

UNIVERSITAT POLITÈCNICA DE CATALUNYA

---

# Exàmen TGA

---

## INFORME

### TARGETES GRÀFIQUES I ACCELERADORS

#### *Autors*

DAVID LATORRE

## Índex

- Pregunta 1.....	3
-Pregunta 2.....	8
-Pregunta 3.....	16
- Pregunta 4.....	22
-Pregunta 5.....	27
-Pregunta 6.....	31
-Pregunta 7.....	36
-Pregunta 8.....	46
-Apartat a.....	46
-Apartat b.....	48
-Apartat c.....	49
-Apartat d.....	50
-Apartat e.....	51
- Apartat f.....	52
- Apartat g.....	52
- Apartat h.....	53
- Apartat i.....	53
- Apartat j.....	53
-Annex.....	54

## Pregunta 1

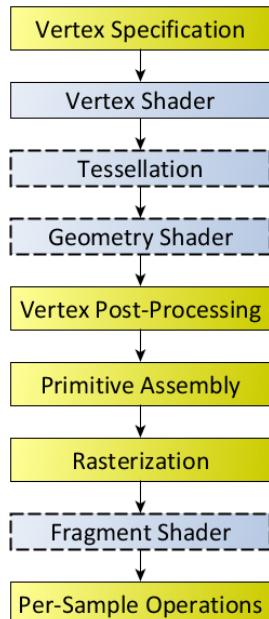
P: Describe el pipeline gràfico tradicional.

Resposta:

Avui dia, degut a que existeixen una gran varietat de motors gràfics, el pipeline gràfic pot variar entre uns o altres motors (pot arribar a variar bastant sobretot en aquells motors que fan les representacions mitjançant ray-tracing), no obstant la majoria comparteixen unes característiques (stages) principals. Aquestes, les podem observar, per exemple, en el motor OpenGL, que podríem dir que conté el pipeline gràfic més tradicional.

Expliquem, doncs, el pipeline gràfic d'OpenGL:

A la imatge següent, podem observar les diverses fases del pipeline (amb blau, aquelles que són programables, és a dir, shaders):



Expliquem cada fase detalladament:

- **Vertex Specification:** es tracta de la fase on l'aplicació configura una llista ordenada de vèrtexs per enviar-los al pipeline. Aquests vèrtexs defineixen els límits d'una primitiva.

Les primitives són formes de dibuix bàsiques, com ara triangles, línies i punts.

Aquesta primera part del pipeline tracta els Vertex Array Objects i els Vertex Buffer Objects. Els Vertex Array Objects defineixen quines dades té cada vèrtex, i els Vertex Buffer Objects emmagatzemen les dades reals del vèrtex.

Les dades d'un vèrtex són una sèrie d'atributs. Tot i que un conjunt d'atributs especifiquen un vèrtex, no hi ha res que digui que una part del conjunt d'atributs d'un vèrtex hagi de ser una posició o normal.

- **Vertex Shader:** El vertex shader realitza el processament bàsic de cada vèrtex de forma individual. Per fer això, reben els atributs de la representació de cada vèrtex, i converteixen aquests vertexs d'entrada, en uns vèrtexs de sortida mitjançant una sèrie de càlculs i transformacions, les quals són les que s'hauran programat.

El vertex shader pot tenir sortides definides per l'usuari, però també hi ha una sortida especial que representa la posició final del vèrtex. En el cas, per exemple, que no s'hagi programat cap vertex shader, llavors aquesta posició final del vèrtexs serà la posició del vèrtex en espai de clipping.

Una limitació del processament de vèrtex és que cada vèrtex d'entrada s'ha de mappear a un vèrtex de sortida específic. Aquest mapping que té una relació 1:1 és útil per no tenir que processar més d'una vegada vertexos diferents, si aquests tenen la mateixa posició inicial, ja que farem servir la caché.

- **Tessel·lació:** Es tracta d'un procés opcional, el qual té dos subetapes: TCS (Tessellation Control Shader), i TES (Tessellation Evaluation Shader). Expliquem aquestes dos.

L'etapa de Tessellation Control Shader (TCS) és la primera i determina la quantitat de tessel·lació que s'ha d'aplicar a una primitiva, a més de garantir la connectivitat entre les primitives tessel·lades adjacents.

L'etapa de Tessellation Evaluation Shader és la última, i aplica interpolacions o altres operacions a les primitives generades després de la tessel·lació.

- **Geometry Shader:** Es tracta d'una etapa que també és opcional. Aquesta etapa processa cada primitiva entrant, retornant zero, una o més primitives de sortida.

Les primitives d'entrada són les primitives de sortida que formen part d'un subconjunt del procés d'assemblatge de primitives. Per tant, si estem treballant amb tires de triangles, al geometry shader li entraran series de triangles.

Com que el geometry shader és capaç de fer output de més d'una primitiva donada una primitiva d'entrada, és capaç per tant de tessel·lar primitives.

Cal destacar també que el geometry shader es pot combinar amb el vertex shader, i fer una part de la feina d'aquest, i el geometry shader també pot convertir les primitives d'un tipus a un altre, per exemple de punts a triangles.

- **Primitive assembly:** Es tracta del procés de recopilar una sèrie de dades de vèrtexs de sortida de les etapes anteriors i compondre-les en una seqüència de primitives. El tipus de primitiva amb el qual l'usuari les ha representat determina com funciona aquest procés.

La sortida d'aquest procés és una seqüència ordenada de primitives simples (línies, punts o triangles). Si l'entrada és una primitiva de strips triangulars que conté 12 vèrtexs, per exemple, la sortida d'aquest procés serà de 10 triangles.

- **Rasteritzation:** Les primitives que arriben a aquesta etapa (dic arriben, ja que prèviament s'ha fet el clipping), es rasteritzen. El resultat de rasteritzar una primitiva és una seqüència de fragments.

Un fragment és un conjunt de dades que seran utilitzades per calcular el valor final d'un pixel. Un fragment inclou informació com la posició en window-screen, dades arbitràries del output del vertex o geometry shader...

Els fragments son calculats fent interpolacions entre els vèrtexs que hi ha dins del fragment, és a dir els vèrtexs que “hi ha” dins d'un pixel.

- **Fragment shader:** Es un procés opcional. Les dades de cada fragment que surten del rasteritzation process, són processades pel fragment shader.

La sortida del fragment shader és una llista de colors, de valors de profunditat (depth values) i valors stencil.

En el cas que no es programi el fragment shader, llavors els valors depth i stencil seran els computats de forma default, i el colors dels fragments quedaran indefinits.

- **Per-Sample Operations:** Es tracta d'una sèrie de fases.

La primera fase són els culling tests. Si un test d'aquest tipus està actiu i falla per al fragment que s'està analitzant, llavors el pixel corresponent no s'actualitza amb aquest fragment. Alguns d'aquests tests són:

- **Pixel ownership test:** Falla si el fragment del pixel no forma part de la finestra d'OpenGL.
- **Scissor test:** Falla si el fragment del pixel està fora del rectangle de la finestra.

- **Altres.**

Després hi ha l'etapa de color blending. Per a cada valor de color de fragment, hi ha una operació de barreja específica entre aquest i el color que ja hi ha al framebuffer en aquesta ubicació.

Finalment les dades de cada fragment són escrites al framebuffer. Pot haver-hi certes operacions de masking que facin que no s'escriguin alguns valors de fragments.

## Pregunta 2

P: Dada la siguiente rutina escrita en C:

```
void Examen22(float mA[N][M], float mB[N][M], float vC[N]) {
    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] + mA[i][0]*mB[0][j];
}
```

- a: En la primera versión cada thread se va a ocupar de 1 columna de la matriz resultado.
- b: En la segunda versión cada thread se va a ocupar de 1 fila de la matriz resultado.
- c: En la última versión cada thread se va a ocupar de 1 elemento de la matriz resultado.

Escribid los kernels CUDA para cada versión, así como la invocación correspondiente. Tened en cuenta que como máximo podéis utilizar 1024 threads por bloque y que las variables N y M pueden tener cualquier valor (p.e. N = 1237, M = 2311, suponed que N, M > 1024).

Resposta:

Per a realitzar aquests tres kernels que se'ns demana, construirem un arxiu .cu el qual té tres funcions. Cada una d'aquestes funcions cridarà el seu respectiu kernel, i per tant, tant abans com després de la crida haurà de fer una sèrie d'operacions (transferència memòria CPU-GPU-CPU, calcular dimgrid i dimbloc, etc...). Cada una d'aquestes funcions rebrà els mateixos paràmetres que la funció Examen22(), però per termes d'eficiència, farem que tant les matrius com el vector vC es passin com a punters.

Un cop tinguem les tres funcions i kernels implementats, executarem per als tres kernels (amb matrius i vector del mateix tamany pels tres kernels) el profiler Nsight Compute (similar a nvprof) per a veure quin dels tres dona més FLOPS. Segons el que hem observat a les classes de teoria, degut als patrons d'accés a

memòria dels warps i la capacitat de paral·lelització, el kernel amb més rendiment hauria de ser el element a element, mentre que el que tingui menys haurà de ser el que cada thread calcula una fila.

### Implementació primera versió:

Per a aquesta primera versió, se'ns demana que cada thread calculi 1 columna de mA.

Si volem fer això, però, de la màxima forma paral·lelitzable, de forma que independentment de l'ordre els threads calculin la seva columna, tenim un problema, i és que es pot donar el cas que el primer thread de tots (el que té l'identificador més petit) encara no hagi calculat tots els valors de la seva columna (la primera de totes) quan un altre thread vulgui accedir a mA[x][0], i per tant aquest llegeixi un valor incorrecte.

Per arreglar aquest error, simplement podem calcular secuencialment la primera columna de mA.

El primer que farem, a la funció, serà calcular la dimensió del bloc i del grid:

```
int blocksize = 1024;
dim3 dimGrid, dimBlock;

dimBlock.x = blocksize;
dimBlock.y = 1;
dimBlock.z = 1;

dimGrid.x = (m + blocksize - 1) / blocksize;
dimGrid.y = 1;
dimGrid.z = 1;
```

Com podem observar a la imatge, estem fent servir el màxim blocksize (1024 threads / block). Aquí només ens interessen les components x de dimBlock i dimGrid ja que entre threads només diferenciem columnes. A l'hora de fer el càlcul de dimGrid, posem +blocksize - 1 per tenir en compte l'últim bloc de tots.

