MASTER INFORMATIQUE | UNIVERSITÉ CÔTE D'AZUR

Rapport de Projet

Architecture des Processeurs Hautes Performances

Analyse expérimentale de la hiérarchie mémoire
Architecture Intel Raptor Lake (i7-13650HX)

Auteur : Achour Djerada
Master 1 Informatique
Année universitaire 2024/2025

Encadrant : Pr. Sid Touati
Université Côte d'Azur

December 29, 2025

Abstract

Ce rapport présente une étude expérimentale approfondie de la hiérarchie mémoire d'un processeur Intel Core i7-13650HX (architecture Raptor Lake). À travers le développement de micro-benchmarks en langage C et l'utilisation de l'outil Calibrator, nous avons mesuré les latences d'accès et les bandes passantes des différents niveaux de cache (L1, L2, L3) ainsi que de la mémoire DDR5.

Notre contribution principale réside dans l'implémentation de la technique du *Pointer Chasing* avec randomisation Fisher-Yates, permettant de contourner les mécanismes de préchargement matériel (*Hardware Prefetcher*) qui masquent les latences réelles. Les résultats expérimentaux confirment les spécifications théoriques du processeur : cache L1 de 48 Ko (≈1,1 ns), cache L2 de 1,25 Mo (≈3,2 ns), cache L3 de 24 Mo (≈15-20 ns), et RAM DDR5 (≈71 ns). Cette étude démontre l'importance cruciale de la *Memory Awareness* dans le développement d'applications hautes performances.

Mots-clés : Hiérarchie mémoire, Cache, Latence, Bande passante, Pointer Chasing, Prefetcher, TLB, Micro-benchmark.

Contents

1 Introduction

L'évaluation de performance (*benchmarking*) sur les architectures modernes est devenue un défi complexe. Contrairement aux processeurs des décennies précédentes, les puces actuelles, comme l'**Intel Core i7-13650HX** (architecture Raptor Lake) utilisé pour ce projet, intègrent des mécanismes d'optimisation agressifs conçus pour masquer la latence mémoire à l'utilisateur.

Le fossé grandissant entre la vitesse des processeurs et celle de la mémoire centrale — communément appelé *Memory Wall* — a conduit les concepteurs de processeurs à développer des hiérarchies de caches de plus en plus sophistiquées. Comprendre et mesurer les caractéristiques de ces hiérarchies est essentiel pour tout développeur souhaitant optimiser ses applications dans un contexte HPC (*High Performance Computing*).

Ce projet a pour objectif de « piéger » les mécanismes d'optimisation matériels pour mesurer les caractéristiques physiques réelles de la hiérarchie mémoire : caches L1, L2, L3 et RAM DDR5. Nous avons développé des micro-benchmarks capables de révéler ces caractéristiques en contournant notamment le *Hardware Prefetcher* du processeur.

Nous verrons au cours de ce rapport que la réalisation de ces mesures ne s'est pas faite sans heurts. Nos premières tentatives ont abouti à des résultats incohérents, masqués par les optimisations matérielles. Nous détaillerons la démarche scientifique qui nous a permis de contourner ces optimisations, notamment l'implémentation du *Pointer Chasing* aléatoire avec l'algorithme de Fisher-Yates.

1.1 Déclaration d'usage d'IA générative

Dans le cadre de ce projet, j'ai utilisé une intelligence artificielle générative (Claude, Anthropic) pour m'aider à améliorer la structure et le contenu de ce rapport, ainsi que pour corriger certaines erreurs dans mes codes. L'historique des prompts est fourni dans le fichier `IAG.txt` joint à l'archive.

2 Description de l'environnement expérimental

Cette section décrit en détail l'environnement matériel et logiciel utilisé pour nos expériences, conformément aux exigences de reproductibilité scientifique.

2.1 Environnement matériel

Les expériences ont été réalisées sur un ordinateur portable Lenovo Legion 5 15IRX10 équipé d'un processeur Intel de 13e génération. Le tableau 1 résume les caractéristiques principales.

Table 1: Configuration matérielle de la machine de test

Composant	**Spécification**
Machine	Lenovo Legion 5 15IRX10
Processeur	Intel Core i7-13650HX
Architecture	Raptor Lake (x86_64)
Nombre de cœurs	14 (6 P-cores + 8 E-cores)
Nombre de threads	20
Fréquence min / max	800 MHz / 4,9 GHz
Mémoire RAM	32 Go DDR5

2.1.1 Hiérarchie mémoire détaillée

Le tableau 2 présente les caractéristiques détaillées de la hiérarchie de cache, obtenues via les commandes `lscpu` et `getconf`.

