Optimisation d'algorithmes sur maillages structurés

Benoit Mathieu

March 24, 2011

Plan de l'exposé

- Motivations, objectifs
- Algorithmes : solveur multigrille géométrique et opérateurs
- Techniques d'optimisations CPU
- Conclusions et perspectives

Motivations

De nombreuses applications tirent parti des maillages structurés

- études amont sur la modélisation des milieux poreux,
- études amont sur la modélisation des écoulements diphasiques,
- études sur le risque d'entraînement de gaz dans les réacteurs GEN IV,
- étude du procédé de vitrification.

Motivations

"Y'en a marre du solveur en pression qui rame !"
Des gains de performances considérables sont possibles par rapport à l'implémentation générique des VDF dans Trio_U, notamment grâce aux solveurs multigrille.

- utilisation de solveurs multi-grille géométriques (selon la littérature : compétitif avec la FFT dans les bons cas)
- spécialisation des structures de données en IJK (amplifie l'efficacité des autres optimisations),
- optimisation CPU (cache-blocking et vectorisation).

Pourquoi commencer par le structuré ?

- Applications plus ciblées et résultats garantis sur les maillages structurés **grâce aux solveurs multigrilles, résultats publiés**.
- Pour les maillages non structurés, il faut d'abord lever la limitation du solveur de Poisson.

Objectifs de ces travaux

Les objectifs du travail réalisé sont les suivants :

- acquérir les algorithmes à niveau avec les meilleures performances publiées,
- réaliser rapidement de premières mises en œuvres dans Trio_U pour les applications grosses consommatrices de temps CPU.
- proposer des méthodes de programmation systématiques pour optimiser le code tout en conservant un niveau de maintenabilité acceptable.

Etat de l'art (sur coeurs Nehalem) :

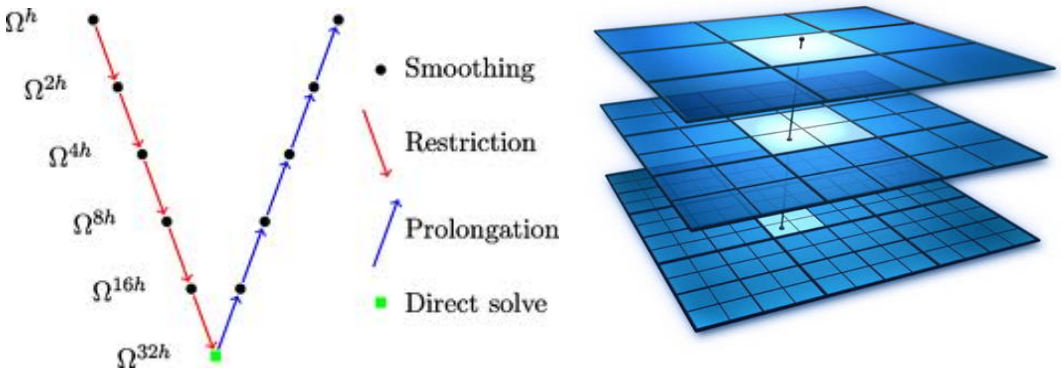
- Perf. du VDF(gcp) sur 100M mailles : 30 000 mailles/coeur, 2.5 s/dt
- Perf. réalisée avec MG géométrique : 156 000 mailles/coeur, 1.2 s/dt
- Objectif probablement atteignable : $1M$ mailles/coeur, 1 s/dt
- Perf des solveurs basés sur FFT : $2M$ mailles, 0.4 s/dt sur 1 coeur
- Perf MG publiée sur GPU : $\simeq 10M$ mailles/GPU, 0.5s sur 1 GPU.

Algorithmes : solveur multigrille géométrique et opérateurs

Solveurs multigrille

Nécessitent le choix de nombreux paramètres :

- Algorithme général: Full-MG ou GCP ou GMRES.
- Pour GCP ou GMRES, MG est utilisé comme préconditionneur. Souvent 1 V-cycle.

