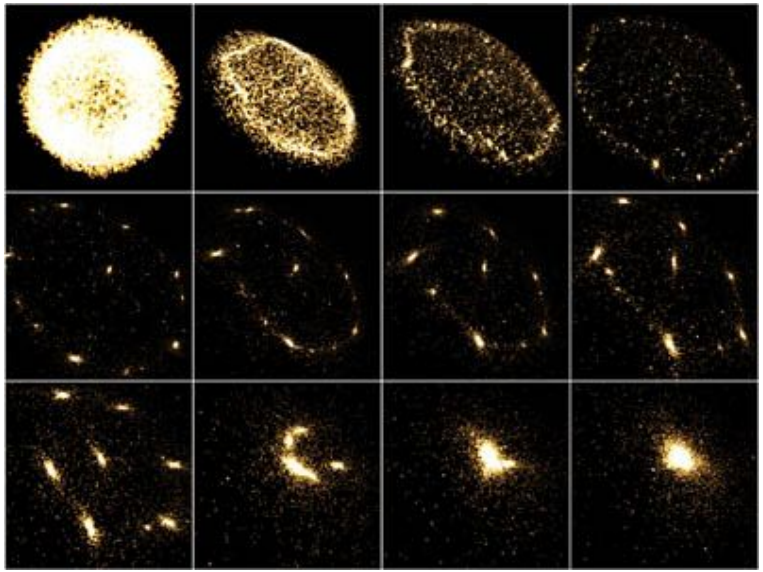
Simulación de N-Cuerpos

Implementación en la CPU y GPU

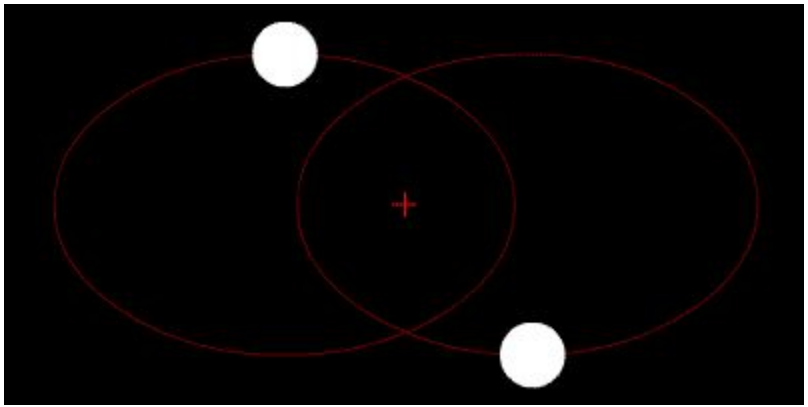
Motivación

- Interés en el área de gráficos por computadora.
- El problema de los N cuerpos presenta una gran oportunidad para simularse en un entorno gráfico, pudiendo aprender varias cosas.
- API de interés: OpenGL



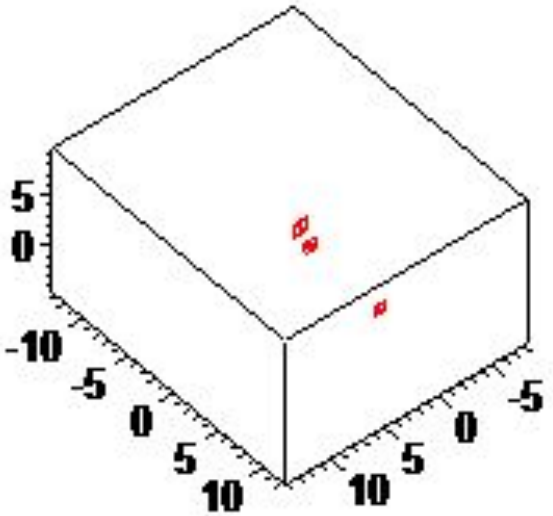
Problema

- Para un sistema de dos cuerpos, existe una **solución general** para describir el movimiento. Esto significa que con las posiciones y velocidades iniciales de los dos cuerpos, podemos saber **dónde** se encontrarán en c**ualquier momento** (posición en función del tiempo).



Sistemas dinámicos

- Para 3 cuerpos, no existe una solución general (y por lo tanto para N-Cuerpos). El sistema se vuelve **caótico**, y por lo tanto solo podemos aproximarlo utilizando métodos numéricos.



Modelado

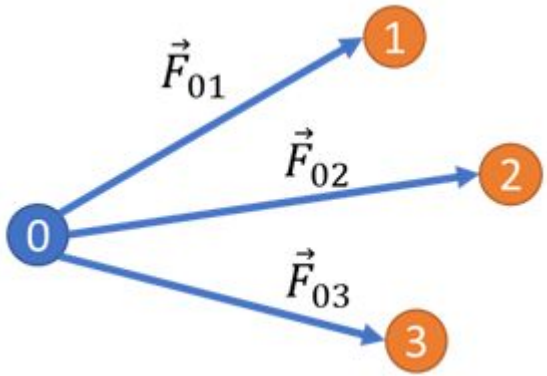
- Tenemos un conjunto de partículas P donde cada partícula tiene:
  - Posición: $\vec{x}$
  - Velocidad: $\vec{v}$
  - Masa: m

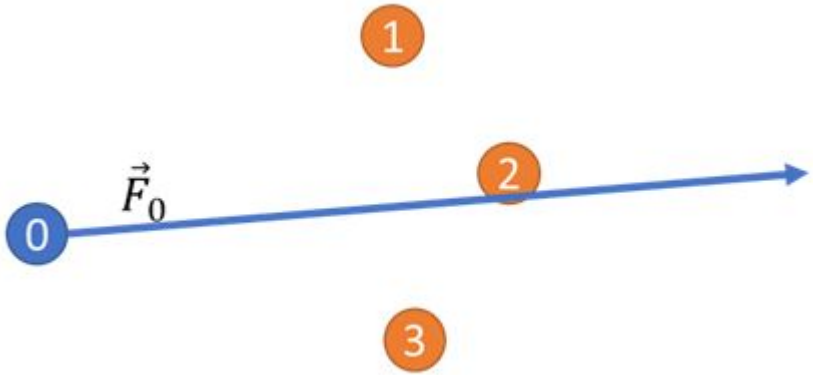
- Todas las partículas tienen condiciones iniciales (posición, velocidad y masa).

Formulas

- Ley de gravitación universal: $\vec{F_{12}} = G\frac{m_1 m_2}{d^2}\hat{u_{21}}$ $G = 6.674 \times 10^{-11}\left[N \cdot m^2 \cdot kg^{-2}\right]$
- Segunda ley de Newton: $\vec{F} = m\vec{a}$ $\vec{a} = \frac{\vec{F}}{m}$
- Velocidad: $\vec{a} = \frac{d\vec{v}}{dt} \approx \frac{\Delta\vec{v}}{\Delta t} = \frac{(\vec{v}_1 - \vec{v}_0)}{\Delta t}$ $\vec{v}_1 = \vec{v}_0 + \vec{a} \cdot \Delta t$
- Posición: $\vec{v} = \frac{d\vec{x}}{dt} \approx \frac{\Delta\vec{x}}{\Delta t} = \frac{(\vec{x}_1 - \vec{x}_0)}{\Delta t}$ $\vec{x}_1 = \vec{x}_0 + \vec{v} \cdot \Delta t$
- Tiempo: $\Delta t = cte$

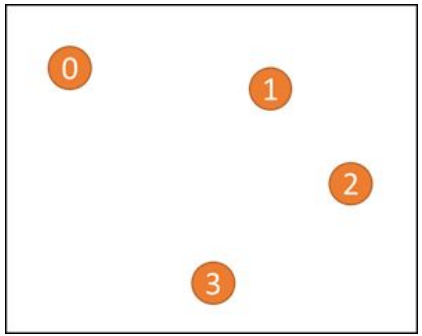
$$\vec{F}_i = \sum_{j=1,\, i \neq j}^{N} G \frac{m_i m_j}{d_{ij}^2} \hat{u}_{ji}$$

