UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

Algorisme genètic & Problema dels N cossos

Estudi sobre l'aplicació de CUDA en el càlcul del moviment de cossos

Targetes Gràfiques i Acceleradors (TGA)

Grup 13

Jan Mas rovira

Andrés Mingorance López

Albert Puente Encinas

Facultat d'Informàtica de Barcelona (FIB)

Juny de 2015

1 Introducció

En aquest projecte s'han desenvolupat en CUDA dos problemes computacionalment complexos, que ofereixen resultats pobres en eficiència si s'intenten abordar de forma seqüencial: l'algorisme genètic, i el problema de les interaccions entre N cossos.

Per poder abastar un ventall més ampli dins de les funcionalitats que ofereixen les targetes gràfiques, hem volgut realitzar una aproximació a aquests dos problemes des de la perspectiva de moviments de punts en l'espai. D'aquesta manera, s'ha pogut crear un programa basat en OpenGL que permet visualitzar els resultats de les execucions tant seqüencials com paral·lelitzades amb CUDA mitjançant shaders propis i vertex buffer objects.

2 Primer cas: l'algorisme genètic

2.1 Definició del problema

Comencem donant una definició del problema que volem resoldre: Donat un conjunt de punts P amb N punts amb amb posicions inicials a $\mathbb{R}^3$, una funció de cost $c : P \to \mathbb{R}$, i una funció de restricció $r : (E, p1, p2) \to \{0, 1\}$ que indica, donat un estat global E, si un punt en la posició $p1$, es pot moure a la posició $p2$ en un pas.

Volem minimitzar $c(P)$ amb M passos tenint en compte que a cada pas s'ha de complir la restricció per tots els punts del conjunt, és a dir, per tot i: $r(E, p_i, p'_i) = 1$. En el nostre cas, la funció de cost serà una funció de distància, ja sigui respecte un punt, o una funció a l'espai. Quant a la funció de restricció, farem servir una funció que limiti el desplaçament d'un punt en un pas i que tingui en compte col·lisions, a fi de simular el moviment físic d'un conjunt de punts. Proposem dues versions. L'escenari en el que es mouen els punts té unes quantes esferes de mida variable que impedeixen el moviment de punts a través d'elles.

Addicionalment a la restricció anterior, cadascun dels punts té volum propi, pel que també es tindran en compte col·lisions entre punts.

2.2 Algorisme

Com molts problemes d'optimització, no existeix un algorisme que doni una solució òptima en un temps raonable, així que cal recórrer a algorismes de cerca local.

Hi ha moltes varietats d'algorismes de cerca local, on molts d'ells estan basats en exploració d'arbres: A*, Ida, Hill climbing, etc. Aquests algorismes no propicien la seva paral·lelització, ja que fan canvis de manera iterativa sobre una solució, cosa que crea dependències. L'algorisme que hem triat, en canvi, s'estructura de manera adequada per poder fer-ne una bona paral·lelització: un algorisme genètic.

La idea general d'un algorisme genètic és la següent.

- Un objectiu: volem trobar un cert element que satisfaci, en la mesura del possible, un cert objectiu. La part encarregada d'avaluar com d'aprop està un element d'assolir tal objectiu es coneix com a **heurístic**, i correspon a la funció de cost mencionada anteriorment.

- Població inicial: conjunt d'elements (individus) amb els que comencem a refinar, amb la intenció que un d'ells compleixi l'objectiu mencionat anteriorment. Aquesta població pot ser part de l'entrada del problema o generada aleatòriament.

- Selecció: A cada pas generacional, es seleccionen els millors individus (els que estan més a prop de complir l'objectiu). Els que no són prou bons, moren.

- Reproducció: d'entre els individus supervients, se n'escullen parelles. Cada parella d'individus té un fill, que s una barreja dels dos pares. La nova generació omple el buit que havien deixat els morts a la fase de selecció.

- Mutacions: Les mutacions són modificacions esporàdiques en un individu. Trobar un balanç adequat en la probabilitat de mutar és important, ja que una probabilitat alta pot causar inestabilitat, mentre que una probabilitat baixa pot provocar que l'algorisme s'estanqui.

- Terminació: L'execució acaba quan s'ha arribat a una certa fita de cost o quan s'arriba a un cert límit de generacions. El millor individu d'aquesta última població es porposa com a solució final.

2.2.1 Versió seqüencial

A continuació comentarem les parts més rellevants de la implementació seqüencial de l'algorisme.

Actors:

- Point: simple struct que representa les coordenades d'un punt (x,y,z). En el cas que tinguin volum, aquest és comú a tots.

- PointSet: vector de Point's. En el context de l'AG és un individu.

- Population: vector de PointSet's. En el context de l'AG és la població[1].

- Obstacle: cada obstacle té unes coordenades i un radi propi. Són immòbils.

Passos:

- Població inicial: es genera aleatòriament. Respectar les restriccions de col·lisió es fa per prova i error.

- Selecció: es fa una simple ordenació amb l'algorisme quicksort i es seleccionen els millors.

- Reproducció: es combinen dos individus, és a dir, dos PointSet's. Cada fill obté els seus punts d'un dels dos pares, triat a l'atzar. Com que s'assumeix que els punts dels pares estaran en posició correcta (és a dir, sense col·lisionar), no cal fer cap comprovació de col·lisions.

- Mutació: les mutacions són a nivell de PointSet. Cadascun dels punts té una certa probabilitat de mutar. Existeixen dos tipus de mutacions:

  - Moviment aleatori: el punt es desplaça una certa distància en una direcció aleatòria. Cal tenir en compte col·lisions.

  - Barreja: el punt pren per referència un altre punt a l'atzar del conjunt, llavors es desplaça sobre la recta que els uneix una certa distància. També cal tenir en compte col·lisions.

