UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

SYSCOM
SYSTÈMES COMMUNICANTS

Programmation multiGPU OpenMP *versus* MPI

Gabriel Noaje, Michaël Krajecki, Christophe Jaillet
gabriel.noaje@univ-reims.fr

Équipe SysCom, Laboratoire CReSTIC
Université de Reims Champagne-Ardenne, France

17/02/2011

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

Sommaire

I. Introduction (GPU & multiGPU)
- *GPU*
- *GPU Computing*
- *Programmation de GPU*
- *Produit matriciel sur GPU (CUDA)*

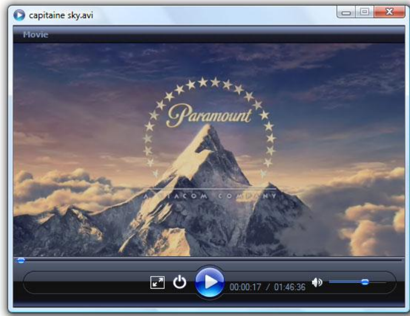
II. Gestion des architectures multi-cartes

III. Résultats expérimentaux

IV. Conclusions

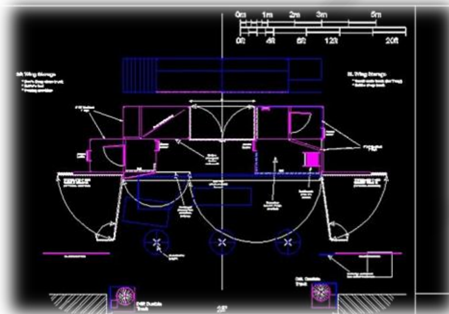
UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

Carte graphique (*Graphic Processing Unit - GPU*)



lecture de films

Animations 3D



CAO (Conception Assistée par Ordinateur)

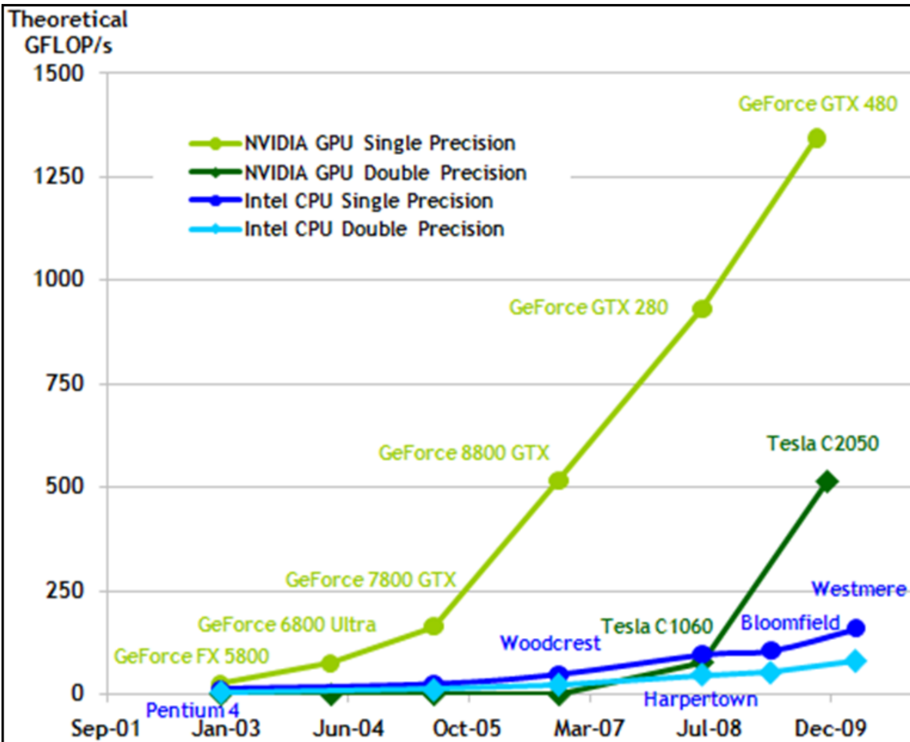
Jeux

GPU, pourquoi ?

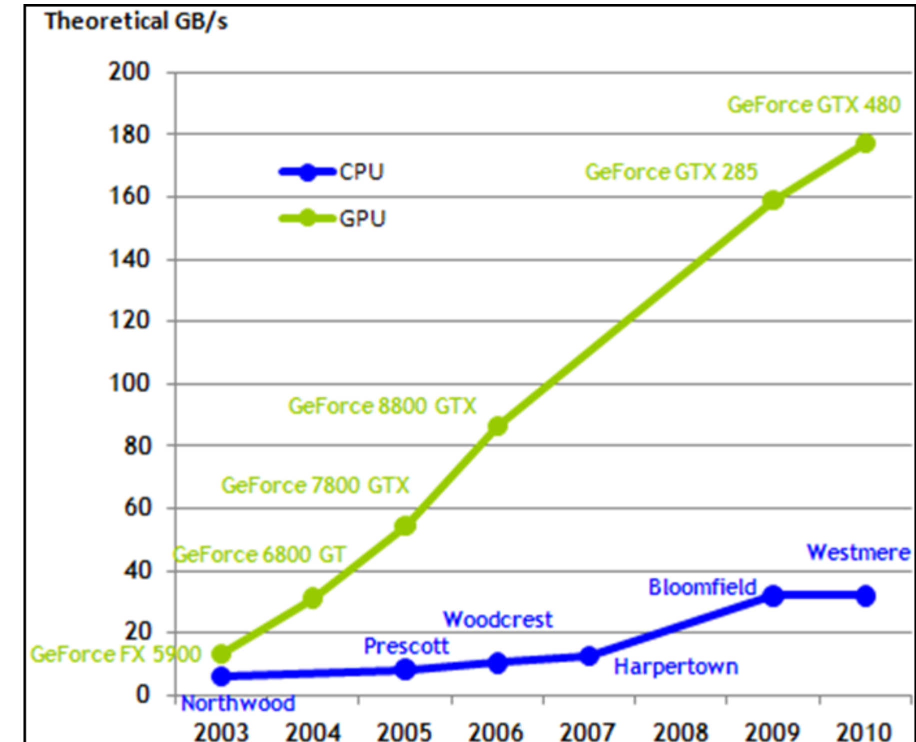
UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

