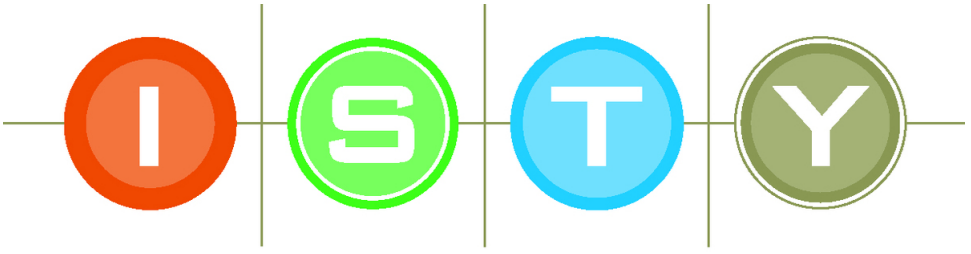
Analyse de performance et optimisation de code

Pierre AYOUB

6 février 2019



INSTITUT DES SCIENCES ET TECHNIQUES DES YVELINES

Résumé

La simulation numérique est un procédé informatique visant à modéliser un phénomène par ordinateur, s'agissant le plus souvent d'un phénomène physique. Cette modélisation prend forme par des systèmes d'équations décrivant l'état du système physique représenté à chaque instant. De nombreux domaines scientifiques convergent vers la simulation informatique, tel que certaines branches de la physique, de l'analyse et de l'optimisation mathématique, ou encore le calcul haute performance en informatique. Enfin, la simulation trouve naturellement de nombreuses applications concernant des sujets variés, tel que la simulation du climat et des évènements météorologiques, la simulation d'essais nucléaires, de l'effet d'un médicament sur un corps, ou encore des astres et de l'univers. Ce rapport s'articulera donc autour de la simulation de fumée, phénomène impliquant les lois de la mécanique des fluides. Notre travail portera sur l'aspect du calcul haute performance de cette simulation.

Table des matières

1 Introduction

Le projet que nous vous présentons aujourd'hui consiste à analyser puis, grâce à nos mesures, optimiser un code de simulation numérique. Ce dernier nous offre une interface graphique permettant d'ajouter de la fumée dans un espace confiné et, ainsi, d'en observer le comportement. Nous pouvons influencer la quantité de fumée et sa vélocité dans l'espace. De plus, l'application nous donne le contrôle sur la résolution de la simulation, cela revient à dire sur sa précision, qui détermine principalement la performance du programme.

Le déroulement du projet s'est effectué en plusieurs étapes distinctes :

Analyse du code Cette phase consiste à analyser le programme d'un point de vue mathématique et informatique. De cette première approche, il s'agira de comprendre les opérations du programme sur les équations qui régissent le système physique. De l'autre approche, il convient d'étudier l'architecture logicielle de l'application, ainsi que les choix mis en œuvres afin d'implémenter le ou les algorithmes nécessaires.

Protocole expérimental Une fois l'analyse effectuée, nous pouvons en déduire le moyen le plus adapté afin de mesurer les performances de notre implémentation. Nous allons donc mettre en avant les critères théoriques à atteindre dans nos mesures, puis nous exposerons la manière dont nous avons mis ceci en pratique.

Optimisations et mesures Grâce au protocole mis en place, nous pouvons quantifier la performance du programme. De ce fait, nous serons en mesure d'expérimenter différentes techniques d'optimisation sur le programme et d'en calculer l'accélération.

2 Analyse du code

L'analyse du code est la première étape à effectuer afin d'optimiser un code. Il s'agit d'identifier les différentes sections et fonctions qui pourraient être critiques, ainsi que de comprendre mathématiquement quelles fonctions remplissent quels rôles.

L'interface du programme et l'interaction avec l'utilisateur est géré par les fichiers « demo.html », « FluidSolver.java » et « WebStart.java ». À priori, ces fichiers nous importe peu dans notre processus d'analyse et d'optimisation. Le cœur de la simulation se déroule dans le fichier « fluid.c », dans lequel se situe les fonctions de calcul des phénomènes physique.