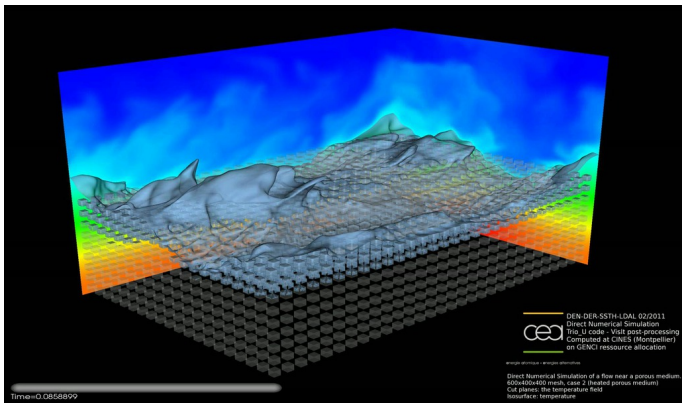


- Choix des maillages grossiers : agglomérations isotrope ou non (but: arriver à un maillage grossier isotrope pour l'équation de Poisson)
- Choix des lisseurs/interpolateurs : plus ou moins robustes (simple et robuste: Jacobi relaxé, avec 4-20 itérations selon le problème)

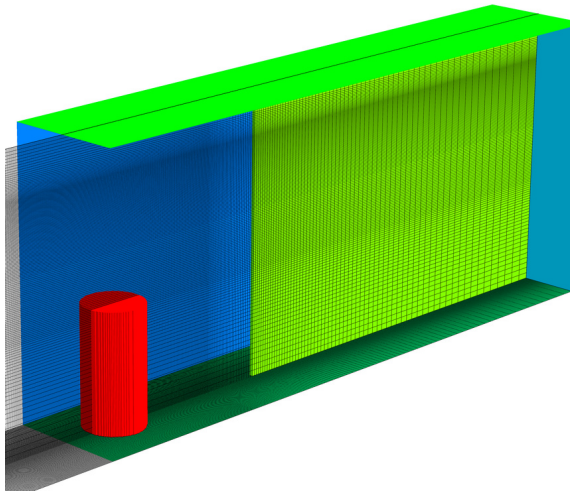
Solveurs multigrille

Etat de l'art :

- Des publications de référence pour les performances : proche d'une FFT (dans les cas très simples).
- Déjà réalisé dans Trio_U : facteur 10 à 30 plus lent que FFT, mais sur des cas complexes inaccessibles à la FFT
  - géométrie non parallélipipédique (DNS poreux, 1M h.cpu sur Jade, 100 M mailles),
  - maillage à pas variable, masse volumique variable (DNS gaz turbulent anisotherme, A. Toutant, 400 M mailles).
- Algoritmes nécessitant 10 à 30 op/maille et 3 à 6 mémoire/maille (memory bound)



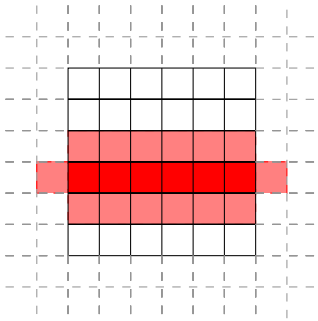
DNS poreux
MG codé et fonctionnel



Modèle numérique de Gerato (entraînement de gaz),
mise en œuvre en cours (diphasique + CL bizarres + δ_h variable)

Opérateurs

- Algorithmes utilisant un stencil et des cellules fantômes (idem pour Jacobi)



- Astuce évidente: progresser par tranche pour réutiliser les données dans les caches
- Ratio calcul / mémoire :
  exemple diffusion+convection QDM: 60-120 opérations par maille, 3 lectures, 3 écritures hors cache (limite memory-bound, il faut agglomérer plusieurs opérateurs).

