Targetes Gràfiques i Acceleradors

Examen

Albert Bernal

21/06/2021

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

ÍNDICE

Pregunta 1

En el ámbito de la computación, la definición de carácter general de un *pipeline* es que éste es un conjunto de elementos de procesamiento de datos conectados en serie, donde la salida de un elemento es la entrada del siguiente. En el caso que nos ocupa, el *pipeline* gráfico, la idea es la misma, solo que aplicada sobre un caso concreto: el procesamiento de gráficos para renderizar una escena en pantalla.

Los pasos que se ejecutan han ido variando a lo largo del tiempo, ya que se han ido adaptando a las GPUs que salían al mercado. En este apartado explicaremos brevemente el *pipeline* tradicional. Como se puede observar en la figura 1, los pasos son los siguientes:

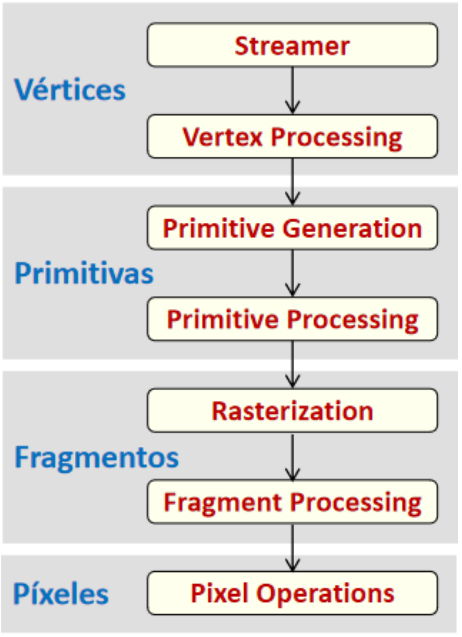


Figura 1. *Pipeline* gráfico tradicional.

- **Streamer**: Este es el primer paso del pipeline y su principal función es preparar los vértices que se le envían para que puedan ser procesados en el siguiente paso. Debe obtener la información, por ejemplo en vectores de vértices en posiciones consecutivas, y tratarla, asociando vértices con sus posibles atributos. Dependiendo de la potencia del hardware, estos vértices podrán tener más o menos atributos, es decir, más o menos información que los describa.
- **Procesado de vértices**: Este paso, también conocido como *vertex shader*, tiene la principal función de proyectar los vértices sobre el plano y aplicar la iluminación. Es programable, pero no se trata de programas muy extensos, ya que se suelen realizar transformaciones similares para todos los vértices (una operación sobre la matriz en sí). Éste es un elemento del *pipeline* masivamente paralelo, ya que todos los vértices son independientes entre sí y se les aplica el mismo *shader*. Por lo tanto, podemos aprovechar la arquitectura de la GPU (por algo están diseñadas así).
- **Generación de primitivas**: Este es un paso no programable que simplemente transforma los tríos de vértices (a,b,c) en primitivas (A), que podríamos entender que hacen referencia al triángulo en sí. Al no haber dependencias entre vértices, tampoco las hay entre triángulos.
- **Procesamiento de primitivas**: Este elemento tampoco es programable. Su función consiste en aplicar *clipping*, eliminar las primitivas que no aparezcan en el área de visión o sean menores a 1 píxel y no hace falta renderizar; y *culling*, eliminar las primitivas que quedarán ocultas por otras primitivas y tampoco hace falta renderizar.

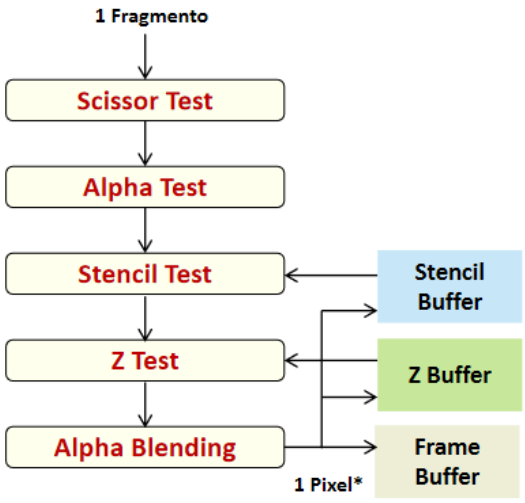
Antes del siguiente paso, se han de preparar los datos para éste. Eso consiste en realizar el *triangle setup*, una operación con la que obtenemos las aristas que forman el triángulo. Es exigente computacionalmente.

- **Rasterización**: Este elemento tampoco es programable. Consiste en recorrer los triángulos para generar fragmentos, algunos de los cuáles se "pintarán" en el *frame buffer* como píxeles. El número de fragmentos generados por triángulo cada vez es menor, con lo que obtenemos más detalle a costa de aumentar la magnitud del cálculo a realizar. También es altamente exigente, por lo que nos encontramos con un *trade-off*: a más fragmentos por triángulo más realismo, pero menos fluidez y jugabilidad (menos *frames* por segundo). Una forma de minimizar este problema es mediante texturas, que se aplican a un mismo objeto y requieren menos cálculo ya que son imágenes ya creadas.

- **Procesamiento de fragmentos**: Este paso, también conocido como *fragment shader*, tiene la principal función de calcular el color en función de valores como transparencia, niebla, texturas, etc. Es programable, y hay que procurar aprovechar la localidad espacial, gracias a que, si juntamos en memoria los fragmentos de un mismo triángulo, es muy probable que éstos requieran una operación similar.

Al igual que el procesado de vértices, es un elemento del *pipeline* masivamente paralelo, ya que todos los fragmentos son independientes entre sí y se les aplica el mismo *shader*. Por lo tanto, también podemos aprovechar la arquitectura de la GPU. Cabe destacar que, pese a que en el *pipeline* tradicional *vertex shader* y *fragment shader* están separados, desde hace unos 15 años se unificaron en un mismo *shading program*.

- **Operaciones de píxeles**: Por último, tenemos que "pintar" la imagen en el *frame buffer*. Este elemento tampoco es programable y consiste en decidir si los fragmentos llegarán al *frame buffer* o no. Para ello, existen muchas técnicas, que pueden ser aplicadas también modo *pipeline*, donde la salida de un elemento sea la entrada del siguiente. Los que se han estudiado en el curso son los siguientes:



1. **Scissor test**: descarta los píxeles que estén fuera del área de renderizado.
2. **Alpha test**: descarta los fragmentos muy transparentes, es decir, con un α muy pequeño.
3. **Stencil test**: generalmente se usa para generar sombras.
4. **Z test**: para dos píxeles con las mismas coordenadas x e y, descarta el que tenga mayor z (profundidad).
5. **Alpha blending**: combina una imagen con un fondo para crear la apariencia de transparencia, ya sea total o parcial.