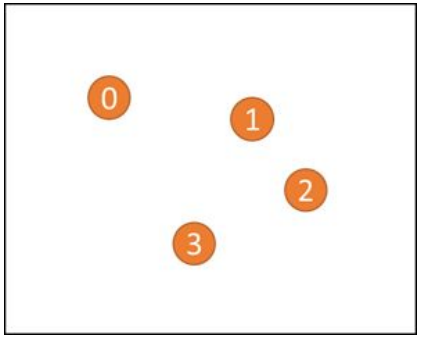


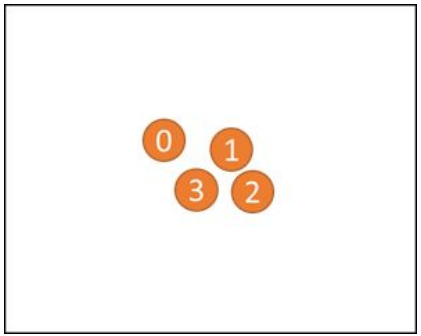


Algoritmo

1. Para cada partícula p_i del conjunto P actualizar su velocidad:
   a. Calcular el vector de fuerza resultante **F** (gravitación universal)
   b. Calcular el vector de aceleración **a**
   c. Calcular la nueva velocidad **v** de la partícula
2. Para cada partícula p_i del conjunto P actualizar su posición:
   a. Calcular la nueva posición **x** de la partícula
3. Ir al punto 1 el número de iteraciones establecidas por el usuario







Implementación

- Utilizaré C++ debido a que es un lenguaje de alto nivel que produce código muy rápido (y predominante en la industria de los gráficos por computadora).
- A diferencia de C, C++ tiene varias bibliotecas estandarizadas, donde sus estructuras y objetos tienen la misma velocidad que C. También implica que el código escrito será menor y más legible por cualquier usuario.

Conjunto de partículas

- Dos formas de modelarlas:

```
/// Struct of arrays (DOP)
struct Particle_Set
{
     std::vector<glm::vec3> positions;
     std::vector<glm::vec3> velocities;
};
```

```
/// Array of structs (OOP)
struct Particle
{
     glm::vec3 position;
     glm::vec3 velocity;
};

std::vector<Particle> set;
```

- Nota: la masa es igual para todas las partículas.

Código

1. Para cada partícula p_i del conjunto P actualizar su velocidad:

```
/// Gravitational constant
const float G = 6.6743e-11f;

/// Loop to calculate every particle new speed
for (unsigned int i = 0; i < numParticles; ++i)
{

	glm::vec3 force(0.0f, 0.0f, 0.0f);

	/// Loop to calculate a particle's force due to all the other particles
	for (unsigned int j = 0; j < numParticles; ++j)
	{

		/// If indexes i & j refer to the same body, do nothing
		if (i == j)
			continue;

		float distance2 = glm::distance2(positions[i], positions[j]);
		glm::vec3 direction = glm::normalize(positions[j] - positions[i]);

		force += G * mass * mass * direction / distance2;

	}

	// f=ma => a=f/m
	glm::vec3 acceleration = force / mass;

	// v=v0+a*dt
	velocities[i] += acceleration * deltaTime;

}
```

Código

1. Para cada partícula p_i del conjunto P actualizar su velocidad:

```
/// Gravitational constant
const float G = 6.6743e-11f;

/// Loop to calculate every particle new speed
for (unsigned int i = 0; i < numParticles; ++i)
{

	glm::vec3 force(0.0f, 0.0f, 0.0f);

	/// Loop to calculate a particle's force due to all the other particles
	for (unsigned int j = 0; j < numParticles; ++j)
	{

		/// If indexes i & j refer to the same body, do nothing
		if (i == j)
			continue;

		float distance2 = glm::distance2(positions[i], positions[j]);
		glm::vec3 direction = glm::normalize(positions[j] - positions[i]);

		force += G * mass * mass * direction / distance2;

	}

	// f=ma => a=f/m
	glm::vec3 acceleration = force / mass;

	// v=v0+a*dt
	velocities[i] += acceleration * deltaTime;

}
```

Código

1. Para cada partícula p_i del conjunto P actualizar su velocidad:

```
/// Gravitational constant
const float G = 6.6743e-11f;

/// Loop to calculate every particle new speed
for (unsigned int i = 0; i < numParticles; ++i)
{

	glm::vec3 force(0.0f, 0.0f, 0.0f);

	/// Loop to calculate a particle's force due to all the other particles
	for (unsigned int j = 0; j < numParticles; ++j)
	{

		/// If indexes i & j refer to the same body, do nothing
		if (i == j)
			continue;

		float distance2 = glm::distance2(positions[i], positions[j]);
		glm::vec3 direction = glm::normalize(positions[j] - positions[i]);

		force += G * mass * mass * direction / distance2;

	}

	// f=ma => a=f/m
	glm::vec3 acceleration = force / mass;

	// v=v0+a*dt
	velocities[i] += acceleration * deltaTime;

}
```

Código

1. Para cada partícula p_i del conjunto P actualizar su velocidad:

```
/// Gravitational constant
const float G = 6.6743e-11f;

/// Loop to calculate every particle new speed
for (unsigned int i = 0; i < numParticles; ++i)
{

	glm::vec3 force(0.0f, 0.0f, 0.0f);

	/// Loop to calculate a particle's force due to all the other particles
	for (unsigned int j = 0; j < numParticles; ++j)
	{

		/// If indexes i & j refer to the same body, do nothing
		if (i == j)
			continue;

		float distance2 = glm::distance2(positions[i], positions[j]);
		glm::vec3 direction = glm::normalize(positions[j] - positions[i]);

		force += G * mass * mass * direction / distance2;

	}

	// f=ma => a=f/m
	glm::vec3 acceleration = force / mass;

	// v=v0+a*dt
	velocities[i] += acceleration * deltaTime;

}
```

Código

2. Para cada partícula p_i del conjunto P actualizar su posición:

```
/// Update positions with new velocities
for (int i = 0; i < numParticles; ++i)
{

     positions[i] += velocities[i] * deltaTime;

}
```

Código

- Juntando ambas partes en una función:

```
auto n_body_iteration(Particle_Set &particles, const float deltaTime) -> void
{
	/// Gravitational constant
	const float G = 6.6743e-11f;

	/// Loop to calculate every particle new speed
	for (unsigned int i = 0; i < numParticles; ++i)
	{
		/// ...
	}

	/// Update positions with new velocities
	for (int i = 0; i < numParticles; ++i)
	{
		/// ...
	}

}
```

Código

3. Ir al punto 1 el número de iteraciones establecidas por el usuario

```
for (unsigned i = 0; i < numIterations; ++i)
     n_body_iteration(particle_set, timeStep);
```

Desempeño

```
./n_body_serial --particles=6400 --mass=1e9 --dt=1.0 --iterations=20
Particle count: 6400 particles
Particle mass: 1e+09 [kg]
Timestep: 1 [s]
Iterations: 20 steps

Compute elapsed time:
        3228.14 [ms] (3.22814 [s])
Average time per iteration:
        161407 [us]
```

```
./n_body_serial --particles=12800 --mass=1e9 --dt=1.0 --iterations=20
Particle count: 12800 particles
Particle mass: 1e+09 [kg]
Timestep: 1 [s]
Iterations: 20 steps

Compute elapsed time:
        12771.2 [ms] (12.7712 [s])
Average time per iteration:
        638559 [us]
```

```
./n_body_serial --particles=25600 --mass=1e9 --dt=1.0 --iterations=20
Particle count: 25600 particles
Particle mass: 1e+09 [kg]
Timestep: 1 [s]
Iterations: 20 steps

Compute elapsed time:
        51245.1 [ms] (51.2451 [s])
Average time per iteration:
        2.56226e+06 [us]
```

