RETOUR D'EXPERIENCE GPU CARTES GRAPHIQUES PROGRAMMABLES EN CALCUL HAUTE PERFORMANCE

Hervé Jourdren, Marc Pérache

CEA/DIF/DSSI

Clément KOYESSE

Stage Master 2 Informatique Professionnelle

Université de Versailles St Quentin

CEA/DIF

Utilisation des technologies alternatives de type GPU ou Cell dans le domaine du calcul scientifique

11 avril 2008

Plan

1. **Description et intérêt**
2. **Mesures de performance**
3. **Limitations**
4. **Conclusions**

Enjeux de l'approche GPGPU

- **Approche GPGPU**
  - Pipeline graphique parallélisé
    - Opérations indépendantes sur les pixels et/ou vertex
  - Prédominance des calculs sur les opérations mémoires
  - Extension récente du domaine d'application
    - Calculs scientifiques
    - Calculs financiers

- **Solutions disponibles**
  - ATI Close To Metal
    - R600 475 GFLOP/s (crête)
    - ...
  - NVIDIA Compute Unified Device Architecture
    - G80 518 GFLOP/s (crête)
    - ...

Évolutions récentes GPU

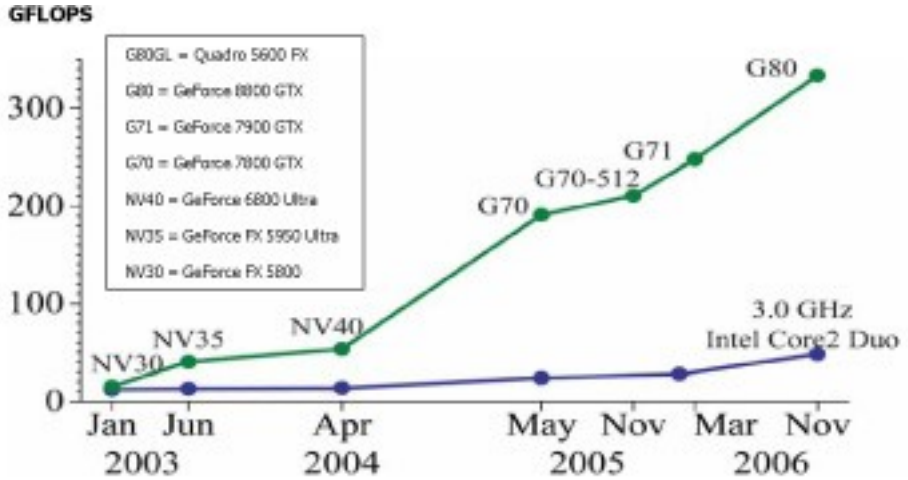
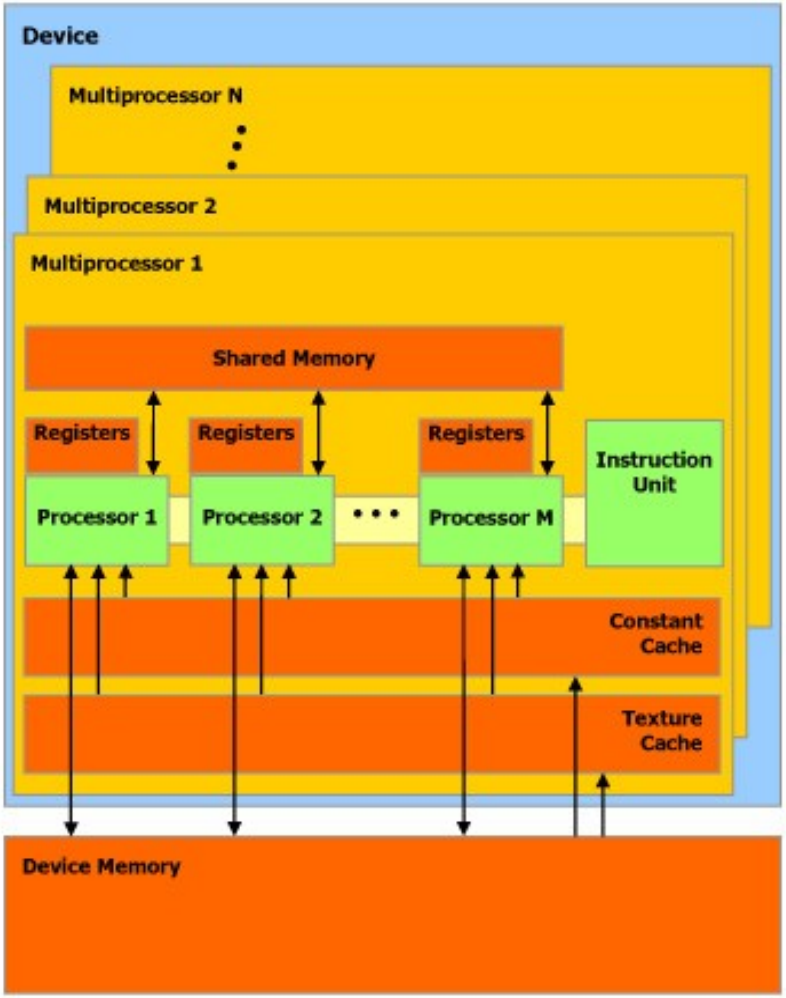


