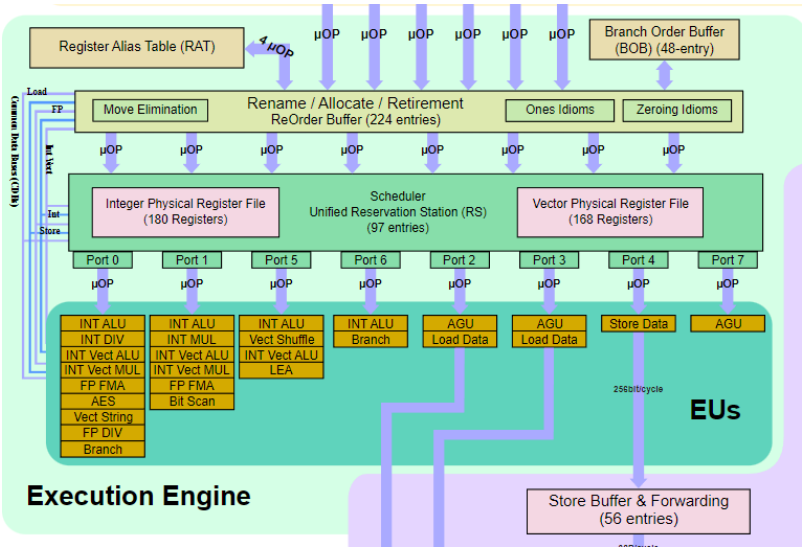
Dans ce rapport de benchmark, on présente des mesures de benchmark pour différentes méthodes de matrice solvers codés en langage C, compilés avec clang et gcc, en utilisant les libraries de math et de blas avec différents flags d'optimisation. On compare les différentes variations de toutes les combinaisons de ces choix par rapport à leurs résultats de bandwidth en MiB/s et on examine les deviations standards liés à ces résultats.

La machine utilisée lors de la prise des résultats possède une CPU de modèle Intel Core i5-8250 U CPU @ 1.60 GHz avec une architecture x86_64 basée sur les 8 cœurs et 3 lignes de cache, L1 est 32K, L2 est 256K et L3 est 6144K, l'information détaillée est fournie dans les fichiers **lscpu.txt, cpuinfo.txt et cat_proc_cpuinfo.txt.**

Pour avoir des résultats assez stables, on a pris les mesures quand la machine était branchée surle secteur, on a utilisé un CPU governor qui est fourni par intel pstate et on a mis tous les CPUs en mode performance. Les informations détaillées sur les données de CPU governor sont fournies dans le fichier **cible**.



La raison pour laquelle on a choisi de faire les mesures avec un code qui calcule des différentes méthodes de multiplication de matrices, la réduction de matrice et le produit scalaire est que chaque méthode mentionnée met du stress sur un aspect différent du CPU. Par exemple, la méthode de réduction met le stress sur les opérations load, store et addition. Le produit scalaire joue quant à lui sur les opérations de load, store, multiplication et addition ; ici, il faut faire bien attention aux portes 0 et 1, qui donnent la liberté de faire une multiplication et une addition en un seule cycle grâce à FMA. La méthode unroll x4 de dgemm met du stress sur les mêmes opérations que le produit scalaire, mais avec une seule différence : cette fois-ci, ces opérations sont censées être exécutées en bloc de 4, donc cela met aussi du stress sur la durée d'attente d'opération. La méthode IJK met du stress sur les mêmes opérations encore, mais avec un nombre d'accès au mémoire maximale, ce qui demande plus de load et store.

Pour calculer les résultats de benchmark utilise RDTSC instruction pour mesurer la performance pour chaque itération. Cette instruction retourne le nombre de cycles complétés lors de l'exécution de fonction.*On répète l'exécution de la fonction r fois pour construire une échantillon de données. Il faut préciser que r doit être plus grand que 31 à cause de la méthode de calcul choisie pour calculer standard variation, celle-ci doit être inférieure à %5 pour qu'on

puisse faire confiance à nos données. Elle nous donne la distance moyenne entre la moyenne et les éléments de l'échantillon. Lors des benchmarks, on a pris r égale à 35.

La sortie de l'exécution des mesures de benchmark fournit l'information sur KiB, MiB, GiB produits par la fonction simulée, la taille de matrice, le nombre d'échantillons, le minimum valeur de cycles elapsed, le maximum valeur de cycles elapsed, la valeur moyenne de cycles elapsed, la déviation pour la méthode de réduction et produit scalaire, la déviation standard et le bandwidth en MiB/s.

On a pris les mesures de benchmark pour différentes méthodes de multiplication de matrices.

