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

Reconfigurable & embedded Digital Systems

Acceleration of an application

ZombieV

Départements : TIC
Unité d'enseignement CNM

Auteurs : **Thomas Stäheli & Ali Zoubir**
Professeur : **Marina Zapater Sancho**
Assistant : **Mehdi Akeddar**
Classe : **A**
Salle de labo : **A07**
Date : **09.01.2026**

Contents

1 Stage 1 - Choix et Description de l'application

1.1 Description de l'application : ZombieV

L'application sélectionnée pour ce laboratoire est **ZombieV**, une simulation de type "top-down shooter" en 2D développée en C++. Elle simule la survie d'agents (joueurs ou bots) face à des vagues successives d'ennemis (Zombies).

Caractéristiques techniques : L'application repose sur la bibliothèque graphique **SFML** pour le fenêtrage et le rendu bas niveau. Cependant, elle intègre un moteur de jeu personnalisé ("Custom Game Engine") qui gère plusieurs aspects coûteux en calcul :

- **Moteur Physique :** Gestion des collisions entre entités (AABB ou cercles), déplacement vectoriel et résolution des forces.
- **Système d'Éclairage dynamique :** Gestion des ombres et des sources de lumière en temps réel.
- **Bot :** Des agents autonomes capables de viser, tirer et se déplacer pour fuir les zombies.
- **Gestion d'entités massive :** Le jeu doit gérer des milliers d'entités simultanées (Zombies, balles, particules de sang, débris).

Scénario de simulation : La boucle principale du jeu fait apparaître des vagues de zombies dont le nombre croît linéairement. Les entités "Zombies" cherchent constamment le chemin le plus court vers les "Bots" ou le "Joueur". Cette logique de recherche de cible (*Pathfinding* ou simple vecteur directionnel) combinée à la mise à jour des positions constitue le coeur de la charge de calcul.

1.2 Instructions de Compilation et d'Exécution (README)

Prérequis

- Compilateur C++ supportant le standard C++17.
- CMake (version 3.18 ou supérieure).
- Bibliothèque SFML (développement) installée.

Procédure de compilation

Depuis original_app ou depuis accelerate_app, exécuter les commandes suivantes dans un terminal :

```
# Si pas déjà fait
mkdir build
cd build
# Build et lancement du jeu
../run.sh
```

1.3 Informations supplémentaires

Avant d'avoir commencer à optimiser le jeu, nous avons pris un peu de temps pour le modifier. Ainsi, nous devrons optimisés un jeu plus gourmand, car nous avons modifier les points suivants :

- Nombre de zombies par vagues augmentés
- Champs de vision du joueur augmentée
- Cadence de tir augmentée et nombre de balles presque illimités
- Changement de la physique des balles

2 Stage 2 - Analyse des goulots d'étranglement

2.1 Temps d'exécution et Profilage

L'application *ZombieV* est une simulation intensive où le nombre d'agents évolue dynamiquement. Nous avons instrumenté le code (ajout de mesures temporelles dans `main.cpp`) pour relever le temps d'exécution par frame en fonction du nombre d'entités.

Voici les deux mesures que nous avons effectué :

