Universitat Pompeu Fabra Barcelona

REPORT LAB 3: CUDA & OPENACC

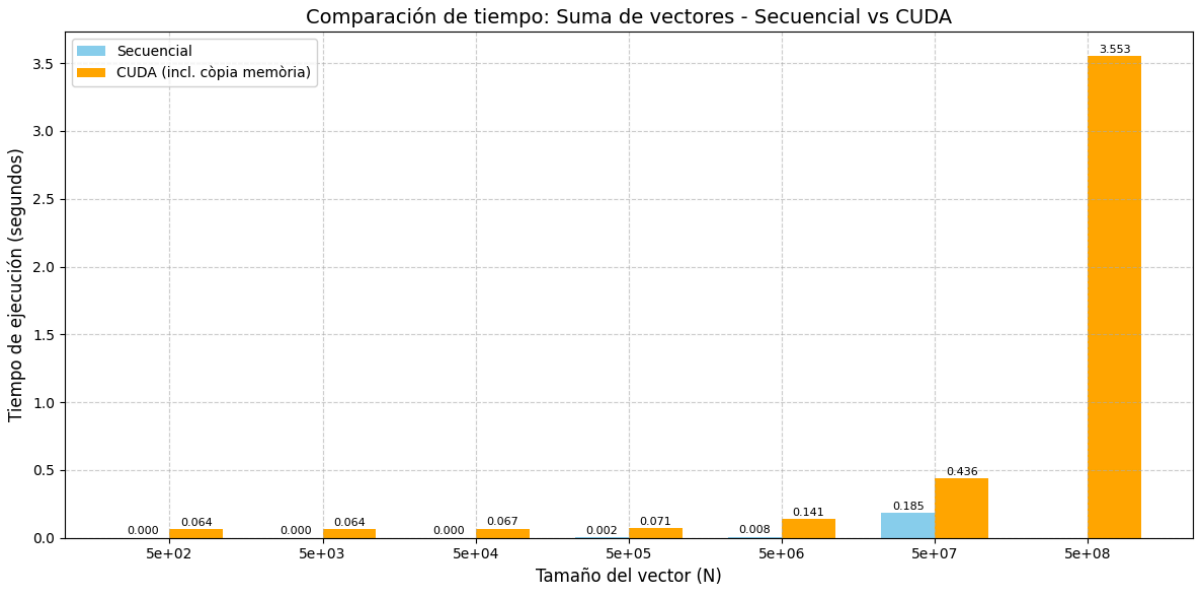
Introducció a la Programació Paral·lela i Distribuïda

Berta Miguel Ninou (U232844)
Núria Esquius Bau (U233645)

1. Vector Addition 🧮

1. Compare the sequential version with the CUDA version. At what vector size does the GPU kernel become faster than the sequential one? Create a bar plot showing the execution time of the sequential code, along with the kernel time of the CUDA code. (including memory transfers) for increasing vector sizes N “ 5 ˆ 10k, for k “ 2,...,8. Label the axes clearly.

Vector size	SEQÜENCIAL	CUDA
500	Vector size: 500 Elapsed time: 0.000000260 seconds Validation successful: C[i] = A[i] + B[i]	Time to copy data to device: 0.019456 ms Kernel execution time: 0.064288 ms Time to copy data back to host: 0.012672 ms Validation successful
5000	Vector size: 5000 Elapsed time: 0.000015420 seconds Validation successful: C[i] = A[i] + B[i]	Time to copy data to device: 0.023264 ms Kernel execution time: 0.064416 ms Time to copy data back to host: 0.020992 ms Validation successful
50000	Vector size: 50000 Elapsed time: 0.000225054 seconds Validation successful: C[i] = A[i] + B[i]	Time to copy data to device: 0.036416 ms Kernel execution time: 0.066848 ms Time to copy data back to host: 0.097376 ms Validation successful
500000	Vector size: 500000 Elapsed time: 0.001878499 seconds Validation successful: C[i] = A[i] + B[i]	Time to copy data to device: 0.158560 ms Kernel execution time: 0.070624 ms Time to copy data back to host: 0.755296 ms Validation successful
5000000	Vector size: 5000000 Elapsed time: 0.008406401 seconds Validation successful: C[i] = A[i] + B[i]	Time to copy data to device: 1.806528 ms Kernel execution time: 0.140576 ms Time to copy data back to host: 2.559712 ms Validation successful
50000000	Vector size: 50000000 Elapsed time: 0.185351211 seconds Validation successful: C[i] = A[i] + B[i]	Time to copy data to device: 18.282816 ms Kernel execution time: 0.436032 ms Time to copy data back to host: 29.189472 ms Validation successful
500000000	Vector size: 500000000 **ERROR (out of memory)** prterun noticed that process rank 0 with PID 1031555 on node amd11 exited on signal 9 (Killed). slurmstepd: error: Detected 1 oom_kill event in StepId=348152.batch. Some of the step tasks have been OOM Killed.	Time to copy data to device: 180.245255 ms Kernel execution time: 3.552864 ms Time to copy data back to host: 228.411545 ms Validation successful