Esquema del bucle principal:

```
generateInitialPopulation()
while (true) {
        mutation()
        selection()
        if checkEndCondition() exit
        reproduction()
}
```

[1] Vigilar de no confondre individu i població: tot i que un PointSet estigui format per molts punts, és un sol individu. La població la forma un conjunt de PointSets.

```
8 return bestFit()
```

El codi d'aquest algorisme es pot consultar en l'arxiu `genetic.c`.

2.2.2 Versió paral·lela (1)

En la versió paral·lela l'estratègia a seguir és equivalent a la seqüencial, per tant, només comentarem les deferències respecte a la versió seqüencial.

- QuickSort dinàmic: durant la fase de selecció, necessitem avaluar cadascun dels PointSet's segons l'heurístic i ordenar-los. Per fer-ho, hem optat per una implementació amb paral·lelisme dinàmic. Comentem-ne algunes particularitats:
  - Els swaps es fan fent servir un vector auxiliar d'índexos. Vam intentar-ho fer amb punters però no va acabar d'anar bé.
  - Quan s'arriba a una certa profunditat o el vector que s'ha d'ordenar és prou curt, es fa una ordenació per selecció iterativa per evitar nivells de profunditat excessius.
  - Les crides recursives es fan per streams.
- Reproducció i Mutació: durant la fase de mutacio cal fer una distinció entre les dues versions de col·lisions que hem proposat. Fixem-nos que si hem de tenir en compte les col·lisions entre punts llavors la funció de restricció depèn del PointSet on és el punt, i com que la mutació modifica el PointSet, llavors les mutacions dins un PointSet han de ser calculades seqüencialment. Aixo és un problema ja que es desaprofita el potencial de paral·lelització que ofereix el hardware. En l'altra versió, sense col·lisions entre punts, la funció de restricció només depèn dels obstacles estàtics de l'escenari, i per tant, és completament paral·lelitzable. En la fase de reproducció tenim un problema anàleg, però aquesta vegada la dependència és a nivell de població.
- Evaluació: donat que la funció d'evaluació que utilitzem és la distància, i aquesta és continua i monòtona, en la versió paral·lela s'utilitza la distància al quadrat (per evitar calcular l'arrel quadrada, que és computacionalment cara).

En la versió paral·lela s'han utilitzat vàries GPUs executant l'algorisme genètic paral·lelitzat simultàniament; en tantes com n'hi hagi de disponibles. A cada iteració, per decidir el millor conjunt de punts d'entre totes les poblacions, es comparen també entre les diferents GPUs. S'ha escollit aquesta manera d'executar en múltiples GPUs per mantenir una independència entre les diferents evolucions de l'algorisme entre les targetes. D'aquesta manera, es dona la possibilitat de que una població que hagi mutat de forma diferent a una altra a l'inici de l'execució del programa, encara que tingui pitjor puntuació, potser acabarà guanyant arribats a les últimes iteracions del programa ja que ofereix una altra varietat genètica en el ventall de solucions. S'ha procurat que l'ús de multiples GPUs no ofusqui un codi ja prou complex mitjançant un `DEFINE` de `c`, anomenat `ForEachGPU`, que es pot fer servir per indexar els vectors de punters que contenen totes les direccions de memòria de les estructures en cadascuna de les GPUs:

```
1 ForEachGPU {
2     ...
3     kernel_X<<<....>>>(vector_punters[GPU_X],...)
4     ...
5 }
```

Una de les dificultats més grans que s'han hagut de tractar ha estat poder generar nombres aleatòris de forma paral·lela en una targeta gràfica. Per a poder-ho fer, ha calgut inicialitzar un vector en cada targeta gràfica contenint una llavor diferent per a cada kernel que s'hagi acabat executant. A continuació es mostra el codi en detall:

```
__global__ void setup_kernel(curandState *state, int seed) {
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    /* Each thread gets same seed, a different sequence
       number, no offset */
    curand_init (seed, id, 0, &state[id] );
}

void randomSetup() {
    srand(SEED); // Llavor fixa de la CPU per poder reproduir resultats
    seeds = (int*) malloc(nGPUs*sizeof(int));
    ForEachGPU {
        seeds[GPU_X] = rand(); // Llavors de cada GPU (determinista)
    }

    devStates = (curandState**) malloc(nGPUs*sizeof(curandState*));

    ForEachGPU {
        cudaSetDevice(GPU_X);
        cudaMalloc((void **)&devStates[GPU_X], N * sizeof(curandState));
        setup_kernel<<<nBlocks, nThreads>>>(devStates[GPU_X], seeds[GPU_X]);
        checkCudaError((char *) "setup random kernel");

    }
    ForEachGPU {
        cudaSetDevice(GPU_X);
        cudaDeviceSynchronize();
    }
}
```

El codi d'aquest algorisme es pot consultar en l'arxiu `genetic.cu`.

2.2.3 Versió paral·lela (2)

Tal com hem comentat, el fet de considerar col·lisions entre punts d'un mateix individu és una restricció a l'hora de paral·lelitzar les etapes de l'algorisme. Per exemple, si volem mutar un individu —és a dir, moure els punts que el formen— ens hem d'assegurar que no hem mogut dos punts a un lloc de l'espai tal que estan massa propers i col·lisionen. Una manera de solucionar aquest fet és distingir l'espai de sortida i l'espai de mutació: a l'hora de mutar un punt, comprovem que aquest no col·lisiona amb els que ja han mutat (estan a l'espai de mutació). Existeix un cas crític, a més, on una sèrie de punts muten de manera que envolten l'espai al voltant d'un punt que encara no ha mutat. Quan arribi el moment de mutar tal punt, es trobarà que no pot ni moure's a cap posició del voltant, ni quedar-se quiet, ja que els punts que el rodegen col·lisionen amb ell. Per evitar aquest fet, a l'hora de mutar un punt també es comprova que no col·lisioni amb cap dels punts que queden per mutar (espai de sortida).

