INSTITUT DES SCIENCES ET TECHNIQUES DES YVELINES

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

université PARIS-SACLAY

Projet Architecture des Ordinateurs – BASES

Réalisé par:

Iyed CHEBERLI

Mohamed Ilyess ELAJROUD

Encadré par:

Mohamed Salah IBNAMAR

Hugo BELLORE

Année universitaire 2018/2019

Introduction

Dans le cadre des études des méthodes d'optimisation et de calcul de haute performance, il nous est demandé d'améliorer les performances d'un code déjà existant en C et en Java utilisant SWIG.

Pour ce faire on va essayer d'utiliser des méthodes déjà étudiées au cours du module d'architecture des ordinateurs ainsi que nos connaissances.

Ce rapport comportera 4 parties, en premier lieu on va commencer par utiliser la bibliothèque openMP afin d'essayer de paralléliser notre code si c'est possible, ensuite par les optimisations fourni par le compilateur gcc de linux qui sont les niveaux d'optimisations avec l'option -O et --ffast-math qui permet d'accélérer les opérations sur les flottants et finalement on va essayer de modifier le code et utiliser la bibliothèque openBLAS optimisé par le calcul vectoriel.

Tout au long de nos améliorations on va essayer de faire des analyses de performances du programme par les outils fournis par linux et les outils open Source disponibles pour comparer de façon précises les optimisations apportés au code.

Sommaire

1. Environnement et méthodes de calculs de performance

On travaille sur une machine Ubuntu 16.04 LTS équipée d'un processeur Intel® Core™ i5-3210M CPU @ 2.50GHz (4 coeurs) et GeForce 610M/PCIe/SSE2 .

Pour le calcul de performance on a été confronté à plusieurs outils fournis par linux (Operf,perf,getticks,time) ainsi que d'autre outils open Source tel que maqao,likwid,VTune, google Benchmark .. .

Le choix s'est porté sur perf vu qu'il est beaucoup plus stable que "operf" et mieux documenté.

Il offre une panoplie d'options qui nous sont bien utile pour faire des calculs de performance précis comme le control de fréquence d'échantillonnage, un rapport bien détaillé du poids de chaque commande utilisée, …

Afin d'avoir des mesures précis on va se baser sur l'échantillonnage (sampling) qui consiste à faire une mesure toutes les **N** secondes, donc clairement, plus le programme est en exécution, plus on capture de samples et plus on en sait sur l'état d'exécution du code cible.

Pour cela on opté à une fréquence d'échantillonnage de 2000 samples/sec pour avoir une bonne précision sur toutes les optimisations qu'on va tester .

Dans notre cas, on ne possède pas un fichier binaire généré à partir de notre code mais plutôt un fichier html généré à partir de la connexion des bibliothèques C avec celles du Java grâce à SWING. D'ou la grande utilité de l'outil perf qui nous permet d'analyser un programme en cours d'exécution juste en lui fournissant son pid. Du coup, à chaque optimisation on va lancer l'appletviewer en faisant un make run et dans un autre terminal on lance la commande ps -a qui permet de nous donner un cliché sur les processus en cours d'exécution.

```
  PID TTY          TIME CMD
 3768 pts/18   00:00:00 dbus-launch
10993 pts/20   00:00:00 make
10994 pts/20   00:00:00 sh
10995 pts/20   00:33:41 appletviewer
11344 pts/18   00:00:00 ps
```

Ensuite on fourni à perf ce pid avec les options de notre test de performance.

```
chabroul@chabroul-K55VD:~$ sudo perf record -F 2000 --pid 10995
```

