Introduction aux simulations numériques sur GPU

Guillaume Provencher

4 octobre 2010

Introduction

Comparaison GPU vs CPU

(a) i7 960 (laboDMS) : 4 coeurs (8 threads HT), 3.2 GHz, 12 Go DDR3

(b) GTX 480 (laboMAT et laboSTAT) : 480 coeurs, 1.4 GHz, 1.5 Go GDDR5

Avantages GPU

- Rapide pour exécuter des instructions en parallèle (gain 100x max).
- Excellent pour les **gros** travaux.
- Lit les instructions de manière *asynchrone* ; retourne le contrôle au CPU après réception du code.
- Offre une grande puissance de calcul à faible coût.

Désavantages GPU

- Mauvais pour petites tâches (délai de *warm-up*).
- Mauvais pour exécuter du code en série (cadence des coeurs faible).
- Plus difficile à programmer que le CPU.
- Transferts mémoire GPU $\rightarrow$ mémoire CPU lents.

Pour programmer le GPU : CUDA

Compute **U**nified **D**evice **A**rchitecture

- Extension du langage C.
- Permet d'exécuter des intructions sur le GPU.
- Permet un contrôle simple de l'exécution en parallèle.
- Libraires incluses avec CUDA :
  - CUBLAS : **CU**DA **B**asic **L**inear **A**lgebra **S**ystem
  - CUFFT : **CU**DA **F**ast **F**ourier **T**ransform

Programmation CUDA

Déclaration de variables sur GPU

- Nombres scalaires sur CPU.
- Pour les vecteurs et matrices :
  `cudaMalloc(void** tab, size_t dim);`
- Pour copier un tableau CPU→GPU (ou vice-versa) :
  `cudaMemcpy(dest, src, dim, Méthode);` où `Méthode` prend la valeur `cudaMemcpyHostToDevice` ou `cudaMemcpyDeviceToHost`.
- L'espace allouée est libérée avec `cudaFree`.

Déclaration de fonctions

CUDA introduit différents *types* de fonctions :

- `__host__` : fonction exécutée sur le CPU (par défaut).
- `__global__` : fonction exécutée sur le GPU mais appelée par le CPU (point d'accès).
- `__device__` : fonction exécutée sur le GPU et appelée depuis une fonction GPU.

Le code CUDA doit être écrit en *C* mais le code CPU peut être écrit en *C/C++*.