Techniques d'optimisations CPU

Ratio Flops / Bande passante

Ratio du hardware
1 flop = addition, multiplication
1 mem = lit ou ecrit une valeur DP ou SP (Double precision, Simple precision)

Architecture	GFlops[1](DP/SP)	GB/s	flop/mem(DP/SP)
Intel 2x X5472@3.00GHz	96 / 192	12	**64**
Intel 2x X5570@2.93GHz	94 / 188	35	22
Intel 2x S.Bridge EP 8c@2GHz	256 / 512	102	20
BlueGene/P chip (4 cores)	13.6	13.6	**8**
NVidia S1070	87 / 1036	104	7 / 40
NVidia M2070	515 / 1030	148	28

Remarques :

- Ratio défavorable sur Intel Harpertown (64)
- Ratio confortable sur BlueGene (8)

[1] performance atteinte seulement si ADD et MUL sont en nombre égal

Ratio Flops / Bande passante

Ratio naturel de différents algorithmes

Hypothèse : le cache conserve les valeurs utilisées plusieurs fois
$\Rightarrow$ chaque valeur n'est chargée qu'une fois depuis la mémoire.

Algorithme	Flop/cell	Mem/cell [2]	flop/mem
Opérateur conv+diff QDM spécialisé	66	6	11
Itération Jacobi spécialisé[3]	10	3	3.3
Itération Jacobi (général)	20	6	3.3

Conclusions :

- ces algorithmes, pris indépendamment, sont souvent "memory bound" si le bloc de maillage ne tient pas dans le cache (environ 30 000 mailles).
- Inutile d'optimiser le codage des noyaux (par exemple avec SSE) si on ne lève pas cette limitation.

[2] une transaction mémoire supplémentaire si on n'utilise pas les instructions de streaming

[3] maillage régulier, masse volumique uniforme

Exemple de calculs nécessitant une division

Rapport entre le coût d'une division et le coût d'une addition/multiplication (pour du code vectorisé).

Architecture	facteur coût (DP/SP)
Intel X5660 VML[4]/HA [5]	44 / 28
Intel X5660 VML/LA [6]	14 / 13
BlueGene/P	120 / 120

- Exemple : algorithme où on utilise simultanément x et $1/x$

Arbitrage selon les architectures :

- précalculer $1/x \Rightarrow$ consomme plus de bande passante.
- recalculer $1/x$ à chaque usage $\Rightarrow$ consomme plus de flops.

[4] Intel Vector Math Library
[5] High Accuracy mode ($\pm$ 0.5 ULP for division)
[6] Low Accuracy mode ($\pm$ 2.5 ULP for division)

