Rapport de Stage

Méthodes de volumes finis sur carte graphique nVidia pour Euler compressible

L.Besson S.Hecht M.Isnard

2 juillet 2012

Résumé

Ce mémoire présente le travail réalisé en stage de L3 à l'ENS de Cachan.

Nous allons d'abord étudier l'équation d'advection, en 1D, puis en 2D, et présenter notre contact sur les schémas numériques explicites d'ordre 1. L'objectif étant de coder un solveur pour ces équations, nous présenterons un premier contact avec le C en implémentant des résolutions numériques avec ces schémas ; puis quelques explications sur l'architecture des GPU et le langage CUDA, ayant pour but de paralléliser ces résolutions et les rendre significativement plus rapides. Un panorama de méthodes pour afficher pertinemment les données ainsi produites sera proposé.

Ceci servant de préliminaire aux objectifs suivants, nous allons étudier l'équation d'Euler en seconde partie, en travaillant sur une généralisation du premier schéma. Un travail théorique sera détaillé sur ce schéma (stabilité, ordre en temps et en espace etc...), et des aperçus d'implémentation seront aussi évoqués.

Et enfin, une part conséquente de notre travail étant occupé par la visualisation, et l'apprentissage d'outils performants, nous donnerons quelques explications sur les différentes méthodes, en particulier sur l'optimisation en rapport avec cet affichage. La présentation de l'optimisation de nos résolutions concluera ce rapport.

Table des matières

Introduction

Le présent document fait office de rapport de stage commun pour les élèves Sophie Hecht, Maxime Isnard et Lilian Besson, tous trois élèves en première année à l'École Normale Supérieure de Cachan, au département de maths. Ce rapport de stage conclut le stage du second semestre de première année. Notre stage s'est effectué au CMLA (*Centre des mathématiques et de leurs applications*, le laboratoire de recherche en mathématiques à l'ENS de Cachan), de février à juin 2012. Nos responsables de stage ont été Florian De Vuyst et Jean-Michel Ghidaglia (responsables seniors) ; et Daniel Chauveheid et Saad Ben Jelloun (responsables juniors). Notre stage s'intitule **"méthodes de volumes finis sur carte graphique nVidia pour la simulation numérique de fluides compressibles"**.

0.1 Objectifs initiaux et intérêts

Sujet Nous rappelons ici les objectifs initiaux qui nous ont été donnés dans le sujet de stage et que nos responsables nous ont présentés :

> *Dans ce stage de 5 mois, il s'agit d'abord de s'approprier la méthode VFFC, d'acquérir une culture générale en systèmes de lois de conservation et leur approximation par volumes finis. Dans un deuxième temps, il s'agira d'abord de coder en C la méthode VFFC pour les équations d'Euler compressibles 2D sur une géométrie rectangulaire, puis de porter le code sur carte graphique nVIDIA via le langage CUDA. Une réflexion sur la définition et l'organisation des structures de données et les échanges et communications entre threads sera nécessaire. Les gains de performance attendus sont de l'ordre de 100 sur maillage type de taille 512 x 512.*

Intérêt L'intérêt de la modélisation des fluides compressibles est tout d'abord industriel : les industries pétrochimiques, métallurgiques et alimentaires manipulent toutes sortes de fluides, dont des fluides réels qui peuvent être modélisés par des fluides compressibles. Ces professionnels doivent pouvoir prédire l'évolution d'un fluide compressible, ainsi ils ont besoin de "bonnes" équations physiques qui modélisent le mouvement de ces fluides, et c'est là le rôle des physiciens. Mais connaître des équations ne suffit pas ! Et c'est ici qu'interviennent les mathématiciens : ces équations physiques doivent être modélisées numériquement, et résolues par ordinateur afin de pouvoir les exploiter.

Précision ? Par ailleurs, ces simulations numériques n'ont d'intérêt que si elles présentent certaines caractéristiques : convergence, stabilité, consistance et surtout une précision suffisamment bonne pour que les résultats de la simulation puissent être exploités. Ces simulations procèdent par **discrétisation spatiale**, et une façon simple d'obtenir une meilleure précision est d'augmenter le nombre de points. Malheureusement, cela implique d'augmenter drastiquement le temps de calcul ! Et ce genre de simulation doit pouvoir être calculé *en temps réel*, ou en tout cas le plus rapidement possible.

Multi-cœurs Durant les vingt dernières années, pour améliorer les performances de sa machine de calcul, il suffisait d'acheter une machine plus récente (la vitesse des processeurs augmentait alors suivant une loi géométrique, appelée *loi de Moore*). Malheureusement, depuis quelques années la vitesse des processeurs a atteint sa limite *(de l'ordre de 3 Ghz)* et restera

plafonnée par cette valeur [1]. Les constructeurs ont alors commencé à produire des processeurs **multi-cœurs**, obligeant les développeurs de jeux-vidéo et les numériciens à repenser leur programmation : c'est ainsi que s'est popularisé la programmation parallèle.

Programmation parallèle Et à propos des jeux-vidéos, il faut noter que le multi-cœur n'est pas apparu en premier sur les processeurs (que nous désignerons par les lettres **CPU**, pour Central Processing Unit), mais sur les cartes graphiques. Le développement de graphismes de plus en plus évolués s'est déroulé conjointement à une amélioration des capacités des cartes vidéos : toujours plus de mémoire, et plus de puissance de calcul. Sans rentrer tout de suite dans les détails de l'architecture des cartes graphiques (que nous désignerons par le signe **GPU**, pour Graphics Processing Units), précisons tout de suite que c'est une architecture fortement parallèle. A titre d'exemple, si un ordinateur portable de milieu de gamme possède aujourd'hui un CPU à 4 cœurs cadencés à 2.13 GHz, son GPU possède de l'ordre de la centaine de "cœurs", bien que cadencés moins rapidement (1.25 GHz, seulement deux fois moins rapide).

nVidia et CUDA Il est alors normal d'envisager d'utiliser cette puissance de calcul pour des simulations numériques ! Et c'est l'objectif du fabricant nVidia, leader du marché des cartes vidéos, avec le langage **CUDA**. Précisons ici que la solution de nVidia n'est pas la seule disponible pour exploiter la puissance des GPUs, par exemple AMD, l'autre leader du marché, propose aussi un langage (*OpenCL*), et les solutions multi-cœurs CPU [2] openMP et MPI sont aussi utilisables.

Le choix pour CUDA est dû au partenariat qu'il existe entre nVidia et le CMLA.

0.2 Problématiques et plan

Comme il est indiqué dans notre sujet de stage, l'objectif est de produire un solveur [3] 2D pour modéliser des fluides compressibles via les équations d'Euler, sur CPU puis sur GPU. Et le but est de réussir à obtenir une simulation plus rapide sur GPU, via le langage CUDA de nVidia, que sur CPU via le langage C. Nous ne pouvions pas nous lancer de but en blanc dans cette réalisation, et donc nous vous présenterons d'abord le travail réalisé initialement, en guise de présentation des méthodes et des notations. Ensuite sera présenté le travail effectué sur les équations d'Euler, d'abord en accentuant la dimension théorique, puis en présentant plus en détails nos implémentations, leurs difficultés et leurs optimisations. Nous conclurons sur un élargissement, en présentant l'ajout d'interactivité dans une simulation, puis avec un retour d'expérience personnel pour chacun d'entre nous, et enfin nous présenterons un bilan des réalisations faites durant ce stage.

Des annexes sont aussi présentes, pour accueillir des remarques ou des points légèrement hors de propos dans le fil de pensée général de ce rapport. La table des matières donnée plus haut résume l'organisation de ce document.

1. Principalement pour des causes de diffusion de chaleur et de vitesse limite du courant électrique.
2. Ces outils permettent d'exécuter des programmes C en parallèle sur les différents cœurs d'un CPU, sans trop de modifications et de difficultés. Les performances sont améliorées mais ne peuvent pas permettre de gagner un facteur supérieur au nombre de processeurs (donc, pour le moment, de l'ordre de 4).
3. C'est à dire un programme permettant la résolution numérique de ces équations physiques dans certaines conditions initiales, et disposant d'une fonctionnalité d'affichage ou d'export des données calculées.

0.3 Remarques, notations

Avant d'aller plus loin, voici quelques remarques :
- les graphiques ont été réalisés en couleur dans la mesure du possible, et peuvent "mal rendre" à l'impression en noir et blanc ;
- les passages de codes sont insérés dans un but illustratif, afin de rendre plus concrets les explications données ;
- ce document est disponible en ligne, à l'adresse `http://www.dptinfo.ens-cachan.fr/~lbesson/publis/rapport_stage.pdf` [4] ;
- pour toutes remarques ou corrections, merci de bien vouloir nous contacter à l'un des adresses suivantes : `mailto:lilian.besson@ens-cachan.fr`, `mailto:sophie.hecht@ens-cachan.fr`, ou `mailto:maxime.isnard@ens-cachan.fr`.

Notations Les notations spécifiques à chaque parties seront introduites au fur et à mesure. Néanmoins, nous utilisons généralement les conventions suivantes :
- d indique le nombre de dimension d'espace de notre résolution ($d = 1, 2, 3$) ;
- N, M, et O sont des entiers, qui spécifient le nombre de points des discrétisations spatiales cartésiennes ;
- n ou **it** est un entier, qui indique le numéro d'étape de la discrétisation temporelle ;
- i, j, k, et l sont des entiers, qui sont utilisés comme *indices*. Souvent, k est pour les dimensions d'espaces ($2 + d$), et les autres pour les axes x, y, ou z ;
- h_x, h_y, et h_z ; ou Δx, Δy, et Δz représentent le pas d'espace sur chaque axe (ie $h_x = \frac{1}{N}$) ;
- c est une vitesse ou une concentration ;
- u, v, w, sont des vitesses, et $\overrightarrow{W}$ le vecteur vitesse ;
- F, G, H, f sont des flux réels (scalaires puis vectoriels) ;
- ϕ, ψ ou Φ est un flux numérique (scalaire puis vectoriel) ;
- S est un terme source ;
- E est l'énergie totale massique ;
- e est l'énergie interne massique ;
- H est l'enthalpie massique ;
- t représente le temps courant, et dt ou Δt le pas de temps du schéma ;
- $\otimes$ est le symbole utilisé pour le produit tensoriel (voir 2.2.1) ;
- . représente souvent un produit scalaire ;
- $\overrightarrow{n}$ est un vecteur unitaire et normal à une interface ;
- A est la matrice jacobienne d'un des flux F, G ou H :
- R et L^T sont respectivement les matrices de vecteurs propres à droite et à gauche de A ;
- **CFL** est le coefficient de Courant-Friedrich-Lévy.

Un paragraphe de remerciements se trouve à la fin de la conclusion (voir 4.5).

4. Si demandé, l'identifiant est 'Luc' et le mot de passe 'luc'.

1 Préliminaires : un premier contact, par l'équation d'advection

Le travail présenté dans cette première partie, bien que servant d'introduction à la suite, fait partie intégrante de nos réalisations. Les points suivants nous ont occupés du 1^{er} février au milieu du moins d'avril.

1.1 Premier contact théorique

1.1.1 Équation d'advection

Présentation L'équation d'advection décrit la façon dont une quantité est transportée dans un courant (par exemple un polluant dans de l'eau). Elle s'écrit de la façon suivante :

$$\partial_t u + div(\overrightarrow{c}\, u) = 0. \tag{1}$$

C'est une équation aux dérivées partielles, d'ordre un en temps et en espace. Nous cherchons à la résoudre pour $t \geq 0$ et sur $[0;1]$ en 1D ou $[0;1] \times [0;1]$ en 2D.

Solution exacte Cette équation a le mérite d'être simple et d'avoir une solution exacte connue en dimension 1.

$$u_{solution}(x,t) = u_0(x - c.t) \ \ , t \geq 0, x \in [0;1]^{per}. \tag{2}$$

avec c constant et u_0 la condition initiale au problème de Cauchy (supposée de classe C^1). Nous notons $x \in [0;1]^{per}$ pour signifier que la solution est périodique de période 1 sur $[0;1]$.

1.1.2 Schéma numérique

Cette équation d'advection sera donc un bon moyen pour appréhender les notions de système de lois de conservation, de schéma numérique, et permettra une première approche des méthodes d'implémentation sur C et CUDA en vue de l'implémentation des équations d'Euler.

Discrétisation Si on se base sur cette équation en 1 dimension, on peut effectuer une discrétisation en temps et en espace :

$$[0;1]^x \equiv \bigcup_{i \in [0;N-1]} \omega_i^x \ ; \ \omega_i^x = [i \times h_x; (i+1) \times h_x], \forall i \in [0; N-1] \tag{3}$$

h_x : pas d'espace selon x, constant ; Posons $x_i = ih_x + \frac{h_x}{2}$;

$$[0;T] \equiv [t_0, \ldots, t_{m-1}], m \leq 1 \ \ : \ \ t_n = t_{n-1} + \Delta t^n \tag{4}$$

Δt^n : pas de temps à l'étape n.

On adopte pour toute la suite le point de vue Eulérien : on ne suit pas une particule de fluide mais un point de la grille cartésienne.

Précisons aussi ici que toutes les discrétisations spatiales utilisées seront constantes au cours du temps, ce qui n'est pas le cas de tous les schémas numériques utilisés [5].

5. Notamment la méthode de Runge-Kutta d'ordre 4, qui utilise un pas d'espace décroissant.

Définition 1 (Cellule de contrôle)**.** On appelle **cellule** ou **volume de contrôle** chaque segment (en 1D), pavé (en 2D) ou volume élémentaire de la discrétisation précédente. Ils sont notés ici ω_i^x et ensuite $K_{i,j,k}$.

Intégration On va chercher à approcher numériquement les valeurs de nos variables (ici u mais aussi c ensuite), non pas par leurs valeurs en certains points, mais par leurs **valeurs moyennes sur chaque cellule de contrôle**. On a alors sous forme intégrale :

$$\frac{d}{dt}\int_{\omega_i^x} u(t,x)dx = -\int_{\omega_i^x}\frac{\partial}{\partial x}c.u(t,x)dx = -((c.u)_{x_{i+\frac{1}{2}}} - (c.u)_{x_{i-\frac{1}{2}}}) \tag{5}$$

Ces dérivées en temps et en espace sont alors approchées de la façon suivante,

Pour la dérivée en temps

$$\frac{d}{dt}\int_{\omega_i} u(t,x)dx \simeq \frac{u_i^{n+1}-u_i^n}{\Delta_t^n}h \tag{6}$$

On rappelle qu'on note aussi bien h que Δx ou hx pour le pas d'espace de notre discrétisation, quant il n'y a pas d'ambiguïté possible.

Pour la dérivée en espace Ici, il y a trois solutions possibles pour calculer les valeurs aux interfaces $(c.u)^n_{i+\frac{1}{2}}$:

Schéma centré(Instable) [6]

$$(c.u)^n_{i+\frac{1}{2}} \simeq \frac{(c.u)_i^n + (c.u)_{i+1}^n}{2} \tag{7}$$

Schéma décentré gauche ou droit (*upwind*) [7]

$$(c.u)^n_{i+\frac{1}{2}} \simeq (c.u)_i^n \; si\; c>0 \textbf{ ou } \simeq (c.u)_{i+1}^n \; si\; c<0 \tag{8}$$

Et en 2D De même pour l'équation 2D, on procède aussi par une discrétisation cartésienne du pavé $[0;1]\times[0;1]$. L'approximation donne à l'aide d'une formule de Green l'équation suivante (pour $i\in[0;N-1], j\in[0;M-1], n\in[0;T]$) :

$$\frac{u_{i,j}^{n+1}-u_{i,j}^n}{\Delta t_n} + \frac{1}{\Delta x\Delta y}[\phi^n_{i+1/2,j}\Delta y + \phi^n_{i-1/2,j}\Delta y + \phi^n_{i,j+1/2}\Delta x + \phi^n_{i,j-1/2}\Delta x] = 0 \tag{9}$$

en posant $\phi^n_{i+1/2,j} = (\overrightarrow{c}u.\overrightarrow{n})_{i+1/2,j}$ les flux numériques (voir [DeDu]).

6. Nous ne donnons pas la démonstration de ce point là ; mais nous avons effectivement constaté une grande instabilité de cette méthode en pratique.

7. Nous ne donnons pas la démonstration de ce point là ; mais nous avons effectivement constaté la bonne stabilité de cette méthode en pratique.

Méthode VFFC[8] Pour calculer ces flux numériques on utilise la méthode VFFC (Voir [Ghi99]) :

$$\phi^n_{i+1/2,j} = \frac{F(u_{i,j}) + F(u_{i+1,j})}{2}.\overrightarrow{n} - sgn(\overrightarrow{c}.\overrightarrow{n})\frac{F(u_{i+1,j}) - F(u_{i,j})}{2}.\overrightarrow{n} \tag{10}$$

avec $F(u) = \overrightarrow{c}\,u$.

On remarque que la présence de $sgn(c)$ donne à cette équation le même comportement binaire observé lors du choix gauche ou droite en 1.1.2. On appelle ce phénomène le principe de se placer "*sous le vent*". C'est-à-dire qu'il faut aller chercher l'information dans la direction depuis laquelle elle nous arrive.

1.1.3 Splitting : un schéma 2D devient 2 schémas 1D

Pour calculer les valeurs $u^{n+1}_{i,j}$ en fonction des valeurs en n, on utilise une méthode de découplement appelée **méthode de splitting**.

Dans ce cas, il s'agit de poser $t^*_n \in]t_n; t_{n+1}[$ un temps intermédiaire, qui n'a pas de réalité physique, mais qui nous permet d'obtenir le système d'équation suivant (après simplification) :

$$\begin{cases} \dfrac{u^{n^*}_{i,j} - u^n_{i,j}}{\Delta t_n} + \dfrac{1}{\Delta x}[\phi^n_{i+1/2,j} + \phi^n_{i-1/2,j}] = 0; \\ \dfrac{u^{n+1}_{i,j} - u^{n^*}_{i,j}}{\Delta t_n} + \dfrac{1}{\Delta y}[\phi^n_{i,j+1/2} + \phi^n_{i,j-1/2}] = 0. \end{cases} \tag{11}$$

En sommant ces 2 équations on retrouve notre équation à résoudre : on ne perd rien en introduisant ce temps intermédiaire. Mais par contre, cela permet de simplifier le problème puisqu'on a désormais deux fois un problème 1D a calculer, et on sait déjà résoudre un tel problème avec le schéma présenté plus haut. Cette technique est appelée la **méthode de splitting**. Dans la suite, nous désignons par *splitting en x* la résolution de la première composante de l'équation précédente, et par *splitting en y* la résolution de sa seconde composante. En pratique, il faut bien-sûr réaliser les deux étapes dans le bon ordre : d'abord le splitting horizontal puis vertical. Le calcul des flux numériques est effectué ici de la même manière que pour le schéma 1D.

1.1.4 Stabilité d'un tel schéma

Le schéma présenté plus haut est **explicite**, d'ordre 1 en espace et en temps car on approche chaque dérivée par un développement limité à l'ordre 1. En ce qui concerne la **stabilité** dans le cas 1D, on remarque que notre schéma peut s'écrire de la façon suivante :

$$u^{n+1}_i = u^n_i(1 - \frac{\Delta t.c}{h}) + u^n_{i-1}.\frac{\Delta t.c}{h} \tag{12}$$

qui est une combinaison convexe des $(u^n_i)_i$ ssi $0 \leq \frac{\Delta t.c}{h} \leq 1$. Ainsi cette condition nécessaire et suffisante sur le pas de temps nous assure que la solution numérique sera bornée. Cela impose donc le choix correct du pas de temps.