Figure 1-1. Floating-Point Operations per Second for the CPU and GPU

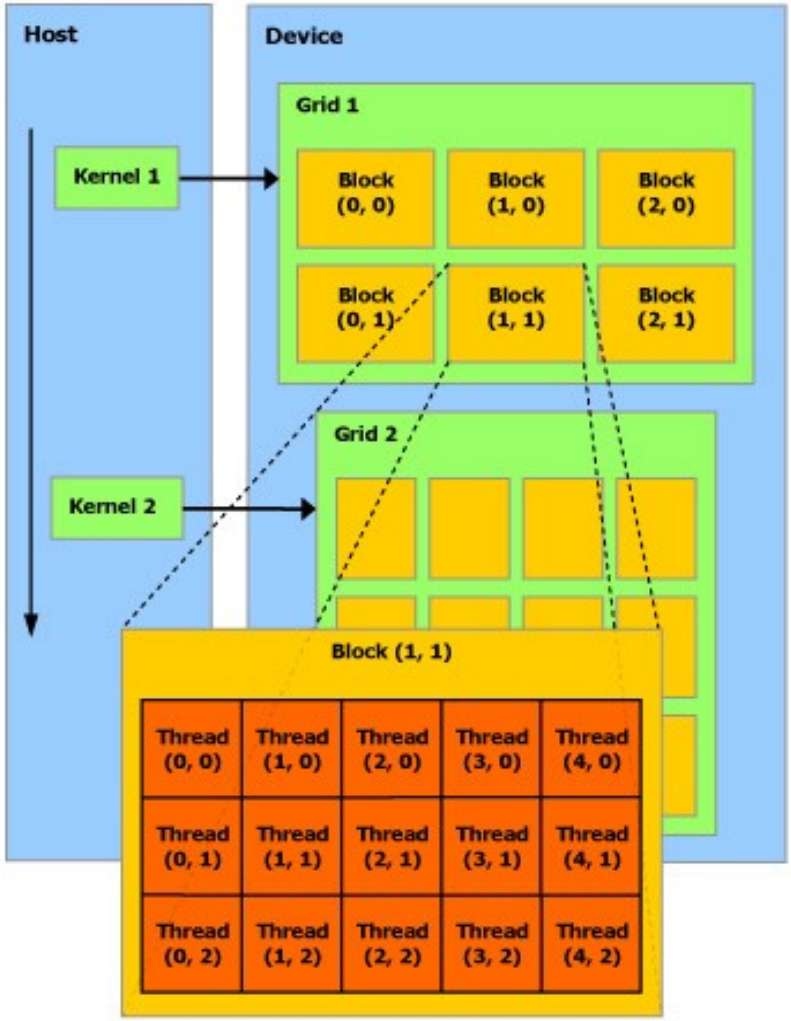
Figure 1-2. The GPU Devotes More Transistors to Data Processing

Architecture G80 Nvidia



A set of SIMD multiprocessors with on-chip shared memory.