Per a mides petites i mitjanes del vector (fins a 5×10^6), la versió seqüencial és significativament més ràpida que la versió CUDA. Aquest comportament es deu principalment elevat cost de transferència de memòria entre el host (CPU) i el dispositiu (GPU), que domina sobre el temps total d'execució en CUDA per aquests casos.

Tot i això, a mesura que la mida del vector augmenta, aquest cost de còpia esdevé relativament menys rellevant i el temps de càlcul paral·lel comença a guanyar pes. En el cas de $N = 5\times10^7$, encara que la versió CUDA no és més ràpida, la diferència de temps s'ha reduït considerablement, mostrant que la GPU comença a ser competitiva.

Finalment, per a $N = 5\times10^8$, la versió seqüencial falla per falta de memòria mentre que la versió CUDA és capaç de completar l'operació. Això mostra un avantatge important de la GPU en escenaris de gran volum de dades, ja que permet computar el càlcul on la CPU no pot.

2. Is the GPU always faster? For a fixed vector size N " 5^108, which version is faster in total execution time? Discuss whether offloading to the GPU always guarantees better performance. What are the main factors that determine whether a computation is suitable for GPU acceleration? Consider aspects like memory transfer cost, arithmetic intensity, etc.

La GPU no és sempre més ràpida. Tot i que les GPUs estan dissenyades per executar càlculs paral·lels de forma eficient, el rendiment global no només depèn de la potència de càlcul, sinó també de factors com el cost de transferència de memòria i la complexitat de l'operació.

Per un vector de mida $N = 5\times10^8$, la versió seqüencial en CPU no es pot executar a causa de la falta de memòria, mentre que la versió CUDA amb GPU sí que completa l'operació amb èxit. Així doncs, per aquest cas concret, la GPU és més efectiva perquè pot gestionar un volum de dades que la CPU no pot.

Hi ha diferents factors que influeixen en el temps d'execució dins la GPU. Per exemple, si ens fixem en el cost de transferència de memòria de host a device, podem veure que les dades s'han de copiar des de la memòria de la CPU a la memòria de la GPU, i això pot ser molt costós temporalment, especialment si el càlcul que es fa després és molt simple. A causa d'això, si el càlcul fa moltes operacions per cada byte transferit (alta intensitat aritmètica), és més probable que la GPU sigui avantatjosa. En canvi, si hi ha poca computació per molta memòria transferida, la CPU pot ser més ràpida.

Com a conclusió, la GPU pot oferir un rendiment superior en operacions paral·leles de gran volum, però no és sempre la millor opció. Cal avaluar si el càlcul és prou significatiu, si les dades es poden moure eficientment i si el problema és prou gran per justificar el cost d'usar la GPU.

3. CUDA vs OpenACC. Compare the performance of the OpenACC and CUDA implementations for N " 5 ˆ108. Which one is faster? What happens if you use pinned host memory in the CUDA version (HINT: cudaMallocHost(...))? Reflect on the Trade-off between programming effort and performance control when using CUDA versus OpenACC

Vector size	**OpenACC**	**CUDA**
500000000	`Vector size: 500000000` `Elapsed time: 1.768549616 seconds` `Validation successful: C[i] = A[i] + B[i]`	`Time to copy data to device: 180.245255 ms` `Kernel execution time: 3.552864 ms` `Time to copy data back to host: 228.411545 ms` `Validation successful`

Per un vector gran ($5 \cdot 10^8$), el codi amb OpenACC va més ràpidament que el CUDA. Això pot ser perquè OpenACC s'encarrega de moltes coses automàticament, com la gestió de memòria i la paral·lelització, sense que hagis de fer gaire manualment.

Però si a CUDA li poses memòria pinned (amb cudaMallocHost), pot reduir molt els temps de transferència de dades entre host i device, millorant significativament el rendiment total. Això pot fer que CUDA vagi més ràpidament que OpenACC, sobretot quan la transferència de dades pesa molt.

En resum, OpenACC és més fàcil i ràpid de programar, ideal si vols fer coses àgils i que funcionin bé sense complicar-te massa. CUDA, en canvi, et dona molt més control i pot arribar a ser més ràpid si optimitzes bé.

