Sommaire intermédiaire

Problématique

Les données géométriques ou d'images sont de grande taille, ont besoin d'une topologie (évidente pour les images, généralement 4 à 8 voisins, fonction *distance* à définir) si on effectue des traitements prenant en compte le voisinage.

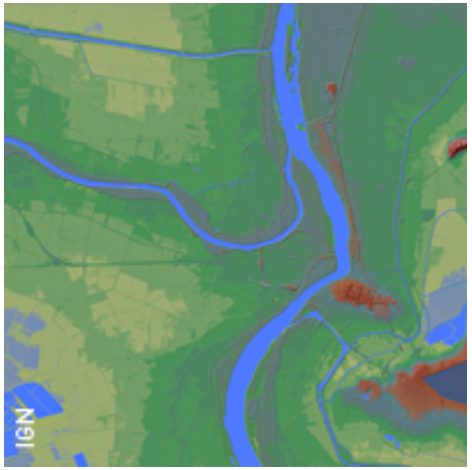
Exemple de données, sans compression binaire (c'est un autre problème)

Si on considère un point en coordonnées homogènes x, y, z, w stockées sur des réels (`float`, 4 octets). On utilise un tableau indexé des indices de points pour gérer des faces triangulaires, chaque numéro de face « pointe » sur 3 entiers (`unsigned int`, ou pointeur sur 4 octets/8 octets [a]). On a donc finalement 16 octets par point et 12 octets par face (24 sur 64 bits).

a. https://en.wikipedia.org/wiki/64-bit_computing#64-bit_data_models

C'est quoi un « gros » maillage ou une grande image ?

Une acquisition lidar (laser aéroporté), une saisie d'une heure génère env. 300 millions de points (x, y, z) [9], on peut avoir une centaine de points par m^2,



9. soit 300M*4 octets * 3 coordonnées, env. 3,5To

C'est quoi un « gros » maillage ?

La numérisation haute résolution d'une œuvre [10], comme le *David* de Michel-Ange de 5,17 m en 1999.

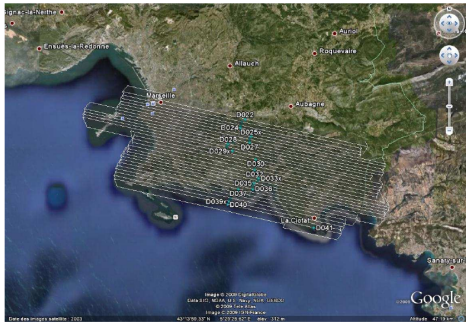
Résultats : 1 fichier PLY binaire de 1,2 Go, avec une précision de 1 mm, 56 millions de faces, 28 millions de points, 30 jours et 480 scans à fusionner.

La numérisation de 2009 est à la précision 0,29 mm utilise 480 millions de polygones.

10. http://graphics.stanford.edu/projects/mich/

C'est quoi un « gros » maillage ou une grande image ?

Cela concerne également les images bitmap. Un ensemble d'images visibles, ortho-photos IGN depuis 2002, numérisées à 15 cm, revient à ce que chaque km^2 de terrain utilise 44,5 millions de pixels (R, G, B). On peut descendre à 2 cm de résolution avec les scans 2009 et +...



Comment cela se traite ?

Les opérations courantes sur les nuages de points sont :

- la recherche de plans moyens, la création de faces,
- le calcul de normales,
- la décimation,
- ...

Évidemment lorsqu'on traite une série d'images, de géométries, ou qu'on effectue un lot de traitement (*batch*) ou qu'on considère une géométrie comme {*positions, faces, normales, masses, couleurs, niveau de résolution*} ou une image comme {*visible(R, G, B) + bande infrarouge + radar*}, ou encore qu'on interprète des images pour reconstruire une géométrie, la situation empire.

Comment cela se visualise ?