Finalement on aura à chaque fois un rapport détaillé des poids de chaque fonction de notre programme comme détaillé ci-dessous :

```
Samples: 72K of event 'cycles:ppp', Event count (approx.): 111269360632
Overhead  Command       Shared Object            Symbol
  15,25%  appletviewer  [kernel.kallsyms]        [k] syscall_return_via_sysret
  12,41%  appletviewer  libfluid.so              [.] linearSolver
   5,13%  appletviewer  [kernel.kallsyms]        [k] __schedule
   4,59%  appletviewer  [kernel.kallsyms]        [k] update_curr
   4,29%  appletviewer  [kernel.kallsyms]        [k] pick_next_task_fair
   3,78%  appletviewer  [unknown]                [k] 0xfffffe000008a01b
   3,45%  appletviewer  [kernel.kallsyms]        [k] do_syscall_64
   3,38%  appletviewer  [unknown]                [k] 0xfffffe000003201b
   2,83%  appletviewer  [unknown]                [k] 0xfffffe000000601b
   2,76%  appletviewer  [kernel.kallsyms]        [k] entry_SYSCALL_64_after_hwfr
   2,50%  appletviewer  [kernel.kallsyms]        [k] pick_next_entity
   2,31%  appletviewer  libc-2.23.so             [.] __sched_yield
   2,27%  appletviewer  [kernel.kallsyms]        [k] sys_sched_yield
   2,00%  appletviewer  libfluid.so              [.] build_index
   1,74%  appletviewer  [kernel.kallsyms]        [k] _raw_spin_lock
   1,71%  appletviewer  [kernel.kallsyms]        [k] yield_task_fair
   1,70%  appletviewer  [kernel.kallsyms]        [k] __calc_delta
   1,51%  appletviewer  [kernel.kallsyms]        [k] native_sched_clock
   1,47%  appletviewer  [kernel.kallsyms]        [k] update_min_vruntime
   1,37%  appletviewer  [kernel.kallsyms]        [k] cpuacct_charge
   1,36%  appletviewer  libfluid.so              [.] project
```

2. Parallélisation avec OPENMP

OpenMP est constitué d'un jeu de directives, d'une bibliothèque de fonctions et d'un ensemble de variables d'environnement qu'on peut facilement contrôler et modifier

OPEN MP permet de gérer :

- la création de threads légers
- le partage du travail entre les threads créés,
- les synchronisations (explicites ou implicites) entre tous les threads
- le statut des variables (privées ou partagées).

La simplicité de mise en oeuvre (la parallélisation ne dénature pas le code) , une seule version du code à gérer pour la version séquentielle et parallèle nous a amené à utiliser cette méthode pour améliorer les performances de notre code et de plus l'existence de plusieurs boucle for qui sont facilement parallélisables.

AVANT OPENMP

```
Samples: 31K of event 'cycles:uppp', Event count (approx.): 17144487031
Overhead  Command       Shared Object        Symbol
  36,19%  appletviewer  libfluid.so         [.] linearSolver
  28,22%  appletviewer  libfluid.so         [.] build_index
   6,00%  appletviewer  libfluid.so         [.] build_index@plt
   2,41%  appletviewer  libawt.so           [.] AnyIntSetRect
   1,92%  appletviewer  libfluid.so         [.] advect
   1,87%  appletviewer  libc-2.23.so        [.] __memcpy_sse2_unaligned
   1,55%  appletviewer  libfluid.so         [.] project
   1,40%  appletviewer  libfluid.so         [.] calculate_curl
   1,38%  appletviewer  [kernel]            [k] 0xffffffffbe200158
   1,38%  appletviewer  libfluid.so         [.] vorticityConfinement
   1,27%  appletviewer  libfluid.so         [.] SWIG_JavaArrayInFloat
   0,76%  appletviewer  libfluid.so         [.] addSource
   0,73%  appletviewer  libfluid.so         [.] setBoundry
   0,71%  appletviewer  libfluid.so         [.] SWIG_JavaArrayArgoutFloat
   0,63%  appletviewer  libfluid.so         [.] swap
   0,61%  appletviewer  [kernel]            [k] 0xffffffffbe2009e7
   0,53%  appletviewer  libfluid.so         [.] buoyancy
```

