE @1,3 Ghz	124 m s **(*10,9)**	5%
Cuda shared mem.	Testla C1060 240 PE @1,3 Ghz	17,5 m s **(*7,1)**	34%
CUBLAS	Testla C1060 240 PE @1,3 Ghz	12,8 m s **(*1,4)**	43%

1. GPU (Graphic Processing Units) : hardware and software
2. Solving (ill-posed) inverse Problems with big dataset
   - Iterative (bayesian) algorithm
   - Applications
3. [Tomo3D] Parallelization on the many cores of each GPU board
4. [Tomo3D] Parallelization on the GPU boards of the server

Without bayesian regularisation



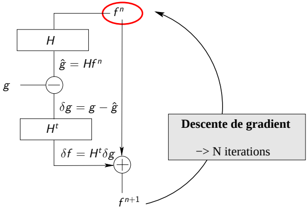
$\mathbf{g} = \mathbf{Hf} + \epsilon$

f : volume
g : tomograph data
H : acquisition model
ϵ : noise

Criterion : Mean Square

$$J(f) = ||g - Hf||^2$$
$$f^{n+1} = f^n - \alpha \cdot \nabla J(f^n)$$
$$\nabla J(f) = -2 \cdot H^t (g - Hf)$$

Without bayesian regularisation



f^n : Estimée du volume

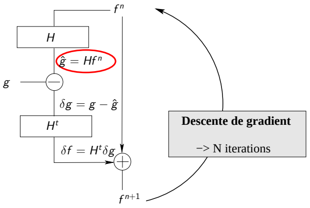
$\mathbf{g} = \mathbf{Hf} + \epsilon$

f : volume
g : tomograph data
H : acquisition model
ϵ : noise

Criterion : Mean Square

$$J(f) = ||g - Hf||^2$$
$$f^{n+1} = f^n - \alpha \cdot \nabla J(f^n)$$
$$\nabla J(f) = -2 \cdot H^t(g - Hf)$$

Without bayesian regularisation



$\hat{g}$ **: Estimée des données**

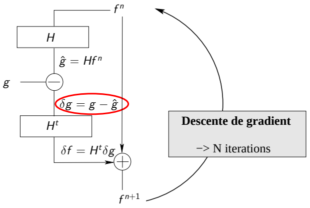
$\mathbf{g} = \mathbf{Hf} + \epsilon$

f : volume
g : tomograph data
H : acquisition model
ϵ : noise

Criterion : Mean Square

$$J(f) = ||g - Hf||^2$$
$$f^{n+1} = f^n - \alpha \cdot \nabla J(f^n)$$
$$\nabla J(f) = -2 \cdot H^t (g - Hf)$$

Without bayesian regularisation



δg : Correction des données

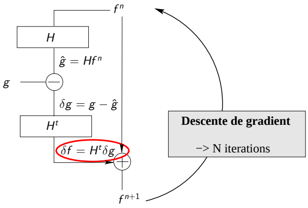
$\mathbf{g} = \mathbf{Hf} + \epsilon$

f : volume
g : tomograph data
H : acquisition model
ϵ : noise

Criterion : Mean Square

$$J(f) = ||g - Hf||^2$$
$$f^{n+1} = f^n - \alpha \cdot \nabla J(f^n)$$
$$\nabla J(f) = -2 \cdot H^t(g - Hf)$$

Without bayesian regularisation



δf : Correction du volume

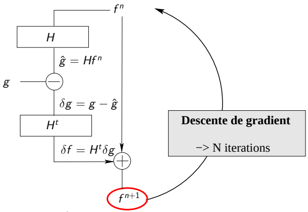
$\mathbf{g} = \mathbf{Hf} + \epsilon$

f : volume
g : tomograph data
H : acquisition model
ϵ : noise

Criterion : Mean Square

$$J(f) = ||g - Hf||^2$$
$$f^{n+1} = f^n - \alpha \cdot \nabla J(f^n)$$
$$\nabla J(f) = -2 \cdot H^t(g - Hf)$$

Without baysian regularisation



f^{n+1} **: Nouvelle estimée du volume**

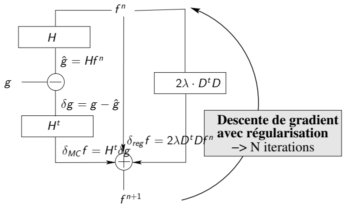
$\mathbf{g} = \mathbf{Hf} + \epsilon$

f : volume
g : tomograph data
H : acquisition model
ϵ : noise

Criterion : Mean Square

$$J(f) = ||g - Hf||^2$$
$$f^{n+1} = f^n - \alpha \cdot \nabla J(f^n)$$
$$\nabla J(f) = -2 \cdot H^t(g - Hf)$$

With bayesian regularisation



$\mathbf{g} = \mathbf{Hf} + \epsilon$

f : volume
g : tomograph data
H : acquisition model
ϵ : noise

Criterion : Mean Square + Quadratic Regularisation (MSQR)