PERFORMANCE !

puissance (GFLOPs)



mémoire (GB/s)



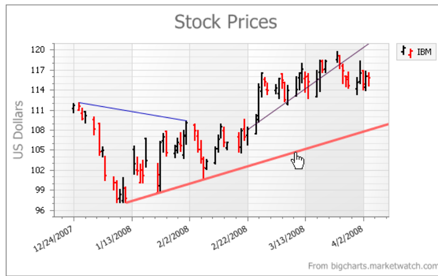
Credit: NVIDIA Programming Guide

- puissance de calcul supérieure aux processeurs classique (simple et double précision)
- accélérateurs matériels

- bande passante mémoire très importante (*mais juste à l'intérieur du GPU*)
- caractéristiques adaptées à ***certains*** types de problèmes

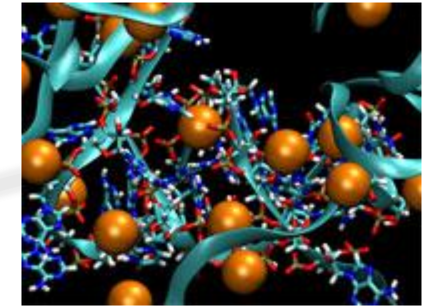
UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

GPU Computing



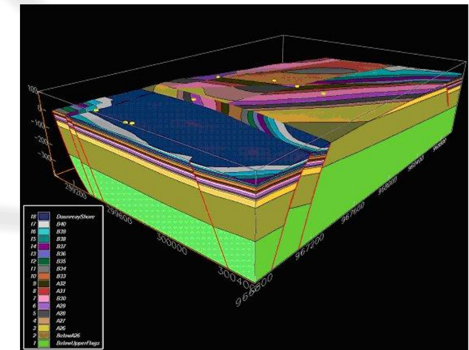
Analyse financière

Simulation météo


Docking moléculaire

Imagerie médicale

Détection de logiciels malveillants


Modélisation géologique

Programmation de GPU

Initialement – utilisation d'appels graphiques :

- *Cg* de NVIDIA
- *HLSL* de Microsoft

Aujourd'hui – écosystème de programmation de GPU :

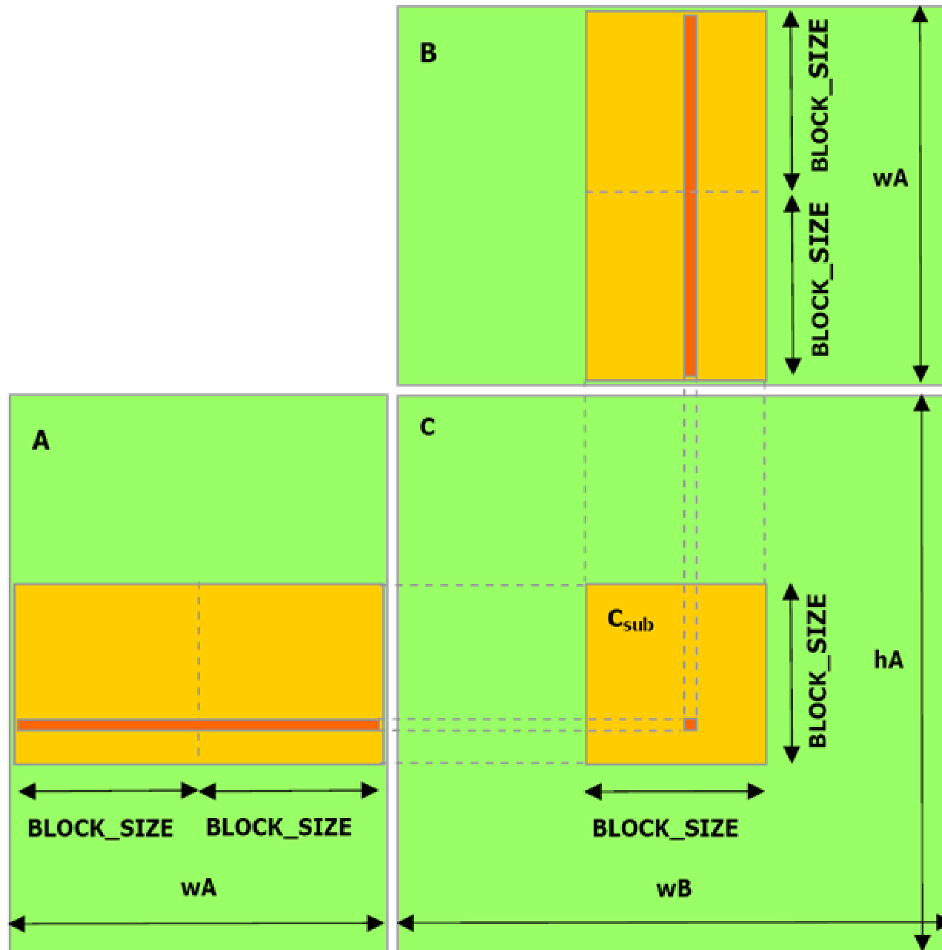
- *ATI Stream* de AMD
- *CUDA* de NVIDIA
- *OpenCL* de Khronos Group
- *Direct Computing* de Microsoft

CUDA (bloqué sur le matériel NVIDIA = *aucune portabilité !)*

- = "**C**ompute **U**nified **D**evice **A**rchitecture"
- extension du standard ANSI C
- plus facile à apprendre (comparé à Cg, HLSL)
- ouvre l'architecture et donne à l'utilisateur un accès complet aux ressources

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

Produit matriciel classique sur GPU (CUDA)



Produit matriciel C = A x B

- correspond au modèle d'exécution ($C_{sub} \leftrightarrow$ block thread)
- utilise les différentes hiérarchies mémoire (mémoire globale, mémoire partagée)