APRÈS OPENMP

```
Samples: 34K of event 'cycles:ppp', Event count (approx.): 76317765514
Overhead  Shared Object        Symbol
  26,40%  libfluid.so          [.] linearSolver._omp_fn.2
  15,35%  libfluid.so          [.] build_index
  13,73%  libgomp.so.1.0.0     [.] 0x0000000000011b27
   9,47%  libgomp.so.1.0.0     [.] 0x0000000000011b29
   8,50%  libgomp.so.1.0.0     [.] 0x0000000000011ce7
   5,20%  libgomp.so.1.0.0     [.] 0x0000000000011b2b
   4,90%  libgomp.so.1.0.0     [.] 0x0000000000011ce9
   2,44%  libgomp.so.1.0.0     [.] 0x0000000000011ceb
   1,66%  libfluid.so          [.] build_index@plt
   0,72%  libfluid.so          [.] advect
   0,70%  libfluid.so          [.] addSource._omp_fn.0
   0,66%  perf-7910.map        [.] 0x00007fc99526913b
   0,57%  libgomp.so.1.0.0     [.] 0x0000000000011b2d
   0,55%  [kernel]             [k] native_write_msr
   0,55%  libc-2.23.so         [.] __memcpy_sse2_unaligned
   0,54%  [kernel]             [k] ioread32
   0,50%  libawt.so            [.] BufImg_GetRasInfo
   0,47%  libawt.so            [.] AnyIntSetRect
```

Interprétation

La parallélisation des boucles for a en effet réduit le poids des fonctions linearSolver et build_index et a été réparti sur plusieurs threads qui dans notre cas sont 4.

Mais cette amélioration dépend de l'architecture du système et l'environnement du travail (Processeur multi threads).

3. Utilisation du flag -ffast-math

-ffast-math fait beaucoup plus que simplement briser la stricte conformité IEEE.

Elle permet la réorganisation des instructions à quelque chose qui est mathématiquement le même mais pas exactement le même en virgule flottante.

Aucun contrôle pour NaN (ou zéro) sont faites en place où ils auraient des effets néfastes. Il est simplement supposé que cela n'arrivera pas.

Il permet des approximations réciproques pour division et racine carrée réciproque.

Il désactive le zéro signé le code suppose que le zéro signé n'existe pas, même si la cible le supporte et l'arrondi mathématique

Il génère du code qui suppose qu'aucune interruption de matériel ne peut se produire en raison de la signalisation mathématiques

Vu l'existence de plusieurs calculs mathématiques et matriciels, on a vu que cette option de compilation pourrait être utile afin d'améliorer les performances de notre programme.

Avant –ffast-math

```
Samples: 31K of event 'cycles:uppp', Event count (approx.): 17144487031
Overhead  Command       Shared Object        Symbol
  36,19%  appletviewer  libfluid.so         [.] linearSolver
  28,22%  appletviewer  libfluid.so         [.] build_index
   6,00%  appletviewer  libfluid.so         [.] build_index@plt
   2,41%  appletviewer  libawt.so           [.] AnyIntSetRect
   1,92%  appletviewer  libfluid.so         [.] advect
   1,87%  appletviewer  libc-2.23.so        [.] __memcpy_sse2_unaligned
   1,55%  appletviewer  libfluid.so         [.] project
   1,40%  appletviewer  libfluid.so         [.] calculate_curl
   1,38%  appletviewer  [kernel]            [k] 0xffffffffbe200158
   1,38%  appletviewer  libfluid.so         [.] vorticityConfinement
   1,27%  appletviewer  libfluid.so         [.] SWIG_JavaArrayInFloat
   0,76%  appletviewer  libfluid.so         [.] addSource
   0,73%  appletviewer  libfluid.so         [.] setBoundry
   0,71%  appletviewer  libfluid.so         [.] SWIG_JavaArrayArgoutFloat
   0,63%  appletviewer  libfluid.so         [.] swap
   0,61%  appletviewer  [kernel]            [k] 0xffffffffbe2009e7
   0,53%  appletviewer  libfluid.so         [.] buoyancy
```