Figura 2. Algunas operaciones de píxeles.

Pregunta 2

Rutina original:

```
void Examen21(float mA[N][M], float mB[N][M], float vC[N], float
vD[M])
{
  int i, j;
  for (i=0; i<N; i++)
    for (j=0; j<M; j++)
      mA[i][j] = mA[i][j]*vC[i] -mB[i][j]*vD[j]+ mA[i][0]*mB[7][j];
}
```

a)

```
__global__void Examen21(int N, int M,float *mA, float *mB, float *vC,
float *vD)
{
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  int ind = j;
  if (j < M)
    for (int i=0;  i<N; i++, ind+=M)
      mA[ind] = mA[ind]*vC[i] - mB[ind]*vD[j] + mA[ind-j]*mB[7M+j];
}
nThreads= 256;
nBlocks= (M+nThreads-1)/nThreads;
dim3 dimGridC(nBlocks, 1, 1);
dim3 dimBlockC(nThreads, 1, 1);
Examen21<<<dimGridC, dimBlockC>>>(N, M, dmA, dmB, dvC, dvD);
```

b)

```
__global__void Examen21(int N, int M,float *mA, float *mB, float *vC,
float *vD)
{
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int ind = i*M;
  if (j < N)
    for (int j=0;  j<M; j++, ind++)
      mA[ind] = mA[ind]*vC[i] - mB[ind]*vD[j] + mA[ind-j]*mB[7M+j];
}
nThreads = 256;
nBlocks = (N+nThreads-1)/nThreads;
dim3 dimGridF(1, nBlocks, 1);
dim3 dimBlockF(1, nThreads, 1);
Examen21<<<dimGridF, dimBlockF>>>(N, M, dmA, dmB, dvC, dvD);
```

c)

```
__global__void Examen21(int N, int M,float *mA, float *mB, float *vC,
float *vD)
{
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  int ind = i*M+j;
  if (i < N && j < M)
      mA[ind] = mA[ind]*vC[i] - mB[ind]*vD[j] + mA[ind-j]*mB[7M+j];
}
nThreads = 16;
nBlocksC = (M+nThreads-1)/nThreads;
nBlocksF = (N+nThreads-1)/nThreads;
dim3 dimGridE(nBlocksC, nBlocksF, 1);
dim3 dimBlockE(nThreads, nThreads, 1);
Examen21<<<dimGridE, dimBlockE>>>(N, M, dmA, dmB, dvC, dvD);
```

Pregunta 3

Antes de empezar, comentar que el mercado multi-GPU para videojuegos ya no es una solución interesante como hace unos años, cuando el rendimiento en SLI era justificable por el precio y el soporte por parte de los desarrolladores era mucho mayor. Por ejemplo, hace unos años era una buena solución comprar dos GTX 970 de segunda mano y correrlas en SLI frente a comprar una de las nuevas 1080Ti, ya que se obtenía un rendimiento comparable y el precio era razonable. A día de hoy, hay ciertas opciones como usar dos RTX 3090 o RX 6800XT. ¿Vale la pena? Bueno, en primer lugar, a mediados de abril de 2021 la cantidad de juegos que soportaban de forma nativa configuraciones SLI era trece. Sí, de todos los juegos que se podían jugar en ese momento, solo trece funcionaban con SLI. En el lado de AMD, la historia es un poco diferente. No hay soporte oficial, pero el *software* Radeon permite ejecutar configuraciones multi-GPU. No hay conexión física, como el enlace de Nvidia, funciona todo por *software*.

Además, la experiencia de ejecutar dos 3090 es de todo menos ideal. Algunos juegos tienen ganancias sólidas, otros ni siquiera arrancan, por lo que el valor que podemos obtener decantándonos por multi-GPU en general es bastante bajo. Se puede decir lo mismo para el caso de las 6800XT. Por último, añadir que se necesita una fuente de alimentación bestial para este tipo de configuración junto con el resto del sistema que también debe ser de primera categoría para poder exprimir la potencia de las gráficas (aunque, si se tiene el dinero para comprar dos 3090 no creo que eso suponga un gran problema).

Por lo tanto, para los juegos, las configuraciones de tarjetas gráficas duales definitivamente no valen la pena en 2021. Son extremadamente caras, especialmente en el estado del mercado que nos encontramos. Y por ese precio, obtienes soporte en una docena de juegos y posibilidades extremadamente escasas de que cualquier título futuro venga con soporte para SLI.

Una vez descrito el contexto, veamos qué tenemos que tener en cuenta a la hora de paralelizar una carga de trabajo en dos GPU. Por lo general, hay dos formas de distribuir la computación entre varios dispositivos: gracias al paralelismo de datos, donde un solo modelo se replica en múltiples dispositivos o múltiples máquinas. Cada uno de ellos procesa diferentes *batches* de datos y luego fusiona sus resultados. Existen muchas variantes de esta configuración, que difieren en cómo se fusionan los resultados de las diferentes réplicas del modelo, en si permanecen sincronizadas en cada *batch* o si están acopladas de manera más flexible, etc. También existe el paralelismo de modelos, donde diferentes partes de un solo modelo se ejecutan en diferentes dispositivos, procesando un solo *batch* de datos juntos. Esto

funciona mejor con modelos que tienen una arquitectura naturalmente paralela, como los modelos que presentan múltiples ramas sin dependencias entre ellas.

