HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Alberto Dassatti, Bruno
Da Rocha Carvalho

Laboratoire de High Performance Coding

semestre printemps 2025

High Performance Coding (HPC)

Temps à disposition: 4 périodes (deux séances de laboratoire)

Récupération du laboratoire dans l'archive sur Cyberlearn

1 Objectifs de ce laboratoire

Dans ce laboratoire, vous récupérerez le code mis en place dans le premier laboratoire afin d'analyser votre implémentation et de pouvoir commencer à l'améliorer.

2 Cahier des charges

Dans ce laboratoire, vous devrez utiliser l'outil likwid. Likwid est un outil open source de profilage et de monitoring de performance pour les applications exécutées sur des processeurs multi-cœurs. Il permet aux développeurs de comprendre et d'optimiser la performance de leurs applications en fournissant des informations détaillées sur l'utilisation du matériel informatique, telles que la consommation de cache, la latence de la mémoire et la bande passante, ainsi que l'utilisation du processeur et des threads. Likwid peut être utilisé pour mesurer des applications telles que l'affinité des threads, la vectorisation et l'optimisation de la bande passante mémoire.

Avec cet outil, vous allez devoir construire le roofline de votre processeur en fonction de deux facteurs : la capacité de calcul et la bande passante mémoire. Une fois celui-ci construit, vous pourrez profiler l'exécution de votre programme et analyser où il se situe dans votre roofline. Ainsi, vous pourrez exploiter au maximum les performances offertes par votre architecture et saurez où investir pour les améliorer.

3 Installation de likwid

Pour l'installation de likwid :

- Pour les utilisateurs natifs de Linux, vous pouvez installer le paquet `likwid`. Veuillez vous référer à la documentation qui vous explique comment l'installer.
- Si vous avez un autre environnement, on vous invite à utiliser les machines de laboratoire et à installer likwid en suivant le lien du premier point.

On vous conseille de jeter un œil à leur git pour toute information supplémentaire si vous avez une architecture spéciale, et de manière générale, pour trouver des informations sur tous les outils qu'ils proposent.

Faîtes tout de même attention à prendre une version récente de likwid (par example 5.3.0)

4 Roofline

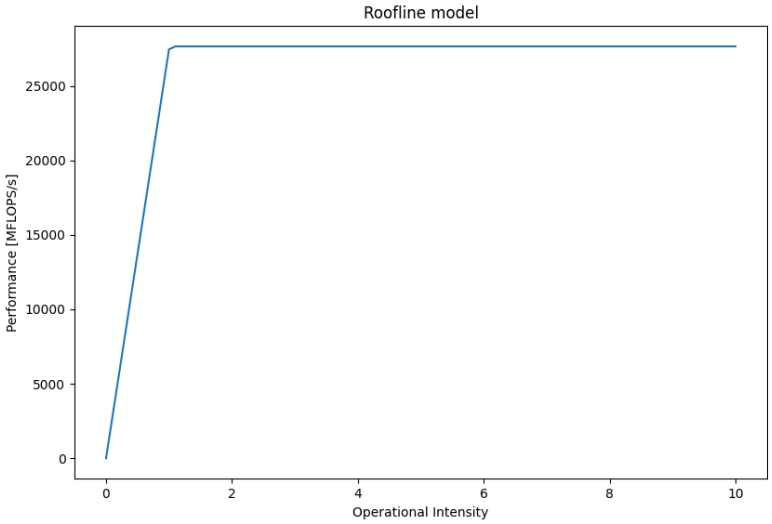
Exploitez les outils fournis par likwid afin de pouvoir dessiner votre roofline. Vous devez réaliser un banc de test pour déterminer le maximum d'opérations à virgule flottante que peut exécuter votre processeur. En consultant les pages de manuel des outils likwid, la documentation de Likwid et en effectuant des recherches sur Internet, vous pouvez trouver les options et les arguments nécessaires pour exécuter ce test de performance (il n'est pas nécessaire d'écrire du code).

Une fois votre valeur obtenue, vous serez intéressé par la recherche de la bande passante mémoire de votre processeur. Vous devrez relancer un banc de test afin de retrouver cette valeur, mais cette fois-ci, il vous est recommandé d'observer les capacités des mémoires L1, L2 et L3 afin de mieux les remplir avec la taille de votre banc de test (si vous choisissez une valeur trop élevée, le banc de test refusera de s'exécuter).