Figure 3-1. Hardware Model



The host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks

Figure 2-1. Thread Batching

API CUDA Nvidia (1/3)

- **API CUDA**
  - Disponible à partir du GPU G80
    - Gamme Cartes Graphiques DirectX-10
      - Quadro FX 5600 1.5 Go de DRAM
      - GeForce 8800 GTX et GTS
      - GeForce 8600, 8400, etc
    - Gamme GPGPU Tesla
  - Langage de haut niveau
    - Plus de programmation de shaders OpenGL
    - Extension SIMD du langage C
  - Environnement d'exécution
  - Mode émulation sur CPU (*)

(*) comportement légèrement différent par rapport à la carte, chaque thread étant executé séquentiellement

Exemple de multiplication d'une matrice par une constante version CPU:

```
void CMmul(float a, int len, float A[]){

  for(int i = 0; i < len; i++){
    for(int j = 0; i < len; i++){
      A[i][j] = a * A[i][j];
    }
  }
}
```

Exemple de multiplication d'une matrice par une constante version GPU:

```
void __global__ CMmul_KER(float a, float A[]){

//kernel executé en mode SIMD sur le device

  int tx = threadIdx.x + threadIdx.y * blockDim.x; //calcul du rang

  A[tx] = a * A[tx];
}
```

```
  float A = (float*) malloc(N*sizeof(float));

  //...initialisation du tableau

  //allocation sur la carte
  float *A_d;
  cudaMalloc( (void**) &A_d, N*sizeof(float));

  //copie vers la carte des données
  cudaMemcpy(A_d, A, N*sizeof(float));

  //appel du kernel
  CMmul_KER <<< N/256, 256>>> (a,A);

  //rapatriement des résultats
  cudaMemcpy(A, A_d, N*sizeof(float));
```

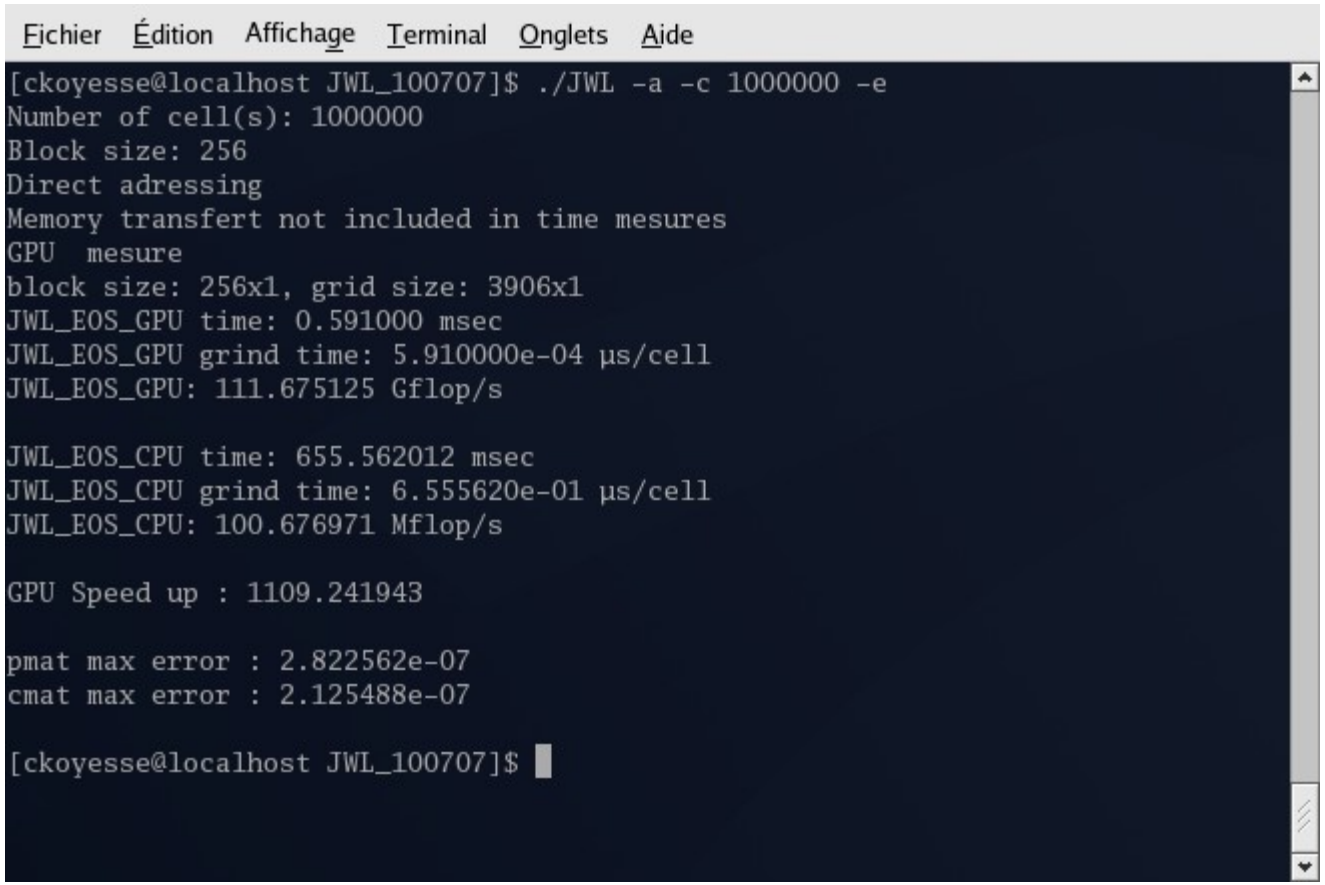
Mesures de performance : EOS JWL (1/4)