$J(f) = J_1(f) + J_2(f)$
$J_1(f) = ||g - Hf||^2$
$J_2(f) = \lambda ||Df||^2$
$f^{n+1} = f^n - \alpha \cdot (\nabla J_1(f^n) + \nabla J_2(f^n))$

[Planéto] Correction de vibrations mécaniques

Collaboration avec l'IDES de l'Univ. Paris-Sud (F. Schmidt)

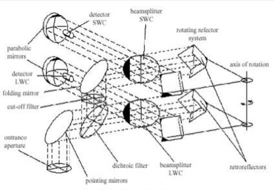
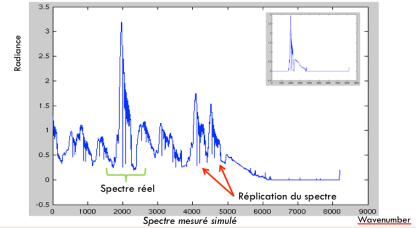


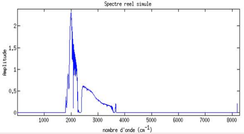
Fig. 1. PFS optical scheme.

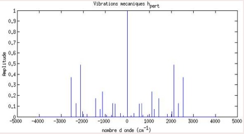
Instrument PFS (Planetary Fourier Spectrum) de la mission MARS EXPRESS



[Planéto] Correction de vibrations mécaniques

Instrument modélisé par une convolution 1D

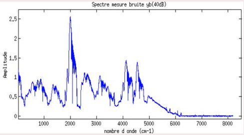




x (spectre réel) * h (instrument PFS)

=

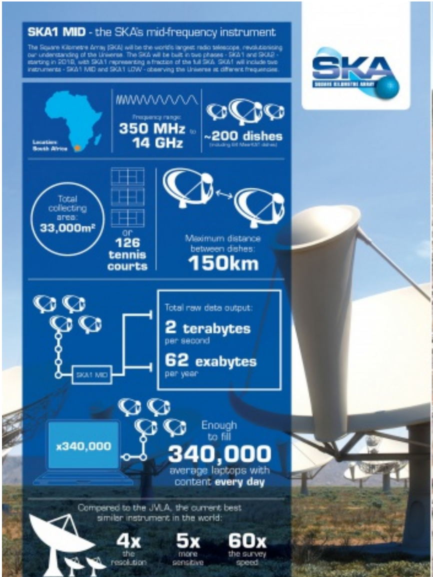
y (spectre mesuré)

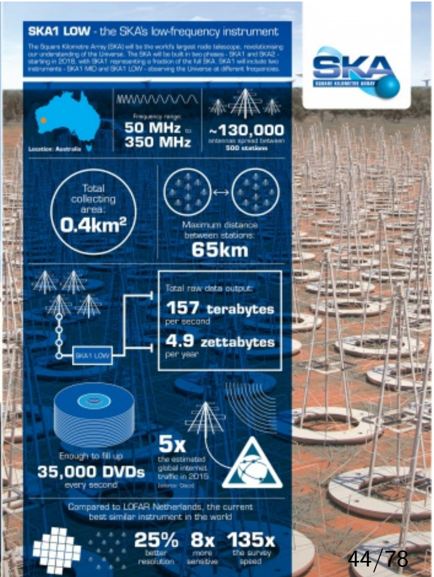


Taille gigantesque des données

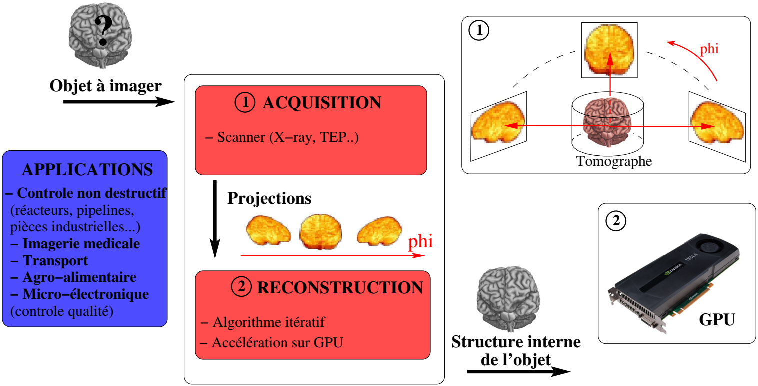
Des années d'enregistrements de la mission MARS EXPRESS (2003) donc potentiellement 1 milliard de spectres (de 8192 échantillons) !