En pratique on se donne un paramètre de sureté (coefficient CFL) $\Delta t = CFL.\frac{c}{h}$ avec $CFL \simeq 0.9$. Ce coefficient tire son nom des mathématiciens Courant, Friedrich et Lévy, qui sont à l'origine d'une telle relation de stabilité.

Plus le coefficient CFL sera faible, plus la stabilité sera assurée, mais ceci ralentit l'évolution du système.

8. Pour volumes finis à flux caractéristiques.

1.2 Implémentations

1.2.1 Implémentation en C et en CUDA

Avant d'aller plus en profondeur dans le travail réalisé, nous pourrions présenter ici quelques points concernant nos méthodes de développement. Une présentation des langages et des outils utilisés semblant nécessaire, le lecteur intéressé pourra se référer à cette partie 5.1 en annexe.

Il s'agit plutôt de faire partager notre expérience qu'un véritable contenu intéressant, cette section est donc laissée en annexe.

1.2.2 Structure de nos codes et affichage des données (GNUplot)

Notre implémentation du schéma présentée plus haut s'est d'abord faite en 1D, en C. Puis en CUDA. Et enfin, nous avons passé les deux codes en 2D assez facilement.

Structure générale Ce programme consiste en une phase d'initialisation, où l'on déclare chaque variable utilisée, puis on alloue la mémoire utilisée pour les tableaux, ensuite ils sont initialisés et on donne aux paramètres de la simulation leurs valeurs (nombre de point, vitesse initiale etc ...).

Ensuite vient une boucle en temps, qui consiste à calculer les valeurs des flux puis à actualiser les valeurs des variables conservatives.

Un grapheur ? Il nous a donc fallu nous demander comment nous allions récupérer les données. Au début, la méthode qui a paru la plus facile était simplement d'afficher les données numériques dans le terminal, mais cela ne nous permettait pas de vérifier l'exactitude des résultats, ni d'avoir une réelle idée de l'évolution de la simulation.

Nous avons donc cherché à obtenir une visualisation. Pour cela nous avons choisi le logiciel GNUPlot[9]. Cet utilitaire a de nombreux avantages : gratuit, Open Source sous licence GPL, multi-plateforme, et *surtout* assez facile à prendre en main. Bien qu'il possède une multitude de fonctionnalités, il est facile de réaliser des choses simples avec ce logiciel.

La première utilisation que nous en avons faite est un simple affichage d'une courbe en 1D, à partir des données numériques écrites dans un fichier dans la mémoire ROM[10] de l'ordinateur.

Ensuite, il suffit de lancer GNUPlot en ligne de commande et d'utiliser le fichier produit, ou les fichiers produits. Une amélioration intéressante est de lancer directement l'affichage en tant que sous processus du programme C.

Le code qui nous permet de faire tout ceci est le suivant :

```
1 // Une fonction pour ecrire dans le fichier <nom> le vecteur <f> de taille <n>
     qui prend ses valeurs en <x>
2 void EcrireGnuplot(const char* nom, float* x, float* f, int N){
3  int i;          // indice de boucle
4  FILE *file=fopen(nom,"w");      // on ouvre le fichier en ecriture "w"
5  for (i=0 ; i<N ; i++)
6   fprintf(file ," %f %f \n", x[i],f[i]); // on ecrit la case x et la valeur de f
7  fclose(file);  // penser a fermer les fichiers ouverts evite bien des erreurs '
     segmentation fault'
```

9. Voir `http://www.gnuplot.info/` pour des détails. Pour une introduction, on pourra se référer à `http://enseignement.ensi-bourges.fr/cours/GNUPLOT/document/gnuplot.html`, ou `http://bdesgraupes.pagesperso-orange.fr/UPX/Master1/presentation_gnuplot.pdf`.

10. Aussi appelée mémoire morte en français, en opposition à la mémoire vive où sont stockées les valeurs durant l'évolution de nos simulations sur CPU, ou à la mémoire GPU pour les simulations exécutées via CUDA.

```
8  }
9  int main (int argc, const char * argv[]) // fonction principale
10 { ...
11  FILE *gnuplot = popen("/usr/bin/gnuplot","w"); // sous processus
12  ...
13  char filename[50]; // nom du fichier texte ou seront ecrites les donnes
14  ...
15  while ((t <= T)&&(it <= BORNE)){
16   sprintf(filename, "valeurs_U_%d.gp", it); // un fichier a chaque etape
17   ...
18   // Plus loin, a la fin de chaque boucle en temps
19   EcrireGnuplot(filename, x, U, N);
20   fprintf(gnuplot, "plot [0:1][0.5:1.5] \"%s\" with lines\n", filename);
21   fflush(gnuplot); // lance l'affichage
22  }
23  pclose(gnuplot); // il faut aussi fermer le sous processus
24 }       // fin du Main
```

Cet extrait de code présente donc comment nous affichons le résultat des simulations numériques en 1D. Plus de détails sur les autres moyens d'affichage utilisés seront présentés plus loin.

1.2.3 Un exemple d'affichage en 2D

Les premières simulations ont été faites sur le cas simple où c est constant. Les images suivantes sont les résultats d'une simulation avec des conditions de bords périodiques [11], sur un maillage cartésien de 120x120 points.

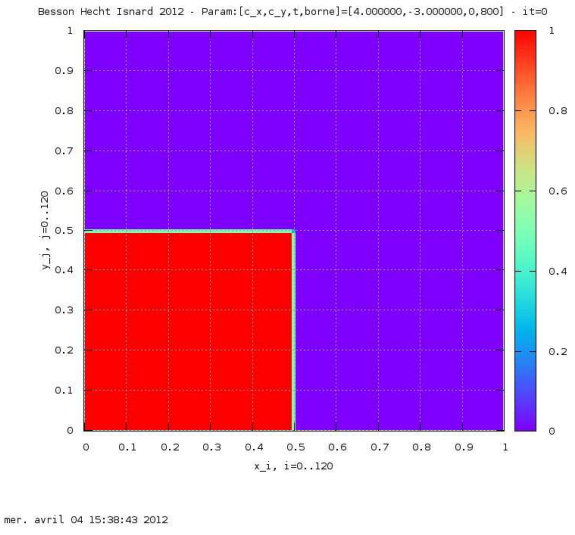


FIGURE 1 – Advection, 2D, avec $\overrightarrow{c} = (4, -3), N = 120 \times 120$, et des conditions limites périodiques. `it=0`

11. Plus d'informations à propos des conditions de bords sont données ici 2.4.

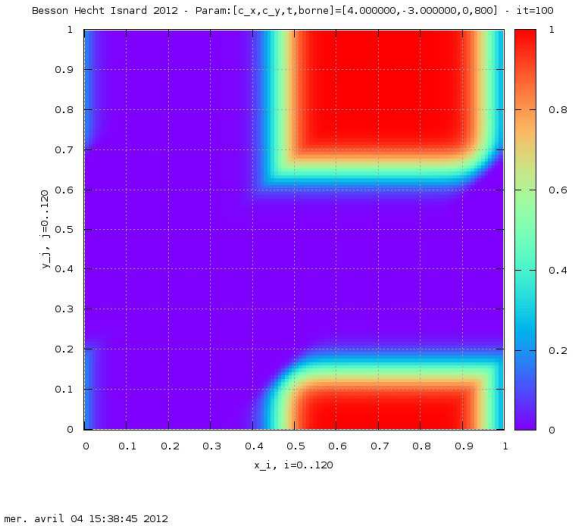


FIGURE 2 – Advection, 2D, avec $\overrightarrow{c} = (4, -3), N = 120 \times 120$, et des conditions limites périodiques. `it=100`

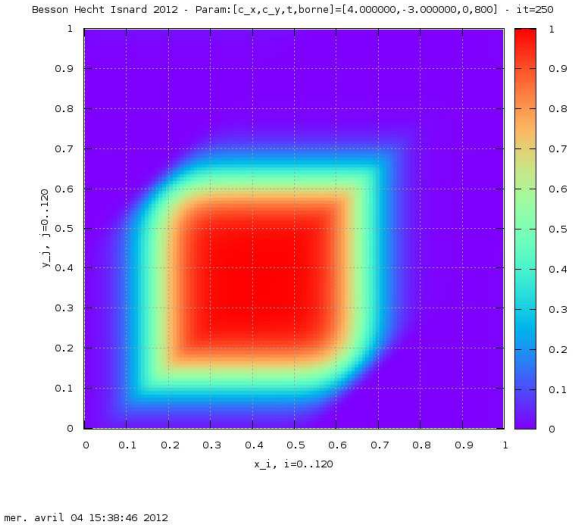


FIGURE 3 – Advection, 2D, avec $\overrightarrow{c} = (4, -3), N = 120 \times 120$, et des conditions limites périodiques. `it=250`

Ces quelques images permettent de constater que le carré de masse volumique élevée se déplace au cours du temps, dans la direction globale du vecteur $\overrightarrow{c}$. Par ailleurs, la troisième image montre que la forme initialement carré a été déformée.

Diffusion numérique Ce phénomène de divergence par rapport à la solution exacte s'appelle la *diffusion numérique*.

C'est-à-dire que les valeurs calculées s'éloignent progressivement des valeurs réelles. En effet, nous avons présenté plus haut la solution exacte à l'équation d'advection, qui consiste dans ce cas en une simple "translation" du carré initial au cours du temps. La simulation numérique, du fait de son inexactitude, introduit petit à petit une différence entre les valeurs numériques (celles qui sont affichées) et les valeurs réelles.

Dans ce cas précis, ceci est dû au fait que nous avons choisi un coefficient CFL < 1. Si on refait la même simulation avec ce coefficient valant 1, on constate que la solution numérique reste égale à la solution réelle, dans le cas 1D.

Il faut aussi noter qu'un autre source d'erreur se situe dans l'utilisation d'un ordinateur pour faire nos calculs. En effet, nous utilisons des types de données ayant une précision *finie*. Par exemple, les "réels" en C sont représentés dans nos codes par le type `float`, qui contient seulement 6 décimales. Ainsi, les erreurs de calcul arrivent assez vite, et se propagent de façon exponentielle.

1.2.4 Rapide présentation du CUDA

Une fois le programme écrit en C, il a fallu se mettre au CUDA. La première étape était donc de comprendre l'organisation d'une carte vidéo.

Un peu d'architecture Un ordinateur est constitué de plusieurs entités, dont entre autre le processeur (CPU) et la carte graphique. Une carte graphiques est composée d'une série processeur, le GPU (Graphical Processing Units), d'une mémoire propre, et d'un composant permettant l'échange de données et d'instructions avec le processeur.

Le GPU est organisé en plusieurs *grilles* qui sont elles même constituées de plusieurs *blocs*. Lors de l'utilisation du GPU, ces blocs seront organisés de *threads*. Les langages tel que CUDA ou OpenCL permettent d'effectuer plusieurs opérations en parallèle : il est en effet possible de *lancer* plusieurs threads en même temps.

C'est la toute l'utilité du calcul parallèle : **faire plusieurs choses en même temps**. Ce type de programmation permet, dans des bonnes conditions d'utilisation, d'obtenir une vitesse d'exécution bien supérieure aux limites du paradigme séquentiel.

Dompter le GPU Pour pouvoir effectuer des opérations en parallèle sur le GPU en CUDA, il faut écrire des fonctions appelées *kernels*. Ces fonctions sont comme les fonctions en C, mais n'auront pas le même comportement.

On les appelle ensuite dans la fonction main en indiquant le nombre de blocs et le nombre de threads par bloc que l'on veut utiliser.

Il faut préciser ici que c'est du travail du programmeur que de trouver la bonne organisation parallèle. Et que ce n'est pas facile ...

L'exemple suivant montre comment on peut définir un kernel CUDA, définir les paramètres avec lesquels il sera appellés, et l'appeler.

```
1        // Exemple kernel
2        __device__ void plus_un_p(float* p){
3                int i = threadIdx.x + blockIdx.x * blockDim.x;
4                p[i] += 1.0; // augmentation globale de la pression
5        }
6
7        // Exemple valeurs :
8        int TAILLE_P    = 1024; // nvcc l'utilisera comme un "dim3" valant
             (1024,1,1)
```

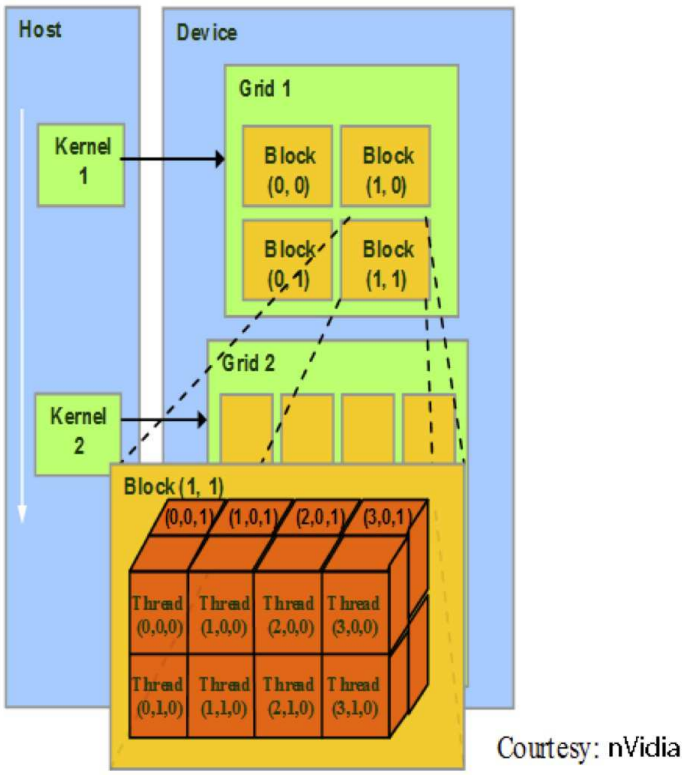
```
9         int THREADS      = 32;    // m me remarque
10        int BLOCKS       = (TAILLE_P / 32):
11
12         // Exemple appel a un kernel
13         plus_un_p<<<THREADS, BLOCKS>>>(p_gpu); // ajoute 1 dans chaque case du
               vecteur GPU p_gpu
```

Il faut aussi noter que du fait de l'organisation *dichotomique* d'un GPU, le nombre de threads et de blocks est nécessairement une puissance de 2.

Différents types de kernel Lors de la définition du kernel d'exemple plus haut, nous avons préfixé son type de retour d'un mot clé entre "__". Il existe 3 types de kernel, et il relève du travail du développeur de préciser le type de chaque fonction déclarée dans un code CUDA, à l'aide d'un des mots clé suivants :

__***global***__ correspond à un kernel exécuté sur le GPU mais appelé par le CPU ;
__***device***__ correspond à un kernel exécuté et appelé par le GPU ;
__***host***__ correspond à un kernel exécuté et appelé par le CPU (comme une fonction normale en C, c'est le fonctionnement par défaut).

Paradigme SIMD Le schéma suivant détaille l'organisation particulaire décrite plus haut (un GPU = des grilles, une grille = des blocs et chaque bloc est utilisé avec plusieurs threads) :



Pour effectuer toutes ces opérations en parallèle on utilise le paradigme SIMD (**Single Instruction** Stream, **Multiple Data** Stream).

C'est-à-dire que les N processeurs identiques opèrent sous contrôle d'un flot d'instructions unique généré et envoyé par une unité centrale unique : le CPU dirige l'exécution globale d'un

programme CUDA, et donne des ordres et des données au GPU. Les threads opèrent de façon synchrone s'ils le peuvent.

Voici un exemple de kernel présentant le paradigme SIMD :

```
1 __global__ void u_actualise_dev(float *ubar_dev, float *u_dev, int signe_c_dev,
    int N_dev){
2 //for (i=0; i <= N; i++){..} est fait par le decoupement en groupe de threads
    et en blocs :
3 int i = threadIdx.x + blockIdx.x * blockDim.x;
4  // on realise le "mapping" des indices ici
5 while (i < N_dev){
6 u_dev[i] -= signe_c_dev * 0.9 * (ubar_dev[i+1] - ubar_dev[i]);
7 i += blockDim.x * gridDim.x; // paradigme SIMD
8 }}
```

C'est la ligne 3 `int i = threadIdx.x + blockIdx.x * blockDim.x;` qui nous permet d'appeler plusieurs threads en même temps, suivant le nombre de blocs et de threads que nous avons déclaré lors de l'appel du kernel.

Si jamais la dimension du vecteur est trop grande pour qu'elle ne soit pas parcourue en entier par la liste d'indice i, il est possible de continuer le parcours en incrémentant i via l'instruction `i += blockDim.x * gridDim.x;`.

Dans le cas de vecteur de dimensions supérieures, il faut adapter les lignes précédentes, par exemple avec `int j = threadIdx.y + blockIdx.y * blockDim.y;`, et ensuite `j += blockDim.y * gridDim.y;`.

Précisons ici qu'il n'est possible de *mapper* que trois dimensions d'indices. C'est notamment pour cette raison que nous ne présentons aucune simulation numérique faite dans un espace à plus de trois dimensions d'espaces. [12]

D'autres explications sur le CUDA sont présentées en annexe, notamment comment déclarer des éléments à stocker dans la mémoire présente sur la carte graphique, comment échanger des données entre cette mémoire et la mémoire RAM [13] du CPU.

1.3 Aller un peu plus loin

Les résolutions qui ont été faite dans les paragraphes précédents, restent assez sommaires. Il va falloir rajouter une difficulté supplémentaire afin qu'elles permettent de modéliser des phénomènes physiques.

L'exemple retenu sera l'évolution d'un polluant dans un lac sous l'influence d'un champs tournant, et nous allons présenter ici les différentes étapes qui mènent à l'animation obtenue.

1.3.1 Une application plus concrète

Il s'agit simplement de rajouter une nouvelle équation, en faisant désormais intervenir la concentration du fluide étudié en plus de sa densité.

12. Mais unr certaine part du travail théorique présenté précédemment et plus loin s'applique en dimension d quelconque.

13. Aussi appelée mémoire vive.

Dès lors, les équations qui caractérisent le fluide parfait sont les suivantes :

$$\begin{cases} \dfrac{\partial \rho}{\partial t} + div(\overrightarrow{u}\,\rho) = 0; \\ \dfrac{\partial \rho c}{\partial t} + div(\overrightarrow{u}\,\rho c) = 0. \end{cases} \tag{13}$$

Nous utilisons encore les notations précédemment employées, mais il convient de noter que c ne désigne plus la vitesse de l'environnement : ρ : densité du fluide ;
c : concentration du polluant ;
Pour l'exemple étudié, nous avons choisi de prendre un champ de vitesse tournant constant au cours du temps comme : $\overrightarrow{u}(x,y) = \begin{bmatrix} -y \\ x \end{bmatrix} (x-1)(x+1)(y-1)(y+1)$. Cette fois, la résolution est effectuée sur $[-1;1] \times [-1;1]$.