Une fois vos valeurs obtenues, la capacité de calcul que l'on appellera `maxperf` et votre bande passante que l'on appellera `maxband`, vous pourrez construire votre graphique du roofline de votre processeur.

4.1 Construction du graphique de performance

Voici un exemple typique d'un roofline :



Ce graphique est défini tel que :

$$f(x) = x \times \text{maxband}, \text{ si } x \times \text{maxband} < \text{maxperf}$$

$$\text{sinon, } f(x) = \text{maxperf}$$

Il vous est fourni dans le laboratoire un script Python qui gère déjà la construction du graphique. Vous devez simplement référencer dans un fichier texte `input.txt` les valeurs de `maxperf` et `maxband` comme suit :

```
maxperf
maxband
```

Ensuite, lancez le script et vous devriez voir votre roofline.

5 Profiling de votre code

Maintenant que vous possédez un graphique de référence, il est intéressant de pouvoir y placer du code afin de voir si l'on exploite au mieux la capacité de votre architecture. La bibliothèque likwid offre la possibilité d'analyser du code à des endroits précis, nous permettant ainsi de nous rendre compte si l'on exploite efficacement notre architecture. Votre but est de récupérer le code que vous avez réalisé pour le laboratoire n°1, de le recompiler avec le nouveau Makefile qui vous est fourni, et d'exploiter les marqueurs de likwid pour cibler les parties de code à mesurer.

Pour utiliser les marqueurs, il vous suffit d'ajouter l'inclusion de `<likwid-marker.h>` et ensuite d'englober les parties de code que vous souhaitez analyser avec les marqueurs :

```
LIKWID_MARKER_INIT;
[...]
LIKWID_MARKER_START("name");
[...] <- code à analyser
LIKWID_MARKER_STOP("name");
[...]
LIKWID_MARKER_CLOSE;
```

Vous pouvez également utiliser la fonction `LIKWID_MARKER_REGISTER("nom")` pour enregistrer plusieurs étiquettes de test et les démarrer et les arrêter au moment opportun, par exemple pour analyser différentes parties et les reconnaître facilement.

Un tutoriel plus détaillé est disponible ici.

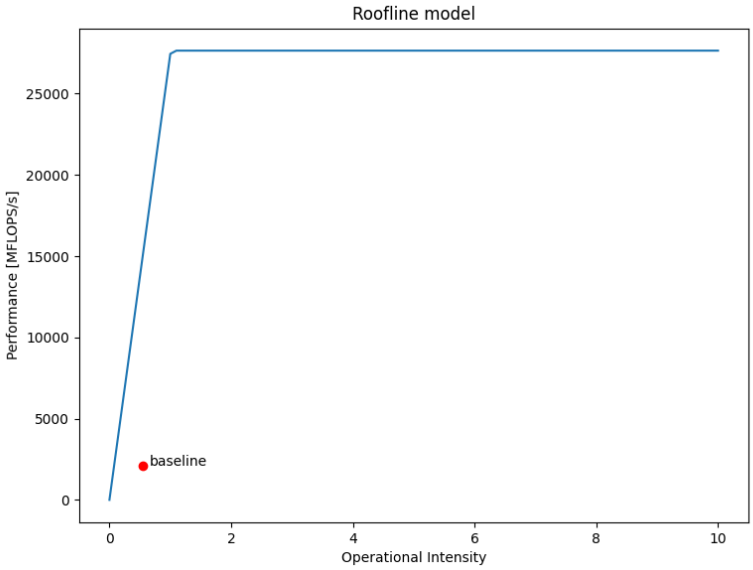
Une fois votre parcelle de code encadrée par les marqueurs, vous utiliserez l'outil d'analyse de performance de likwid. Faites bien attention d'utiliser le bon groupe de surveillance ainsi que de spécifier correctement un identifiant de processeur. Comme précédemment, lisez attentivement le manuel de l'analyseur de performances pour retrouver facilement toutes les informations dont vous avez besoin.