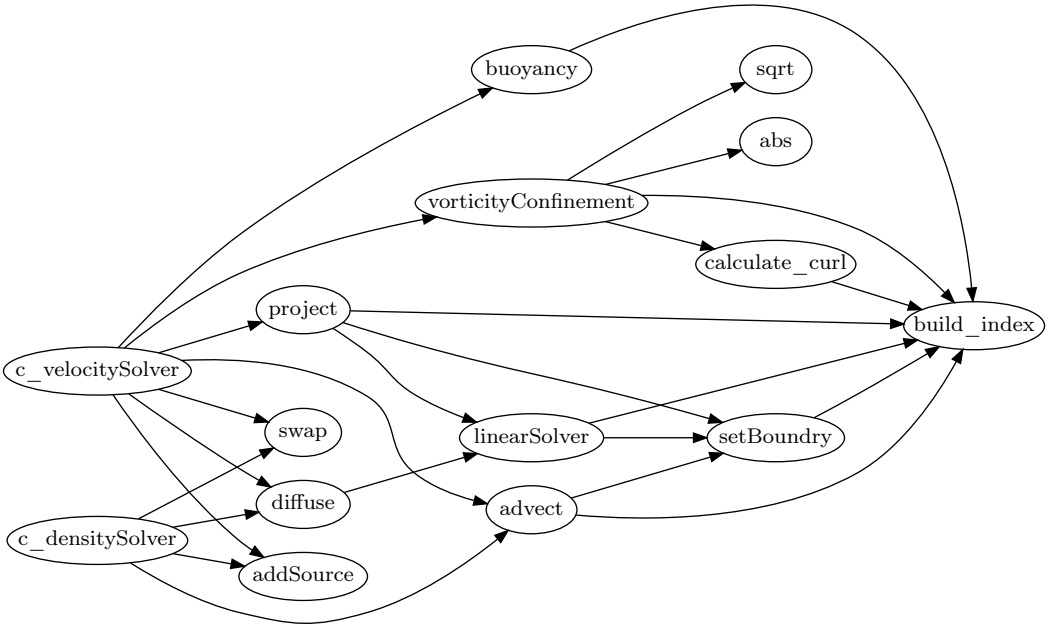


FIGURE 1 – Graphe d'appel du fichier *fluid.c*

Avant d'observer et d'analyser attentivement le code du fichier « fluid.c », nous avons utilisé **Cflow** afin d'obtenir un graphe d'appel pour avoir une vue d'ensemble et observer les liens entre les fonctions. Le résultat est présenté dans la figure 1. Nous décrivons ici les fonctions principales. On remarque très rapidement les deux points d'entrés du programme : « c_densitySolver() » et « c_velocitySolver() ». Ces deux fonctions sont utilisées conjointement pour assurer les deux fonctionnalités de notre simulation : la première permet de calculer la position d'une particule de fumée ajouté par la souris, la deuxième permet de calculer leur déplacement en fonction des différentes vélocités dans l'espace. Nous observons ensuite deux fonctions qui sont ici pour assurer des opérations physique : « project() » et « diffuse() ». Le nom laisse sous entendre que la première permet de faire une projection (des coordonnées d'un système à un autre ?) et la seconde permet de diffuser (une particule dans l'espace ?), mais nos connaissances en mécanique des fluides nous arrêtent ici pour les suppositions. Cependant, il est intéressant d'observer que ces deux fonctions font appel à une nouvelle fonction : « linearSolver() ». Cette fonction, comme le nom le laisse entendre, est le cœur de calcul de l'application : elle permet de résoudre

un système d'équation linéaire. Il ne fait aucune doute que c'est dans cette fonction que de nombreuses optimisations seront possibles ! Enfin, on observe quelques autres fonctions de physique (« vorticityConfinement() », « advect() », « buoyancy() ») ou des fonctions de mathématiques (« sqrt() », « abs() ») qui gravitent autour des fonctions précédemment décrites. Enfin, une fonction semble être appelée par quasiment toutes les autres de manière récursive : la fonction « build_index() » . Cette fonction, très courte, permet de passer d'un couple de coordonnées (i, j) identifiant une donnée à un offset correspondant à son emplacement mémoire par rapport à la base de la matrice. Cette fonction, présente à de nombreux endroits dans le code, devrait aussi avoir droit à une attention particulière.

3 Protocole expérimental

La mise en place d'un protocole expérimental de mesure est une étape nécessaire et cruciale dans tout optimisation de code. D'une part, le but de ce protocole est de mettre en lumière les points chauds du programme, c'est-à-dire les parties du code qui ralentissent considérablement l'exécution de la simulation. Ces points chauds seront les cibles de nos optimisations. D'autre part, après chaque tentative d'optimisation, le protocole doit nous permettre de mesurer l'impact de cette dernière, qu'il soit positif ou négatif, et enfin de le quantifier.

3.1 Théorie

Le protocole se doit d'être le plus représentatif possible d'une utilisation typique du programme. Pour cela, on ne doit pas uniquement mesurer la simulation exécutant une fonctionnalité particulière, mais l'ensemble des fonctionnalités. Il est bon de noter que ceci est vrai dans le cas d'un unique protocole, comme ici : en effet, on pourrait imaginer plusieurs protocoles différents pour optimiser chaque fonctionnalité indépendamment des autres. Ici, le programme nous offre deux fonctionnalités : ajouter de la fumée dans un espace confiné et ajouter une vélocité (c'est-à-dire un vecteur vitesse) modifiant ainsi la trajectoire de la fumée. Notre protocole doit donc faire intervenir ces deux possibilités dans notre expérience.

Lors de nos expériences, il ne faut pas oublier que le hasard ou l'aléa des mesures peuvent biaiser un résultat. Afin d'éviter cela, il faut donc utiliser une valeur moyenne ou une valeur médiane.

3.2 Pratique

Pour effectuer nos mesures, il nous faut utiliser un outil de d'analyse de performance. Plusieurs possibilités s'offrent à nous, parmi lesquels nous pouvons notamment citer : **Performance Counters for Linux (perf)**, **OProfile**, **Gprof**, **MAQAO**, **Callgrind/Kcachegrind**, et bien d'autre. Parmi ces choix-là, nous avons choisis d'utiliser **perf**. Nous pouvons lui citer plusieurs avantages qui nous ont intéressés : intégration en ligne de commande, outil bas-niveau, développement stable, accès aux compteurs hardware. Nous pouvons donc en déduire que c'est un outil tout à fait adapter à l'analyse de code bas-niveau (C, assembleur).