Table 2: Caractéristiques de la hiérarchie de cache (i7-13650HX)

Niveau	**Taille**	**Ligne cache**	**Associativité**	**Type**
L1 Data (P-core)	48 Ko	64 octets	12-way	Privé par cœur
L1 Instruction	32 Ko	64 octets	8-way	Privé par cœur
L2 (P-core)	1,25 Mo	64 octets	10-way	Privé par cœur
L3 (Smart Cache)	24 Mo	64 octets	12-way	Partagé

2.1.2 Caractéristiques du pipeline et parallélisme d'instructions

Le processeur i7-13650HX utilise une architecture hybride avec deux types de cœurs :

- **P-cores (Performance)** : Architecture Golden Cove, pipeline profond (≈20 étages), exécution *out-of-order* avec jusqu'à 6 instructions par cycle, support de l'Hyper-Threading.
- **E-cores (Efficient)** : Architecture Gracemont, pipeline plus court (≈13 étages), optimisés pour l'efficacité énergétique, pas d'Hyper-Threading.

Le processeur dispose également d'un *Hardware Prefetcher* avancé capable de détecter et précharger les motifs d'accès linéaires et à pas fixe (*stride*).

2.2 Environnement logiciel

Table 3: Configuration logicielle

Composant	Version
Système d'exploitation	Ubuntu 24.04.3 LTS (Noble Numbat)
Noyau Linux	6.14.0-37-generic
Compilateur	GCC 13.3.0
Outil de graphiques	Gnuplot 5.4

2.2.1 Options de compilation

Les micro-benchmarks ont été compilés avec les options suivantes :

```
gcc -O2 -Wall -Wextra -o benchmark benchmark.c
```

Le niveau d'optimisation `-O2` a été choisi pour permettre des optimisations raisonnables sans que le compilateur ne supprime les boucles de mesure (ce qui peut arriver avec `-O3`).

2.2.2 Outil de mesure du temps

Nous avons utilisé la fonction `clock_gettime()` avec l'horloge `CLOCK_MONOTONIC` pour obtenir une précision à la nanoseconde :

```c
static inline uint64_t get_time_ns(void) {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return (uint64_t)ts.tv_sec * 1000000000ULL +
           (uint64_t)ts.tv_nsec;
}
```

Listing 1: Fonction de mesure du temps

2.3 Configuration de la machine pendant les expériences

Pour garantir la fiabilité des mesures, nous avons suivi un protocole strict :

1. **Réduction de la charge système :** Fermeture de l'interface graphique (passage en mode console via `Ctrl+Alt+F3`), arrêt des services non essentiels.

2. **Gestion de la fréquence CPU :** Le gouverneur de fréquence a été configuré en mode « performance » pour éviter les variations de fréquence pendant les mesures :

   ```
   sudo cpupower frequency-set -g performance
   ```

3. **Affinage du processus :** Utilisation de `taskset` pour fixer l'exécution sur un seul P-core :

   ```
   taskset -c 0 ./benchmark
   ```

4. **Priorité d'exécution :** Exécution avec priorité temps réel quand possible :

```
sudo nice -n -20 ./benchmark
```

Difficultés rencontrées : Sur un système moderne avec Turbo Boost et un noyau préemptif, il est difficile d'obtenir des conditions parfaitement reproductibles. Nous avons donc effectué de multiples répétitions et conservé les valeurs minimales (représentant le cas sans interférence).

3 Exercice 1 : Détection des tailles de cache par mesure de latence

L'objectif de cet exercice est de tracer la courbe de latence d'accès mémoire en fonction de la taille des données (*working set*), afin de visualiser les « marches » correspondant aux transitions entre les niveaux de cache.

3.1 Problématique : L'échec de la mesure naïve

Lors de nos premières expérimentations, nous avons utilisé un accès mémoire séquentiel (lire `tab[0]`, puis `tab[1]`, etc.) ou avec un pas fixe (*stride*). Les résultats obtenus étaient déconcertants : la courbe de latence restait désespérément plate, aux alentours de 2 à 3 nanosecondes, même pour des tailles de données dépassant largement la taille du cache L3 (24 Mo).