Aprés --ffast-math

```
Samples: 35K of event 'cycles:uppp', Event count (approx.): 19578366482
Overhead  Command       Shared Object        Symbol
  36,94%  appletviewer  libfluid.so         [.] linearSolver
  28,51%  appletviewer  libfluid.so         [.] build_index
   5,81%  appletviewer  libfluid.so         [.] build_index@plt
   2,39%  appletviewer  libawt.so           [.] AnyIntSetRect
   1,90%  appletviewer  libfluid.so         [.] advect
   1,66%  appletviewer  libc-2.23.so        [.] __memcpy_sse2_unaligned
   1,65%  appletviewer  libfluid.so         [.] project
   1,40%  appletviewer  libfluid.so         [.] vorticityConfinement
   1,39%  appletviewer  [kernel]            [k] 0xffffffffbe200158
   1,26%  appletviewer  libfluid.so         [.] SWIG_JavaArrayInFloat
   1,21%  appletviewer  libfluid.so         [.] calculate_curl
   0,81%  appletviewer  libfluid.so         [.] addSource
   0,78%  appletviewer  libfluid.so         [.] setBoundry
   0,69%  appletviewer  libfluid.so         [.] swap
   0,65%  appletviewer  libfluid.so         [.] SWIG_JavaArrayArgoutFloat
   0,56%  appletviewer  [kernel]            [k] 0xffffffffbe2009e7
```

Interprétation

Malgré l'existence de plusieurs opérations sur les flottants dans notre programme --ffast-math semble selon les résultats du rapport fourni par perf ne pas apporter des résultats améliorés par rapport au code original. Cette option est donc non envisageable dans notre cas.

4. Compilation avec l'option -O3

L'option -O contrôle le niveau global d'optimisation. Ceci rend le temps de compilation quelque peu plus long, et peut nécessiter plus de mémoire, en particulier si on augmente le niveau d'optimisation.

O3 C'est le plus haut niveau d'optimisations possibles. Il active des optimisations qui sont coûteuses en terme de temps de compilation et d'usage de la mémoire.

SANS –O3

```
Samples: 31K of event 'cycles:uppp', Event count (approx.): 17144487031
Overhead  Command       Shared Object        Symbol
  36,19%  appletviewer  libfluid.so         [.] linearSolver
  28,22%  appletviewer  libfluid.so         [.] build_index
   6,00%  appletviewer  libfluid.so         [.] build_index@plt
   2,41%  appletviewer  libawt.so           [.] AnyIntSetRect
   1,92%  appletviewer  libfluid.so         [.] advect
   1,87%  appletviewer  libc-2.23.so        [.] __memcpy_sse2_unaligned
   1,55%  appletviewer  libfluid.so         [.] project
   1,40%  appletviewer  libfluid.so         [.] calculate_curl
   1,38%  appletviewer  [kernel]            [k] 0xffffffffbe200158
   1,38%  appletviewer  libfluid.so         [.] vorticityConfinement
   1,27%  appletviewer  libfluid.so         [.] SWIG_JavaArrayInFloat
   0,76%  appletviewer  libfluid.so         [.] addSource
   0,73%  appletviewer  libfluid.so         [.] setBoundry
   0,71%  appletviewer  libfluid.so         [.] SWIG_JavaArrayArgoutFloat
   0,63%  appletviewer  libfluid.so         [.] swap
   0,61%  appletviewer  [kernel]            [k] 0xffffffffbe2009e7
   0,53%  appletviewer  libfluid.so         [.] buoyancy
```