Exemple: lisseur Jacobi pour maillage isotrope, coefficient constant

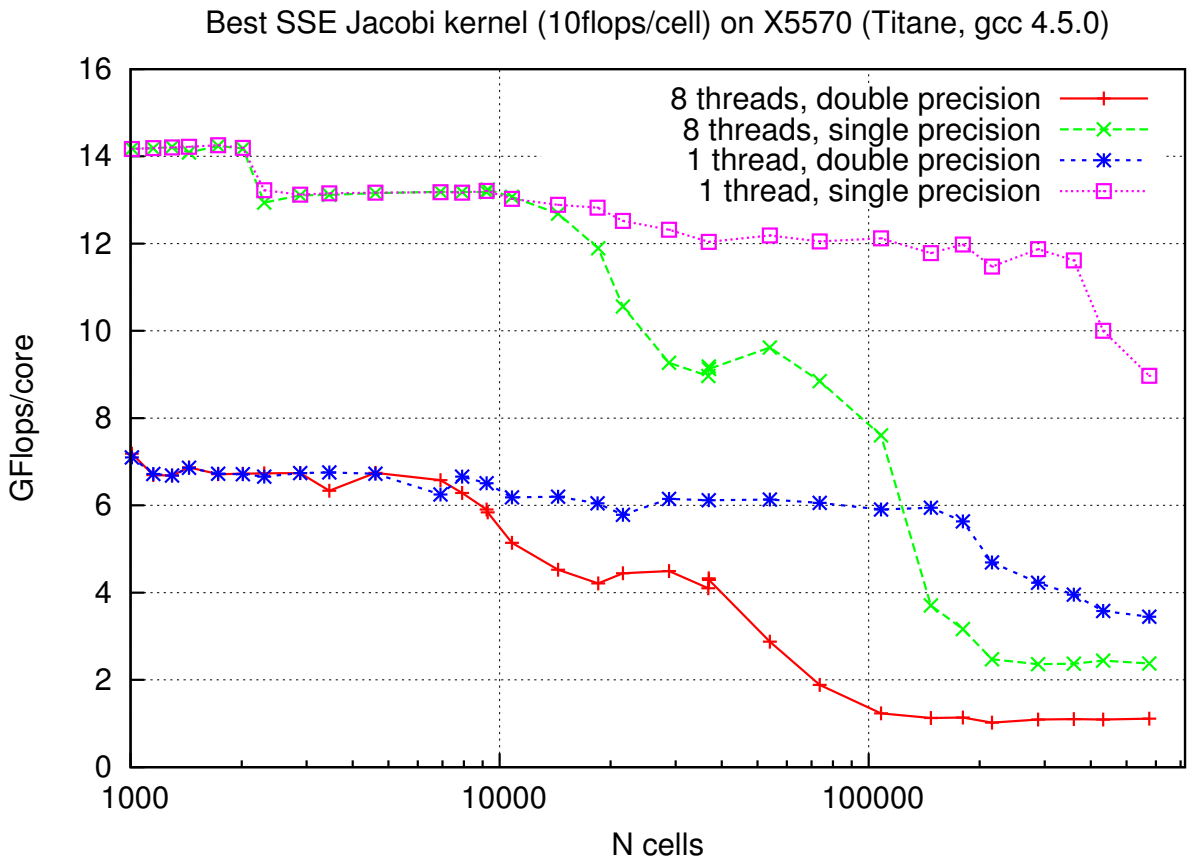
```
// Boucle sur i et j fusionnée: n = j_stride * nj
for (int i = 0; i < n; i += vsize) {
  VT vleft, vcenter, vright;
  // Lecture des valeurs gauche et droite (non alignes)
  SimdGetLeftCenterRight(ptr, vleft, vcenter, vright);
  // Somme des six valeurs des cellules voisines
  VT x = vleft + vright
    + VT::SimdGet(ptr_kmoins) + VT::SimdGet(ptr_kplus)
    + VT::SimdGet(ptr + j_stride) + VT::SimdGet(ptr - j_stride);
  // Ajout de la cellule centrale et du second membre
  x = x * coeff_extra + vcenter * coeff_center + VT::SimdGet(rhs) * coeff_rhs;
  // Ecriture du resultat
  SimdPut(resu, x);

  ptr_kplus += vsize; ptr_kmoins += vsize; ptr += vsize;
  rhs += vsize; resu += vsize;
}
```