3.1.1 Analyse du problème

Le processeur i7-13650HX dispose d'une unité de *Hardware Prefetching* très performante. Cette unité surveille les motifs d'accès mémoire et, lorsqu'elle détecte un accès linéaire ou à pas fixe, elle précharge automatiquement les lignes de cache suivantes depuis la RAM **avant même** que le programme n'en ait besoin.

Conséquence : nous ne mesurions pas la latence de la RAM, mais simplement la vitesse à laquelle le CPU peut consommer des données déjà présentes en cache L1/L2.

3.2 Solution : Pointer Chasing avec randomisation

Pour mesurer la latence réelle, il est impératif de briser la localité spatiale. Le CPU ne doit pas pouvoir deviner quelle sera la prochaine adresse mémoire à lire.

3.2.1 Principe du Pointer Chasing

Nous avons implémenté la technique du **Pointer Chasing** : chaque case mémoire contient l'adresse de la case suivante à visiter. Ainsi, le CPU doit attendre le retour de la donnée `*p` avant de connaître l'adresse suivante `p`, créant une dépendance de données qui empêche toute anticipation.

Cependant, un chaînage linéaire ($1 \to 2 \to 3 \to ...$) ne suffit pas car le prefetcher peut encore détecter le motif. Nous avons donc implémenté l'algorithme de mélange de **Fisher-Yates** pour randomiser totalement l'ordre de visite des lignes de cache.

3.2.2 Implémentation

```
void create_pointer_chain(void **array, size_t size_bytes,
                          size_t stride) {
    size_t count = size_bytes / stride;
    size_t *indices = malloc(count * sizeof(size_t));
    size_t step_index = stride / sizeof(void*);

    // Initialisation des indices
    for (size_t i = 0; i < count; i++) {
        indices[i] = i * step_index;
```

```
    }

    // Melange de Fisher-Yates
    for (size_t i = count - 1; i > 0; i--) {
        size_t j = rand() % (i + 1);
        size_t temp = indices[i];
        indices[i] = indices[j];
        indices[j] = temp;
    }

    // Chainage des pointeurs selon l'ordre melange
    for (size_t i = 0; i < count - 1; i++) {
        array[indices[i]] = (void*)&array[indices[i+1]];
    }
    // Fermeture du cycle
    array[indices[count-1]] = (void*)&array[indices[0]];

    free(indices);
}
```

Listing 2: Création de la chaîne de pointeurs aléatoire

La boucle de mesure est alors très simple :

```
double measure_latency(void **array, size_t num_accesses) {
    void **p = array;
    uint64_t start = get_time_ns();

    for (size_t i = 0; i < num_accesses; i++) {
        p = (void **)*p;  // Dependance: adresse suivante = *p
    }

    uint64_t end = get_time_ns();
    sink = p;  // Empeche l'optimisation
    return (double)(end - start) / (double)num_accesses;
}
```

Listing 3: Boucle de mesure avec dépendance de données

3.3 Résultats expérimentaux

La figure 1 présente la courbe de latence obtenue avec notre implémentation du Pointer Chasing aléatoire.