- Première méthode IJK: c'est l'algorithme avec la complexité en nombre d'accès au mémoire le plus haut, O(n^3). La boucle interne est le produit scalaire et l'accès à lamémoire se fait sur les lignes. Quand on parle de l'accès à lamémoire, il faut se rappeler que le langage C favorise row-major layout pour accéder aux éléments de la matrice.
- Deuxième méthode IKJ : la boucle interne est une AXPY, donc la complexité en nombre d'accès au mémoire est plus basse que celle de IJK. L'accès à la mémoire se fait sur les colonnes.
- Troisieme methode IEX : On a encore un AXPY dans la boucle interne, mais cette fois-ci l'accès à l'array se fait en dehors de la boucle interne, ce qui diminue la complexité en nombre d'accès à lamémoire qui se fait sur les colonnes.
- Quatrième méthode UNROLL 4: La boucle interne est encore un AXPY mais cette fois-ci l'accès à la mémoire se fait par les blocs de 4. En déroulant la boucle, on veut optimiser le temps d'exécution de boucle aux dépens de la taille de fichier binaire.
- Cinquième méthode CBLAS : Dans cette méthode, on fait appel à la méthode cblas dgemm de la librairie cblas.
- Sixième méthode UNROLL x8 : Même chose que quatrième méthode sauf qu'ici, on accède à la mémoire par les blocs de 8.
- La méthode de réduction : il s'agit de réduire l'array à une seule valeur en accumulant. La complexité en nombre d'accès au mémoire est linéaire, on fait seulement l'addition.
- La méthode de produit scalaire : on fait le produit scalaire, et on utilise la possibilité de faire une multiplication et une addition en un seul cycle, ce qui nous amène de la complexité en nombre accès en mémoire linéaire.

On profite des flags d'optimisation fournis par les compilateurs pour comparer les performances des différentes méthodes de matrice multiplication compilées avec différents flags.*

- Avec O1: le gcc et le clang essayent de réduire la taille de code et le temps d'exécution. Le manuel de gcc indique que l'optimisation de la compilation peut prendre un peu plus de temps et plus de mémoire pour les grandes fonctions.
- Avec O2, le gcc essaye d'optimiser encore plus, mais il n'augmente pas le déroulage des loop ou l'inlining. O2 hérite les flags d'optimisations fournis par O et O1, et pour clang aussi l'héritage marche de la même manière que gcc. selon les versions, clang peut ajouter : -vectorize-loops, -vectorize-slp flags aussi.

- O3 a aussi hérité des flags d'optimisation précédentes, mais il fournit en plus : -finline-functions, -funswitch-loops and -fgcse-after-reload options pour gcc. et pour clang les options -callsite-splitting -argpromotion sont ajoutées selon les versions.
- Enfin, avec Ofast le compiler fait tout pour obtenir le programme le plus optimisé possible en négligeant la précision numérique. Il fait l'optimisation agressive qui peut être non conforme aux standards de langage.

Pour obtenir ces résultats le plus vite possible et avec le moins d'erreur humaine, comme des fautes de frappes, et pour bien organiser les données obtenues, on a profité d'un script qui :

- prend les valeurs nécessaires par l'utilisateur, comme la taille de matrice, la quantité d'échantillon des samples, le nom de compiler, et le flag d'optimisation voulu.
- fait les changements nécessaires dans les Makefile des programmes.
- met les CPU à la fréquence voulue, et colle le programme sur le cœur 1 de la machine.
- rend les données en un format facilement utilisable par gnuplot.
- fait l'organisation des fichiers dans les files.

Dans la suite, on examinera les graphes pour :
les methodes de dgemm, reduction et produit scalaire compliés avec gcc et clang, avec des differents flags d'optimisation et leurs comparaisons.

Clang avec dgemm avec le flag d'optimisation O3:
flags_clang_2x2.png ; 4eme case
On voit que CBLAS a le bandwidth le plus élevé, on continuera de voir ce trend dans les autres graphes aussi. L'efficacité d'accéder aux sub-level programmes montre à quels points ils sont forts.
On voit que IJK a le bandwidth le plus bas, on peut lier cela au fait qu'il possède un complexité en nombre d'accès à la mémoire la plus élevé.
On voit que Unrollx4 et Unrollx8 ne donnent pas le résultat attendu, on attendait un trade-off entre la taille de fichier binaire et le bandwidth..

Gcc avec dgemm avec le flag d'optimisation O3:
flags_gcc_2x2.png ; 4eme case
Même raisonnement que la même version avec clang**.**

La comparaison entre gcc et clang avec dgemm pour O3:
flags_gcc_vs_clangs_2x2.png ; 4eme case
On peut constater que gcc est plus efficace avec la méthode unroll x4 alors que ce n'est plus le cas pour unroll x8.
Pour IKJ, on voit que clang est plus efficace que gcc et le bandwidth de IKJ compilé avec clang est vraiment proche au bandwidth de IEX compilé avec clang. En connaissant la différence entre les deux méthodes, on peut se demander si clang a pris la liberté de faire sortir array en dehors de la boucle interne quand il compilait IKJ, ce qui constitue la difference principale entre ces deux méthodes de multiplication.