Aquest procés és costos; cada punt ha de comprovar tots els seus veïns, cosa que implica un cost total de $O(n^2)$ només per un pas de mutació. A més, la seqüencialitat és inherent al fet de realitzar tals comprovacions; cal que els punts mutin un a un i ordenadament per poder realitzar-les.

Per tal d'intentar aprofitar al màxim la potència de càlcul de les targetes gràfiques, hem creat una segona versió de l'algorisme genètic on es relaxa la restricció de col·lisions entre punts d'un mateix individu, mantenint així exclusivament les col·lisions amb obstacles. Aquest fet permet no només paral·lelitzar a nivell de població (Population) sinó també d'individu (PointSet).

El resultat és un codi on tots els kernels que feien treball per individu, passen a fer treball per punt. Les funcions noves són anàlogues a la primera versió de l'algorisme genètic, i en conserven el nom però li

afegeixen `Flat` al final, per representar que el que abans es feia en dos nivells diferents (població-individu) s'ha aplanat (*flattened*) en un de sol.

Per motius didàctics i per intuïtivitat, hem enfocat el codi dels kernels de manera diferent:

- Encara que paral·lelitzem tots els punts alhora, la distribució lògica d'aquests en PointSets encara és present, i cal tenir-la en compte per produir un resultat correcte. Aquest fet encaixa molt bé amb l'ús de mides de grid i block bidimensionals. Un exemple de crida de kernel es troba a continuació:

```
int n_threads = 32; //sqrt(1024)
int width = N / n_threads;
// N_POINTS is the number of points PER pointset
int height = N_POINTS / n_threads;
// kernel
dim3 gridSize(width, height, 1);
dim3 blockSize(n_threads, n_threads, 1);
// width*n_threads * height*n_threads == total of points
kernel_mutateFlat<<<gridSize, blockSize>>>(gpu_P, gpu_Q,
                                           gpu_obstacles, devStates);
```

- A més a més, degut a que els punts s'han de comportar de manera diferent en funció del PointSet al que pertanyin (per exemple, cada PointSet té al seva pròpia probabilitat de mutar), els kernels segueixen tots una estructura similar on cada thread corresponent al primer element (Point) de cada individu (PointSet) realitza un petit càlcul addicional necessari per tots els altres, i ho guarda en memòria compartida. A continuació, un exemple extret del kernel de mutació. Tots els punts d'un mateix pointset han de mutar de la mateixa manera, i aquesta manera la calcula el primer punt:

```
//...
//pointset al que pertany el thread
int ps_id = blockIdx.x * blockDim.x + threadIdx.x;
//punt dins del pointset al que correspon el thread
int point_id = blockIdx.y * blockDim.y + threadIdx.y;
// decide, for each pointset, its mutation type
// 0 = no mutation, 1 = mix, 2 = randomMove
__shared__ int mutate_pointset[N]; // store if pointsets need to be mutated
if (point_id == 0) {
    if (cuda_randomChoice(POINT_SET_MUTATION_PROB, &state[ps_id])) {
        //50/50 of mixing or random moving
        if (cuda_randomChoice(0.5, &state[ps_id])) mutate_pointset[ps_id] = 1;
        else mutate_pointset[ps_id] = 2;
    }
    else mutate_pointset[ps_id] = 0;
}
__syncthreads();
//...
```

El codi d'aquest algorisme alternatiu es pot consultar en l'arxiu `genetic_alternatiu.cu`. No obstant no és funcional ja que, per motius que desconeixem, genera un `Unknown error`[2] tot just començar la seva execució en el servidor remot.

[2]Concretament, genera un error en un `cudaMalloc` idèntic a la versió principal de l'algorisme genètic.

2.3 Resultats

A continuació es mostra el temps d'execució de cadascuna de les fases de l'algorisme en les dues versions, la seqüencial i la paral·lela. S'han utilitzat una població de 4096 individus amb 1024 punts cadascun i s'han comprovat tots els tipus de col·lisions. Cal fer notar que la versió paral·lela (gràcies a tenir multiGPU) executa simultàniament 4 algorismes genètics.

	Versió seqüencial	**Versió paral·lela**
Generació inicial	22,5819	4,7922
Mutacions	5,6815	1,6877
Evaluacions	0,0389	0,0111
Ordenació	0,0108	0,0149
Reproduccions	0,0799	0,0144

Taula 1: Temps d'execució de les diferents fases en una iteració amb 4096 conjunts de 1024 punts.

Els resultats obtinguts no mostren un speed-up tan gran com s'esperava (tal com ha aconseguit el programa dels N-cossos). No obstant, creiem que donada la naturalesa aleatòria del problema, on cada thread pot tenir molts camins d'execució, es planteja una dificultat addicional a l'hora de poder homogeneïtzar el comportament dels threads en els warps i aconseguir un rendiment més gran.