A continuació, donat que els punters que hem rebut com a paràmetres no formen part de la memòria pinned, no ens podem beneficiar d'aquesta al fer la

transferència a la GPU (tècnicament si que podríem, però això requeriria fer una alocació pinned a la RAM i fer una duplicació dels punters a la RAM amb pinned). Fem per tant, uns simples cudaMemcpy:

```
cudaMemcpy(mADevice, mA, n * m * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(mBDevice, mB, n * m * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(vCDevice, vC, n * sizeof(float), cudaMemcpyHostToDevice);
```

I, finalment, a la funció, executem el kernel, copiem a CPU la matriu calculada i alliberem els punters de la GPU.

```
PrimeraVersioKernel << ~dimGrid, dimBlock >> > (mADevice, mBDevice, vCDevice, n, m);

cudaMemcpy(mA, mADevice, n * m * sizeof(float), cudaMemcpyDeviceToHost);
cudaFree(mADevice);
cudaFree(mBDevice);
cudaFree(vCDevice);
```

Pel que fa al kernel:

```
__global__ void PrimeraVersioKernel(float* mA, float* mB, float* vC, int n, int m)
{
    int columna = blockIdx.x * blockDim.x + threadIdx.x;
    if (columna < m)
    {
        for (int i = 0; i < n; i++)
        {
            mA[i * m + columna] = mA[i * m + columna] * vC[i] + mB[i * m + columna] + mA[i * m] * mB[columna];
        }
    }
}
```

Per cada thread, calculem la seva columna, i llavors iterem per totes les files de la columna per calcular tota la columna.

Si ens hi fixem, podem observar que com que ja hem calculat la primera columna anteriorment, podríem fer un “if” de forma que el thread que li toca la columna o no faci res. No obstant crec que això empitjoraria el rendiment de l’algorisme ja que llavors tots els threads haurien d’executar la instrucció “if”.

### Implementació segona versió:

Per a aquesta versió, no cal que calculem seqüencialment primer en CPU la primera columna de totes, doncs cada thread calcularà seqüencialment els valors d'una fila, de forma que sempre calcularem el primer element d'una fila, és a dir el que pertany a la primera columna de totes.

Pel que fa al codi de la funció d'aquesta versió, aquest és molt semblant a la de l'anterior. No obstant, aquí, com acabem de dir, no cal fer el càlcul de la primera columna en CPU, i, com que diferenciem files entre els threads, utilitzarem les components 'y' de dimBlock i DimGrid:

```
void SegonaVersio(float* mA, float* mB, float* vC, int n, int m)
{
    int blocksize = 1024;
    dim3 dimGrid, dimBlock;

    dimBlock.x = 1;
    dimBlock.y = blocksize;
    global__ void SegonaVersioKernel(float* mA, float* mB, float* vC, int n, int m)
    {
        int fila = blockIdx.y * blockDim.y + threadIdx.y;
        if (fila < n)
        {
            for (int j = 0; j < m; j++)
            {
                mA[fila * m + j] = mA[fila * m + j] * vC[fila] + mB[fila * m + j] + mA[fila * m] * mB[j];
            }
        }
    }

    float* vCDevice;
    cudaMalloc((float**)&mADevice, n * m * sizeof(float));
    cudaMalloc((float**)&mBDevice, n * m * sizeof(float));
    cudaMalloc((float**)&vCDevice, n * sizeof(float));

    cudaMemcpy(mADevice, mA, n * m * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(mBDevice, mB, n * m * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(vCDevice, vC, n * sizeof(float), cudaMemcpyHostToDevice);

    SegonaVersioKernel << ~dimGrid, dimBlock >> > (mADevice, mBDevice, vCDevice, n, m);

    cudaMemcpy(mA, mADevice, n * m * sizeof(float), cudaMemcpyDeviceToHost);
    cudaFree(mADevice);
    cudaFree(mBDevice);
    cudaFree(vCDevice);
}
```

Pel que fa al kernel d'aquesta versió, aquest és similar també a l'anterior, però en aquest cas per a cada thread calcularem la fila a la que fa referència, i iterarem per totes les columnes de la fila:

### Implementació tercera versió:

Per a aquesta versió, tornem a tenir el mateix problema que en la primera versió, ja que necessitem primer precalcular els valors de la primera columna. No obstant, com que (segons el que ens diu l'enunciat) podem assumir que  $N \geq M > 1024$ , precalcular primer en CPU secuencialment la primera columna de totes de la matriu mA no afectarà molt al rendiment total de l'algorisme.

En aquest cas, com que cada thread calcula un sol element, entre threads sí que distingim tant columnes com files, de forma que cada bloc tindrà dimensions de 32 threads d'amplada i 32 threads de llargada ( $32 \times 32 = 1024$ ). La funció d'aquesta funció, per tant, queda d'aquesta manera:

```
void TerceraVersio(float* mA, float* mB, float* vC, int n, int m)
{
    int blocksize = 32;
    dim3 dimGrid, dimBlock;

    dimBlock.x = blocksize;
    dimBlock.y = blocksize;
    dimBlock.z = 1;

    dimGrid.x = (m + blocksize - 1) / blocksize;
    dimGrid.y = (n + blocksize - 1) / blocksize;
    dimGrid.z = 1;

    //Primera columna CPU
    for (int i = 0; i < n; i++)
    {
        mA[i * m] = mA[i * m] * vC[i] + mB[i * m] + mA[i * m] * mB[0];
    }

    float* mADevice;
    float* mBDevice;
    float* vCDevice;
    cudaMalloc((float**)&mADevice, n * m * sizeof(float));
    cudaMalloc((float**)&mBDevice, n * m * sizeof(float));
    cudaMalloc((float**)&vCDevice, n * sizeof(float));

    cudaMemcpy(mADevice, mA, n * m * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(mBDevice, mB, n * m * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(vCDevice, vC, n * sizeof(float), cudaMemcpyHostToDevice);

    TerceraVersioKernel << ~dimGrid, dimBlock >> (mADevice, mBDevice, vCDevice, n, m);

    cudaMemcpy(mA, mADevice, n * m * sizeof(float), cudaMemcpyDeviceToHost);
    cudaFree(mADevice);
    cudaFree(mBDevice);
    cudaFree(vCDevice);
}
```

I, pel que fa al kernel, per a cada thread calcularem la fila i la columna a la que fan referència:

```
__global__ void TerceraVersioKernel(float* mA, float* mB, float* vC, int n, int m)
{
    int fila = blockIdx.y * blockDim.y + threadIdx.y;
    int columna = blockIdx.x * blockDim.x + threadIdx.x;

    if (fila < n && columna < m)
    {
        mA[fila * m + columna] = mA[fila * m + columna] * vC[fila] + mB[fila * m + columna] + mA[fila * m] + mB[columna];
    }
}
```

Ara que ja tenim les tres versions implementades, ja les podem provar. Per això, farem que, per exemple N=20.000, i M=20.000, i crearem les matrius i vectors amb valors aleatoris. Amb aquests valors de N i M estarem calculant una matriu de un total de 1.6GB. Farem una execució per a cada un dels kernels, i amb Nsight anem a comprovar quin rendiment hem obtingut. Vegem, però, primer, com queda el main:

```

int main()
{
    cout << "Tria la versio del kernel" << endl;

    int numeroKernel;
    cin >> numeroKernel;

    int n = 20000;
    int m = 20000;

    float* mA = (float*)malloc(n * m * sizeof(float));
    float* mB = (float*)malloc(n * m * sizeof(float));
    float* vC = (float*)malloc(n * sizeof(float));

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            mA[i * m + j] = rand() % 100;
            mB[i * m + j] = rand() % 100;
        }
    }

    for (int i = 0; i < n; i++)
    {
        vC[i] = rand() % 100;
    }

    if (numeroKernel == 1)
    {
        PrimeraVersio(mA, mB, vC, n, m);
    }
    else if (numeroKernel == 2)
    {
        SegonaVersio(mA, mB, vC, n, m);
    }
    else if (numeroKernel == 3)
    {
        TerceraVersio(mA, mB, vC, n, m);
    }

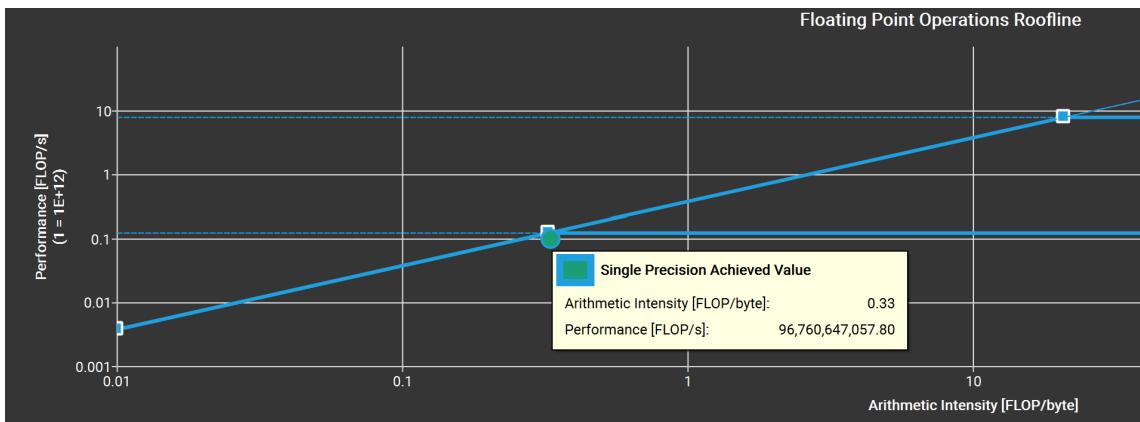
    cout << "Fet" << endl;
}

```

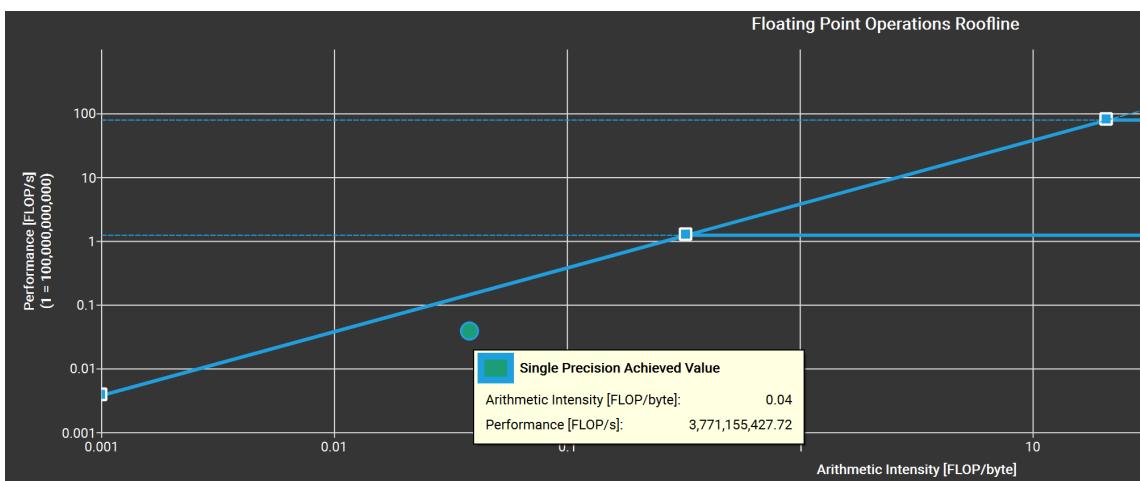
Com podem observar, el que fem al main és executar la versió del kernel que volem, i inicialitzar les matrius i vector, amb valors aleatoris.