Mal... Quelques problèmes rencontrés :

- les contraintes `CPU` : architecture 32 ou 64 bits [11], accès aux données (disques, bus, RAM), concurrence des processus, interruptions,
- les contraintes `CPU` ↔ `GPU` : transfert `CPU` /`GPU` , bande passante, synchronisation avec le *framerate* vidéo,
- les contraintes `GPU` : peu de `RAM` , utilisation parallèle massive (peu de communication inter-process - IPC)
- au final, le nombre de données est plus important que le nombre de pixels, est-utile de faire transiter toutes ces informations ?
- ...

11. ne pas faire d'allocation sur la pile... i.e. remplacer les `float t[10000]` par `float * t = new float [100]` ou mieux, les structures `array deque vector`...

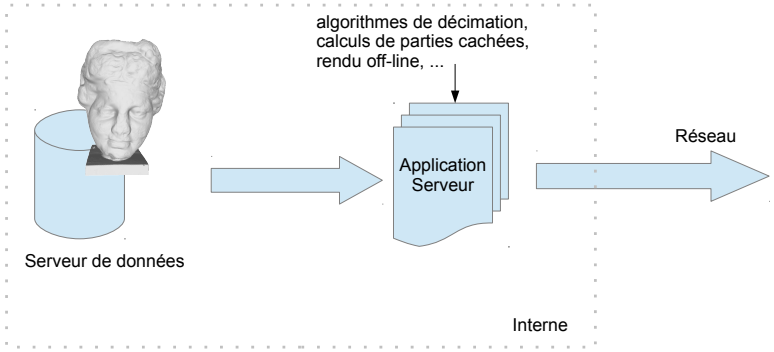
Qu'est-ce qu'on fait maintenant ?

On se tourne de plus en plus vers :

- des machines massivement parallèles de grandes capacités,
- des traitements *out-of-core* (hors mémoire).

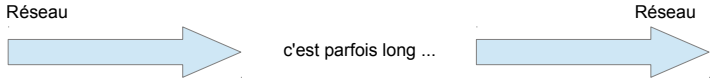
La tendance de traitement/visualisation

Une application répartie, dont les données géométriques « pleine résolution » sont sur un serveur, et dont le client est distant.



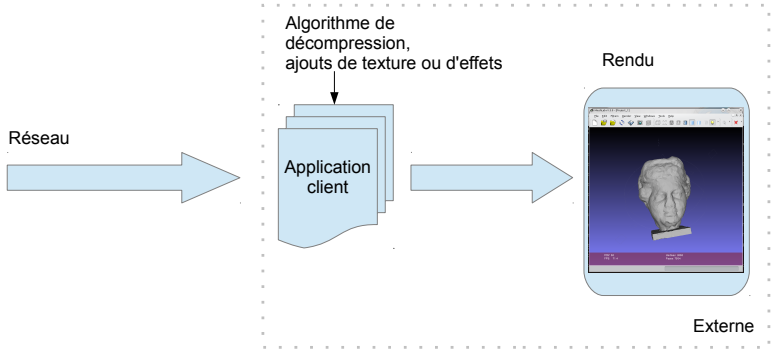
La tendance de traitement/visualisation

Une application répartie, dont les données géométriques « pleine résolution » sont sur un serveur, et dont le client est distant.



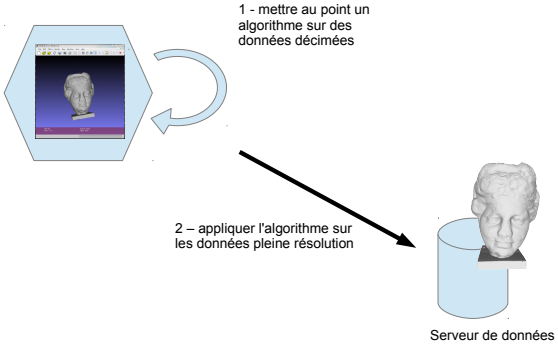
La tendance de traitement/visualisation

Une application répartie, dont les données géométriques « pleine résolution » sont sur un serveur, et dont le client est distant.