Astuces supplémentaires

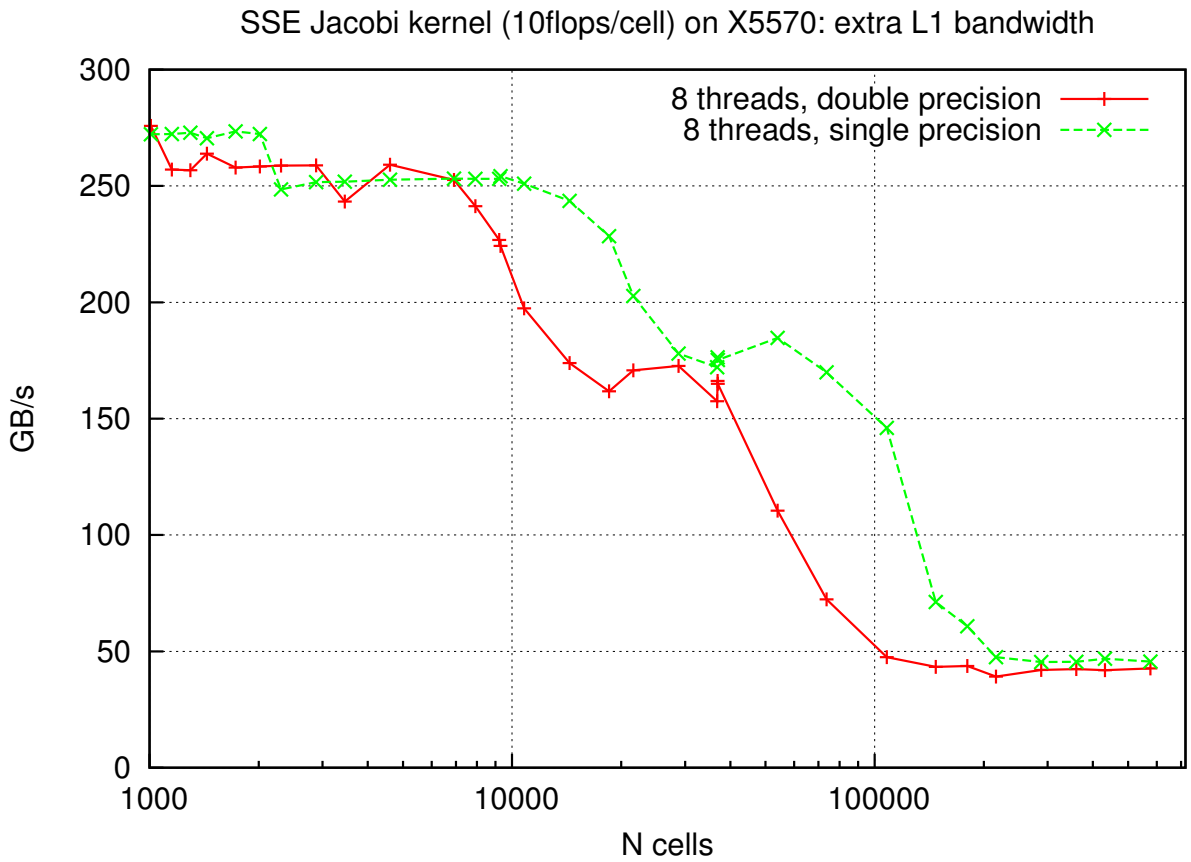
- Canevas identique pour tous les tableaux (même taille ij, même nombre de ghost-cells) $\Rightarrow$ pointeurs simplifiés, permet la "loop fusion" suivante,
- Fusion des boucles i,j $\Rightarrow$ déroulage, "branch prediction", etc.

Performance brute du "kernel" Jacobi spécialisé



- 7 additions, 3 multiplications ⇒ performance crête = 16.75 GFlops (SP)
- 14.1 GFlops = 85 % de la performance crête.

Performance brute du "kernel" Jacobi spécialisé



- Saturation de la bande passante du cache L2 et L3
- Sandy-bridge : performance crête doublée... cache L3 à 48 GB/s/core/direction

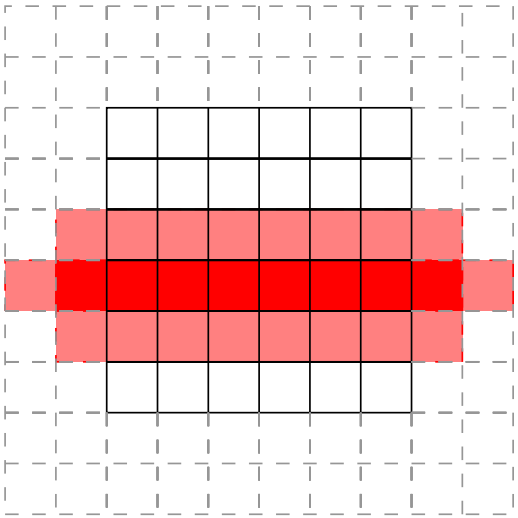
Utilisation du cache : principe

- Pour des opérateurs à stencil 3D, si plusieurs plans de maillage tiennent dans le cache, chaque valeur n'est chargée qu'une fois depuis la mémoire.
- Découper le maillage de sorte que les plans IJ tiennent dans le cache (64x64xN ou 128x128xN)
- Pour un opérateur de diffusion QDM, plus de 10 plans de maillage utilisés (flux aux faces pour 3 directions, coefficient de diffusion, 3 composantes de vitesse, etc)
- Pour les GPU, shared memory très petite
  $\Rightarrow$ patchs de petite taille, volume des bords importants
  $\Rightarrow$ bande passante mémoire moins bien exploitée
  (4Ko = 10 plans 10x10, accès non alignés pour les cellules fantômes)