En la següent figura es mostra la millora que representa la implementació en CUDA respecte la seqüencial segons la mida de la població de l'algorisme genètic (s'utilitza una població de 128 punts per cadascun dels N individus):

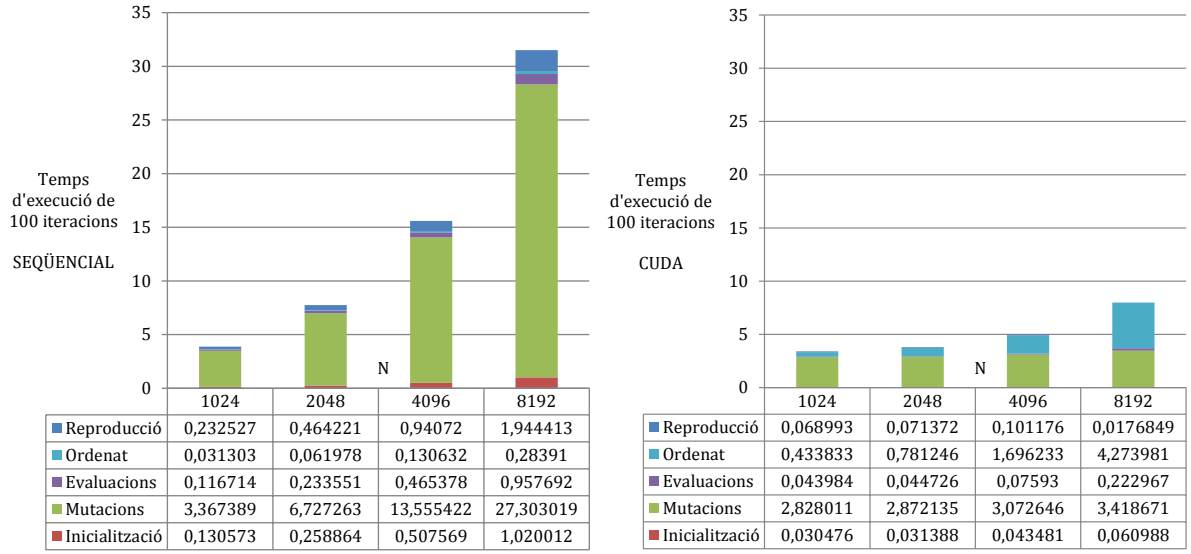

Figura 1: Comparació entre l'algorisme seqüencial i el paral·lel en les diferents fases segons la mida de la població.

Tal com es pot observar, la principal reducció del temps ve de realitzar les mutacions de forma paral·lela. Totes les altres fases de l'algorisme passen a significar molt poc respecte el temps total, excepte la ordenació. Realitzar la ordenació en les quatre GPUs a la vegada sobre estructures de dades grans mitjançant paral·lelisme dinàmic provoca un overhead que acaba fent que sigui més lent que el seqüencial.

Sobre l'equilibri entre el nombre de threads i de blocs s'ha observat que amb un bon balanç s'obtenen resultats lleugerament millors, però en cap cas afecta molt als temps d'execució.

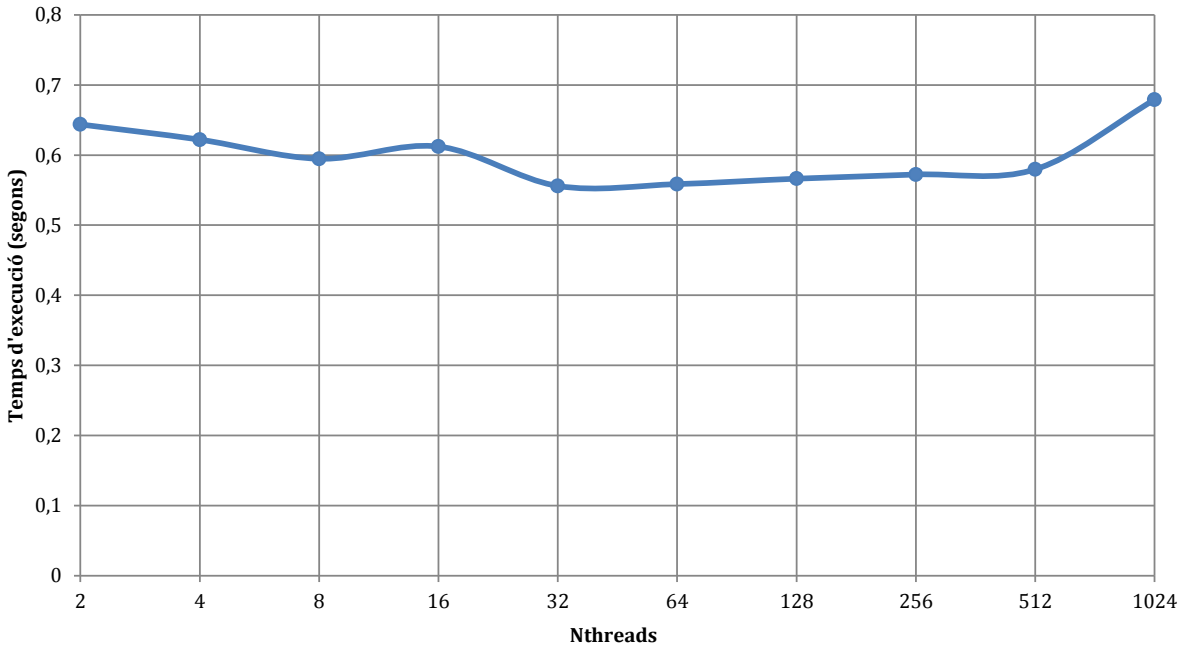

Figura 2: Temps d'execució total en funció del nombre de threads (4096 conjunts de 128 punts en 10 iteracions)

En el següent enllaç es mostra el resultat d'una execució amb 4096 conjunts de 1024 punts cadascun utilitzant 4 GPUs en 1200 iteracions. En aquesta execució, per cada iteració, el nuvol de punts que es dibuixa és el millor individu de cada població d'entre les 4 targetes gràfiques. Els núvols de punts busquen acostar-se a l'esfera vermella centrada en l'origen. Com a resultat d'aquest criteri d'avaluació de les mutacions, l'algorisme genètic acaba produint un núvol de punts que rodeja la susdita esfera.