Les méthodes de dgemm, reduc et dotprod de taille 60 et de taille 100 compilées avec gcc avec le flag d'optimisation O3:
gcc_dif_size_flagsO3.png
On peut observer le trend que le bandwidth diminue quand la quantité de données augmente, ce qui est le résultat attendu, vu que le nombre d'acces à memoire augmente quand la taille de données augmentent.

Les méthodes de dgemm reduc et dotprod de taille 60 et de taille 100 compilées avec gcc avec le flag d'optimisation O3:
clang_dif_size_flagsO3.png
On voit le meme trend obtenu avec le gcc compiler sur les bandwidth. Par contre on constate que les deviations standarts varient, qui peut etre lié à plusieurs raisonnes, on ne peut pas faire une hypothese en regardant seulement ces deux stats.

Les méthodes de dgemm reduc et dotprod de taille 60 et de taille 100 compilées avec gcc vc clang avec le flag d'optimisation O3:
gcc_clang_dif_size_flagsO3.png
Sur l'ensemble, on peut observer que clang a donné les meilleurs résultats de bandwith pour chaque taille de données.

*A partir de là; Il faut bien retenir que les mesures obtenues pour la methode de reduction compilé avec gcc ne sont pas stables à part de Ofast. Et les mesures prises pour la meme fonction avec clang sont assez stables à part de Ofast. La raison de résultats inconsistents peut etre causé par notre methode de benchmark qui est lié au RDTSC, cette instruction est vraiment dependant à la frequence de CPU et on peut seulement obtenir les resultats stables si la cible mesuré prend au moins 500 cycles.***

Gcc avec dgemm avec differents flags d'optimisation examinés séparément:
flags_gcc_2x2.png ;
On peut observer que O1 suis le meme trend que O2, et O3 suis le meme trend que Ofast.
Cela peut etre lié au fait que les flags d'optimisations sont herités.
On peut aussi constater que IEX est considerablement plus performante que IKJ quand elle est compilée avec O3 et Ofast, cela peut etre lié au fait que les flags de vectorisation sont inclus dans O3 et Ofast mais pas dans O1 et O2.

Clang avec dgemm avec différents flags d'optimisation examinés séparément:
flags_clang_2x2.png
On voit pas une IEX plus performante que IKJ comme on voit dans la version compliée avec gcc avec les flags d'optimisations de O3 et Ofast.

Clang avec dgemm avec la comparaisons entre les differents flags
clang_flags.png

Il est intéressant de constater que pour unroll x4 toutes les optimisations donnent des résultats proches, on peut alors se dire que même avec les flags d'optimisations, on n'arrive pas à obtenir un bandwidth plus grand. Donc pour notre cas, il faut se demander si cela vaut le coût de l'implémenter aux dépens de la taille de fichier binaire.

Gcc avec dgemm avec differents flags d'optimisation examinés séparément
flags_gcc_2x2.png
On voit que Ofast et O3 suivent un même trend et que O1 et O2 en suivent une autre. Comme on le sait, les flags hérités de leurs précédents, et O3 et Ofast possèdent les flags d'optimizations de inlining, loop switch et unrolling. Ces trois flags peuvent être la cause du fait qu'ils suivent le même trend.

Gcc avec dgemm avec la comparaisons entre les différents flags
gcc_flags.png
le trend qu'on a vu dans la graphe précédente se montre à part d'une seule méthode qui est IJK, si on regarde les flags d'optimisation de Ofast, on peut constater que Ofast possède 3 flags d'optimisation plus que O3, ce qui peut être la raison pour laquelle on a un meilleur bandwidth.

Gcc et Clang avec dgemm avec la comparaison entre les mêmes flags
flags_gcc_vs_clang_2x2.png
Si on fait la comparaison seulement pour le flag O3, on voit que gcc est plus performante que clang pour Unroll x4, alors que ce n'est pas le cas pour Unroll x8.
Et on voit que le binaire fournit par clang pour IKJ est assez performante que celui fournit par IEX, mais gcc ne montre pas le meme trend de mesure pour ces deux fonctions. Le fait de separer le variable independant de boucle interne joue sur la performance de gcc pour cette fonction.