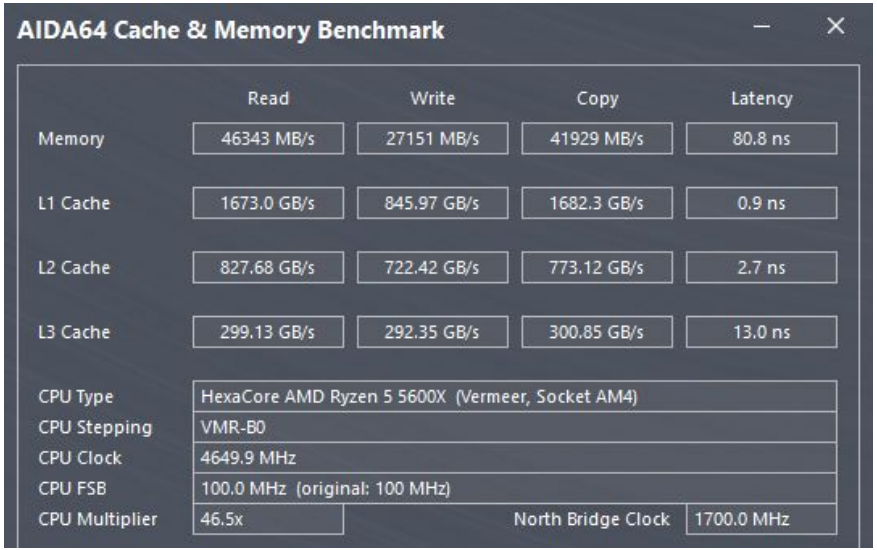
Desempeño

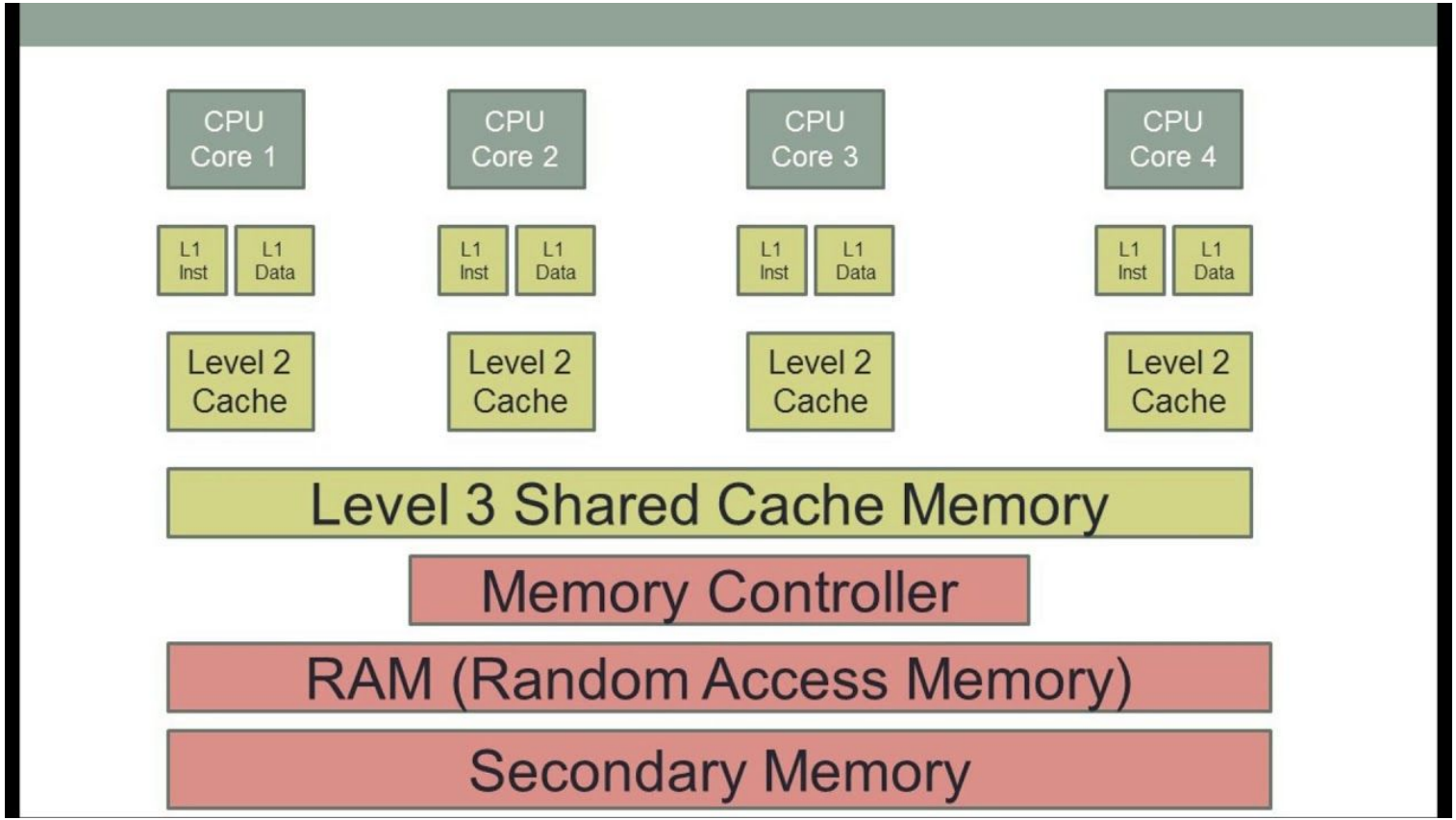
- En ambientes gráficos, es necesario que la animación sea fluida.
- Las pantallas normalmente tienen una tasa de actualización de 60 Hz, que es lo mismo que tener que actualizar la pantalla cada 16.66 ms.
- Si tardamos más de 16.66 ms en mostrar una imágen, la animación empieza a perder calidad rápidamente.

- Para 6,400 partículas, cada iteración tarda 161 ms, por lo que solo podríamos mostrar 6 cuadros por segundo (de 60 que tendríamos que mostrar).

Sobre DOP (Data Oriented Programming)

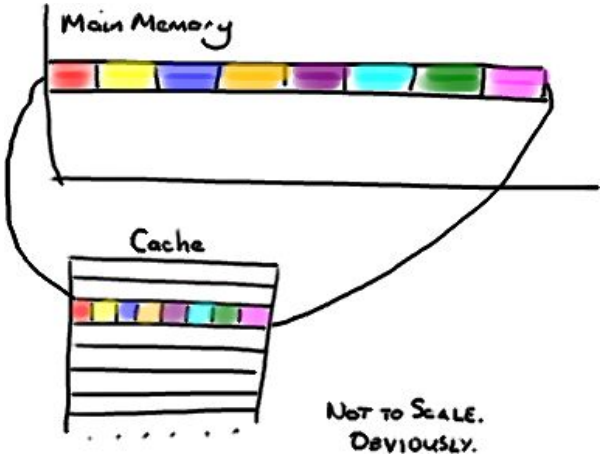
- Uno de los mayores cuellos de botella a la hora de correr programas es la velocidad a la que se leen los datos.
- Para la rapidez de la CPU, la memoria RAM es algo lenta.




CPU Core 1
CPU Core 2
CPU Core 3
CPU Core 4
L1 Inst
L1 Data
L1 Inst
L1 Data
L1 Inst
L1 Data
L1 Inst
L1 Data
Level 2 Cache
Level 2 Cache
Level 2 Cache
Level 2 Cache
Level 3 Shared Cache Memory
Memory Controller
RAM (Random Access Memory)
Secondary Memory

Memoria Caché

- La lectura y escritura de datos en la memoria caché es mucho más rápida que en la memoria RAM.
- No podemos guardar explícitamente datos en la caché.
- Su tamaño es muy limitado.