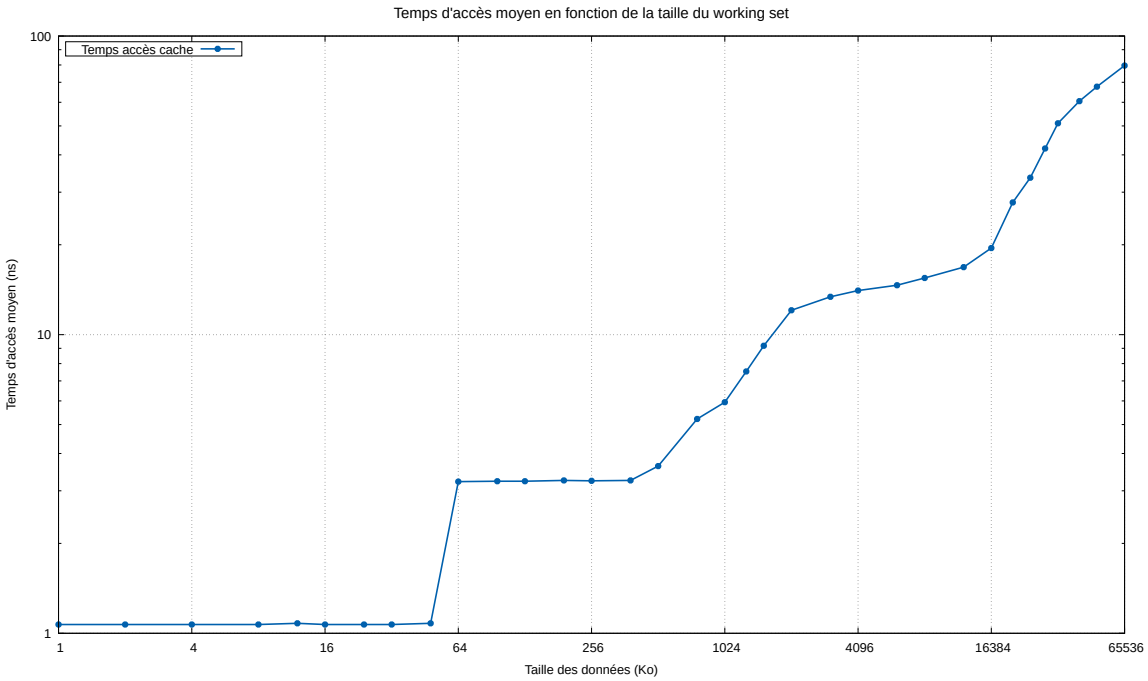

Figure 1: Latence d'accès mémoire en fonction de la taille du working set (Pointer Chasing aléatoire, échelle log-log)

3.4 Analyse des résultats

L'analyse de la courbe révèle clairement la topologie mémoire du i7-13650HX :

1. **Zone L1 (1 – 48 Ko) :** Latence stable et minimale d'environ **1,1 ns**. Cette valeur représente le coût d'un accès au cache L1 Data, confirmant sa taille de 48 Ko.

2. **Zone L2 (64 Ko – 1,25 Mo) :** Premier décrochage vers **3,2 ns**. Cette augmentation correspond à la transition vers le cache L2. La marche est visible autour de 64 Ko, ce qui confirme le débordement du L1.

3. **Zone L3 (1,5 Mo – 24 Mo) :** La latence augmente progressivement jusqu'à environ **15-20 ns**. Le cache L3 étant partagé entre tous les cœurs et de grande taille (24 Mo), le temps d'accès varie selon la localisation des données dans la structure de cache.

4. **Zone RAM (> 24 Mo) :** Au-delà de 24 Mo, la latence explose pour atteindre **71 ns** à 64 Mo. C'est le *Memory Wall* : le coût d'un défaut de cache L3 (*cache miss*) qui nécessite un accès à la DDR5.

Le tableau 4 résume les latences mesurées et les compare aux valeurs théoriques.

Table 4: Comparaison des latences mesurées et théoriques

Niveau	**Latence mesurée**	**Latence typique**	**Ratio vs L1**
L1 Cache	≈ 1,1 ns	1–2 ns	1x
L2 Cache	≈ 3,2 ns	3–4 ns	3x
L3 Cache	≈ 15-20 ns	10–20 ns	15x
RAM DDR5	≈ 71 ns	60–80 ns	65x

Observation clé : Le ratio de latence entre le L1 et la RAM est d'environ **65x**. Cela illustre parfaitement pourquoi les optimisations de localité mémoire sont cruciales en HPC : un programme avec de nombreux *cache misses* peut être 65 fois plus lent qu'un programme bien optimisé !

4 Exercice 2 : Évaluation de la bande passante mémoire

Si l'exercice 1 mesurait le temps d'attente (latence), l'exercice 2 mesure le débit (bande passante). L'objectif est d'évaluer la quantité de données que le processeur peut transférer par unité de temps depuis les différents niveaux de la hiérarchie mémoire.

4.1 Partie 1 : Impact du pas d'accès (Stride)

Nous avons mesuré la bande passante en faisant varier le pas d'accès lors de la lecture d'un grand tableau (96 Mo, soit 4 fois la taille du L3).