Ara, vegem quins rendiments hem obtinguts (en FLOPS):

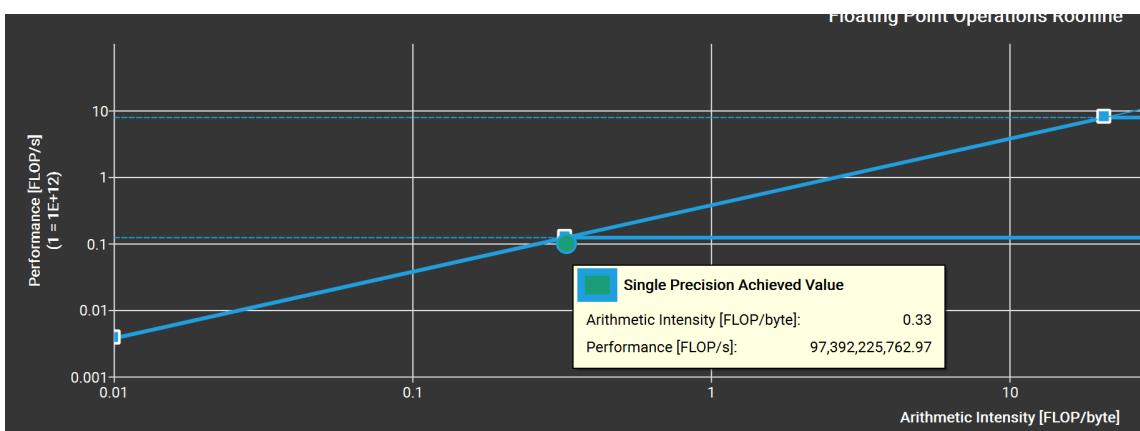
## Versió Kernel 1:



## Versió Kernel 2:



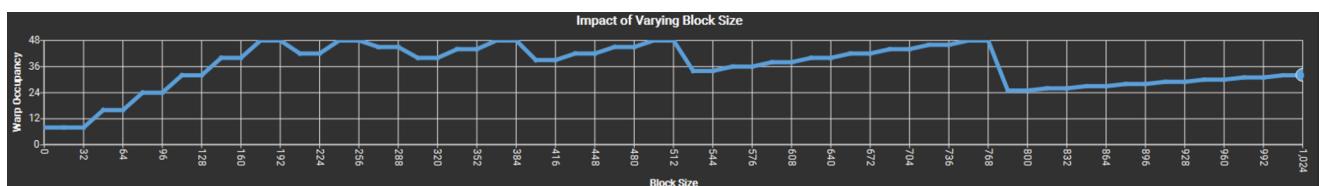
## Versió Kernel 3:



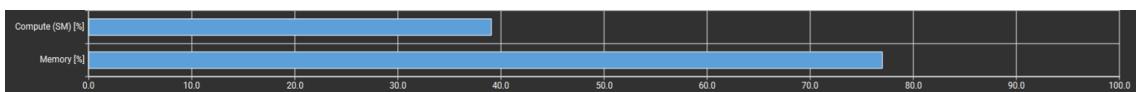
Amb aquestes imatges, sembla que el que hem dit anteriorment sembla complir-se. No obstant, els algorismes que hem implementat es tracta d'algorismes molt poc eficients, podriem millorar el rendiment d'aquests encara molt més. Aquests resultats s'han obtingut amb una GPU [rtx 3070 Max-Q](#) (laptop), la qual pot arribar a un rendiment de 13.21 teraflops, de forma que encara estem molt lluny.

Hi ha dos raons principals per les quals obtenim un rendiment tant pobre, i que també fan que la versió kernel 3 no sigui molt més superior a les dos anteriors:

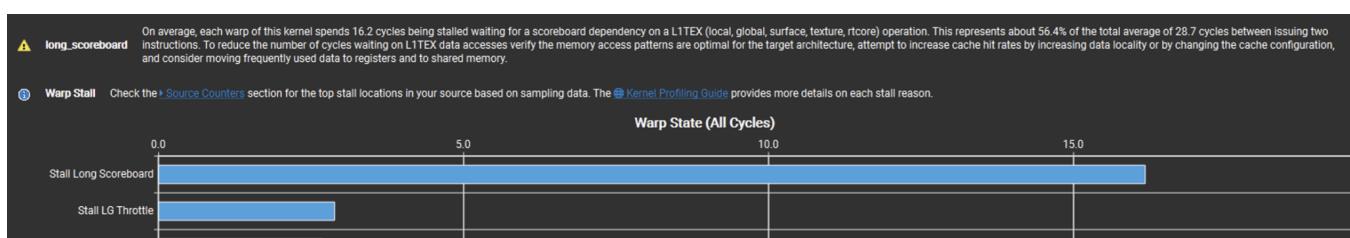
- D'una banda, a l'haver executat els kernels amb un blocksize de 1024 threads / block, tenim una warp occupancy més limitada que, si per exemple els executessim amb 256 threads / block. Això, ho podem observar a la següent imatge:



- D'altra banda en tots 3 casos no ens estem acostant gents al maxim throughput tant de computació ni de memòria de la GPU. Exemple per a kernel versió 3:



Això és degut a que, a l'estar tot el rato fent consultes a la memòria global, molts warps s'hagin de congelar i esperar-se molts cicles perquè el pipeline de memòria estigui buit (això ho podem solucionar amb memòria shared):



## Pregunta 3

P: A la vista de cómo han ido evolucionando las tarjetas de NVIDIA. Selecciona la tarjeta que consideres más relevante de toda su historia. Justifica porqué has elegido esa tarjeta y describe sus características más relevantes.

Resposta:

Nvidia sempre ha destacat per la seva gran i constant evolució any rere any, ja que sempre acostuma a llançar una nova arquitectura de targetes gràfiques cada any i mig o així, i amb cada nova arquitectura no només les targetes són cada cop més ràpides, sinó que acostumen a incorporar noves característiques.

Dit això, per tant, és d'esperar que sempre considerem que la targeta més rellevant sigui formi part de l'última arquitectura que han tret, i dins d'aquesta arquitectura, la més potent.

No obstant, no sempre les GPU més potents són les que més marquen a la història. Hi ha que marquen degut a les noves característiques que incorporen, com per exemple la [NVIDIA Tesla P100 SXM<sub>2</sub>](#) (arquitectura Pascal), que al 2016, es va convertir amb la primera GPU de Nvidia en incorporar memòria tipus HBM<sub>2</sub>; la [NVIDIA GeForce 256 DDR](#), que va ser una de les primeres en incorporar memòria DDR, i a la vegada és reconeguda com la primera GPU com a tal; o la primera “TITAN” de la història, la [NVIDIA GeForce GTX TITAN](#) (Kepler, 2013).

Però, personalment, se'ns dubte la millor innovació que ha fet mai Nvidia s'ha fet amb les GPUs [Nvidia TITAN V](#) i [Nvidia Quadro GV100](#), ja que és quan es va poder parlar per primera vegada seriament d'acceleració ràpida en intel·ligència artificial i ray tracing. Tècnicament aquestes dues ja formen completament tota la gamma de l'arquitectura Volta, cosa que és super estrany perquè Nvidia sempre acostuma a llançar moltes més GPU per arquitectura, però això ja porta a pensar de lo especial que és aquesta arquitectura. A més, aquestes dues GPU són quasi bé el mateix, ja que porten xips quasi bé idèntics, no obstant una GPU

(la TITAN) està més encarada al nivell entusiasta, mentre que la Quadro està més encarada al nivell professional (vaja, per a servidors).

Durant la conferència annual d’Nvidia de l’any 2017 ([GTC 2017](#)), el CEO d’Nvidia, Jen-Hsun Huang, va anunciar els productes de la gamma Volta. Es va tractar d’un moviment molt important d’Nvidia, amb l’objectiu de liderar el mercat del Deep Learning.

L’arquitectura Volta, és una arquitectura que va portar molts anys desenvolupant-se. De fet, Nvidia ja la va anunciar l’any 2013 (quan tot just s’acabava de llançar l’arquitectura Kepler), no obstant va trigar molt més en llançar-se finalment. El que va passar, és que el que pretenia primerament Nvidia, que era llançar l’arquitectura Maxwell i posteriorment Volta, finalment és va convertir en Maxwell, Pascal, i Volta, fent que hi hagués aquesta arquitectura de transició (Pascal) entre Maxwell i Volta.

Amb Volta, Nvidia va dedicar-se a llançar una arquitectura que no estigués gens centrada en les GPU de consum (les que es comercialitzen per a ús personal, vaja), sinó centrada amb el focus a tasques professionals, HPC i deep learning.

Anem a observar les característiques de la Nvidia GV100 (el xip), junt amb els dos xips predecessors del GV100 (el [GP100](#) (Pascal) i el [GK110](#) (Kepler) respectivament):

	GV100	GP100	GK110
CUDA Cores	5376	3840	2880
Tensor Cores	672	N/A	N/A
SMs	84	60	15
CUDA Cores/SM	64	64	192
Tensor Cores/SM	8	N/A	N/A
Texture Units	336	240	240
Memory	HBM2	HBM2	GDDR5
Memory Bus Width	4096-bit	4096-bit	384-bit
Shared Memory	128KB, Configurable	24KB L1, 64KB Shared	48KB
L2 Cache	6MB	4MB	1.5MB
Half Precision	2:1 (Vec2)	2:1 (Vec2)	1:1
Double Precision	1:2	1:2	1:3
Die Size	<b>815mm<sup>2</sup></b>	610mm <sup>2</sup>	552mm <sup>2</sup>
Transistor Count	21.1B	15.3B	7.1B
TDP	300W	300W	235W
Manufacturing Process	TSMC 12nm FFN	TSMC 16nm FinFET	TSMC 28nm
Architecture	Volta	Pascal	Kepler

Sí que és veritat que el xip GP100 pel fet d'incorporar memòria HBM<sub>2</sub> per primera vegada ja es pot considerar un xip molt rellevant, però el GV100 incorpora una cosa nova, anomenada **Tensor Cores**. A més, a diferència de Pascal, el xip pasar a ser de 12nm.

De fet, Gv100 a diferència de les arquitectures predecessors és també different pel que fa a: l'execució dels threads, thread scheduling, core layout, controladors de memòria, ISA, etc. Comencem a endinsar-nos en les especificacions d'aquest xip:

Per començar, Gv100 al 2017 es va convertir se'ns dubte en la GPU amb més transistors per molta diferència, amb 21.1 bilions de transistors. El die size tampoc es quedava curt, amb 815mm<sup>2</sup> és un die size 33% superior al xip GP100.

Aquest gran die size es tracta d'una dada molt important a considerar. No sols perquè com més gran sigui l'àrea d'un die més rendiment tindrem (que també), sinó perquè aquí podem veure quines intencions tenia realment Nvidia, que no era precisament vendre xips de forma massiva. Dic això perquè, a l'any 2017, fabricar un die tant gran es tractava (i segueix essent així) d'una tasca bastant conflictiva, pel fet que la qualitat de fabricació dels xips (la mesura chip yield) acaba essent bastant pobre, fent que Nvidia hagués d'utilitzar grans quantitats de silici per a un nombre molt petit de GPUs. Això, em porta a pensar que en aquell moment Nvidia preferia estirar al màxim els límits de la seva tecnologia i vendre-la a petita escala (de forma molt cara) abans que comercialitzar-la a gran escala. I és potser gràcies a això que Nvidia a arribat a ser qui és avui a l'any 2022.