Le programme de simulation utilise une interface graphique pour rentrer en interaction avec l'utilisateur. De ce fait, nous ne pouvons pas lui donner un jeu de données défini à l'avance qui serais constant entre les différentes mesures. Nous devons alors définir des actions à réaliser de la manière la plus similaire possible entre les exécutions. De manière « arbitraire », nous définissions un run de la manière suivante : une fois l'applet lancé, nous traçons une ligne de fumée en diagonale, du point en bas à droite au point en haut à gauche [clic gauche enfoncé], puis nous refaisons la même opération en appliquant une vélocité sur ce trajet [clic droit enfoncé], dans le même sens.

Cependant, même en appliquant les mêmes actions entre chaque expérience, les temps d'exécution seront irrémédiablement différents. Dès lors, pour mesurer la performance d'une fonction, nous n'allons pas regarder le nombre de cycles passés dans cette fonction mais son pourcentage de cycle par rapport à l'ensemble du programme. En effet, étant donné que le pourcentage est un rapport sur le temps de l'ensemble du programme, ce dernier peux varier tout en gardant un pourcentage similaire pour une même fonction, car le nombre de cycles passé dans cette fonction augmentera proportionnellement.

Pendant nos expériences, nous choisirons une grille de 225x225 car c'est le premier seuil où l'application devient trop lente pour être utilisée sur notre machine de test. Cela nous permettre de mieux nous rendre compte de nos optimisations.

Lors de nos mesures préliminaires, on observe au maximum un écart de ± 5 % entre les différentes mesures successives. Nous choisissons donc, pour chaque résultat, d'effectuer 5 tests et de prendre la valeur médiane : cela nous semble suffisant.

4 Optimisations et mesures

Le compilateur utilisé pour ce projet est GNU Compiler Collection (GCC) v8.2.0. La mise en place de la compilation consiste à déterminer les bons flags de compilation sur lesquels nous allons partir pour optimiser notre code. Nous ajoutons tout d'abord « -g3 » afin d'inclure dans le binaire les symboles et informations sur le code source. Nous nous assurons que ce flag n'as pas d'incidence sur les performances. Ci-dessous les résultats des analyses du projet au départ, avec GCC.

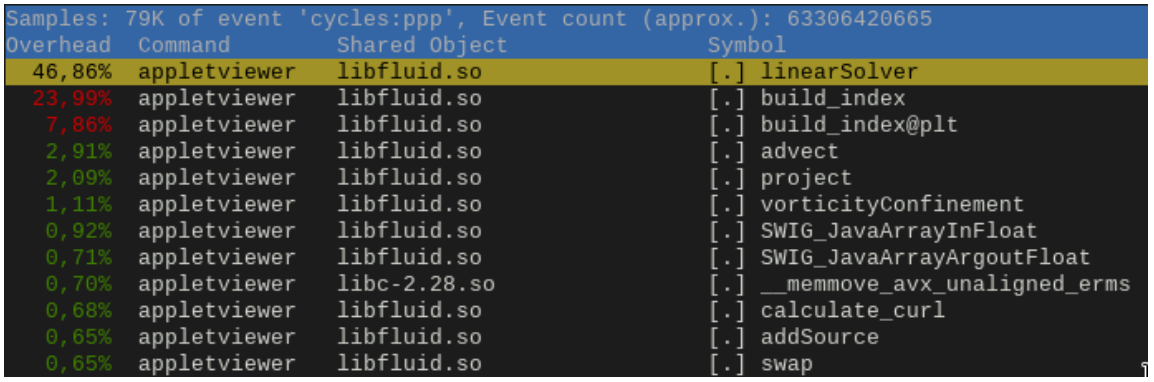


FIGURE 2 – Profilage global avec GCC en « -O0 »

Comme nous pouvions l'imaginer, la Figure 2 nous confirme que les fonctions les plus chaudes sont « linearSolver() » et « build_index() », la première de par sa nature de calcul intensif, la deuxième du fait qu'elle soit appelée énormément de fois. On remarque que toutes les autres fonctions sont d'une importance mineure dans le temps d'exécution du programme : inférieure à 3% pour la plus chaude. À elle seule, la fonction « linearSolver() » prend presque 50% du temps d'exécution du programme : c'est la plus importante.

La Figure 3 nous permet d'analyser quelles instructions de la fonction « linearSolver() » ont un impact important sur le temps d'exécution. La première image nous montre un lot successif d'instructions prenant chacune 1 à 4% du temps d'exécution : des *add*, *mul*, *mov*, *lea*. La deuxième image met en évidence une instruction *div* qui prends à elle seule 16% du temps de la fonction. On identifie donc parfaitement les deux sources de ralentissement de la fonction : la première étant un trop grand nombre d'instructions pour effectuer les calculs et la seconde étant la division.