- problème standard de calcul intensif $\Rightarrow$ notre étude

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

Sommaire

I. **Introduction (GPU & multiGPU)**

II. **Gestion des architectures multi-cartes**

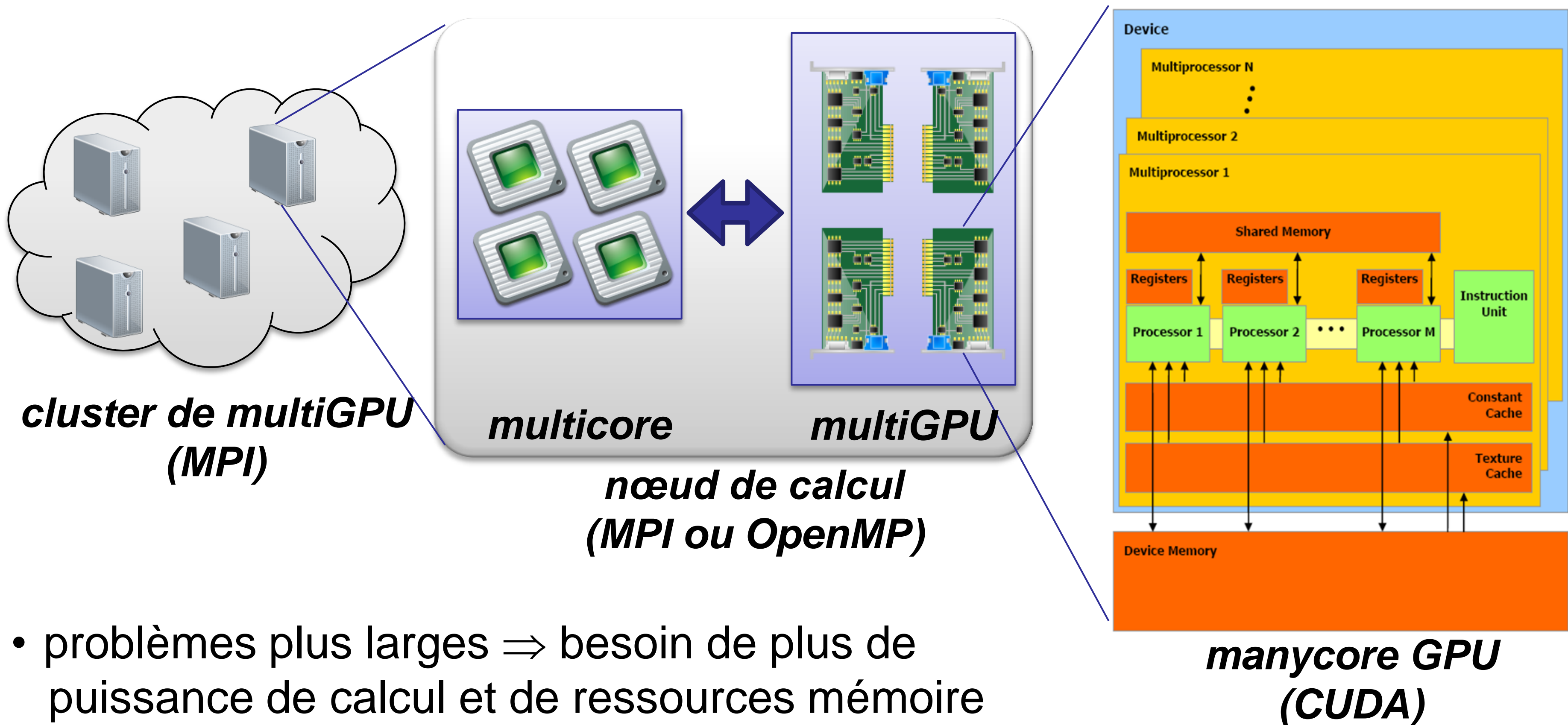
- *Environnement multiGPU*
- *Contrôle des GPU avec OpenMP ou MPI*
- *Produit matriciel (strategie MPI versus OpenMP)*

III. **Résultats expérimentaux**

IV. **Conclusions**

Environnment multiGPU

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE



- problèmes plus larges ⇒ besoin de plus de puissance de calcul et de ressources mémoire
- applications “simples” | multithread / multiprocesseur
  - OK avec n’importe quelle API multithread ou avec passage de messages
  - régions de calcul intensif déportées sur GPU
  - multithreading $\perp$ CUDA

Contrôle des GPU

	MPI	OpenMP
environnement	processus	thread
mémoire	distribuée	partagée
échange des données	explicite (passage des messages)	implicite (mémoire partagée)
synchronisation	explicite	explicite / implicite
attachement à une carte graphique	rang du processus	identifiant du thread

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

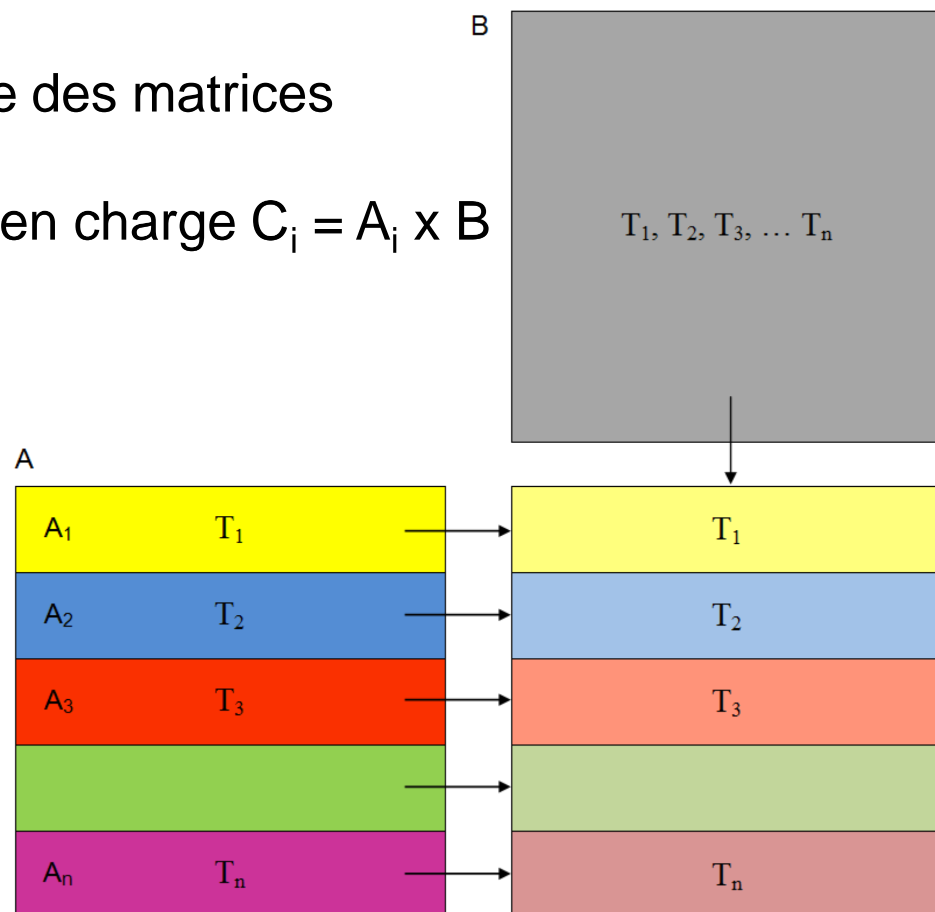
Produit matriciel (1)