[Astro] Méthode de reconstruction en astronomie



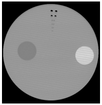
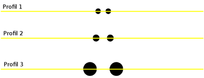


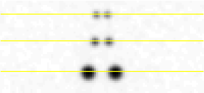
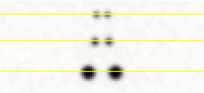
[Tomo3D] Algorithmes de reconstruction tomographique

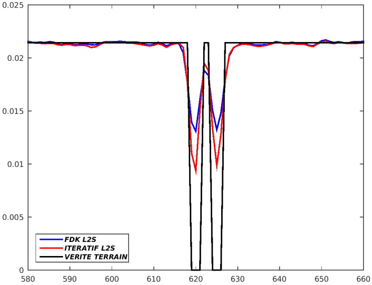


$1K^3$ volume from 1K projections with $1K^2$ pixels (SAFRAN data set)

Work done in collaboration with SAFRAN (Post-doc Thomas Boulay)



1. GPU (Graphic Processing Units) : hardware and software
2. Solving (ill-posed) inverse Problems with big dataset
3. [Tomo3D] Parallelization on the many cores of each GPU board
   - Hardware acceleration of Hf and H^t operators
   - Projection on GPU
   - Backprojection on GPU
4. [Tomo3D] Parallelization on the GPU boards of the server

Hf and $H^t\delta g$ computation

❶ Matrix multiplication

➲ reading h_{ij} coefficients in SDRAM memory

⚠ volume 2048^3 $->$ matrix H $=$ 1 Exa Bytes !

Hf and $H^t\delta g$ computation

❶ Matrix multiplication

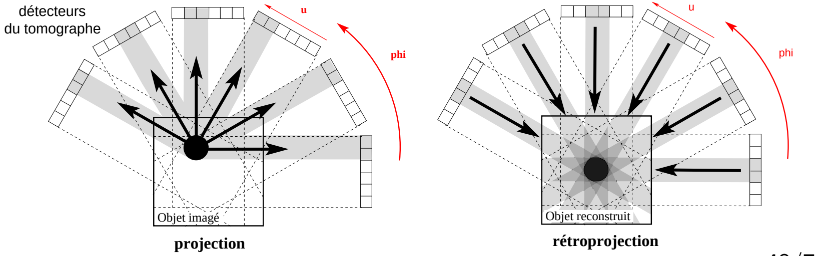
➲ reading h_{ij} coefficients in SDRAM memory

⚠volume 2048^3 $->$ matrix H $=$ 1 Exa Bytes !

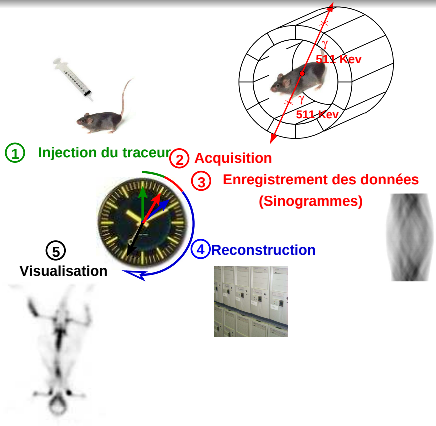
❷ Geometric operators

➲ on line computation of h_{ij} coefficients

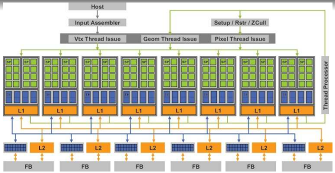
Paire de projection/rétroprojection en tomographie à émission (géométrie parallèle)



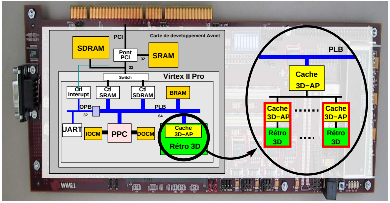
Thèse soutenue en 2008 : “Adéquation Algorithme Architecture pour la reconstruction 3D en imagerie médicale TEP” (Gipsa-lab, Grenoble-INP sous la direction de M. Desvignes et S. Mancini)



Tomographie TEP



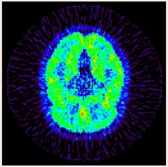
GPU (carte graphique)



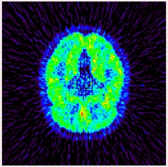
SoPC (prototypage)

Exploration architecturale

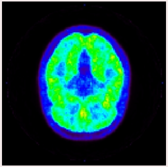
Thesis conclusions



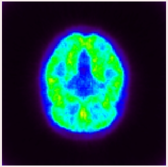
CPU
(non iterative)



FPGA
(non iterative)



CPU
(iterative)



FPGA
(iterative)

CPU/GPU/FPGA comparaison

	CPU	**GPU**	**FPGA**
Time	3^{eme} (*4 P4)	1^{er} (*50 P4)	2^{eme} (*5 P4)
Efficacity	2^{eme} (7 C/op)	1^{eme} (14 C/Op)	1^{er} (2 C/Op)

- **GPU is the hardware accelerator the most performant**
- **FPGA is the hardawre accelerator the most effcient in term of cycles/op (thanks to our cache 3D)**

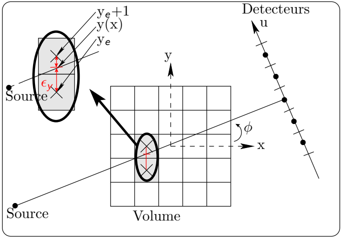
GPU quickly adopted by the tomography community