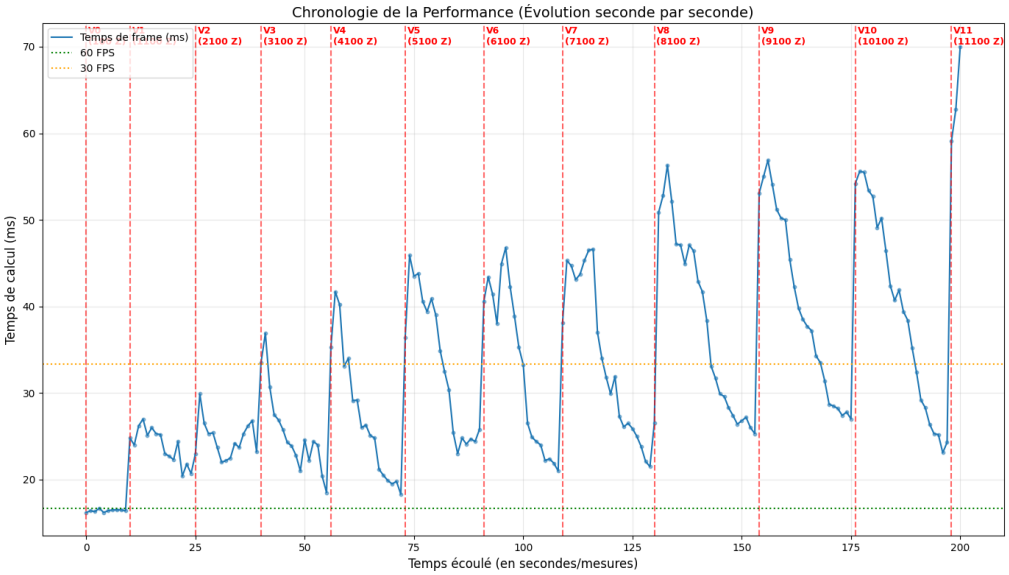

Figure 1: Mesure 1 - FPS par rapport au nombre de zombies - sans hunter (uniquement bot)

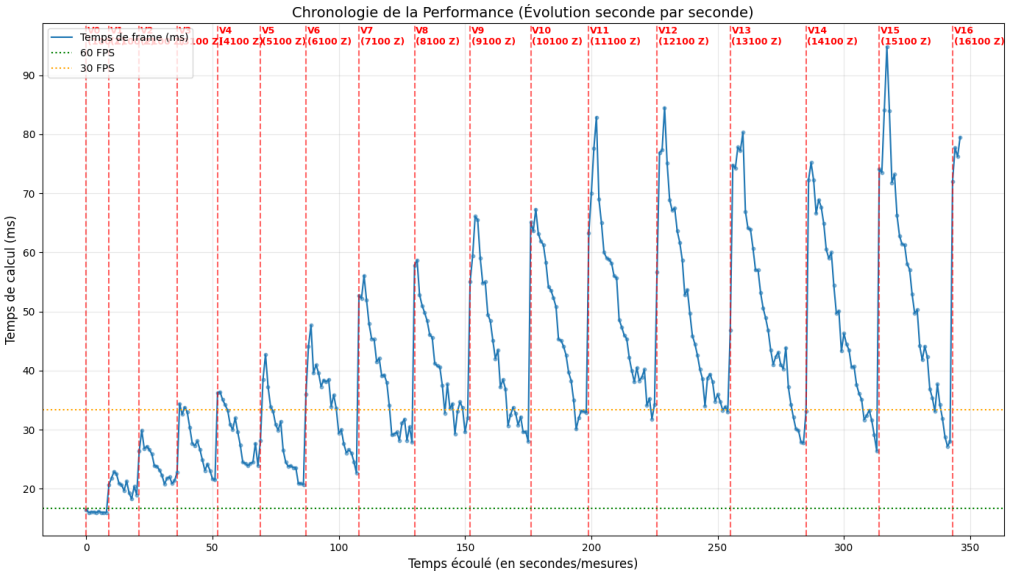

Figure 2: Mesure 2 - FPS par rapport au nombre de zombies - avec hunter

Afin de mesurer les performances de notre jeu, nous avons décidé d'effectuer deux mesures sur les FPS. La première mesure, 10 vagues de zombies, et la deuxième mesure 15 vagues de zombies. Les mesures de FPS ont été effectuées chaque seconde et plotter sur un graphe.

Avant d'analyser ces deux graphes, il est important de comprendre combien de zombies sont générés à chaque début de vague.