Cache	
Cache L1:	64K (per core)
Cache L2:	512K (per core)
Cache L3:	32MB

Eficiencia

- Un programa puede correr más rápido si el código es amigable con el caché (*cache friendly*).
- Para que esto suceda, debemos “comprimir” los datos para leerlos más rápidamente.

Arreglo de estructuras (OOP)

Estructura de arreglos (DOP)

- Para calcular la velocidad de una partícula p_i necesitamos:
  - La velocidad de p_i
  - Todas las posiciones del conjunto P
- El caché se divide en líneas típicamente de 64 bytes
- Las partículas representadas por OOP llenan muy rápido las líneas de caché con información que no necesitamos

- Las representadas por DOP aprovechan las líneas con información que si vamos a requerir

Cache Miss y Cache Hit

- Si el dato que queremos leer está en el caché, solo tenemos que recuperar esa línea (*Cache Hit*)
- Si el dato **no** se encuentra en el caché, es necesario traerlo desde memoria principal (o secundaria) y esperar a que llegue (*Cache Miss*)
- Como se esperaría, entre más fallemos en leer memoria del caché, tenemos que hacer varias peticiones a la memoria principal para traer los datos al caché. Por eso la clara ventaja de la DOP.

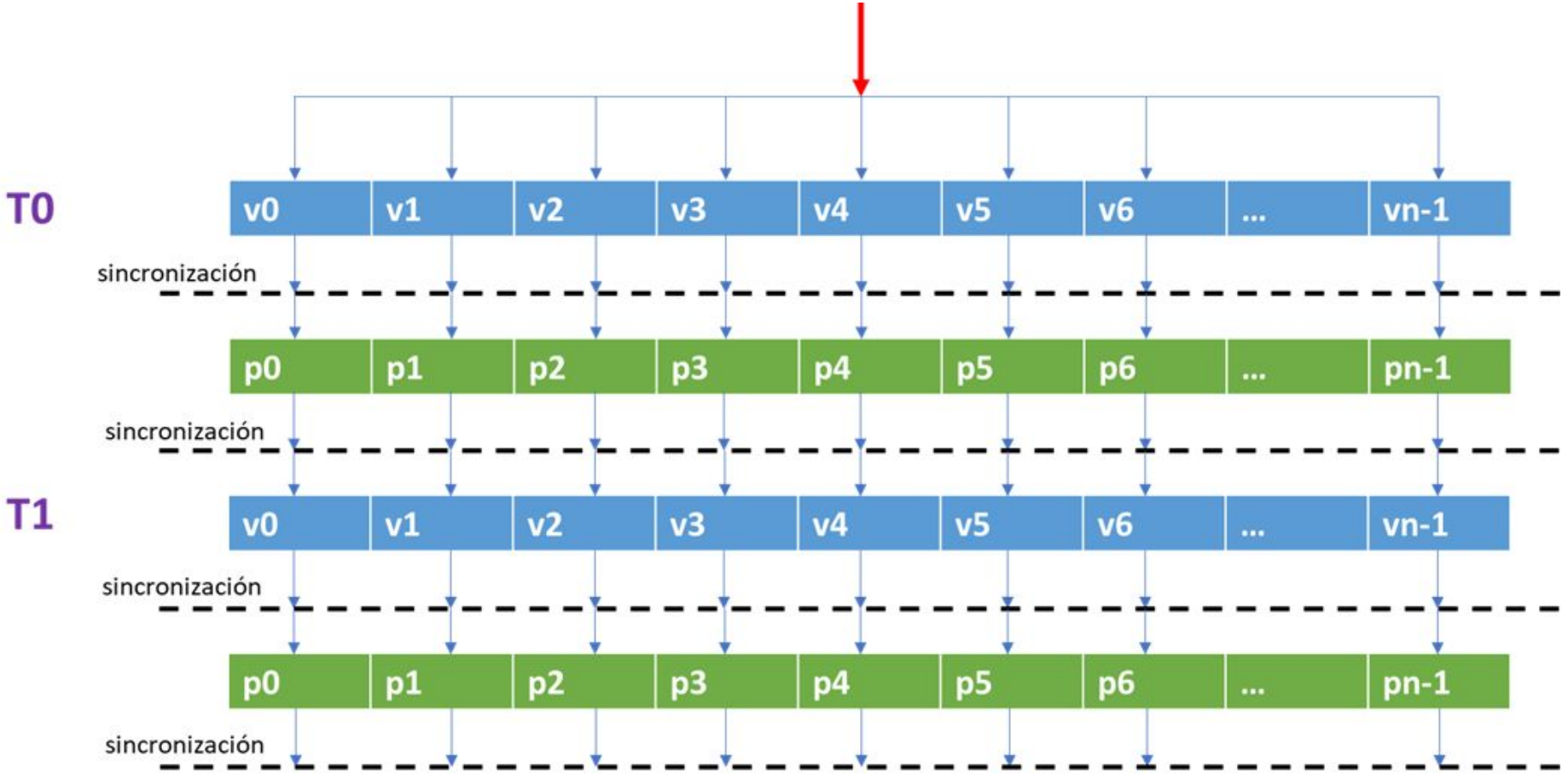
Paralelización (CPU)

Repasando el algoritmo

1. Para cada partícula p_i del conjunto P actualizar su velocidad:
   a. Calcular el vector de fuerza resultante **F** (gravitación universal)
   b. Calcular el vector de aceleración **a**
   c. Calcular la nueva velocidad **v** de la partícula
2. Para cada partícula p_i del conjunto P actualizar su posición:
   a. Calcular la nueva posición **x** de la partícula
3. Ir al punto 1 el número de iteraciones establecidas por el usuario

Oportunidades de paralelización

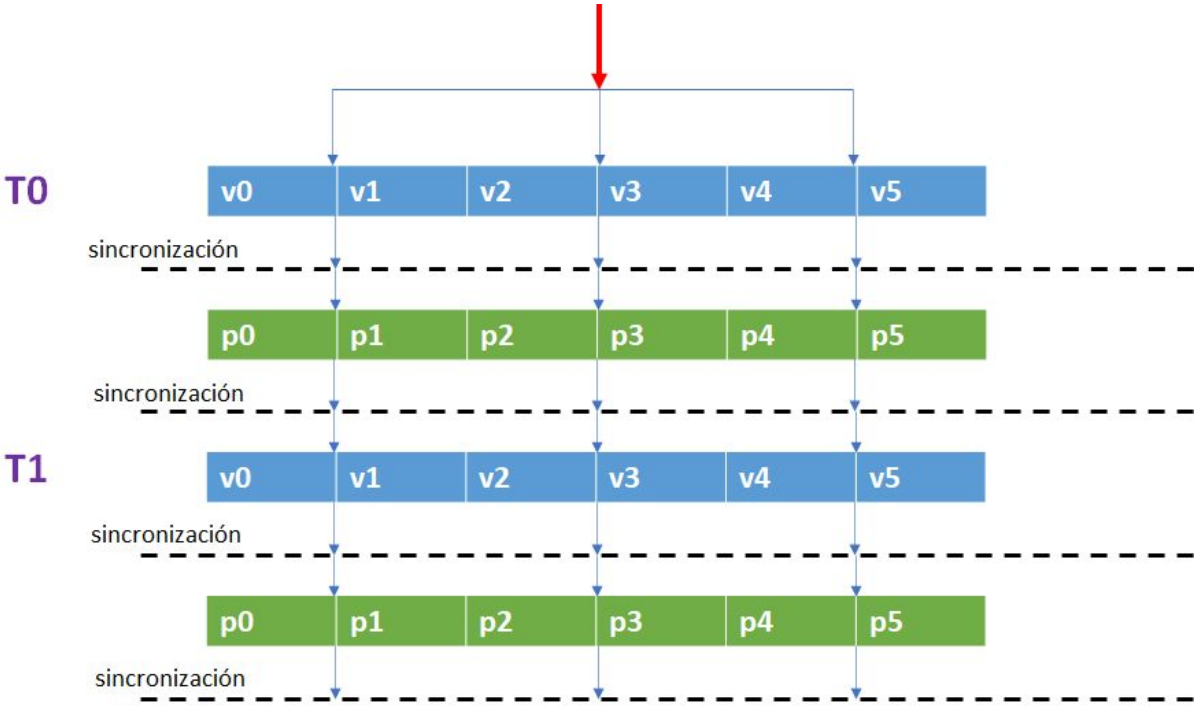
1. El cálculo de la velocidad de una partícula no afecta el cálculo de la velocidad de otra partícula, por lo que **cada velocidad puede ser calculada en paralelo**.
2. De igual forma, el cálculo de la posición de una partícula no afecta el cálculo de la posición de otra partícula, por lo que **cada posición puede ser calculada en paralelo**.
3. Para actualizar una posición, **debemos esperar a que todas las partículas hayan calculado su nueva velocidad**, o podrían estar leyendo una posición adelantada al tiempo.