Une fois l'analyse terminée, vous devrez reporter vos valeurs dans le graphique afin de situer où se situe votre algorithme. Ajoutez au fichier `input.txt` les valeurs d'intensité opérationnelle (abscisse) et le nombre d'opérations à virgule flottante (ordonnée) afin de pouvoir les visualiser, en faisant de cette mesure une "baseline".

Vous devrez typiquement vous retrouver avec un fichier comme :

```
27648.63
27452.33
5.522620e-1 2127.4496 baseline
```

Une fois lancé, vous pourrez observer où se situe votre programme. Notez que vous pouvez ajouter plusieurs points en même temps en continuant à en ajouter à la suite dans le fichier `input.txt`.



6 Complément

Chaque architecture est différente et parfois votre architecture ne permet pas les mêmes analyses qu'une autre. De plus, il vous faudra souvent adapter les types d'analyses à votre propre code, certains n'utilisant que des entiers tandis que d'autres des nombres à virgule, etc.

Likwid offre des groupes de performances de base pour des utilisations standards. Cependant, comme chaque architecture n'est pas nécessairement identique, les groupes de performances diffèrent. Vous le remarquerez car certains d'entre vous auront certains groupes et d'autres non.

Un groupe de performance est un fichier de description qui permet à likwid de savoir quels compteurs et quels événements du CPU il doit utiliser. Cela signifie que vous pouvez créer votre propre groupe de performance pour analyser ce que vous voulez. Pour lister tous les événements disponibles pour votre architecture, vous pouvez utiliser la commande `likwid-perfctr -e`.

Cherchez les événements qui peuvent vous intéresser.

6.1 Créer votre groupe de performance

Vous pouvez créer un fichier, par exemple `HPC.txt`, que vous pourrez déplacer comme suit :

```
LIKWID_PATH=$(whereis likwid | awk '{print $2}')
cp HPC.txt "$LIKWID_PATH/perfgroups/{votre_architecture}" #pour les machines de labo haswell
```

Dans ce fichier, vous devrez spécifier :

- Un **SHORT**, décrivant ce qu'est votre fichier.
- Un **EVENTSET** décrivant ce que vous souhaitez compter. Attention, pour cette partie, un événement ne peut pas être compté avec n'importe quel compteur. Pour comprendre quels sont les compteurs disponibles, vous pouvez les lister avec `likwid-perfctr -e`. Pour savoir quel compteur permet de compter l'événement que vous désirez, il sera spécifié dans sa description. Par exemple : `AVX_INSTS_ALL, 0xC6, 0x7, PMC` nous montre que l'événement `AVX_INSTS_ALL` utilise le compteur PMC. Toutefois, le nombre de compteurs PMC est limité.
- Les **METRICS** que vous désirez calculer, typiquement l'intensité opérationnelle et vos opérations par seconde.

Inspirez-vous des groupes existants pour la rédaction de votre fichier.

6.2 Example d'utlisation

Prenons comme exemple le code suivant (disponible dans l'archive du laboratoire) : *Ce code est basé sur l'architecture des machines de laboratoire.*

```
#include <likwid-marker.h>
#include <stdlib.h>
#include <stdio.h>

void init(int* ai, double* af, int SIZE){
        for(int i = 0; i < SIZE; ++i){
                ai[i] = rand();
                af[i] = rand();
        }
}

void cache_clean(int *tmp, int CS){
        for(int i = 0; i < CS; ++i){
                tmp[i] = rand();
        }
}

int main(int argc, char** argv){
        int count = 0;
        double count_fp = 0.0;
        //Ici cache_size est initialisé à 4MB car il va nous permettre de remplir les caches
        // entièrement afin d'être certain que array et array_f soit dans la DRAM est plus en cache.
        size_t cache_size = 4*1024*1024;
        size_t SIZE = atoi(argv[1]);
        int *array = malloc(sizeof(*array)*SIZE);
        double *array_f = malloc(sizeof(*array_f)*SIZE);
        //tmp aura donc un size de 16MB ce qui nous assure de remplir
        // nos 3 caches (L1 32kB, L2 256kB et L3 8MB)
        int *tmp = malloc(cache_size*sizeof(int));

        if(tmp == NULL){
                printf("tmp pointer is null");
                return;
        } else if (array == NULL) {
                printf("array pointer is null");
                return;
        } else if (array_f == NULL){
                printf("array_f pointer is null");
                return;
        }

        init(array, array_f, SIZE);
        cache_clean(tmp, cache_size);

        LIKWID_MARKER_INIT;
        LIKWID_MARKER_START("int");
        for(int i = 0; i < SIZE; ++i){
                count += array[i] * i;
        }
        LIKWID_MARKER_STOP("int");
        LIKWID_MARKER_START("double");
        for(int i = 0; i < SIZE; ++i){
                count_fp += array_f[i] * (double)i;
        }
        LIKWID_MARKER_STOP("double");
        LIKWID_MARKER_CLOSE;

        printf("int READS %ld, double READS %ld\n", sizeof(int)*SIZE,
        sizeof(double)*SIZE);
        return count + count_fp;
}
```