Pour la première vague, nous avons tout simplement 100 zombies. Ensuite pour chaque vague suivante, on a :

$$nbZombie = N_{vague} \times 1000 + 100$$

Analyze

On peut simplement découper les graphiques en trois phases distinctes

1. **Zone Fluide (Vagues 0 à 3 - Jusqu'à 3100 Zombies)** : La courbe reste majoritairement sous la ligne pointillée orange (33ms / 30 FPS). Le jeu est parfaitement jouable, souvent proche des 60 FPS (ligne verte) en fin de vague.
2. **Zone de Charge (Vagues 4 à 9 - De 4100 à 9100 Zombies)** : Le début de vague provoque un "lag spike" (pic de ralentissement) visible (entre 35ms et 60ms), ce qui fait tomber le jeu aux alentours de 15-20 FPS temporairement. Cependant, au fur et à mesure que le joueur "nettoie" la vague, le jeu redevient fluide (retour vers les 30ms).
3. **Zone Critique (Vagues 10 à 16 - De 10k à 16k Zombies)** : Des gros piques apparaissent mais montre une évolution moins linéaire.

Les pics atteignent 70ms à 90ms (soit environ 11 FPS). C'est très saccadé.

Même en fin de vague, le jeu a du mal à redescendre à une fluidité parfaite, mais cela est sûrement dû aux ajouts des différents éléments graphiques, lorsqu'un zombie est tué.

2.2 Analyse de la Complexité Algorithmique

L'analyse du code source nous permet d'identifier les classes de complexité suivantes :

1. **Logique des Bots ($O(B \times Z)$) :** La fonction `Bot::getTarget` parcourt la liste complète des zombies vivants pour identifier la cible la plus proche.

   $$Cost_{bot} = N_{bots} \times N_{zombies}$$

   Bien que N_{bots} soit constant (10), $N_{zombies}$ augmente de 1000 à chaque vague. À la vague 5, cela représente $10 \times 5100 = 51'000$ calculs de distance par itération.
2. **Mise à jour des Zombies ($O(Z)$) :** Chaque zombie calcule son déplacement et son orientation indépendamment dans `Zombie::update`. Bien que la complexité soit linéaire, le facteur multiplicatif est élevé (calculs trigonométriques `cos/sin`, racine carrée pour la norme).

3. **Physique et Collisions :** Le moteur physique (non détaillé ici mais appelé via `world.update()`) doit gérer les interactions spatiales. Si la détection est naïve, elle pourrait tendre vers $O(Z^2)$, mais nous considérons ici le coût de mise à jour des positions comme dominant pour l'instant.

Conclusion : La complexité globale de l'application est dominée par le terme linéaire $O(Z)$ avec une très forte constante due aux interactions Bots-Zombies et aux mises à jour physiques.

2.3 Identification des Bottlenecks

Les profils d'exécution et l'analyse statique désignent deux goulots majeurs :

- **Calcul de distances :** La recherche du plus proche voisin dans `Bot::getTarget` est extrêmement coûteuse. De plus, parcourir une liste chaînée ou un vecteur de pointeurs (`Zombie::getNext`) disperse les accès mémoire, provoquant de nombreux "Cache Misses" CPU.
- **Mise à jour des positions :** La fonction `Zombie::update` est appelée des milliers de fois. Les calculs de vecteurs (normalisation, produit scalaire pour l'orientation) saturent le thread principal.

2.4 Stratégie d'Accélération

Pour atteindre les objectifs du laboratoire, nous proposons la stratégie suivante :

1. **Accélération GPU (CUDA) - Priorité 1 :** Nous allons déporter le calcul des distances et la mise à jour des positions des Zombies sur le GPU.
    - *Pourquoi ?* Le problème peut-être gérer parallèlement. Chaque zombie ou bot peut être traité indépendamment. Le GPU arrive bien à effectuer des calculs vectoriels $(x, y, angle)$ sur de grands tableaux contigus.
    - *Cible :* Remplacer la boucle de mise à jour des zombies et la recherche de cibles des bots par des kernels CUDA.
2. **Accélération CPU (OpenMP) - Priorité 2 :** Nous utiliserons OpenMP pour paralléliser les boucles de traitement restantes sur le CPU (par exemple la gestion des collisions si elle reste sur CPU).