Gcc, Clang, Gcc et Clang avec dot product avec la comparaison entre les différents flags
dotprod_gcc_vs_clang_flags.png
Produit scalaire, comme expliqué précedemment, profite du fait qu'on peut faire multiplication et addition en meme temps. Dans ces graphes on peut constater que O1, O2 et O3 pour chaque compiler produit des bandwidth qui sont assez proches. Il y a seulement Ofast de chaque compilateur qui differe. En meme temps on peut voir que Ofast de clang est plus performante que celui de gcc.

Gcc, Clang, Gcc et Clang avec réduction avec la comparaison entre les différents flags
reduc_gcc_vs_clang_flags.png
On ne peut pas faire confiance à plus parts de données qui sont fournis par la fonction de reduction. On ne peut pas faire une comparaison entre les differents compilateurs ici.

Gcc, Clang, Gcc et Clang avec chaque méthode de dgemm avec la comparaison entre les différents flags
/gcc_vs_clang_flags_6_types

Gcc vs Clang avec differents flags d'optimisation pour IJK:

gcc_vs_clang_flags_IJK.png

Dans cette partie on voit que la deviation standart pour gcc avec O2 est %12.7, avec O3 est %5.177 donc on ne peut dire les mesures pour cette benchmark ne sont pas stables. Et avec O1 la déviation standart est %3.5, qui peut etre considéré comme dans la limite. On obtient une mesure de benchmark stable pour cette methode seulement avec Ofast, qui est inferieure à %1. Cette difference de deviation standart peut etre liée au fait que c'est la methode qui demande le plus grand nombre d'acces à memoire, par contre on ne voit pas le meme type de comportement avec les binaires produit par clang avec les quatres flags d'optimization.

Gcc vs Clang avec differents flags d'optimisation pour Unrollx8:

gcc_vs_clang_flags_Unrollx8.png

Avec clang Ofast on voit que la deviation standart est plus grand que %5, donc on peut dire que nos mesures ne sont pas assez stables pour cette methode avec clang Ofast.

Avec gcc O1 on obtient une deviation standard à la limite acceptable, %4.6 et avec le flag Ofast on obtient %5.6.

Gcc vs Clang avec differents flags d'optimisation pour IKJ:

gcc_vs_clang_flags_IKJ.png

Chaques mesures prises pour cette methode possedent une deviation standart inferieure à %1 à part le binaire produit par clang avec O2, qui est %16.33, donc pas confiable.

O1 et O2 de gcc sont completement detruit

Gcc vs Clang avec differents flags d'optimisation pour IEX:

gcc_vs_clang_flags_IEX.png

Ici on constate que la deviation standart pour gcc avec O2 et avec clang O1 sont plus grande que %5, donc nos mesures ne sont pas stables.

Ici on peut voir que les binaires produits par le compiler gcc avec O1 et O2 sont assez

Gcc vs Clang avec differents flags d'optimisation pour Unrollx4:

gcc_vs_clang_flags_Unrollx4.png

Tous les mesures prises pour cette methode possedent une standart deviation inferieure à %5, ont peut dire que nos mesures sont stables.

On constante que le compiler clang n'est pas aussi performante que le gcc pour les flags O3 et Ofast.

Gcc vs Clang avec differents flags d'optimisation pour CBLAS:

gcc_vs_clang_flags_CBLAS.png

Le reultat le plus interessant de ce rapport. On voit que la performance de CBLAS n'augmente pas linéairement contrairement à ce qui est attendu avec les flags d'optimisation. Cela peut etre liée au fait que les mesures prises pour cette comparaison posédent des déviations standard entre %2.177 et %3.030, ils sont encore inferieure à %5 donc on peut dire que nos mesures sont encore assez stables, par contre il faut encore faire attention quand les differences de

bandwidth entre les methodes sont assez petites, meme ces deviations standards ont des grands effets sur les resultats.

Gcc et Clang avec dgemm reduc et dotprod avec la comparaison entre les mêmes flags
gcc_clang_flags_reduc_dotprod.png
Pour dotprod compilé avec clang O2 on obtient pas les mesures fiable, la devation standard est plus grande que %5. Mais meme avec les mesures qui ne sont pas stables, on peut constater que la redusction et le produit scalaire sont plus vite que les methodes de multiplication de matrices. Cela peut etre lié eu fait que ces deux possèdent les complexités linéaires en nombre d'acces au memoire.

gcc_clang_Ofast_reduc_dotprod.png
On ne pouvait pas representer les mesures prises avec Ofast en meme echelle que les autres mesures, donc ce png fournit un close-up.

* and ** areinspired by nbody readme file.
*https://wiki.gentoo.org/wiki/GCC_optimization/fr
*information about the CPU: *https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)
*L'information détaillée sur les flags d'optimisations est fournie par **gcc.gnu.org**.
*l'information sur les flags d'optimisation de clang
https://clang.llvm.org/docs/CommandGuide/clang.html et https://llvm.org/docs/Passes.html

Gülçin Gedik