- **Equation d'état JWL (Jones–Wilkins–Lee)**
  - Produits de détonation
  - Calcul de la pression et de la vitesse du son en fonction de la densité, de l'énergie interne et de la fraction brûlée
  - Équation d'état relativement coûteuse avec plusieurs exponentielles (en version CPU, 10% du coût d'un solveur de dynamique des gaz)

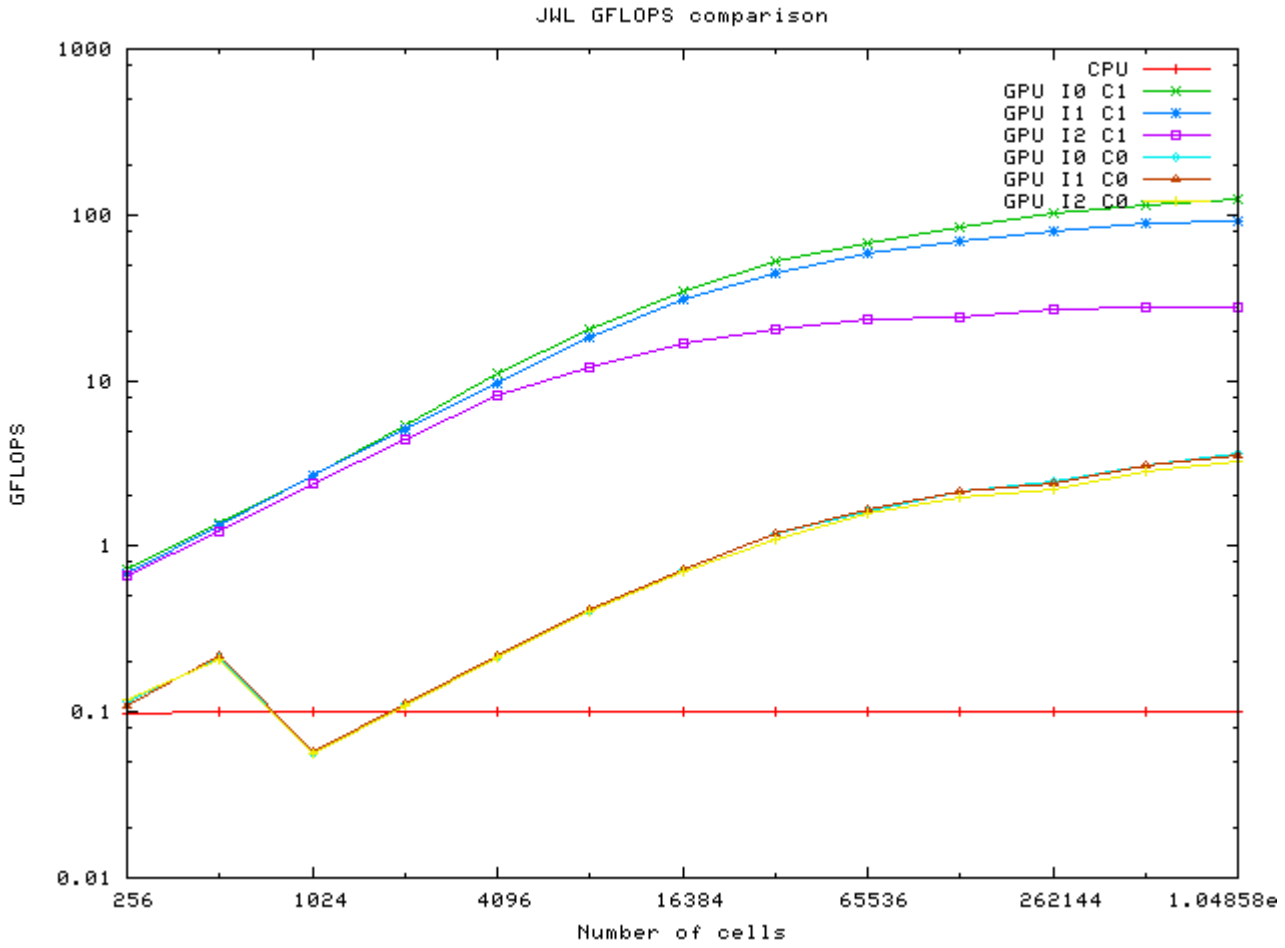
- **Condition des mesures**
  - Version CPU
  - Version GPU
    - Mesure avec et sans transferts
    - Vecteur d'indirection