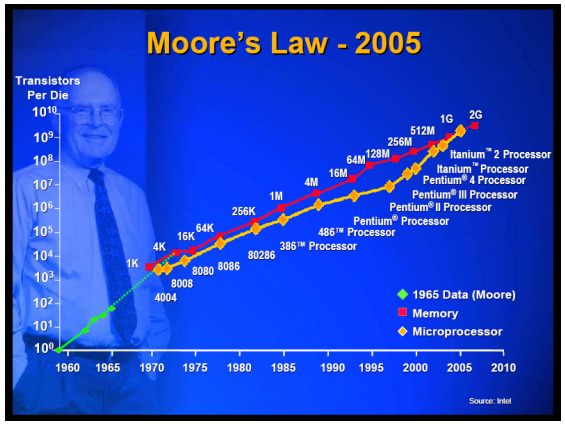
Ou ce que l'on pourrait faire aussi

Construire et tester des algorithmes (ou un travail en batch) sur un modèle décimé, puis les porter sur le modèle en pleine résolution, quitte à y passer quelques heures...



Quelles solutions ?

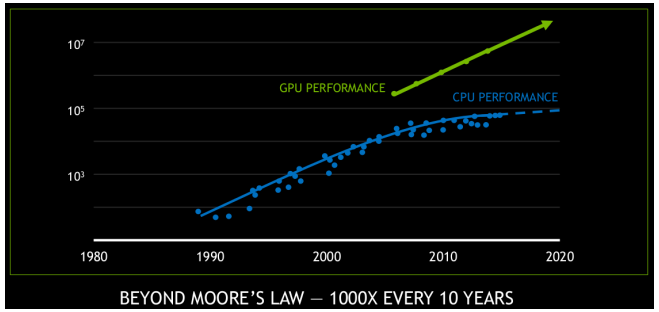
- attendre... que les machines soient assez puissantes, si on croit dur comme fer à la loi de Moore [12].



12. doublement du nombre de transistors tous les 2 ans

Quelles solutions ?

- attendre... que les machines soient assez puissantes, si on croit dur comme fer à la loi de Moore [12]. Les fabricants de cartes graphiques disent qu'on est à la limite de cette courbe.



Nvidia GTC 2018

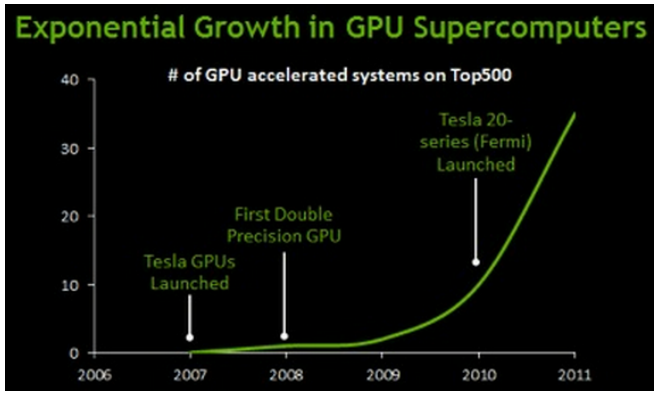
12. doublement du nombre de transistors tous les 2 ans

Quelles solutions ?

- utiliser l'informatique parallèle pour monter des grappes de machines ou de processeurs qui sauront accélérer le traitement de ces algorithmes. Du coup il faut ré-écrire les programmes, c'est évidemment dépendant de l'architecture (scalaires/vectoriels, mémoire partagée, latences),

Quelles solutions ?

- croire en une loi des fabricants de cartes graphiques, qui explosent la loi de Moore ! Mais tout ré-écrire, avec des contraintes d'architectures pires encore.