Publications in Fully 3D

- 2007 : 1st Workshop HPIR (High Performance Image Reconstruction)
- 2011 : Keyword Multi GPU first appeared

	2003	2005	2007	2009	2011	2013	2015	2017
Cluster (MPI/Open MP)	2	3	5	6	3	2	1	2
GPU (NVIDIA)			10	14	17	7	10	20
GPU (AMD)				1	1		1	
Cell (IBM)			3					
FPGA			4		1		1	
DSP			2	1				
Intel(Larabee,Xeon phi)				2		2	2	

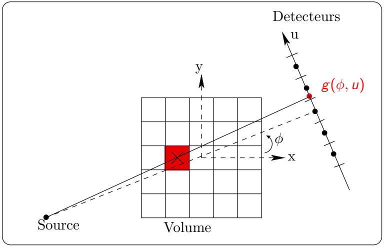
2D projector “regular sampling”



for $(\mathtt{un}, \mathtt{phi})$ in Projection **do**
  for $\mathtt{xn} = 0$ to $\mathtt{xn}_{\mathtt{max}} - 1$ **do**
    // coordinates computation
    $\mathtt{yn}(\mathtt{xn}, \mathtt{un}, \mathtt{phi}) = \ldots$
    // bi-linear interpolation
    $f_{interp} = \ldots$
    // accumulation
    $g^*(\mathtt{un}, \mathtt{phi}) += f_{interp}$
  end for
end for

2D backprojection : algorithm

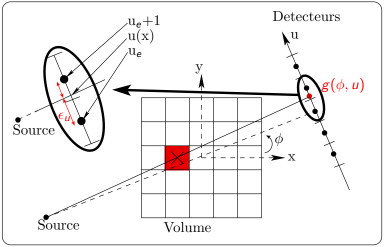
CALCUL DES COORDONNEES



for $(\mathtt{xn}, \mathtt{yn})$ in Volume **do**
  for $\mathtt{phi} = 0$ to $\mathtt{phi}_{\mathtt{max}} - 1$ **do**
    // coordinates computation
    $\mathtt{u}(\mathtt{phi}, \mathtt{xn}, \mathtt{yn}) = \ldots$
    // accumulation
    $f^*(\mathtt{xn}, \mathtt{yn}) += g(u, \phi)$
  end for
end for

2D backprojection : linear interpolation

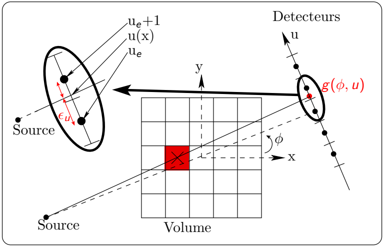
CALCUL DES COORDONNEES



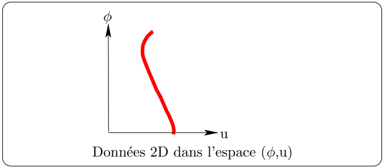
for $(\mathtt{xn}, \mathtt{yn})$ in Volume **do**
  for $\mathtt{phi} = 0$ to $\mathtt{phi}_{\mathtt{max}} - 1$ **do**
    // coordinates computation
    $\mathtt{u}(\mathtt{phi}, \mathtt{xn}, \mathtt{yn}) = \ldots$
    // linear interpolation
    $g_{interp} = (1 - \epsilon_u) \cdot g(\mathtt{phi}, \mathtt{u_e}) + \epsilon_u \cdot g(\mathtt{phi}, \mathtt{u_e} + 1)$
    // accumulation
    $f^*(\mathtt{xn}, \mathtt{yn}) += g_{interp}$
  end for
end for

2D backprojection : scattered data access

CALCUL DES COORDONNEES



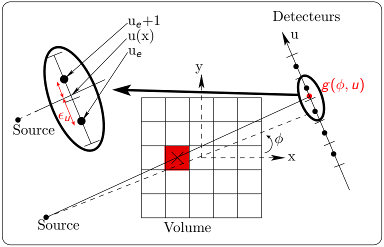
ACCES AUX DONNEES $g(\phi, u)$



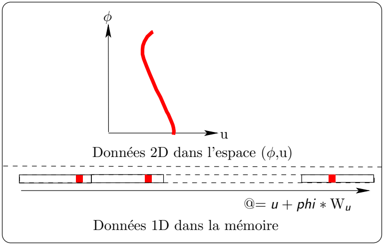
for $(\mathtt{xn}, \mathtt{yn})$ in Volume **do**
    for $\mathtt{phi} = 0$ to $\mathtt{phi}_{\mathtt{max}} - 1$ **do**
        // coordinates computation
        $\mathtt{u}(\mathtt{phi}, \mathtt{xn}, \mathtt{yn}) = \ldots$
        // linear Interpolation
        $g_{interp} = (1 - \epsilon_u) \cdot g(\mathtt{phi}, \mathtt{u_e}) + \epsilon_u \cdot g(\mathtt{phi}, \mathtt{u_e} + 1)$
        // accumulation
        $f^*(\mathtt{xn}, \mathtt{yn}) += g_{interp}$
    end for
end for