2. Matrix Multiplication ✖️

1. CPU vs. Naive GPU Kernel. Compare the performance of the sequential CPU implementation with the naive CUDA kernel. Run experiments with increasing matrix sizes. Is matrix-matrix multiplication a good candidate for GPU acceleration? Justify your answer based on the computation-to-memory ratio, parallelism, and observed results.

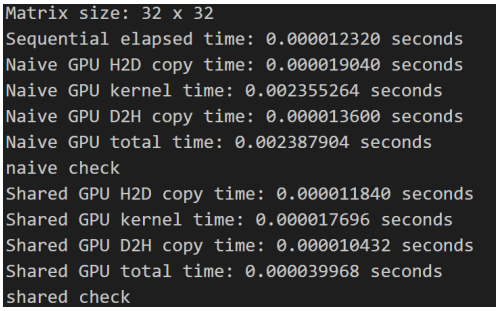
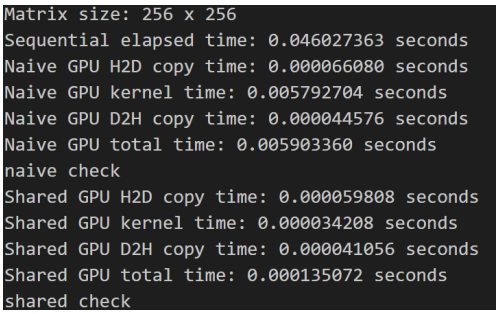
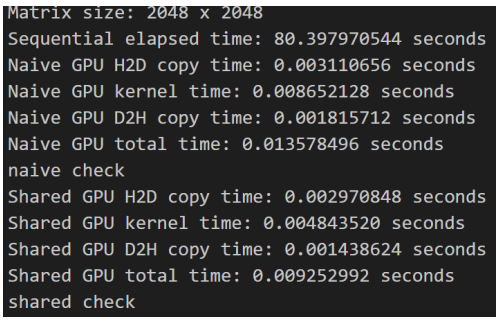
Mida matriu	Temps CPU	Temps naive GPU	Captura
32x32	**0.000012**	**0.00239**	Matrix size: 32 x 32 Sequential elapsed time: 0.000012320 seconds Naive GPU H2D copy time: 0.000019040 seconds Naive GPU kernel time: 0.002355264 seconds Naive GPU D2H copy time: 0.000013600 seconds Naive GPU total time: 0.002387904 seconds naive check
256x256	**0.046027**	**0.00590**	Matrix size: 256 x 256 Sequential elapsed time: 0.046027363 seconds Naive GPU H2D copy time: 0.000066080 seconds Naive GPU kernel time: 0.005792704 seconds Naive GPU D2H copy time: 0.000044576 seconds Naive GPU total time: 0.005903360 seconds naive check
2048x2048	**80.39797**	**0.01357**	Matrix size: 2048 x 2048 Sequential elapsed time: 80.397970544 seconds Naive GPU H2D copy time: 0.003110656 seconds Naive GPU kernel time: 0.008652128 seconds Naive GPU D2H copy time: 0.001815712 seconds Naive GPU total time: 0.013578496 seconds naive check

Per a matrius petites (com 32×32), la CPU és més eficient a causa del cost de temps de transferència de dades a la GPU. Per a mides mitjanes (256×256), la GPU comença a igualar en temps d'execució a la CPU. Per a mides grans (2048×2048), la GPU és notablement més ràpida que la CPU.

Per tant, la multiplicació de matrius és un bon exemple per l'acceleració en GPU, ja que, en general, la multiplicació de matrius implica moltes operacions de càlcul i pocs accessos de memòria, cosa que afavoreix a la GPU naive, perquè té molts nuclis de càlcul. A més, cada cel·la de la matriu resultant es pot calcular paral·lelament de forma independent, el que afavoreix als fils GPU. En últim lloc, l'ús de memòria compartida (com en matmul_shared_kernel) pot reduir l'accés a memòria global, millorant el rendiment.

En conclusió, la GPU naive supera la CPU en matrius grans.

2. Naive vs. Shared Memory Kernel. Evaluate the performance difference between the naive kernel and the shared memory version. How much impact does shared memory have on performance? Is the improvement consistent across different matrix sizes? Explain the reasons behind the observed behavior.

Mida matriu	Temps Shared Memory	Temps naive GPU	Captura
32x32	**0.000039**	**0.00239**	
256x256	**0.000135**	**0.00590**	
2048x2048	**0.009253**	**0.01357**	

Per a les matrius de 32x32 i 256x256, la memòria compartida és molt més eficient. Això passa perquè la memòria compartida permet reutilitzar dades dins d'un bloc de fils, minimitzant els accessos lents a la memòria global.