AVEC -O3

```
Samples: 32K of event 'cycles:uppp', Event count (approx.): 17870640768
Overhead  Command       Shared Object        Symbol
  37,79%  appletviewer  libfluid.so         [.] linearSolver
  28,12%  appletviewer  libfluid.so         [.] build_index
   5,89%  appletviewer  libfluid.so         [.] build_index@plt
   2,49%  appletviewer  libawt.so           [.] AnyIntSetRect
   2,03%  appletviewer  libfluid.so         [.] advect
   1,64%  appletviewer  libc-2.23.so        [.] __memcpy_sse2_unaligned
   1,63%  appletviewer  libfluid.so         [.] project
   1,39%  appletviewer  libfluid.so         [.] vorticityConfinement
   1,38%  appletviewer  [kernel]            [k] 0xffffffffbe200158
   1,27%  appletviewer  libfluid.so         [.] SWIG_JavaArrayInFloat
   1,22%  appletviewer  libfluid.so         [.] calculate_curl
   0,83%  appletviewer  libfluid.so         [.] addSource
   0,79%  appletviewer  libfluid.so         [.] setBoundry
   0,68%  appletviewer  libfluid.so         [.] SWIG_JavaArrayArgoutFloat
   0,66%  appletviewer  libfluid.so         [.] swap
   0,55%  appletviewer  libfluid.so         [.] buoyancy
   0,53%  appletviewer  [kernel]            [k] 0xffffffffbe2009e7
```

Interprétation

Compiler avec -O3 ne garantit pas une amélioration de la performance. En réalité, cela ralentit un peu l'exécution et cela à cause des binaires plus volumineux qui réclament plus de mémoire. De plus cette option est réputé casser de nombreux paquets. C'est pourquoi utiliser -O3 n'est pas recommandé dans notre cas.

5. Vectorisation et Modification de la fonction LinearSolver

5.1. LinearSolver

On va essayer de limiter l'appel à la fonction Build_index qui d'après l'outil perf prenait 28% du travail,pour obtenir le code suivant :

```
void linearSolver(int b, float* x, float* x0, float a, float c, float dt, int grid_size)
  {
  int i,j,k;
  int pas = grid_size + 2;
  int pos;
  c=1./c ;
  for (int k =0; k < 20; k++)
    {
      for (int i = 1; i <= grid_size ; i++)
      {
        pos = build_index(1, i, grid_size );
        for (int j = 1; j <= grid_size ; j++)
        {
          x[pos] = ( a*( x[pos-1] + x[pos+1] + x[pos - pas] + x[pos + pas] )  + x0[pos] ) * c;
          pos++;
        }
      }
    setBoundry(b, x, grid_size);
    }


  }
```

En premier lieu on a modifié la division par la variable 'c' à l'intérieur des 3 boucles car la division de flottant est plus coûteuse que la multiplication. Ainsi, le fait de pipeliner deux multiplication à l'intérieur des boucles est plus facile et plus optimale d'être interprété par le processeur que pipeliner une multiplication et une division.

Après, selon l'outil perf la fonction build_index prenait 28% du travail vu le nombre énorme d'appel à cette fonction à l'intérieur de linearSolver, du coup fallait trouver une formule adéquate de calcul de position vu l'existence d'une relation entre les indices qu'on doit traiter.

5.2. Vectorisation avec BLAS

La bibliothèque Blas fournit des opérations de bas niveau pour la manipulation des vecteurs et des matrices. Elle est subdivisée en trois parties, baptisées «level 1», «level 2», «level 3», qui prennent en charge respectivement les opérations vecteur-vecteur, vecteur-matrice, et matrice-matrice.BLAS est devenu un standard en calcul haute performance et utiliser dans de nombreuses librairies de plus haut niveau (LAPACK, ScaLAPACK, ...) .

Dans notre cas on a va utiliser OpenBlas. Cette librairie est inspirée de GotoBlas2 (Texas instruments) et qui a apporté plusieurs améliorations (en terme de performances et de mémoire) par rapport aux premiers algorithmes BLAS .

A noter : La bibliothèque OpenBlas n'est pas installé par défaut sur Ubuntu, vous pouvez vous référer au readme joint avec le projet.

BLAS niveau 1 : opérations sur des vecteurs

- addition de 2 vecteurs,
- produit scalaire,
- echelonnage du vecteur ..