2.5 Performance Théorique

D'après la loi d'Amdahl, si nous parvenons à paralléliser la mise à jour des zombies (qui représente estimativement 70% du temps de calcul, soit une partie parallèle $p = 0.7$), l'accélération maximale théorique sur un GPU avec un nombre infini de coeurs ($N \to \infty$) est limitée par la fraction séquentielle restante ($s = 1 - p = 0.3$) :

$$S_{max} = \frac{1}{s} = \frac{1}{1 - 0.7} \approx 3.33\times$$

Cependant, ce calcul théorique ne prend pas en compte l'overhead $c(N)$ introduit par l'architecture. En intégrant le coût des transferts mémoire Host-Device nécessaires à chaque frame, nous visons une accélération réaliste comprise entre 1.5× et 2×.

3 Stage 3 - Accélération

Avant de commencer cette partie, je tiens à préciser que les différents tests n'ont pas été effectuées sur la jetson nano, mais sur un PC.

Voici la configuration du PC :

```
./deviceQuery
```

```
Detected 1 CUDA Capable device(s)

Device 0:  NVIDIA GeForce RTX 3050 Ti Laptop GPU
  CUDA Driver Version / Runtime Version          13.1 / 12.3
  CUDA Capability Major/Minor version number:    8.6
  Total amount of global memory:                 4096 MBytes (4294443008 bytes)
  (020) Multiprocessors, (128) CUDA Cores/MP:    2560 CUDA Cores
  GPU Max Clock rate:                            1485 MHz (1.49 GHz)
  Memory Clock rate:                             5871 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 2097152 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072,
                                                 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        102400 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