Positions des fabricants de GPU

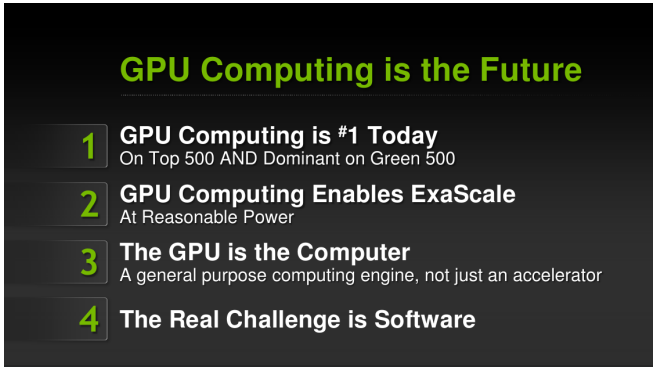


Figure – Conclusion Nvidia de SC'10 [12]

L'avenir c'est le GPU ! [13]

12. *GPU Computing to Exascalee and Beyond* - Bill Dally, Nvidia
13. `http://www.top500.org/`

Sommaire intermédiaire

Sommaire intermédiaire

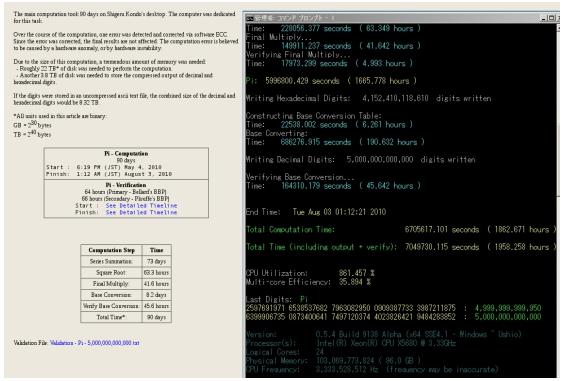
En bref

On se dirige vers des algorithmes multi-processus (il était temps !). Sur des architectures hybrides mixant processeurs scalaires bons à tout faire, et des processeurs vectoriels très bons dans le massivement parallèle.

Il ne faut plus penser l'application comme une somme de traitements séquentiels, effectués en Mémoire vive -> le règne de l'*Out-of-Core* arrive.

Définition de l'Out-of-Core

Les algorithmes « Out-of-core »servent à traiter des données de grandes tailles, qui ne peuvent tenir entièrement dans la mémoire d'un ordinateur. Ils cherchent un compromis entre les capacités de traitement en CPU, et les interactions Mémoire vive / Mémoire secondaire [14].



15

14. Vitter, JS (2001). "External Memory Algorithms and Data Structures : Dealing with MASSIVE DATA.". ACM Computing Surveys 33 (2) : 209–271.
15. http://www.numberworld.org/misc_runs/pi-5t/details.html

Quelques références

- « Out-of-Core Compression for Gigantic Polygon Meshes », M. Isenburg et S. Gumhold (`http://www.cs.unc.edu/~isenburg/oocc/`),
- « Out-of-Core Simplification of Large Polygonal Models », P. Lindstrom,
- « Out-Of-Core Algorithms for Scientific Visualization and Computer Graphics », Course notes for IEEE Visualization 2002, C T. Silva, Y.J. Chiang, J. El-Sana, P. Lindstrom,
- ...

541 000 publications jusqu'à aujourd'hui (« `out-of-core` », site ACM), 21 en 2000, 74 en 2011.

Quelques références

Les publications pointent les problèmes suivant :

- comment ordonner la liste des faces et des sommets pour que la lecture des données (la plupart du temps sur le disque) ne soient pas une lecture aléatoire sur le disque dur ?
- comment avoir une organisation spatiale efficace (pour gérer les voisins, pour calculer ou stocker facilement les distances...) ?
- est-ce que les structures pour le traitement sont efficaces pour la visualisation ?
- comment convertir ces structures de données en tables indicées pour la sauvegarde ?

Mais si !

Évidemment tous les algorithmes de la synthèse d'images ne sont pas parallélisables...

- si on pense assez vite à du traitement d'images, sur une grille discrète uniforme, par un masque de convolution tout se passera bien (sauf pour les bords),

Mais si !