Le temps passé dans la fonction « build_index() » est divisé équitablement en *mov*/*add* d'après la Figure 4, permettant de faire les deux additions et la multiplication que doit faire la fonction. On remarque tout de même que 32% du temps de la fonction est passé dans la mise en place de la stack frame de la fonction (2 premières instructions *mov*). Enfin, on remarquera que 7% du temps de la simulation est passée dans la fonction « build_index@plt ». Pour comprendre d'où vient cette fonction, il faut revenir sur les bases du fonctionnement du chargement dynamique de code. Afin d'avoir des bibliothèques partagées qui puissent être chargées qu'une seule fois en mémoire mais à des emplacements indépendants pour deux processus différents, on utilise un mécanisme de code à position indépendante (Position-Independent

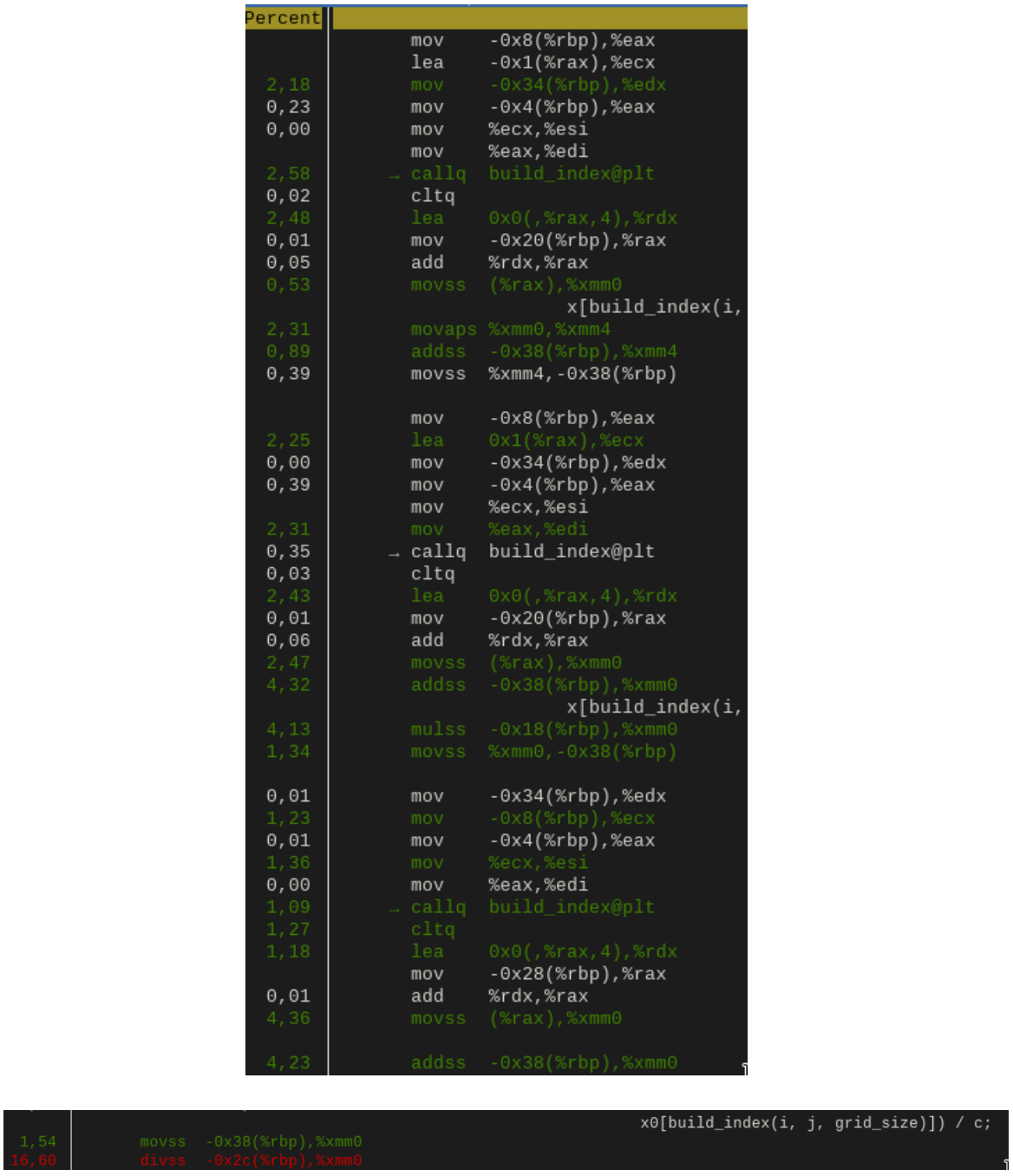


FIGURE 3 – Profilage de « linearSolver() » avec GCC en « -O0 »