`https://youtu.be/EMdzvuBNcTI`

3 El problema dels N-cossos

3.0.1 Definició del problema

Siguin N punts (pilotes) en l'espai, el problema dels N-cossos consisteix en calcular les seves interaccions i moviment a través del temps. Com que cada punt pot col·lisionar amb qualsevol altre, si no s'utilitza cap optimització algorísmica (com una subdivisió de l'espai), el nombre d'interaccions que cal comprovar és de l'ordre de N^2 i resulta molt adequat per ser paral·lelitzat amb CUDA.

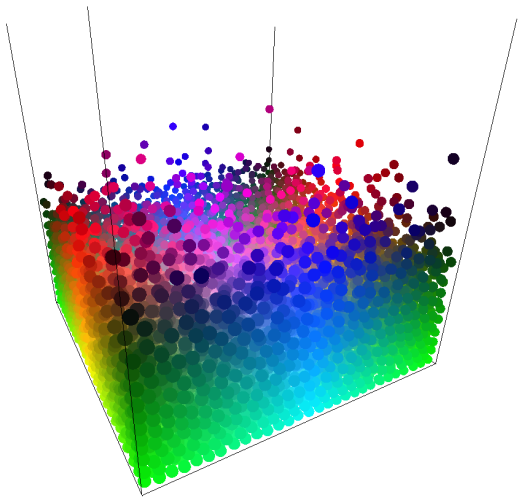

Figura 3: Visualització 3D amb OpenGL de l'execució remota utilitzant CUDA.

En el nostre cas, s'ha creat un espai tridimensional contingut en una caixa, i un conjunt de N pilotes que comencen a l'aire. El sistema està sotmès a una pseudo-gravetat, que incrementa la velocitat vertical de les pilotes a mesura que passa el temps. Les pilotes segueixen un model de col·lisió elàstica simplificada (s'ha buscat simplificar la part de càlcul físic del problema).

Idealment, aquest programa hauria de ser executat disposant de les targetes de forma interactiva i no remota. Llavors, el resultat es podria guardar directament als buffers de vèrtexs d'OpenGL i la visualització es podria realitzar en temps real. No obstant, com que el projecte s'ha realitzat de forma remota, s'ha obtat pel bolcat a fitxer (que pot ser potencialment molt gran, per exemple: 8192 punts en 1024 iteracions generen un fitxer de 223,06 MB).

3.0.2 Algorisme

La configuració del sistema a simular ve determinada per els paràmetres especificats en l'inici dels codis font:

- N: nombre de pilotes.
- `ITERATIONS`: nombre d'iteracions que el programa ha de simular.
- `DUMP`: determina si es dessitja que el programa imprimeixi en la sortida totes les posicions a través del temps per tal d'efectuar un visualitzat gràfic posteriorment.
- `DUMP_RATIO`: en cas de que `DUMP` sigui cert, estableix el ratio entre iteracions simulades i iteracions a visualitzar (en sistemes molt complexos pot ser recomanable efectuar més càlculs que els 30 o 60 FPS del visualitzat per mantenir la correctesa de la simulació).
- `POINT_RADIUS`: mida (radi) de cadascuna de les pilotes (s'assumeix mateixa massa).
- `WALLS`: activa o desactiva les parets i les col·lisions amb elles.

A més a més, es poden configurar les opcions de la simulació física amb els paràmetres `G` (gravetat), `BOUNCE_DECAY` (rebot amb pèrdua), `GLOBAL_DECAY` (fregament global), `TIME_SPEED` (increment del temps en cada iteració).

A partir de la generació aleatòria de la posició inicial de cada pilota, el sistema itera sobre els següents passos:

1. Aplicar la gravetat a totes les pilotes.
2. Calcular la interacció de cada pilota amb el món (parets i terra).
3. Calcular les interaccions entre totes les pilotes.
4. Fer avançar el temps en el sistema i calcular les següents posicions.

En tot moment es manté en memòria dos conjunts de N pilotes, el conjunt P i el conjunt Q. L'objectiu de mantenir dos conjunts és que en un es calculin les col·lisions de l'estat actual i en l'altre es guardin les noves velocitats. D'aquesta manera, els càlculs efectuats en un moment determinat no es veuen afectats per les col·lisions ja comprovades en la mateixa iteració i la simulació esdevé correcta.

3.0.3 Versió seqüencial

La versió seqüencial d'aquest programa realitza totes els passos de cada iteració utilitzant bucles simples. Les comprovacions entre les pilotes es realitzen amb el següent bucle per tal d'evitar comprovar i calcular dos cops una mateixa col·lisió:

```
1 for (i = 0; i < N; ++i)
2         for (j = i + 1; j < N; ++j)
3                 calcularInteraccio(pilota i,  pilota j)
```

El codi d'aquest algorisme es pot consultar en l'arxiu `physics.c`.

3.0.4 Versió paral·lela

La generació de les posicions inicials s'ha omès en la paral·lelització i s'ha deixat de forma seqüencial (el seu impacte en el temps d'execució es negligible).

Els diferents passos de cada iteració de la simulació s'ha resolt creant kernels que executen els diferents bucles de forma simultània. A continuació, ens centrarem en la part del codi que suposa pràcticament el 100% del temps d'execució en la versió seqüencial (ja que escala de forma quadràtica en funció del nombre de pilotes), és a dir, el càlcul de les col·lisions entre totes les pilotes.

La primera aproximació que s'ha provat per resoldre aquest problema ha estat crear un kernel que s'executi per a cada pilota "i" i comprovi les col·lisions amb les seves "N-i" següents:

```
1 kernel_interaccio_pilotes_versio_1(...) // N execucions
2         i = blockIdx.x * blockDim.x + threadIdx.x
3         for (j = i + 1; j < N; ++j)
4                 calcularInteraccio(pilota i,  pilota j)
```

No obstant, el resultat d'aquesta aproximació ha estat força pobre, obtenint només un speedUp igual a 2, per casos amb 2048 pilotes.