- **Écrit en « CUDA naïf »**

Mesures de performance : EOS JWL (2/4)



Mesures de performance : EOS JWL (3/4)



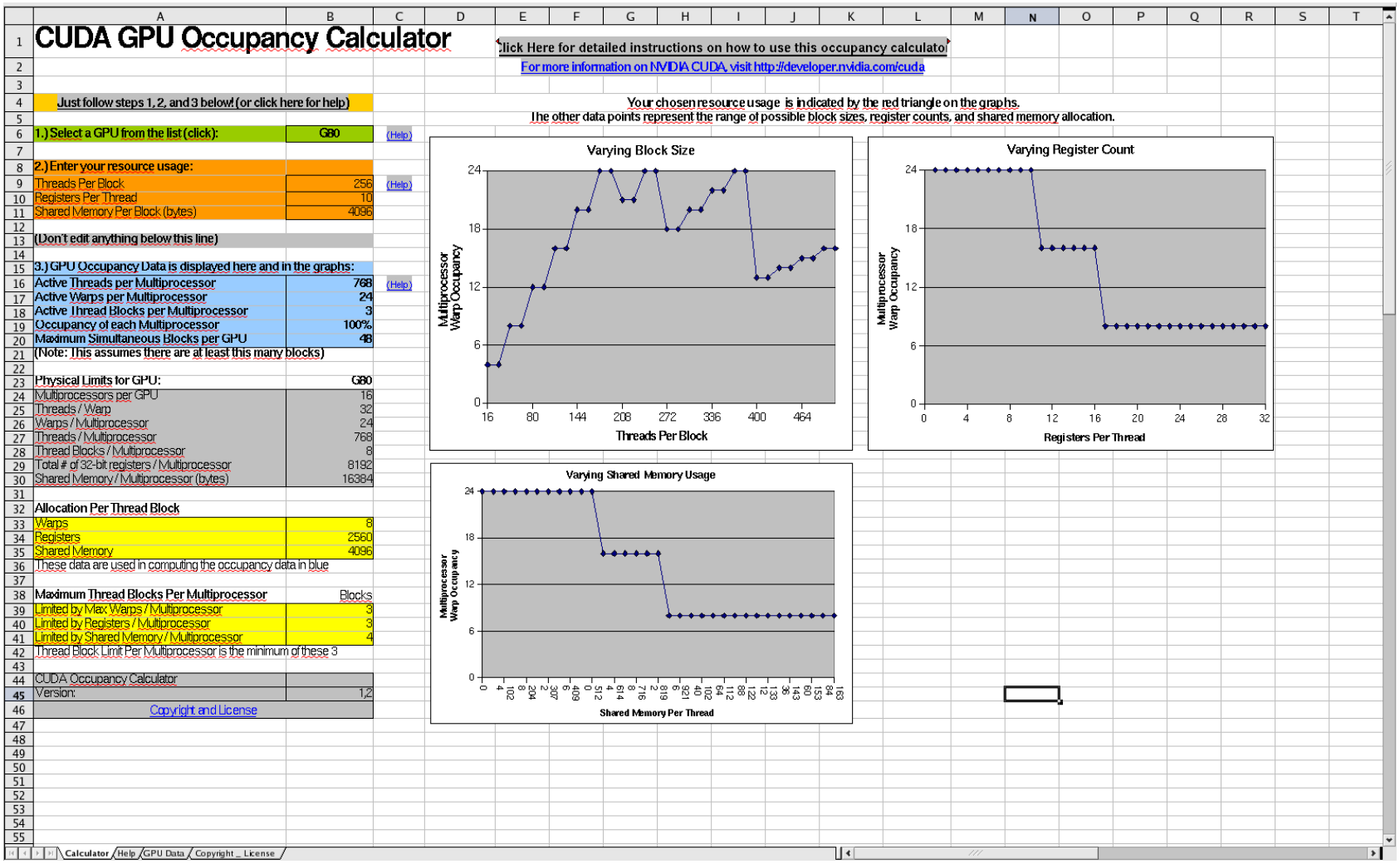
Mesures de performance : EOS JWL (4/4)