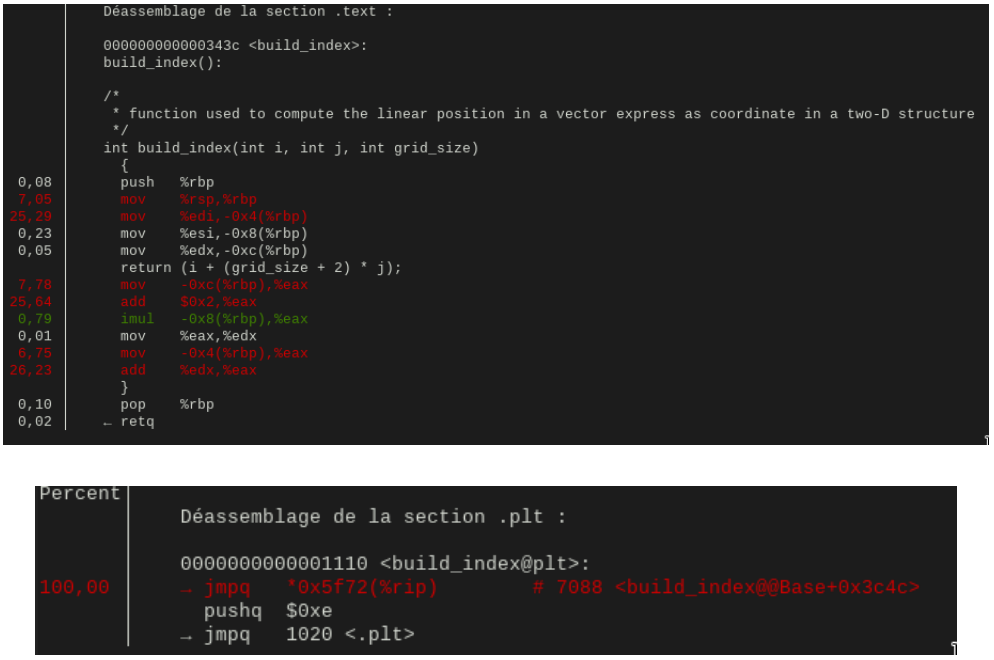


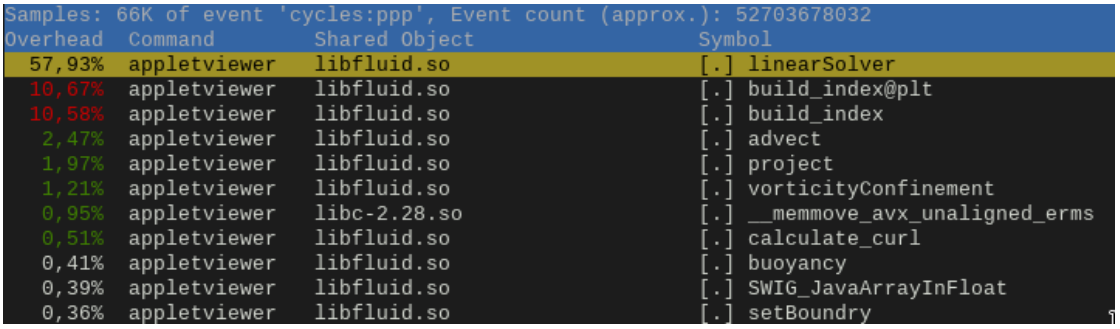
FIGURE 4 – Profilage de « build_index() » et « build_index()@plt » avec GCC en « -O0 »

Code (PIC)). Ce système fait recours à une Global Offset Table (GOT) contenant les adresses des variables et fonctions dont l'emplacement n'est pas connu au moment de la compilation. Cette GOT est consultée par les fonctions contenues dans la Procedure Linkage Table (PLT), une table contenant des fonctions *stubs* (wrapper) qui sont appelées à la place de la vraie fonction. Cette fonction stub (@plt) permet d'aller consulter la GOT, de remplir la GOT lors du premier appel (*lazy loader*), enfin d'exécuter la vraie fonction. Toutes les prochaines fois ou la fonction stub sera appelée, alors on va directement charger l'adresse contenu dans la *GOT* pour appeler la fonction désirée.

4.1 Mise en place de la compilation

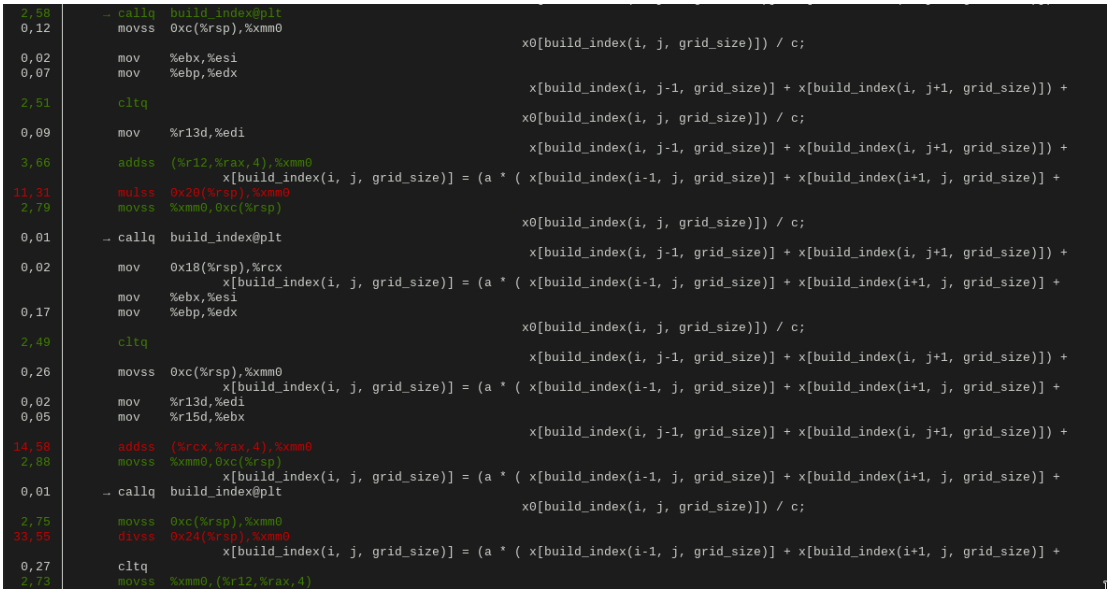
Premièrement, nous avons testé les flags suivants activant des options de lot d'optimisation : « -O1 », « -O2 », « -O3 » (agressivité au détriment de la taille du code généré), « -Os » (optimisation de la taille du code). L'utilisation d'un de ces flags nous apporte une légère amélioration de fluidité visuelle, cependant il se trouve que nous n'avons pas observé de différence majeure entre ces 4 différentes flags, que ce soit en termes de code généré ou de performances et répartition du temps entre les fonctions. Ci-dessous les images des résultats avec GCC et « -O2 ».