Observem com és el GV100 per dins:



Es tracta d'un xip que conté 84 SMs, i dins de cada un dels SMs hi ha 64 CUDA cores. Però no només això, sinó que a dins també ens trobem amb uns cores anomenats Tensor Cores. Mirem un SM per dins:



Els tensor cores podem pensar que són com CUDA cores però molt més rígids i menys flexibles (tot i que programables) adaptats específicament a operacions de deep learning. De fet es tracta de cores que incorporen ALUs que l'únic que fan es multiplicacions de matrius 4x4:

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}_{\text{FP16 or FP32}} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}_{\text{FP16}} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}_{\text{FP16 or FP32}}$$

Això, primerament pot semblar inútil. Per què volem cores que l'únic que fan són multiplicacions de matrius 4x4? Doncs bé, perquè tot i que no ens ho creiem, aquesta operació és una de les que més s'utilitza, i de forma massiva, sobretot en camps com en machine learning.

Això, fa que per tant poguem obtenir molts més FLOPS / segon en general. Per exemple, per al xip GV100, si cada tensor core es capaç de rendir 128 FLOPS, llavors, com que cada SM té 8 tensor cores, això ja són 1024 FLOPS més per SM. Per tant, el rendiment, pot arribar a ser, en el cas ideal, 4 vegades superior al xip predecessor (GP100, Pascal).

De fet, la raó per la qual hi ha tants transistors en aquest xip és per els tensor cores. Degut als tensor cores, tenim moltes més ALUs per SM, i com a conseqüència, tenim molts més transistors per SM.

Pel que fa a altres millores respecte el xip GP100, cal destacar que hi ha 40KB més de cache per SM, i, tot i que GP100 també incorpora memòria HBM<sub>2</sub>, GV100 té un guany del 50% en ample de banda efectiu de la memòria tot i que les seves freqüències de memòria només hagin pujat un 25%. Aquest guany principalment es deu a una millor eficiència de la memòria.

Finalment, també cal destacar que, ara ja parlant de la GPU Quadro v100, aquesta va començar a incorporar la nova generació de NVLink, NVLink 2 (la nova tecnologia multi-GPU de Nvidia), i que gràcies a les característiques del xip

que acabem d'explicar junt amb una VRAM de 16GB, és una GPU capaç de realitzar operacions en coma flotant de meitat de presició a fins a 15TFLOPS / segon (una xifra bona fins i tot avui dia) així com arribar a un ample de banda de 900GB/segon.

Ara, ja per acabar, em sembla important explicar també que el març d'aquest any (2022), Nvidia ha llançat la nova arquitectura per a les seves GPU de tipus Tesla (és a dir les GPU d'alt rendiment de NVidia (per a data-centers)), la qual s'anomena Hopper, i s'ha llançat la [H100](#) (més específicament la versió SXM), la qual tot apunta que també serà una GPU que marcarà la història de Nvidia, ja que és la primera en incorporar memòria HBM3, i és capaç d'aconseguir un rendiment en coma flotant de meitat de presició de 120.3 TFLOPS / segon (lol).

## Pregunta 4

**P:** Si queremos utilizar GPUs para cálculo de propósito general (GPGPU) puedes escoger entre CUDA, OpenCL y OpenACC. Describe las ventajas e inconvenientes de cada alternativa. Además, se pueden combinar OpenACC con CUDA (o OpenCL), ¿qué posibilidades ofrece esta opción?

**Resposta:**

Avui dia, en computació d'alt rendiment, tant en l'aprenentatge automàtic com en un conjunt creixent d'altres àrees, les GPUs, i en general, els acceleradors, s'estan convertint en una cosa obligatòria.

La capacitat de realitzar una paral·lelització eficient i de baix nivell ens permet als programadors augmentar molt el rendiment dels nostres codis. Hem arribat a un punt en que molts dels superordinadors del TOP500 utilitzen targetes gràfiques com a acceleradors. És per això que, avui dia, les GPUs modernes accepten una àmplia gamma de models de programació paral·lela, com CUDA, OpenACC o OpenCL.

No obstant, l'elecció entre aquests models pot ser molt difícil. El truc es tracta de triar aquell que ens proporcioni el màxim rendiment i eficiència, però també productivitat (esforç de programació).

Per comparar aquests tres models (CUDA, OpenACC i OpenCL) anem primer a descriure de que tracta cadascun dels tres:

D'una banda, CUDA (Compute Unified Device Architecture), és un paradigma de programació paral·lel que va ser llançat l'any 2007 per NVIDIA. Actualment, hi ha una comunitat de desenvolupadors força gran relacionada amb CUDA.

El paradigma de programació CUDA és una combinació d'execucions en sèrie i en paral·lel i conté una funció C especial anomenada kernel, que és en termes senzills és un codi C que s'executa en una targeta gràfica en un nombre fix de threads simultàniament.

D'altre banda, OpenCL és un model que va ser llançat per Apple i el grup Khronos, per tal de proporcionar una alternativa a CUDA que no es limités només a les targetes gràfiques Nvidia. OpenCL es tracta d'un llenguatge força portable, el qual s'utilitza per dissenyar programes o aplicacions que siguin prou generals com per executar-se en arquitectures considerablement diferents, alhora que són prou adaptables per permetre que cada plataforma de maquinari aconsegueixi un alt rendiment.

I, pel que fa a OpenACC, aquest es bastant similar en tant al que acabem de dir per a OpenCL. Amb OpenACC, es pot treballar tant amb C, C++ com en Fortran per identificar les àrees que s'han d'accelerar utilitzant directrius del compilador i funcions addicionals.

CUDA permet al programador manipular directament els recursos de la GPU dins del kernel, com els threads, blocs, el grid, l'assignació de memòria (shared, constant, per exemple)... CUDA també destaca per ser un llenguatge bastant més fàcil de programar respecte els altres.

Contràri a l'enfocament de paral·lelisme que té CUDA, ens tobariem amb OpenACC, que representa un model declaratiu de programació paral·lela utilitzant pragmes del compilador. Amb OpenACC, la tasca de generar codi paral·lel per a la GPU recau en compiladors, que no sempre són capaços de fer front a aquesta tasca de manera eficaç. Per aquest motiu, els programadors sovint es veuran obligats a escriure codi que repeteixi conceptualment el codi escrit en CUDA o OpenCL per tal d'obtenir un programa OpenACC prou eficaç.

Una altre comparació que cal fer també es la implementació d'aquests models per als diversos venedors de hardware. Mentre que per a executar codi CUDA només podem fer servir GPUs d'Nvidia, per a OpenCL i OpenACC hi ha molts altres venedors que podem considerar, com per exemple:

- CPUs de Intel i AMD.
- GPUs de AMD.
- GPUs de Nvidia descatalogades que no tenen suport ni de CUDA.

- El sistema operatiu MacOS (que per a les noves versions no dona suport a CUDA ja que directament no dona suport a res relacionat amb Nvidia).
- etc.

Una altre característica que cal considerar (que té els seus pros i contres) és que OpenCL i OpenACC, a diferència de CUDA, són Open Source.

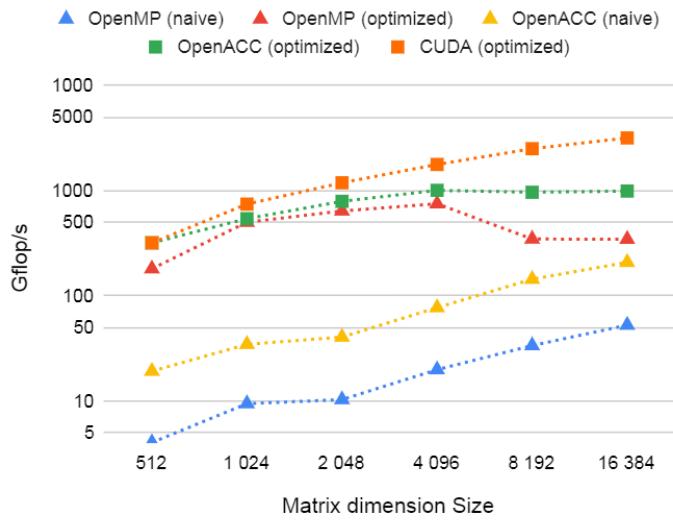
Pel que fa al suport de llibreries externes, aquí cal destacar que CUDA té suport de llibreries molt completes i bones, com CuBLAS, cuSPARSE, NPP, Thrust, etc.

Dit tot això, ara caldria, donat un sistema específic, realitzar un benchmark sobre aquests 3 tipus de llenguatges per veure realment quin és millor segons la tasca que vulguem realitzar.

Hi ha molts estudis que realitzen diversos tipus de benchmarks. Dos bastant útils, els qual treuen conclusions prou clares, són: Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100GPU (Mikhail Khalilov and Alexey Timoveev 2021 J. Phys.: Conf. Ser. 1740 012056); i Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption (Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kolodziej, Christoph Kessler), els qual implementen una sèrie d'algorismes per cada un dels llenguatges i més tard els comparen, executant-los en una GPU (per exemple una Nvidia Tesla v100 per al primer article)

De les conclusions d'aquests estudis, en podem concloure que mentre que, un cop haguem fet una implementació de l'algorisme que es vulgui de forma eficient per cada llenguatge, l'ample de banda de la memòria tant per a OpenACC com per a OpenCL com per a CUDA poden arribar a ser molt alts (i bastant similars), a mesura que la complexitat del codi dels kernels augmenta, es comença a observar un gran gap a favor de CUDA.

A continuació, podem observar una comparació en GFLOPS/segon d'un algorisme de multiplicació de matrius executat en una Nvidia Tesla V100 per a diversos llenguatges:



També cal destacar que, a mesura que augmenta el tamany de les dades d'entrada, OpenACC acostuma a mostrar una certa degradació.

Finalment, també és molt important destacar que el temps que es perd i l'esforç en programar un kernel per a aquests llenguatges és un punt també molt important a tenir en compte. Ja que mentre que, per exemple, per a OpenCL, si dissenyem un codi molt eficient podem arribar a obtenir resultats (pel que fa a temps d'execució) molt millors que si els comparem fins i tot amb CUDA, al cap i a la fi CUDA acaba essent el llenguatge que requereix menys esforç de programació. De mitjana (ho diuen els estudis), es preveu que per dissenyar un codi OpenCL de forma eficient es triga el doble de temps que dissenyant-lo amb CUDA.

Pel que fa a la qüestió de si es pot combinar CUDA amb OpenACC, o CUDA amb OpenCL, OpenACC té un cert nivell de interoperabilitat amb CUDA, així que sí, es pot combinar amb CUDA, mentre que no hi ha manera que es pugui combinar OpenCL amb CUDA.

Existeix un [repositori](#) de Github bastant bo que demostra aquesta interoperabilitat entre OpenACC i CUDA.