```
~$ nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Wed_Nov_22_10:17:15_PST_2023
Cuda compilation tools, release 12.3, V12.3.107
Build cuda_12.3.r12.3/compiler.33567101_0
```

3.1 Analyse des besoins mémoires

Actuellement, la structure du programme utilise des données dispersées (elles ne sont pas contigus en mémoire). Nous avons des objets U_2DBody éparpillées en mémoire et chaque objet contient un certain nombre d'information. Le jeu utilise une Pool d'objet.

- **Problème** : C'est très peu performant, car il faut que les tableaux et les accès mémoires soient contigus.
- **Localité** : Mauvaise, beaucoup de cache misses

Pour utiliser cuBLAS de manière perfomant, il faut donc aplatir les données. Au lieu d'avoir 10'000 objets, éparpillés en mémoire, contenants chacun 2 floats, on aura 2 grands tableaux de 10'000 floats. Cependant. cette opération d'applatissement bloque fortement nos perfomances, la meilleure solution serait de revoir entièrement la manière de stocker les entités du jeu, afin qu'elle soit toute de manière contigus, on pourrait créer un vecteur par entité par exemple, ce qui faciliterait beaucoup l'accès mémoire et l'envoi des données au GPU.

Imaginons que nous avons 10'000 zombies, chaque vecteur pèse 10'000 * 4 octets = 40 KB. C'est vraiment pas beaucoup pour un GPU qui a des Go de mémoire.

Normalement, avec cette mise en place, on aura des accès plus rapide en mémoire et des calculs qui pourront se faire sans soucis en parallèle.

3.2 Choix d'accélération GPU : cuBLAS

Puisque nous sommes obligés de choisir une libraire, il est évident que le meilleur choix est cuBLAS, puisque cuDNN est adapté pour les réseaux de neuronnes. cuDNN ne s'applique pas dans notre cas.

Comment adapter la physique à cuBLAS ? L'intégration de Verlet (updatePosition) est une opération vectorielle. Formule actuelle :

$$Pos = Pos + (Pos - LastPos) + Acc * dt$$

On peut décomposer cela en opérations vectorielles standard (SAXPY : Single Precision A*X + Y) :

1. Calcul de la vélocité : Vel = Pos - LastPos
   - Opération cuBLAS : cublasSaxpy (alpha=-1.0, X=LastPos, Y=Pos). Attention, cela modifie Pos, il faut ruser ou utiliser des buffers temporaires.
2. Application de l'accélération : Vel = Vel + Acc * dt
   - Opération cuBLAS : cublasSaxpy (alpha=dt, X=Acc, Y=Vel).
3. Mise à jour Position : Pos = Pos + Vel

- Opération cuBLAS : cublasSaxpy (alpha=1.0, X=Vel, Y=Pos).

Nous allons utiliser cuBLAS Level 1 BLAS functions (opérations vecteur-vecteur) pour effectuer l'intégration physique de toutes les entités en parallèle.

3.3 Optimisation avec cuBLASS

L'implémentation de l'optimisation GPU se déroule en trois phases distinctes : la préparation des données (transformation *Array of Structures* vers *Structure of Arrays*), le calcul vectoriel via cuBLAS, et la réinjection des résultats.

3.3.1 Préparation et Transfert des données (CPU)

Comme identifié dans l'analyse mémoire, les données des objets `U_2DBody` sont dispersées. Dans la méthode `U_2DCollisionManager::update`, nous effectuons une passe de linéarisation :

- **Extraction et Friction :** Nous itérons sur tous les corps dynamiques pour extraire leurs positions (pX, pY), anciennes positions $(oldX, oldY)$ et accélérations (aX, aY) vers des vecteurs `std::vector<float>` contigus.
- **Pré-calcul CPU :** Une optimisation notable est effectuée ici : la friction $(-10 \cdot \text{vélocité})$ est appliquée directement sur le vecteur d'accélération côté CPU. Cela permet de simplifier le kernel GPU qui n'aura plus qu'à effectuer des sommes, sans avoir à recalculer la vélocité pour la friction.
- **Gestion Mémoire :** La classe `CudaPhysics` gère l'allocation GPU dynamiquement. Pour éviter des réallocations coûteuses (`cudaMalloc`) à chaque frame si le nombre de zombies varie peu, une capacité tampon (taille requise + 1000) est maintenue.

Une fois les vecteurs prêts, ils sont envoyés dans la mémoire du device via `cudaMemcpy`.

3.3.2 Implémentation de Verlet avec SAXPY

Le cœur du calcul réside dans la méthode `CudaPhysics::updatePositions`. Puisque cuBLAS est une librairie d'algèbre linéaire, nous ne pouvons pas écrire de kernel C++ personnalisé (car demandé dans la consigne d'utiliser une lib), nous devons composer l'algorithme à l'aide de l'opération standard **SAXPY** ($Y = \alpha X + Y$).