1.3.2 Schéma numérique actualisé

Il est évident qu'il faut adapter le schéma précédent, ce que nous présentons ici. Ensuite sera présenté un second moyen d'affichage, puis le résultat de ce travail sera illustré avec quelques graphiques. Nous conclurons en évoquant la faiblesse des deux moyens d'affichages jusqu'alors présentés.

Quel changement ? Cette fois ci, il y a donc 2 équations et deux variables. Les données qui nous intéressent sont ρ et c. Pour obtenir c, il faut donc résoudre les deux équations différentielles et diviser les résultats ($\frac{\rho c}{\rho} = c$).

Il va donc falloir faire la même discrétisation spatio-temporelle pour les deux membres. La seule nuance avec les équations précédentes et la première équation d'advection étudiée initialement est le coefficient noté plus haut c et noté ici $\overrightarrow{u}$.

Dans notre système, $\overrightarrow{u}$ n'est pas une constante [14]. Cela ne change rien au schéma précédent car les calculs on été faits comme si ce coefficient était variable.

Rapide présentation du nouveau schéma Notre schéma est donc désormais le suivant :

$$\begin{cases} \dfrac{\rho_{i,j}^{n+1} - \rho_{i,j}^{n}}{\Delta t_n} + \dfrac{1}{\Delta x \Delta y}[\phi_{i+1/2,j}^{n}\Delta y + \phi_{i-1/2,j}^{n}\Delta y + \phi_{i,j+1/2}^{n}\Delta x + \phi_{i,j-1/2}^{n}\Delta x] = 0; \\ \dfrac{\rho c_{i,j}^{n+1} - \rho c_{i,j}^{n}}{\Delta t_n} + \dfrac{1}{\Delta x \Delta y}[\varphi_{i+1/2,j}^{n}\Delta y + \varphi_{i-1/2,j}^{n}\Delta y + \varphi_{i,j+1/2}^{n}\Delta x + \varphi_{i,j-1/2}^{n}\Delta x] = 0. \end{cases} \tag{14}$$

en posant $\phi_{i+1/2,j}^{n} = (\rho\,\overrightarrow{u}.\overrightarrow{n})_{i+1/2,j}$, et $\varphi_{i+1/2,j}^{n} = (\rho c\,\overrightarrow{u}.\overrightarrow{n})_{i+1/2,j}$.

Pour le choix du flux, on utilise encore la méthode VFFC. On a alors :

$$\begin{cases} \phi_{i+1/2,j}^{n} = \dfrac{F(\rho_{i,j}) + F(\rho_{i+1,j})}{2}.\overrightarrow{n} - sgn(\overrightarrow{u}.\overrightarrow{n})\dfrac{F(\rho_{i+1,j}) - F(\rho_{i,j})}{2}.\overrightarrow{n}; \\ \varphi_{i+1/2,j}^{n} = \dfrac{F(\rho c_{i,j}) + F(\rho c_{i+1,j})}{2}.\overrightarrow{n} - sgn(\overrightarrow{u}.\overrightarrow{n})\dfrac{F(\rho c_{i+1,j}) - F(\rho c_{i,j})}{2}.\overrightarrow{n}. \end{cases} \tag{15}$$

On fait ensuite un splitting directionnel. Le reste de la résolution est très simillaire à ce qui a été fait plus haut.

14. Ainsi, cette simulation s'appelle à juste titre : "simulation avec champ de vitesse variable".

On aboutit donc à un schéma explicite, d'ordre 1 en temps et en espace, décentré ; et qui présente une condition de stabilité simillaire à celle évoquée plus haut (voir 1.1.4).

1.3.3 Un autre moyen d'affichage (VTK et ParaView)

La visualisation avec GNUPlot est assez rapide pour des schémas 1D mais elle commence à montrer de cruels signes de lenteur dès que l'affichage demande du 2D.

Nous avons donc cherché, suite à l'impulsion de Daniel, une nouvelle méthode d'affichage, qui serait plus rapide. Avec ces conseils, nous avons donc choisit d'utiliser le couple VTK et ParaView.

VTK Ce sigle désigne initialement une énorme bibliothèque d'affichage disponible en C et en C++, et signifie *Visualisation Tool Kit*. De cette bibliothèque est dérivé un format de données numériques, qui semble être très couramment employé dans le domaine de la simulation numérique. Une description précise du format de données que nous utiliserons par la suite est disponible à l'adresse suivante : `http://www.vtk.org/VTK/img/file-formats.pdf`.

ParaView ParaView est un logiciel libre, gratuit et multi-plateforme, permettant une visualisation efficace, esthétique, performante et aux options très variées à partir de données écrites notamment dans le format VTK. Il est disponible à l'adresse suivante : `http://www.paraview.org/paraview/resources/software.php`.

A noter que ce n'est pas le seul logiciel permettant ce genre de visualisation.

Le couple VTK et ParaView permet une visualisation post-mortem, c'est-à-dire que durant l'exécution du programme, les données calculées sont enregistrées dans un fichier .vtk. Et après l'exécution, il est possible de visualiser un graphique avec ParaView.

1.3.4 Résultats

Voici quelques graphiques présentant les résultats obtenues lors de l'implémentation du schéma précédent.

Il faut les faire et les inclure !

1.3.5 Problèmes de communications

L'un des principaux problèmes en CUDA est le suivant : le GPU a une mémoire propre.

Allocation de mémoire Pour pouvoir utiliser des tableaux dans des kernels exécutés sur le GPU, il faut donc les créer sur le périphérique. En C, l'allocation de mémoire pour des tableaux se fait via la fonction `malloc(<nombre d'octet a donner>)`. Elle est appellée de la manière suivante :

```
1 u_cpu  = (float*) malloc(N*M * sizeof(float));
```

Son équivalent en CUDA est la fonction `cudaMalloc`.

Elle est appellée de la manière suivante :

```
1 cudaMalloc( (void**)&u , N*M * sizeof(float));
```

Echange de mémoire Or nous voulons aussi utiliser sur l'hôte les valeurs de nos tableaux, ne serait-ce que pour récupérer les données et les imprimer dans un fichier externe.

Tous les tableaux qui veulent être affichés sont donc créés sur le CPU **et** le GPU. Il faut alors penser à recopier les valeurs du tableaux depuis le GPU vers son équivalent CPU à chaque pas de temps.

Sachant qu'il faut faire ces opération pour plusieurs tableaux, d'une taille correspondant a celle de notre nombre de points en espaces, les échange mémoire deviennent rapidement nombreux.

Ces échanges entre les deux mémoires CPU et GPU sont effectués comme suit :

```
1 cudaMemcpy( rho, rho_cpu, N*M   * sizeof(float), cudaMemcpyHostToDevice );
    // sens CPU --> GPU
2 // plus loin
3 cudaMemcpy( rho_cpu, rho, N*M   * sizeof(float), cudaMemcpyDeviceToHost );//
    Sens GPU --> CPU
```

Où se situe la lenteur ? On pourrait penser que ce sont ces échanges entre les deux mémoires vives de l'hôte et du périphérique qui sont les plus couteux en temps. Mais en fait, c'est surtout la lenteur de l'écriture en mémoire ROM pour exporter en .vtk qui prend beaucoup de temps.

Exemple 1 (Lenteur de l'écriture en mémoire ROM)**.** A titre d'exemple, lors d'une simulation à 512x512 points, en CUDA pour nos Euler 2D, et en faisant 128 étapes. En comptant le temps passé à la compilation et à l'initialisation, 28% du temps est passé à écrire un seul des vecteurs (512 x 512) en .vtk à chaque étape.

2 Équation d'Euler

L'étude des schémas numériques pour les équations d'Euler compressibles est un préalable à la simulation d'écoulements visqueux par les équations de Navier-Stokes. D'une manière générale, il s'agit d'une généralisation des simulations effectuée précédemment.

Les équations d'Euler, depuis leur origine physique jusqu'aux résultats des simulations en passant par les détails du schéma numérique utilisé, constitueront donc notre sujet d'étude pour la partie suivante.

2.1 Des équations physiques au schéma numérique

2.1.1 Un peu de physique

Présentation du système Les équations suivantes résultent de $2 + d$ équations provenant directement de l'étude physique.

- Une équation de continuité (conservation de la masse)

$$\partial_t \rho + div(\rho.\overrightarrow{W}) = 0; \tag{16}$$

- Des équations du mouvement (1 par dimension)

$$\partial_t(\rho.\overrightarrow{W}) + div(\rho.\overrightarrow{W} \otimes \overrightarrow{W} + p.\overline{\overline{Id}}) = 0; \tag{17}$$

- Une équation d'énergie (température)

$$\partial_t(\rho.e + \frac{1}{2}.\rho.W^2) + div((\rho.e + \frac{1}{2}.\rho.W^2 + p).\overrightarrow{W}) = 0. \tag{18}$$

Fermer le système Il manque alors une équation pour fermer le système. En effet, nous avons $3 + d$ variables (ρ, p, e et $\overrightarrow{W}$) pour $2 + d$ équations. On considère l'équation d'état des gaz parfaits suivante :

$$e = \frac{1}{\gamma - 1} \times \frac{p}{\rho}. \tag{19}$$

Il faut bien noter que ceci implique une restriction du domaine d'application de ces équations d'Euler.

Nous utilisons les notations suivantes (inspirée de celle de [Buf06]) :

- p : pression,
- ρ : masse volumique,
- $\overrightarrow{W} = [u, v, w]^T$: vitesse du fluide (1, 2 ou 3 composantes) ;
- e : énergie interne massique du fluide,
- E : énergie totale massique du fluide, $E = e + \frac{1}{2}W^2$;
- γ : rapport des chaleurs spécifiques (constant) et vaut $\gamma = 1.4$ pour un gaz diatomique (air) ;
- $\otimes$: produit tensoriel [15],
- $\overline{\overline{Id}}$: tenseur identité ;
- c est la vitesse du son : $c = \sqrt{\frac{\gamma \times p}{\rho}}$.

15. Pour une définition et un exemple, voir 2.2.1.

2.1.2 Écriture vectorielle

Nous pouvons maintenant présenter les équations sous forme vectorielle, en 3 dimensions :

$$\overrightarrow{U} \equiv [\rho; \rho\overrightarrow{W}; \rho E]^T = [\rho; \rho u; \rho v; \rho w; \rho E]^T; \tag{20}$$

$$\frac{\partial}{\partial t}\overrightarrow{U} + \frac{\partial}{\partial x}\overrightarrow{F}(\overrightarrow{U}) + \frac{\partial}{\partial y}\overrightarrow{G}(\overrightarrow{U}) + \frac{\partial}{\partial z}\overrightarrow{H}(\overrightarrow{U}) = \overrightarrow{S}(\overrightarrow{U}). \tag{21}$$

Les flux associés sont alors définis par les trois formules suivantes :

$$\overrightarrow{F}(\overrightarrow{U}) \equiv \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uw \\ u(E+p) \end{bmatrix}; \ \overrightarrow{G}(\overrightarrow{U}) \equiv \begin{bmatrix} \rho v \\ \rho vu \\ \rho v^2 + p \\ \rho vw \\ v(E+p) \end{bmatrix}; \ \overrightarrow{H}(\overrightarrow{U}) \equiv \begin{bmatrix} \rho w \\ \rho wu \\ \rho wv \\ \rho w^2 + p \\ w(E+p) \end{bmatrix}.$$

Une formule plus générale donne l'expression du flux $\overrightarrow{f}_{\overrightarrow{n}}(\overrightarrow{U})$ à l'interface de normale $\overrightarrow{n}$:

$$\overrightarrow{f}_{\overrightarrow{n}}(\overrightarrow{U}) \equiv \begin{bmatrix} \rho(\overrightarrow{W}.\overrightarrow{n}) \\ \rho(\overrightarrow{W}.\overrightarrow{n})\overrightarrow{n} + p\overrightarrow{n} \\ (E+p)(\overrightarrow{W}.\overrightarrow{n}) \end{bmatrix}. \tag{22}$$

De plus, $\overrightarrow{S}(\overrightarrow{U})$ est un terme source (voir 2.4.4), qui peut être nul.

2.1.3 Discrétisation et splitting

Nous allons alors résoudre ce problème à l'aide d'un splitting (trois étapes mono-dimensionnelles).

Discrétisation en temps et en espace En utilisant les mêmes notations que pour l'équation d'advection on intègre sur une cellule d'espace $K_{i,j,k}$:

$$\frac{\partial}{\partial t}\int_{K_{i,j,k}} \overrightarrow{U}\, dV \simeq \frac{\overrightarrow{U}^{n+1}_{i,j,k} - \overrightarrow{U}^{n}_{i,j,k}}{\Delta t^n}\Delta x\Delta y\Delta z. \tag{23}$$

On obtient ainsi le schéma suivant :

$$\frac{\overrightarrow{U}^{n+1}_{i,j,k} - \overrightarrow{U}^{n}_{i,j,k}}{\Delta t_n} + \frac{1}{\Delta x\Delta y\Delta z}[(\overrightarrow{\phi x}^n_{i+1/2}\Delta y\Delta z + \overrightarrow{\phi x}^n_{i-1/2}\Delta y\Delta z) +$$

$$(\overrightarrow{\phi y}^n_{j+1/2}\Delta x\Delta z + \overrightarrow{\phi y}^n_{j-1/2}\Delta x\Delta z) + (\overrightarrow{\phi z}^n_{k+1/2}\Delta x\Delta y + \overrightarrow{\phi z}^n_{k-1/2}\Delta x\Delta y)] = 0. \tag{24}$$

Nous désignons ici par $\overrightarrow{\phi x}$ les flux numériques à l'interface rectangulaire de taille $\Delta y \times \Delta z$, orthogonale à l'axe x. De même pour les deux autres dimensions.

Splitting On va alors , comme pour l'advection, utiliser un splitting directionnel en écrivant notre schéma sous la forme suivante :

$$\begin{cases} \dfrac{\overrightarrow{U}^{n^*}_{i,j,k} - \overrightarrow{U}^{n}_{i,j,k}}{\Delta t_n} + \dfrac{1}{\Delta x}[\phi x^n_{i+1/2,j,k} + \phi x^n_{i-1/2,j,k}] = 0; \\ \dfrac{\overrightarrow{U}^{n^{**}}_{i,j,k} - \overrightarrow{U}^{n^*}_{i,j,k}}{\Delta t_n} + \dfrac{1}{\Delta y}[\phi y^n_{i,j+1/2,k} + \phi y^n_{i,j-1/2,k}] = 0; \\ \dfrac{\overrightarrow{U}^{n+1}_{i,j,k} - \overrightarrow{U}^{n^{**}}_{i,j,k}}{\Delta t_n} + \dfrac{1}{\Delta z}[\phi z^n_{i,j,k+1/2} + \phi z^n_{i,j,k-1/2}] = 0. \end{cases} \tag{25}$$

Cette fois, deux temps intermédiaires sont introduits, n^* et n^{**}. On notera la grande ressemblance avec le système présenté précédemment. Encore une fois, en sommant les d équations du système précédent, on retrouve l'écriture présentée à l'équation 2.1.3.

Calcul des flux Rappel de l'expression des flux ϕ pour l'advection (selon x)

$$\phi^n_{i+1/2,j} = \frac{F(u_{i,j}) + F(u_{i+1,j})}{2}\overrightarrow{n} - sgn(\overrightarrow{c}.\overrightarrow{n})\frac{F(u_{i+1,j}) - F(u_{i,j})}{2}\overrightarrow{n} \tag{26}$$

De même on a ici :

Pour les flux selon x Se transforme ici en (On n'écrit pas les j, k pour chaque valeur) :

$$\overrightarrow{\phi x}^n_{i+1/2} = \frac{\overrightarrow{F}(\overrightarrow{U}^n_{i+1}) + \overrightarrow{F}(\overrightarrow{U}^n_i)}{2} - sign(A^n_{i+1/2})\frac{\overrightarrow{F}(\overrightarrow{U}^n_{i+1}) - \overrightarrow{F}(\overrightarrow{U}^n_i)}{2} \tag{27}$$

où on a posé $A^n_{i+1/2}$ la matrice jacobienne du vecteur $\overrightarrow{F}(\overrightarrow{U}^n_{i+1/2})$.
$\Longrightarrow$ *La difficulté est le calcul de cette matrice signe !* Ces détails sont présentés plus loin.

2.1.4 Première résolution

Nous avons tout d'abord implémenter la résolution de ce schéma en C, en 1D. Avant d'aller plus loin dans les détails sur le schéma, nous présentons ici les résultats de cette première simulation.

On observe une déformation du rectangle initial, ainsi qu'un déplacement, symétrique par rapport au milieu de ce rectangle. De plus, nous sommes rassurés d'observer le fait que les valeurs restent bornées dans l'intervalle de départ ($[0.125; 1.0]$).