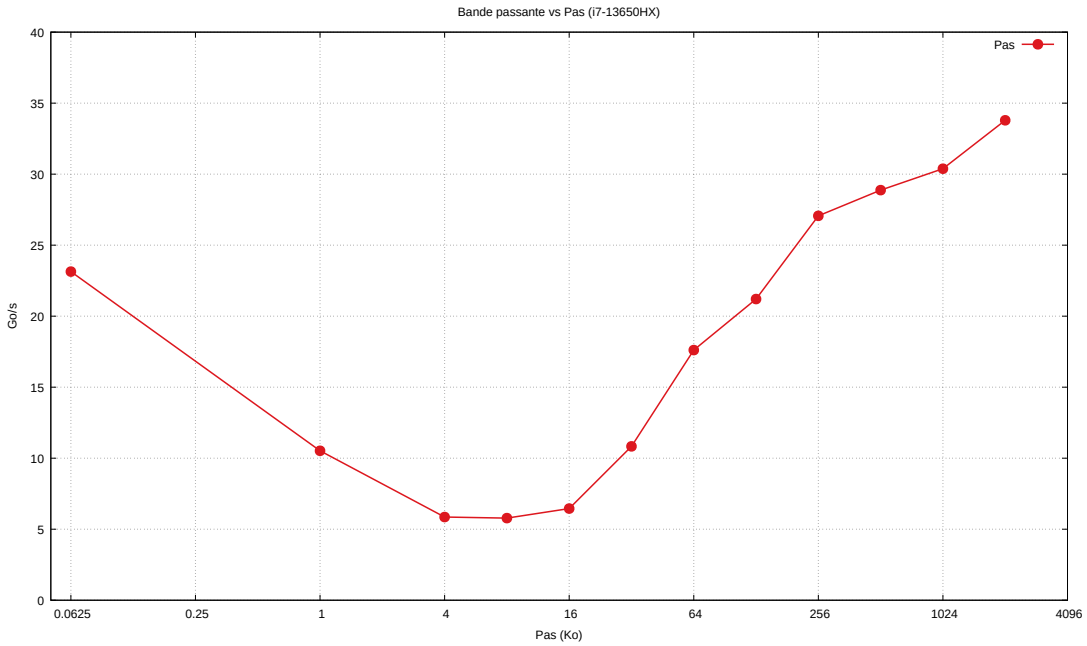

Figure 2: Évolution de la bande passante en fonction du pas d'accès

4.1.1 Analyse des résultats

Le graphique de la figure 2 révèle plusieurs phénomènes :

1. **Pas faible (64 octets) :** Bande passante maximale d'environ **35 Go/s**. À ce pas, chaque ligne de cache chargée est entièrement utilisée, et le prefetcher peut anticiper les accès suivants.

2. **Pas de 4 Ko (taille d'une page) :** Effondrement brutal à environ **5-6 Go/s**. Cette chute s'explique par la saturation du **TLB** (*Translation Lookaside Buffer*). À chaque accès avec un pas de 4 Ko, le processeur accède à une nouvelle page virtuelle, ce qui nécessite une traduction d'adresse. Quand le nombre de pages dépasse la capacité du TLB, chaque accès provoque un *TLB miss* coûteux.

3. **Pas très grands ($>$ 64 Ko) :** La bande passante continue de diminuer vers **2-3 Go/s** car le prefetcher est totalement inefficace et chaque accès génère un *cache miss* complet.

4.2 Partie 2 : Transition Cache L3 / RAM

Pour isoler la bande passante réelle de chaque niveau mémoire, nous avons réalisé un test d'accès séquentiel en faisant varier la taille du buffer.