Pour respecter la formule d'intégration de Verlet ($Pos_{new} = Pos + (Pos - OldPos) + Acc \cdot dt$), nous utilisons un buffer temporaire (`d_tempX/Y`) pour accumuler les résultats intermédiaires. Voici la séquence d'opérations exécutée pour chaque axe (exemple pour l'axe X) :

1. **Initialisation :** Copie de la position actuelle dans le buffer temporaire via `cudaMemcpy` (DeviceToDevice).
$$Temp \leftarrow Pos$$
2. **Calcul de la vélocité :** Soustraction de l'ancienne position ($\alpha = -1.0$).
$$\texttt{cublasSaxpy}(\alpha = -1, X = OldPos, Y = Temp) \implies Temp = Pos - OldPos$$

3. **Application de l'accélération :** Ajout de l'accélération pondérée par le temps ($\alpha = dt$).

$$\texttt{cublasSaxpy}(\alpha = dt, X = Acc, Y = Temp) \implies Temp = Vel + Acc \cdot dt$$

4. **Mise à jour finale :** Addition du déplacement calculé à la position courante ($\alpha = 1.0$).

$$\texttt{cublasSaxpy}(\alpha = 1, X = Temp, Y = Pos) \implies Pos_{new} = Pos + \text{Déplacement}$$

Cette approche permet de traiter des dizaines de milliers d'entités en parallèle en exploitant la bande passante mémoire du GPU, bien que l'utilisation de cuBLAS impose plus d'appels mémoires (lectures/écritures successives dans `d_temp`) qu'un kernel CUDA natif qui ferait le calcul en registres.

3.3.3 Récupération et Mise à jour

Une fois les calculs terminés, les vecteurs de positions mis à jour sont rapatriés sur le CPU. Le CPU met ensuite à jour les positions du zombie.

3.4 Optimisation avec OpenMP

L'utilisation de la librairie OpenMP nous a permis de paralléliser les traitements de "l'IA" (bot) sur le CPU, spécifiquement pour la recherche de cibles (Raycasting et calculs de distance). Nous avons implémenté deux stratégies différentes selon la nature des données.

3.4.1 Stratégie 1 : Réduction Manuelle (Bot::getTarget)

La recherche de la cible la plus proche pour les Bots implique de parcourir l'intégralité du vecteur de Zombies (pouvant contenir des milliers d'entités). Une boucle simple avec une section critique (`#pragma omp critical`) à l'intérieur serait désastreuse pour les performances à cause de la contention.

Nous avons opté pour une approche de réduction manuelle :

- **Mémoire locale par thread :** Nous allouons un tableau de structures `SearchResult` dont la taille correspond au nombre de threads actifs. Cela permet à chaque thread de travailler sur sa propre variable `localMinDist` et `localTarget` sans verrou.
- **Phase Parallèle :** Via `#pragma omp parallel`, les threads se répartissent le parcours du tableau des zombies. Chaque thread ne retient que le meilleur candidat qu'il a vu personnellement.
- **Agrégation (Merge) :** Une fois la région parallèle terminée, le thread principal (Master) parcourt le petit tableau des résultats locaux pour déterminer le minimum global.

Cette méthode permet d'utiliser 100% des capacités multicœurs sans aucun goulot d'étranglement lié à la synchronisation durant la boucle de calcul.

3.4.2 Stratégie 2 : Clause Collapse (Zombie::_getTargetOptimized)

Pour les Zombies cherchant des hunters, nous avons combiné une optimisation algorithmique (grille spatiale) avec OpenMP. Au lieu de chercher dans tout le monde, le zombie ne cherche

que dans les cellules voisines de la grille physique (5x5 cellules autour de lui, mais ce paramètre est modifiable).

L'implémentation utilise des fonctionnalités d'OpenMP pour gérer les boucles imbriquées :

- **Collapse de boucles :** La directive `collapse(2)` est utilisée pour fusionner les deux boucles for imbriquées (parcours en X et en Y de la grille) en une seule boucle linéaire parallélisable. Cela augmente le nombre d'itérations disponibles pour les threads et améliore l'équilibrage de charge.
- **Réduction :** Contrairement aux Bots, nous utilisons ici la clause `reduction(min:closestDist2)`. OpenMP gère automatiquement la compétition entre les threads pour trouver la distance minimale, simplifiant grandement le code par rapport à la méthode manuelle.

Code implémenté :