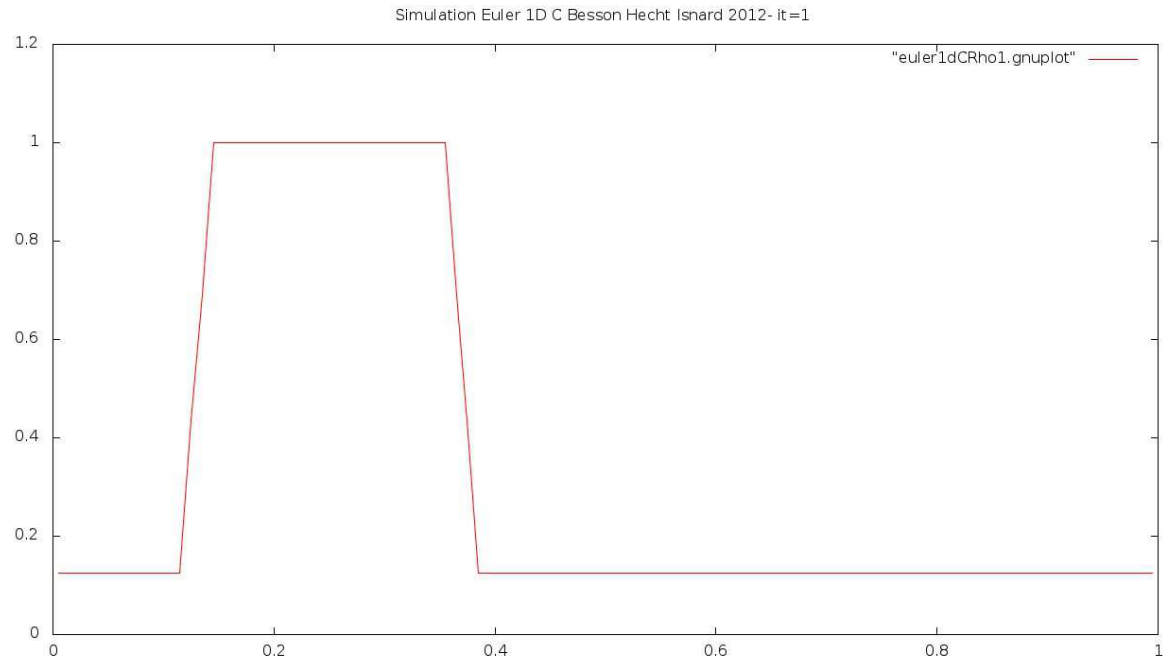


FIGURE 4 – Affichage en 1D de ρ pour Euler compressible (GNUPlot). `it = 1`

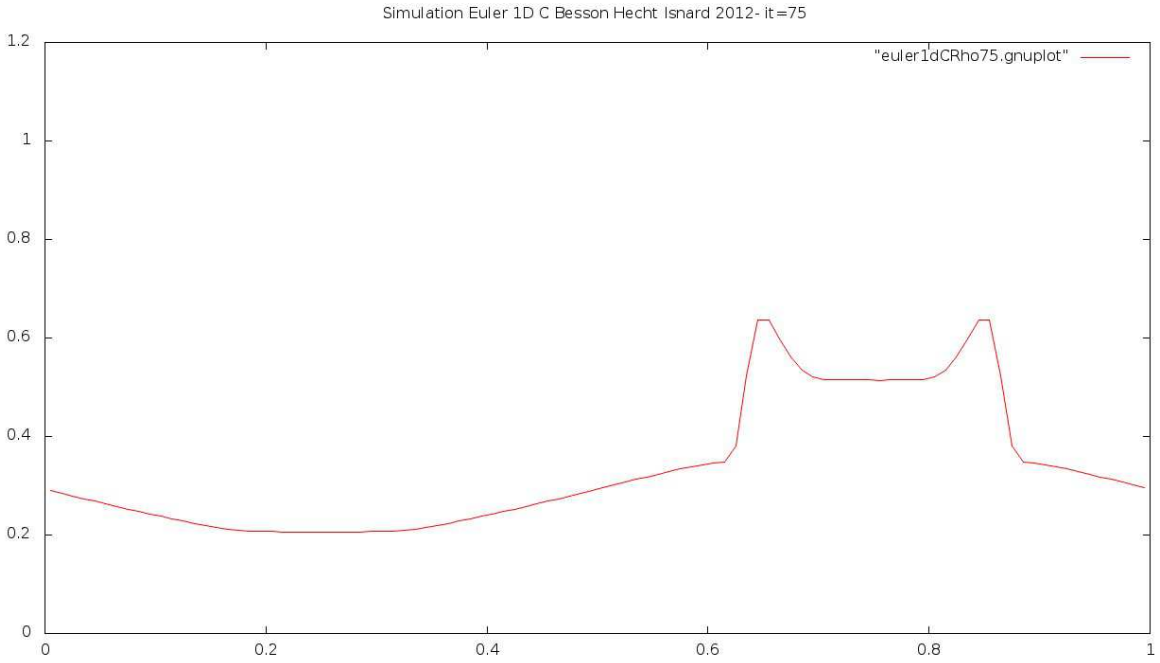


FIGURE 5 – Affichage en 1D de ρ pour Euler compressible (GNUPlot). `it = 75`

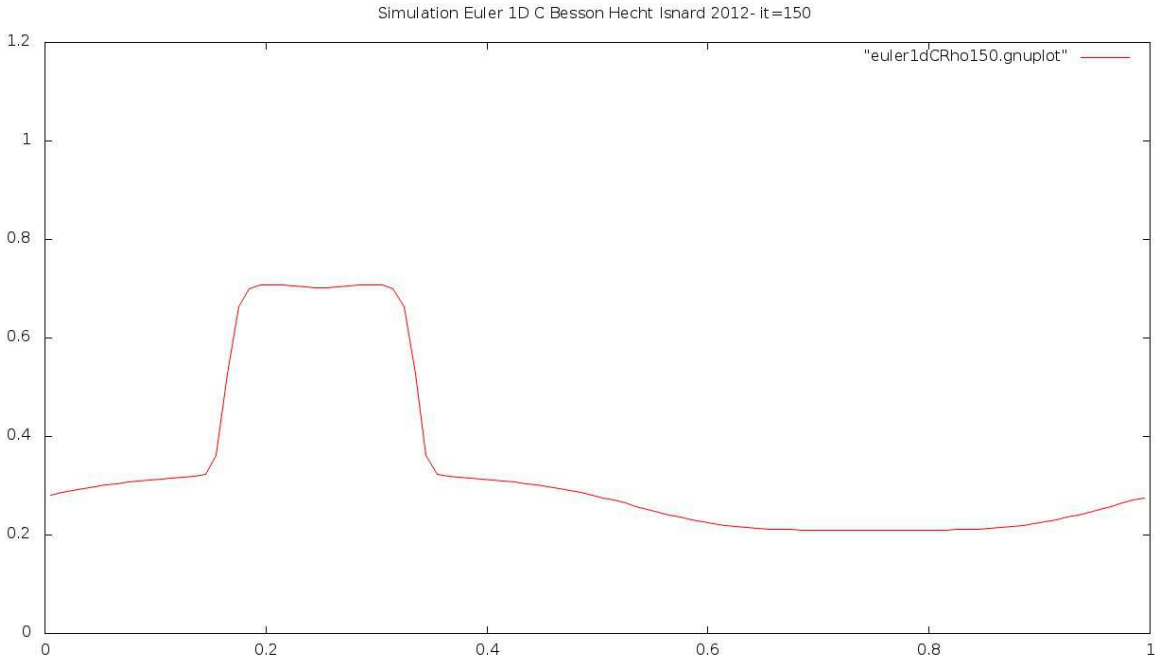


FIGURE 6 – Affichage en 1D de ρ pour Euler compressible (GNUPlot). `it = 150`

2.2 Détails supplémentaires sur le schéma numérique

Cette première simulation illustre donc le comportement de ces équations d'Euler.

Nous allons donner ici des détails supplémentaires à propos du schéma numérique utilisé dans nos simulations.

Tout d'abord, il faut préciser comment sont calculés les flux numériques, ce qui nécessite l'introduction de la notion de matrice signe, puis nous présenterons les résultats d'une démonstration, assistée par ordinateur, qui donne les valeurs propres et les vecteurs propres de cette matrice signe. Enfin, nous détaillerons l'étude de stabilité du schéma, en présentant une condition CFL (voir 2.2.2 page 23), généralisant la notion déjà évoquée dans les préliminaires pour l'équation d'advection.

2.2.1 Calcul des flux numériques : matrice jacobienne A, matrice sign(A) et diagonalisation

On rappelle qu'on a posé $A^{\overrightarrow{n}}$ la matrice jacobienne des flux $f_{\overrightarrow{n}}(U)$ selon la normale $\overrightarrow{n}$. L'expression des flux numériques $\phi^{\overrightarrow{n}}_{(i,j,k)+\frac{\overrightarrow{n}}{2}}$ fait intervenir une matrice notée plus haut $sign(A)$.

sign(A) et diagonalisation de la matrice A Voici un rappel de la définition de la fonction *signe* usuelle en mathématiques :

Définition 2 (Fonction signe)**.** $sign(x) := 1$ si $x > 0$, $:= -1$ si $x < 0$ et $:= 0$ si $x = 0$.

La notion de matrice signe en résulte alors :

Définition 3 (Matrice signe)**.** $sign(A)$ est la matrice semblable à A et dont les valeurs propres sont les signes des valeurs propres de A ;
ie : $\ker sign(A) = \{sign(\lambda_1); \ldots; sign(\lambda_p)\}$, où $\ker A = \{\lambda_1; \ldots; \lambda_p\}$.

Ainsi, on remarque déjà que le calcul de la matrice signe demande à priori de connaître les valeurs propres et une base de diagonalisation de la matrice d'origine.

Mais cela suppose que cette matrice A admette une telle base. Pour notre schéma, on a le théorème suivant qui va permettre le calcul théorique de cette matrice signe.

Théorème 1 (Calcul de $A^{\overrightarrow{n}}$ et $sign(A^{\overrightarrow{n}})$)**.** *La matrice $A^{\overrightarrow{n}}$ est diagonalisable.*

On en déduit que $sign(A)$ est aussi diagonalisable. On peut alors écrire : $A^{\overrightarrow{n}} = L^{\overrightarrow{n}}.\Lambda^{\overrightarrow{n}}.R^{\overrightarrow{n}}$, et $sign(A^{\overrightarrow{n}}) = L^{\overrightarrow{n}}.sign(\Lambda^{\overrightarrow{n}}).R^{\overrightarrow{n}}$. Le papier [GhiPa] nous donne les valeurs des matrices de vecteurs propres à gauche L , à droite R et des valeurs propres Λ.
Le calcul "à la main" n'est pas faisable. En effet, même en dimension 1, où nos matrices sont 3×3, le calcul de $\chi_A(X)$ [16] est déjà non trivial, et le calcul à la main de ses racines n'est pas faisable.

Valeurs des matrices A et Λ Nous avons retrouvé les valeurs formelles de ces matrices par une feuille de calcul formelle via `Maple` [17].

16. polynôme caractéristique de A.
17. Son contenu est présenté en annexe, et elle est disponible en ligne à l'adresse suivante : `http://www.dptinfo.ens-cachan.fr/~lbesson/publis/demo_A_At.mw`.

En pratique, nos discrétisations sont toujours cartésiennes, donc la normale $\overrightarrow{n}$ vaut l'un des trois vecteurs de la base orthonormale directe $\overrightarrow{e_x}$, $\overrightarrow{e_y}$, $\overrightarrow{e_z}$, mais il est plus simple de présenter les formules générales.

Nous utilisons ici les notations suivantes :

$$k := \gamma - 1;$$

$$H := e + \frac{1}{2}W^2 + \frac{p}{\rho};$$

$$K := c^2 + k(W^2 - H).$$

Il convient de préciser que H tel qu'il est définit ici représente l'enthalpie massique[18] comme la considèrent les physiciens.

Notre matrice $A^{\overrightarrow{n}}$ vaut :

$$A^{\overrightarrow{n}} = \begin{bmatrix} 0 & \overrightarrow{n} & 0 \\ K.\overrightarrow{n} - (\overrightarrow{W}.\overrightarrow{n})\overrightarrow{n} & \overrightarrow{W} \otimes \overrightarrow{n} - k\overrightarrow{n} \otimes \overrightarrow{W} + \overrightarrow{u}.\overrightarrow{n}\overline{\overline{Id}} & k\overrightarrow{w} \\ (K-H)\overrightarrow{W}.\overrightarrow{n} & H\overrightarrow{n} - k(\overrightarrow{W}.\overrightarrow{n})\overrightarrow{W} & (1+k)\overrightarrow{W}.\overrightarrow{n} \end{bmatrix} \tag{28}$$

On utilise ici la notation condensée à l'aide de produit tensoriel (le symbole $\otimes$).

Définition 4 (Produit tensoriel)**.** En termes de composantes :

$$(\underline{A} \otimes \underline{B})^i{}_j = C^i{}_j \tag{29}$$

$$= A^i B_j \tag{30}$$

On remarque que l'on peut exprimer ce produit tensoriel par un produit matriciel :

$$\underline{A} \otimes \underline{B} = \begin{bmatrix} A^1 \\ . \\ . \\ A^p \end{bmatrix} . \begin{bmatrix} B_1 & .. & B_p \end{bmatrix}$$

Nous donnons aussi le vecteur de valeurs propres $\Lambda^{\overrightarrow{n}}$ selon l'interface de normale $\overrightarrow{n}$ qui vaut :

$$\Lambda^{\overrightarrow{n}} = \mathbf{diag}[\overrightarrow{W}.\overrightarrow{n} - c; \overrightarrow{W}.\overrightarrow{n}; \overrightarrow{W}.\overrightarrow{n}; \overrightarrow{W}.\overrightarrow{n}; \overrightarrow{W}.\overrightarrow{n} + c]. \tag{31}$$

En dimension 1 on a $\Lambda^{\overrightarrow{n}} = \mathbf{diag}[\overrightarrow{W}.\overrightarrow{n} - c; \overrightarrow{W}.\overrightarrow{n}; \overrightarrow{W}.\overrightarrow{n} + c]$; et en dimension 2, $\Lambda^{\overrightarrow{n}} = \mathbf{diag}[\overrightarrow{W}.\overrightarrow{n} - c; \overrightarrow{W}.\overrightarrow{n}; \overrightarrow{W}.\overrightarrow{n}; \overrightarrow{W}.\overrightarrow{n} + c]$.

Valeurs des matrices L et R On rappelle que L désigne la transposée de la matrice des vecteurs propres à gauche de A et que R désigne la matrice des vecteurs propres à droite de notre matrice A.[19]

Pour les formules suivantes, on pose $\overrightarrow{\Omega_1}$ et $\overrightarrow{\Omega_2}$ une base orthogonale directe de l'hyperplan de normale $\overrightarrow{n}$.

18. les autres quantités physiques, e pour l'énergie interne, E pour l'énergie totale, ρ pour la masse volumique, sont aussi massiques.

19. L et R sont donc aussi matrice des vecteurs propres à gauche et à droite de $\mathbf{sign}(A)$, voir la remarque en 2.2.1.

Exemple 2 ($\overrightarrow{\Omega_1}$ et $\overrightarrow{\Omega_2}$)**.** En 3 dimensions, lors du *splitting* selon $\overrightarrow{e_x} = [1;0;0]$, on a $\overrightarrow{\Omega_1} = [0;1;0] = \overrightarrow{e_y}$; et $\overrightarrow{\Omega_2} = [0;0;1] = \overrightarrow{e_z}$.

À noter qu'en dimension 1, l'écriture des matrices précédentes se fait sans faire intervenir $\overrightarrow{\Omega_1}$ ni $\overrightarrow{\Omega_2}$; et qu'en dimension 2, seul $\overrightarrow{\Omega_1}$ intervient.

Notre matrice $R^{\overrightarrow{n}}$ (selon la normale $\overrightarrow{n}$) vaut

$$R^{\overrightarrow{n}} = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 \\ \overrightarrow{W} - c\overrightarrow{n} & \overrightarrow{W} & \overrightarrow{\Omega_1} & \overrightarrow{\Omega_2} & \overrightarrow{W} + c.\overrightarrow{n} \\ H - (\overrightarrow{W}.\overrightarrow{n})c & H - \frac{c^2}{k} & \overrightarrow{W}.\overrightarrow{\Omega_1} & \overrightarrow{W}.\overrightarrow{\Omega_2} & H + (\overrightarrow{W}.\overrightarrow{n})c \end{bmatrix} \tag{32}$$

Notre matrice $L^{\overrightarrow{n}}$ (selon la normale $\overrightarrow{n}$) vaut

$$L^{\overrightarrow{n}} = \begin{bmatrix} \frac{1}{2c^2}(K + \overrightarrow{W}.\overrightarrow{n}\,c) & \frac{k}{c^2}(H - W^2) & -\overrightarrow{W}.\overrightarrow{\Omega_1} & -\overrightarrow{W}.\overrightarrow{\Omega_2} & \frac{1}{2c^2}(K - \overrightarrow{W}.\overrightarrow{n}\,c) \\ \frac{1}{2c^2}(-k\overrightarrow{W} - \overrightarrow{n}\,c) & \frac{k}{c^2}\overrightarrow{W} & \overrightarrow{\Omega_1} & \overrightarrow{\Omega_2} & \frac{1}{2c^2}(-k\overrightarrow{W} + \overrightarrow{n}\,c) \\ \frac{1}{2c^2}.k & -\frac{k}{c^2} & 0 & 0 & \frac{1}{2c^2}.k \end{bmatrix} \tag{33}$$

En pratique dans nos codes pour Euler, nous calculons **sign**(A) à partir de ces valeurs formelles de Λ, R et L, en calculant explicitement chacune de ces trois matrices, et en faisant le calcul matriciel (deux multiplications matrice matrice, une multiplication matrice vecteur) demandé par l'équation 27.

Précisons ici que l'effort demandé pour calculer théoriquement les valeurs de ces matrices nous semblait être récompensé par une amélioration algorithmique de notre algorithme de résolution : en effet le produit matriciel est moins couteux que l'inversion matricielle. Mais en fait, nos matrices A, R, et L sont de taille $2 + d$ [20], donc la complexité algorithmique de ces procédures n'intervient pas !

Nous précisons cela plus en détail plus loin, mais il convient d'avoir en tête que les opérations les plus couteuses dans le calcul ne sont pas ces opérations matricielles, mais l'écriture dans la mémoire ROM de l'ordinateur pour l'export des données, ou le traitement "temps réel" des données pour l'affichage.

2.2.2 Stabilité du schéma

Comme pour le schéma présenté en 1.3.2, un des objectifs est de s'assurer la stabilité du schéma considéré.

Si la condition de stabilité du schéma 1D pour l'équation d'advection découle presque directement de l'écriture de u_i^{n+1} en fonction de u_i^n, $ubar_{i-1}^n$ et $ubar_i^n$; l'équivalent de cette condition **CFL** [21] pour le schéma 3D pour les équations d'Euler n'est pas aussi facile à obtenir.

Il faut préciser ici que le schéma est explicite, que la résolution procède par un splitting par direction (*ie* un en 1D, deux en 2D et trois en 3D), et que le pas de temps est le même pour chaque étape du splitting directionnel, mais varie à chaque étape temporelle.

Calcul du pas de temps Δt_n On ne détaille pas la démonstration de la relation à laquelle on arrive, mais comme pour l'équation d'advection on arrive à la condition nécessaire et suffisante de stabilité suivante (voir [DeDu] chapitre 2, section 3 pour une idée de la démonstration).

20. ici d désigne le nombre de dimension
21. voir 1.1.4.

Théorème 2 (Condition **CFL**). *Notre schéma numérique est stable **ssi** on a :*

$$\forall n, \Delta t_n = CFL.\frac{\min(\Delta x, \Delta y, \Delta z)}{c^n_{i,j,k} + \max\limits_{i,j,k} |W^n_{i,j,k}|} \tag{34}$$

où CFL est le nombre de Courant-Friedrich-Lévy, tel que $0 \leq CFL \leq 1$ (comme pour l'advection en 1.1.4).

Valeur du nombre CFL En pratique, il vaut mieux prendre $CFL = 0.5$ ou $CFL = 0.6$ plutôt que d'espérer être stable pour $CFL = 0.99$. Pour nos simulations, nous commençons en général par développer avec une valeur faible (de l'ordre de $CFL \simeq 0.05$) et dès que la résolution semble stable, on augmente progressivement cette valeur.

D'après les conseils de Florian De Vuyst, nous sommes satisfaits de notre simulation quand elle est stable pour une valeur 0.95, et quand elle est clairement instable pour des valeurs supérieures à 1.05. Pour chacune de nos réalisations, ces deux "*conditions informelles*" sont vérifiées pour chaque cas tests.

2.2.3 Autres propriétés

Le schéma présenté plus haut (2.1.3) est un schéma **explicite**.

De plus, il est dit **"à deux points"** puisque chaque splitting fait intervenir seulement les valeurs en i et $i-1$ pour calculer la nouvelle valeur $\overrightarrow{U}_i^{n+1}$.

Par ailleurs, c'est un **schéma d'ordre 1 en temps** (on approche la dérivée temporelle par un développement limité d'ordre 1) et **en espace** (idem).

Le schéma est stable dès que le pas de temps vérifie la condition 2.2.2. Et en temps que schéma explicite d'ordre 1, il est **consistant** (voir [DeDu] chapitre 2).

Ainsi, le théorème de Lax permet d'assurer la **convergence** du schéma.