Évidemment tous les algorithmes de la synthèse d'images ne sont pas parallélisables...

- si on pense assez vite à du traitement d'images, sur une grille discrète uniforme, par un masque de convolution tout se passera bien (sauf pour les bords),
- de la même façon effectuer des produits scalaires entre des milliers de vecteurs pour calculer un `backface culling`,

Mais si !

Évidemment tous les algorithmes de la synthèse d'images ne sont pas parallélisables...

- si on pense assez vite à du traitement d'images, sur une grille discrète uniforme, par un masque de convolution tout se passera bien (sauf pour les bords),
- de la même façon effectuer des produits scalaires entre des milliers de vecteurs pour calculer un `backface culling`,
- pour des calculs de parcours de plus grandes longueurs sur des arêtes de maillages discrets,

Mais si !

Évidemment tous les algorithmes de la synthèse d'images ne sont pas parallélisables...

- si on pense assez vite à du traitement d'images, sur une grille discrète uniforme, par un masque de convolution tout se passera bien (sauf pour les bords),
- de la même façon effectuer des produits scalaires entre des milliers de vecteurs pour calculer un `backface culling`,
- pour des calculs de parcours de plus grandes longueurs sur des arêtes de maillages discrets,
- ou encore des calculs de points sur une surface paramétrique,

Mais non !

Par contre :

- un algorithme comme le *Progressive Mesh* de H. Hoppe qui parcourt chaque sommet et détermine s'il doit être fusionné avec un de ces voisins,

Mais non !

Par contre :

- un algorithme comme le *Progressive Mesh* de H. Hoppe qui parcourt chaque sommet et détermine s'il doit être fusionné avec un de ces voisins,
- un système itératif ou une déformation de mailles hexaèdriques,

Mais non !

Par contre :

- un algorithme comme le *Progressive Mesh* de H. Hoppe qui parcourt chaque sommet et détermine s'il doit être fusionné avec un de ces voisins,
- un système itératif ou une déformation de mailles hexaèdriques,
- ou même un bête algorithme dichotomique

Sommaire intermédiaire

Avant de développer des applications de traitements de données d'images ou géométriques concurrentes ou distribuées, il faut comprendre l'organisation de données et de jobs possibles. Cette partie est là pour cela.

Dans quels environnements ?

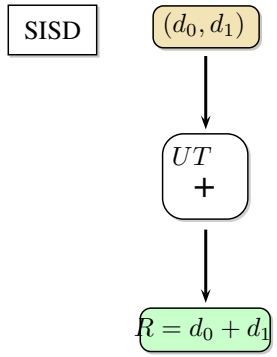
La programmation parallèle, ou massivement parallèle, est une discipline connue de l'informatique. On distingue 4 architectures principales des traitements, en utilisant la notation de Flynn [16] :

- SISD, *Single Instruction - Single Data*, pour chaque unité de traitement (un processeur par exemple), une donnée différente et une instruction (un traitement) particulière. Architecture et programmation classiques.
- SIMD, *Single Instruction - Multiple Data*, le parallélisme s'effectue sur les données, qui sont injectées dans plusieurs unités de traitement qui effectuent la même instruction (comme un processeur vectoriel).
- MISD, *Multiple Instructions - Single Data*, la même donnée est utilisée plusieurs fois par des unités de traitement différentes (peu utilisé).
- MIMD, *Multiple Instructions - Multiple Data*. Chaque unité traite une donnée spécifique, chacune possède donc sa propre mémoire (le plus courant).

16. Michael J. Flynn, 1966

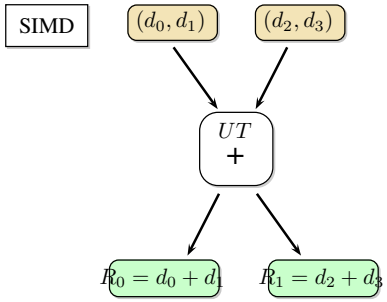
Pour fixer les esprits...