Les fonctions qu'on va utiliser dans notre modification de code sont celles fournies par la bibliothéque cblas.h .

```
#include <cblas.h>
```
All functions are prefaced by `cblas_`

Level 1 BLAS: vector, $O(n)$ operations

precisions	name	(size arguments	)	description	equation
s, d, c, z	axpy	(n, alpha, x, incx, y, incy	)	update vector	$y = y + \alpha x$
s, d, c, z, cs, zd	scal	(n, alpha, x, incx	)	scale vector	$y = \alpha y$
s, d, c, z	copy	(n, x, incx, y, incy	)	copy vector	$y = x$
s, d, c, z	swap	(n, x, incx, y, incy	)	swap vectors	$x \leftrightarrow y$

Lors de nos premiers tests de performances, on a remarqué que ces deux fonctions consomment une quantité non négligeables d'échantillon. On note bien que setBoundry appelle build_index plus de (16 × grid_size fois) ce qui augmente les echantillons niveau build_index. C'est pour cela on opté à vectoriser ces deux fonctions.

Dans notre cas , on a considéré qu'un vecteur est un tableau de floats .

Ci-dessous les optimisations apportées sur les deux fonctions :

la lettre s de cblas_saxpy indique que toutes les opérations sont faites sur des floats .

```
void addSource(float *x, float *x0, int vector_size, float factor)
  {
  int i;
  //y=y+alpha*x
  cblas_saxpy(vector_size, factor, x0, 1, x, 1);
  }
```

```
void setBoundry(int b, float* x, int grid_size)
  {
  int step=grid_size + 2;
  if (b==1)
  {
    build_vect2(x,grid_size,step+1,step,-1,x,step,step);
    build_vect2(x,grid_size,1+step,step,-1,x,(grid_size+1)*step,step);
  }
  else{
    build_vect2(x,grid_size,step+1,step,1,x,step,step);
    build_vect2(x,grid_size,1+step,step,1,x,(grid_size+1)*step,step);
  }
  if (b==2)
  {
    build_vect2(x,grid_size,1+step,1,-1,x,1,1);
  build_vect2(x,grid_size,((grid_size)*step)+1,1,-1,x,((grid_size+1)*step)+1,1);
  }
  else
  {
    build_vect2(x,grid_size,1+step,1,1,x,1,1);
  build_vect2(x,grid_size,((grid_size)*step)+1,1,1,x,((grid_size+1)*step)+1,1);

  }
```

la fonction build_vect nous permet de calculer un vecteur res a partir d'un tableau x et un coefficient coef .

deb_res et deb représente le décalage (d'ou commence le traitement ou le calcul de vecteur)

et step et step_res représente le pas de chacun des vecteurs

```
  void build_vect2(float* x,int vector_size,int deb,int step,int coef,float* res,int deb_res,int step_res)
//entree vect sortie un autre vect
  {
    cblas_scopy(vector_size,(x+deb),step,(res+deb_res),step_res);
    if (coef != 0)
      //( n,  alpha, x, incx)
      cblas_sscal(vector_size,coef,(res+deb_res),step_res);
  }
```

Avant les optimisations :

```
Samples: 31K of event 'cycles:uppp', Event count (approx.): 17144487031
Overhead  Command       Shared Object        Symbol
  36,19%  appletviewer  libfluid.so         [.] linearSolver
  28,22%  appletviewer  libfluid.so         [.] build_index
   6,00%  appletviewer  libfluid.so         [.] build_index@plt
   2,41%  appletviewer  libawt.so           [.] AnyIntSetRect
   1,92%  appletviewer  libfluid.so         [.] advect
   1,87%  appletviewer  libc-2.23.so        [.] __memcpy_sse2_unaligned
   1,55%  appletviewer  libfluid.so         [.] project
   1,40%  appletviewer  libfluid.so         [.] calculate_curl
   1,38%  appletviewer  [kernel]            [k] 0xffffffffbe200158
   1,38%  appletviewer  libfluid.so         [.] vorticityConfinement
   1,27%  appletviewer  libfluid.so         [.] SWIG_JavaArrayInFloat
   0,76%  appletviewer  libfluid.so         [.] addSource
   0,73%  appletviewer  libfluid.so         [.] setBoundry
   0,71%  appletviewer  libfluid.so         [.] SWIG_JavaArrayArgoutFloat
   0,63%  appletviewer  libfluid.so         [.] swap
   0,61%  appletviewer  [kernel]            [k] 0xffffffffbe2009e7
   0,53%  appletviewer  libfluid.so         [.] buoyancy
```