- 100 **GFLOP/s simple précision en adressage direct et tableaux résidants sur la carte**
- **Très léger abattement** (99 **GFLOP/s**) **avec « fausse » indirection index**[**i**] = **i**
- **Facteur** 3-4 **d'abattement** (27 **GFLOP/s**) **avec « vraie » indirection, mais encore** 2 **ordres de grandeur de gain par rapport à la version CPU** (0.1 **GFLOP/s**)
- **En mode déporté avec transferts CPU–GPU/GPU–CPU, l'écart n'est plus que d'un ordre de grandeur** (3.5 **GFLOP/s**), **conséquence de la bande passante limitée du port PCIE**

- **Solveur Lagrange 1D**
  - Équations de la dynamique des gaz
  - Solveur GAD de type Godunov ordre 2 avec limiteur

- **Condition des mesures**
  - Gestion optimale des différentes mémoires
    - Shared memory
    - Texture memory
  - Block bi-dimensionnels
    - 16x16
  - Implémentations avec et sans mailles fantômes
  - Taille de block optimale
    - CUDA GPU occupancy calculator

Outil d'optimisation



CUDA GPU Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator

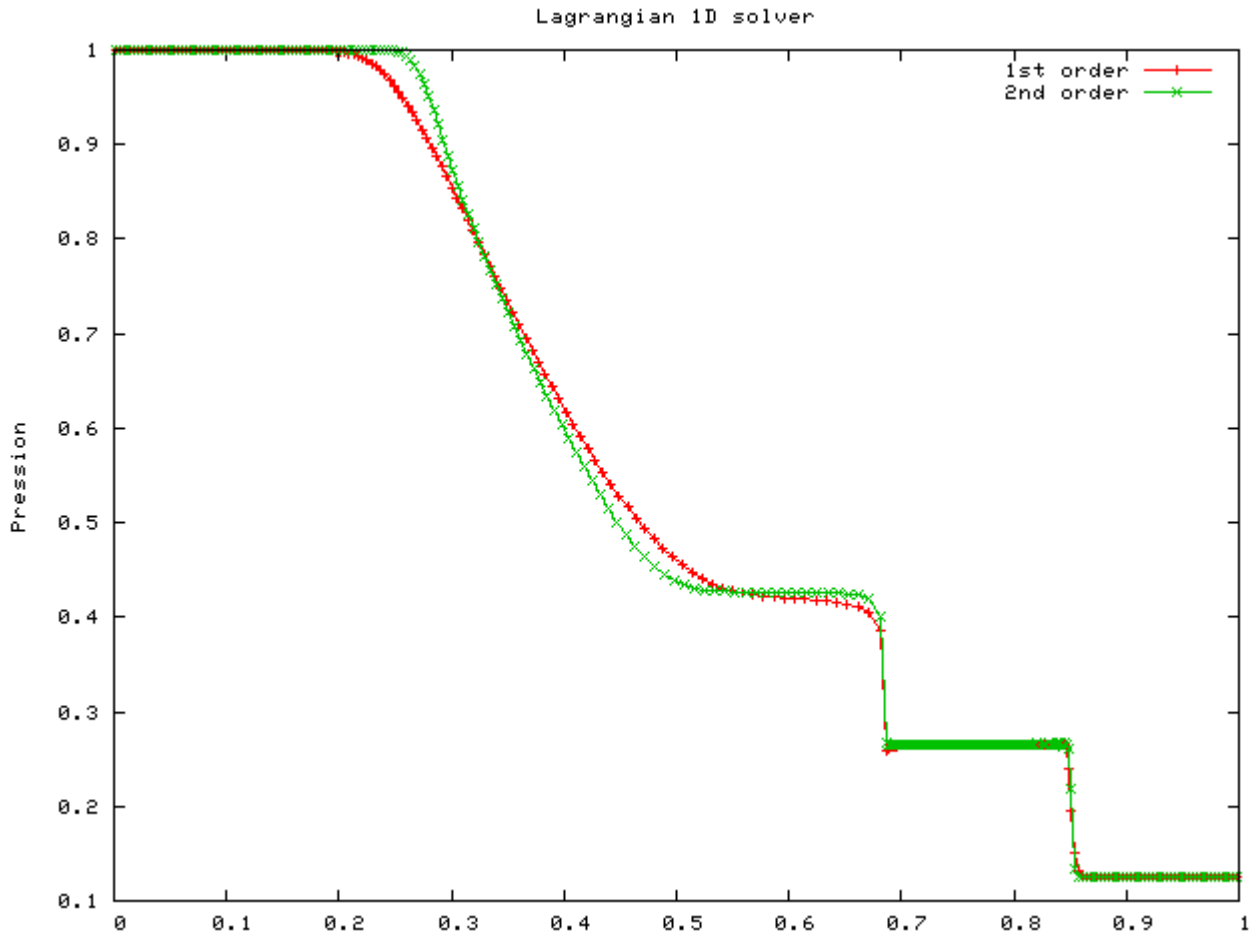
For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Just follow steps 1, 2, and 3 below! (or click here for help)