2D backprojection : scattered data access

CALCUL DES COORDONNEES



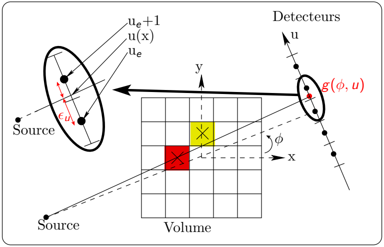
ACCES AUX DONNEES $g(\phi, u)$



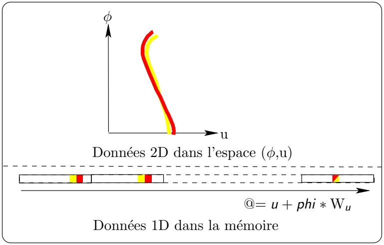
for $(\mathtt{xn}, \mathtt{yn})$ in Volume **do**
  for $\mathtt{phi} = 0$ to $\mathtt{phi}_{\mathtt{max}} - 1$ **do**
    // coordinates computation
    $\mathtt{u}(\mathtt{phi}, \mathtt{xn}, \mathtt{yn}) = \ldots$
    // linear interpolation
    $g_{interp} = (1 - \epsilon_u) \cdot g(\mathtt{phi}, \mathtt{u_e}) + \epsilon_u \cdot g(\mathtt{phi}, \mathtt{u_e} + 1)$
    // accumulation
    $f^*(\mathtt{xn}, \mathtt{yn}) += g_{interp}$
  end for
end for

2D backprojection : scattered data access

CALCUL DES COORDONNEES



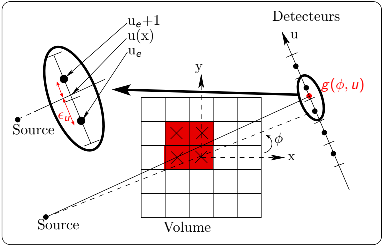
ACCES AUX DONNEES $g(\phi, u)$



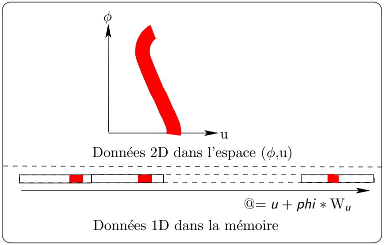
for $(\mathtt{xn}, \mathtt{yn})$ in Volume **do**
  for $\mathtt{phi} = 0$ to $\mathtt{phi}_{\mathtt{max}} - 1$ **do**
    // coordinates computation
    $\mathtt{u}(\mathtt{phi}, \mathtt{xn}, \mathtt{yn}) = \ldots$
    // linear interpolation
    $g_{interp} = (1 - \epsilon_u) \cdot g(\mathtt{phi}, \mathtt{u_e}) + \epsilon_u \cdot g(\mathtt{phi}, \mathtt{u_e} + 1)$
    // accumulation
    $f^*(\mathtt{xn}, \mathtt{yn}) += g_{interp}$
  end for
end for

2D backprojection by blocks : localized data access

CALCUL DES COORDONNEES

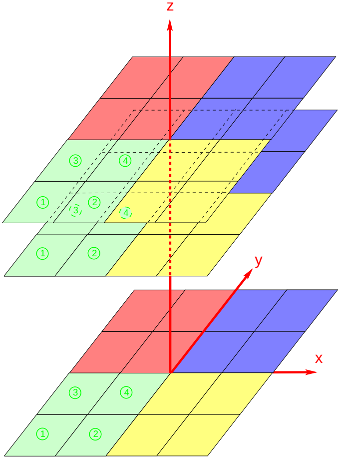


ACCES AUX DONNEES $g(\phi, u)$



for $(\mathtt{Bx}, \mathtt{By})$ in Volume **do**
  for $\mathtt{phi} = 0$ to $\mathtt{phi}_{\mathtt{max}} - 1$ **do**
    for $(\mathtt{xn}, \mathtt{yn})$ in Bloc **do**
      // coordinates computation
      $\mathtt{u}(\mathtt{phi}, \mathtt{xn}, \mathtt{yn}) = \ldots$
      // linear interpolation
      $g_{interp} = (1 - \epsilon_u) \cdot$
      $g(\mathtt{phi}, \mathtt{u_e}) + \epsilon_u \cdot g(\mathtt{phi}, \mathtt{u_e} + 1)$
      // accumulation
      $f^*(\mathtt{xn}, \mathtt{yn}) += g_{interp}$
    end for
  end for
end for

3D backprojection parallelization



(a) Sequential computation on processor element

- Loop on z
- Loop on ϕ

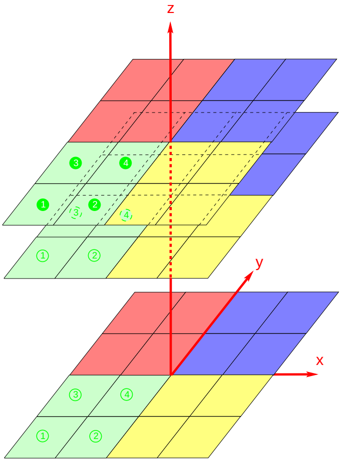
(b) Parallel computation on a block of processors (SIMT)