Après les optimisations de LinearSolver, setBoudary et addSource :

```
Samples: 33K of event 'cycles:ppp', Event count (approx.): 21206618402
Overhead  Shared Object            Symbol
  51,27%  libfluid.so              [.] linearSolver
   8,88%  libfluid.so              [.] advect
   4,17%  [kernel]                 [k] nmi
   3,80%  libfluid.so              [.] build_index
   3,55%  libfluid.so              [.] project
   2,80%  libfluid.so              [.] SWIG_JavaArrayArgoutFloat
   2,42%  libawt.so                [.] AnyIntSetRect
   2,38%  libfluid.so              [.] swap
   2,14%  libfluid.so              [.] calculate_curl
   2,13%  libfluid.so              [.] build_index@plt
   1,57%  libc-2.23.so             [.] __memcpy_sse2_unaligned
   1,37%  libawt.so                [.] Java_sun_java2d_loops_FillRect_FillRect
   0,93%  libfluid.so              [.] vorticityConfinement
   0,72%  libfluid.so              [.] SWIG_JavaArrayInFloat
   0,61%  libjvm.so                [.] jni_GetObjectField
   0,60%  libjvm.so                [.] _ZN14JNIHandleBlock15allocate_handleEP7oo
   0,48%  libjvm.so                [.] _ZN10HandleMark15pop_and_restoreEv
```

Interprétation

-on remarque que la vectorisation en utilisant BLAS 1 ainsi que l'optimisation faite sur build_index niveau linear solver nous a permis de diminuer considérablement la charge sur le CPU . il s'est concentré principalement sur le linearSolver avec plus de 51% des échantillons relevés parmi 33K échantillons .

Après les optimisations apportés on remarque aussi que setboundry et addsource n'existe quasiment plus (un pourcentage inférieur à 0.5%) ainsi que build_index qui est passé de 28% à 3 % d'échantillons.

6. Axe d'améliorations

Le compilateur ICC d'INTEL est l'un des meilleur alternative de GCC qui permet principalement l'optimisation et la vectorisation automatique . ICC est optimisé principalement pour les processeurs intel .il permet une meilleur gestion niveau cache de données et répartition de tâches sur les cœurs du processeur.

Remarque : on a pu compiler avec ICC et générer des rapports d'optimisations, mais on a pas pu lancé appletviewer suite à des problèmes de bibliothèques partagés et de version JAVA et swig.

On a généré 3 rapports en html avec les différentes optimisations faite avec ce compilateur. On a préféré ne pas les évoquer dans nos optimisations vu qu'on n'a pas pu s'assurer qu'ils fonctionnent sur la appletviewer.

La vectorisation et le calcul matriciel sur les GPUs sont de plus en plus démocratisées (malgré des restrictions hardware chez de grands constructeurs comme Nvidia) , on envisage travailler sur ces unités gpu afin d'optimiser la parallélisation de différentes opérations simples ainsi que la vectorisation des opérations sur les grands tableaux tels que d,u,v,curl...

7. Conclusion

Tout au long de ce projet, on était confronté à plusieurs difficultés dont :

- La gestion des bibliothèques SWIG,OpenMP,OpenBLAS,..
- L'utilisation de l'outil de mesure de performance perf et la mise en place des métriques.
- La complexité du code et sa vectorisation.

Pour conclure, selon les manipulations apportées au code, on a opté pour l'utilisation de OpenBLAS avec des modifications au niveau de la fonction linearSolver qui selon le rapport fourni par perf est la solution la plus performante. Néanmoins, on envisage de paralléliser certains bout de code avec OpenMP et travailler sur des matrices plutôt que des vecteurs, c'est-à-dire BLAS niveau 3, qui nous donnera une meilleur maniabilité des jeux de données.