En el caso particular de renderizar escenas de un videojuego, tenemos una aplicación masivamente paralela. Pero claro, una GPU sola ya aprovecha ese paralelismo, y solo veremos una mejora de rendimiento si encontramos un caso en el que se sature una GPU y pueda ser necesaria más potencia o memoria. Pongamos el caso de una 3090. Ésta cuenta con 24GB de memoria GDDR6X y un ancho de banda de cerca de 1TB/s. A menos que queramos ejecutar un juego a altísima resolución, el *gaming* no es el escenario en el que se saturaría. En un caso más limitado como una 970, que cuenta con solo 4GB GDDR5, juegos a QHD o UHD la saturarán fácilmente. Por lo tanto, puede ser rentable el *overhead* que obtenemos debido a la tarea de dividir la carga de trabajo y sincronizar las GPU. Existen distintas formas de distribuir la carga de trabajo entre GPUs. El más corriente es AFR (*Alternate Frame Rendering*) en el que una GPU se ocupa de un *frame* y la otra del siguiente, es decir, el controlador divide la carga de trabajo alternando GPU en cada fotograma. Por ejemplo, en un sistema con dos GPU habilitadas para SLI, la GPU 1 representaría el cuadro 1, la GPU 2 representaría el cuadro 2, la GPU 1 representaría el cuadro 3, y así sucesivamente. Este suele ser el modo de representación SLI preferido, ya que divide la carga de trabajo de manera uniforme entre las GPU y requiere poca comunicación entre GPU. Aunque también hay una propuesta, impulsada principalmente por AMD estos últimos años, llamado SFR (*Split Frame Rendering*), que básicamente divide la pantalla. En ese caso, el controlador dividirá la carga de trabajo de la escena en varias regiones y asignará estas regiones a diferentes GPU. Por ejemplo, en un sistema con dos GPU habilitadas para SLI, la imagen puede dividirse verticalmente, con GPU 1 representando la región izquierda y GPU 2 representando la región derecha. El renderizado también equilibra la carga dinámicamente, por lo que la división cambiará siempre que el controlador determine que una GPU está funcionando más que otra. Este modo de renderizado SLI normalmente no es tan óptimo como el modo AFR, ya que parte del trabajo se duplica y el *overhead* de comunicación es mayor.

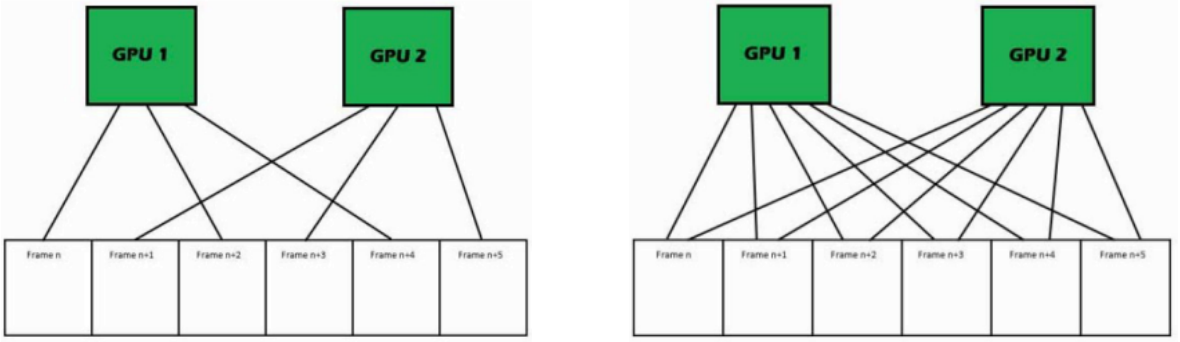


Figura 3. Distribución de *frames* que procesa cada GPU, AFR vs SFR, Nvidia, 2014

Por último, veamos brevemente las distintas tecnologías que proporcionan soporte multi-GPU y que permiten sincronizarse y comunicarse a las GPUs:

- **SLI**: *Scalable Link Interface*, es la tecnología que presentó en 2004 Nvidia para repartir la carga en más de una GPU. Desgraciadamente para AMD, ha sido la tecnología referente. Se pueden configurar de 2 a 4 tarjetas, siendo una de ellas *master* y las demás *slave*. Todas reciben la misma carga de trabajo para renderizar, pero la salida final de cada tarjeta se envía a la *master* a través de un conector llamado SLI Bridge.
- **Crossfire**: Esta es la propuesta de AMD para competir con SLI. En general, funciona igual, en una configuración *master-slave*. La mayor ventaja que tiene sobre SLI es que puede ejecutar tarjetas que no son exactamente del mismo modelo o del mismo fabricante. Sin embargo, las cartas deben ser de la misma serie.
- **NVLink**: Esta es la sucesora de SLI, presentada por Nvidia en 2014. Difiere de su predecesora en que el conector es distinto y soporta más ancho de banda y en que la relación entre las GPU no es *master-slave*, sino que se usa *mesh networking*, una topología de red local que permite tratar a todos los nodos (cada GPU) como iguales y mejorar la velocidad de renderizado. Su mayor beneficio, a parte del mayor ancho de banda, es que la memoria de ambas gráficas es accesible todo el tiempo. Por lo tanto, si unimos dos 3090, tendremos 48GB de memoria (24+24).

Pregunta 4

Pese a que el paralelismo es una gran mejora para la eficiencia de nuestros programas, siempre viene acompañado de dificultades con las que tenemos que lidiar para que todo funcione correctamente, en el orden que se quiere y respetando las dependencias en el código. Los eventos en CUDA son una forma de hacer más fácil la paralelización con un método similar al de las tareas de toda la vida. En pocas palabras, son entidades abstractas que nos ayudan a separar el código en partes distintas para poder definir cuáles pueden ser ejecutadas y en qué orden.

Además de ayudar en la sincronización, los eventos facilitan uno de los aspectos más importantes a la hora de escribir código: evaluar su rendimiento. Esto es importante sobre todo en lenguajes que estén enfocados a aprovechar el paralelismo, ya que puede ser muy útil comprobar si vale la pena invertir el tiempo de desarrollo frente a ejecutar simplemente la versión secuencial.

Pese a que se puede *hardcodear* en cualquier lenguaje usando herramientas del sistema, CUDA introdujo una API (CUDA Runtime API) con funcionalidades para hacer más fácil y preciso el *tracking* de tiempos y estadísticas de nuestros programas y la sincronización de tareas. Mediante unas funciones proporcionadas por NVIDIA, podemos administrar eventos y recopilar información sobre lo que sucede en éstos.
Las funciones son las siguientes[1]:

```
__host__ cudaError_t cudaEventCreate ( cudaEvent_t* event )
```
Crea un evento usando la definición de evento por defecto.

```
__host__ __device__ cudaError_t cudaEventCreateWithFlags ( cudaEvent_t*
event, unsigned int  flags )
```
Crea un evento con *flags*. Posibles *flags*:

- **cudaEventDefault**: *flag* de creación de eventos predeterminado.
- **cudaEventBlockingSync**: especifica que el evento debe usar sincronización de bloqueo, es decir, un *thread* que use cudaEventSynchronize() para esperar un evento creado con este *flag* se bloqueará hasta que el evento se complete.
- **cudaEventDisableTiming**: especifica que el evento creado no necesita registrar datos de tiempo. Los eventos creados con este indicador especificado y el indicador cudaEventBlockingSync no especificado proporcionarán el mejor rendimiento cuando se utilicen con cudaStreamWaitEvent() y cudaEventQuery().
- **cudaEventInterprocess**: especifica que cudaIpcGetEventHandle() puede usar el evento creado como un evento entre procesos. Debe especificarse junto con cudaEventDisableTiming.