Produit matriciel C = A x B

- matrices des grandes tailles
  - ⇒ diviser le travail ⇒ découpage des matrices
- partitionner la matrice A
  - ⇒ chaque unité de calcul prend en charge $C_i = A_i \times B$

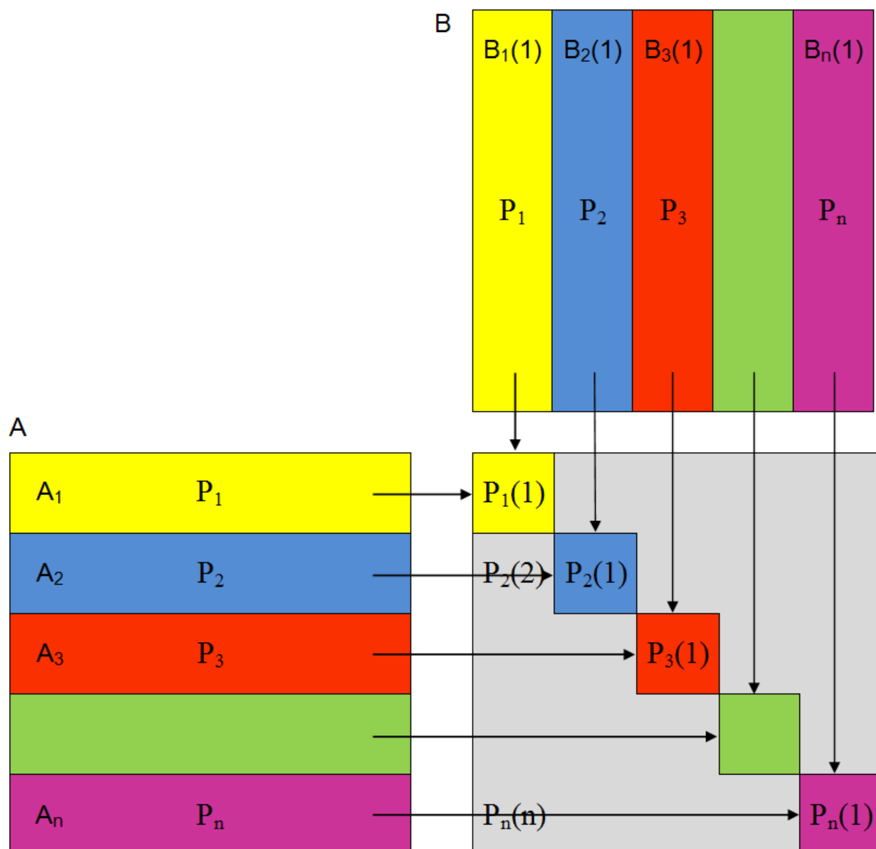
Partitionnement OpenMP

- mémoire partagée
  - ⇒ plus de mémoire (nœud entier)
  - ⇒ matrice B non partitionnée
  - ⇒ pas d'effort du communication
- chaque thread s'attache à une carte
  - ⇒ allocation mémoire sur GPU
  - ⇒ transfert des blocs nécessaires (CPU → GPU)
  - ⇒ calcul (GPU)
  - ⇒ récupération des résultats (GPU → CPU)



UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

Produit matriciel (2)



Partitionnement MPI

- mémoire distribuée
  - ⇒ matrice B découpée
  - ⇒ sous-matrices de B cyclées sur les processus
  - ⇒ celles de A restent fixes
- algorithme

⇒ processus *i* attaché au GPU_i

⇒ allocation mémoire (GPU) + transfert A_i

⇒ [à chaque pas]

- transfert B_j (CPU→GPU)
- $C_{ij} = A_i \times B_j$ (GPU)
- transfert C_{ij} (GPU→CPU)

|| transfert B_j ($CPU_i \rightarrow CPU_{i-1}$)

- communications MPI

⇒ buffering

- stratégie du préchargement
  - recouvrement des communications MPI par des calculs GPU
  - coût de communication nul

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

Sommaire

I. Introduction (GPU & multiGPU)

II. Gestion des architectures multi-cartes

III. Résultats expérimentaux

- *Conditions expérimentales*
- *Comportement général (mesures de temps, irrégularités)*
- *Aspects spécifiques (pinned memory, recouvrement)*
- *Accélérations & limites des applications*

IV. Conclusions

Conditions expérimentales

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

Nombre du threads

- 1, 2 ou 4 threads/processus pour contrôler autant de cartes
  4 threads/processus = utilisation complète du Tesla S1070
- 8 threads/processus ⇒ situations de compétition
  deux threads/processus accèdent à une carte en même temps

Les temps mesurés

- **temps kernel** = temps nécessaire à calculer un bloc-ligne C_i
- **temps de transfert mémoire CUDA** = temps total nécessaire pour les transferts CPU-GPU / GPU-CPU
- **temps application** = temps total de l'application (initialisation de l'environnement d'exécution, initialisations des matrices, transferts mémoire, calculs)