Et enfin, sous l'hypothèse que la solution exacte $\overrightarrow{U}_{\mathbf{exacte}}$ au problème de Cauchy soit de classe C^1, on peut montrer le résultat suivant :

Proposition 1 (Erreur numérique). *Si on pose $\epsilon_{\Delta t,\Delta \mathbf{espace}}(t^n, K^{i,j,k}) = \overrightarrow{U}_{\mathbf{exacte}}(t^n, K^{i,j,k}) - \overrightarrow{U}_{\mathbf{num}}(t^n, K^{i,j,k})$ l'erreur numérique, alors*

$$\forall n > 0, \forall i, j, k \in \Omega,\ \ \epsilon_{\Delta t,\Delta \mathbf{espace}}(t^n, K^{i,j,k}) = O(\Delta t) + O(\Delta x) + O(\Delta y) + O(\Delta z) \tag{35}$$

Juste une idée. On peut raisonner grossièrement par récurrence sur l'évolution en temps (*ie* selon n).

- $n = 0$ Comme on impose les conditions initiales numériques égales à celle de la solution exacte, la propriété 1 est vérifiée trivialement ;
- $n > 0$ On écrit la nouvelle valeur comme fonction des anciennes. Par hypothèse de récurrence, on écrit chaque valeur de $\overrightarrow{U}_{\mathbf{num}}$ comme $\overrightarrow{U}_{\mathbf{exacte}} + \epsilon$.
  Ensuite, chaque valeur de $\overrightarrow{U}_{\mathbf{exacte}}$ qui n'est pas en i, j, k est à une case de i, j, k donc comme la solution du problème de Cauchy est supposée de classe C^1, on utilise la relation de Taylor à l'ordre 1, pour approcher ces valeurs par celle en i, j, k. Les erreurs d'approximation sont donc toutes des $O(\Delta x)$, $O(\Delta y)$ ou $O(\Delta z)$, qui se somment.
  Ensuite, on fait la même approximation pour arriver aux valeurs en i, j, k et t^{n+1}. Il sort une erreur en $O(\Delta t)$.

Et enfin, la condition CFL 2.2.2, permet de minorer l'erreur en t^{n+1} par la somme de l'erreur précédente en t^n et des sommes des O spatiaux (Δx, Δy, Δz) et temporel (Δt). Donc, par récurrence, cette erreur est bien en O de la somme des pas de discrétisation. □

Bref, notre schéma a donc toutes les bonnes propriétés [22] que l'on devait assurer (voir 0.1).

2.3 Implémentations et remarques

2.3.1 En séquentiel

Nous avons débuté l'implémentation du schéma résolvant l'équation d'Euler compressible en C, d'abord en 1D, début mai.

Nous préciserons ici les quelques points particuliers de nos codes séquentiels, et les difficultés rencontrées dans sa conception.

Remarques Nous détaillons quelques points de notre solveur C.

Valeurs aux interfaces Dans notre code, nous désignons par u**bar**x les valeurs aux interfaces de la variable u lors du splitting en x [23]. Plutôt que d'astucer la mémoire utilisée, nous préférons avoir trois vecteurs pour chaque dimension et chaque variable. Ce n'est pas économique, mais ça à l'avantage de rendre plus lisible les différentes étapes.

Structures de données Nous utilisons des structures de données très simplistes. Chaque vecteur est représenté par un `float *`. Ainsi, $\overrightarrow{U}$ est un vecteur de nombres flottants, qui contient $2+d$ informations en chaque point.

Le passage de code suivant illustre les deux points précédents [24].

```
printf(" Declarations variables en cours ... \n");
        float   *u;     // vitesse SELON X
        float   *ubarx; // valeurs de u aux interfaces selon x
// ... //
printf(" Allocation memoire C en cours ... ");
        U       = (float*) malloc(5*N*M*O * sizeof(float));     // d = 3 => 2+d
            = 5
        u       = (float*) malloc(N*M*O * sizeof(float));
        ubarx   = (float*) malloc((N + 1)*M*O * sizeof(float)); // un point en
            plus sur x
// ... //
// Utilisation d'une composante de U : actualisation de W
        rho[i*N*O + j*O + l ]    = U[i*N*O + j*O + l];    // rho = U_0
        u[i*N*O + j*O + l ]      = U[1*N*M*O + i*N*O + j*O + l]/U[0*N*M*O + i*N*O
            + j*O + l];     // u = U_1 / U_0
```

Ainsi, pour chaque vecteur et chaque appel, on doit calculer nous même l'indice à utiliser (par exemple $i*N*O+j*O+l$ pour accéder à la valeur de la case i,j,k d'un vecteur de taille $N*M*O$ d'une grille cartésienne ; ou $1*N*M*O+i*N*O+j*O+l$ pour la seconde case du

22. En tout cas, il a les propriétés que la littérature et nos responsables nous ont indiqué comme importantes.
23. et pareil selon y et z
24. On le montre en 3D directement, afin de bien montrer les différences entre u et u**bar**x

vecteur U dans cette case). Ce choix à deux raisons : il permet d'abord un meilleur contrôle de la mémoire, et une implémentation simpliste qui ne posera aucun problème lors du passage en CUDA. Et par ailleurs, d'après Florian De Vuyst, c'est la stratégie la plus efficace. [25] Pour plus de détails sur notre code, le lecteur intéressé peut trouver une archive regroupant nos codes ici : `http://www.dptinfo.ens-cachan.fr/~lbesson/publis/code_stage.zip`. [26]

Difficultés rencontrées La réalisation de notre premier solveur 1D nous a occupé plusieurs séances, presque tout le mois de mai.

Calcul matriciel Comme il est indiqué plus haut (voir 2.1.3), le calcul des flux numériques fait intervenir la matrice $sign(A)$.

Avant que nous n'ayons les valeurs théoriques de L nous avions cherché à inverser R. Dans l'espoir de gagner du temps (ce fut une mauvaise idée !) nous avons donc cherché une solution déjà disponible de calcul matriciel en C.

Sophie a travaillé avec succès mais beaucoup de difficulté avec CLapack [27]. Maxime et Lilian ont essayé d'abord avec une librairie trouvée sur `http://c.developpez.com/` (malheureusement, la source s'est perdue dans la bataille).

Il nous a fallu beaucoup de temps et de débuggage pour se rendre compte d'un problème dans l'inversion matricielle. C'est suite à cet échec et à cette perte de temps que Daniel Chauveheid nous a conseillé de chercher à pré-calculer L. Il nous a ensuite aidé à réaliser la démonstration informatique via Maple de ces valeurs.

La connaissance théorique des valeurs de L en chaque point permet de faciliter l'implémentation. Nous avons ensuite implémenté les deux fonctions de calcul matriciel dont nous avions besoin : produit matrice matrice, et produit matrice vecteur.

Difficultés générales Le schéma pour Euler est significativement plus complexe que celui pour l'advection. Beaucoup plus de variables, plus d'étapes, un schéma vectoriel dès la dimension 1 : beaucoup de ces raisons font que le développement n'a pas été facile.

2.3.2 En parallèle (CUDA)

Le développement en CUDA n'a pas été de tout repos. Et d'un certain point de vue, on peut même dire qu'il n'est pas tout à fait terminé. En tout cas, il nous a manqué un peu de temps pour affiner le code CUDA au niveau auquel nous avons détaillé et amélioré le code C.

Organisation des données En pratique, nos codes parallèles ont tous été produit comme des *passage en CUDA* de nos codes C. Ansi, on s'est "contentés" de convertir du code en CUDA, et pas de partir de rien.

En général, nos schémas utilisent beaucoup de tableaux. En CUDA, chaque tableau est localisé dans la mémoire GPU. Le corps de base de nos simulations est effectué par l'hôte : déclarations des variables, allocations de mémoires, valeurs des paramètres, puis initialisation des vecteurs, et copie vers la mémoire GPU.

25. Nous n'avons pas réalisé de tels tests nous mêmes.
26. Ces codes sont diffusées sous licence GPL, merci de respecter ces conditions.
27. Voir `http://www.netlib.org/clapack/` pour plus de détails. Pour les courageux et les personnes déjà famillières avec Lapack

Ensuite, la boucle de temps commence, est chaque étape de chaque splitting est réalisé avec un **kernel** CUDA écrit à part. Et à la fin de chaque boucle en temps, l'actualisation de Δt et la visualisation par l'écriture des données dans des fichiers de la mémoire morte de l'ordinateur. Ainsi, les seules étapes que l'on parallélise sont : le calcul des valeurs aux interfaces (variables indicées **bar**), le calcul des flux centrés ($\frac{F_{i+1}+Fi}{2}$) et moyens ($\frac{F_{i+1}-Fi}{2}$) ; puis le calcul des matrices R, L et Λ, le calcul de ϕ_i et l'actualisation des variables conservatives qui en découle.

Choix des constantes de parallélisation Chaque portion de code destiné à être parallélisé doit être écrit dans un **kernel**, qui sera appelé par le code CPU.

Un appel à un tel **kernel** demande de renseigner deux valeurs pour la parallélisation : le nombre de grilles (`GRIDS`) et le nombre de threads (`THREADS`). En pratique, il s'agit de valeurs de type `dim3`, un vecteur d'entiers puissances de 2 de taille 3.

Mais la sorcellerie réside dans le choix des valeurs de ces paramètres. En effet, nous n'avions pas connu de difficulté dans l'implémentation du solveur pour l'advection. Mais dans ce schéma plus complexe, ce problème est un des plus importants.

Et en plus, ce choix, qui nécessite de beaucoup tester, dépend de la machine et de l'architecture de la carte vidéo.

Calcul matriciel Notre code Euler en C utilisait un pointeur unique pour chaque étape, et cet "astuce" nous a couté du temps sur le développement en CUDA.

2.4 La question des conditions limites

Jusqu'ici, nous avons présenté des simulations en 1D et en 2D qui partageaient toutes un point commun : leurs conditions aux limites. Nous présentons ici les différents choix pour ces conditions, en adoptant un point de vue uniquement "*numérique*" et sans retourner à l'écriture en tant que système différentiel.

En effet, dans les deux schémas explicites présentés plus haut (voir 1.3.2 et 2.1.3), le vecteur des variables conservatives $\overrightarrow{U}$ dépend des flux aux interfaces : à gauche $(i-1/2)$ et à droite $(i+1/2)$, en bas $(j-1/2)$ et en haut $(j+1/2)$, et en arrière $(k-1/2)$ et en avant $(k+1/2)$.

Mais, comment calculer les flux aux interfaces extrêmes ?
C'est ici qu'intervient la notion de condition aux limites. En effet, quand on calcul de le flux droit (*ie* $\overrightarrow{F}(\overrightarrow{U}^n_{i+1})$) à la cellule à l'extrémité droite de la grille cartésienne (*ie* $i = N-1$), l'équation fait intervenir la valeur de $\overrightarrow{U}^n_N$, qui n'est pas défini.

Il faut donc choisir une "méthode" pour calculer ces $2 * d$ valeurs non définies.

Définition 5 (Condition aux limites)**.** On appelle condition aux limites une **méthode** pour calculer les valeurs de $\overrightarrow{U}^n_{i,j,k}$ pour les i, j, k pour lesquels elles ne sont pas définies.

De plus, une telle stratégie doit être cohérente : le schéma doit restr stable et convergent. [28]

Ainsi, on présente ici les *différents types de conditions aux limites !*

28. Ce qui n'est évidemment pas le cas pour n'importe quelle stratégie. Imposer une valeur absurde pour ces flux ne fonctionne pas !

2.4.1 Conditions limites périodiques

Aussi appelées conditions nulles de Neumann. En 1D, cela peut modéliser un tube circulaire. En 2D, cela peut modéliser l'atmosphère par exemple (si on néglige les variations selon l'altitude, l'atmosphère est une nappe en 2D "repliée sur elle même" dans le sens où elle n'a pas de bords). En dimensions supérieures, rien de très physique n'est modélisé. [29]

On choisit $\overrightarrow{F}(\overrightarrow{U}^n_N) = \overrightarrow{F}(\overrightarrow{U}^n_0)$; et $\overrightarrow{F}(\overrightarrow{U}^n_{-1}) = \overrightarrow{F}(\overrightarrow{U}^n_{N-1})$. Idem pour j,M selon y et k,O selon z.

2.4.2 Conditions limites absorbantes

Modélise un tube (en 1D), une surface de lac (en 2D) ou une piscine (en 3D) de taille *infinie*, lorsque les conditions initiales sont suffisamment éloignées des bords. On choisit $\overrightarrow{F}(\overrightarrow{U}^n_N) = \overrightarrow{F}(\overrightarrow{U}^n_{N-1})$; et $\overrightarrow{F}(\overrightarrow{U}^n_{-1}) = \overrightarrow{F}(\overrightarrow{U}^n_0)$; Idem pour j,M selon y et k,O selon z.

2.4.3 Conditions limites de mur

Modélise un tube (en 1D), une surface de lac (en 2D) ou une piscine (en 3D) avec des bords sur lesquelles se réfléchissent les ondes étudiées. Pour calculer les flux aux interfaces extrêmes, on suppose $\overrightarrow{W}.\overrightarrow{n} = 0$.

On voit apparaître une simplification dans le calcul des flux numériques $\overrightarrow{f}_{\overrightarrow{n}}(\overrightarrow{U}^n_{i,j,k})$. En effet, on rappelle que $\overrightarrow{f}_{\overrightarrow{n}}(\overrightarrow{U}^n_{i,j,k}) = [\rho\overrightarrow{W}.\overrightarrow{n}; \rho\overrightarrow{W}.\overrightarrow{n}\overrightarrow{W} + p.\overrightarrow{n}; (\rho E + p)\overrightarrow{W}.\overrightarrow{n}]$.

Ainsi, sous cette hypothèse de mur, pour les calculs aux interfaces extrêmes, on a : $\overrightarrow{f}_{\overrightarrow{n}}(\overrightarrow{U}^n_{extreme}) = [0; +p_{extreme}.\overrightarrow{n}; 0]$. Pour expliciter la valeur de cette $p_{extreme}$, plusieurs stratégies nous ont été présentées (voir [GhiPa]) : la stratégie dite *du pauvre*, celle dite du *pas pauvre*, et une autre plus explicite donnée par Florian de Vuyst [30].

Stratégie du "pauvre" Cette première stratégie est la plus simple.

Il suffit de dire que $p_{extreme}$ vaut la valeur de p dans la cellule intérieure.

Exemple 3 (Selon x)**.** Dans la direction horizontale, $p_{extreme} = p_{N,j,k}$ [31]. Ce qu'on appelle cellule intérieure signifie ici la cellule $K_{N-1,j,k}$.

Donc on utilise $p_{extreme} = p_{N-1,j,k}$.

En pratique, cette méthode fonctionne bien. Voici un extrait du passage du code Euler 2D en CUDA, lors splitting horizontal.

```
1 // condition limites de mur strategie pauvre
2 pression_speciale = p[i*N + 0]; // cellule extreme a gauche
3
4         Fc[0*(N + 1)*M + i*(N + 1) + 0] = 0.5*(F[0*N*M + i*N + 0] + 0.0);
5         Fc[1*(N + 1)*M + i*(N + 1) + 0] = 0.5*(F[1*N*M + i*N + 0] +
              pression_speciale);
6         Fc[2*(N + 1)*M + i*(N + 1) + 0] = 0.5*(F[2*N*M + i*N + 0] + 0.0);
7         Fc[3*(N + 1)*M + i*(N + 1) + 0] = 0.5*(F[3*N*M + i*N + 0] + 0.0);
8         deltaF[0*(N + 1)*M + i*(N + 1) + 0]      = F[0*N*M + i*N + 0] - 0.0;
```

29. Même en cherchant un peu, notre imagination peine à trouver un sens physique à de telles conditions de bords en 3D. C'est pour avoir des simulations 3D plus "cohérentes" que nous avons cherché d'autres conditions.

30. Par manque de temps, nous n'avons pas essayé d'implémenter cette dernière stratégie.

31. En effet, quand $i = N - 1$ et pour calculer $p_{i+1,j,k}$.

```
9           deltaF[1*(N + 1)*M + i*(N + 1) + 0]      = F[1*N*M + i*N + 0] -
                pression_speciale;
10          deltaF[2*(N + 1)*M + i*(N + 1) + 0]      = F[2*N*M + i*N + 0] - 0.0;
11          deltaF[3*(N + 1)*M + i*(N + 1) + 0]      = F[3*N*M + i*N + 0] - 0.0;
12  pression_speciale = p[i*N + (N-1)];;     // cellule extreme a droite
13          Fc[0*(N + 1)*M + i*(N + 1) + N] = 0.5*(F[0*N*M + i*N + (N-1)] + 0.0);
14          Fc[1*(N + 1)*M + i*(N + 1) + N] = 0.5*(F[1*N*M + i*N + (N-1)] +
                pression_speciale);
15          Fc[2*(N + 1)*M + i*(N + 1) + N] = 0.5*(F[2*N*M + i*N + (N-1)] + 0.0);
16          Fc[3*(N + 1)*M + i*(N + 1) + N] = 0.5*(F[3*N*M + i*N + (N-1)] + 0.0);
17          deltaF[0*(N + 1)*M + i*(N + 1) + N]      = - F[0*N*M + i*N + (N-1)] +
                0.0;
18          deltaF[1*(N + 1)*M + i*(N + 1) + N]      = - F[1*N*M + i*N + (N-1)] +
                pression_speciale;
19          deltaF[2*(N + 1)*M + i*(N + 1) + N]      = - F[2*N*M + i*N + (N-1)] +
                0.0;
20          deltaF[3*(N + 1)*M + i*(N + 1) + N]      = - F[3*N*M + i*N + (N-1)] +
                0.0;
```

Stratégie du "pas pauvre" Cette seconde stratégie est dérivée de la première (voir [GhiPa], page 58 (10.54)).

Elle consiste à rajouter un terme correctif pour tenir compte du sens de la vitesse du fluide dans la direction du splitting (*ie* de $\overrightarrow{W}.\overrightarrow{n}$).

Il suffit de dire que $p_{extreme}$ vaut la valeur de p dans la cellule intérieure, multipliée par le coefficient suivant :

$$p_{extreme} = p_{N-1,j,k} \times (1 + \frac{\gamma(\overrightarrow{W}_{N-1,j,k}.\overrightarrow{n})}{c_{N-1,j,k} - (\gamma-1)\overrightarrow{W}_{N-1,j,k}.\overrightarrow{n}}) \tag{36}$$

En pratique, nous avons implémenté cette stratégie dans nos codes C et CUDA, mais sans succès. Un exemple du résultat obtenu avec cette stratégie "pas pauvre" (à droite dans l'image), en comparaison avec la même simulation avec la stratégie "pauvre (à gauche) montre que cette seconde stratégie n'a pas fonctionnée. [32]

Stratégie explicite Cette troisième stratégie est plus générale.

Elle consiste à rajouter une cellule "miroir" à l'extrémité de chaque direction, où la valeur du vecteur des variables conservatives $\overrightarrow{U}$ est prise "miroir" de celle de la cellule intérieure.

C'est-à-dire, lors du splitting en x, $\overrightarrow{U}_{N,j,k}$ vaut $[\rho; -\rho u; \rho v; \rho w; \rho E]$.

D'une manière générale, lors du calcul des flux numériques selon $\overrightarrow{n}$, $\overrightarrow{U}_{extreme} = [\rho; -(\overrightarrow{n}.\overrightarrow{W})\overrightarrow{n}; \rho E]$

En pratique, nous n'avons pas testé d'implémenter cette stratégie.