[1] Información extraída del manual de la CUDA Runtime API, enlace en la bibliografía.

```
__host__ __device__ cudaError_t cudaEventDestroy ( cudaEvent_t event )
```
Destruye un evento. Un evento puede destruirse antes de que se complete. En este caso, la llamada no se bloquea una vez finalizado el evento, y cualquier recurso asociado se liberará automáticamente de forma asíncrona al finalizar.

```
__host__ cudaError_t cudaEventElapsedTime ( float* ms, cudaEvent_t start, cudaEvent_t end )
```
Calcula el tiempo transcurrido entre eventos en milisegundos, con un error de ±0.5 ms.

```
__host__ cudaError_t cudaEventQuery ( cudaEvent_t event )
```
Consulta el estado de un evento. Retorna *cudaSuccess* si el trabajo por hacer ha sido completado o *cudaErrorNotReady* si aún está incompleto.

```
__host__ __device__ cudaError_t cudaEventRecord ( cudaEvent_t event, cudaStream_t stream = 0 )
```
Registra un evento. Captura el contenido del *stream* en el momento de ejecutar esta llamada. Se puede llamar varias veces en el mismo evento y sobrescribirá el estado capturado previamente.

```
__host__ cudaError_t cudaEventRecordWithFlags ( cudaEvent_t event, cudaStream_t stream = 0, unsigned int flags = 0 )
```
Registra un evento con *flags*. Posibles *flags*:

- **cudaEventRecordDefault**: *flag* de creación de eventos predeterminado.
- **cudaEventRecordExternal**: el evento se captura en el grafo como un nodo de evento externo.

```
__host__ cudaError_t cudaEventSynchronize ( cudaEvent_t event )
```
Espera a que se complete un evento. Como se ha mencionado anteriormente, sirve para sincronizar tareas y que se respeten las dependencias.

Para terminar, un pequeño ejemplo. Para medir el tiempo de ejecución de KernelHistograma:

```
cudaEventCreate(&E0);
cudaEventCreate(&E3);
cudaEventRecord(E0, 0);
KernelHistograma<<<nBlocks, nThreads>>>(d_image, d_redChannel, d_greenChannel, d_blueChannel, minR, minG, minB, multR, multG, multB, numPixels);
cudaEventRecord(E3, 0);
cudaEventSynchronize(E3);
cudaEventElapsedTime(&TiempoTotal,  E0, E3);
cudaEventDestroy(E0);
cudaEventDestroy(E3);
```

Pregunta 5

Para introducir este tema de plataformas para GPGPU, es importante entender que no pertenecen al mismo segmento en cuanto a estilo: mientras que CUDA y OpenCL son lenguajes de programación paralelos, OpenACC consiste simplemente en directivas de compilación y está basado en lenguajes de programación. En este caso, soporta C, C++ y Fortran.

Los lenguajes de programación paralelos permiten ir más al detalle en bajo nivel, y así obtener mucho más rendimiento al aprovechar las características de la arquitectura del sistema sobre el que ejecutaremos el código. Esa posibilidad de optimización los hace más complejos y, consecuentemente, más difíciles de aprender y desarrollar. Por lo tanto, es más adecuado en aplicaciones que puedan ser muy optimizadas o en casos en que sepamos que se tratará de un código que ejecutaremos con mucha frecuencia y que nos permitirá aprovechar ese *speedup* a la larga.

En el caso de las directivas de compilación y por ende OpenACC, encuentran su potencial en exponer algunas optimizaciones más generalistas pero mucho más fáciles de aprender e implementar, sobre todo dado que, si nos estamos adentrando en paralelizar una aplicación, probablemente ya conozcamos el lenguaje que se usa. Esa sencillez trae consigo un rendimiento máximo menor, pero también un código más portable que no está centrado en una arquitectura en concreto.

Un claro ejemplo de la diferencia de rendimiento entre las dos perspectivas es el siguiente, en el que se puede apreciar que ambas técnicas obtienen un muy buen resultado sobre secuencial, con una diferencia de x46 a x96 en *speedup* en una GTX 1070, pero también la superioridad de los lenguajes de programación frente al uso de directivas:

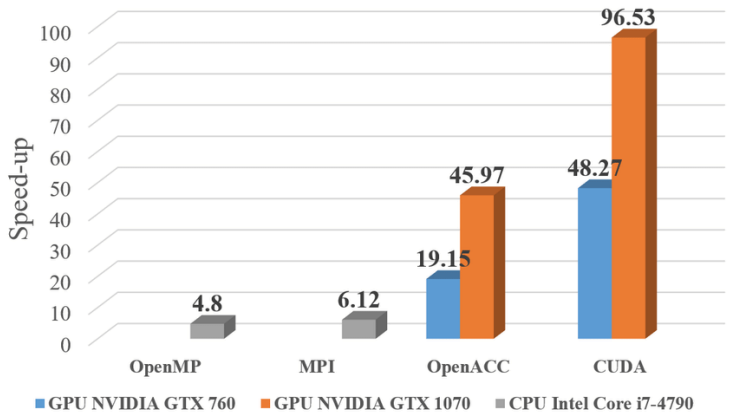


Figura 4. Comparativa de rendimiento entre OpenMP, MPI, OpenACC y CUDA.

Siendo CUDA y OpenCL del mismo grupo, es interesante mencionar ventajas e inconvenientes de ambos. A día de hoy, CUDA y OpenCL son los *frameworks* GPGPU líderes. CUDA es un *framework* cerrado de Nvidia que apareció en el mercado alrededor de 2007 y que no es compatible con tantas

aplicaciones como OpenCL (el soporte sigue siendo amplio, solo que no tanto), pero que cuenta con un soporte integrado de Nvidia que ofrece más calidad y garantiza un rendimiento a un nivel sencillamente superior en la gran mayoría de escenarios. Por lo tanto, el rendimiento es bueno pero está ligado a que el caso de uso del usuario soporte CUDA.