Grille, blocs et *threads*

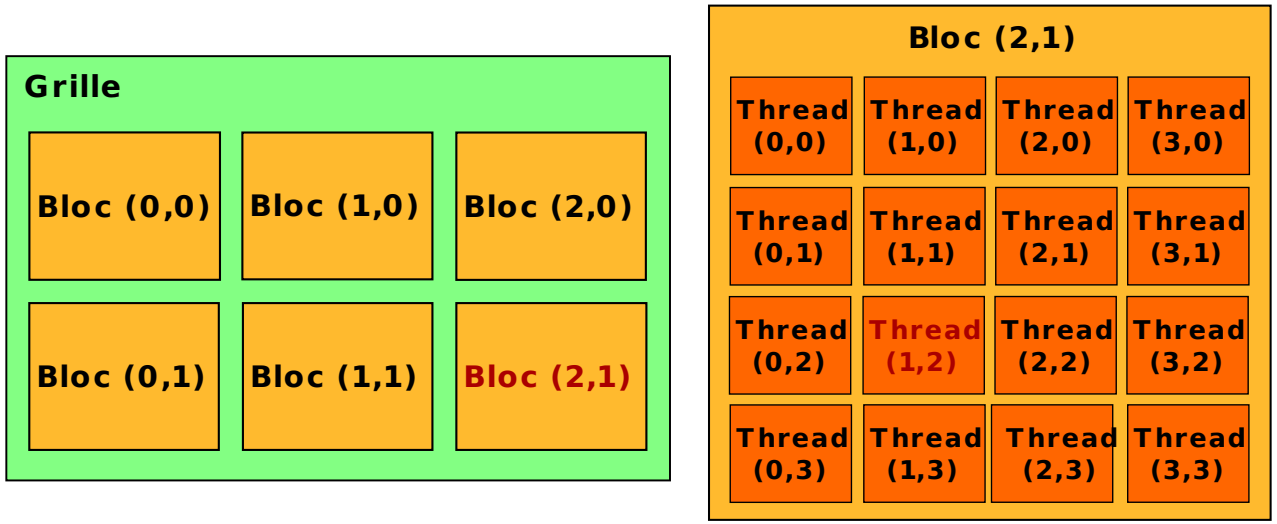

FIG.: gridDim=(3,2,0), blockDim=(4,4,0), threadIdx=(1,2,0), blockIdx=(2,1,0).

- `dim3 gridDim` : format de la grille.
- `dim3 blockDim` : format des blocs.
- `uint3 blockIdx` : position du bloc contenant le *thread* exécuté dans la grille.
- `uint3 threadIdx` : position du *thread* exécuté dans le block.

Appel de fonctions

Un appel à une fonction `f` de type `__global__` doit être effectué comme suit :

```
f<<<dimGri, dimBloc>>>(arg1, arg2, ...)
```

où `dimGri` et `dimBloc` sont, respectivement, deux structures de type `dim3` (x, y et z) contenant le format de la grille et le nombre de *threads* par bloc.

Types de mémoires à utiliser

Dans le code fonctionnant sur GPU (fonctions `__device__` et `__global__`), la mémoire employée pour stocker les variables peut être spécifiée.

- `__shared__` : variable stockée dans la mémoire commune aux *threads* du bloc (rapide mais vie limitée au bloc).
- `__device__` : variable stockée dans la mémoire globale du GPU (plus lent mais existe pour toute la durée du programme).
- `__constant__` : comme `__device__` mais la variable est stockée dans la mémoire constante.

Si rien n'est spécifié, la variable sera placée dans un registre (vie : *thread* seulement, rapidité : excellente).

Exemple

```
__global__ void AddVec(int N, float *v1, float *v2)
{
 int i = blockDim.x * blockIdx.x + threadIdx.x;
 if (i < N)
  v1[i] = v1[i] + v2[i];
}
```

```
int main()
{
 float h_v1[1000] = {...};
 float h_v2[1000] = {...};
 float *d_v1;
 float *d_v2;
 cudaMalloc( (void**) &d_v1, 1000 );
 cudaMalloc( (void**) &d_v2, 1000 );
 cudaMemcpy(d_v1, h_v1, 1000, cudaMemcpyHostToDevice);
 cudaMemcpy(d_v2, h_v2, 1000, cudaMemcpyHostToDevice);

 dim3 dimBloc(256, 1, 1);
 dim3 dimGri( (1000 + dimBloc.x - 1) / dimBloc.x );

 AddVec<<<dimGri, dimBloc>>>(1000, d_v1, d_v2);
}
```

Compilation

- La compilation des programmes comportant du code CUDA (.cu) doit être effectuée par `nvcc`.
- Syntaxe et options similaires à `gcc`.
- Exemple :

```
nvcc -O3 (-arch sm_13) -o monprog monprog.cu
```

Programmation MATLAB + CUDA

Stratégies d'optimisation

1. Situer les engorgements : `profile -timer real`, `profile on`, `profile viewer`, `profile off` ou encore avec `tic` et `toc`.
2. Sélection de la stratégie optimale pour vos besoins :
   - Fichier MEX
   - MEX + CUDA
   - Toolbox GPU
   - GPUmat + (MEX + CUDA)