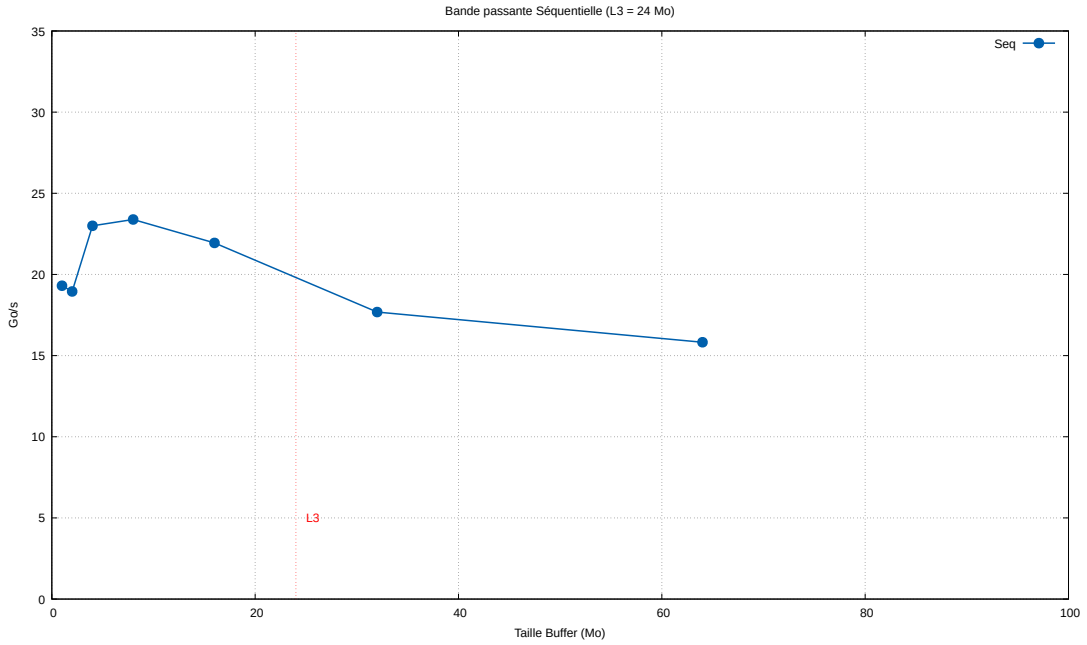

Figure 3: Bande passante séquentielle en fonction de la taille du buffer (ligne verticale : limite L3 = 24 Mo)

4.2.1 Analyse des résultats

Le graphique de la figure 3 illustre parfaitement la transition Cache/RAM :

- **Zone $<$ 24 Mo :** Le débit est très élevé, entre **23 et 30 Go/s**. Les données tiennent entièrement dans le cache L3, et le prefetcher peut alimenter le processeur efficacement.
- **Point de rupture (24 Mo) :** Entre 16 Mo et 32 Mo, la courbe chute brutalement. C'est la **preuve expérimentale** que le cache L3 de notre i7-13650HX fait bien 24 Mo.
- **Zone $>$ 24 Mo :** La courbe se stabilise autour de **16 Go/s**. C'est la bande passante effective de la DDR5 pour un seul cœur CPU.

4.2.2 Discussion sur la bande passante RAM

La DDR5-4800 de notre système a une bande passante théorique de :

$$\text{BP}_{\text{théorique}} = 4800 \times 10^6 \times 8 \times 2 = 76,8 \text{ Go/s}$$

(4800 MT/s $\times$ 8 octets par transfert $\times$ 2 canaux)

Cependant, nous mesurons seulement $\approx$16 Go/s pour un seul cœur. Cette différence s'explique par plusieurs facteurs :

1. Un seul cœur ne peut pas saturer le bus mémoire à lui seul.
2. Le contrôleur mémoire doit gérer les *Row Buffer Misses*.
3. Les protocoles de cohérence de cache ajoutent de la latence.

5 Exercice 5 : L'outil Calibrator

L'outil `calibrator` est un micro-benchmark de référence développé par Stefan Manegold au CWI (Centrum Wiskunde & Informatica) aux Pays-Bas. Il utilise une méthodologie sophistiquée pour détecter automatiquement les caractéristiques de la hiérarchie mémoire.

5.1 Défis de portage et compilation

L'utilisation d'un code source datant de plus de 20 ans sur un système moderne (GCC 13, Ubuntu 24.04) a nécessité un travail d'adaptation que l'on peut qualifier d'« archéologie logicielle ».

5.1.1 Conflit de nom avec la fonction round()

Nous avons rencontré une erreur bloquante lors de la compilation :

```
calibrator.c:131:5: error: conflicting types for 'round'
  131 | lng round(dbl x)
```

Cause : Le code original définissait sa propre fonction `round()`. Or, depuis la norme C99, `round()` est une fonction standard de la bibliothèque mathématique `libm`, déclarée dans `<math.h>`.

Solution : Nous avons modifié le code source pour renommer la fonction interne en `my_round()` :

```
sed -i 's/lng round/lng my_round/g' calibrator.c
sed -i 's/round(/my_round(/g' calibrator.c
```

5.1.2 Gestion de la fréquence CPU

Calibrator nécessite de connaître la fréquence CPU pour convertir les temps en cycles. Sur le i7-13650HX avec Turbo Boost, la fréquence varie dynamiquement. Nous avons utilisé la fréquence de base (2600 MHz) comme référence :

```
./calibrator 2600 128M results
```

5.2 Méthodologie de Calibrator

Contrairement à notre benchmark de l'exercice 1, Calibrator utilise une approche bidimensionnelle : il fait varier simultanément la taille du buffer (*Range*) et le pas d'accès (*Stride*). Cette méthode permet de détecter :

- Les tailles et latences des caches (en fixant le stride à la taille de ligne cache)
- Les caractéristiques du TLB (en utilisant un stride égal à la taille de page)

5.3 Résultats Calibrator

La figure 4 présente les graphiques générés par Calibrator.