Ce code va effectuer des opérations sur des entiers et des nombres à virgule flottante. Nous allons détailler pas à pas les calculs à la main pour vérifier et comprendre ce que vous pourriez faire avec likwid.

Si l'on donne en entrée à ce programme une taille de tableau de 100'000 pour les tableaux `array` et `array_f`, nous nous focaliserons sur la partie entière du programme car la partie double est identique :

- Pour le nombre de données lues, nous nous attendons à avoir `sizeof(int) * SIZE` soit 400'000 octets. Voici ce que nous obtenons avec l'événement `DRAM_READS`:

```
+--------------------+---------+------------+
| Event              | Counter | HWThread 0 |
+--------------------+---------+------------+
| DRAM_READS         | MBOX0C1 |       6524 |
```

  Ce n'est pas le chiffre attendu, mais il faut comprendre que ce qu'il nous retourne n'est pas le nombre de bytes lu, mais le nombre de lignes de cache lues. Les lignes de cache de notre architecture Haswell sont de 64 octets par ligne. Il ne reste plus qu'à multiplier le compteur par 64, soit : 6524 * 64 = 417 536 octets. Donc, un résultat beaucoup plus correct, sachant que likwid en lui-même va générer des lectures.

- Pour l'intensité opérationnelle, nous cherchons à compter le nombre total d'instructions divisé par la taille totale de données lues. Si nous réduisons notre champ de calcul à une seule itération et que nous examinons le code assembleur de la boucle :

```
    count += array[i] * i;
11d8:       41 8b 14 87             mov    (%r15,%rax,4),%edx
11dc:       0f af d0                imul   %eax,%edx
11df:       01 d3                   add    %edx,%ebx
for(int i = 0; i < SIZE; ++i){
11e1:       48 89 c2                mov    %rax,%rdx
11e4:       48 83 c0 01             add    $0x1,%rax
11e8:       48 39 ea                cmp    %rbp,%rdx
11eb:       75 eb                   jne    11d8 <main+0x98>
}
```

  On compte 7 instructions. On sait aussi qu'il y a 4 octets lus par itération. L'intensité opérationnelle est donc de 7/4 = **1.75**. Avec le calcul total de likwid, on obtient `| Operational intensity | 1.7020 |`.

- Pour le nombre d'opérations par seconde, le calcul consiste à diviser le nombre d'opérations par le temps écoulé. Faites attention au type de données.

Finalement voici quelque lien utile:

- Tous les groupes de performances de likwid
- Les descriptions des événements CPU de la machine de laboratoire

7 Travail à rendre

Maintenant que vous disposez des connaissances nécessaires et d'un environnement prêt à être utilisé pour profiler votre programme, vous êtes invités à explorer le code que vous avez réalisé lors du premier laboratoire. Analysez-le attentivement, utilisez les outils à votre disposition pour identifier ses points forts et ses points faibles, puis essayez d'améliorer le code avec les connaissances acquises en cours.

Dans un rapport, documentez toutes vos analyses et expérimentations. Assurez-vous que le rapport soit concis et rédigé dans un format compatible avec le markdown ou en format PDF. N'oubliez pas d'inclure également votre code.

L'objectif est de fournir une **évaluation critique du code existant et de proposer des améliorations** pertinentes. Soyez méthodiques dans vos analyses et créatifs dans vos suggestions d'amélioration.