Una segona aproximació a aquest problema ha estat utilitzant el paral·lelisme dinàmic del que disposa la generació Kepler i posteriors de targetes gràfiques. En aquesta versió, s'utilitza un kernel igual que abans per a cada pilota "i" però, enlloc d'efectuar un bucle, crida a "N-i" kernels que comproven les col·lisions individualment.

```
1 kernel_individual(i,...) // N-i execucions
2         j = blockIdx.x * blockDim.x + threadIdx.x
3         calcularInteraccio(pilota i,  pilota j)
```

```
4
5  kernel_interaccio_pilotes_versio_2(...) // N execucions
6          i = blockIdx.x * blockDim.x + threadIdx.x
7          kernel_individual<<<...>>>(i,...)
```

Tot i que aquesta aproximació sí que ha presentat una millora de varis ordres de magnitud respecte la versió seqüencial (similar a la versió final, veure secció de resultats) degut a que, a diferència de la versió anterior, homogeneïtza el comportament dels threads dins dels warps, ha provocat un overhead innecessari en la creació dels kernels des de kernels que s'ha pogut suprimir amb una versió més senzilla i directa:

La versió final que s'ha utilitzat per resoldre el problema de les col·lisions entre N cossos ha estat pensar el bucle de la versió seqüencial sobre les "i" i les "j = i+1" com un sol bucle de (N * (N-1))/2 elements.

```
1  for (index = 0; index < (N*(N-1))/2; ++index)
2          calcular i, j a partir de index
3          calcularInteraccio(pilota i,  pilota j)
```

En cada cas d'aquest bucle, els índexs de les dues pilotes han de ser calculats a partir d'un index únic que va de 0 a (N * (N-1))/2 elements utilitzant una adaptació pròpia de la fórmula matemàtica de càlcul de fila i columna a partir d'índexs d'una matriu diagonal qualsevol:

```
1  kernel_interaccio_pilotes_versio_3(...) // (N*(N-1))/2 execucions
2          index = blockIdx.x * blockDim.x + threadIdx.x
3          j = 1 + floor(-0.5 + sqrt(0.25 + 2 * index));
4          triangularNumber = j * (j - 1) / 2;
5          i = index - triangularNumber;
6          calcularInteraccio(pilota i,  pilota j)
```

D'aquesta manera, es criden des de CPU de forma directa exactament tots els kernels necessaris per efectuar els càlculs de les col·lisions. Els resultats en comparació amb la versió seqüencial es poden consultar en l'apartat de resultats.

Durant el desenvolupament del codi paral·el CUDA, s'ha provat d'utilitzar STREAMS per tal de passar la informació a CPU i imprimir-la en un fitxer seguint l'esquema de la figura que es mostra. Això es pot fer de forma simultània amb els càlculs de les iteracions aprofitant el fet que només cal imprimir el resultat cada DUMP_RATIO iteracions i, que es pot utilitzar una còpia del resultat guardada en una altra posició de memòria de GPU.

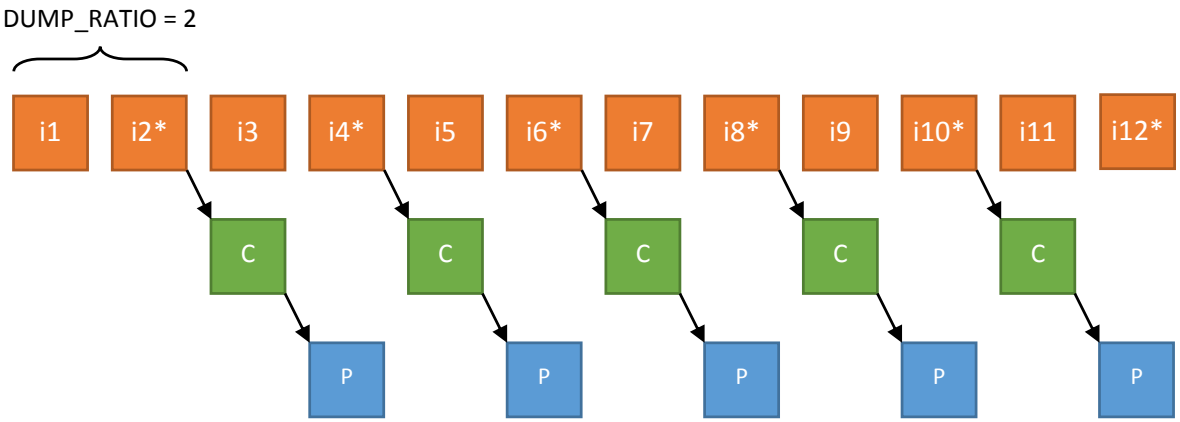

Figura 4: En taronja apareixen els diferents càlculs de cada iteració (realitzats amb diferents kernels). L'asterisc indica una còpia addicional del resultat en una altra posició de memòria GPU. En verd apareixen les còpies asíncrones des de l'estructura addicional cap a CPU. Finalment, en blau apareixen els printf realitzats en la CPU.

De la mateixa manera que ha passat amb el paral·lelisme dinàmic, s'ha trobat una altra solució amb millor temps d'execució i més senzilla. Desde la introducció de l'arquitectura Fermi (Compute capability 2.x), CUDA permet realitzar directament l'operació printf dins d'un kernel. Aquesta impressió es realitza en un buffer FIFO que per defecte té una mida d'1 MB. Ja que aquesta mida és massa petita, s'ha hagut de fer servir la crida següent per tal de fer créixer el buffer fins als aproximadament 100 MB (suficient per tots els exemples que s'han provat):

```
cudaDeviceSetLimit(cudaLimitPrintfFifoSize, 100*1024*1024);
```

Així doncs, només ha calgut realitzar una crida a N kernels:

```
kernel_print(...) {
        i = blockIdx.x * blockDim.x + threadIdx.x;
        printf(pilota i);
}
```

El codi d'aquest algorisme es pot consultar en l'arxiu `physics.cu`.