A mesura que les matrius són més grans, com en el cas de la matriu 2048×2048, tot i que és millor el rendiment de la memòria compartida, la millora ja no és tan clara. Això passa perquè, amb tanta informació per processar, la feina es reparteix molt bé entre tots els nuclis de la GPU, i fa que aquesta encara sigui eficient. A més, com que la memòria compartida i els registres s'utilitzen molt en aquest tipus d'optimització, pot passar que se saturin una mica i això limiti la millora.

En resum, l'ús de memòria compartida té un impacte molt positiu en el rendiment, sobretot en problemes de mida petita o mitjana. Tot i això, en problemes molt grans, el benefici continua estant però és menor.

3. Shared Memory Kernel vs. cuBLAS. Compare your best custom implementation to the highly optimized cuBLAS version. How close does your implementation get to cuBLAS performance? What are the main difficulties in achieving high performance with custom CUDA code? When is it worth writing custom kernels versus using GPU libraries?

Mida matriu	Temps Shared Memory	Temps cuBLAS	Captura
32x32	**0.0000399**	**0.02393**	Matrix size: 32 x 32 Sequential elapsed time: 0.000012320 seconds Naive GPU H2D copy time: 0.000019040 seconds Naive GPU kernel time: 0.002355264 seconds Naive GPU D2H copy time: 0.000013600 seconds Naive GPU total time: 0.002387904 seconds naive check Shared GPU H2D copy time: 0.000011840 seconds Shared GPU kernel time: 0.000017696 seconds Shared GPU D2H copy time: 0.000010432 seconds Shared GPU total time: 0.000039968 seconds shared check cuBLAS GPU H2D copy time: 0.000013408 seconds cuBLAS GPU kernel time: 0.023935903 seconds cuBLAS GPU D2H copy time: 0.000018048 seconds cuBLAS GPU total time: 0.023967359 seconds cublas check
2048x2048	**0.00925299**	**0.02871**	Matrix size: 2048 x 2048 Sequential elapsed time: 80.397970544 seconds Naive GPU H2D copy time: 0.003110656 seconds Naive GPU kernel time: 0.008652128 seconds Naive GPU D2H copy time: 0.001815712 seconds Naive GPU total time: 0.013578496 seconds naive check Shared GPU H2D copy time: 0.002970848 seconds Shared GPU kernel time: 0.004843520 seconds Shared GPU D2H copy time: 0.001438624 seconds Shared GPU total time: 0.009252992 seconds shared check cuBLAS GPU H2D copy time: 0.003012608 seconds cuBLAS GPU kernel time: 0.024220575 seconds cuBLAS GPU D2H copy time: 0.001476928 seconds cuBLAS GPU total time: 0.028710112 seconds cublas check

10000x10000	**0.66927**	**0.180988**	Matrix size: 10000 x 10000 Sequential and validation deactivated Naive GPU H2D copy time: 0.071801990 seconds Naive GPU kernel time: 0.694989920 seconds Naive GPU D2H copy time: 0.034647327 seconds Naive GPU total time: 0.801439285 seconds naive check Shared GPU H2D copy time: 0.072906271 seconds Shared GPU kernel time: 0.562472582 seconds Shared GPU D2H copy time: 0.033893920 seconds Shared GPU total time: 0.669272780 seconds shared check cuBLAS GPU H2D copy time: 0.072212160 seconds cuBLAS GPU kernel time: 0.074873216 seconds cuBLAS GPU D2H copy time: 0.033902943 seconds cuBLAS GPU total time: 0.180988312 seconds cublas check

Per a matrius petites (32x32) i mitjanes (2048x2048), la memòria compartida és molt més ràpida que cuBLAS, però per a matrius grans (10000x10000), la versió cuBLAS és més ràpida i eficient.

Crear codi CUDA altament eficient requereix una gestió acurada de la memòria, especialment aprofitant la memòria compartida per reduir l'ús de la memòria global, que és més lenta. També cal organitzar bé els threads i blocs per maximitzar l'ocupació de la GPU.

És recomanable utilitzar biblioteques GPU com cuBLAS quan es treballa amb operacions estàndard, ja que aquestes biblioteques estan molt optimitzades i permeten un desenvolupament més ràpid i fiable. En canvi, escriure kernels personalitzats és útil quan l'algoritme és molt específic o quan es necessita un control per optimitzar un cas concret o per experimentar.

3. Particles 🧑‍🎨

1. Sequential program. Implement the sequential version of the program and ensure that the validation test passes successfully. Identify the GPU acceleration opportunity inside the while loop—explain why and how the computation can be parallelized. Run the simulation with 1000 particles, saving the solution to disk.