Architecture matérielle

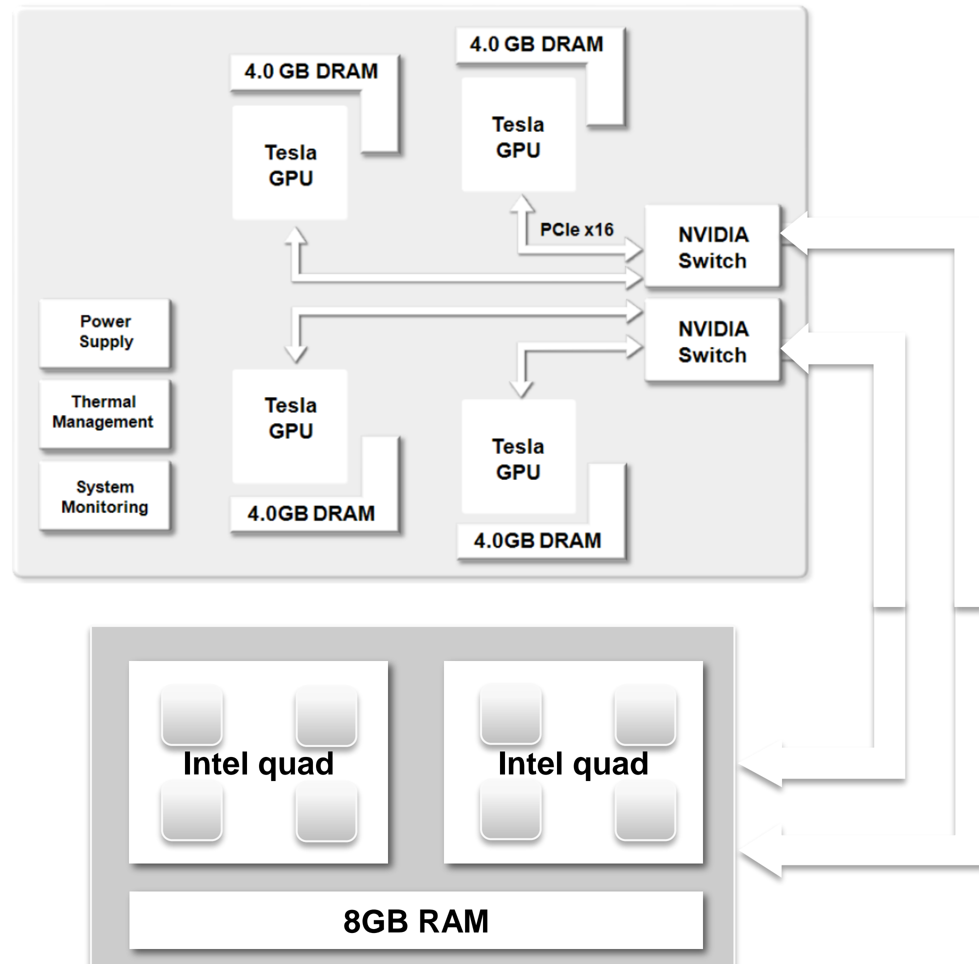
ROMEO – Centre de Calcul de Champagne-Ardenne

NVIDIA Tesla S1070

- 4 cartes graphiques “standard”
- 4Go DRAM / carte
- interconnectées par deux bridges PCI Express
- deux cartes d'interface

Serveur Bull NovaScale R425

- 2 quad-core Intel Xeon E5472
- 8Go mémoire vivre DDR3



Comportement général

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

OpenMP (1, 2, 4 threads)

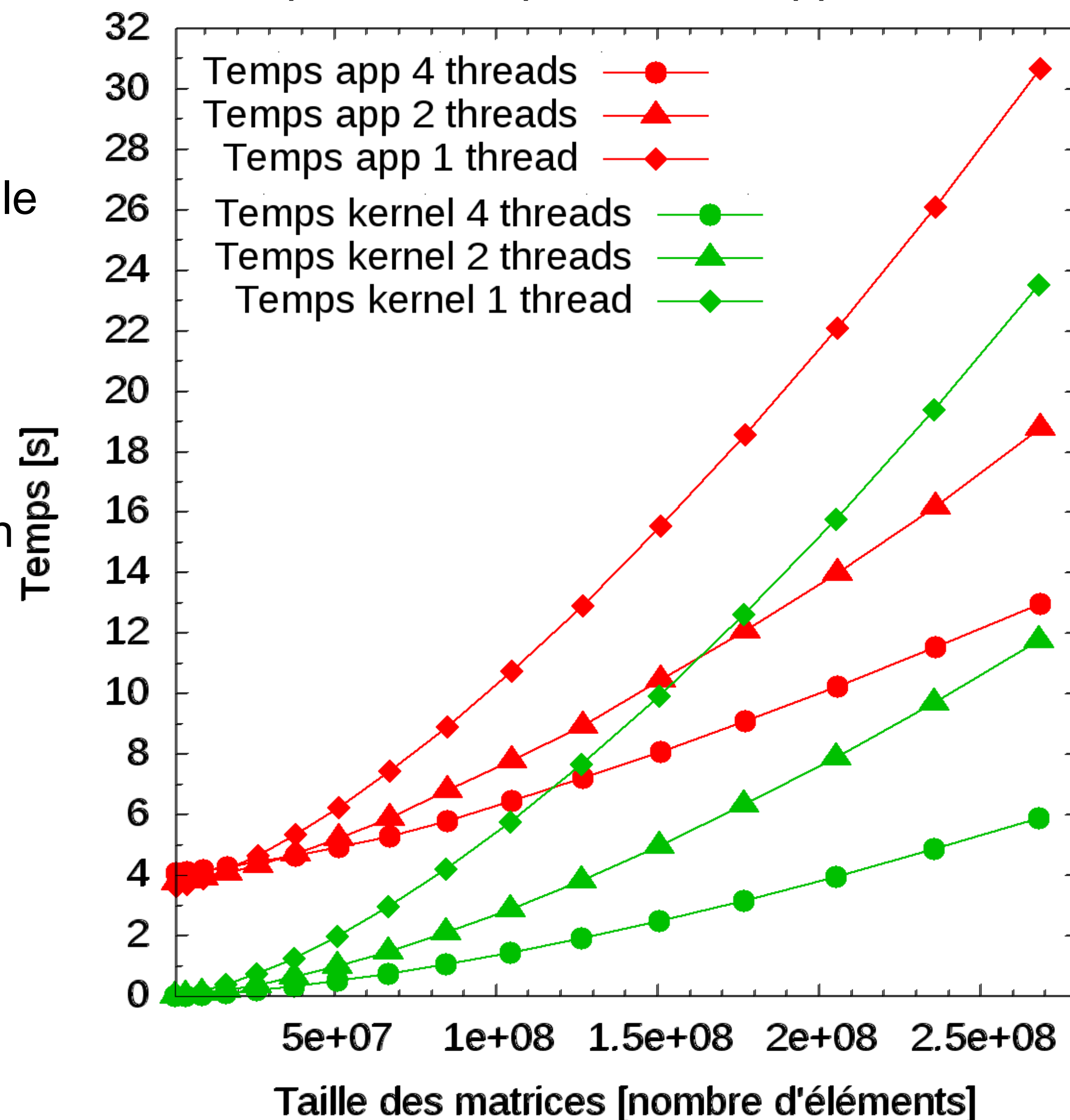
Temps kernel :
constamment divisé par 2
car le nombre de GPU double
(dim. sous-matrices / 2)