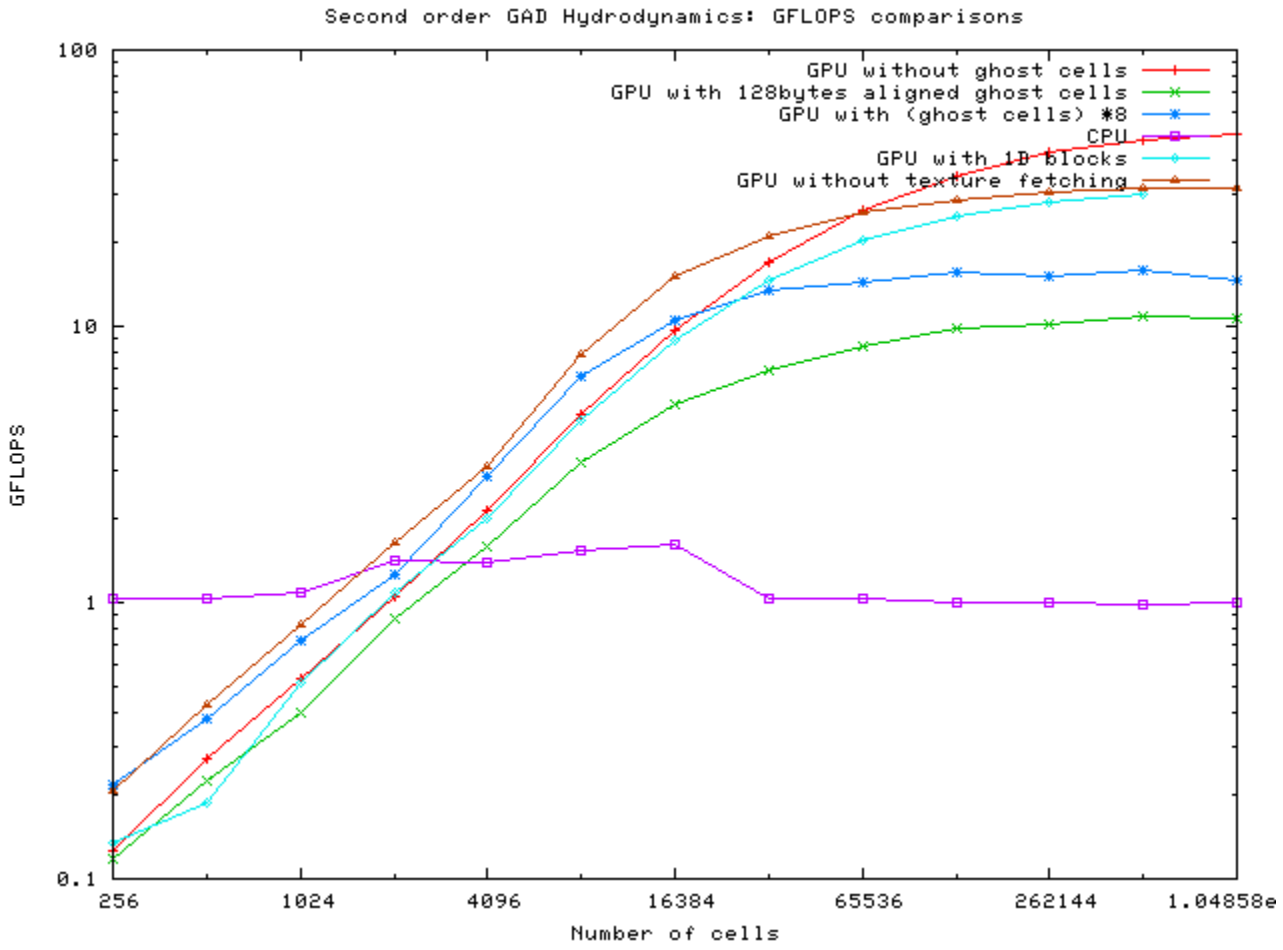
Your chosen resource usage is indicated by the red triangle on the graphs.
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