Single Instruction - Single Data



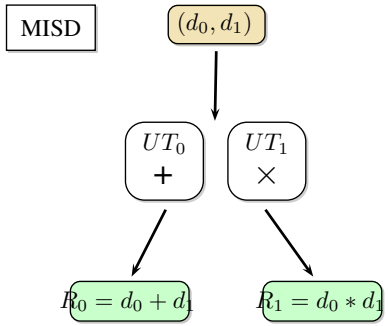
Pour fixer les esprits...

Single Instruction - Multiple Data



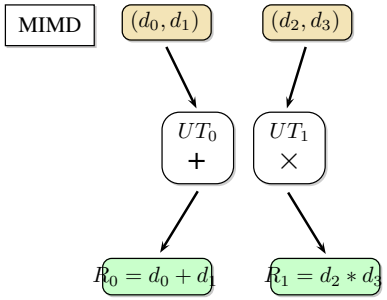
Pour fixer les esprits...

Multiple Instructions - Single Data



Pour fixer les esprits...

Multiple Instructions - Multiple Data



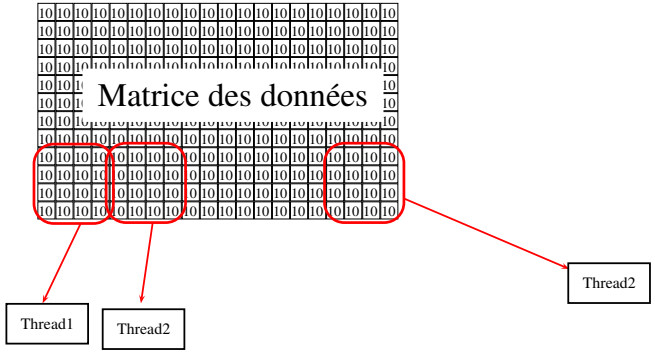
Les modèles MIMD ont 2 variantes :

- MIMD à mémoire partagée, Les processeurs ont accès à la mémoire comme un espace d'adressage global. Tout changement dans une case mémoire est vu par les autres CPU. La communication inter-CPU est effectuée via la mémoire globale.
- MIMD à mémoire distribuée, Chaque CPU a sa propre mémoire et son propre système d'exploitation. Ce second cas de figure nécessite un middleware pour la synchronisation et la communication.

On note aussi le *SIMT* (*Single Instruction - Multiple Threads*), notation utilisée par Nvidia pour définir leurs architectures Fermi. Les unités de traitement sont similaires, les instructions sont réparties sur un groupe de `threads`, cela permet donc de maximiser l'efficacité des unités, SI le travail à effectuer est massivement parallèle.

Présentation des `Threads`

Le b.a.-ba de la programmation concurrente commence par la mise en place de threads[17]. Le principe du traitement simple à base de threads (i.e. si on ne veut pas tomber dans la synchronisation, l'exclusion mutuelle et les *dead-lock*) repose avant tout sur l'indépendance des données.



17. http://fr.wikipedia.org/wiki/Thread_(informatique)

Les threads

Les threads sont rapides à mettre en place (par rapport à un clonage de processus), ils sont liés au processus appelés et partagent sa mémoire (et donc limités globalement à 4Go sur une architecture 32 bits). Ils font partie de l'espace utilisateur (gestion plus simple, moins de risques pour le système). Par exemple, dans un `main()`, la différence se fait par :

```
int main() {
...
//appel de la fonction1
int r1 = fonction1(param1);
//appel de la fonction2
int r2 = fonction2(param2);
...
}
```

```
int main() {
...
pthread_t th1, th2;
//appel de la fonction1
pthread_create(&th1, NULL, fonction1, (void*)param1);
//appel de la fonction2
pthread_create(&th2, NULL, fonction2, (void*)param2);

//retour des threads
pthread_join(th1, &r1);
pthread_join(th2, &r2);
...
}
```