2.4.4 Exemple de simulation en 2D illustrant les trois conditions de bords

On présente ici trois simulations, faites en 2D avec notre solveur Euler en CUDA. Les conditions initiales sont identiques, circulaires.

32. [Lilian] L'erreur semble ne pas venir de notre code, qui a été corrigé et relu, et qui fonctionne parfaitement pour les autres conditions. Peut-être qu'une des conditions d'application de cette stratégie n'est pas vérifiée ici.

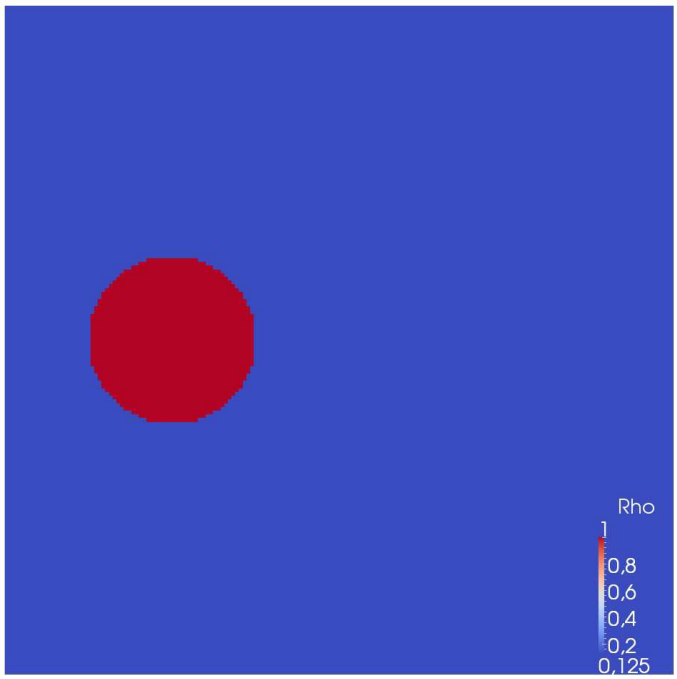


FIGURE 7 – Affichage en 2D pour Euler compressible (ParaView) : conditions initiales.

Remarque 1 (À propos des conditions initiales). Nous avons testé seulement $1 + d$ types de conditions initiales pour nos simulations.

Mono-dimensionnelle C'est-à-dire que ρ et p sont initialement posés constants par morceau selon une des d directions. Par exemple orientée selon x :

$$u_{i,j,k} = \begin{cases} 0.125 \ \textbf{si} \ x_j \leq 0.5; \\ 1 \ \textbf{sinon}. \end{cases} \tag{37}$$

Circulaire C'est-à-dire que ρ et p sont initialement posés constants sur un cercle centré ou non, de rayon petit devant 1 :

$$u_{i,j,k} = \begin{cases} 0.125 \ \textbf{si} \ x_j^2 + y_i^2 + z_l^2 \leq 0.125; \\ 1 \ \textbf{sinon}. \end{cases} \tag{38}$$

L'évolution des simulations est plus intéressante avec des conditions initiales circulaires [33]. Pour les trois simulations suivantes, nous utilisons le même type de conditions initiales, et le seul paramètre à varier est le type de conditions aux bords.

On présente d'abord avec des conditions périodiques ; ensuite avec des conditions absorbantes. Assez vite, la surface est calme ; enfin avec des conditions de mur.

33. Les conditions mono-dimensionnelles limitent la propagation à une seule dimension.

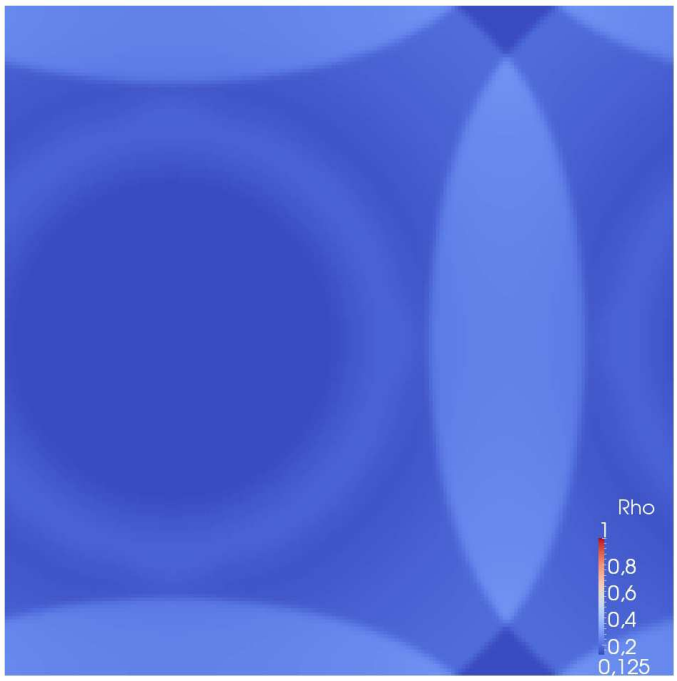


FIGURE 8 – Affichage en 2D pour Euler compressible (ParaView) : conditions périodiques. `it=125`

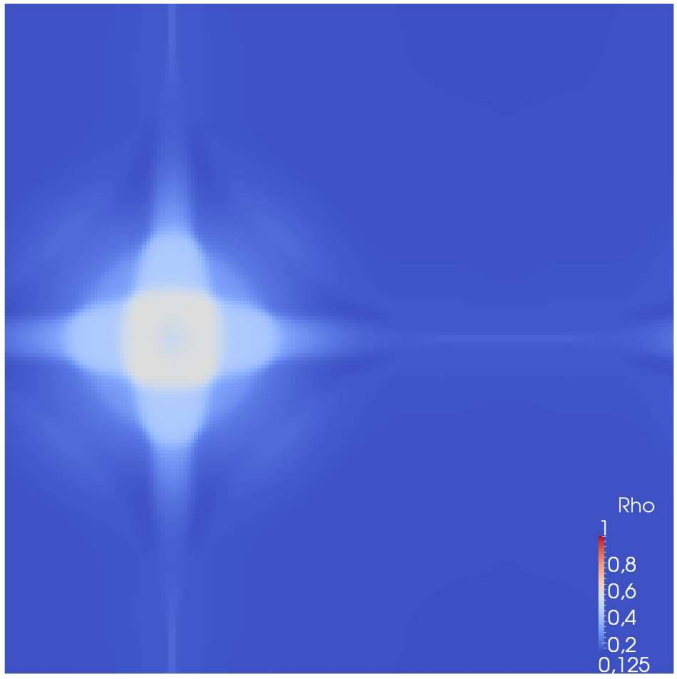


FIGURE 9 – Affichage en 2D pour Euler compressible (ParaView) : conditions périodiques. `it=250`

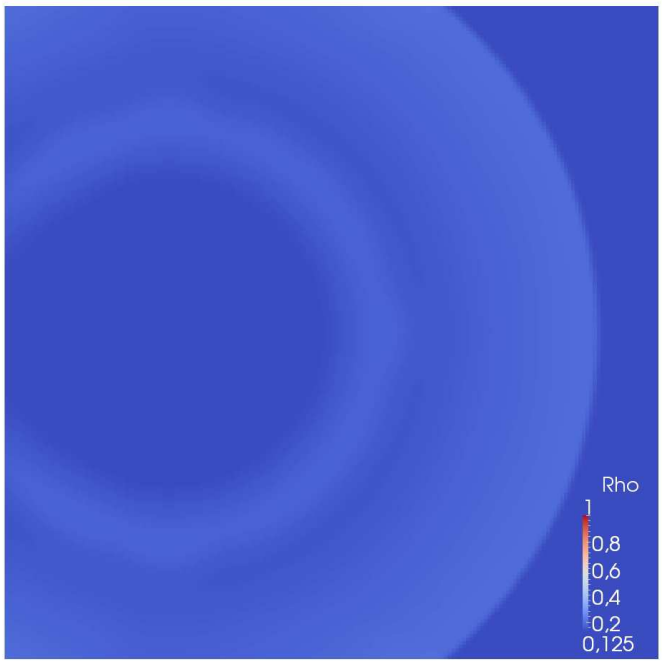


FIGURE 10 – Affichage en 2D pour Euler compressible (ParaView) : conditions absorbantes. `it=125`

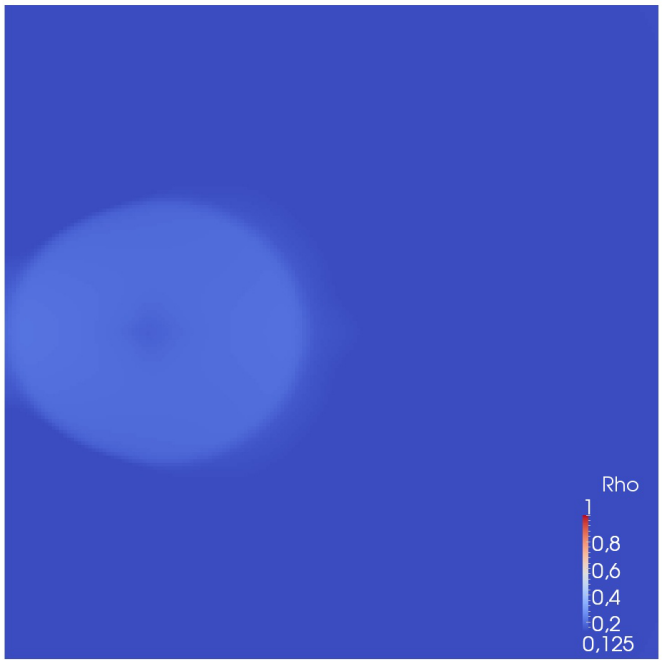


FIGURE 11 – Affichage en 2D pour Euler compressible (ParaView) : conditions absorbantes. `it=250`

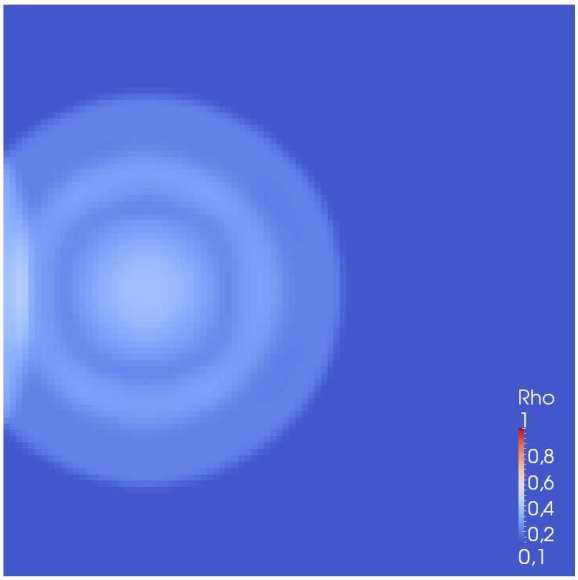


FIGURE 12 – Affichage en 2D pour Euler compressible (ParaView) : conditions de mur "pauvre". `it=25`

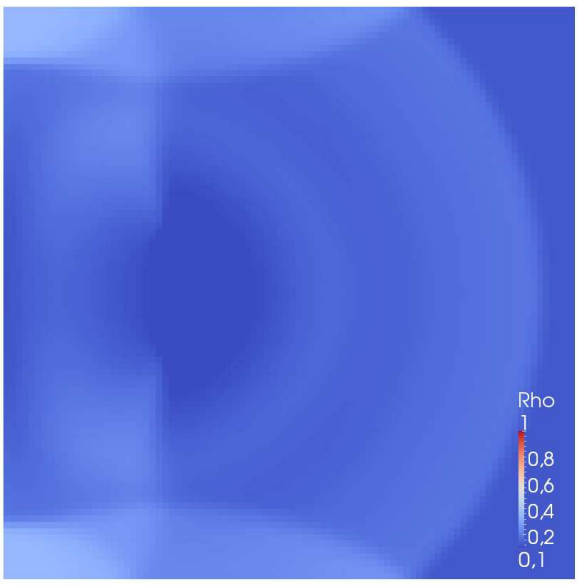


FIGURE 13 – Affichage en 2D pour Euler compressible (ParaView) : conditions de mur "pauvre". `it=70`

2.5 Rajout d'un terme source dans les équations physiques

2.5.1 Modification des équations

Dans les équations précédentes, on peut aussi ajouter un terme source. Pour rajouter un sens physique à la simulation, nous avons choisi de se placer en dimension 3, et nous avons introduit un terme source modélisant l'action de pesanteur.

En reprenant l'équation présentée plus haut, on rajoute un terme $S(U)$ non nul.

On choisit de modéliser l'action du champ de pensanteur, ainsi le terme source vaut

$$\overrightarrow{S}(\overrightarrow{U}) = \begin{bmatrix} 0 \\ \rho.\overrightarrow{g} \\ \rho.\overrightarrow{g}.\overrightarrow{W} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ -\rho g \\ -\rho g w \end{bmatrix}. \tag{39}$$

2.5.2 Actualisation du schéma

Le nouveau schéma est donc :

$$\frac{\overrightarrow{U}^{n+1}_{i,j,k} - \overrightarrow{U}^{n}_{i,j,k}}{\Delta t_n} + \frac{1}{\Delta x \Delta y \Delta z}[(\overrightarrow{\phi x}^n_{i+1/2}\Delta y \Delta z + \overrightarrow{\phi x}^n_{i-1/2}\Delta y \Delta z) +$$

$$(\overrightarrow{\phi y}^n_{j+1/2}\Delta x \Delta z + \overrightarrow{\phi y}^n_{j-1/2}\Delta x \Delta z) + (\overrightarrow{\phi z}^n_{k+1/2}\Delta x \Delta y + \overrightarrow{\phi z}^n_{k-1/2}\Delta x \Delta y)] = \overrightarrow{S}(\overrightarrow{U}^n_{i,j,k}).$$

pour $(i \in [0; N-1], j \in [0; M-1], k \in [0; O-1])$.
Les flux sont choisis selon la méthode VFFC : (ici selon x)

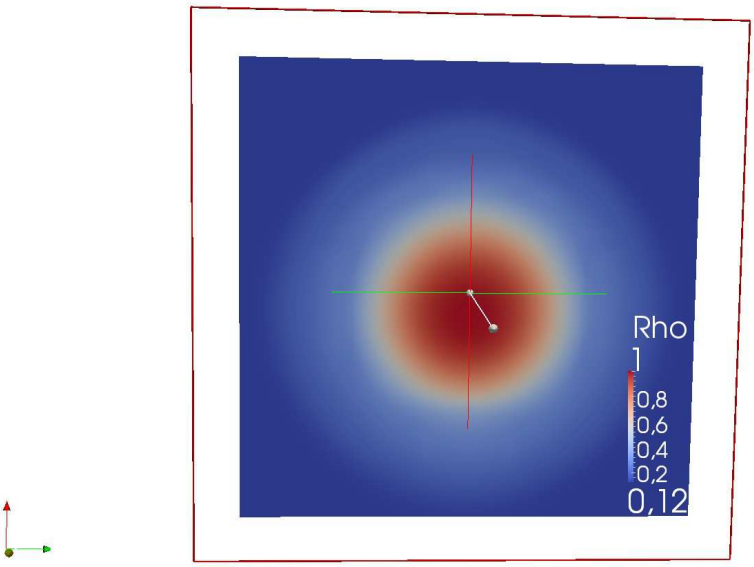
$$\overrightarrow{\phi x}^n_{i+1/2} = \frac{\overrightarrow{F}(\overrightarrow{U}^n_{i+1}) + \overrightarrow{F}(\overrightarrow{U}^n_i)}{2} - sign(A^n_{i+1/2})\frac{\overrightarrow{F}(\overrightarrow{U}^n_{i+1}) - \overrightarrow{F}(\overrightarrow{U}^n_i)}{2} \tag{40}$$

où on a posé $A^n_{i+1/2}$ la matrice jacobienne du vecteur $\overrightarrow{F}(\overrightarrow{U}^n_{i+1/2})$.

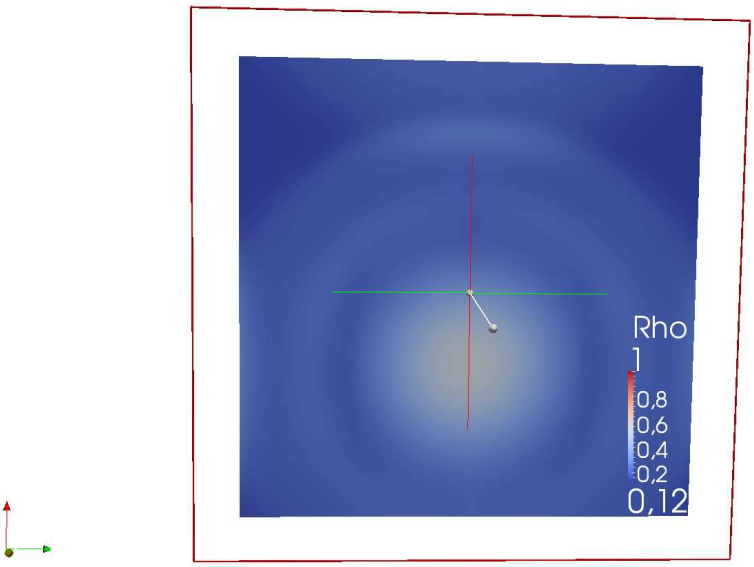
2.5.3 Détails et résultats

Finalisation du schéma La dernière étape est le splitting directionnel. Il faut rajouter le terme source selon l'une des directions. Nous avons choisi de le mettre selon l'axe des z.

$$\begin{cases} \dfrac{\overrightarrow{U}^{n^*}_{i,j,k} - \overrightarrow{U}^{n}_{i,j,k}}{\Delta t_n} + \dfrac{1}{\Delta x}[\phi x^n_{i+1/2,j,k} + \phi x^n_{i-1/2,j,k}] = 0; \\ \dfrac{\overrightarrow{U}^{n^1}_{i,j,k} - \overrightarrow{U}^{n^*}_{i,j,k}}{\Delta t_n} + \dfrac{1}{\Delta y}[\phi y^n_{i,j+1/2,k} + \phi y^n_{i,j-1/2,k}] = 0; \\ \dfrac{\overrightarrow{U}^{n+1}_{i,j,k} - \overrightarrow{U}^{n^1}_{i,j,k}}{\Delta t_n} + \dfrac{1}{\Delta z}[\phi z^n_{i,j,k+1/2} + \phi z^n_{i,j,k-1/2}] = \overrightarrow{S}(\overrightarrow{U}^n_{i,j,k}). \end{cases} \tag{41}$$



it=35



it=85

FIGURE 14 – Affichage en 3D pour Euler compressible (ParaView) : conditions périodiques, avec terme source.

Un résultat sympathique Un graphique pour Euler 3D avec terme source est difficile à réaliser, car montrer une vue "plate" d'une simulation 3D n'est pas très parlante. Nous avons préféré faire avec des coupes.

En pratique, on observe que la bulle de gaz centré initialement au milieu de la grille cartésienne est diffusée, comme il en est en 2D, et se trouve de plus déplacée verticalement et vers le bas.

Quelques tests ont permis de valider l'actualisation du schéma, notamment de valider l'influence de la *valeur* de ρ ou g qui se répercute directement sur l'efficacité de l'attraction modélisée par ce terme source supplémentaire.