Inicialitzem la posició inicial a 0.	`// TODO` `// Initial position` `particles[i].pos.x = 0.0;` `particles[i].pos.y = 0.0;` `particles[i].pos.z = 0.0;`

Assignem valors aleatoris de theta phi i v0 entre els mínims i màxims establerts. També establim la velocitat inicial de les partícules.

```
// TODO
// Generate random velocity and direction
theta = random_double(THETA_MIN, THETA_MAX);
phi = random_double(PHI_MIN, PHI_MAX);
v0 = random_double(V_MIN, V_MAX);
particles[i].vel.x = v0 * sin(theta) * cos(phi);
particles[i].vel.y = v0 * sin(theta) * sin(phi);
particles[i].vel.z = - v0 * cos(theta);
```

Calculem la nova posició de les partícules amb les següents fórmules:

$$x^{n+1} = x^n + v_x^n \Delta t$$
$$y^{n+1} = y^n + v_y^n \Delta t$$
$$z^{n+1} = z^n + v_z^n \Delta t$$

Actualitzem també la velocitat:

$$v_x^{n+1} = v_x^n - \frac{k|\vec{v}|}{m} v_x^n \Delta t$$
$$v_y^{n+1} = v_y^n - \frac{k|\vec{v}|}{m} v_y^n \Delta t$$
$$v_z^{n+1} = v_z^n - \frac{k|\vec{v}|}{m} v_z^n \Delta t$$

```
// Calculate new position and velocity
void integrateEuler(Particle *particles, const int N)
{
    for(int i=0;i<N;i++){
        particles[i].pos.x += particles[i].vel.x * DT;
        particles[i].pos.y += particles[i].vel.y * DT;
        particles[i].pos.z += particles[i].vel.z * DT;
                double modul
        double modul = sqrt(particles[i].vel.x * particles[i].vel.x + particles[i].vel.y
        particles[i].vel.x -= K * modul * particles[i].vel.x * DT / M;
        particles[i].vel.y -= K * modul * particles[i].vel.y * DT / M;
        particles[i].vel.z -= K * modul * particles[i].vel.z * DT / M;

    }
}
```

Fem la funció de còpia, per copiar totes les dades les partícules.

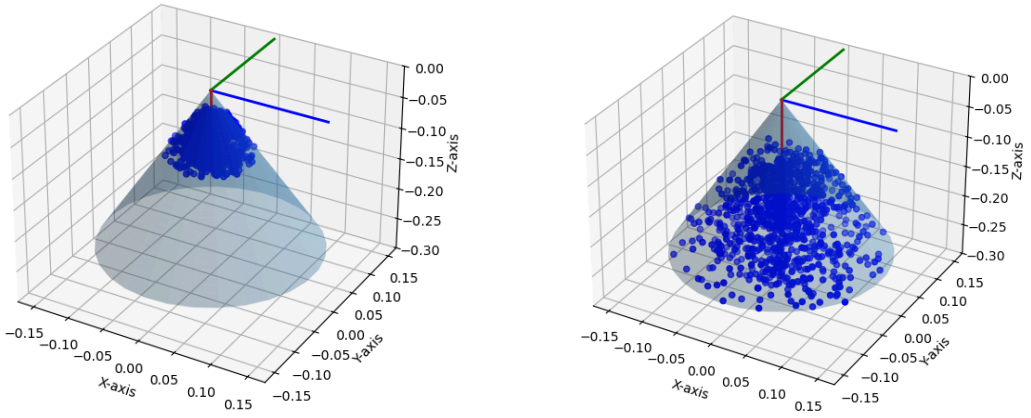
```
// Copy the state of the particles to a backup buffer.
void copyFrame(Particle *p_dst, Particle *p_src, const int N)
{
    for (int i = 0; i < N; ++i)
    {
        p_dst[i].pos.x = p_src[i].pos.x;
        p_dst[i].pos.y = p_src[i].pos.y;
        p_dst[i].pos.z = p_src[i].pos.z;

        p_dst[i].vel.x = p_src[i].vel.x;
        p_dst[i].vel.y = p_src[i].vel.y;
        p_dst[i].vel.z = p_src[i].vel.z;
    }

}
```