1.) Select a GPU from the list (click):	G80
2.) Enter your resource usage:	
Threads Per Block	256
Registers Per Thread	10
Shared Memory Per Block (bytes)	4096
(Don't edit anything below this line)	
3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	768
Active Warps per Multiprocessor	24
Active Thread Blocks per Multiprocessor	3
Occupancy of each Multiprocessor	100%
Maximum Simultaneous Blocks per GPU	48
(Note: This assumes there are at least this many blocks)	
Physical Limits for GPU:	G80
Multiprocessors per GPU	16
Threads / Warp	32
Warps / Multiprocessor	24
Threads / Multiprocessor	768
Thread Blocks / Multiprocessor	8
Total # of 32-bit registers / Multiprocessor	8192
Shared Memory / Multiprocessor (bytes)	16384
Allocation Per Thread Block	
Warps	8
Registers	2560
Shared Memory	4096
These data are used in computing the occupancy data in blue	
Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Multiprocessor	3
Limited by Registers / Multiprocessor	3
Limited by Shared Memory / Multiprocessor	4
Thread Block Limit Per Multiprocessor is the minimum of these 3	
CUDA Occupancy Calculator	
Version:	1.2
Copyright and License	

Mesures de performance : solveur hydrodynamique (2/4)



Mesures de performance : solveur hydrodynamique (3/4)



Mesures de performance : solveur hydrodynamique (4/4)

- 50 **GFLOP/s sans mailles fantômes**
- **Facteur 3 à 4 d'abattement (13 GFLOP/s) avec les mailles fantômes en dépit d'un alignement sur 128 octets**
- **A partir de 2^{16} mailles**
  - Utiliser des Blocks 2D plutôt que 1D
  - Utiliser le cache de texture
- **Meilleurs résultats de l'alignement 'simple'(*) par rapport aux recommandations de Nvidia**

(*) le nombre de mailles fantômes est égale à 8 fois le numéro d'ordre

Principales limitations

- **Taille block**
  - 512 threads / block
- **Taille grille**
  - 2^{40} threads / grille
- **Quantité de registres/shared memory utilisé**
- **Alignement en mémoire des données en cas d'indirection**
- **Bande passante**
  - PCIE 1.1 (4 Go/s)
  - PCIE 2.0 (8 Go/s)

- **Calculs simple précision non-IEEE**
- **Pas de code de correction d'erreur pour la mémoire GPU**

Conclusions de l'étude

- **Très bonnes performances en simple précision**
  - JWL : 125 GFLOP/s
  - GAD : 50 GFLOP/s
  - Schéma GAIA ordres très élevés : 200-300 GFLOP/s
- **Mise en oeuvre aisée *via* API CUDA**
  - JWL 480 lignes
  - GAD 1300 lignes
- **Mise au point optimale néanmoins assez pointue**
  - Alignement en mémoire globale
  - Conflits de bancs shared memory
  - Nombre limité de registres
  - Très longs vecteurs *indispensables*
  - Débit Port PCI-E limité (4 ou 8 Go/s)
    - Privilégier si possible les solutions tout GPU
    - Nouvelles difficultés pour de très gros maillages (programmation parallèle type Out-of-Core)