Temps application :
surcoût constant
- initialisation des environnements d'exécution
- allocations & initialisations de la mémoire hôte

Temps transfert CPU-GPU :
- faible
- croissance constante

MPI

- même tendance
- surcoût plus important



UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

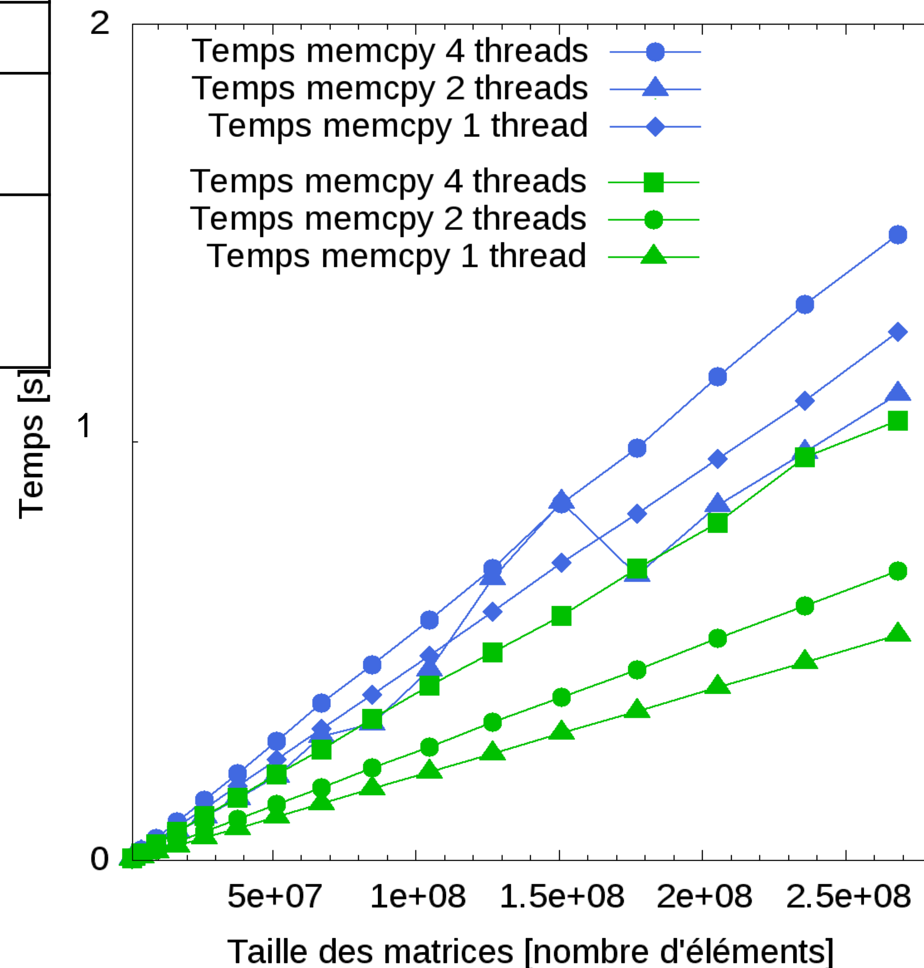
Types d'allocation mémoire

	malloc clasique	pinned memory
environ-nement	POSIX	CUDA
fonction	malloc	cudaHostAlloc
localisation	adresse variable (pagination)	même adresse
swap	oui	non + permet des transferts zero-copy

Pinned memory

avantages	inconvénients
plus rapide	taille limitée
temps d'accès plus réguliers (pas de swap)	blocage du système en cas de sur-utilisation