4 Resultats

A continuació es mostra el temps d'execució d'una única iteració de l'algorisme seqüencial i paral·lel en funció del nombre de punts (pilotes) que es tracten:

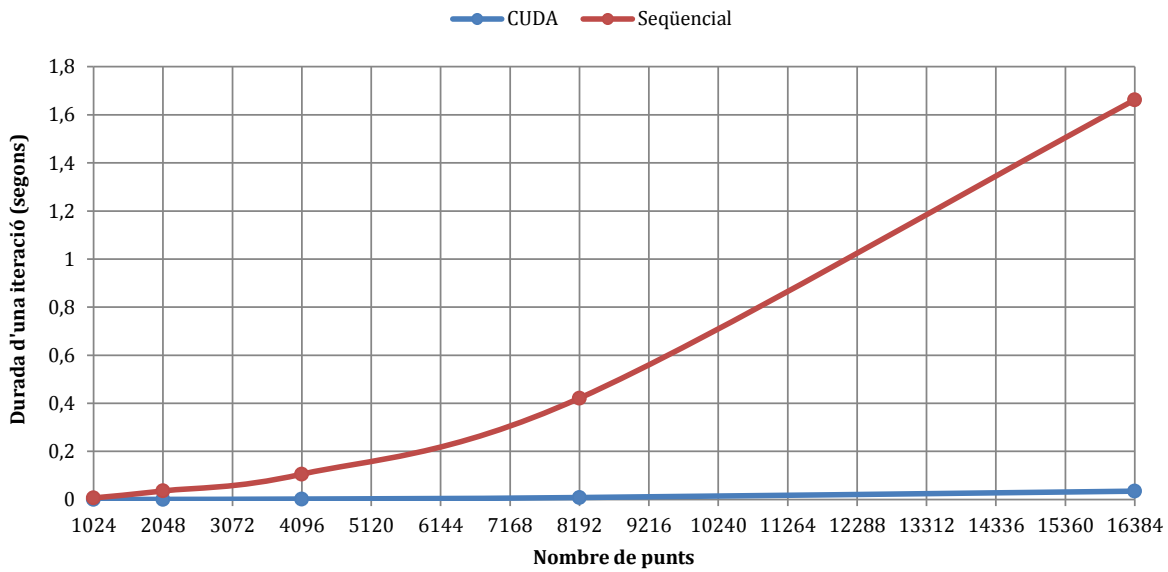

Figura 5: Temps d'execució d'una iteració en funció del nombre de punts.

Resulta evident que la versió en CUDA escala molt millor que la versió seqüencial. Per poder observar millor el comportament entre les dues versions, es mostra la mateixa gràfica amb una escala logarítmica base 10 en l'eix temporal (vertical):

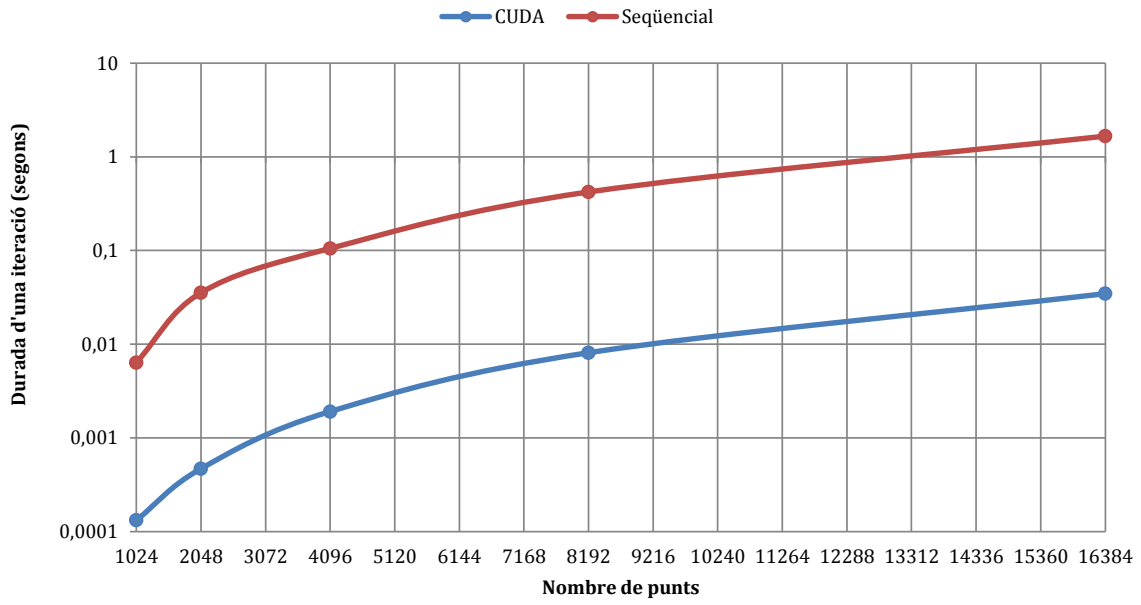

Figura 6: Temps d'execució d'una iteració en funció del nombre de punts fent servir una escala logarítmica en l'eix vertical.

Aquesta gràfica resulta particularment il·lustrativa en el sentit que mostra com les dues versions es comporten de forma similar però en diferents ordres de magntitud: en gairebé tot moment, es manté un speed-up aproximadament igual a 100.