T0
v0
v1
v2
v3
v4
v5
v6
...
vn-1
sincronización
p0
p1
p2
p3
p4
p5
p6
...
pn-1
sincronización
T1
v0
v1
v2
v3
v4
v5
v6
...
vn-1
sincronización
p0
p1
p2
p3
p4
p5
p6
...
pn-1
sincronización

- Idealmente habría un hilo por cada partícula y todos los hilos se ejecutarían en paralelo
- En una computadora de N núcleos físicos, solo N hilos pueden estar trabajando al mismo tiempo.
- Aun así son útiles los núcleos lógicos, así que para poder aprovechar al máximo nuestro procesador, necesitamos dividir los trabajos en M núcleos lógicos.

Lotes uniformes

- Si tenemos N hilos y M partículas, a cada hilo le corresponderían N/M partículas.



Código

- Primero reducimos el cálculo a la velocidad y posición de una sola partícula

```
auto n_body_velocity_calc(unsigned index, Particle_Set &particles, const float deltaTime) -> void
{
	/// ...
	velocities[index] += acceleration * deltaTime;
}
auto n_body_position_calc(unsigned index, Particle_Set &particles, const float deltaTime) -> void
{
	/// ...
	positions[index] += velocities[index] * deltaTime;
}
```

Código

- Luego escribimos la función de un hilo, el cual se encargará de actualizar N/M partículas

```
auto n_body_thread(unsigned begin, unsigned end, Particle_Set &particles, const Computation_Info &info,
                   boost::barrier &sync_point) -> void
{

      for (unsigned i = 0; i < info.numIterations; ++i)
      {
            /// Dispatch the velocity calculation
            for (unsigned j = begin; j < end; ++j)
            {
                  n_body_velocity_calc(j, particles, info.timeStep);
            }

            /// Sync threads for memory coherence
            sync_point.wait();

            /// Dispatch the position calculation
            for (unsigned j = begin; j < end; ++j)
            {
                  n_body_position_calc(j, particles, info.timeStep);
            }

            /// Sync again
            sync_point.wait();
      }

      return;

}
```

Código

- Luego escribimos la función de un hilo, el cual se encargará de actualizar N/M partículas

```
auto n_body_thread(unsigned begin, unsigned end, Particle_Set &particles, const Computation_Info &info,
                   boost::barrier &sync_point) -> void
{
      for (unsigned i = 0; i < info.numIterations; ++i)
      {
            /// Dispatch the velocity calculation
            for (unsigned j = begin; j < end; ++j)
            {
                  n_body_velocity_calc(j, particles, info.timeStep);
            }

            /// Sync threads for memory coherence
            sync_point.wait();

            /// Dispatch the position calculation
            for (unsigned j = begin; j < end; ++j)
            {
                  n_body_position_calc(j, particles, info.timeStep);
            }

            /// Sync again
            sync_point.wait();
      }

      return;
}
```

Código

- Luego escribimos la función de un hilo, el cual se encargará de actualizar N/M partículas

```
auto n_body_thread(unsigned begin, unsigned end, Particle_Set &particles, const Computation_Info &info,
                   boost::barrier &sync_point) -> void
{

      for (unsigned i = 0; i < info.numIterations; ++i)
      {
            /// Dispatch the velocity calculation
            for (unsigned j = begin; j < end; ++j)
            {
                  n_body_velocity_calc(j, particles, info.timeStep);
            }

            /// Sync threads for memory coherence
            sync_point.wait();

            /// Dispatch the position calculation
            for (unsigned j = begin; j < end; ++j)
            {
                  n_body_position_calc(j, particles, info.timeStep);
            }

            /// Sync again
            sync_point.wait();
      }

      return;

}
```

Código

- Luego escribimos la función de un hilo, el cual se encargará de actualizar N/M partículas

```
auto n_body_thread(unsigned begin, unsigned end, Particle_Set &particles, const Computation_Info &info,
                   boost::barrier &sync_point) -> void
{

      for (unsigned i = 0; i < info.numIterations; ++i)
      {
            /// Dispatch the velocity calculation
            for (unsigned j = begin; j < end; ++j)
            {
                  n_body_velocity_calc(j, particles, info.timeStep);
            }

            /// Sync threads for memory coherence
            sync_point.wait();

            /// Dispatch the position calculation
            for (unsigned j = begin; j < end; ++j)
            {
                  n_body_position_calc(j, particles, info.timeStep);
            }

            /// Sync again
            sync_point.wait();
      }

      return;

}
```

Código

- Por último, otra función se encargará de asignar los hilos, cada uno con un rango de partículas con el cual va a trabajar.

```
auto n_body_computation_dispatcher(Particle_Set &particles, const Computation_Info &info) -> void
{

	/// Thread dispatch for updating particles speed
	unsigned step_index = numParticles / numThreads;
	unsigned remainder = numParticles % numThreads;

	/// Barrier for syncing all threads
	boost::barrier sync_point{numThreads};

	std::vector<std::jthread> threads;

	for (unsigned curr_index = 0; curr_index < particles.numParticles; curr_index += step_index)
	{
		if (remainder != 0)
		{
			threads.emplace_back(n_body_thread, curr_index, curr_index + step_index + 1,
					std::ref(particles), std::cref(info), std::ref(sync_point));
			++curr_index;
			--remainder;
		}
		else
			threads.emplace_back(n_body_thread, curr_index, curr_index + step_index, std::ref(particles),
					std::cref(info), std::ref(sync_point));
	}

}
```

Código

- Por último, otra función se encargará de asignar los hilos, cada uno con un rango de partículas con el cual va a trabajar.

```
auto n_body_computation_dispatcher(Particle_Set &particles, const Computation_Info &info) -> void
{

	/// Thread dispatch for updating particles speed
	unsigned step_index = numParticles / numThreads;
	unsigned remainder = numParticles % numThreads;

	/// Barrier for syncing all threads
	boost::barrier sync_point{numThreads};

	std::vector<std::jthread> threads;

	for (unsigned curr_index = 0; curr_index < particles.numParticles; curr_index += step_index)
	{
		if (remainder != 0)
		{
			threads.emplace_back(n_body_thread, curr_index, curr_index + step_index + 1,
					std::ref(particles), std::cref(info), std::ref(sync_point));
			++curr_index;
			--remainder;
		}
		else
			threads.emplace_back(n_body_thread, curr_index, curr_index + step_index, std::ref(particles),
					std::cref(info), std::ref(sync_point));
	}

}
```