Au premier abord de la Figure 5, nous pourrions être étonnés que la fonction « linearSolver() » soit passée de 46% à 57%, comme si l'optimisation par le compilateur l'avait rendu plus lente. En fait il n'en est rien : on passe en effet plus de temps dans la fonction « linearSolver() » car on passe moins de temps dans les autres fonctions, ces dernières étant elles aussi optimisées. En effet, on voit que le temps passé dans la



```
Samples: 66K of event 'cycles:ppp', Event count (approx.): 52703678032
Overhead  Command        Shared Object                Symbol
  57,93%  appletviewer   libfluid.so                  [.] linearSolver
  10,67%  appletviewer   libfluid.so                  [.] build_index@plt
  10,58%  appletviewer   libfluid.so                  [.] build_index
   2,47%  appletviewer   libfluid.so                  [.] advect
   1,97%  appletviewer   libfluid.so                  [.] project
   1,21%  appletviewer   libfluid.so                  [.] vorticityConfinement
   0,95%  appletviewer   libc-2.28.so                 [.] __memmove_avx_unaligned_erms
   0,51%  appletviewer   libfluid.so                  [.] calculate_curl
   0,41%  appletviewer   libfluid.so                  [.] buoyancy
   0,39%  appletviewer   libfluid.so                  [.] SWIG_JavaArrayInFloat
   0,36%  appletviewer   libfluid.so                  [.] setBoundry
```

FIGURE 5 – Profilage global avec GCC en « -O2 »



```
 2,58    → callq  build_index@plt
 0,12      movss  0xc(%rsp),%xmm0
                                                    x0[build_index(i, j, grid_size)]) / c;
 0,02      mov    %ebx,%esi
 0,07      mov    %ebp,%edx
                                                     x[build_index(i, j-1, grid_size)] + x[build_index(i, j+1, grid_size)]) +
 2,51      cltq
                                                    x0[build_index(i, j, grid_size)]) / c;
 0,09      mov    %r13d,%edi
                                                     x[build_index(i, j-1, grid_size)] + x[build_index(i, j+1, grid_size)]) +
 3,66      addss  (%r12,%rax,4),%xmm0
                        x[build_index(i, j, grid_size)] = (a * ( x[build_index(i-1, j, grid_size)] + x[build_index(i+1, j, grid_size)] +
11,31      mulss  0x20(%rsp),%xmm0
 2,79      movss  %xmm0,0xc(%rsp)
                                                    x0[build_index(i, j, grid_size)]) / c;
 0,01    → callq  build_index@plt
                                                     x[build_index(i, j-1, grid_size)] + x[build_index(i, j+1, grid_size)]) +
 0,02      mov    0x18(%rsp),%rcx
                        x[build_index(i, j, grid_size)] = (a * ( x[build_index(i-1, j, grid_size)] + x[build_index(i+1, j, grid_size)] +
           mov    %ebx,%esi
 0,17      mov    %ebp,%edx
                                                    x0[build_index(i, j, grid_size)]) / c;
 2,49      cltq
                                                     x[build_index(i, j-1, grid_size)] + x[build_index(i, j+1, grid_size)]) +
 0,26      movss  0xc(%rsp),%xmm0
                        x[build_index(i, j, grid_size)] = (a * ( x[build_index(i-1, j, grid_size)] + x[build_index(i+1, j, grid_size)] +
 0,02      mov    %r13d,%edi
 0,05      mov    %r15d,%ebx
                                                     x[build_index(i, j-1, grid_size)] + x[build_index(i, j+1, grid_size)]) +
14,58      addss  (%rcx,%rax,4),%xmm0
 2,88      movss  %xmm0,0xc(%rsp)
                        x[build_index(i, j, grid_size)] = (a * ( x[build_index(i-1, j, grid_size)] + x[build_index(i+1, j, grid_size)] +
 0,01    → callq  build_index@plt
                                                    x0[build_index(i, j, grid_size)]) / c;
 2,75      movss  0xc(%rsp),%xmm0
33,55      divss  0x24(%rsp),%xmm0
                        x[build_index(i, j, grid_size)] = (a * ( x[build_index(i-1, j, grid_size)] + x[build_index(i+1, j, grid_size)] +
 0,27      cltq
 2,73      movss  %xmm0,(%r12,%rax,4)
```

FIGURE 6 – Profilage de « linearSolver() » avec GCC en « -O2 »

fonction « build_index() » est passé de 24% à 10%. Enfin, il en va de même pour le temps qui a augmenté de 3% pour « build_index@plt » : puisque « build_index() » est plus courte, alors on passe autant de temps à appeler la fonction qu'à effectuer les calculs.

Concernant la fonction « linearSolver() » de la Figure 6, on voit que le nombre d'instructions à nettement été réduit. On est passé de beaucoup d'instruction prenant chacune 1 à 4% à quelques petites instructions et 3 instructions principales de 10 à 33%. Ce nouveau schéma permettra sans doute de mieux cibler les instructions qui vont avoir besoin d'être optimisé.