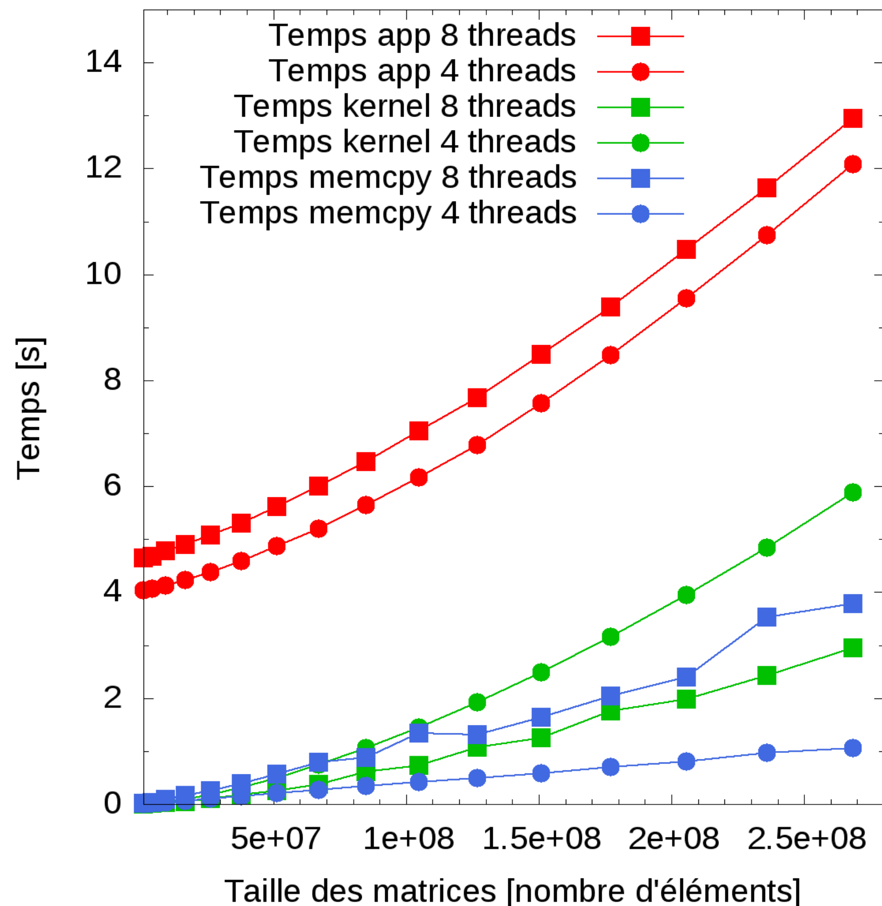
OpenMP temps memcopy (malloc / cudaHostAlloc)



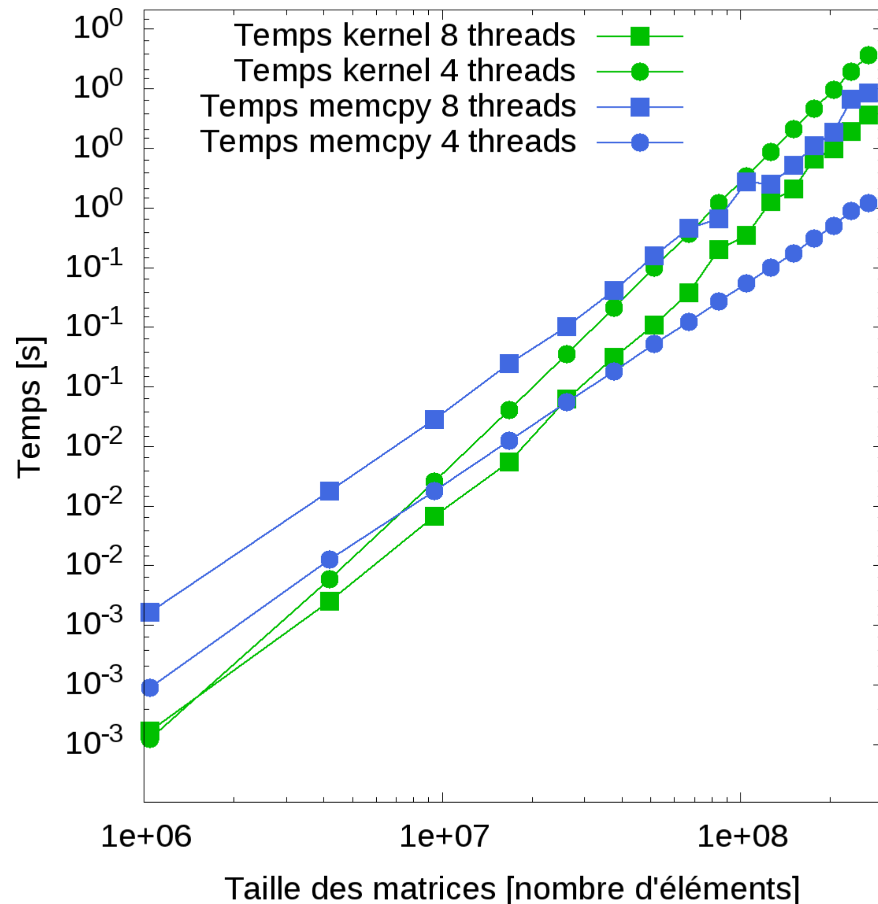
UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

8 threads : concurrence pour ressources

Temps OpenMP – pinned memory



Temps OpenMP – pinned memory (échelle logarithmique)



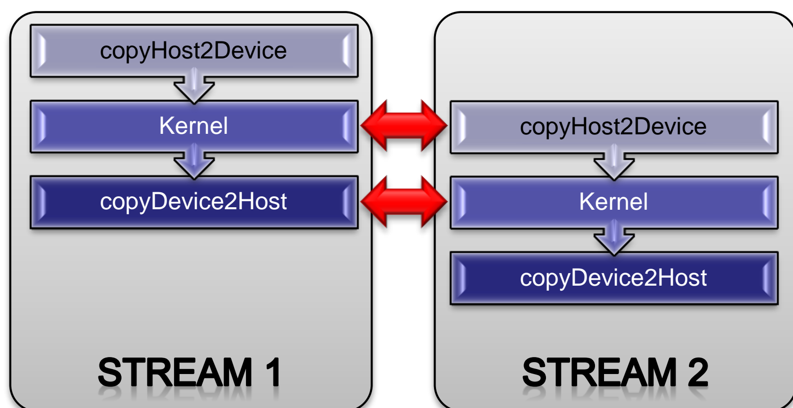
- temps kernel divisés par 2 (comme prévu)
- temps mémoire doublés à cause de la concurrence
- plus irrégulier (voir l'échelle logarithmique)

Université de Reims Champagne-Ardenne

Recouvrement des transferts mémoire GPU

Principe :

- superposer les transferts CPU-GPU & l'exécution kernel
- *streams* (concept proche des threads)
- restriction *: pinned memory obligatoire*



Dans notre cas :

- Temps kernel >> temps transferts

$\Rightarrow$ aucune amélioration

- ***Toujours envisager cette possibilité !***

Temps OpenMP – pinned memory (échelle logarithmique)

Temps kernel 4 threads
Temps kernel 2 threads
Temps kernel 1 thread
Temps memcpy 4 threads
Temps memcpy 2 threads
Temps memcpy 1 thread