La versió que utilitza CUDA fa servir 128 threads ja que s'obté un temps d'execució més baix si comparem amb altres configuracions, tal com mostra la figura següent:

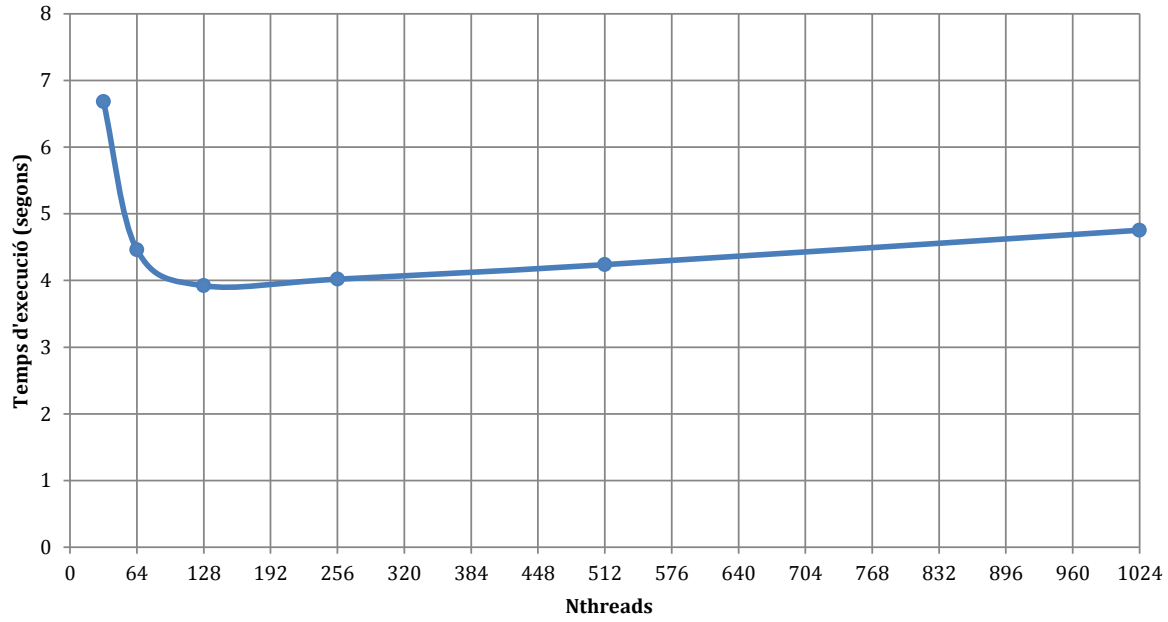

Figura 7: Temps d'execució total en funció del nombre de threads (4096 punts en 2000 iteracions)

En el següent enllaç es mostra el resultat d'una execució amb 8192 pilotes i 1024 iteracions:

`https://youtu.be/bl9UM-GlgHs`

5 Conclusions finals

Els dos problemes que s'han abordat han presentat dificultats ben diferents (veure apartats de resultats corresponents) i han demostrat que l'aplicació de certes tecnologies (paral·lelisme dinàmic, streams, multigpu, nombres aleatòris en GPUs), no és sempre sinònim d'una millora directa. Això és així perquè en alguns problemes resulten més difícils d'aplicar (per exemple, paral·lelisme dinàmic en l'algorisme genètic degut a l'existència de nombres aleatòris). Tot i així, resulten eines molt potents que, combinades amb bons algorismes que siguin escalables, ajuden a aconseguir un gran rendiment i a marcar encara més la diferència respecte les versions seqüencials dels codis.

L'elaboració de visualitzadors 3D dels resultats no ha estat, només, una feina addicional purament estètica, sinó que ha ajudat en gran mesura a entendre el funcionament dels codis i a poder solucionar errors que, de no poder haver disposat d'ells, haguessin estat quasi impossibles d'entendre a partir de milers de línies de resultats numèrics.

Finalment, comentar que l'aplicació de CUDA en els dos algorismes ha permès realitzar els càlculs dels resultats amb més celeritat de la que qualsevol processador pot proporcionar a data d'avui (i molt probablement durant força temps) gràcies a la seva arquitectura massivament paral·lela. Considerem doncs, que de forma general, els resultats obtinguts són més que satisfactoris. Entenem que davant de la limitació de la freqüència de rellotge a la que han arribat els fabricants de processadors (per consum elèctric i temperatura), la programació paral·lela està a l'alça i CUDA és una eina important a tenir en compte en el món de la supercomputació i de la computació en general.

6 Instruccions per compilar executar els codis

Els dos projectes (genetic i physics) contenen fitxers Makefile que permeten compilar de forma general, parts específiques, o executar en remot amb una sola crida. Alguns exemples:

```
> make show
```

Realitza una execució local seqüencial del programa i, un cop acabada, obre el visualitzador i mostra el resultat automàticament.

```
> make exec
```

Realitza una execució local seqüencial del programa i, durant la seva execució, mostra resultats del seu progrés i temps d'execució.

```
> make sub
```

Realitza una execució remota paral·lela del programa.

```
> make subseq
```

Realitza una execució remota seqüencial del programa.

Per compilar i executar el visualitzador 3D es necessita tenir instal·lat glut o freeglut a l'ordinador. En sistemes que disposin del gestor aptitude (OpenSuse, Ubuntu, Debian, etc.):

```
> sudo apt-get install freeglut3-dev
> make geneticViewer / make physicsViewer
> ./geneticViewer data_file / ./physicsViewer data_file
```

A més a més, és necessari disposar d'una targeta gràfica compatible amb OpenGL 2.1 (s'utilitzen vertex i fragment shaders propis).