Podem veure que el test de validació, un cop executat el job, és correcte.

```
Particles: 1000
Iter: 100. Saving snapshot t = 1.000000e-04 in pFrame
Iter: 200. Saving snapshot t = 2.000000e-04 in pFrame
Iter: 300. Saving snapshot t = 3.000000e-04 in pFrame
Iter: 400. Saving snapshot t = 4.000000e-04 in pFrame
Iter: 500. Saving snapshot t = 5.000000e-04 in pFrame
Iter: 600. Saving snapshot t = 6.000000e-04 in pFrame
Iter: 700. Saving snapshot t = 7.000000e-04 in pFrame
Iter: 800. Saving snapshot t = 8.000000e-04 in pFrame
Iter: 900. Saving snapshot t = 9.000000e-04 in pFrame
Iter: 1000. Saving snapshot t = 1.000000e-03 in pFrame
Elapsed time: 0.044785 seconds
```

Podem veure que en el bucle while els càlculs de la funció integrateEuler es poden paral·lelitzar, ja que l'actualització de les posicions i les velocitats de les partícules són independents entre cada partícula. Així mateix, la part de copyFrame també es pot paral·lelitzar. Ho farem amb OpenACC paralel·litzant els loops de les funcions integrateEuler i copyFrame amb la directiva *#pragma acc parallel loop*.

Use the provided Python script plot.py to create an animated video of the droplets (note: it requires the .csv files generated in the out folder). Add your favorite snapshot from the animation to the assignment. Before running the script, you will need to create a Conda environment and install the required dependencies: *$ module load conda $ conda create -n $ conda activate $ conda install matplotlib opencv $ python plot.py*



2. OpenACC: Unified vs Programmer-Managed Memory. Use OpenACC directives to parallelize the computation and create two versions of the program:

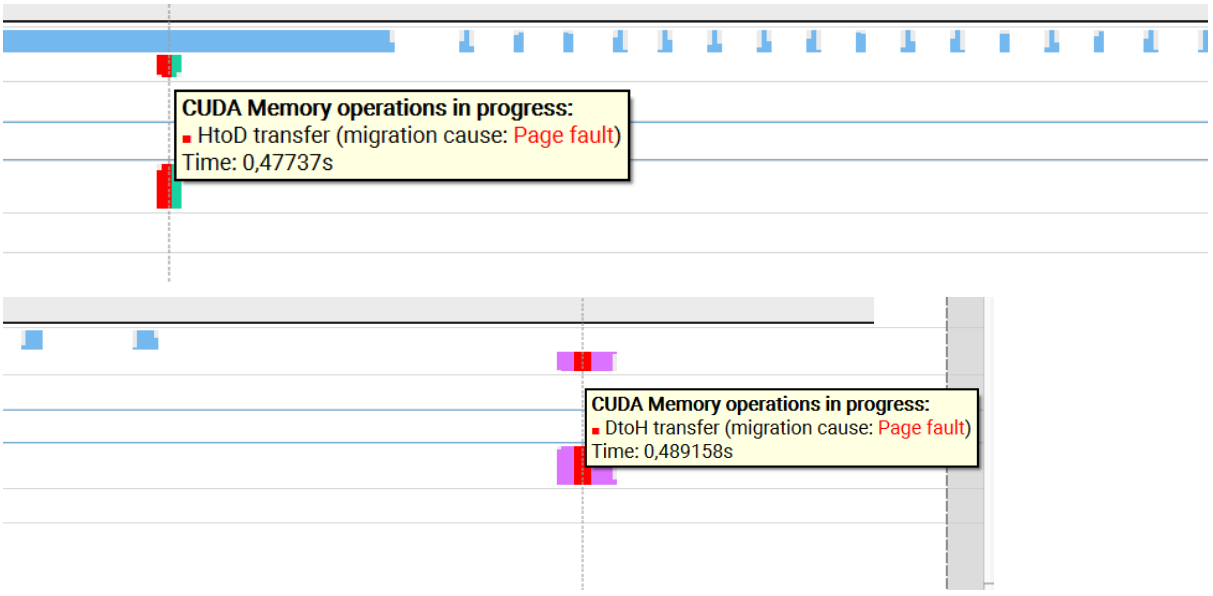
Utilitzem la directiva #pragma acc parallel loop per paral·litzar els càlculs.	`void integrateEuler(Particle *particles, const int N)` `{` `  #pragma acc parallel loop` `  for(int i=0;i<N;i++){` `  particles[i].pos.x += particles[i].vel.x * DT;` `  particles[i].pos.y += particles[i].vel.y * DT;` `  particles[i].pos.z += particles[i].vel.z * DT;` `  double modul = sqrt(particles[i].vel.x * particles[i].vel.x + particles[i].vel.y * p` `  particles[i].vel.x -= K * modul * particles[i].vel.x * DT / M;` `  particles[i].vel.y -= K * modul * particles[i].vel.y * DT / M;` `  particles[i].vel.z -= K * modul * particles[i].vel.z * DT / M;` `  }` `}`
Fem el mateix en la funció de còpia.	`void copyFrame(Particle *p_dst, Particle *p_src, const int N)` `{` `  #pragma acc parallel loop` `  for (int i = 0; i < N; ++i)` `  {` `  p_dst[i].pos.x = p_src[i].pos.x;` `  p_dst[i].pos.y = p_src[i].pos.y;` `  p_dst[i].pos.z = p_src[i].pos.z;` `  p_dst[i].vel.x = p_src[i].vel.x;` `  p_dst[i].vel.y = p_src[i].vel.y;` `  p_dst[i].vel.z = p_src[i].vel.z;` `  }`