3 Optimisation du code séquentiel et parallèle

A ce stade, nous avons déjà présenté les équations d'Euler, le schéma utilisé pour les résoudre, et les grandes lignes de nos deux solveurs C et CUDA.

Cette dernière partie traite tout d'abord d'une première comparaison effectuée avec notre solveur 2D pour Euler, ensuite des améliorations effectuées sur ces codes, principalement lors de l'étape de visualisation / écriture.

3.1 Première comparaison des performances

A partir du code Advection2D en C et en CUDA, nous avons voulu comparer les performances respectives de nos deux simulations.

Les paramètres de la comparaison sont d'abord 256×256 points, puis 512×512 points, ensuite 1024×1024 et enfin 2048×2048 points[34], points, et 1024 étapes en temps.

Codes	Temps mis (256x256)	(512x512)	(1024x1024)	(2048x2048)
C	43s	2m32s	10m41s	40m21s
CUDA	37s	39s	42s	45s

Bref, il est immédiat de constater que, comme prévu, l'affichage parallèle avec CUDA est le plus rapide. Nous précisons qu'ici n'est pas compté le temps de compilation[35], mais que l'initialisation et les écritures dans les fichiers .vtk sont comptées (une étape sur 20 seulement).

Définition 6 (Speed-up)**.** On désigne par **speed-up** le gain de performances obtenus entre un code séquentiel et un code parallèle.

Une mesure de speed-up n'a de sens que pour des nombres de points élevés, et pour des codes optimisés aux même niveau.

La dernière colonne du tableau satisfait donc ces deux conditions. Ainsi, cette première comparaison se conclut par un **speed-up de près de 43 !**

3.2 Optimiser la visualisation des données produites

3.2.1 GNUplot et VTK + ParaView

Pour l'instant nous avons utilisé 2 logiciels de visualisation : GNUplot et VTK+Paraview. L'avantage de GNUplot est qui est facilement utilisable et qu'il permet une visualisation en temps réel. Paraview quant à lui permet de faire une visualisation plus rapide mais post-mortem. Des explications supplémentaires sur ces deux outils ont déjà été données plus haut.

Il convient de préciser qu'ils nécessitent tous deux une étape que l'expérience a montré comme étant la **plus couteuse en temps de calcul** : l'écriture des valeurs du vecteur à afficher, une à une, dans un fichier de la mémoire morte de l'ordinateur.

En effet, de nombreuses simulations ont permis de montrer que jusqu'à 40% du temps pouvait être dépensé à cette phase de création du fichier, écriture, et fermeture du fichier. C'est tout de même embêtant non ?

34. Précisons ici que nos codes ne *sont pas* optimisés pour tourner plus vite avec un nombre de points valant une puissance de deux. C'est seulement plus pratique, et c'est une question d'habitude.

35. toujours plus court pour le CUDA, bien que cette observation soit illogique.

3.2.2 OpenGL

Présentation Nous nous sommes intéressé à un troisième moyen de visualisation : OpenGL (Open Graphics Library) . C'est une librairie qui permet un affichage en 3D en temps réel et qui est très utilisée pour les jeux vidéo. La majorité de ces derniers étant codés en parallèle, c'est un outil qui semblait compatible avec CUDA.

Il nous a donc semblé intéressant d'essayer d'appliquer cet affichage pour nos simulations. En effet, l'une des source de lenteur de nos programmes est la perte de temps au niveau de la transmission d'information entre le GPU et le CPU, et plus encore lors de l'écriture dans la mémoire morte du pc. OpenGL est donc sensé nous permettre de ne travailler que sur le GPU, dans la cas d'un portage réussi en CUDA, ou seulement en mémoire vive dans le cas d'une implémentation en C.

Premier pas sur le CPU La première étape a été de réussir à faire un affichage sur le code C du Euler 2D. Si l'on regarde le problème d'un point de vue général, il faut à chaque pas de temps créer un maillage carré de notre espace de base et donner une couleur à ce carré en fonction de la valeur de la fonction à afficher.

De plus amples explications sont données en 5.2.

Et sur le GPU Pour pouvoir faire l'affichage sur le GPU directement toutes les fonction ont été crées sur le device. Ce programme ne marche pas encore. Il faudrait quelques semaine de stage en plus pour réussir à le faire marcher...

3.2.3 Comparaison GNUplot vs OpenGL

A partir du code Euler 2D en C, trois versions ont été faites. L'une utilisant un affichage dynamique via GNUPlot, la seconde un affichage post-mortem en VTK et la dernière avec OpenGL ; dans le but de comparer les performances entre chaque simulation.

Les paramètres de la comparaison sont d'abord 256×256 points, puis 512×512 points, et 128 étapes en temps. L'initialisation est comptée, mais pas la compilation.

Code	Temps mis (512x512)	Temps mis (256x256)
GNUplot	4m59s	3m27s
VTK	43s	31s
OpenGL	29s	20s

Bref, il est immédiat de constater que, comme prévu, l'affichage interactif avec GNUplot est le plus lent, car il nécessite l'impression du triple de données, l'appel du sous processus, et le traitement des données. Ensuite, la mesure pour l'affichage via VTK est légèrement faussée, car ne contient que le temps pris par l'impression et no compte pas la visualisation post-calcul. En enfin, il est rassurant de constater que l'investissement en OpenGL aura apporté un vrai plus sur le plan des performances : plus de 30% de gagné !

3.3 Optimisation de notre solveur Euler 2D

Dans cette dernière partie, nous présentons les essais d'amélioration des performances de notre solveur volume finis en C et en CUDA pour Euler 2D. Nous évoquerons d'abord quelques conseils et astuces pour coder proprement et améliorer un peu l'efficacité de son code ; puis serons évoqués les difficultés rencontrées dans le développement et l'amélioration de nos codes

CUDA ; puis nous terminerons par une comparaison entre les performances des deux codes séquentiels et parallèles.

3.3.1 Coder proprement

Une des premières étapes du travail d'amélioration de notre code a été de le nétoyer un peu. Cela consiste à effacer toutes les lignes inutiles, et à réduire au minimum les explications, commentaires, et détails futiles.

Ensuite, simplifier chaque expression assez longue en utilisant quelques variables intermédiaires permet d'améliorer grandement la lisibilité du code. Par ailleurs, récalculer certaines constantes, ou utiliser des variables globales pour les constantes numériques ($\gamma, \mathbf{CFL}, g$ par exemple) et simplifier les expressions arithmétiques permet d'obtenir un code plus propre, donc plus fiable.

3.3.2 Essais

Nous avons successivement tenté de paralléliser les opérations suivantes :

Calcul matriciel Les fonctions de calculs matriciels sont appelées par le GPU et exécutées sur le GPU : la parallélisation par le *mapping* des indices n'est pas possible (ou en tout cas, nous n'avons pas abordé les techniques le permettant). De plus, on fait beaucoup de produits matrice vecteur et matrice matrice, mais la taille des produits est toujours faible (en fait, seulement $2 + d$) : la parallélisation n'est pas intéressante.

Δt Nous avons été confronté au problème de réduction : calculer en parallèle et efficacement le maximum d'un vecteur GPU. En pratique, nous ramenons les données vers le CPU et faisons le calcul classique.

– La boucle sur les composantes de U et W est de taille $2 + d$ aussi : la parallélisation n'est pas intéressante.

En pratique, une étape qui aurait été très intéressante et qui aurait pu bien améliorer les performances finales aurait été de supporter l'affichage en OpenGL de notre solveur CUDA en CUDA. Effectivement, bien que nous soyons très content des résultats obtenus, il nous aurait fallu quelques jours de plus pour permettre à cet affichage de manière parallèle. En effet, il s'agit juste d'une traduction en CUDA de l'environnement d'affichage en OpenGL fait pour le C : les calculs d'affichages ne sont pas parallélisé.

3.3.3 Difficultés

On notera ici seulement notre mauvais contact avec l'optimisation sous CUDA mais surtout avec les outils aui auraient permis de simplifier cette étape là : nous détaillons cet échec techologique en annexe.

3.3.4 Comparaison

Une comparaison pour un maillage 728×728, 1024 étapes et des conditions de mur.

Code	En écrivant en .vtk	Sans écrire en .vtk
C	46m28s	32m20s
CUDA	12m3s	5m19s

$\longrightarrow$ Un **speed-up** de près de 7 !

Nous avons aussi manqué de temps pour comparer les versions finales de nos solveur C et CUDA affichant via OpenGL, mais les conséquences de l'investissement sur l'OpenGL ont été très positive car les performances obtenues par ce dernier supplantent magistralement les performances initiales obtenues avec GNUplot.

Code	OpenGL	GNUPlot
C	rapide	lent
CUDA	très rapide	lent

En effet, les dernières mesures informelles réalisées montrent une nette supériorité de l'affichage interne (OpenGL) sur l'externe (GNUPlot) qui était prévisible ; ainsi qu'une amélioration d'un facteur 12 du code CUDA par rapport au séquentiel CPU.

4 Conclusion

4.1 Un petit bonus : une visualisation interactive pour Euler 2D en C avec OpenGL

L'un des autres avantages d'OpenGL est qu'il permet une interaction, c'est-à-dire que nous pouvons modifier des données pendant le calcul. Nous avons donc créé une fonction qui nous permet quand on clic à un endroit de notre carré, d'augmenter la pression sur un petit cercle autour de ce point. Nous avons choisi d'augmenter la pression car c'est le paramètre qui a le plus d'impact sur l'évolution du problème.

Les résultats obtenus ne sont pas facilement intégrables dans un rapport en version papier, mais normalement, un petit aperçu du résultat final a été donné lors de la soutenance du mardi 3 juillet.

Nous avons obtenus les derniers résultats les derniers jours de travail. Et finalement, l'affichage interne en OpenGL est performant, assez difficile à maîtriser, mais fonctionne bien pour nos codes C et CUDA en 2D.

Et l'intéracivité n'a pas été déployée pour le CUDA, mais ce détail n'est qu'une question de temps.

4.2 Retours d'expériences personnels

4.2.1 Sophie

Ce stage m'a permit d'apprendre beaucoup de choses, essentiellement au niveau de la programmation. Avant je n'avais que quelques notions en C, et maintenant je suis capable de programmer en C et en CUDA. Le calcul parallèle étant quelque chose de très nouveau il est intéressant d'avoir pu acquérir de bonnes bases dans ce domaine. J'ai aussi appris à me servir de plusieurs logiciels de visualisation, plus particulièrement d'OpenGL qui est assez difficile à prendre en main. Je pense que c'est un bon point car ce sont des compétences qui pourront me servir plus tard.

4.2.2 Maxime

N'ayant jamais touché à la programmation, ce stage a été une parfaite initiation au langage C et à l'implémentation de schémas numériques.

La partie GPU est intéressante, bien que délicate à prendre en main (on ne comprend pas toujours ce qui se passe).

Sur un aspect plus humain ce stage m'a permit de mieux appréhender le monde de la recherche, le "quotidien" du métier.

J'ai aussi apprécier le mélange de plusieurs domaine dans le but de créer quelque chose (physique, mathématique et informatique).

4.2.3 Lilian

Je commencerais par noter une particularité de ce stage *de mathématiques* : très peu de maths ... et beaucoup de code ! La simulation numérique n'était pas un sous-domaine de la programmation qui m'était étranger [36], mais la part de développement en C m'a permis de

36. J'avais déjà réalisé un solveur d'ED par la méthode de Runge-Kutta à l'ordre 4 sur calculatrice TI.

consolider ces bases. Le CUDA et la programmation parallèle étaient par contre tous deux entièrement nouveaux pour moi.

D'une manière générale, je suis très déçu du contact que nous avons eu avec la langage proposé par nVidia. Comme il a été indiqué plusieurs fois dans ce rapport, nous retenons plutôt un avis négatif de l'investissement dans le CUDA. Tout d'abord une installation délicate, presque cabalistique. Puis des outils peu faciles à prendre en main, et pour les deux plus utiles d'entre eux, un échec complet [37]. Ce qui est tout de même dommage, surtout que CUDA n'est pas un projet amateur codé dans un garage par deux étudiants, mais la solution de programmation GPGPU proposée par le leader du marché. Et enfin, presqu'aucune ressource en français, seulement les différents tutoriels de Thibaut Cuvelier [38].

Par ailleurs, nous n'avons finalement que peu progressé sur le plan théorique concernant les points abordés, car aucun travail de recherche n'a été vraiment fait. Ensuite, l'objectif initial n'ayant pas été atteint, il reste un sentiment de déception. La première séance de stage nous fit assister à un TP Matlab, qui consistait à coder ...un solveur 1D puis 2D pour Euler. Donc, en résumant grossièrement, 5 mois de stage nous ont permis de réaliser la même simulation en C et en CUDA. Et avec pour seule amélioration un *speed-up* d'à peine 10 pour le code parallèle.

Avouez que c'est tout de même décevant...Le travail mathématique réalisé est toutefois vraiment intéressant, permettant une vision peut-être nouvelle des systèmes hyperboliques.

Enfin, en cherchant un peu de mon côté, j'ai découvert qu'il existait des couplages entre CUDA et la plupart des logiciels de calculs utilisés : Matlab, Maple, Mathematica... Même un portage avec Python est disponible : PyCUDA. Nous avons manqué de temps pour faire nos propres comparaisons, mais il est bien possible que la simple utilisation du couplage *Matlab-CUDA* aurait permis d'avoir un meilleur speed-up que 5 ... Avec *beaucoup* moins d'investissement et de prises de têtes.

Malgré tout, le bilan global est plutôt positif. Mais les points sur lesquels j'ai progressé grâce au travail fait en stage [39] n'étaient pas vraiment parmi les objectifs, et n'ont peut-être pas de liens avec les compétences que j'aurais dû obtenir en seconde moitié de licence de maths. Le point le plus instructif a peut-être été la nature même du stage, à savoir travailler sur un même sujet, pendant plusieurs mois, avec les mêmes personnes.

Pour conclure, je m'avoue légèrement déçu du stage, en tant que stage *d'initiation à la recherche en maths*, mais ravi de l'expérience globale de programmation, de coopération en tant que travail de groupe, et de découverte sur le monde de la recherche qu'il m'a apporté.

37. On notera trois cuisants échecs successifs lors de plusieurs tentatives de prise en main du débuggeur *cuda-gdb*, et du profiler *nvvp*...

38. Par exemple ici : `http://tcuvelier.developpez.com/tutoriels/gpgpu/cuda/introduction/`.

39. Sans établir une liste exhaustive, le développement en C et la compilation en ligne de commande via *gcc*, la programmation bash, l'écriture de Makefile, la rédaction en LaTeX 2_ε, l'anglais informatique, l'architecture des GPU ...et quelques notions sur les schémas explicites d'ordre 1.

4.3 Bibliographie

Nous donnons ici une courte liste des ouvrages et des sites webs les plus importants que nous avons consultés. Au cours du rapport, de nombreuses autres références web sont données, surtout en ce qui concerne les outils utilisés.

Livres :

DeDu *Systèmes hyperboliques de lois de conservation*, de Bruno Despres et Francois Dubois ;

SaKr *CUDA by example*, de Jason Sanders et Edward Kandrot ;

Ghi99 *An overview of the VFFC-method and tools for the simulation of two-phase flows* de J. M. Ghidaglia ;

GhiPa *On Boundary conditions for multidimensional hyperbolic systems of conservation laws in the finite volume framework* de J. M. Ghidaglia et F. Pascal ;

GoRa *Numerical approximation of hyperbolic systems of conservation laws* par Edwige Godlewski et Pierre-Arnaud Raviart.

Sites web, articles :

– `http://www.nvidia.fr/object/what_is_cuda_new_fr.html` Page de `NVidia CUDA` ;

Buf06 `http://ufrmeca.univ-lyon1.fr/~buffat/COURS/AERO_HTML/courshtml1.html` Cours de dynamique des gaz, par Marc BUFFAT ;

4.4 Bilan des réalisations pratiques et théoriques

Nous rappelons et dressons ici un bilan des réalisations tant pratiques et théoriques auxquelles nous sommes parvenus dans ce stage.

Nous avons d'abord étudié l'équation d'advection, en 1D, puis en 2D, et étudié des schémas numériques explicites d'ordre 1. Il a ensuite fallu s'initier au C en implémentant des résolutions numériques avec ces schémas, afin de commencer à étudier l'architecture des GPU et le langage CUDA, pour paralléliser ces résolutions et les rendre significativement plus rapides. Une réflexion et un temps d'apprentissage ont été nécessaires pour afficher pertinemment les données ainsi produites, via GNUPlot puis le couple VTK et ParaView.

Ceci servant de préliminaire aux objectifs suivants, nous avons étudié après l'équation d'Euler, en travaillant sur une généralisation du premier schéma. Un travail théorique a été fait sur chaque composante de l'étude de ce schéma : stabilité, convergence, types de conditions initiales et de conditions de bords. Les difficultés supplémentaires posées par ce schéma nous ont ralentis dans l'implémentation des simulations, d'abord en C puis en CUDA.

Ayant manqué de temps mais surtout d'outils performants, nos codes finaux n'ont guère pus être améliorés ; alors qu'une part importante avait été donnée à l'*optimisation* dans le plan initial. Nous n'avons pas obtenu meilleur gain de vitesse qu'un facteur 5 entre le code séquentiel et parallèle final.

Enfin, une part conséquente de notre travail fut la visualisation, et l'apprentissage d'un outil plus performant (OpenGL), mais l'échec relatif du portage en CUDA fait ombre à ce point là. Enfin, l'interactivité proposée a été à la hauteur de nos espérances, permettant de conclure sur une touche créative et amusante.

4.5 Remerciements

Nous tenons à remercier, dans l'ordre, les personnes suivantes :

Florian de Vuyst Un grand merci à lui pour son suivi régulier, ses conseils pour les phases d'implémentation et pour le contenu théorique ;

Jean-Michel Ghidaglia Merci en particulier pour sa contribution dans l'étude des différentes conditions de bords ;

Daniel Chauveheid Du fait que nous partagions son bureau au LRC, il a été très présent avec nous. Que se soit pour des conseils de codes, des vérifications ou des relectures, pour des astuces techniques (pour le format VTK ou pour les animations en LaTeXou encore l'utilisation de Maple), ou pour des idées constructives (ajout d'un terme source, choix des conditions initiales), Daniel a toujours été d'un grand soutien. Merci beaucoup !

Saad Ben Jelloun Pour son aide lors du premier TD matlab et ses remarques pertinentes lors de nos soutenances blanches ;

Nicolas Pajor Pour ses conseils techniques judicieux pour Linux, en Bash ou avec LaTeX, et son humour ;

Christophe Labourdette Pour son support technique pour la machine du LRC, ses conseils sur le SSH et sur Linux ;

Frédéric Pascal Merci à ses remarques nombreuses et constructives lors du groupe de Travail du 18 juin ;

Frédéric Hecht [40] Merci à l'aide constante qu'il nous a prodigué en C, via Sophie, notamment pour l'OpenGL.

40. Père de Sophie, enseignant à l'Université de Paris-Jussieu ; `http://www.ann.jussieu.fr/hecht/`

5 Annexes

5.1 Implémentation en C et en CUDA

Avant d'aller plus en profondeur dans le travail réalisé, nous présentons ici quelques points concernant nos méthodes de développement. Une présentation des langages et des outils utilisés semblant nécessaire, mais sans trop de rapport avec le fil d'organisation de notre rapport, nous avons préféré placer ces remarques en annexe.