OpenCL, por contra, es de código abierto y es compatible con más aplicaciones que CUDA. Sin embargo, el soporte a menudo es mediocre y actualmente no proporciona los mismos aumentos de rendimiento que tiende a ofrecer CUDA. Para aplicaciones tan populares como el editor de vídeo de Apple, Final Cut Pro, es la única opción. Tiene soporte en las tarjetas gráficas de ambos AMD y Nvidia.

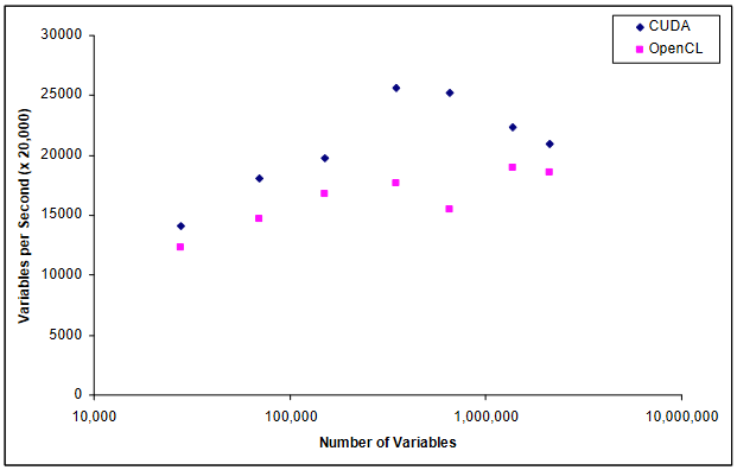


Figura 5. Velocidad de procesamiento para distintos tamaños de problema, CUDA vs OpenCL.

Como se puede apreciar en la imagen, el rendimiento en CUDA es notablemente superior, especialmente en algunos casos en los que la cantidad de variables por segundo que se procesan es alrededor de un 60% más que en OpenCL.

Sobre la cuestión de combinar plataformas, solo es posible la compatibilidad entre OpenACC y CUDA. Como hemos visto, OpenACC tiene una alta productividad pero bajo rendimiento, y es el caso opuesto para CUDA. Si realizamos un buen trabajo dividiendo las partes de nuestra aplicación que serán aceleradas mediante cada opción, podemos obtener una solución más equilibrada, de productividad y rendimiento medios.

Para concluir, me parece interesante comparar qué fabricante de GPUs es mejor para GPGPU. En mi opinión, si comparamos las opciones que hay actualmente en el mercado, las GPUs de Nvidia (especialmente las más nuevas) parecen ser la mejor opción ya que incorporan compatibilidad con CUDA y un sólido rendimiento de OpenCL para cuando CUDA no es compatible. Por lo tanto, si el ratio precio a rendimiento es parecido y nos dan a elegir entre AMD o Nvidia, la segunda es una decisión más segura.

Pregunta 6

Como hemos visto en la pregunta del *pipeline* gráfico, en el algoritmo de renderización encontramos un *trade-off*: a más fragmentos por triángulo más realismo, pero menos fluidez y jugabilidad (menos *frames* por segundo). También hemos mencionado que una forma de reducir el impacto que tiene el realismo en el rendimiento es mediante el uso de texturas.

Aplicar texturas, o texturizar, en una GPU 3D moderna significa el proceso de muestrear una textura, ya sea que contenga una imagen en sí u otros datos, y usarlo como entrada en un programa de sombreado para su posterior procesamiento. Pueden servir para aumentar la calidad de la imagen con un coste menor que con el renderizado tradicional o para aplicar efectos sobre partes de la escena como reflejos en determinados materiales o dar sensación de relieve.

Para aplicar una textura, hay que mapear la imagen de textura con la superficie del elemento u objeto de la escena al que se la queremos aplicar. En muchas ocasiones, como en el ejemplo de una textura que añade reflejos, es muy importante tener en cuenta la posición del observador para que la imagen resultante sea lo más fidedigna posible a un reflejo real. Los reflejos también añaden el coste de renderizar partes de la escena que teóricamente no estarían dentro del campo de visión.

Una vez introducidas las texturas, pasemos a filtros que se les pueden aplicar para mejorar el resultado obtenido, determinando el color de textura para un píxel usando los colores de *texels* cercanos, es decir, píxeles de la textura. Los filtros son necesarios porque, si no se aplican, normalmente nos encontraremos con una escena donde lo que está cerca del punto de vista se verá bien pero lo que está lejos se verá mucho peor, y tendremos una clara línea de separación entre ambas zonas. Los filtros tienen como objetivo reducir la visibilidad de esa línea para hacer la experiencia más realista e inmersiva. Existen dos categorías principales: magnificación y minificación. El primero, en el que el tamaño del píxel es menor que el del *texel*, consiste en un filtro de reconstrucción en el que se interpolan datos con separación entre sí para rellenar huecos. En el segundo, donde la relación de tamaño es inversa, las muestras de textura tienen una frecuencia más alta que la requerida para el relleno de textura, es decir, los datos de textura tienen menos separación entre ellos que en la fuente original. Existen muchos filtros distintos. Los más relevantes, ordenados de menor a mayor potencia requerida y por tanto de peor a mejor resultado, son los siguientes:

- **Lineal**: Consiste en usar el color del *texel* más cercano para determinar el color del píxel. Es muy simple y no obtiene muy buenos resultados. Son aceptables si el tamaño de píxel y *texel* son

similares. Se puede aplicar usando *mipmaps*[2] e interpolando el valor de los *mipmaps* más cercanos para obtener el resultado. Su coste es de 1 acceso a memoria.