Temps [s]

10^{1} 10^{1} 10^{0} 10^{0} 10^{0} 10^{0} 10^{-1} 10^{-1} 10^{-1} 10^{-2} 10^{-2} 10^{-2} 10^{-3} 10^{-3} 10^{-3}

1e+06 1e+07 1e+08

Taille des matrices [nombre d'éléments]

Accélérations & limites des applications

Limites des applications

- limites mémoire vive hôte
  vs. mémoire device
- différentes stratégies de partitionnement
  ⇒ différentes limites mémoire
  (implémentation MPI ⇒
  2 matrices pour le buffering)
  ⇒ OpenMP est meilleur

Dim. matrice	MPI				OpenMP			
th / pr	1	2	4	8	1	2	4	8
18432								
20480								
22528								
24576								

Accélérations

- 4 threads/processus = utilisation optimale du Tesla S1070 (4 cartes)
- dim. matrices ≤ 18432x18432
  (plus large ⇒ aucun résultat pour un seul thread/processus)
- efficacité : OpenMP = 64% *vs.* MPI = 50.5%
- OpenMP meilleur que MPI (augmentation en taille matrice ; + surcoût MPI)

Accélérations (4 threads/processus vs. 1 thread/processus)

Matrix dim	1024	2048	4096	6144	8192	10240	12288	14336	16384	18432
OpenMP	0.89	0.90	0.96	1.11	1.34	1.63	1.92	2.18	2.40	2.57
MPI	0.96	0.97	1.03	1.16	1.37	1.60	1.83	2.05	2.08	2.02

Efficacité

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

- limite, mais juste par la taille mémoire
- autrement, croissance linéaire

Performances obtenues suivant la taille des données

Matrix dim. n (x 1024)	Operations n^3 (x 1024^3)	Time t(s)	Measured GFlops
1	1	4.03	0.27
2	8	4.06	2.12
10	1000	6.16	174.31
12	1728	7.57	245.10
22	10648	23.85	479.38
24	13824	29.40	504.88

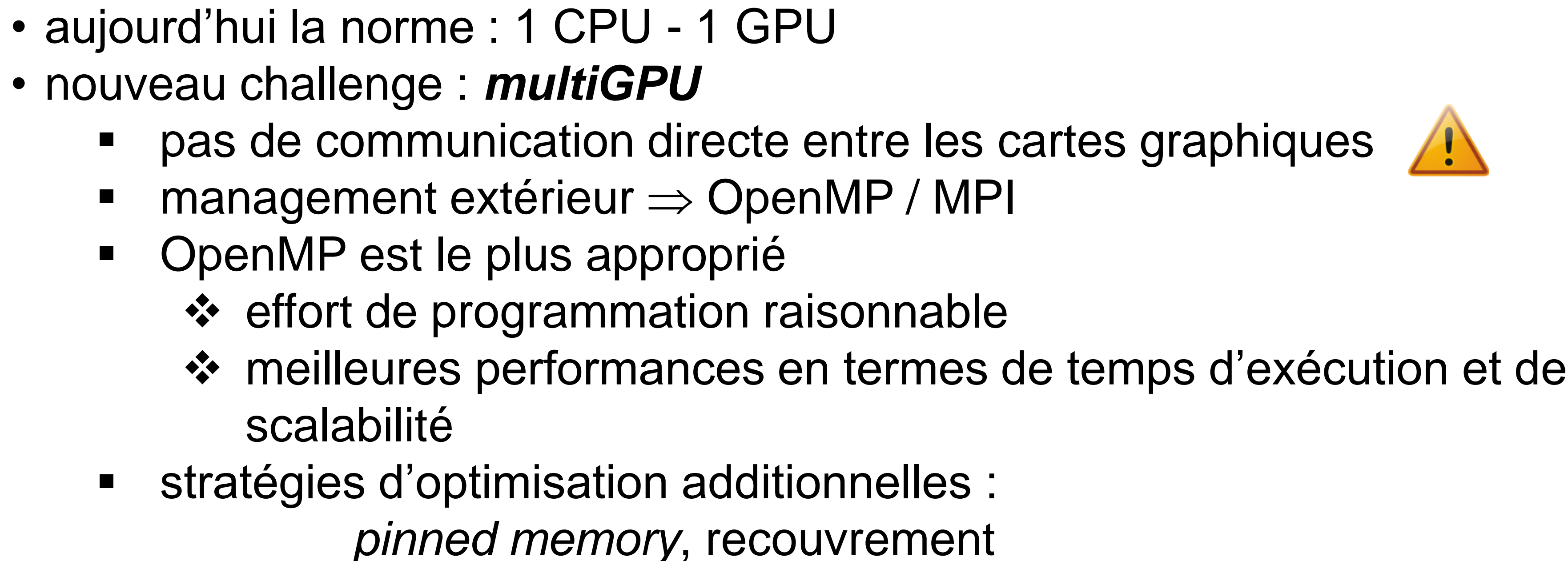
Sommaire

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

I. Introduction (GPU & multiGPU)
II. Gestion des architectures multi-cartes
III. Résultats expérimentaux
IV. Conclusions
- *Discussion finale*
- *Perspectives*

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

Conclusions & perspectives



- aujourd'hui la norme : 1 CPU - 1 GPU
- nouveau challenge : ***multiGPU***
  - pas de communication directe entre les cartes graphiques
  - management extérieur ⇒ OpenMP / MPI
  - OpenMP est le plus approprié
    - effort de programmation raisonnable
    - meilleures performances en termes de temps d'exécution et de scalabilité
  - stratégies d'optimisation additionnelles :
    pinned memory, recouvrement

Perspectives

= grappes de nœuds multiGPU, avec trois niveaux :

- communication inter-nœuds ⇒ processus MPI (mémoire distribuée)
- gestion d'un nœud multiGPU ⇒ threads OpenMP (mémoire partagée)
- calcul, au sein des GPU

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

SYSCOM SYSTÈMES COMMUNICANTS

Programmation multiGPU OpenMP *versus* MPI

Questions ?

Contact: gabriel.noaje@univ-reims.fr

cea

RÉGION CHAMPAGNE ARDENNE