Présentation d'OpenMP

Deuxième méthode de programmation concurrente, l'utilisation d'OpenMP[18]. Son utilisation est effective dans g++ depuis la version 4. Cette API définit un ensemble d'instruction de haut-niveau (pragma) à utiliser dans les programmes en C++. Par exemple :

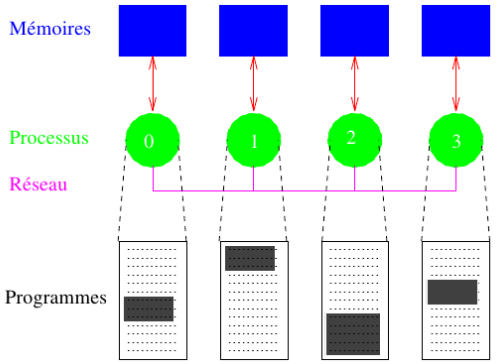
```
#pragma omp for
   for (unsigned int i = 0; i < MAXSIZE; ++i)
    {
     tab[i] = tab[i] / float(MAXSIZE);
    }
```

Le compilateur se charge ensuite de créer et détruire les threads (basés sur Posix), établir les communications, les synchronisations...

18. http://www.openmp.org

Présentation de MPI

Troisième et dernière méthode de programmation concurrente, la distribution du calcul sur un ensemble de machines. MPI est une API multi-machines qui permet de spécifier l'envoi de messages de haut-niveau. Elle date de 1994 pour la version 1.0 et est régulièrement améliorée. Elle devrait notamment, en version 3.0, supporter le GPU.
On trouve beaucoup de documentation, en français, sur le site de l'IDRIS[19], Institut du développement et des ressources en informatique Scientifique du CNRS.



19. http://www.idris.fr/

Sommaire intermédiaire

Old School GPGPU

OpenGL, depuis sa naissance, permet de spécifier des matrices de transformations (au lieu des glTranslate, glRotate, gluPerspective par exemple).
Voilà un exemple de multiplication de matrices :

```
//définition des valeurs (à plat)
float data1 [] = { 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
   0.0, 0.0, 1.0, 0.0, -1.0,-1.0, 0.0, 1.0};

//remise à 0 des transmissions par chargement de la matrice Identité
glLoadIdentity();

//multiplication par la matrice de données
   glMultMatrixf(data1);
   glMultMatrixf(data1);

//lecture des valeurs dans la matrice de modelisation
   glGetFloatv(GL_MODELVIEW_MATRIX, result);
```

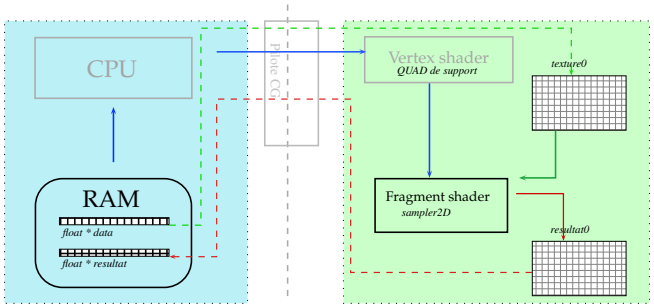
Beaucoup d'inconvénients

Évidemment cette méthode n'effectue que des calculs matriciels, et il faut ruser pour faire des opérations du type $AX + B$. Il faut aussi donner un contexte graphique au programme car `glMultMatrix()` a besoin d'une environnement `glMatrixMode(GL_MODELVIEW)` pour travailler.
Elle ne travaille que sur des matrices 4×4 au maximum, ce qui oblige à découper ses données pour l'appliquer, en séquences (pas d'exécution parallèle sans contexte OpengGL en parallèle).

Enfin, le prototype de `glMultMatrix` peut être :
`void glMultMatrixd( const GLdouble *m )` ou
`void glMultMatrixf( const GLfloat *m )`.