- **Bilineal**: Consiste en obtener el color muestreando los cuatro *texels* más cercanos, combinando sus valores con ponderaciones por cercanía al píxel en cuestión, es decir, con mayor peso en el color del píxel cuanto más cercano esté el *textel*. Su coste es de 4 accesos a memoria más la interpolación.
- **Trilineal**: Consiste en obtener el color mediante un filtro bilineal en los dos niveles más cercanos en resolución (los que más se ajusten al tamaño del píxel) e interpolar los resultados con un filtro lineal. Esto soluciona uno de los problemas del filtro bilineal, las caídas abruptas de calidad entre los límites donde se cambiaba de *mipmap*. Su coste es de 8 accesos a memoria más la interpolación.
- **Anisotrópico**: Este es el filtro más usado hoy en día por su rendimiento. Es también el más exigente computacionalmente pero la potencia de las tarjetas gráficas suele ser suficiente. En lugar de simplemente usar texturas progresivamente más pequeñas que reducen a la mitad el ancho y el alto de la textura anterior, también se crean texturas de media anchura y de altura completa y media altura y de anchura completa. Por ejemplo, estas texturas de anchura completa y de media altura proporcionan la calidad necesariamente reducida, pero no hace falta estirarlas, ya que ya tienen el mismo ancho que la textura original, lo que evita el efecto de desenfoque significativo. Su coste es de entre 8 y 128 accesos a memoria más la interpolación.

Figura 6. Ejemplo de aplicación de un filtro anisotrópico 16x.

[2] *Mipmapping* es una técnica que reduce el coste de minificación gracias a que filtra previamente la textura y la guarda en memoria en distintos tamaños, hasta incluso un único píxel.

Pregunta 7

Definición

En pocas palabras, el *ray tracing* es un modelo de iluminación que pretende reproducir el comportamiento que tiene la luz en la vida real. Funciona mediante la simulación de rayos de luz que siguen un camino de la misma forma que lo haría la luz en el mundo físico. Pese a que pueda parecer que es una tecnología nueva debido al auge de popularidad que ha experimentado gracias al lanzamiento de la serie RTX de Nvidia, es una idea que fue conceptualizada ya en 1969 por Arthur Appel, de IBM, hace más de 50 años. No obstante, la fama que ha obtenido recientemente no es para nada injustificada: ha sido la primera vez que se ha presentado una solución viable para aplicar esta tecnología a tiempo real ofreciendo una experiencia jugable.

Funcionamiento

Teóricamente, el trazado de rayos implica proyectar rayos de cada fuente de luz en una escena, generar (generalmente al azar) rayos de luz a partir de ella y seguirlos cuando impactan y se reflejan en las superficies de los elementos de la escena. Para obtener un resultado realista, en cada superficie, las propiedades de la luz se combinan con las propiedades del material que impacta y el ángulo en el que se cruza. La luz, que puede haber adquirido un color diferente al reflejarse en el objeto, se sigue trazando, utilizando múltiples rayos que simulan la luz reflejada, de ahí el término trazado de rayos o *ray tracing*. El proceso de trazado continúa hasta que los rayos abandonan la escena.

Si bien ese proceso obtiene resultados excelentes, a día de hoy se trata simplemente el caso teórico en videojuegos ya que es increíblemente lento, debido a que es un proceso computacionalmente muy exigente. Pese a ello, es optimizable, ya que la mayoría de los rayos no impactan en nada que nos interese y otros rayos pueden rebotar casi indefinidamente. Por lo tanto, las soluciones que han presentado Nvidia, en 2018, y AMD, en 2020, tratan de hacer una optimización inteligente de esos casos, como una especie de *cut-off* sobre el algoritmo. Usan un principio de luz llamado reciprocidad, que establece que el inverso de un haz de luz funciona de la misma manera que el original para proyectar rayos desde la cámara virtual hacia la escena. Eso significa que solo se emiten los rayos que contribuirán a la escena final, lo que aumenta enormemente la eficiencia. Esos rayos se siguen trazando mientras rebotan hasta que impactan en una fuente de luz o salen de la escena. Incluso en este último caso, podría ser en un punto que agrega luz ambiente (como el cielo), por lo que de cualquier manera, la cantidad de iluminación agregada a cada superficie en la que impacta el rayo se agrega a la escena. El *software* también puede limitar la cantidad de reflejos que seguirá un rayo si es probable que la contribución de luz sea pequeña, ya que no representará un cambio notable en la imagen resultado. Muchos juegos ofrecen distintos niveles de trazado de rayos, para ofrecer al usuario poder de decisión en cuanto al

trade-off realismo contra rendimiento. En conjunción con tecnologías como DLSS (*Deep Learning Super Sampling*), que desgraciadamente se escapa de los límites de esta pregunta, permite un rendimiento y detalle aceptables en la mayoría de juegos.

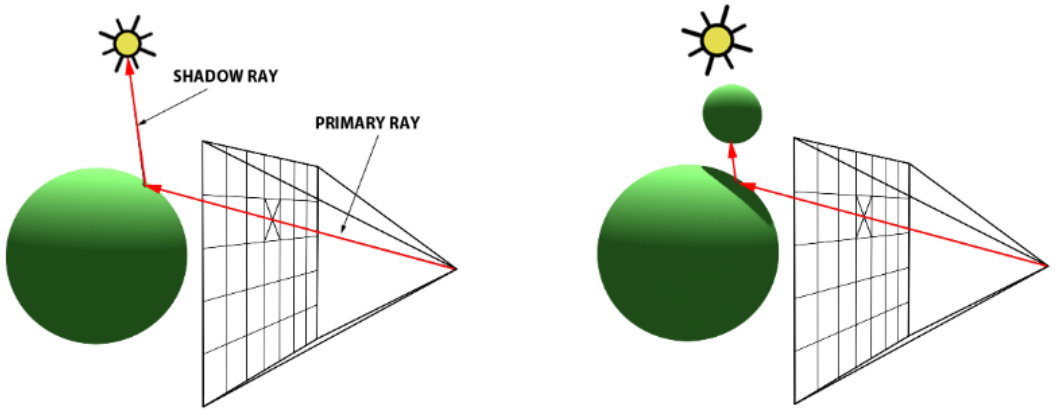


Figura 7. Ejemplo del proceso para representar sombras.

Un escenario muy común que considero interesante comentar es el de la reproducción de sombras. En este ejemplo, el jugador está situado a la derecha de la esfera en ambas situaciones y la principal fuente de luz es representada por el sol. Como podemos ver en la parte izquierda, enviamos un rayo primario desde el punto de vista, con la intención de comprobar si impactará en algún objeto. En este caso, impacta en la esfera grande. Entonces, desde ese punto en el que ha chocado, generamos un rayo de sombra para tratar de averiguar si el punto está iluminado o bajo una sombra. En el escenario de la izquierda, ese rayo llega sin obstrucciones a la fuente de luz, así que podemos deducir que está iluminado. En el escenario de la derecha, ese rayo de sombra se encuentra con una esfera pequeña antes de llegar a la fuente de luz y, por lo tanto, podemos determinar que ésta aplicará sombra sobre el punto de la esfera grande donde generamos el rayo de sombra.