A propos du C

Pourquoi le C ? Le C est un des langages le plus pratiqué au monde. Mis au point dans les années 70, il n'a cessé de se développer, de s'uniformiser et de se populariser au cours du temps. C'est aussi la solution la plus "performante" pour coder sur un processeur actuel. Il est utilisable sur n'importe quelle plateforme, via n'importe quel éditeur de texte, et permet de produire du code machine [41] via un compilateur comme *gcc* [42], qui a l'avantage d'être *Open Source*, de supporter presque toutes les architectures, et d'être très performant.

Comment coder en C ? Par ailleurs, de nombreux EDIs (*pour **E**nvironnement de **D**éveloppement **I**ntégré*) sont disponibles. Nous signalons ici les outils utilisés pour coder via le langage C.

Remarque 2 (Quels outils pour coder en C ?). Les outils suivants sont normalement gratuitement accessibles sur leurs plateformes respectives.

Sophie Travaillant sous MacOS, Sophie a choisi de développer à l'aide de l'EDI *xCode* [43] ;

Maxime Travaillant sous Windows, Maxime a choisi de développer à l'aide de l'EDI *codeBlocks* [44].

Lilian Travaillant sous Ubuntu, Lilian a choisi de développer sans EDI, avec l'éditeur de texte *gEdit* [45] et de compiler en ligne de commande avec *gcc*. Pour plus de détails sur la compilation, voir 5.3 en annexe.

Le lecteur intéressé pourra notamment se référer aux document suivants : `http://www.siteduzero.com/tutoriel-3-14189-apprenez-a-programmer-en-c.html`, ou `http://www-ipst.u-strasbg.fr/pat/program/tpc.htm`. Ces deux tutoriels constituent de très bonnes initiations.

Autres outils Nous profitons de ce passage pour indiquer les autres outils utiles pour développer en C.

– Le site `http://www.cplusplus.com/reference/clibrary/cstdio/` contient une documentation précise et complète sur les bibliothèques standards utilisées en C ;

41. C'est ce qui lui permet d'être très performant.
42. Voir `http://gcc.gnu.org/` pour plus de détails. Normalement, *gcc* est installé sur toutes distribution `Unix`-like : MacOS, Ubuntu, KDE etc ...
43. Voir **??** pour plus de détails.
44. Voir `http://www.codeblocks.org/`. A l'avantage d'être Open Source et multi-plateforme.
45. Voir `http://projects.gnome.org/gedit/` pour plus de détails. Un très bon choix pour travailler sous Ubuntu ou KDE.

– Le débuggeur[46] *gdb*[47] est un outil permettant d'examiner avec minutie l'évolution d'un programme. Sa prise en main n'est pas facile, mais une fois bien compris, ou bien intégré dans un EDI correct, il permet de trouver rapidement les sources d'erreur et les bugs ;
– L'outil *GNU-Make*[48] permet de faciliter la compilation.

A propos du CUDA

Un langage plus spécifique En opposition aux points précédents, le langage CUDA *n'est pas aussi facile d'utilisation*. Tout d'abord parce que c'est un langage qui dépend fortement de l'architecture de la machine sur laquelle le code doit être exécuter. En effet, ce langage *propriétaire*[49] a pour but de permettre l'exécution de code **sur la carte vidéo** de votre ordinateur ! En effet, un même code C peut être compile et exécuté sur la plupart des ordinateurs usuels, sans modifications, ce qui est impensable[50] pour un programe en CUDA ...

Ainsi, il n'est pas possible de tester un programme CUDA sur n'importe quelle machine. La première condition est déjà d'avoir une carte graphique produite par NVIDIA, et qui supporte la dernière version de CUDA. Ce qui représente moins de 40% du marché, en fait. Par exemple, Lilian a du coder sur l'ordinateur mis à disposition au LRC car sa machine personnelle était équipée d'une carte AMD.

Un point positif néanmoins est l'organisation de ce langage. En effet, CUDA (qui signifie Compute Unified Device Architecture) a était conçu comme une *sur-couche* au C. C'est-à-dire que la syntaxe du langage est la même que le C, il n'y a donc rien à "oublier" lorsque l'on code en C et en CUDA ; et rajoute seulement quelques notations, et quelques concepts.

Un seul compilateur Ensuite, il faut nécessairement utiliser le compilateur fourni gratuitement[51] par l'entreprise californienne : *nvcc*. L'installation nécessite aussi une multitude de composants aux noms obscurs et à la raison d'être encore plus mystérieuses.

La *tool-kit* est multiplateforme, ce qui est un point positif. Pour l'obtenir, il faut consulter l'adresse suivante : `http://developer.nvidia.com/cuda-downloads`.

Ensuite, un second problème réside dans l'édition de code en CUDA. En effet, les seuls EDI à permettre le développement en CUDA à l'heure actuelle semble être Eclipse, l'EDI d'Oracle[52], ou Visual Studio, l'EDI de Microsoft[53]. Nous n'avons testé aucune de ces deux solutions. Il faut préciser qu'il est nécessaire d'obtenir une autorisation de NVIDIA pour coder avec Visual Studio.

46. Voir `http://fr.wikipedia.org/wiki/D\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{e\global\mathchardef\accent@spacefactor\spacefactor}\accent19e\egroup\spacefactor\accent@spacefactorbogueur` pour plus de détails
47. Voir `http://sources.redhat.com/gdb/documentation/` pour une documentation.
48. Comme pour *gcc* ou *gdb*, cet outil est déjà installé sur tout `Unix`-like. Voir `http://www.gnu.org/software/make/` pour plus de détails. Par ailleurs, le lecteur intéressé pourra trouver deux scripts bash facilitant la compilation de nos codes en ligne ici `http://www.dptinfo.ens-cachan.fr/~lbesson/publis/autocompilation.zip`
49. C'est-à-dire que le code source du compilateur de NVIDIA n'est pas disponible, contrairement à celui de *gcc* par exemple.
50. Et nous l'avons constaté à plusieurs reprises, du fait que chaque membre du groupe travaillait sur une plateforme différente.
51. Pour l'instant ...
52. Voir `http://developer.nvidia.com/nsight-eclipse-edition`.
53. Voir `http://developer.nvidia.com/nvidia-nsight-visual-studio-edition`

Ceci étant, nous nous sommes donc résignés à coder avec un éditeur de texte, et à compiler *en ligne de commande* avec *nvcc*. Pour plus de détails sur la prise en main de cet outil, voir le paragraphe sur la compilation en annexe (5.3).

Des outils difficiles à utiliser Nous présentions plus haut les outils pour développer en C : un débuggeur comme *gdb* permet de localiser facilement les problèmes de conception, et un outil comme *gprof*[54] permet de localiser facilement les problèmes d'efficacité du code, et donc de savoir quel modification apporter pour **optimiser** les performances d'une simulation.

Ainsi, il serait normal de trouver les mêmes genres d'outils pour le CUDA. Et en effet, NVIDIA propose deux équivalents à ces outils : le débuggeur *cuda-gdb* et le profiler *nvvp* (pour NVidia Visual Profiler). **Mais**, malgré nos efforts nombreux et réguliers pour installer, comprendre et appréhender ces deux outils, nous n'avons pas été capables de nous en servir avec succès ne serait-ce qu'une seule fois. En particulier, pour *cuda-gdb*, suivre les instructions du tutoriel officiel n'a pas marché.

Un mauvais contact avec le CUDA Bref, nous signalons une grosse déception quant aux outils disponibles pour coder en CUDA ; et ce manque rend délicat un bon développement, impliquant une perte de temps significative dans les phases d'implémentation.

Autres remarques Parmi les objectifs du stage, acquérir une connaissance de la programmation parallèle via CUDA fut une des lignes de conduite de notre travail.

Même si le but était d'écrire un programme en CUDA, il a d'abord fallu commencer par coder en C pour se familiariser avec les méthodes de programmation, d'organisation des données et d'implémentation de schéma numérique. Par la suite, nous donnons des extraits de codes, en C ou en CUDA, qui se veulent illustratifs de certains points particuliers. De plus amples détails sur CUDA sont présentés plus loin (voir 1.2.3).

5.2 Détails supplémentaire sur OpenGL

Pour le choix de la couleur nous avons créé une fonction qui renvoie une couleur par rapport aux bornes max et min de notre fonction.

```
1                 void hsvToRgb (float h, float s, float v, float * r, float * g,
                      float * b)
2 {
3     int i;
4     float aa, bb, cc, f;
5
6     if (s == 0) /* Grayscale */
7         *r = *g = *b = v;
8     else {
9         if (h == 1.0) h = 0;
10        h *= 6.0;
11        i = (int) h;
12        f = h - i;
13        aa = v * (1 - s);
14        bb = v * (1 - (s * f));
15        cc = v * (1 - (s * (1 - f)));
16        switch (i) {
```

54. Nous n'avons pas utilisé de *profiler* pour nos codes C.

```
            case 0: *r = v;  *g = cc; *b = aa; break;
            case 1: *r = bb; *g = v;  *b = aa; break;
            case 2: *r = aa; *g = v;  *b = cc; break;
            case 3: *r = aa; *g = bb; *b = v;  break;
            case 4: *r = cc; *g = aa; *b = v;  break;
            case 5: *r = v;  *g = aa; *b = bb; break;
        }
    }
}
void   SetColor(R f)
{
    float r=0,g=0,b=0;
    assert(global);
    R fmin=global->zmin; // borne de la fonction
    R fmax=global->zmax;

    hsvToRgb(0.99*(f-fmin)/(fmax-fmin),1,1,&r,&g,&b);
    glColor3f(r,g,b);
}
```

La fonction hsvToRgb nous permet de convertir le code couleur "teinte, saturation, valeur" a celui "rouge, vert, bleu".

Pour la création du maillage et le choix des couleurs nous n'avons donc plus qu'à créer une fonction comme celle qui suit.

```
                void Global_Affiche(Global*this)
{
    glPolygonMode(GL_FRONT,GL_FILL); // mode affichage des polygones

    int NX = this->N, NY =this->M; // nb de pas en x et y
    double x0 = 0, y0=0;
    double x1 =1, y1=1;
    double dx0 = (x1-x0)/NX;
    double dy0 = (y1-y0)/NY;
    double P[4][3];
    int ip[4] = { 0,1,1,0};
    int jp[4] = { 0,0,1,1};
    int i,j,k;
    for ( i=0;i<NX;++i)
        for (j=0;j<NY;++j)
        {
            for( k=0;k<4;++k)
            {
                P[k][0] = x0 + (i+ip[k]) * dx0;
                P[k][1] = y0 + (j+jp[k]) * dy0;
                P[k][2] = this->f[i*NX+j];
            }

            glBegin(GL_POLYGON);
            for (k=0; k<4; ++k)
            {
                SetColor(this->f[i*NX+j]);
                glVertex3f(P[k][0],P[k][1],P[k][2]);
            }
            glEnd();

```

```
32          }
33  }
```

Le plus dur dans cette affichage est en fait de faire une animation. Pour cela nous avons utilisé 2 flots en parallèle ; un qui actualise le vecteur que nous voulons afficher, et un qui affiche. Pour cela nous avons donc créé une fonction main_ calcul dans laquelle tous les calculs sont faits et dans le main nous nous servons de la bibliothèque de programmation multi-threads `pthread` pour faire l'affichage et le calcul en parallèle.

```
1                   int main(int argc, char** argv)
2   {
3       bool stereo=0;
4       bool fullscreen = 0;
5       int k,kas=0;
6       R rapz=1;
7       int nbiso=20;
8       for ( k=1;k < argc;k++)
9       {
10          if( strcmp("-s",argv[k])==0)
11          { stereo=1;}
12          else if (strcmp("-f",argv[k])==0)
13          { fullscreen=1;}
14      }
15
16      glutInit(&argc, argv);
17      rho = (double*) malloc(sizeof(R)*N*M);
18      p = (double*) malloc(sizeof(R)*N*M);
19
20      fprintf(stderr," #### f = %p \n",rho);
21
22      int R = pthread_create(&threadCalcul,NULL,&main_calcul,rho);
23      if(R != 0)printf("Thread::Start: Thread could not be created %d\n",R);
24
25      if(stereo)
26          glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH | GLUT_STEREO);
27      else
28          glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH );
29      int Height = 512;
30      int Width = 512;
31      glutInitWindowSize(Width , Height);
32      glutInitWindowPosition(100, 100);
33
34      glutCreateWindow("Euler2D ");
35      glutPushWindow();
36      if (fullscreen)
37          glutFullScreen();
38      void * Th=0;
39
40      global= Global_Global(Th,rho,N,M,Height,Width,rapz,nbiso,stereo);
41
42      glEnable(GL_DEPTH_TEST);
43      glutReshapeFunc( Reshape ); // pour changement de fenetre
44      glutKeyboardFunc( Key );    // pour les evenements clavier
45      glutSpecialFunc(SpecialKey);
46      glutMouseFunc(Mouse);       // pour les evenements souris
```

```
47      glutMotionFunc(MotionMouse); // les mouvements de la souris
48      glutDisplayFunc( Display ); // l'affichage
49      glutMainLoop();
50      return 0;
51  }
```

Il y a ensuite plein d'autres fonctions qui permettent de changer l'angle de vue etc... Elles sont disponibles dans le code téléchargeable en ligne.

Et sur le GPU Pour pouvoir faire l'affichage sur le GPU directement toutes les fonctions ont été créés sur le device. Ce programme ne marche pas encore. Il faudrait quelques semaines de stage en plus pour réussir à le faire marcher...

Nous avons néanmoins réussi à intégrer à notre solveur CUDA la possibilité d'afficher avec OpenGL, mais en laissant le CPU faire les calculs d'affichages.

5.3 A propos de compilation

Nous détaillons ici : comment compiler avec *gcc* et *nvcc*, comment éviter les pièges classiques en CUDA, et un exemple conséquent de code. Le lecteur intéressé peut télécharger les dernières versions de nos solveurs en ligne à l'adresse suivante : `http://www.dptinfo.ens-cachan.fr/~lbesson/publis/code_stage.zip`. [55]

En C Pour compiler notre solveur C, appelé poétiquement "`Euler_3D_C_avecG_conditions_limites_differentes.c`", la ligne de commande suivante suffit :

```
gcc -O3 -o solveurC Euler_3D_C_avecG_conditions_limites_differentes.c -lm
```

Le `-O3` précise qu'on utilise le niveau maximum d'optimisation que propose *gcc* et *nvcc*. Le `-lm` précise qu'on utilise des fonctions de la librairie mathématique (`sqrt` par exemple). La dernière version de notre solveur permet de faire varier presque tous les paramètres de la simulation. La ligne suivante décrit les paramètres passables en arguments de l'exécutable `solveurC` ainsi produit :

```
time ./solveurC <N> <M> <O> <T> <cond_init>
  <cond_bordX> <cond_bordY> <cond_bordZ> [<debug>]
```

Ainsi, il est possible, sans recompiler ni ré-éditer le code, de faire varier sur chaque dimension le nombre de points et le type de conditions au bord, et de modifier le nombre d'étapes maximum (T) et les conditions initiales.

Durant le calcul, le programme informe l'utilisateur des différentes étapes parcourues, et affiche le nom de chaque fichier produit. L'évolution du pas de temps est aussi renseignée. Enfin, si le terminal le permet, les informations intéressantes sont affichées en couleurs pour bien les voir.

55. Le résultat de nos travaux est librement disponible, mais distribué sous licence GPL.

En CUDA Pour compiler notre solveur CUDA, appelé "`Euler_2D_CUDA_cu`", la ligne de commande suivante suffit :

```
nvcc -O3 -o solveurCUDA Euler_2D_CUDA.cu -lm
```

Le `-O3` précise qu'on utilise le niveau maximum d'optimisation que propose *gcc* et *nvcc*. Le `-lm` précise qu'on utilise des fonctions de la librairie mathématique (`sqrt` par exemple). La dernière version de notre solveur permet de faire varier presque tous les paramètres de la simulation. La ligne suivante décrit les paramètres passables en arguments de l'exécutable `solveurCUDA` ainsi produit :

```
time ./solveurCUDA <N> <M> <T> <cond_init> <cond_bordX>
  <cond_bordY> <debugVTK> <THREADS> <GRIDS> [<debug>]
```

Ainsi, il est possible, sans recompiler ni ré-éditer le code, de faire varier sur chaque dimension le nombre de points et le type de condition au bord, et de modifier le nombre d'étapes maximum (T) et les conditions initiales.

En pratique, les paramètres peuvent être fixés comme suit :

- $N = 512$, ou $N = 256, 128, 64, 32$. Et $M = N$;
- $T = 1024$ pour avoir le temps de voir bien évolur la simulation ;
- `cond_init`. Cet entier vaut 1 si on veut des conditions initiales orientées selon x, 0 si selon y, 2 si circulaires et 3 si constantes ;
- `cond_bordX` et `cond_bordY` peuvent prendre les valeurs 0, pour du périodique, 1, pour de l'absorbant, 2 pour du mur ;
- `debugVtk` doit valoir 0 ;
- `THREADS` doit valoir 16, ou 32. Si aucun des deux ne génère une simulation convenable, il faut essayer 8 ou 64 ;
- `GRIDS` : même remarque, d'abord essayer 64, puis diminuer avec des puissances de 2 ;

5.4 Caractéristiques de notre machine de test

Voici les détails techniques de l'architecture CUDA de la machine sur laquelle nous avons réalisé nos simulations : [56]

```
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Found 1 CUDA Capable device(s)

Device 0: "Quadro 2000"
  CUDA Driver Version / Runtime Version          4.1 / 4.1
  CUDA Capability Major/Minor version number:    2.1
  Total amount of global memory:                 1023 MBytes (1072889856 bytes)
  ( 4) Multiprocessors x (48) CUDA Cores/MP:     192 CUDA Cores
  GPU Clock Speed:                               1.25 GHz
```

56. Elles ont été obtenues à partir du programme "deviceQuery" de la CUDA Computing Toolkit.

```
  Memory Clock rate:                             1304.00 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 262144 bytes
  Max Texture Dimension Size (x,y,z)
    1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers
    1D=(16384) x 2048, 2D=(16384,16384) x 2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and execution:                 Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Concurrent kernel execution:                   Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support enabled:                No
  Device is using TCC driver mode:               No
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           3 / 0
  Compute Mode:
     < Default (multiple host threads can use
    ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 4.1,
    CUDA Runtime Version = 4.1, NumDevs = 1, Device = Quadro 2000
```

Les points importants à noter sont :

- Nombre de cœurs CUDA : 192 ;
- Nombre maximum de threads par bloc : 1024 ;
- Nombre maximum des dimensions pour un bloc : 1024 x 1024 x 64 ;
- Nombre maximum des dimensions pour une grille : 65535 x 65535 x 65535.

Ces informations permettent de déterminer le nombre maximum de threads et de blocs utilisés lors de l'appel à un kernel CUDA dans du code CPU (*ie* une fonction définie avec le mot clé `__device__`). Et pour l'architecture séquentielle, tous nos programmes C ont été exécuté sur un seul cœur (Intel Xeon E5606), cadencé à 2.13 GHz. Enfin, les mesures de temps d'exécution ont été réalisées de manière externe au code, via l'outil GNU time disponible en ligne de commande.

Fin.