```
#pragma omp parallel for collapse(2) reduction(min:closestDist2) shared(closestTarget)
for (int dx = -SEARCH_RADIUS; dx <= SEARCH_RADIUS; dx++)
{
    for (int dy = -SEARCH_RADIUS; dy <= SEARCH_RADIUS; dy++)
    {
        // ... Logique de verification de la cellule ...
    }
}
```

Pour cette partie, on aurait pu utiliser la même méthode que pour la recherche de target Zombie pour un hunter. Cependant, puisqu'il n'y a pas beaucoup de hunters ($\tilde{1}0$), cela serait overkill d'allouer des threads pour si peu de hunters.

4 Stage 4 - Analyse des résultats

Voici les mesures effectuées pour 15 vagues de zombies, comme pour le stage 1 :

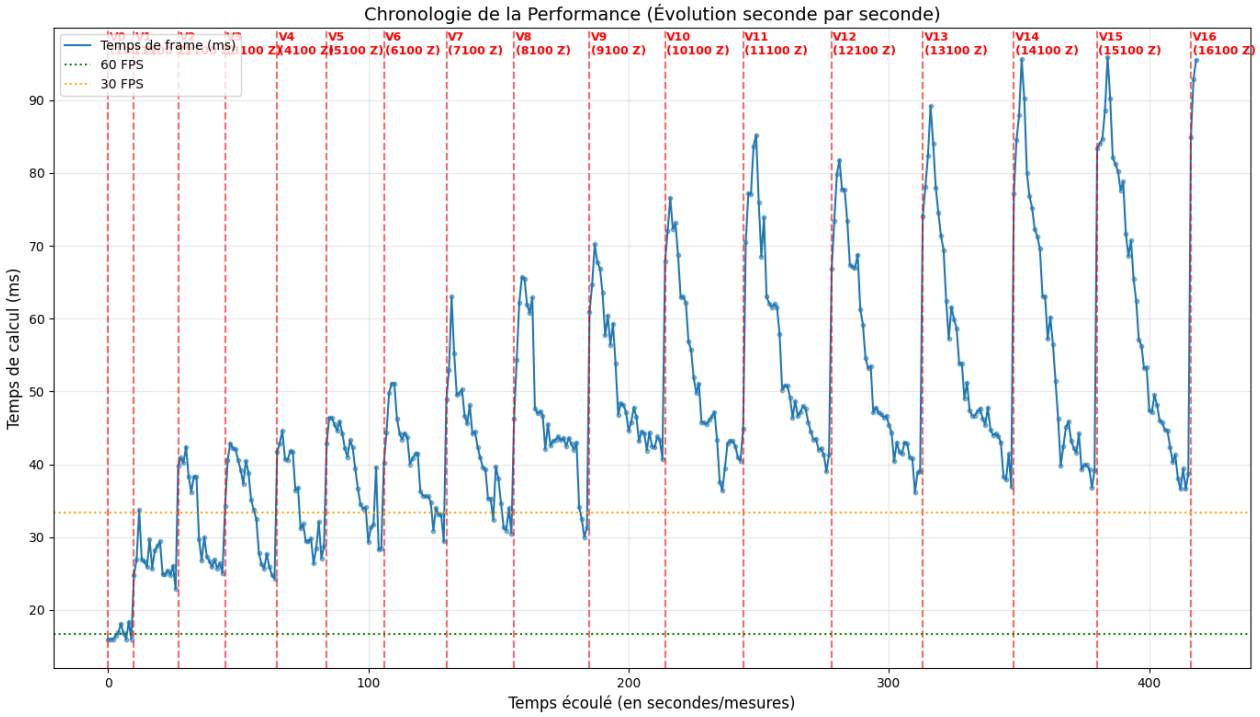

Figure 3: Mesure 3 - FPS par rapport au nombre de zombies - avec hunter

4.1 Analyse de la performance

Contrairement aux attentes, l'analyse comparative des graphiques de performance révèle que la version "optimisée" (GPU + OpenMP) présente des performances similaires, voire légèrement inférieures, à la version naïve.

- **Observation :** Sur le graphique sans optimisation (Fig. 2), le pic de temps de calcul à la vague V16 (16'100 zombies) atteint environ **80ms**. Sur la version optimisée (Fig. 3), ce même pic atteint **95ms**. De plus, le temps minimum entre les vagues est plus élevé dans la version optimisée ($\approx$ 35ms contre $\approx$ 25ms).
- **Bottleneck :** Le gain théorique offert par le parallélisme massif du GPU pour les calculs physiques (intégration de Verlet) est totalement éclipsé par l'**overhead de gestion mémoire**. L'architecture POO actuelle du projet stocke les entités de manière dispersée (*Array of Structures*). Pour utiliser cuBLAS, nous devons effectuer à chaque frame une sérialisation coûteuse (*Structure of Arrays*) sur le CPU :
  1. Allocation et remplissage des `std::vector` (linéarisation) : $O(N)$.
  2. Transfert CPU vers GPU $\rightarrow$ latence PCI-e.
  3. Calcul GPU $\rightarrow$ très rapide.
  4. Transfert GPU vers CPU.

5. Réécriture des résultats dans les objets dispersés : $O(N)$.

- **Lien avec la théorie (Loi d'Amdahl) :** Selon la loi d'Amdahl, l'accélération potentielle est limitée par la fraction séquentielle du programme. Ici, la préparation des données (séquentielle sur le CPU) prend plus de temps que le calcul physique lui-même. En déplaçant le calcul sur le GPU, nous avons paradoxalement augmenté la portion séquentielle (temps de copie + overhead API) par rapport au temps de calcul pur, ce qui explique la dégradation de performance.
- **Analyse NVIDIA Nsight :** L'utilisation d'un profiler comme NVIDIA Nsight confirmerait que le GPU est sous-utilisé. On observerait probablement sur la timeline :
  - De larges "trous" d'activité GPU correspondant à la préparation CPU.
  - Un temps de transfert mémoire (`cudaMemcpy`) supérieur au temps d'exécution des kernels (`cublasSaxpy`).