Aquesta interoperabilitat acostuma a realitzar-se mitjançant un dels tres mètodes següents: les construccions “host\_data”, la cláusula “deviceptr”, i la funció “acc\_map\_data()”.

Les principals avantatges de combinar CUDA i OpenACC és que podem tenir el millor d'aquests dos mons. Per exemple, imaginem que tenim un programa escrit en OpenACC, però ara volem multiplicar dos matrius. Llavors no cal que gastem cap esforç en construir un algorisme eficient que multipliqui dos matrius, ja que podem fer servir la llibreria de CUDA anomenada cuBLAS library.

Gràcies a aquesta interoperabilitat, per tant, podem tant accelerar aplicacions amb OpenACC fent servir llibreries optimitzades de CUDA, com també estendre aplicacions CUDA accelerant aquelles rutines no accelerades mitjançant OpenACC.

## Pregunta 5

P: ¿Cómo has de estructurar una aplicación CUDA para trabajar con streams? Y ¿para trabajar con varias GPUs?

Resposta:

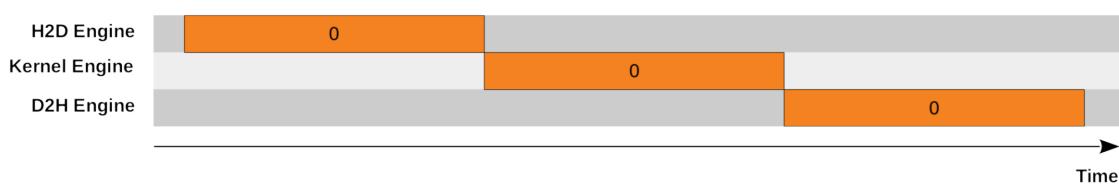
La majoria de vegades, quan programem aplicacions CUDA, ho fem d'una forma serial, és a dir, amb la intenció que cada comanda CUDA s'executi una després de l'altre. No obstant, això no ho tenim que fer sempre així.

Un simple programa CUDA consisteix en tres principals fases: copia de la memòria del host al device, execució del kernel, i còpia de la memòria del device al host. Llavors, tenint en compte això, i que:

- La GPU mai s'utilitzarà al màxim.
- El kernel que s'executi pot ser dividit en N kernels més petits, i l'execució de cada un d'aquests trigaria (en el cas ideal)  $1/N$  del temps que triga el kernel original.
- El temps de transferències de memòria entre host-device i device-host és més o menys igual (en el cas ideal).
- Cada motor CUDA executa sempre les comandes i kernels en ordre.

Llavors, la programació d'un programa CUDA, la podem fer mitjançant un model concurrent, el qual, a diferència del serial, encara podríem augmentar més la paral·lelització i per tant reduir el temps d'execució, tal com es mostra a la imatge següent:

**Serial Model**



**Concurrent Model**



A la imatge, per al model concurrent, a diferència que el serial, el que fem és la còpia de memòria host-device, la execució del kernel, i la còpia de memòria del device-host de forma asíncrona.

Per fer això, s'ha dividit la memòria en 4 parts (4 per a l'exemple de la imatge).

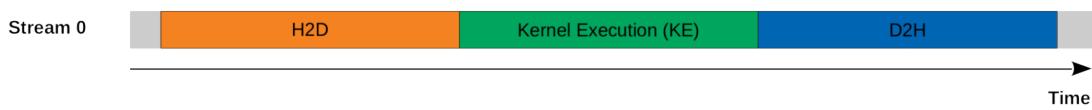
Un cop s'ha acabat de copiar la primera part de host a device, s'executa el primer kernel que processarà el primer tros de memòria. Mentre això passa, podem anar enviant ja el segon tros de memòria del host al device, i un cop acabi la execució del primer kernel, executar el segon. I fer això per a tots els trossos de memòria amb el que hagim dividit la memòria.

D'aquesta manera, per al cas ideal, en l'exemple de la imatge trigariem la meitat del temps.

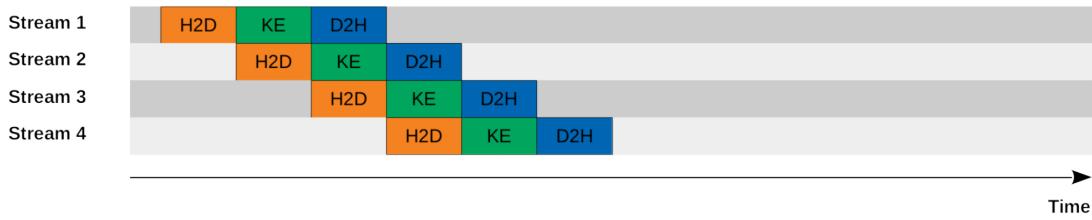
Ara bé, com programem el programa en CUDA per fer aquest model concurrent? Això ho podem fer gràcies als streams. La definició d'un stream és una seqüència de comandes CUDA que s'executen en ordre. Diferents streams, per tant, es podran executar sense cap ordre definit respecte altres streams.

D'aquesta manera, per tant, la imatge anterior es pot representar amb streams de la següent manera:

#### **Serial Model**



#### **Concurrent Model**



De fet, quan programem el programa serial, ja estem programant amb streams, el que passa que no ens adonem. No ens adonem que estem utilitzant streams perquè utilitzem el “default” stream. Sempre que no especifiquem amb quin

stream treballem, sempre ho acabarem fent amb el stream “null” o “0” (és el mateix). Per tant, si cridem a cudaMemcpy o, cridem a cudaMemcpyAsync sense especificar cap stream, utilitzarem el stream 0.

Quan treballem amb més d'un stream, però, ens hem d'ocupar nosaltres manualment de crear-los i destruir-los. Això ho podem fer fàcilment amb bucles:

```
cudaStream_t stream[nStreams];
for (int i = 0; i < nStreams; i++)
{
    checkCuda(cudaStreamCreate(&stream[i]));
}
//programa CUDA...
for (int i = 0; i < nStreams; i++)
{
    checkCuda(cudaStreamDestroy(stream[i]));
}
```

La implementació del model concurrent, per tant, també es pot fer amb un bucle, on cada iteració representa un stream:

```
for (int i = 0; i < nStreams; i++)
{
    int offset = i * streamSize;
    checkCuda(cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes,
cudaMemcpyHostToDevice, stream[i]));
    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a,
offset);
    checkCuda(cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes,
cudaMemcpyDeviceToHost, stream[i]));
}
```

Si ens hi fixem en el codi anterior, perquè les transferències de memòria es puguin fer de forma paral·lela a les altres crides de CUDA, és necessari utilitzar la comanda cudaMemcpyAsync, especificant el stream en el que estem fent la transferència de la memòria.

Finalment, és important destacar que la representació de les imatges és un cas ideal, en el qual la majoria de vegades no es complirà. La majoria de vegades els diversos streams faran overlap entre ells.

Pel que fa a l'estructura que hem de seguir quan programem el codi per a multi-GPU, aquí la intenció és la mateixa que acabem d'explicar, és a dir construirem el model concurrent, però llavors de canviar de streams, el que farem es canviar de GPU.

És important destacar que, els events i streams de CUDA **sempre són per GPU**. Això vol dir que cada GPU sempre tindrà els seus propis streams. No és el mateix l'stream 2 de la GPU 1 que l'stream 2 de la GPU 2. Això, ben pensat, és un avantatge, ja que podem paral·lelitzar encara més el programa, si el que fem és utilitzar els diversos streams de cada GPU, i a més, el programa utilitza més de una GPU.

La programació per a un codi multi-GPU, per tant, també la podem fer amb un bucle, on aquí cada iteració representarà el treball d'una GPU. Per canviar entre GPUs fem servir la comanda `cudaSetDevice (device_id);`. Un codi d'exemple, per tant, podria ser el següent:

```
for (unsigned int device_id = 0; device_id < devices_count; device_id++)
{
    cudaSetDevice (device_id);
    const unsigned int chunk_size = chunk_ends[device_id] - chunk_begins[device_id];
    const unsigned char *host_chunk_input = img->data.get () + chunk_begins[device_id];
    unsigned char *host_chunk_output = host_result + chunk_begins[device_id];
    cudaMemcpy (devices_inputs[device_id], host_chunk_input, chunk_size * sizeof (unsigned char), cudaMemcpyHostToDevice);
    gpu_div_kernel_vec<div> << block_sizes[device_id], threads_per_block>> (devices_inputs[device_id], devices_outputs[device_id]);
    cudaMemcpy (host_chunk_output, devices_outputs[device_id], chunk_size * sizeof (unsigned char), cudaMemcpyDeviceToHost);
}
```

## Pregunta 6

**P:** Hablando de texturas, entre otras cosas tenemos, filtros bilineales, trilineales y anisotrópicos, ¿puedes describirlos? ¿qué implicaciones tienen en el diseño de la GPU?

**Resposta:**

Tant els filters bilineals, com els trilineals, com els anisotòpics, es tracta de mètodes que milloren la qualitat de les imatges (frames) de les textures en aquelles superfícies on aquestes es troben en un angle de visió respecte de la càmera que és oblic i que sembla ser no ortogonal.

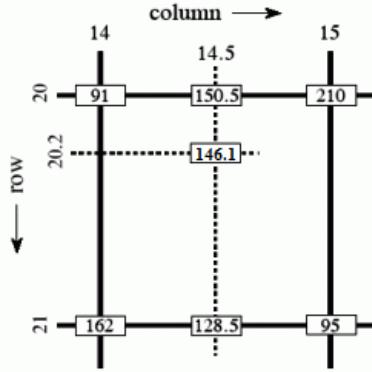
Per tant, es tracta de tècniques que intenten eliminar efectes de aliasing, els quals acostumen a ser presents quan volem renderitzar o mostrar elements que es troben cada cop més lluny de la càmera. No obstant, els filters bilineals donen resultats més pobres que els trilineals i els anisotòpics, i els trilineals donen resultats més pobres que els anisotòpics. Anem a descriure aquests filters:

El primer són els filters bilineals. La base d'aquests és la interpolació bilineal, és a dir, interpolar funcions de dos variables repetidament amb interpolació lineal.

En aquest mètode, s'utilitza un algorisme per assignar la ubicació d'un píxel de la pantalla a un punt determinat d'una textura (del mapa de textures) en el qual, per a calcular el valor d'un pixel, es fa la mitjana ponderada dels atributs (com per exemple el color, transparència) dels quatre texels que es troben més a prop del píxel. Aquest procés es fa per tots els píxels.

Per tant, en bilinear filtering s'utilitzen el conjunt dels 4 veïns de texels més propers a un pixel. Recordem que els texels són les unitats de mesura dels mapes de textura. En OpenGL, aplicar un filter bilineal seria l'equivalent a fer un mipmapping quan l'opció GL\_LINEAR està activada.

Vegem un exemple d'aquest mètode:



Imaginem que volem calcular el valor del píxel que es troba a les coordenades  $(20.2, 14.5)$ . Per calcular el valor d'aquest pixel, per tant, necessitarem interpolar els valors dels 4 texels veïns (els quals tenen valors: 91, 210, 162 i 95) respectivament.