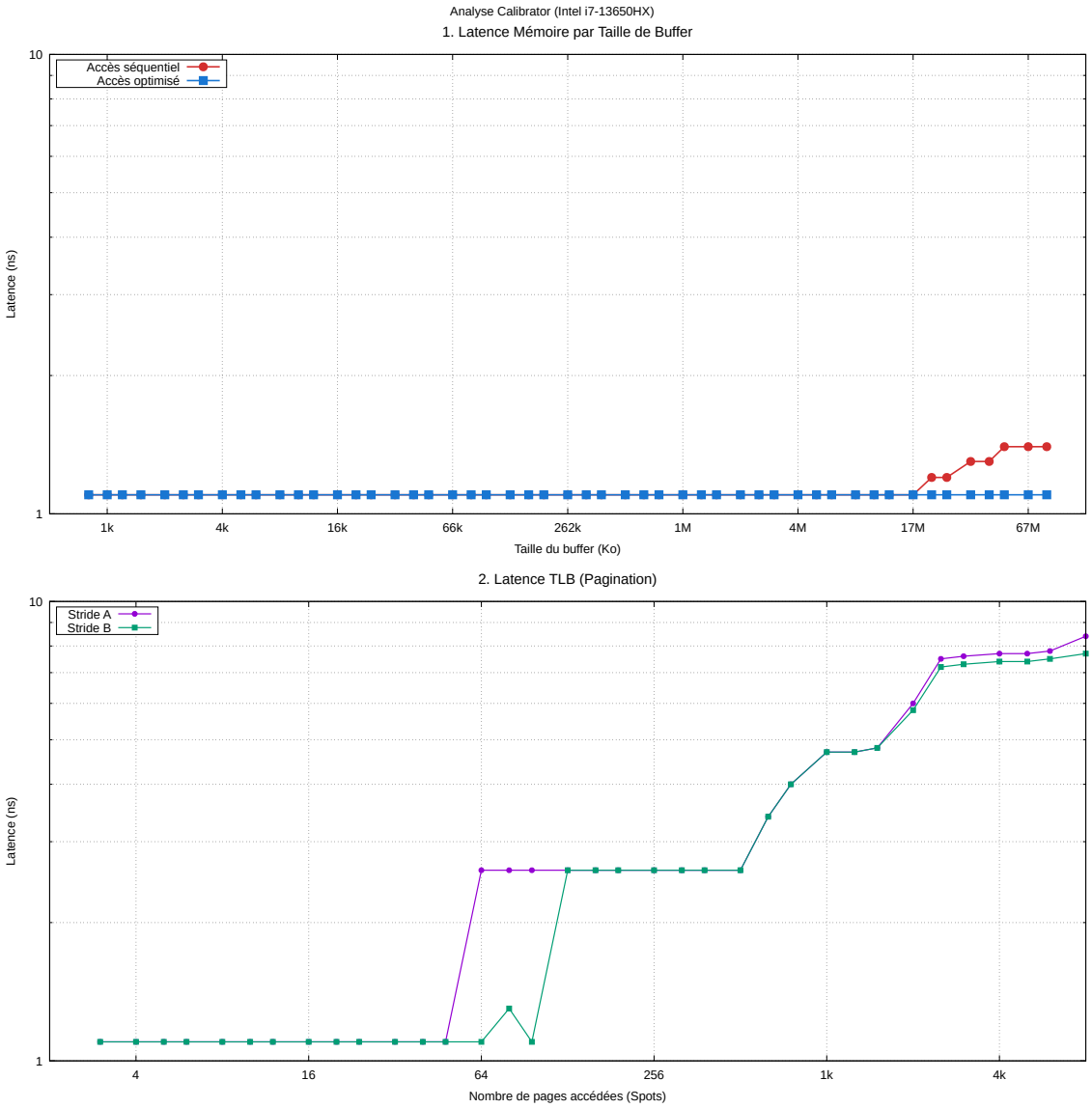

Figure 4: Résultats Calibrator : (Haut) Latence Cache, (Bas) Latence TLB

5.3.1 Analyse du graphique de latence cache

Le graphique supérieur montre deux courbes :

- **Accès séquentiel :** Latence basse et relativement stable grâce au prefetcher.
- **Accès optimisé (pointer chasing) :** Montre clairement les paliers correspondant aux différents niveaux de cache.