Comme nous pouvons le constater sur la Figure 7, le code a été considérablement réduit et optimisé par le compilateur pour la fonction « build_index() ». En effet, une grande partie des instructions ont été supprimés car on utilise un passage de valeur par registres (plus rapide) et non plus par la pile. De plus, on utilise l'instruction *lea* spécialisé dans le calcul d'adresse. Finalement, il nous reste *lea* qui prend 15% de la fonction et le retour de la fonction qui prend 85% du temps.

Nous avons aussi testé les performances du programme avec « Clang/Low Level

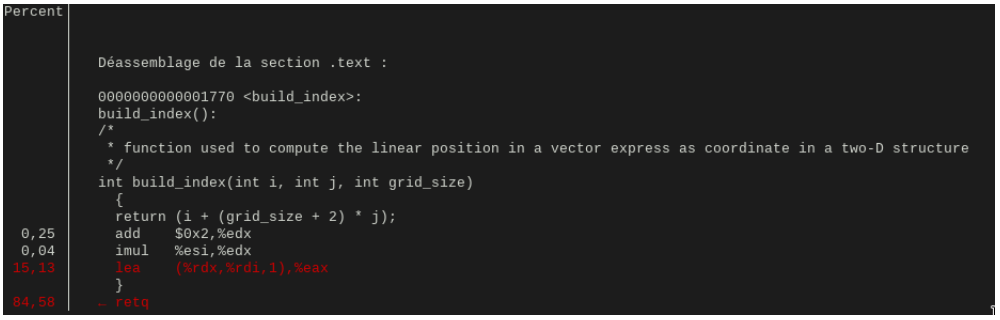


FIGURE 7 – Profilage de « build_index() » avec GCC en « -O2 »

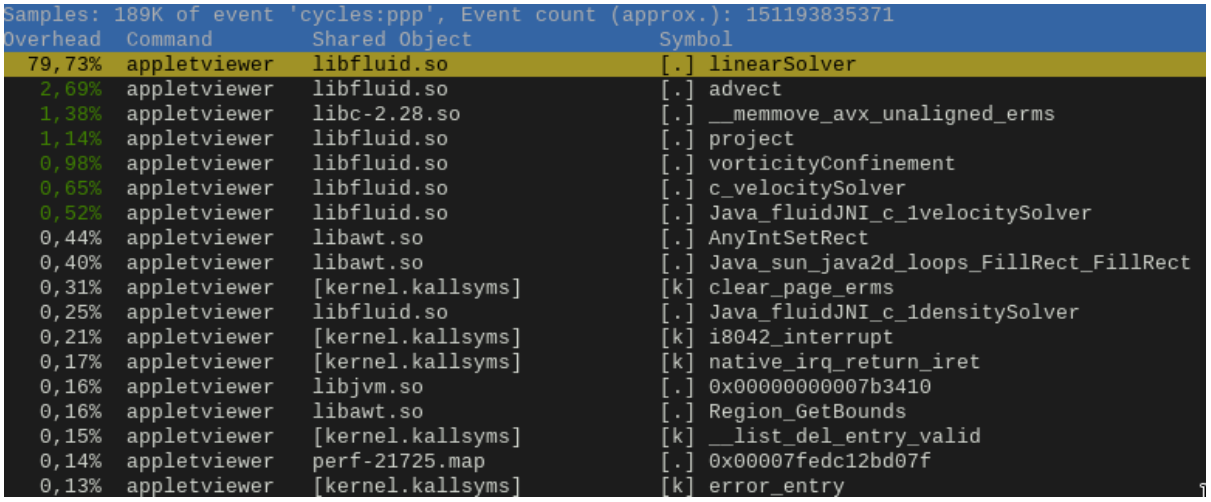


FIGURE 8 – Profilage global avec Clang en « -O3 »

Virtual Machine (LLVM) », et les flags d'optimisation les plus agressifs (« -03 »). Ci-dessous les images des résultats avec Clang et « -O3 ».

Illustré par la Figure 8, avec Clang et les options de compilation les plus agressives, on voit que le temps passés dans la fonction « linearSolver() » est passé de 57% avec GCC à 80%. Encore une fois, cela veux dire que les autres fonctions ont été encore plus radicalement minimisées. On notera aussi les fonctions « build_index() » et « build_index@plt » ont complètement été supprimés.

Par la Figure 9, nous observons que le code de « linearSolver() » a été extrêmement réduit en 5 instructions principales, introduites par Streaming SIMD Extensions (SSE). C'est vers ce genre de code que devrait converger nos optimisations manuelles.

Nous observons donc un net gain à utiliser Clang par rapport à GCC, qu'il soit visuel ou factuel. Cependant, pour nous permettre d'appliquer des optimisations vu en cours à la main, nous allons opter pour la version que produit GCC comme base de travail.