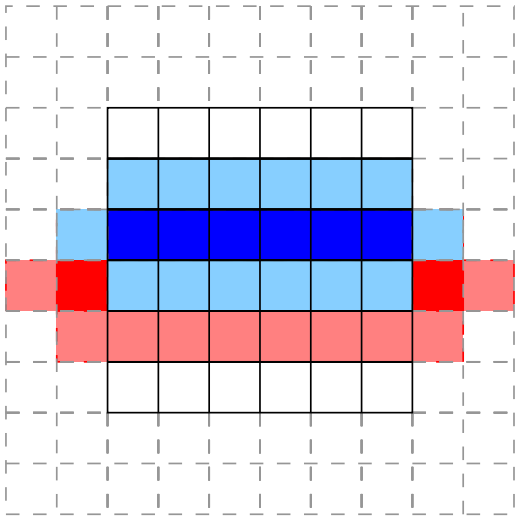
Utilisation du cache : mieux

Pour de nombreux opérateurs, la bande passante mémoire reste le facteur limitant.

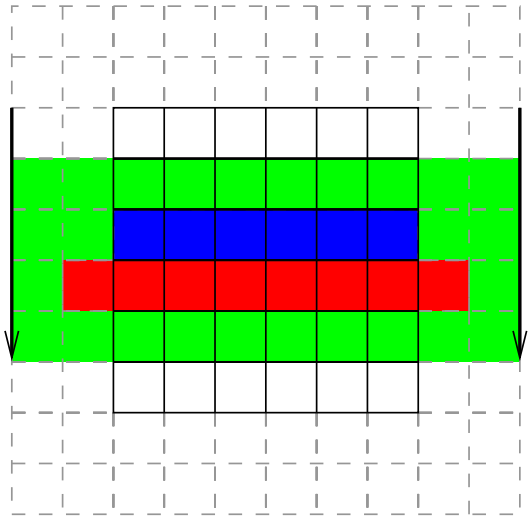
- Regroupement d'opérateurs (exemple diffusion + convection ensemble)
- Regrouper les itérations temporelles successives ($\simeq$ loop fusion)
  $\Rightarrow$ élargit le stencil $\Rightarrow$ calculs supplémentaires.



First pass (use 2 layers of ghost cells)



Second pass done immediately



Data kept in the cache (green)

- On regroupe autant qu'il faut pour ne pas saturer la bande passante mémoire.

Performance de l'algorithme complet

Tests sur Titane, compilateur icpc (peut faire mieux).

- ghost : nombre de passes simultanées
- Gflops crête : performance du noyau de calcul, données dans le cache L1
- Gflops réels : nombre d'opérations réellement effectuées par seconde
- Gflops utiles : nb. op qu'il aurait fallu avec une seule passe, divisé par le temps.
- BW : total des transactions avec la RAM, divisé par le temps.

Précision	ghost	Gflops crete	Gflops réels	Gflops utiles	BW (GB/s)
DP	4	4.8	2.45	2.15	12.2
DP	8	4.8	3.10	2.13	8.19
SP	4	9.5	5.44	4.57	13.9
SP	8	9.5	6.53	4.95	8.24