Código

- Por último, otra función se encargará de asignar los hilos, cada uno con un rango de partículas con el cual va a trabajar.

```
auto n_body_computation_dispatcher(Particle_Set &particles, const Computation_Info &info) -> void
{

	/// Thread dispatch for updating particles speed
	unsigned step_index = numParticles / numThreads;
	unsigned remainder = numParticles % numThreads;

	/// Barrier for syncing all threads
	boost::barrier sync_point{numThreads};

	std::vector<std::jthread> threads;

	for (unsigned curr_index = 0; curr_index < particles.numParticles; curr_index += step_index)
	{
		if (remainder != 0)
		{
			threads.emplace_back(n_body_thread, curr_index, curr_index + step_index + 1,
					std::ref(particles), std::cref(info), std::ref(sync_point));
			++curr_index;
			--remainder;
		}
		else
			threads.emplace_back(n_body_thread, curr_index, curr_index + step_index, std::ref(particles),
					std::cref(info), std::ref(sync_point));
	}

}
```

Código

- Por último, otra función se encargará de asignar los hilos, cada uno con un rango de partículas con el cual va a trabajar.

```
auto n_body_computation_dispatcher(Particle_Set &particles, const Computation_Info &info) -> void
{

	/// Thread dispatch for updating particles speed
	unsigned step_index = numParticles / numThreads;
	unsigned remainder = numParticles % numThreads;

	/// Barrier for syncing all threads
	boost::barrier sync_point{numThreads};

	std::vector<std::jthread> threads;

	for (unsigned curr_index = 0; curr_index < particles.numParticles; curr_index += step_index)
	{
		if (remainder != 0)
		{
			threads.emplace_back(n_body_thread, curr_index, curr_index + step_index + 1,
					std::ref(particles), std::cref(info), std::ref(sync_point));
			++curr_index;
			--remainder;
		}
		else
			threads.emplace_back(n_body_thread, curr_index, curr_index + step_index, std::ref(particles),
					std::cref(info), std::ref(sync_point));
	}

}
```

Ejecución (Ryzen 5 5600X 6P/12L)

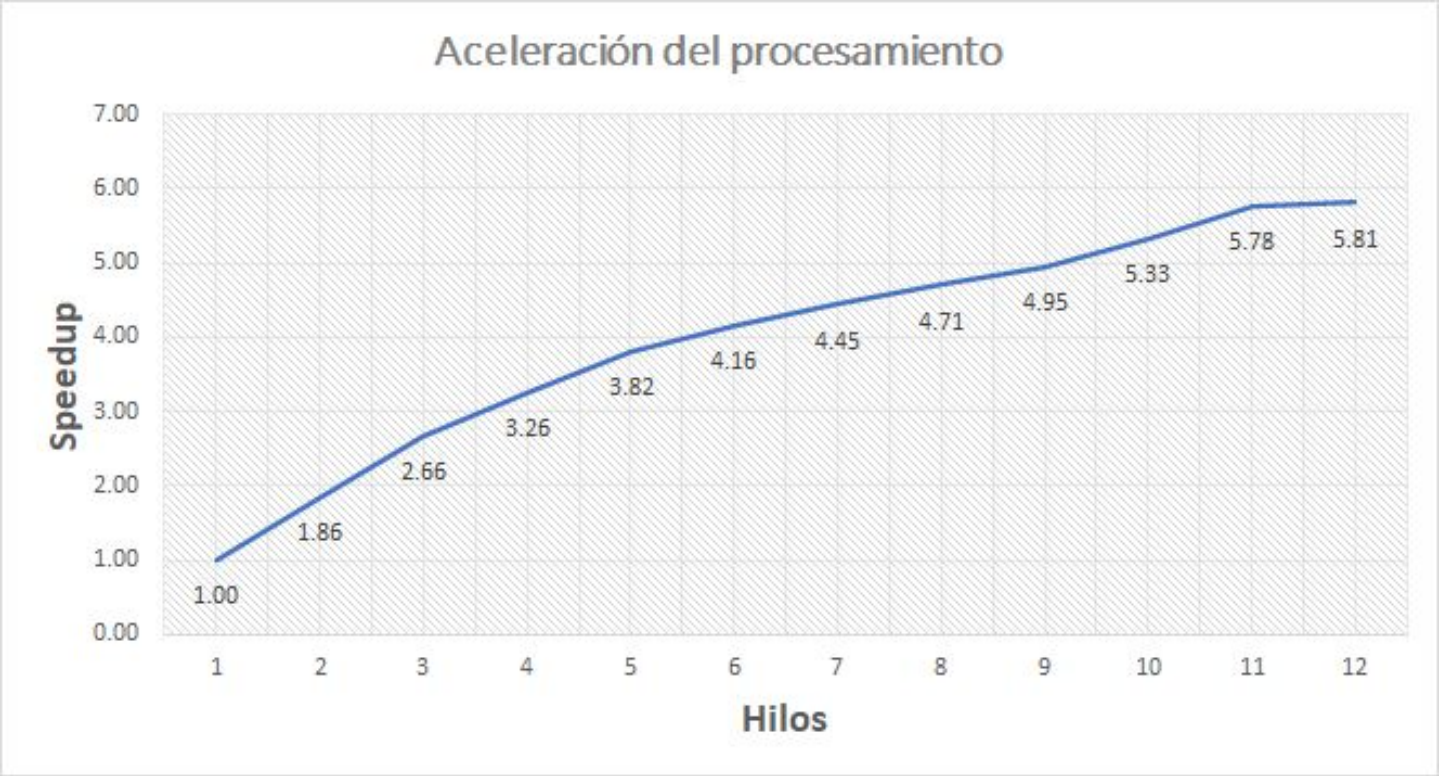
Argumentos del programa:

--particles=25600 --mass=1.0e9 --dt=1.0 --threads=[1-12] --iterations=20

```
Compute info:
        Particle count: 25600 particles
        Particle mass: 1e+09 [kg]
        Timestep: 1 [s]
        Iterations: 20 steps
        Threads: 1

Compute elapsed time:
        49917.1 [ms] (49.9171 [s])
Average time per iteration:
        2.49585e+06 [us]

Process returned 0 (0x0)   execution time : 49.954 s
Press any key to continue.
```

```
Compute info:
        Particle count: 25600 particles
        Particle mass: 1e+09 [kg]
        Timestep: 1 [s]
        Iterations: 20 steps
        Threads: 12

Compute elapsed time:
        8592.51 [ms] (8.59251 [s])
Average time per iteration:
        429626 [us]

Process returned 0 (0x0)   execution time : 8.630 s
Press any key to continue.
```

Ejecución (Ryzen 5 5600X 6P/12L)



Ejecución (Ryzen 5 5600X 6P/12L)

- Aceleramos la ejecución linealmente con el número de núcleos físicos.
- Aun con el aceleramiento, no es suficiente para mostrar más de 4,000 o 5,000 partículas en tiempo real (con 5,000 partículas apenas podemos alcanzar los 16.66 ms).
- ¿Podemos conseguir más velocidad?