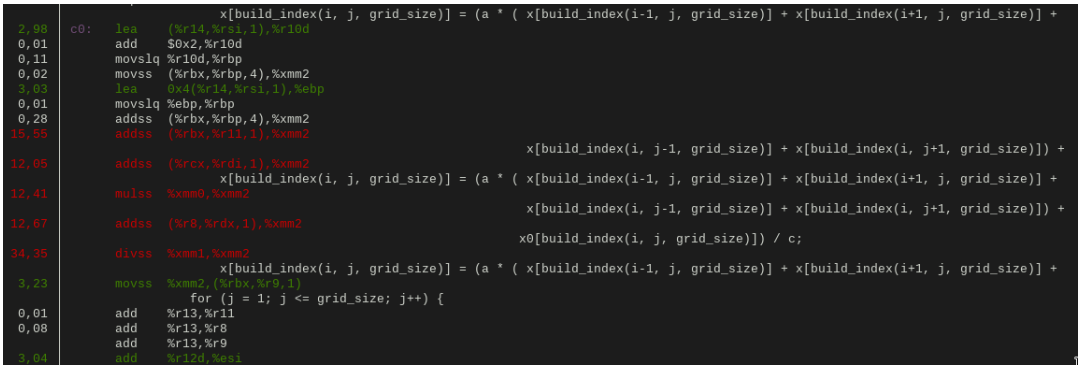


FIGURE 9 – Profilage de « linearSolver() » avec Clang en « -O3 »

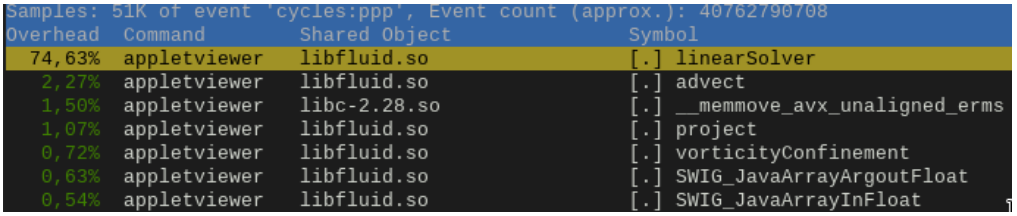


FIGURE 10 – Profilage global après inlining de la fonction « build_index() »

4.2 Inlining

Comme vu dans la section précédente, la fonction « build_index() » était très consommatrice en cycle (malgré l'optimisation du passage de paramètres par les registres plutôt que par la pile) à cause de 2 phénomènes :

1. L'appel de la fonction, qui passe par deux indirections du fait de la PLT, nous fait exécuter des instructions supplémentaires.
2. Les branchements dans le code ne favorise pas le principe de localité des caches.

Une optimisation bien connue consiste donc à *inliner* une fonction : cela revient à remplacer automatiquement lors de la compilation, dans le code généré du binaire, tous les appels à ladite fonction par le corps de la fonction. Cela à pour avantage de réduire considérablement le temps d'utilisation d'une fonction (plus de branchement ni de passage de paramètre). Cependant, cela peux grossir de manière drastique la taille du code si la fonction inliner est une longue fonction : ce qui n'est pas le cas ici, de toute évidence.

Pour procéder à l'inlining, cela est très simple puisqu'il suffit de l'indiquer au compilateur avec le mot clé approprié. On passe donc du code suivant (Listing 1) à la ligne 1 à celui de la ligne 2.

```
1 int build_index(int i, int j, int grid_size)
2 inline int build_index(int i, int j, int grid_size)
```

Listing 1 – Fonction « build_index() » avant et après inlining

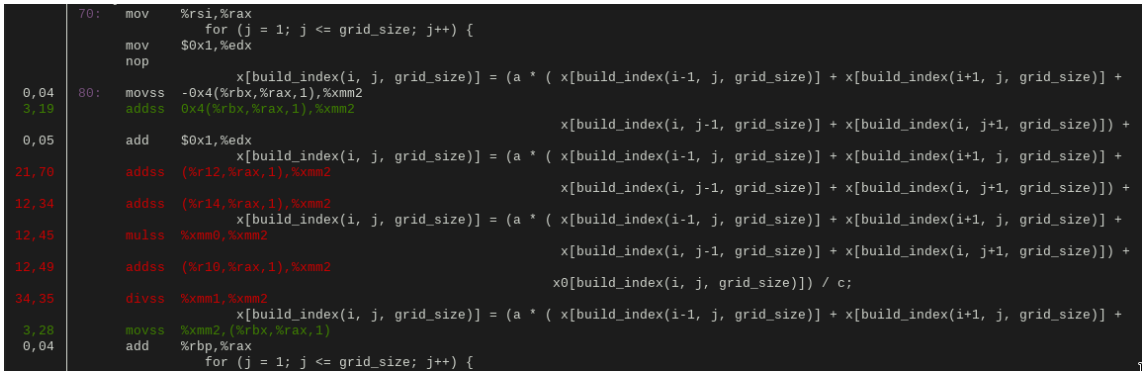


FIGURE 11 – Profilage de « linearSolver() » après inlining de la fonction « build_index() »

En termes de gain, la fluidité visuelle est grandement améliorée et tout de suite remarquable grâce à l'inlining. Par le profilage globale (Figure 10), on peut observer que les fonctions « build_index() » et « build_index@plt » ont été complètement supprimées et que l'on est passé d'un temps de 56% dans la fonction « linearSolver() » à maintenant 74%, ce qui est bon signe : il faut maintenant se concentrer sur l'optimisation de cette fonction.

Concernant le code de la fonction « linearSolver() », on observe dans la Figure 11 qu'il a été considérablement réduit et compacté, il ressemble maintenant fortement au code créer par Clang, décrit dans la section précédente.

4.3 Déroulage de boucle

TODO

4.4 Vectorisation

TODO

5 Conclusion

TODO

Acronymes

GCC GNU Compiler Collection
LLVM Low Level Virtual Machine
PIC Position-Independent Code
GOT Global Offset Table
PLT Procedure Linkage Table
AVX Advanced Vector Extensions
SSE Streaming SIMD Extensions