Matemàticament aquesta interpolació la podem fer primer per separat per als dos texels de la fila 20 i per als dos texels de la columna 21, i després fer la interpolació d'aquests dos interpolacions. Per exemple, per tant, de la següent manera:

$$I_{20,14.5} = \frac{15 - 14.5}{15 - 14} \cdot 91 + \frac{14.5 - 14}{15 - 14} \cdot 210 = 150.5,$$

$$I_{21,14.5} = \frac{15 - 14.5}{15 - 14} \cdot 162 + \frac{14.5 - 14}{15 - 14} \cdot 95 = 128.5,$$

$$I_{20.2,14.5} = \frac{21 - 20.2}{21 - 20} \cdot 150.5 + \frac{20.2 - 20}{21 - 20} \cdot 128.5 = 146.1.$$

A continuació, ens trobem amb els filtres trilineals. Aquest mètode és una extensió del filtre bilineal. Per tant, també fa una interpolació dels texels mitjançant mipmapping.

La raó per la qual el filtre bilineal no és una solució molt atractiva per solucionar l'aliasing, és perquè per exemple quan l'utilitzem per una textura de tamany molt gran (respecte el tamany en pantalla que aquesta textura ocupa), llavors

quan escalem la imatge es poden provocar problemes de presisió a causa de texels perduts.

Per solucionar això, el que fa el filtre trilineal és interpolar entre els resultats del filtre bilineal per als dos mipmaps de textures que s'ajusten més al detall dels texels als pixels. D'aquesta manera, per tant, si per exemple el recorregut entre un pixel i el següent només avança una distància de 1/100 en les coordenades de textura, llavors el que faríem seria interpolar el resultat del mipmap que fa 128x128 i el que fa 64x64, aquesta interpolació, però, es faria sobre 100.

En OpenGL, aplicar un filtre trilineal seria l'equivalent a fer un mipmapping quan l'opció `GL_LINEAR_MIPMAP_LINEAR` està activada.

Cal destacar que el filtre trilineal no pot ser usat en aquells casos on un pixel és més petit que un texel, ja que com que el filtre trilineal utilitza tant mipmaps menors com majors en tant al que seria el tamany d'un texel respecte un pixel, llavors els mipmaps que són més grans que el tamany original d'una textura no estan definits.

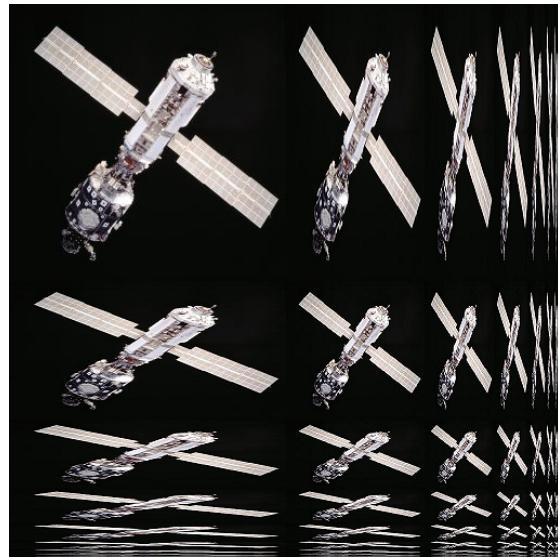
Explicat tot això, també hem de dir que el filtre trilineal tampoc dona resultats dalt tot correctes, ja que quan usem el filtre trilineal suposem que un pixel ocupa una àrea quadrada a la textura. Per tant, quan una textura es mostra en un angle força pronunciat respecte la càmera, es pot perdre detall.

Per solucionar aquest problema hi ha el filtre anisotòpic. La principal avantatge que té el filtre anisotòpic respecte els dos anteriors, és que redueix el blur, i preserva molts més detalls. Això, ho podem veure per exemple a la següent imatge:



La raó per la qual el filtre anisotòpic és capaç de preservar el detall, és perquè a part de fer interpolacions a partir de mipmaps downsamplejats preservant el rati de la textura (per exemple per a una textura quadrada, 128x128), també ho fa per a altres ratis, com per exemple 256x128, 32x128, etc.

D'aquesta manera, per tant, un exemple dels diversos mipmaps que es generarien podria ser la següent imatge:



És important remarcar que, mentre que cada un dels filters que hem explicat és millor que l'anterior, els recursos necessaris per computar cada un també incrementen considerablement, fent que el filtre anisotòpic sigui amb diferència el que més recursos utilitza. És per això que, si per exemple mirem les configuracions gràfiques de molts videojocs (o altres aplicacions gràfiques 3D), veurem que la majoria de vegades podrem triar la qualitat del filtre anisotropic. Aquesta qualitat bé determinada pel màxim rati que es considerarà durant el filtratge anisotòpic.

Tots aquests mètodes que acabem d'explicar es realitzen sempre dins la computació de la GPU, ja que recordem que les textures es guarden a la memòria de textures que es troba a la GPU. Quan es realitzen interpolacions lineals a la GPU, aquestes es fan amb una precisió baixa (amb números de 9 bits de coma

fixa i 8 bits de valor fraccional) per tal d'intentar aconseguir computar resultats en temps real.

Tal com acabem d'explicar, el filtratge lineal utilitza molts recursos de la GPU. Però per què? Bé, la raó és perquè és un filtratge extremadamen intensiu pel que fa a l'ample de banda del pipeline de la memòria de textures. Imaginem, per exemple, que cada pixel d'una textura que fa servir el filtre anisotròpic ocupa 4 bytes (podria ser més). Llavors cada pixel anisotròpic tècnicament ocuparia, per exemple, 512 bytes a la memòria de textures. Per tant, considerant que avui dia la majoria de pantalles tenen com a mínim 2 milions de píxels, si el que volem és executar una aplicació gràfica a com a mínim 60 fps, llavors l'ample de banda del pipeline de la memòria de textures serà facilment de centenars de GB/ segon.

Per sort, per solucionar aquest problema avui dia a nivell de hardware de GPU existeixen diversos sistemes. Dos sistemes molt efectius són:

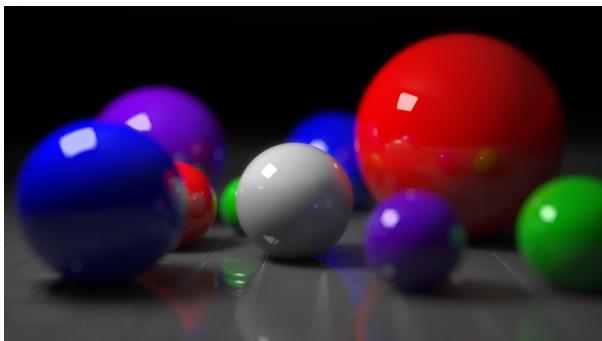
- **Cache de textures:** Tal com diu el nom, es tracta de la cache de textures. Això fa que l'ample de banda d'accés a memòria de textures incrementi molt més. Avui dia, gràcies a profilers com els d'Nvidia, som capaços de veure el hit rate d'aquestes caches.
- **Compressió de textures:** Actualment moltes GPU modernes porten incorporat a nivell de hardware un encodejador de textures per tal de comprimir-les, i aconseguir computar les textures de forma més ràpida amb el mateix ample de banda.

## Pregunta 7

P: Pregunta doble. Explica qué es el raytracing, cómo funciona, y cómo se implementa en las nuevas tarjetas de Nvidia, las RTX. Os ayudará que esta pregunta la encaréis como si fuera un pequeño trabajo.

Resposta:

Segur que tots, més d'alguna vegada, haurem vist algun render d'una escena 3D que ens ha marcat/sorprès per la seva gran qualitat, qualitat que a primera vista mai pensariem que podriem obtenir en temps real, en un videojoc (sí, és veritat que actualment en temps real es pot jugar amb ray-tracing, però no té res a veure amb la qualitat de ray-tracing que es pot observar amb un motor gràfic destinat a la qualitat (com Arnold (en el cas per exemple de Autodesk Maya), o Cycles (en el cas de Blender))). Renders com aquests:



Avui dia, si estem mínimament al dia amb el que fa les notícies de videojocs (o fins i tot de tecnologia en general) sabrem que això s'aconsegueix mitjançant la tècnica ray-tracing. Tot i que no sàpiguem exactament que és, el fet que ens soni d'alguna cosa el mot ray-tracing, o RTX, és el més normal avui dia, però no per exemple fa 10 anys. I, per què? Que fa 10 anys encara no s'havia inventat aquesta tècnica?

Doncs si, fa 10 anys ja existia aquesta tècnica, i també fa molts més enrere. El fet, però, que en aquests sobretot 3 últims anys aquesta tècnica hagi fet el boom pel que fa a la popularitat, es deu a que es tracta d'una tècnica literalment super

costosa pel que comporta als recursos que cal utilitzar per implementar-la. Una tècnica que, fa 6 anys, pensariem que és impossible d'implementar més o menys correctament en temps real. Però que, últimament, gràcies a una empresa anomenada Nvidia, això s'està fent possible, això ja és possible. Tot és qüestió de temps. I això no ho dic jo, ja ho va dir Kajiya (un dels pioners d'aquesta tècnica), l'any 1986:

"Ray tracing isn't too slow; computers are too slow." (Kajiya 1986)

De fet segurament ja ho haurem observat: segurament la majoria de nosaltres, en un dia normal i corrent quan obrim Youtube i recarreguem l'algorisme recomanador de videos ens haurà sortit el típic vídeo amb una marca en verd que diu "RTX", i la miniatura del video es veu caracteritzada per un videojoc amb uns gràfics brutals. Com [aquest](#).

Es tracta d'una tècnica, per tant, que ha vingut per quedar-se, i que tot apunta que en poc temps forçarà la creació de nous motors gràfics (que la implementin) els quals substitueixin els més populars d'avui dia (com DirectX, Vulkan, OpenGL...). De fet, això ja està passant dins del motors gràfics en temps real, com per exemple amb Unreal Engine.

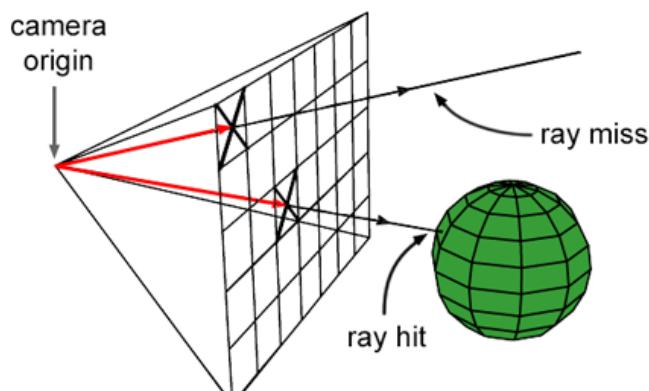
Però, i com pot ser possible que aquest mètode doni resultats molt més realistes que els típics motors de rasterització els quals estem acostumats? I, com pot ser que poguem implementar això a nivell de hardware? Per resoldre aquestes preguntes, primer, anem a veure com funciona l'algorisme de ray-tracing (hi ha moltes implementacions, de l'algorisme, com: el ray casting algorithm, el volume ray casting algorithm, SDF ray marching algorithm, Recursive ray tracing algorithm, etc. Nosaltres anem a veure una implementació general):

Ray-tracing té tres principals fases: la primera, la fase de casting Rays (llançament de rajos); la segona, Ray-Geometry Intersection (intersecció dels rajos); i la tercera, Shading. Anem a explicar aquestes tres fases:

- **Casting Rays:** La primera cosa que hem de fer per crear una imatge amb Ray-Tracing, és llançar un raig per a cada pixel de la imatge. Aquests rajos s'anomenen rajos de càmera, o rajos primàris, perquè són els primers rajos que llencem en una escena.