Bonus: OpenMP

```
auto n_body_computation_dispatcher(Particle_Set &particles, const Computation_Info &info) -> void
{
	/// Loop for every iteration
	for (unsigned iter = 0; iter < numIterations; ++iter)
	{
		/** 'pragma omp parallel for' indicates that the next for loop is going to
		    split the work equally in 'num threads' (max threads if not specified).
		    First we start with the velocity calculations and then do the positions.
		*/
		#pragma omp parallel for num threads(numThreads)
		for (unsigned p = 0; p < numParticles; ++p)
		{
			n_body_velocity_calc(p, particles, timeStep);
		}

		/** By default OpenMP sets a barrier after the parallel for loop and waits
		    for all threads to finish their work.
		*/

		#pragma omp parallel for num threads(numThreads)
		for (unsigned p = 0; p < numParticles; ++p)
		{
			n_body_position_calc(p, particles, timeStep);
		}
	}

}
```

OpenMP

OpenMP

```
Compute info:
        Particle count: 25600 particles
        Particle mass: 1e+09 [kg]
        Timestep: 1 [s]
        Iterations: 20 steps
        Threads: 12

Compute elapsed time:
        8463.57 [ms] (8.46357 [s])
Average time per iteration:
        423179 [us]

Process returned 0 (0x0)   execution time : 8.550 s
Press any key to continue.
```

C++ threads + boost barrier

```
Compute info:
        Particle count: 25600 particles
        Particle mass: 1e+09 [kg]
        Timestep: 1 [s]
        Iterations: 20 steps
        Threads: 12

Compute elapsed time:
        8371.62 [ms] (8.37162 [s])
Average time per iteration:
        418581 [us]

Process returned 0 (0x0)   execution time : 8.410 s
Press any key to continue.
```

CUDA/OpenGL

¿Cómo empezar?

- Cada partícula es responsable de calcular su posición y velocidad
- Hay muchas partículas en los sistemas que queremos modelar

Entonces podemos crear un “hilo” por cada partícula en la GPU y ejecutar de manera paralela todos los hilos.

- Potencialmente todas las partículas calcularán su posición y velocidad de manera perfectamente paralela, por lo que el tiempo de ejecución sería igual al tiempo que le toma a una sola partícula calcular su posición y velocidad

Kernel

- Es una función que describe la ejecución de un solo hilo.
- Cuando despachamos un kernel le decimos cuantos hilos queremos ejecutar.
- **Si** cumplimos con algunos requisitos de cantidad de hilos y memoria, todos los hilos **podrían ejecutarse de forma paralela.**

```
__device__ float dist2(float3 A, float3 B)
{
	float3 C = A - B;
	return dot(C, C);
}

__global__ void n_body_calc(float3* positions, float3* velocities,
                            unsigned numParticles, float mass, float deltaTime)
{

	unsigned i = blockDim.x * blockIdx.x + threadIdx.x;

	if (i >= numParticles)
	return;

	const float G = 6.6743e-11f;

	float3 force = make_float3(0.0f, 0.0f, 0.0f);

	for (unsigned j = 0; j < numParticles; ++j)
	{
		if (i == j)
			continue;

		float inv_distance2 = 1.0f / dist2(positions[i], positions[j]);
		float3 direction = normalize(positions[j] - positions[i]);

		force += G * mass * mass * inv_distance2 * direction;
	}

	float3 acceleration = force / mass;

	velocities[i] += acceleration * deltaTime;

	/// ???

	positions[i] += velocities[i] * deltaTime;
}
```

```
__device__ float dist2(float3 A, float3 B)
{
	float3 C = A - B;
	return dot(C, C);
}

__global__ void n_body_calc(float3* positions, float3* velocities,
                            unsigned numParticles, float mass, float deltaTime)
{

	unsigned i = blockDim.x * blockIdx.x + threadIdx.x;

	if (i >= numParticles)
	return;

	const float G = 6.6743e-11f;

	float3 force = make_float3(0.0f, 0.0f, 0.0f);

	for (unsigned j = 0; j < numParticles; ++j)
	{
		if (i == j)
			continue;

		float inv_distance2 = 1.0f / dist2(positions[i], positions[j]);
		float3 direction = normalize(positions[j] - positions[i]);

		force += G * mass * mass * inv_distance2 * direction;
	}

	float3 acceleration = force / mass;

	velocities[i] += acceleration * deltaTime;

	/// ???

	positions[i] += velocities[i] * deltaTime;
}
```

```
__device__ float dist2(float3 A, float3 B)
{
	float3 C = A - B;
	return dot(C, C);
}

__global__ void n_body_calc(float3* positions, float3* velocities,
                            unsigned numParticles, float mass, float deltaTime)
{

	unsigned i = blockDim.x * blockIdx.x + threadIdx.x;

	if (i >= numParticles)
	return;

	const float G = 6.6743e-11f;

	float3 force = make_float3(0.0f, 0.0f, 0.0f);

	for (unsigned j = 0; j < numParticles; ++j)
	{
		if (i == j)
			continue;

		float inv_distance2 = 1.0f / dist2(positions[i], positions[j]);
		float3 direction = normalize(positions[j] - positions[i]);

		force += G * mass * mass * inv_distance2 * direction;
	}

	float3 acceleration = force / mass;

	velocities[i] += acceleration * deltaTime;

	/// ???

	positions[i] += velocities[i] * deltaTime;
}
```

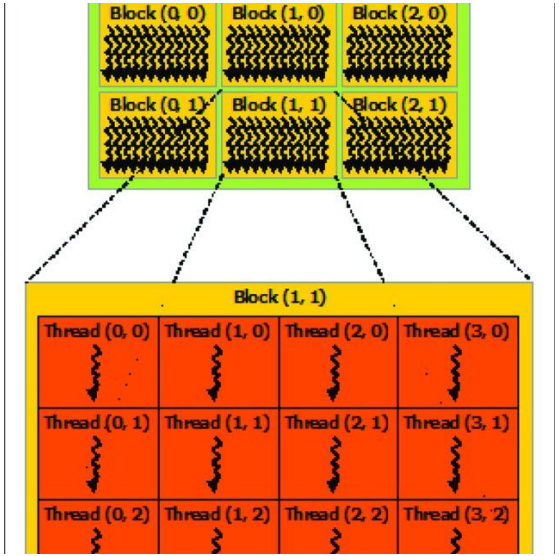
```
__device__ float dist2(float3 A, float3 B)
{
	float3 C = A - B;
	return dot(C, C);
}

__global__ void n_body_calc(float3* positions, float3* velocities,
                            unsigned numParticles, float mass, float deltaTime)
{

	unsigned i = blockDim.x * blockIdx.x + threadIdx.x;

	if (i >= numParticles)
	return;

	const float G = 6.6743e-11f;

	float3 force = make_float3(0.0f, 0.0f, 0.0f);

	for (unsigned j = 0; j < numParticles; ++j)
	{
		if (i == j)
			continue;

		float inv_distance2 = 1.0f / dist2(positions[i], positions[j]);
		float3 direction = normalize(positions[j] - positions[i]);

		force += G * mass * mass * inv_distance2 * direction;
	}

	float3 acceleration = force / mass;

	velocities[i] += acceleration * deltaTime;

	/// __syncthreads() ????

	positions[i] += velocities[i] * deltaTime;
}
```

Cómo funcionan las GPUs

- Al despachar un kernel, especificamos el **número de hilos por bloque**, y el **número de bloques por red**.
- (hilos por bloque) x (bloques por red) = número de hilos totales

¿Por qué dividir los hilos de esta forma?

La respuesta es por cómo está estructurado el hardware de una GPU.



Problema #1

- Saltandonos la explicación del hardware, cada hilo solo puede comunicarse con su propio bloque (desconoce la existencia de los demás).
- Por lo tanto no puede haber una sincronización “global” (bueno, hay una forma [**Cooperative Groups**] pero requiere cumplir varias condiciones, una de ellas es que todos los hilos se pueden ejecutar al mismo tiempo…).