- Loop on (x,y)

(c) Parallel computation on one card

- Loop on blocks (Bx,By,Bz)

3D backprojection parallelization



(a) Sequential computation on processor element

- Loop on z
- Loop on ϕ

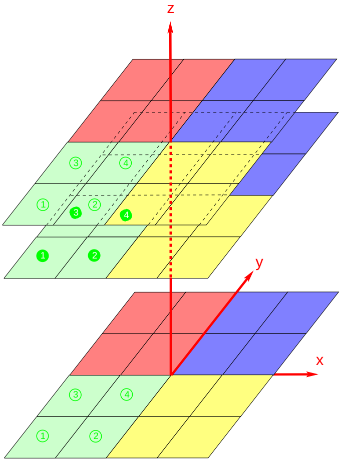
(b) Parallel computation on a block of processors (SIMT)

- Loop on (x,y)

(c) Parallel computation on one card

- Loop on blocks (Bx,By,Bz)

3D backprojection parallelization



(a) Sequential computation on processor element

- Loop on z
- Loop on ϕ

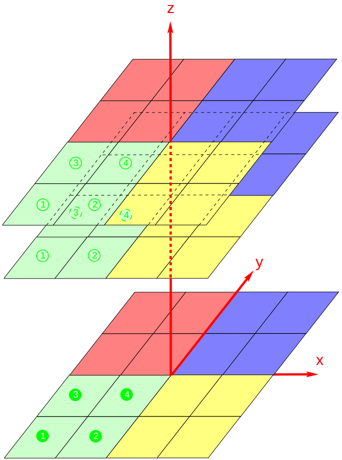
(b) Parallel computation on a block of processors (SIMT)

- Loop on (x,y)

(c) Parallel computation on one card

- Loop on blocks (Bx,By,Bz)

3D backprojection parallelization



(a) Sequential computation on processor element

- Loop on z
- Loop on ϕ

(b) Parallel computation on a block of processors (SIMT)

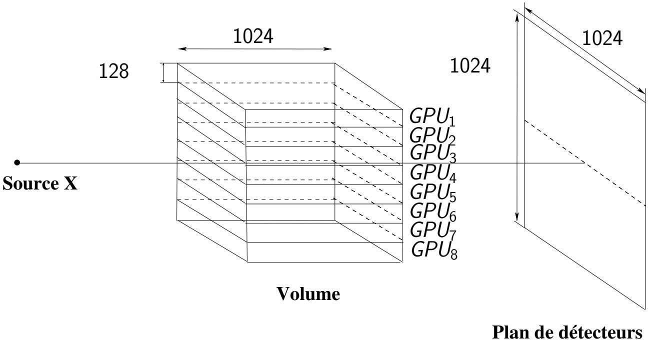
- Loop on (x,y)

(c) Parallel computation on one card

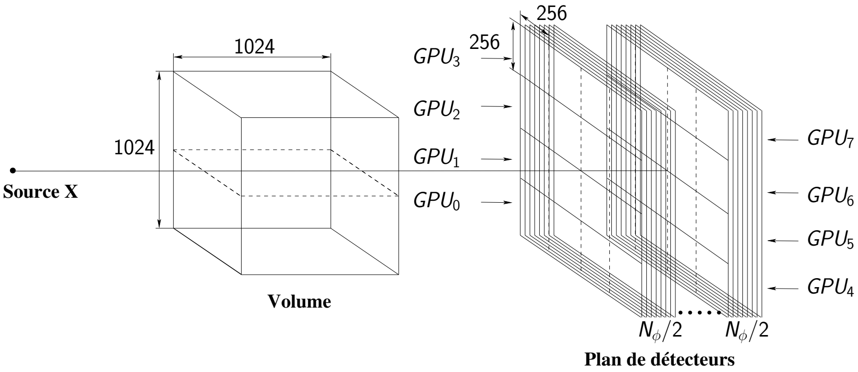
- Loop on blocks (Bx,By,Bz)

Multi GPUs server (Carri Systems)

3D backprojection multi GPU parallelization



3D projection multi-GPU parallelization

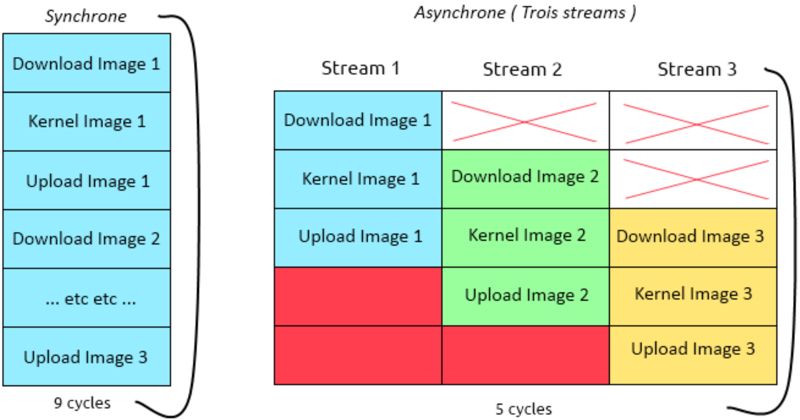


Multi-GPU reconstruction time

Volume $1K^3$ (float) with 1024 projections on 1 to 8 Titans X (3072 cores at 1,075 Ghz)