A part dels rajos primàris, també ens trobem amb els rajos secundaris, els quals són rajos que poden ser llançats pels primaris. Aquests rajos són útils per a esbrinar si un punt de la escena forma part d'alguna ombra, o per computar efectes com reflexions o refraccions.

Un cop ja hem llançat els rajos primàris, podem passar a la següent fase.



© www.scratchapixel.com

- **Ray-Geometry Intersection:** El que fem en aquesta fase és veure si un raig intersecta amb algun objecte de la escena. Això, per tant, requereix iterar per tots els objectes de la escena, i veure, donat un raig, si l'objecte fa intersecció amb el raig.

Aquí, cal destacar que, segons com es pugui descriure un objecte 3D, aquest serà més fàcil o més difícil de veure si fa intersecció amb el raig.

Per una part, objectes simples que poden ser descrits matemàticament, com esferes, discs... Es pot calcular la intersecció d'aquests fàcilment amb enfocs geomètrics o analítics.

No obstant, pel que fa a objectes més complexes, els quals, per exemple, només poden ser descrits per malles de polígons, superfícies de subdivision, NURBS... Aquí ja es complica la cosa, ja que cal dissenyar un mètode de càlcul de interseccions diferent per a cada tipus.

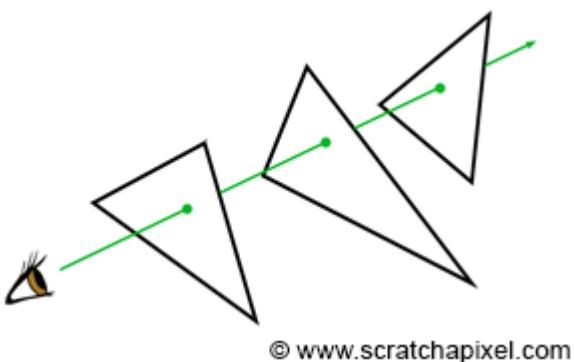
És per això que, el que s'acostuma a fer, en el 99% dels casos, és convertir qualsevol tipus de geometria en malles de polígons que bàsicament són triangles. Els triangles ens van molt bé a l'hora de treballar amb ray-tracing, no sols perquè es tracta d'un tipus de geometria al qual es pot convertir fàcilment des de qualsevol altre tipus, i pel qual calcular interseccions és molt senzill, sinó perquè tenen propietats geomètriques que altres tipus no tenen i que els fan realment interessants, com per exemple: són coplanaris, i és fàcil computar les seves coordenades baricèntriques (important per a la fase de shading).

Parlant de triangles, aquí cal destacar que hi ha un gran desavantatge de ray-tracing respecte el que passa amb la rasterització, i és que mentre que amb la rasterització podem realitzar certes optimitzacions en que podem directament descartar molts triangles abans que la fase de render comenci, això amb ray-tracing no ho podem fer.

Dit això, continuem amb la fase que estem explicant. El loop el qual hem explicat, i que itera per tots els objectes d'una escena donat un raig, en la majoria d'implementacions de ray-tracing acostuma a estar implementat en una funció anomenada “trace()”.

Realitzar correctament aquesta fase no és tan fàcil com directament veure si hi ha intersecció i ja està (donat un objecte i un raig), sinó que ens

podem trobar amb diversos problemes. Per exemple, que passa si un raig intersecta amb més d'un triangle?



© www.scratchapixel.com

Això, amb rasterització ho podem solucionar amb un depth-buffer. En el cas de ray-tracing, ho podem solucionar afegint una variable dins la funció “trace()” que enmagatzemi la distància més propera entre un raig i el punt d'intersecció.

Explicat tot això ja podem passar a la següent fase, shading. Cal destacar però, que a la fase de shading, serà necessari accedir a alguna informació pròpia d'aquesta segona fase, com la posició del punt d'intersecció d'un raig, la normal de la superfície en un punt d'intersecció... Gràcies, però, a que utilitzem triangles, això pot ser calculat fàcilment.

- **Shading:** Una vegada hem trobat l'objecte amb el qual un raig intersecta (en el cas que ho faci), ara necessitem saber quin serà el color d'aquest objecte al punt d'intersecció.

Mentre que, com deiem abans, ray-tracing és un algorisme que té un cost computacional molt gran, aquesta fase en canvi és bastant interessant, ja que té avantatges que sigui molt més atractiu que altres mètodes.

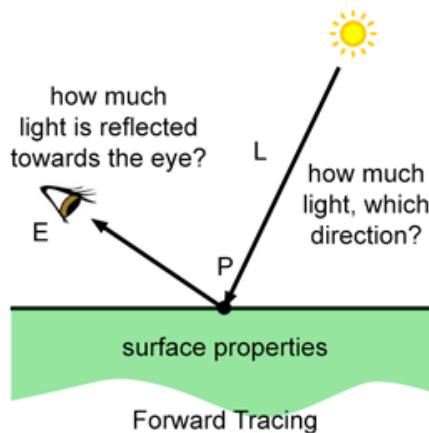
El color dels objectes o el seu grau d'intensitat acostuma a variar a mesura que recorrem la superfície d'un objecte. Això és degut al canvi de il·luminació així com també el fet que la textura de l'objecte varia al llarg de la seva superfície. El color d'un objecte en qualsevol punt de la seva

superficie és "només" el resultat de la manera com l'objecte retorna o reflecteix la llum que incideix a la seva superfície cap a l'observador.

El color que calcularem a partir d'una intersecció, depèn, per tant, de:

- **Quanta llum** incideix sobre la superfície de l'objecte (al punt d'intersecció però també lluny del punt si la superfície és transparent o translúcida).
- **La direcció d'aquesta llum.**
- Les propietats de la superfície (sobretot el color).
- **La posició de l'observador.** Moltes superfícies no reflecteixen la llum de forma equivalent en totes les direccions.

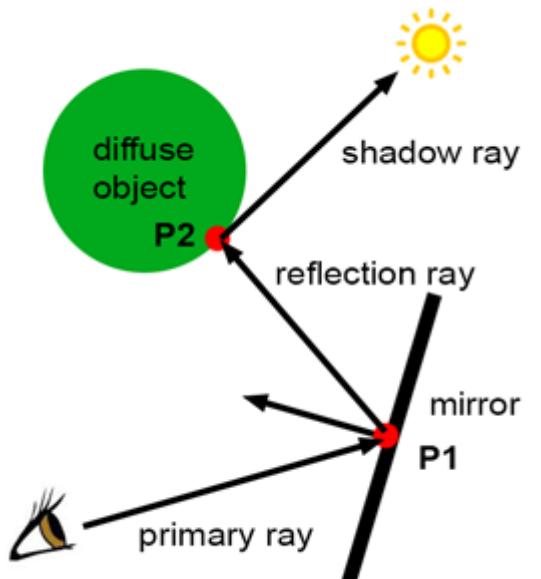
© www.scratchapixel.com



Per calcular aquestes propietats, les implementacions de ray-tracing implementen algorismes (hi ha diversos) sobre el transport de la llum. Aquí, cal destacar que, mentre que generalment triem traçar aquests camins des de l'ull (càmera) fins la superfície, i llavors des de la superfície fins a un punt de llum, aquest procés pot ser invertit.

El principal objectiu d'un algorisme que calcular el transport de la llum, és simular la manera en que la llum (com a forma d'energia) es distribueix per una escena. Aquí entren en joc propietats com, per exemple, donat un material, quina quantitat de llum és capaç de reflectir aquest al seu voltant. És per això que aquí necessitem un model d'il·luminació. Un

model d'il·luminació simple, per exemple, seria el que ja coneixem, Phong. No obstant, si utilitzem ray-tracing, no té sentit que utilitzem un model tan simple com Phong, l'objectiu és renderitzar una imatge realista, per tant s'acostumen a tenir moltes més coses en compte, com per exemple els BRDF.



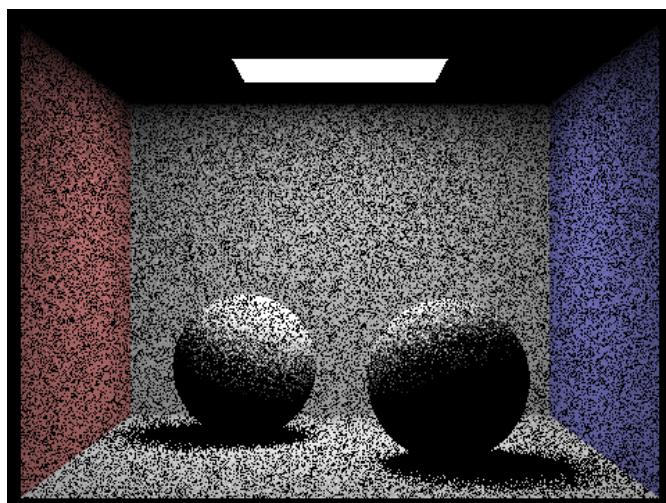
© www.scratchapixel.com

Aquí, per tal de calcular les característiques de les quals depèn el color que hem de calcular que ja hem dit, és necessari tornar a mencionar sobre els rajos secundaris, aquells rajos que llancem a partir dels rajos primaris, ja que necessitem aquests per calcular el color de forma correcta, fins que arribem a un punt, com per exemple a un punt de llum (o a un límit marcat per un threshold).

Els rajos secundaris acostumen a tenir altres noms, segons el tipus de raig que realment siguin, com per exemple: **reflection rays**, o **shadow rays**.

Destacar, també, que per calcular aquests rajos secundaris, serà necessari calcular molts més punts d'intersecció dels que necessitem per als rajos primàris.