- __syncthreads() solo sincroniza los hilos de **un** solo grupo

¿Solución? Dividir el trabajo en dos kernels

```
__global__ void n_body_vel_calc(float3* positions, float3* velocities,
                                unsigned numParticles, float mass, float deltaTime)
{
      /// ...
      velocities[i] += acceleration * deltaTime;
}
__global__ void n_body_pos_calc(float3 *positions, float3 *velocities,
                                 unsigned numParticles, float deltaTime)
{
      unsigned i = blockDim.x * blockIdx.x + threadIdx.x;
      if (i >= numParticles)
            return;
      positions[i] += velocities[i] * deltaTime;
}
```

Despacho

- Teniendo el número de partículas (= número de hilos) y el número de hilos por bloque (elegido arbitrariamente un múltiplo de 32):

```
/// Kernel variables for dispatching
unsigned threadsPerBlock = workgroupSize; /* 32, 64, 128, ... */
//unsigned blocksPerGrid = numParticles / workgroupSize; //incorrect
unsigned blocksPerGrid = (numParticles + workgroupSize - 1) / workgroupSize;


/** d_positions -> array of positions already in VRAM
    d_velocities -> array of velocities already in VRAM
*/
for (unsigned iter = 0; iter < numIterations; ++iter)
{
      n_body_vel_calc<<<blocksPerGrid, threadsPerBlock>>>(d_positions, d_velocities, numParticles, mass,
                                                          timeStep);

      n_body_pos_calc<<<blocksPerGrid, threadsPerBlock>>>(d_positions, d_velocities, numParticles, timeStep);
}
```

- Al no especificar un stream, un kernel se ejecuta cuando haya terminado el anterior.

OpenGL Compute Shader

- Los compute shaders empiezan con la versión de OpenGL a utilizar, el número de hilos por bloque y los “argumentos”.

```
#version 460
layout(local_size_x=32) in;

layout(location = 0) uniform uint numParticles;
layout(location = 1) uniform float mass;
layout(location = 2) uniform float dt;

layout(std430, binding=0) buffer pblock { vec3 positions[]; };
layout(std430, binding=1) buffer vblock { vec3 velocities[]; };
```

OpenGL Compute Shader

- El cuerpo de la función es casi igual al de CUDA.

```
float dist2(vec3 A, vec3 B)
{
	vec3 C = A - B;
	return dot( C, C );
}

void main()
{
	int i = int(gl_GlobalInvocationID);

	if (i >= numParticles)
		return;

	const float G = 6.6743e-11f;

	vec3 force = vec3(0.0);

	for (uint j = 0; j < numParticles; ++j)
	{
		if (i == j)
			continue;

		float inv_distance2 = 1.0 / dist2(positions[i], positions[j]);
		vec3 direction = normalize(positions[j] - positions[i]);

		force += G * mass * mass * inv_distance2 * direction;
	}

	vec3 acceleration = force / mass;

	velocities[i] += acceleration * dt;

}
```

OpenGL Compute Shader

- A diferencia de CUDA, en OpenGL los shaders se compilan en tiempo de ejecución, por lo que el código está como texto dentro del programa (también se puede importar de un archivo de texto).

```
std::string n_body_pos_calculation= R"""(
      #version 460
      layout(local_size_x=32) in;

      layout(location = 0) uniform int numParticles;
      layout(location = 1) uniform float dt;

      layout(std430, binding=0) buffer pblock { vec3 positions[]; };
      layout(std430, binding=1) buffer vblock { vec3 velocities[]; };


      void main()
      {
            int i = int(gl_GlobalInvocationID);

            if (i >= numParticles)
                  return;

            positions[i] += velocities[i] * dt;

      }   )""";
```

OpenGL Compute Shader

- Para realizar el cómputo, es necesario indicar el número de bloques y que programa se va a ejecutar. Como los shaders se ejecutan de manera paralela, es necesario sincronizarlos.

```
// Compute dispatch
for (unsigned iter = 0; iter < numIterations; ++iter)
{
	glUseProgram(n body vel program);
	glDispatchCompute((numParticles + workgroupSize - 1)/workgroupSize, 1, 1);

	/// Sync point
	glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);

	glUseProgram(n body pos program);
	glDispatchCompute((numParticles + workgroupSize - 1)/workgroupSize, 1, 1);

	/// Sync point
	glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
}
```

Resultados CUDA (RTX 3070 Ti)

n_body.cu

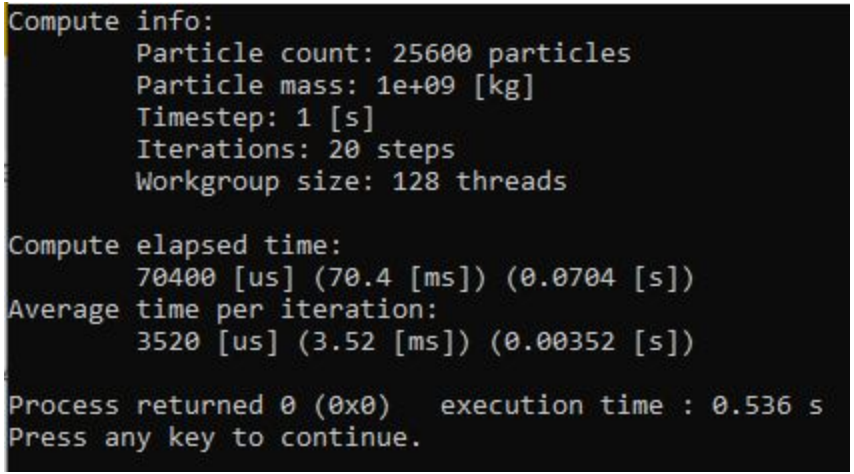
-O2 -Xptxas -O3

```
Compute info:
        Particle count: 25600 particles
        Particle mass: 1e+09 [kg]
        Timestep: 1 [s]
        Iterations: 20 steps
        Workgroup size: 128 threads

cudaMalloc elapsed time:
        85 [us]
Time to copy memory from host to device:
        334 [us]
Compute elapsed time:
        76962 [us] (76.962 [ms]) (0.076962 [s])
Average time per iteration:
        3848.1 [us] (3.8481 [ms]) (0.0038481 [s])
Time to copy memory from device back to host:
        134 [us]
Total time:
        77515 [us] (77.515 [ms]) (0.077515 [s])

Process returned 0 (0x0)   execution time : 0.199 s
Press any key to continue.
```

Resultados OpenGL (RTX 3070 Ti)

n_body.cu

-O2



Cómo funciona CUDA y OpenGL

- Por defecto, OpenGL utiliza “fast-maths” para sus operaciones de punto flotante.
  - Cambia las divisiones de la aproximación más exacta posible (IEEE 754) a una aproximación rápida (división intrínseca).
  - Cambia las raíces de la aproximación más exacta posible a una aproximación rápida
  - Entre otros

Resultados CUDA (RTX 3070 Ti)

n_body.cu

-O2 -Xptxas -O3

--use_fast_math

```
Compute info:
        Particle count: 25600 particles
        Particle mass: 1e+09 [kg]
        Timestep: 1 [s]
        Iterations: 20 steps
        Workgroup size: 128 threads

cudaMalloc elapsed time:
        110 [us]
Time to copy memory from host to device:
        315 [us]
Compute elapsed time:
        60095 [us] (60.095 [ms]) (0.060095 [s])
Average time per iteration:
        3004.75 [us] (3.00475 [ms]) (0.00300475 [s])
Time to copy memory from device back to host:
        162 [us]
Total time:
        60682 [us] (60.682 [ms]) (0.060682 [s])

Process returned 0 (0x0)   execution time : 0.175 s
Press any key to continue.
```

Asi que…

- Con las aproximaciones rápidas, la versión de CUDA es más rápida que la de OpenGL.
- Mientras que en OpenGL no podemos personalizar mucho la compilación de los shaders (no podemos desactivar las aproximaciones rápidas), en CUDA podemos tener lo mejor de dos mundos (aproximaciones exactas o aproximaciones rápidas).

Finalmente

- Con OpenGL y CUDA pudimos calcular las nuevas posiciones de las partículas en 16.66 ms o menos, dándonos la velocidad necesaria para mostrar miles o millones de partículas en tiempo real (hasta 70,000 partículas a 60 Hz).
- Este aceleramiento no hubiera sido posible en la CPU.
- Hay cosas que la CPU hace mejor, y otras la GPU (una no reemplaza a la otra).

Fin