Visto un ejemplo, sigamos hablando de la tecnología. El número total de rayos de luz que caen sobre una escena supera lo que pueden modelar incluso los ordenadores más potentes. Prácticamente, eso significa que los algoritmos de trazado tendrán que elegir un nivel de calidad que determine cuántos rayos se proyectan desde cada píxel de la escena en varias direcciones y, después de calcular dónde cada rayo se cruza con un objeto en la escena, necesita escoger la cierta cantidad de rayos que se seguirán desde cada intersección para modelar la luz reflejada. Calcular esas intersecciones es relativamente caro. Hasta hace poco, también se limitaba a las CPU. Moverlo a la GPU ha proporcionado una mejora importante en la velocidad, pero los resultados de calidad de cine aún requieren casi 10 horas por *frame* en un ordenador de alta gama con múltiples GPU (también debido a que esas imágenes no se renderizan a 1080p, sino a resoluciones como 4K, 8K o incluso 12K, para tener mucha más información y detalle en postproducción).

Debido a que el trazado de rayos es tan intensivo, las aplicaciones interactivas como los juegos no han podido usarlo excepto para generar escenas atractivas de antemano. Para el renderizado en tiempo real, se basan en la rasterización, donde las superficies de cada objeto se sombrean en función de las propiedades del material y las luces que inciden sobre ellos. Después de tantos años optimizando y mejorando los algoritmos de renderizado, se pueden obtener resultados muy vistosos sin hacer uso de *ray tracing*, tanto que, en algunos casos, es difícil para el jugador medio distinguirlos[3]. Ese progreso hace posible que los juegos se rendericen a altas velocidades sin dejar de verse geniales. No obstante, en algunas situaciones se quedan cortos cuando se trata de interacciones sutiles como el *sub-surface scattering*[4]. Para casos como este, el objetivo siempre ha sido el trazado de rayos en tiempo real.

Aparte de la respuesta obvia de años de optimización de código, hay algunos elementos nuevos específicos que ayudan a hacer posible RTX. En primer lugar, para obtener el máximo rendimiento, los *shaders* de todos los objetos de la escena deben cargarse en la memoria de la GPU y estar listos para funcionar cuando sea necesario calcular las intersecciones. RTX también se basa en un nuevo módulo de eliminación de ruido. Ésta es muy importante en el trazado de rayos porque solo podemos emitir un número limitado de rayos de cada píxel en la cámara virtual. Por lo tanto, a menos que tracemos todos los rayos para completar la escena dejando al algoritmo correr durante el tiempo suficiente, tendrá muchos puntos muertos o ruido de aspecto desagradable. Hay algunos proyectos de investigación que ayudan a optimizar qué rayos se emiten, pero aún así el resultado tiene mucho ruido. Si ese ruido se puede reducir por separado, puede producir una salida de calidad mucho más rápido que si abordamos el problema enviando muchos más rayos. Nvidia usa esta técnica para producir *frames* más rápidamente.

El trabajo pesado en cualquier tipo de trazado de rayos es el cálculo de intersecciones. Para cada rayo de luz emitido por la cámara (de los cuales se necesita al menos uno para cada píxel de cada fotograma, y muchos más si también se están simulando ópticas), el *software* necesita encontrar el objeto con el que se cruza, luego, se deben enviar múltiples rayos continuos desde ese objeto como hemos visto en el ejemplo, y se deben calcular las intersecciones, y así sucesivamente hasta que el rayo abandone la escena (y se le asigne un valor basado en el mapa de iluminación ambiental) o impacte en una fuente de luz como el sol de la figura 7. La eliminación de ruido reduce la cantidad de rayos necesarios, pero la cantidad sigue siendo extraordinaria.

---

[3] Is RTX a Total Waste of Money?, Linus Tech Tips, 2021
https://www.youtube.com/watch?v=2VGwHoSrIEU

[4] El *sub-surface scattering* es un mecanismo en el que la luz que penetra en la superficie de un objeto translúcido se dispersa al interactuar con el material y sale de la superficie en un punto diferente.

Para ayudar a abordar este problema, Nvidia ha desarrollado núcleos RT especialmente diseñados como parte de su arquitectura Turing. No he encontrado mucha información de alto nivel sobre ellos y la que era más detallada simplemente escapa de mis capacidades. Mi conclusión es que permiten un cálculo mucho más rápido de la intersección de un rayo y que usan una estructura de datos llamada BVH (*Bounding Volume Hierarchy*) que es común en *ray tracing*.

Comparativa

Para terminar, me gustaría hacer una pequeña comparativa entre las propuestas que tienen ambos Nvidia y AMD en el mercado, muy interesante ya que hace unos meses Nvidia presentó sus nuevas tarjetas de la serie 30 y AMD su propia implementación del trazado de rayos.

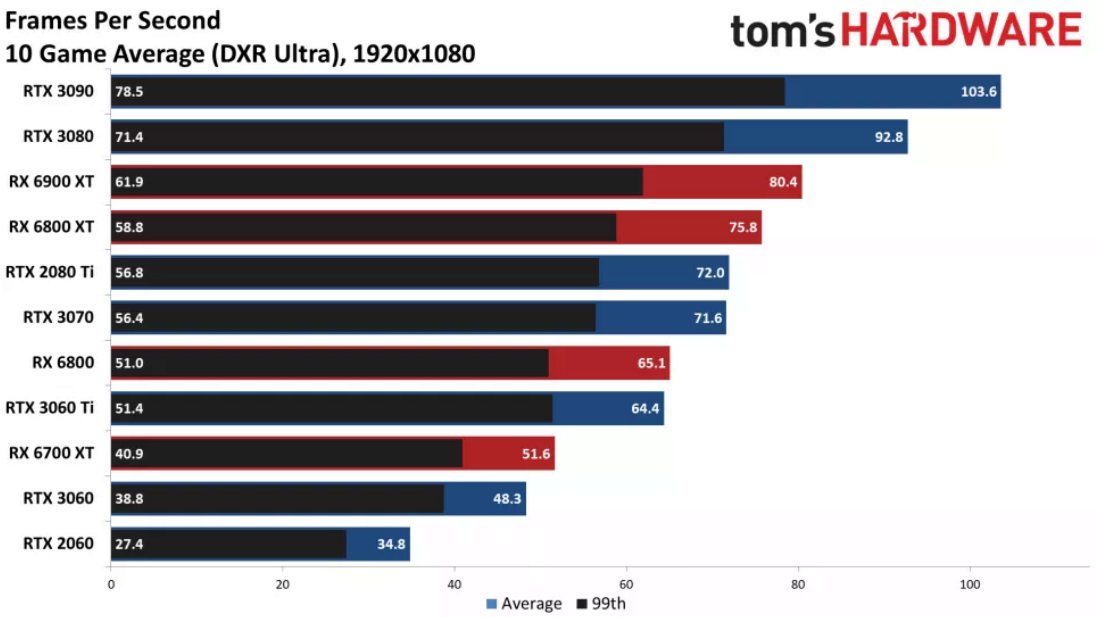


Figura 8. Benchmarks de la mayoría de tarjetas de alta gama, media de 10 juegos, Tom's hardware, 2021