Fichier MEX : la base

- Fichiers compilés écrits en *C* («.c») pour MATLAB.
- `include "mex.h"`
- `void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])`
- `size_t mxGetM(mxArray A)` et `size_t mxGetN(mxArray A)`
- `mxArray *mxCreateNumericMatrix(mwSize m, mwSize n, mxClassID classId, mxComplexity complexFlag);`
- `double *mxGetPr(mxArray mxA)`

```
#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
                 const mxArray *prhs[]){
 unsigned int m = mxGetM(prhs[0]);
 unsigned int n = mxGetN(prhs[0]);
 plhs[0] = mxCreateNumericMatrix(m, n, mxDOUBLE_CLASS,
 mxREAL);
 double *dataOUT = mxGetPr(plhs[0]);
 double *dataIN1 = mxGetPr(prhs[0]);
 double *dataIN2 = mxGetPr(prhs[1]);

 unsigned int i = 0;
 while(i < m*n)
  dataOUT[i] = dataIN1[i] + dataIN2[i], i++;
}
```

MEX + CUDA

Pour introduire du code CUDA dans un fichier MEX, il faut :

- Une extension «.cu»au fichier.
- Inclure `mex.h` et `cuda.h`.

Les variables MATLAB sont lues comme auparavant mais ensuite il faut les envoyer sur la mémoire du GPU, avec `cudaMemcpy`.

GPUmat

GPUmat est un *toolbox* gratuit pour MATLAB permettant d'utiliser le GPU. Exemple (gain ~50x sur GTX 285) :

```
GPUstart
...
A = GPUdouble(rand(5000,5000));
B = GPUdouble(rand(5000,5000));
tic; A + B; GPUsync; toc
...
GPUstop
```

Il existe aussi le type `GPUsingle` de nombres réels à 32 bits (contre 64 pour `GPUdouble`). *GPUmat* est disponible sur les machines CUDA du département. Pour l'activer, démarrez `GPUstart.m` situé dans `/usr/local/math/GPUmat/`.

Jacket

Jacket est un autre *toolbox* GPU pour MATLAB :

- Non gratuit (~300$ version étudiante + ~150$ DLA).
- Beaucoup de fonctions MATLAB adaptées.
- Accélération de l'affichage des graphiques et animations.
- Documentation complète et beaucoup d'exemples fournis.
- Syntaxe très similaire à GPUmat : `gsingle`, `gdouble`, etc.

Ce toolbox sera bientôt disponible sur deux machines au département.

GPUmat + (MEX + CUDA)

Il est possible de créer des fonctions MEX contenant du code CUDA manipulant des variables déclarées à l'aide de GPUmat. Puisque les variables MATLAB résident déjà sur le GPU, leur lecture est différente :

MATLAB

```
A=GPUdouble(rand(1000,1000));
Resultat=MaFonction(getPtr(A));
```

Fichier MEX

```
double *Agpu = (double *) ( (uintptr_t)
                    mxGetScalar(prhs[0]) );
```

Compilation des fichiers MEX

Pour compiler les fichiers MEX sans code CUDA :

```
$MATLAB/bin/mex monprog.c
```

où `$MATLAB` est `/usr/local/math/matlab/matlab-2010a`.
Pour compiler des fichiers MEX avec CUDA (avec ou sans GPUmat), utilisez d'abord `nvcc` et ensuite `mex` :

```
nvcc -c monprog.cu -Xcompiler -fPIC (-arch sm_13) -I
$MATLAB/extern/include/

$MATLAB/bin/mex monprog.o -L
/usr/local/cuda/cuda31/lib/ -lcudart -cxx
```

Références

- ★ *nVidia CUDA Programming Guide*, nVIDIA Corporation, Santa Clara, août 2009.
- ★ *Scientific Computing on a GPU using CUDA*, Steve Conover, Atlanta, février 2009.
- ★ *A beginner's guide to programming GPUs with CUDA*, Mike Peardon, Dublin, avril 2009.
- ★ *Accelerating MATLAB with CUDA Using MEX Files*, nVIDIA Corporation, Santa Clara, septembre 2007.
- ★ *CUBLAS Library 2.3*, nVIDIA Corporation, Santa Clara, juin 2009.
- ★ `http://www.mathworks.com/support/tech-notes/1600/1605.html`