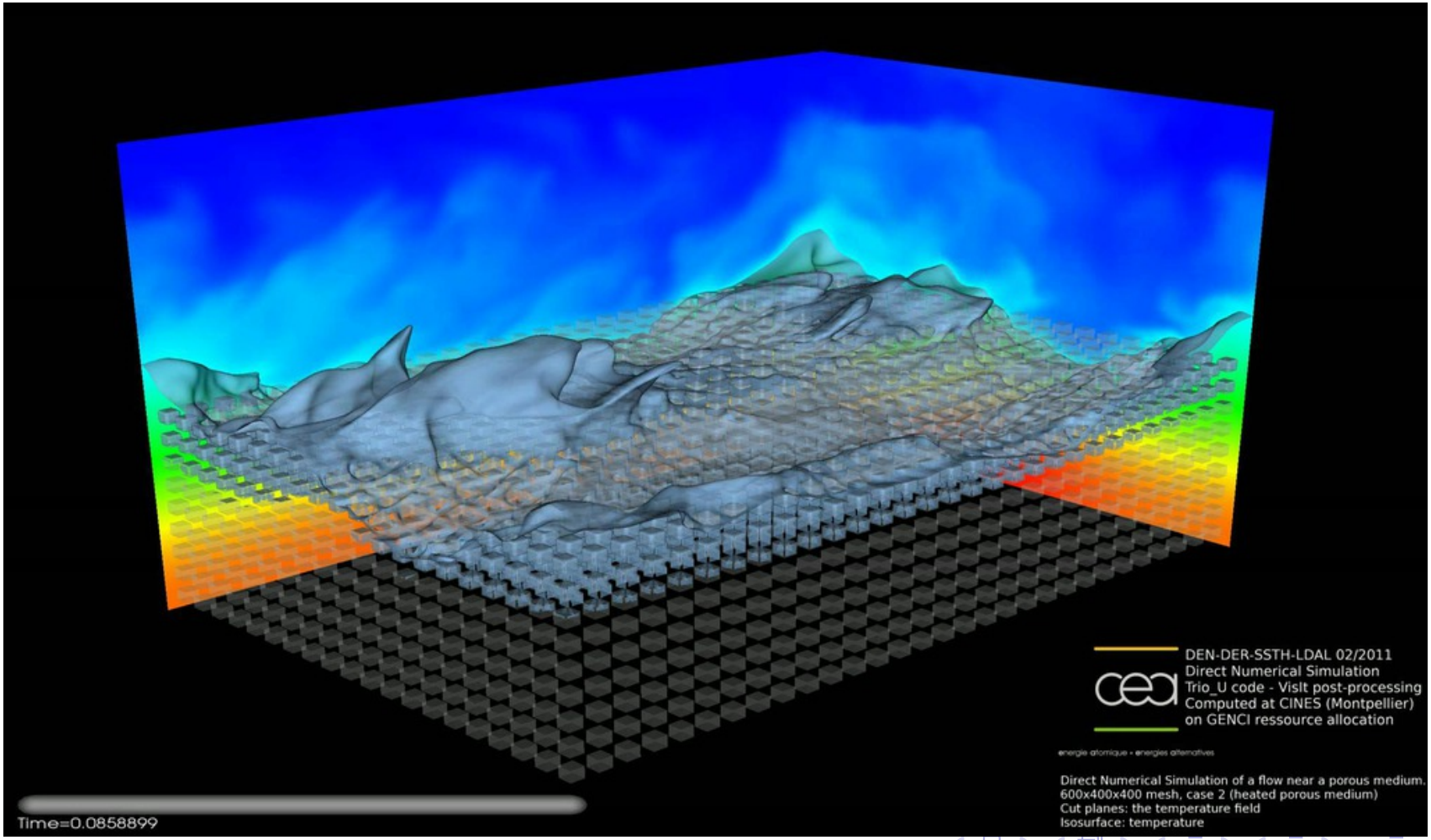
Station Harpertown 8 coeurs@2.8Ghz

Précision	ghost	Gflops crete	Gflops réels	Gflops utiles	BW (GB/s)
DP	8	4.3	1.90	1.30	5.0
SP	8	8.8	4.20	3.18	5.3

Conclusion et perspectives

Sur le multigrille géométrique

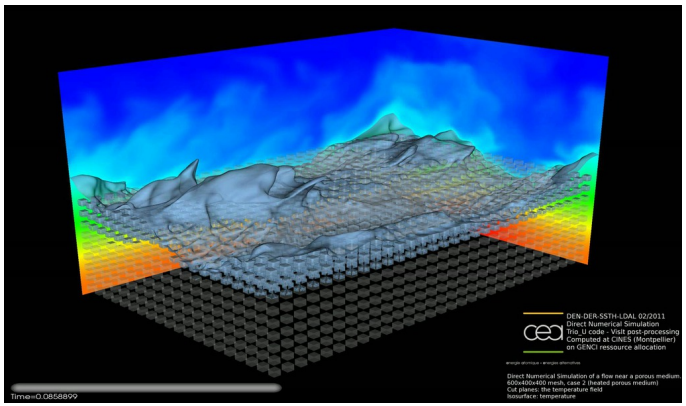
Expérience acquise sur le calcul "DNS poreux" (100M mailles) :