El trazado de rayos tiene un impacto muy importante en el rendimiento, ya que solo las 3 más potentes aguantan el 99th *percentile* sin bajar de 60 fps, el valor normalmente aceptado como bueno por la comunidad. En cuanto al Nvidia vs AMD, podemos observar que los años de ventaja juegan a favor de Nvidia, que obtiene mejores resultados en tarjetas que, al menos en precio de lanzamiento, son parejas. El más claro es el ejemplo de la 6900XT, que AMD lanzó en contraposición a la 3090 y tiene un rendimiento en *ray tracing* un 15% peor que la 3080.

Por último, comentar que todas las tarjetas excepto la RTX 2060, de la anterior generación, obtienen *framerates* relativamente jugables, algo por encima de los 30 fps, suficiente en juegos de poca acción, y eso es una gran noticia para los que nos gusta jugar y apreciar el gran trabajo que hacen los desarrolladores para traer esta tecnología a los consumidores.

Pregunta 8

a) Mientras que un fragmento es un punto dentro de un polígono que se proyecta en el *frame buffer* y contiene la información necesaria para generar un píxel, un píxel es un punto en ese *frame buffer*, no lo que se proyecta, y viene definido por su posición y su color.

b) Es el elemento del *pipeline* gráfico que realiza la tarea de generar los fragmentos que rellenan un triángulo. Se usan los vértices de éste para obtener la información de los fragmentos (posición, color...).

c) La latencia con memoria se oculta utilizando *multithreading*, es decir, cuando pedimos un dato, ejecutamos otras instrucciones mientras esperamos a que la memoria nos lo devuelva.

d) Originalmente, el *vertex shader* y el *fragment shader* eran diferentes ya que tenían usos y lenguajes distintos. Alrededor del 2007 se unificaron, cambiando a una arquitectura que permite un uso más flexible del *hardware* de renderizado de gráficos.

e) El Z-buffer sirve para poder saber qué fragmentos están más cerca y más lejos. Simplemente consiste en un valor de profundidad que podemos usar para comparar y descartar los elementos que quedarán ocultos por otros.

f) Clipping es el proceso que elimina las primitivas que no aparezcan en el área de visión o sean menores a 1 píxel y no hace falta renderizar.

g) Culling es el proceso que elimina las primitivas que quedarán ocultas por otras primitivas y no hace falta renderizar.

h) REYES es un *pipeline* gráfico cuyo acrónimo es Render Everything Your Eyes See.

i) El *aspect ratio* es la proporción entre ancho y altura de una imagen o pantalla.

j) *General Purpose Graphics Processing Unit*, usar una tarjeta gráfica para computación de propósito general, no solo para procesar gráficos.

Referencias

Pregunta 1

The Traditional Graphics Pipeline, Rensselaer Polytechnic Institute, 2009

https://www.cs.rpi.edu/~cutler/classes/advancedgraphics/S09/lectures/15_Graphics_Pipeline.pdf

Traditional Graphics Pipeline, University of Utah Engineering, 2016

https://my.eng.utah.edu/~cs5610/lectures/graphics_HW_vertex.pdf

Pregunta 2

CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops, NVIDIA, 2013

https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/

Pregunta 3

Work distribution methods on GPUs, Christian Lauterback, Qi Mo, Dinesh Manocha, 2010

https://gamma.cs.unc.edu/GPUCOL/GPU-Workqueues.pdf

Pregunta 4

CUDA Runtime API :: CUDA Toolkit Documentation, NVIDIA, 2021

https://docs.nvidia.com/cuda/cuda-runtime-api/index.html

How to Implement Performance Metrics in CUDA C/C++, Mark Harris, NVIDIA, 2012

https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/

Pregunta 5

A performance comparison of CUDA and OpenCL, Kamran Karimi, Neil G. Dickson, Firas Hamze, 2010

https://arxiv.org/ftp/arxiv/papers/1005/1005.2581.pdf

Pregunta 6

Texture Mapping: Concepts, University of North Carolina at Charlotte, 2006

https://webpages.uncc.edu/krs/courses/5010/ged/lectures/texture2.pdf#page=6

The Differential Geometry of Texture Mapping and Shading, Ken Turkowski, Apple, 1998

http://www.realitypixels.com/turk/computergraphics/DifferentialMappings.pdf

Pregunta 7

What is ray tracing? The games, the graphics cards and everything else you need to know, Bill Thomas, Andrew Hayward, Techradar, 2019

https://www.techradar.com/news/ray-tracing

Introduction to Ray Tracing: a Simple Method for Creating 3D Images, Scratchapixel, 2018

https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing

NVIDIA RTX Ray Tracing, NVIDIA, 2021

https://developer.nvidia.com/rtx/raytracing

A detailed study of ray tracing performance: render time and energy cost, Elena Vasiou, Konstantin Shkurko, Ian Mallett, Erik Brunvand, 2018
https://www.researchgate.net/publication/324840216_A_detailed_study_of_ray_tracing_performance_render_time_and_energy_cost

Pregunta 8

Chapter 29. Efficient Occlusion Culling, NVIDIA, 2007
https://developer.nvidia.com/gpugems/gpugems/part-v-performance-and-practicalities/chapter-29-efficient-occlusion-culling
What is GPGPU? Definition and FAQs, Omnisci, 2021
https://www.omnisci.com/technical-glossary/gpgpu

Para todas las preguntas se ha recurrido a las diapositivas de la asignatura además de las fuentes citadas.