In the first version, rely on the compiler to handle memory transfers using CUDA Unified Memory. Compile with the -gpu=managed flag, and use the visual profiler to analyze what happens during execution.

Posem gpu managed perquè el compilador gestioni les transferències de memòria.

```
CC = nvc
CFLAGS = -acc=gpu -gpu=managed -Minfo=all
OBJ = partis_oacc_prog_managed
all:
    $(CC) $(CFLAGS) $(OBJ).c -mp -o $(OBJ) -lm
clean:
    rm -f $(OBJ)
```

En el profiler podem veure que hi ha una transferència de memòria de HtoD al principi de tot. A continuació, es fan tots els càlculs en la GPU i finalment es torna a fer una transferència de memòria de DtoH.

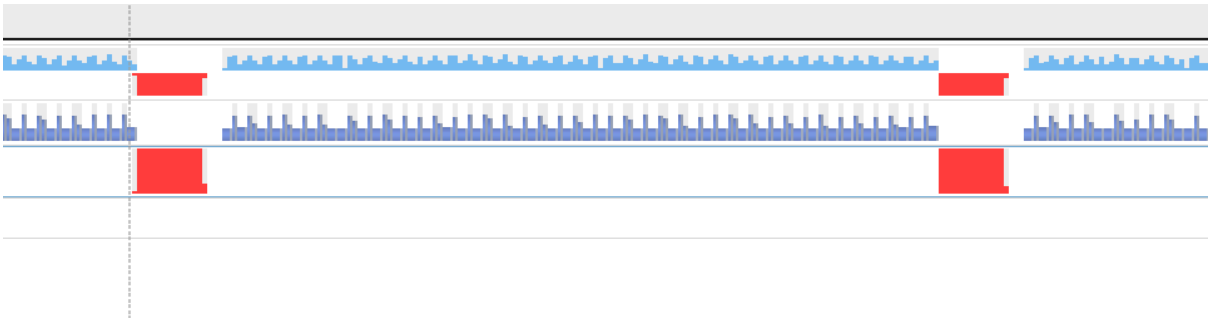


In the second version, manually manage memory transfers. Compile with the -gpu=cc90 flag, and use OpenACC data directives and clauses carefully to minimize data movement between host and device. Use the profiler to identify opportunities for optimization.

En les funcions integrateEuler i copyFrame necessitem l'array particles en la GPU, per tant, creem una regió data on copiem l'array particles de la CPU a la GPU i un cop acabem de la GPU a la CPU. A més, guardem memòria a la GPU per l'array pFrame.	`#pragma acc data copy(particles[0:N]) create(pFrame[0:N])` `{` `  while (t <= TOTAL_TIME)` `  {` `  // Time integration` `  integrateEuler(particles, N);` `  t += DT;` `  // Snapshot of the solution` `  if (iter % FREQ == 0)` `  {` `  curr_frame_time = t;` `  printf("Iter: %d. Saving snapshot t = %e in pFrame\n", iter,`