	1 GPU	2 GPUs	4 GPUs	8 GPUs
Proj (ms)	14416	8183 1,76	4610 3,13	2659 5,42
Back (ms)	7604	5181 1,47	3027 2,51	1929 3,94
Conv (ms)	3062	2987 1,02	2438 1,26	1668 1,84

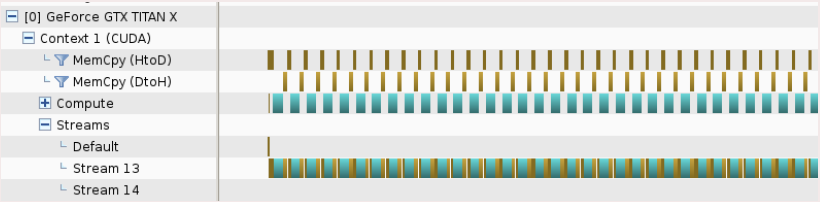
Goal of streams : hide PC/GPU memory transfer



Différence entre synchrone et asynchrone

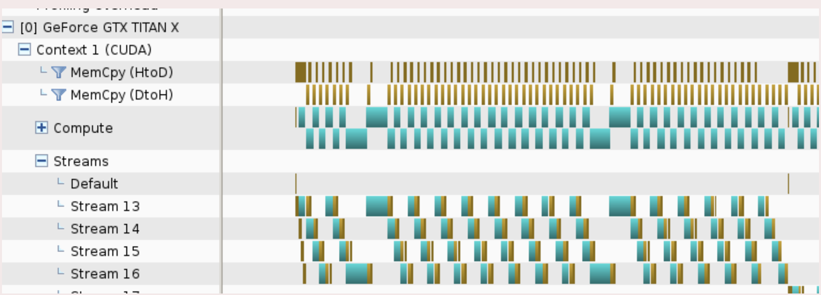
CUDA streams for mono GPU backprojection (1024 angles 1024^2 plan

1 stream



CUDA streams for mono GPU backprojection (1024 angles 1024^2 plan

4 streams



single GPU time with streams

$1K^3$ Volume (float) with 1024 projections on 1 Titan X (3072 cores at 1,075 Ghz)

	compute	upload	download	w/o stream	w/ streams	Acc.
Proj (ms)	88 %	6 %	6 %	14416	11551	1,25
Rétro (ms)	71,1 %	16,9 %	12,1 %	7604	5358	1,42
Conv (ms)	5 %	28,1 %	66,9 %	3062	3072	0,99

multi-GPU time with streams

$1K^3$ volume (float) with 1024 projections on 1 to 8 Titans X (3072 cores at 1,075 Ghz)

	1 GPU	2 GPUs	4 GPUs	8 GPUs
Proj (ms) w/o streams	14416	8183 1,76	4610 3,13	2659 5,42
Proj (ms) w/ streams	**11551**	**5783 2,0**	**3142 3,68**	**1756 6,58**
Back (ms) w/o streams	7604	5181 1,47	3027 2,51	1929 3,94
Back (ms) w/ streams	**5358**	**2609 2,0**	**1672 3,20**	**1731 3,10**
Conv (ms) w/o streams	3062	2987 1,02	2438 1,26	1668 1,84
Conv (ms) w/ streams	**3072**	**2482 1,24**	**2340 1,31**	**1674 1,83**

Limitations due to PCI express gen2 bandwith (2 to 4 GB/s)

Half float data storage

CUDA 7.5 allows half float storage of data on GPU memory

- 16 bits format : sign (1bit), exponant (5bits), mantissa (10bits)
- assembler instructions allow the conversion half/float and float/half in CUDA kernels
- Advantage (i) : reduction of data volume to store on the GPU board
- Advantage (ii) : reduction of memory transfer
- Advantage (iii) : reduction of SDRAM GPU memory access by the GPU cores

single GPU time with streams and half-float storage

$1K^3$ volume (float) with 1024 projections on 1 Titan X (3072 cores at 1,075 Ghz)

	float	half float	Acc.
Proj (ms)	11551	8970	1,29
Back (ms)	5358	4252	1,26
Conv (ms)	3072	1608	1,91

Additional acceleration with half float storage for projection and backprojection

$->$ Reduction of SDRAM GPU memory access time by the GPU cores

multi-GPU Time with streams and half-float storage

$1K^3$ volume (float) with 1024 projections on 1 to 8 Titans X (3072 cores at 1,075 Ghz)

