Escuela de Diseño, Ingeniería y Arquitectura
Programación Multinúcleo

Proyecto Final

Profesor

Octavio Navarro Hinojosa

Alumno

Ludovic Cyril Michel, A00819447

3 de diciembre de 2018

Proyecto: N-body simulation

Equipo: Ludovic Cyril Michel, A00819447

Problema que se resuelve: Conocer la evolución de un sistema físico simple, con base en las leyes de la mecánica clásica, siendo la única fuerza relevante la atracción gravitatoria.

Descripción: El programa simula, en tres dimensiones, un sistema de partículas físicas sometidas a la fuerza de gravedad.

Las entradas del programa son:

- el número n de partículas,
- la posición inicial p_i, la velocidad inicial v_i y la masa m_i de cada partícula i en el instante cero,
- el intervalo de tiempo λ entre dos instantes de la simulación.

La salida es una representación gráfica de la evolución del sistema en cada instante t.

Para generar esta representación, se calcula en cada instante t y para cada partícula i:

- la suma de fuerzas F_t, usando la fórmula:

$$F_t = G \sum_{i=1}^{n} \frac{(m_i m_j)}{|p_j - p_i|^2} (p_j - p_i) \;\; con\; i \neq j$$

  siendo G la constante de gravitación universal, y siendo p_i y p_j los vectores de posición en el instante t de las partículas i y j, respectivamente.

- la aceleración a_t, que se obtiene aplicando la tercera ley del movimiento de Newton:

$$F_t = m.a_t$$

- la velocidad v_t, calculada con:

$$v_t = v_{t-1} + a_t.\lambda$$

- la posición p_t, calculada con:

$$p_t = p_{t-1} + v_t.\lambda$$

Necesidad de paralelizar: La complejidad de este programa, en cada paso de la simulación, es $O(n^2)$. Por lo tanto, para realizar simulaciones interesantes sobre sistemas complejos, será necesario paralelizar el programa.

Usos posibles: Este tipo de simulación solo es realista cuando:

- los objetos bajo consideración tienen dimensiones muy pequeñas en comparación con las distancias que los separan,
- la única fuerza relevante es la gravitacional.

Por lo tanto, este sistema se adapta particularmente bien a estudios astronómicos.

Proyectos similares: Este sistema es un *physics engine*. Este tipo de software tiene numerosos usos, por ejemplo, en la industria de los videojuegos (Unity), para diseño industrial (AutoCAD Simulation), y para propósitos académicos.

Implementación: Para implementar este sistema, codificamos dos kernels, que corren sucesivamente en cada instante de la simulación:

- Un primer kernel para calcular el valor de aceleración de cada partícula con base en las fuerzas de gravitación ejercidas por las demás partículas del sistema:

```
dim3 num_blocks_acceleration(n_vertices);
dim3 num_threads_acceleration(n_vertices);
calculate_acceleration<<<num_blocks_acceleration, num_threads_acceleration>>>(
    devPtr, n_vertices);
```

  Este kernel corre con un grid unidimensional de n bloques con n threads. Cada bloque calcula la aceleración de una partícula. Dentro de cada bloque, cada thread calcula la interacción de la partícula en cuestión con otra partícula del sistema, y la suma a una variable compartida por medio de una operación atómica.

```cpp
__global__ void calculate_acceleration(Vertex *v, unsigned int n) {
  unsigned int i = blockIdx.x;
  unsigned int j = threadIdx.x;

  __shared__ float3 sub;

  if (j == 0) {
    sub.x = 0.0f;
    sub.y = 0.0f;
    sub.z = 0.0f;
  }

  __syncthreads();

  if (i < n && j < n && i != j) {
    float distance = sqrt(pow(v[i].position.x - v[j].position.x, 2) +
                          pow(v[i].position.y - v[j].position.y, 2) +
                          pow(v[i].position.z - v[j].position.z, 2));

    float magnitude = G_CONSTANT / pow(distance, 3);

    float3 vector;
    vector.x = magnitude * (v[i].position.x - v[j].position.x);
    vector.y = magnitude * (v[i].position.y - v[j].position.y);
    vector.z = magnitude * (v[i].position.z - v[j].position.z);

    atomicAdd(&(sub.x), -vector.x * v[j].mass);
    atomicAdd(&(sub.y), -vector.y * v[j].mass);
    atomicAdd(&(sub.z), -vector.z * v[j].mass);
  }

  __syncthreads();

  if (j == 0) {
    v[i].acceleration.x = sub.x;
    v[i].acceleration.y = sub.y;
    v[i].acceleration.z = sub.z;
  }
}
```

- Un segundo kernel para calcular, a partir de la aceleración de una partícula, su posición en el siguiente instante de la simulación.

```cpp
dim3 num_blocks_position(n_vertices);
dim3 num_threads_position(n_vertices);
calculate_position<<<num_blocks_position, num_threads_position>>>(
    devPtr, n_vertices, delta, max_distance);
```