  - Une faible occupation des cœurs CUDA, car les vecteurs sont traités par de multiples appels de noyaux très courts (SAXPY niveau 1), limités par la bande passante mémoire (*Memory Bound*) plutôt que par la puissance de calcul (*Compute Bound*).

4.2 Pistes d'améliorations futures

Au vu de ces résultats, il apparaît que l'optimisation isolée d'une fonction mathématique ne suffit pas si l'architecture des données n'est pas adaptée. Voici ce qui aurait dû être fait différemment :

4.2.1 Changement de Paradigme : Data-Oriented Design (DOD)

La POO est ici l'obstacle majeur. La solution idéale serait de refaire le stockage des entités pour utiliser simplement une architecture de tableaux contigus natifs persistants. **Impact :** Si les positions (pX, pY) et vitesses existaient déjà sous forme de vecteurs contigus en mémoire, l'étape de linéarisation disparaîtrait. Le coût deviendrait uniquement celui du transfert PCI-e.

4.2.2 Kernels CUDA Personnalisés vs cuBLAS

L'utilisation forcée de cuBLAS est sous-optimale pour la physique de jeu.

- **Problème cuBLAS :** Pour faire $Pos = Pos + Vel + Acc \cdot dt$, nous devons lancer 3 appels `cublasSaxpy`. Cela implique 3 lectures/écritures complètes de la mémoire globale du GPU.
- **Solution Custom Kernel :** Un kernel CUDA écrit à la main pourrait effectuer toute l'intégration en une seule opération de lecture/écriture par particule, réduisant la pression sur la bande passante mémoire par un facteur 3.

4.2.3 Persistance GPU

Au lieu d'envoyer les données à chaque frame, nous pourrions garder les positions sur la mémoire du GPU en permanence. Seules les modifications externes (collisions, inputs) seraient envoyées, et le rendu pourrait se faire directement depuis les buffers GPU.

5 Conclusion Générale

Ce laboratoire nous a permis d'explorer deux approches distinctes du parallélisme : le multi-threading CPU avec OpenMP et l'accélération matérielle GPU avec cuBLAS.

D'un point de vue fonctionnel, l'intégration est un succès : nous avons réussi à déporter les calculs lourds de l'IA (recherche de cibles) sur les coeurs du processeur grâce aux directives de réduction et de *collapse* d'OpenMP. De même, le moteur physique a été porté sur le GPU en réécrivant l'intégration de Verlet via des opérations d'algèbre linéaire (SAXPY).

Cependant, d'un point de vue performance, les résultats mettent en lumière une réalité fondamentale du calcul haute performance : l'architecture des données prévaut sur la puissance brute de calcul. La baisse de performance observée démontre que le coût de transfert et de restructuration des données (d'une approche Orientée Objet vers des vecteurs contigus) dépasse largement le gain offert par le GPU pour ce volume de données.

En définitive, ce projet illustre que pour tirer parti des GPU, il ne suffit pas de paralléliser le code existant. Une refonte structurelle vers une approche orientée données est indispensable pour éliminer les goulots d'étranglement mémoire qui brident aujourd'hui nos optimisations.