	1 GPU	2 GPUs	4 GPUs	8 GPUs
Proj (ms) f	11551	5783 2,0	3142 3,68	1756 6,58
Proj (ms) hf	**8970**	**4620 1,94**	**2357 3,80**	**1265 7,09**
Back (ms) f	5358	2609 2,0	1672 3,20	1731 3,10
Back (ms) hf	**4252**	**2164 1,96**	**1229 3,46**	**876 4,83**
Conv (ms) f	3072	2482 1,24	2340 1,31	1674 1,83
Conv (ms) hf	**1608**	**1267 1,27**	**1171 1,37**	**843 1,91**

Limitations due to PCI express gen2 bandwith (2 to 4 GB/s)

Data storage during the iterative loop

CPU centralisation

All the data (f^n and f^{n+1} volume, real g and estimated $\hat{g}$ sinograms...) could not stay on the GPU board (true from $1K^3$) volumes)
Because of the cone beam geometry, data could not easily cut in independant block of data
$->$ Data need to be backed up on the CPU at least one time after each iteration

(single)GPU centralization

All the data (f^n and f^{n+1} volume, real g and estimated $\hat{g}$ sinograms...) could stay on the GPU board (true up to 512^3 volumes)
$->$ All the iterative loop could be done on the GPU

(multi)GPU centralisation

All the data (n and n+1 volume, real and estimate sinograms...) could be distributed on the diffrenets GPU boards (true up to $2K^3$ volumes)
$->$ All the iterative loop could be done without data storage on the CPU

CPU centralization

Current strategy : result of each operator (proj, back, conv) is backed up on the CPU

- Advantage : operators (proj, back, conv) are independants (usefull for utilization with Matlab and mex function)
- Disadvantage : several synchronizations CPU/GPU and memory transfer time cost

Solutions to avoid these multiples synchronizations and its impact on reconstruction time

- Use of only one synchronization per iteration by merging operators working on subblock of data (need of a reduction step)
- Hide memory transfer time thanks to streams and half float data storage.

Reconstruction time (per iteration with computation of the optimized gradient step) with CPU centralization

$1K^3$ volume (float) with 1024 projections on Titans X (3072 cores at 1,075 Ghz)

	proj (*2)	retro	conv(*3)	autres	total	Acc.
1 GPU	49,6 %	20,2 %	30,1 %	28,1%	47,1 s	
2 GPUs	36,6 %	7,5%	14,9 %	40,9%	32,4 s	1,45
4 GPUs	23,6 %	7,5 %	21,6 %	47,2 %	27,9 s	1,69
8 GPUs	15,9 %	6,6%	21,6%	55,9 %	23,6 s	1,99

$2K^3$ volume (float) with 2048 projections on Titans X (3072 cores at 1,075 Ghz)

	proj (*2)	retro	conv(*3)	autres	total	Acc.
4 GPUs	36,27 %	20,65%	10,38 %	32,69%	5,4 mn	
8 GPUs	26,38 %	13,31 %	15,22 %	45,09 %	3,8 mn	1,4

Reconstruction time (per iteration with computation of the optimized gradient step) with CPU centralization

Limitations of this CPU centralization

$->$ The "little" operations (norm L2, substraction...) are becoming preponderants....
Solutions :

- Parallelization on the CPU cores (the minimum to do....)
- Merge the operators (break the frontier between each operators)
- Use of half float storage to get a GPU centralization (code 100% GPU)

Reconstruction time (per iteration with computation of the optimized gradient step) with **GPU centralization**

$1K^3$ volume (float) with 1024 projections on one Titan X (3072 cores at 1,075 Ghz)

	proj (*2)	back	conv(*3)	others	total	Acc.
CPU centralization						
1 GPU	49,7 %	9,8 %	12,7 %	27,0 %	43,9 s	
GPU centralization and half float						
1 GPU	78,3 %	18,3 %	2,2 %	1,2 %	21,9 s	2

Conclusions

Towards an efficient computation on GPU for each operator

- Local and spatial memory locality
- Threads/Blocks “optimal”definition (thread parallelism)
- Unrolling loop (instruction parallelism)
- Incremetal computation

Use of streams to hide CPU/GPU memory transfer time

Half-float data storage on GPU

- Reduction of CPU/GPU memory transfer
- Reduction of SDRAM GPU/coeurs GPU memory transfer
- Reduction of storage on SDRAM GPU

$->$ A significant accelaration factor (1.2/1.3) on a single GPU and a more efficient multi-GPU parallelization
$->$ A 100 % GPU code for $1K^3$ volume is becoming possible

iterative reconstruction of $2K^3$ volume

Perspective for SKA project

Short term perspectives

- Acceleration of the convolution (H and Ht)
- Multi-GPU parallelization with CPU Centralisation of data

Median/long term perspectives

- Multi-GPU parallelization with multi-GPU distribution of data
- Use of FPGA Architecture with HLS (High Level Synthesis) tools