5.3.2 Analyse du graphique TLB

Le graphique inférieur (*TLB Latency*) est particulièrement intéressant. Il montre :

- Une latence faible tant que le nombre de pages accédées reste inférieur à la capacité du TLB L1 (≈64 entrées).
- Une augmentation significative au-delà, correspondant aux *TLB misses* qui déclenchent des *Page Walks* coûteux dans les tables de pages.

5.4 Comparaison : Notre benchmark vs Calibrator

Table 5: Comparaison des approches

Critère	**Notre Benchmark (Ex.1)**	**Calibrator**
Approche	Variation taille uniquement	Variation taille + stride
Visualisation	Courbe simple, lisible	Graphiques détaillés
Détection TLB	Non	Oui
Portabilité	Moderne, compile facilement	Nécessite adaptations
Automatisation	Manuelle	Détection automatique

Calibrator est plus complet mais plus complexe. Notre approche a l'avantage de la simplicité et de la lisibilité des résultats.

6 Conclusion

Ce projet nous a permis de confronter la théorie vue en cours à la réalité complexe d'un processeur moderne. Au-delà des simples mesures, nous avons acquis une compréhension profonde des mécanismes qui régissent les performances mémoire.

6.1 Enseignements techniques

1. **La mesure est un art difficile :** Les mécanismes d'optimisation matériels (Prefetching, Turbo Boost, exécution *out-of-order*) tentent activement de masquer les caractéristiques réelles du système. Sans protocole adapté (Pointer Chasing aléatoire, contrôle de la fréquence), les mesures sont faussées.

2. **La hiérarchie mémoire est tangible :** Nous avons pu « voir » physiquement la limite des 24 Mo du cache L3 sur nos courbes de bande passante. La transition Cache/RAM n'est pas une abstraction théorique, c'est un phénomène mesurable avec des conséquences directes sur les performances.

3. **Le coût du cache miss est énorme :** Le ratio de latence entre L1 et RAM est d'environ 65x. Un algorithme qui génère beaucoup de *cache misses* peut être des dizaines de fois plus lent qu'un algorithme bien optimisé travaillant sur les mêmes données.

6.2 Implications pour le développement HPC

Ces expériences confirment l'importance cruciale de la « *Memory Awareness* » dans le développement d'applications hautes performances :

- **Favoriser la localité spatiale :** Accéder aux données de manière séquentielle pour bénéficier du prefetcher et maximiser l'utilisation de chaque ligne de cache.
- **Favoriser la localité temporelle :** Réutiliser les données tant qu'elles sont en cache (technique du *blocking* ou *tiling*).
- **Dimensionner les structures de données :** Adapter la taille des buffers de travail aux capacités des caches pour éviter les débordements.
- **Attention au TLB :** Éviter les accès avec des pas de 4 Ko qui saturent le TLB.

6.3 Perspectives

Ce travail pourrait être étendu de plusieurs manières :

- Mesurer l'impact du multithreading sur la bande passante partagée du L3.
- Comparer les performances entre P-cores et E-cores de l'architecture hybride.
- Utiliser les compteurs matériels de performance (`perf`) pour corréler les mesures de temps avec les événements micro-architecturaux (cache misses, TLB misses, etc.).

En conclusion, ce projet démontre que même sur des processeurs modernes hautement optimisés, la compréhension fine de la hiérarchie mémoire reste un prérequis indispensable pour quiconque souhaite développer des applications véritablement performantes.

References

[1] S. Manegold, P. Boncz, et M. Kersten, *Generic Database Cost Models for Hierarchical Memory Systems*, Proceedings of the 28th VLDB Conference, 2002.

[2] U. Drepper, *What Every Programmer Should Know About Memory*, Red Hat, Inc., 2007. https://people.freebsd.org/~lstewart/articles/cpumemory.pdf

[3] S. Manegold, *Calibrator: A Cache-Memory and TLB Calibration Tool*, CWI Amsterdam. http://homepages.cwi.nl/~manegold/Calibrator/

[4] Intel Corporation, *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Order Number: 248966-047, 2023.

[5] J. L. Hennessy et D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th Edition, Morgan Kaufmann, 2019.