Este kernel usa el intervalo λ para obtener primero la velocidad, y luego la posición de la partícula. La posición es normalizada para que todas las partículas del sistema quepan en la representación gráfica que se muestra.

```
__global__ void calculate_position(Vertex *v, unsigned int n, float delta,
                                   float max_distance) {
  unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

  if (i < n) {
    v[i].speed.x += v[i].acceleration.x * delta;
    v[i].speed.y += v[i].acceleration.y * delta;
    v[i].speed.z += v[i].acceleration.z * delta;

    v[i].position.x += v[i].speed.x * delta;
    v[i].position.y += v[i].speed.y * delta;
    v[i].position.z += v[i].speed.z * delta;

    v[i].gl_position.x = v[i].position.x / max_distance;
    v[i].gl_position.y = v[i].position.y / max_distance;
    v[i].gl_position.z = v[i].position.z / max_distance;

    v[i].acceleration.x = 0.0f;
    v[i].acceleration.y = 0.0f;
    v[i].acceleration.z = 0.0f;
  }
}
```

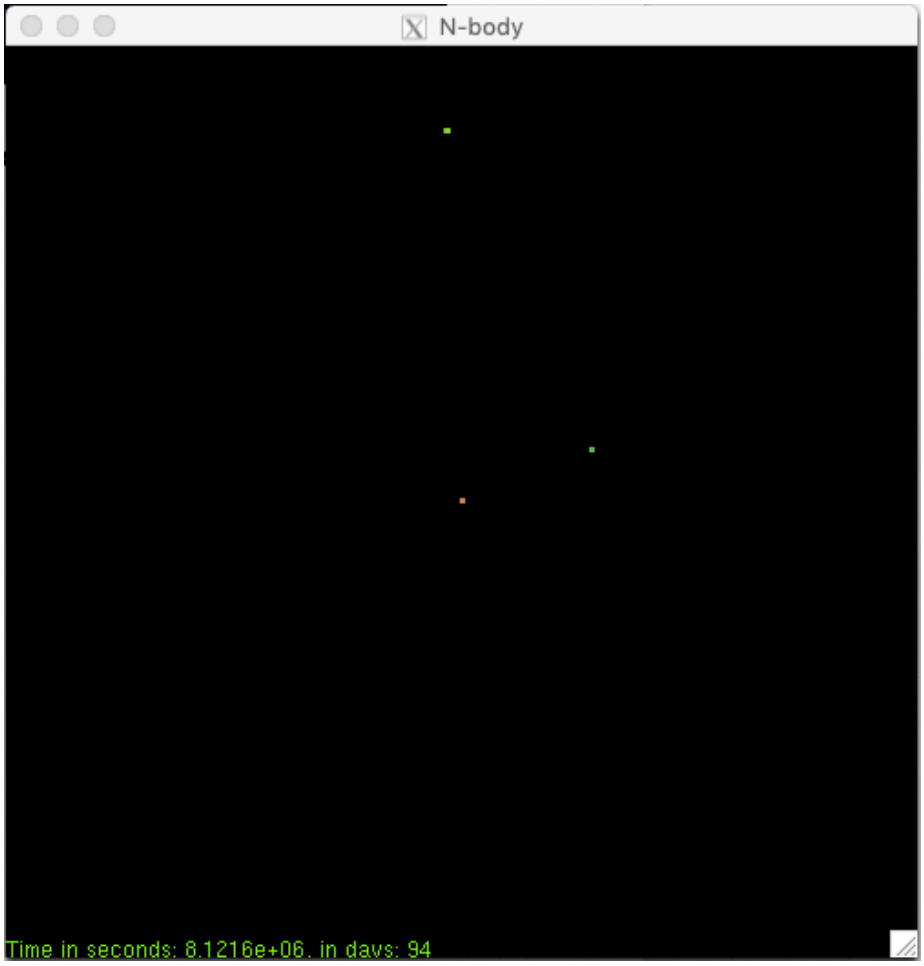
Uso: El programa se corre en la línea de comandos, usando la opción "—file=" para especificar un archivo que contiene las entradas.

La primera línea del archivo debe contener λ. La segunda línea debe contener n.

Las siguientes n líneas describen, cada una, una partícula, y contienen:

- La masa m de la partícula en kg,
- Los tres valores de posición de la partícula, en m,
- Los tres valores de velocidad de la partícula, en m/s.

Aprovechando la interoperabilidad entre CUDA y OpenGL, se despliega entonces la simulación dentro de una GUI:



Resultados: Comparamos los tiempos de corrida con una implementación en CPU, para un archivo de entrada con 500 partículas y después de 100 iteraciones de la simulación. Obtuvimos los siguientes resultados (los datos son promedios sobre 20 corridas):

Tiempo promedio en GPU (ms)	*Tiempo promedio en GPU (ms)*	*Speedup*
323.2	0.225	1434