Principes

Depuis la version 1.4, on peut utiliser les outils d'OpenGL et la programmation de shaders. Les données sont transmises via des textures. Elles sont appliquées sur un quad et sont traitées dans le `Fragment Shader`. On doit ensuite rapatrier la texture résultat en RAM.



Le code...

```
//create FBO and bind it (that is, use offscreen render target)
GLuint fb;
glGenFramebuffersEXT(1,&fb);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);

//create texture
GLuint tex;
glGenTextures (1, &tex);
glBindTexture(GL_TEXTURE_RECTANGLE, tex);

//set texture parameters
glTexParameteri(GL_TEXTURE_RECTANGLE, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_RECTANGLE, GL_TEXTURE_WRAP_T, GL_CLAMP);
```


Le code (suite)

```
//define texture with floating point format
glTexImage2D(GL_TEXTURE_RECTANGLE, 0, GL_RGBA32F, texSize, texSize,
     0, GL_RGBA, GL_FLOAT, 0);

//attach texture
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
     GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_RECTANGLE, tex, 0);

//transfer data to texture
glTexSubImage2D(GL_TEXTURE_RECTANGLE, 0, 0, 0, texSize, texSize,
     GL_RGBA, GL_FLOAT, data);

//and read back
glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
glReadPixels(0, 0, texSize, texSize, GL_RGBA, GL_FLOAT, result);
```

Ne pas oublier les projections 1 :1

```
//viewport transform for 1:1 pixel=texel=data mapping
glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(0.0,texSize,0.0,texSize);

glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glViewport(0,0,texSize,texSize);
```

Note : on peut répéter les étapes de calcul en fournissant la texture "résultat" en entrée d'un autre calcul.

les PBO : Pixel Buffer Objects

L'idée est d'ouvrir encore un peu la relation CPU/GPU en accélérant la lecture de données de type Pixels. L'envoi vers le GPU se fait grâce un mapping de mémoire (rappel de Posix [20] ?) :

```
glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, ioBuf[array]);
glBufferData(GL_PIXEL_UNPACK_BUFFER_ARB,
     texSize*texSize*sizeof(float), NULL, GL_STREAM_DRAW);

void* ioMem = glMapBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, GL_WRITE_ONLY);

assert(ioMem);
copy(chunks[chunk][array], ioMem);

glUnmapBuffer(GL_PIXEL_UNPACK_BUFFER_ARB);

glTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, 0, 0, texSize,
     texSize, GL_LUMINANCE, GL_FLOAT, BUFFER_OFFSET(0));

glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, 0);
```

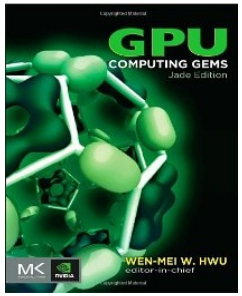
20. mmap

La lecture de fait via `glReadPixels` mais dans un buffer dont on attend la complétion avant de le récupérer en RAM :

```
glReadBuffer(attachmentpoints[readTex]);
glBindBuffer(GL_PIXEL_PACK_BUFFER_ARB, ioBuf[9]);
glBufferData(GL_PIXEL_PACK_BUFFER_ARB,
     texSize*texSize*sizeof(float), NULL, GL_STREAM_READ);

glReadPixels (0, 0, texSize, texSize, GL_LUMINANCE,
     GL_FLOAT, BUFFER_OFFSET(0));

void* mem = glMapBuffer(GL_PIXEL_PACK_BUFFER_ARB, GL_READ_ONLY);
assert(mem);
copy(mem,data);

//clear
glUnmapBuffer(GL_PIXEL_PACK_BUFFER_ARB);
glBindBuffer(GL_PIXEL_PACK_BUFFER_ARB, 0);
```

Remarque, via les PBO les vitesses de transfert sont environ multipliées par 2 par rapport à la méthode précédente.

Quelques lectures :



Exemples accessibles sur le site *développeur* Nvidia, `https://developer.nvidia.com/category/zone/game-graphics-development`.

Et le site de référence : `www.gpgpu.org`.