Sur le multigrille géométrique

Expérience acquise sur le calcul "DNS poreux" (100M mailles) :

- Pour avoir un résidu à 10^{-9} il faut :
  - 3 restarts de GMRES avec 3 vecteurs (9 V-cycles)
  - pre-smoothing: 4 it., post-smoothing: 14 it. (relaxed Jacobi)
  - 0.75s / solveur sur 640 coeurs avec ghost=2.
- Avec 2 GFlops, on peut faire, avec 1M mailles/coeur (100 coeurs) :
  - $20 \cdot 9$ itérations en 0.9s,
  - solution du système linéaire en 1.5s ?



Sur le multigrille géométrique et les opérateurs

Perspectives :

- Mise en œvre des instructions AVX (1h de travail ?)
- Optimisation des autres opérateurs de Trio_U
- Challenge : réalisation d'une DNS à $2 \cdot 10^9$ mailles d'un canal turbulent anisotherme (ρ variable), sur Curie (Sandy-Bridge) ?
  10 000 cœurs = 200 000 mailles par cœur.
- Extension pour les problèmes d'entraînement de gaz
  actuellement : 512 cœurs, envisagé : 10-50 cœurs.

Sur la vectorisation SIMD

- La vectorisation impose quasiment à coup sûr un reformatage des structures de données.
- Organisation des algorithmes en trois phases :
  - copie / reformatage des données dans un tampon local (cache/shared mem des GPU)
  - algorithmes (vectorisation)
  - copie / reformatage des résultats vers la mémoire
- Inconvénient : sérialise opérations mémoire et calculs.
- Simple précision, parfois suffisant ?

Support SIMD sur différentes architectures (taille des vecteurs) :

	float	double
Nehalem	4	2
Sandy Bridge	8	4
BlueGene	NA	2
Power 7	4	2
Sparc64 viii	?	2

Autre effet bénéfique :

- forçage d'un déroulage des boucles
- algorithmes de copie: +50% en utilisant des vecteurs de vecteurs simd (Simd_float4 = vecteur de 16 valeurs)

Sur la génération de code

- Génération de code : "mieux" que les templates pour les grosses portions de code (débuggage et compréhension du code), discussion...
- Exemple: déclaration et implémentation des classes SIMD
- Permet de générer des versions génériques et des versions optimisées avec des constantes codées en dur (pour les tailles de blocs optimales en fonction de l'architecture).

```
#Pmacro SIMD_CLASS_DEF(__type__,__psd__,__mtype__,__vsize__)
class Simd___type__
{
public:
  typedef __type__ value_type;
}
#Pendmacro(SIMD_CLASS_DEF)

// Implementation for single precision type
#Pusemacro(SIMD_CLASS_DEF)(float,ps,__m128,4)
// Implementation for double precision type
#Pusemacro(SIMD_CLASS_DEF)(double,pd,__m128d,2)
```

Autres idées en vrac

- hyperthreading :
  - pendant les phases de "copie" (memory bound) les unités flottantes sont sous-exploitées
  - l'hyperthreading peut agir comme les warps entrelacés sur GPU : cacher la latence des phases de copie.
- recouvrement communications-calculs :
  - le cache blocking oblige à découper les maillages en blocs
  - profitons-en pour mettre à part les blocs qui sont au bords
  - on économise le gros effort de mise en œuvre de l'overlapping (transformation des algorithmes pour séparer le calcul de l'intérieur du calcul du bord, transformation généralement intrusive dans tous les algorithmes de calcul).
  - boucles génériques sur les blocs (calcul intérieur + communication des bords, puis calcul des bords).