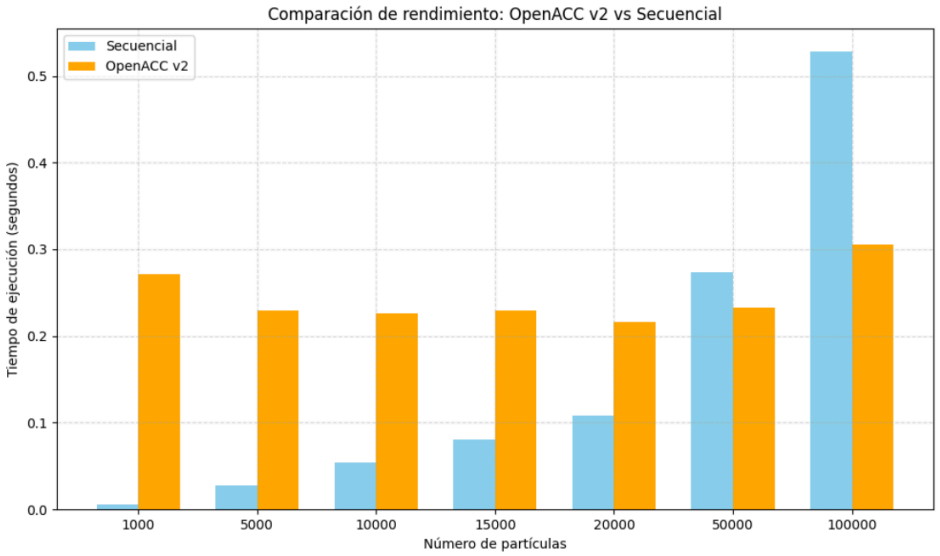
Altrament, necessitem fer update de l'array pFrame, ja que si write_flag està activat haurem d'escriure la posició i velocitat de pFrame en un file en CPU i necessitem que pFrame estigui actualitzat.	`#pragma acc update self(pFrame[0:N])` `if (write_flag)` `write_solution(pFrame, N, curr_frame_time, filename);` `}`
En els loop que hem paral·lelitzat afegim present perquè utilitzi l'array que es troba en LA GPU.	`#pragma acc parallel loop present(particles[0:N])` `for(int i=0;i<N;i++){` `particles[i].pos.x += particles[i].vel.x * DT;` `particles[i].pos.y += particles[i].vel.y * DT;` `particles[i].pos.z += particles[i].vel.z * DT;` `#pragma acc parallel loop present(p_dst[0:N], p_src[0:N])` `for (int i = 0; i < N; ++i)` `{` `p_dst[i].pos.x = p_src[i].pos.x;` `p_dst[i].pos.y = p_src[i].pos.y;` `p_dst[i].pos.z = p_src[i].pos.z;`

Podem veure en el profiler que la transferència de memòria del DtoH per actualitzar el pFrame ocupa molt de temps i fa que s'aturin els càlculs d'altres funcions com integrateEuler. Una bona opció per optimitzar el codi seria fer asíncronament la transferència de memòria i els següents càlculs de integrateEuler.



Compare the performance of the second OpenACC version against the sequential implementation. Create a bar plot showing execution time versus number of particles, as you did in the first question of the Vector Addition problem. For this performance study, disable output writing by setting the write flag to 0.

Podem veure que per un nombre de partícules menor a 50000 el temps en seqüencial és menor que en OpenACC. A partir de 50000 els temps són més semblants i cap a 100000 partícules el temps seqüencial és molt més gran que el de OpenACC. També podem veure que el temps de OpenACC per qualsevol nombre de partícules és molt semblant.



3. Asynchronous Operations. As you may have noticed, the copyFrame function is used to save snapshots of the transient solution at a fixed frequency—specifically, every 100 iterations. This creates an opportunity to overlap device-to-host memory transfers with ongoing computations during those 100 simulation steps. Use the async and wait clauses to enable concurrent execution of data transfers and computation. For this task, set the write flag to 0 to avoid writing output to disk. Use the visual profiler to verify and provide evidence of the achieved overlap

Tant en el integrateEuler i copyFrame els afegim en la mateixa cua “1”, ja que necessitem que els càlculs de integrateEuler hagin acabat un cop fem a còpia.	`void integrateEuler(Particle *particles, const int N)` `{` `  #pragma acc parallel loop present(particles[0:N]) async(1)` `  for(int i=0;i<N;i++){` `  particles[i].pos.x += particles[i].vel.x * DT;` `  particles[i].pos.y += particles[i].vel.y * DT;` `  particles[i].pos.z += particles[i].vel.z * DT;` `#pragma acc parallel loop present(p_dst[0:N], p_src[0:N]) async(1)` `for (int i = 0; i < N; ++i)` `{` `  p_dst[i].pos.x = p_src[i].pos.x;` `  p_dst[i].pos.y = p_src[i].pos.y;` `  p_dst[i].pos.z = p_src[i].pos.z;`
D’altra banda, la part d’update del DtoH es pot fer paral·lelament amb els següents càlculs d’integrateEuler, perquè no depenen entre ells. No obstant això, necessitem que copyFrame hagi acabat i, per tant, posem un wait(1) abans que la cua “2” pugui començar.	`  #pragma acc wait(1) async(2)` `  #pragma acc update self(pFrame[0:N]) async(2)` `  if (write_flag)` `  write_solution(pFrame, N, curr_frame_time, filename);` `}`

Finalment, posem un wait al final del bucle per assegurar-nos que tots els càlculs i les transferències s'hagin complert.	``` ++iter; } #pragma acc wait } ```

Podem veure en el profiler que la transferència de memòria de DtoH se superposa amb el següent càlcul de integrateEuler.