Un cop explicat l'algorisme principal de ray-tracing, s'ha de dir que aquest acostuma a variar molt segons la implementació, i acostuma a haver-hi moltes opcions de configuració segons la qualitat que vulguem de la imatge, entre d'altres coses. Una d'aquestes principals opcions, la qual defineix principalment la qualitat d'un render de ray-tracing, és el nombre de rajos que volem tirar per pixel. Anteriorment hem explicat que principalment tirem un raig per pixel, però això no és així a la pràctica, ja que si fos així sempre ens sortirien escenes amb molt soroll, ja que necessitem més mostres per calcular d'una forma correcta el color d'un pixel. Aquest, és un valor, a més, que haurem d'anar incrementant gradualment a mesura que incrementem la resolució amb que volem fer els renders, o a mesura que hi posem més objectes i llums en una escena. Amb pocs rajos per pixel, ens quedaría una imatge com:



Com és d'esperar, aquesta configuració també és la que determina principalment els recursos del sistema que utilitzarem. És clar que sempre voldrem que tingui un valor molt alt de rajos per pixel, però hi haurà situacions on, degut a la nostra capacitat de computació, i l'escena que volem renderitzar, que trigariem un temps molt gran, massa gran, en realitzar un render, amb uns rajos / pixel molt gran. És aquí on entren en joc els denoisers, que com bé diu el nom, es tracta d'algorismes que s'encarreguen d'eliminar el soroll d'una imatge. Avui dia, hi ha denoisers com el propi d'Nvidia, els quals són accelerats per hardware, i que amb

poc temps donen resultats molt prometedors. No obstant, sempre que es pugui, és millor evitar la utilització d'aquests.

Ara que hem explicat les bases sobre què és un algorisme de ray-tracing, toca explicar com ha sigut capaç d'accelerar Nvidia això dins del hardware de les seves GPUs, i com a fet del ray-tracing casi una marca de la seva empresa, produint que avui el ray-tracing sigui pràcticament un monopoli liderat per Nvidia (cosa que trobo que es tracta d'una situació bastant greu, sobretot pel que fa a l'existència de la competència en aquest camp, però això ja és un altre tema que cal parlar amb més profunditat, a part).

Tot va començar l'any 2020, durant la conferència anual que fa Nvidia, liderada pel ja bastant conegit Jensen Huang, a la GTC 2020, Aquesta conferència va estar marcada per l'anunci d'una nova tecnologia que incorporaria la nova arquitectura de GPUs de Nvidia (Turing), tot i que realment també està suportejada per Volta. Aquesta tecnologia, s'anomena, **RTX**, i gràcies a aquesta, podem avui dia renderitzar escenes ray-tracing en temps real. Això, durant la [GTC 2020](#), ho vam poder experimentar amb vídeos com [aquest](#) (bé, no realment aquest, el de la GTC 2020 no era de nit, però aquest impacta més).

RTX (Ray Tracing Texel eXtreme) és una plataforma professional de computació visual principalment utilitzada per dissenyar: models a gran escala d'arquitectura i dissenys d'altres productes, visualització científica, exploració de l'energia, videojocs, producció de video... Bàsicament, utilitzada per tota escena 3D que es pugui beneficiar del ray-tracing.

RTX, actualment està incorporada a totes les GPU de la generació Volta cap endavant. Això últim és més aviat una generalització, no és dalt tot correcte, ja que hi ha per exemple targetes de la gamma Turing (com les gtx 16xx) que no la incorporen. Només aquelles targetes que disposin de Tensor Cores incorporen RTX, ja que RTX fa servir principalment els Tensor Cores. A la pregunta 3 d'aquest treball s'explica amb profunditat què és un Tensor Core.

RTX aconsegueix renderitzar escenes ray-tracing en temps real mitjançant una acceleració tant a nivell de software com a nivell de hardware. Pel que fa a nivell de hardware, com ja s'ha dit, les GPU es beneficien dels seus Tensor cores per accelerar operacions matemàtiques necessàries per accelerar els rajos, com per exemple el BVH (bounding volume hierarchy). A nivell de software, la implementació és lliure. Cal destacar, tot i tot el que s'ha dit, que per molt que Nvidia vengui la seva gamma de GPUs RTX com a targetes capaces de renderitzar escenes ray-tracing en temps real, “temps real” no crec que sigui la paraula dalt tot correcte en la majoria de casos, només cal veure com una rtx 3090 amb bastant esforç amb certs mods que incorporen ray-tracing en Minecraft puja dels 60 fps (parlo en resolucions altes (4K, és clar)). Podem observar això al següent [video](#).

Finalment, cal dir que ray-tracing actualment es pot implementar amb RTX mitjançant DXR (DirectX Raytracing API), OptiX (una API de Nvidia), DirectX, i Vulkan.

## Pregunta 8

P: Cuestionario. Pregunta Doble. Las 5 últimas cuestiones (f-j) se contestan con 1 frase. Las 5 primeras (a-e) requieren una explicación más elaborada.

a: ¿Qué es nvprof? ¿Qué información da? ¿Para qué sirve? Indica las 2 opciones de nvprof que consideres más relevantes.

b: Siempre decimos que en una GPU ocultamos la latencia con memoria. ¿Puedes explicar cómo lo hacemos?

c: ¿Qué significa que una GPU tenga los shaders unificados?

d: ¿Para qué sirve el Z buffer? ¿Cómo funciona?

e: En la invocación de un kernel de CUDA se pueden utilizar hasta 4 parámetros: KernelCUDA<<<par1, par2, par3, par4>>>(...). Describe par3 y par4.

f: ¿Cuál es la tarjeta gráfica más potente hoy (20/mayo–20/junio 2022)? Indica la fecha de la consulta.

g: ¿Para qué sirvela rutina de CUDA “\_\_syncthreads()”?

h: ¿Cómo hay que declarar un vector de 1024 floats dentro de un kernel para que se almacene en la memoria compartida de la GPU?

i: Enumera los elementos programables (shaders) que tienen el pipeline de DirectX 11.

j: Explicaen una frase qué es el CLIPPING.

Resposta:

a:

NVprof és un profiler que pot funcionar tant amb la commandline de Linux, Windows, com MacOs (en aquelles versions on encara es donava suport a les targetes gràfiques de Nvidia).

Gràcies a Nvprof podem recollir i visualitzar dades de profiling relacionades tant amb els kernels CUDA, com amb altres crides CUDA, com per exemple informació sobre les transferències de memòria, etc.

Nvprof disposa d'una gran varietat d'opcions, i pot operar en diversos modes.

Per exemple:

- **Summary Mode:** Es tracta del mode default de nvprof. En aquest mode, nvprof genera una única línia de resultats per a cada funció del kernel i cada tipus de còpia/conjunt de memòria CUDA realitzada per l'aplicació. Per a cada kernel, nvprof mostra el temps total de totes les instàncies del kernel i el tipus de còpia de memòria, així com el temps mitjà, mínim i màxim...
- **GPU-Trace and API-Trace Modes:** El mode GPU-Trace proporciona una cronología de totes les activitats que tenen lloc a la GPU en ordre cronològic. Cada execució del kernel i transferències de memòria es mostra a la sortida de la línia de comandes.
- **etc...**

Pel que fa a les 2 opcions més rellevants d'aquesta eina, en hi ha moltes que realment són importants, segons el que volguem examinar en un moment donat.

Dos, per exemple, de bastant útils, són les següents:

- **--print-gpu-trace:** Imprimeix el resultat del GPU-Trace mode que hem explicat abans.
- **--analysis-metrics:** Captura totes les mètriques de la GPU de la forma correcta perquè el visual profiler de Nvidia pugui realitzar el respectiu anàlisi.

És clar que nvprof sempre ha sigut la eina més completa pel que fa al profiling d'una targeta Nvidia. No obstant (i relacionat amb la última opció que acabem d'explicar), últimament Nvidia està desenvolupant tres principals profilers els quals són capaços de representar també una gran varietat de dades tal com ho fa nvprof, però que són molt més intuïtius, fàcils d'usar, i capaços d'interpretar les

dades d'una forma gràfica de forma molt més ràpida. Es tracta dels profilers Nsight. Existeixen tres principals profilers Nsight:

- **Nsight Graphics:** Potser el menys important quan estem treballant amb CUDA, ja que aquest s'encarrega del profiling de motors 3D com: Directx 11, Vulkan...
- **Nsight Systems:** Podem dir que és tracta d'un profiler universal de tot el que passa en una GPU d'Nvidia. És capaç tant de analitzar traçades de crides CUDA, com d'events gràfics de render del sistema operatiu...
- **Nsight Compute:** És sens dubte l'eina imprescindible de profiling quan treballem amb CUDA, ja que fa un profiling super específic de tot el que passa al kernel, donant consells i recomanacions dels problemes que poden estar passant. Els gràfics, per exemple, de la pregunta 2, s'han desenvolupat amb aquest profiler.

**b:**

Primer de tot entenem que és latència. Latència és el temps que una operació triga en completar-se. Dit això, per tant, és llògic que un dels principals bottlenecks en un sistema sigui la latència de memòria, ja que les transferències (instruccions) de memòria acostumen a tardar molt.

Les GPUs, per sort, són capaces de amagar aquesta latència, de forma que poden realitzar altres operacions mentre s'està esperant que el pipeline de transferències de memòria (el qual està saturat per altres threads) es buidi.

I com ho poden fer això? Doncs bé, això ho poden fer executant threads d'altres warps. Quan un warp accedeix a una ubicació de memòria que no està disponible, s'emet una sol·licitud de lectura o escriptura a la memòria, i aquesta sol·licitud es fusionarà amb altres d'altres threads del mateix warp.

Ara bé, sempre hem de pensar que no podem fer màgia, i arribarà un moment en que no podrem sempre amagar tota aquesta latència de memòria, sobretot en aquells casos on estem tot el rato fent per exemple accessos a la memòria global.

És per això, que és molt important fer ús dels altres tipus de memòria de la GPU, com la memòria shared.

**c:**

Tots sabem a què ens referim quan parlem de shaders, és a dir peces de codi que són executades en una GPU que serveixen per manipular una imatge abans que aquesta sigui emesa a la pantalla.

Si mirem un simple pipeline gràfic, com per exemple el pipeline del motor OpenGL, veurem que passem per diversos tipus de shaders, segons a la fase que estiguem, per exemple: vertex shader, fragment shader, etc...

Ara bé, quan nosaltres avui dia mirem les especificacions d'una targeta gràfica relativament moderna, observarem que només trobarem informació sobre el nombre general de “Shading Units” que té la GPU. No es fan distincions, per tant, d'entre un tipus de shader o un altre. Per exemple, si mirem la render config d'una rtx 4090 (GPU que per cert encara no està a la venda, i de fet tampoc ha sigut anunciada oficialment per Nvidia, però que ja s'han filtrat les especificacions) a la web [techpowerup](#), veurem que aquesta té 17408 shading units.

Per què no es fan distincions, llavors, entre diversos tipus de shaders, dins del hardware de les GPU modernes? La resposta a això, és perquè aquestes utilitzen el model de shaders unificats.

El model de shaders unificats fa referència a la computació dels shaders en una GPU on totes les etapes dels shaders del pipeline gràfic tenen les mateixes capacitats. Totes, poden llegir, per tant, textures i buffers, i utilitzen conjunts d'instruccions que són gairebé idèntics. Cada “shading unit” de la GPU, per tant, és capaç d'executar diversos tipus de shaders.

El model de shaders unificats és un model molt més eficient, i flexible que si no els unifiquem. Ja que, per exemple, en una situació amb una gran càrrega de

treball de geometria, d'aquesta manera la GPU podria assignar moltes més “shader units” a que operin vertex shaders.

d:

El z-buffer, o també anomenat depth buffer, és un buffer en el qual és guarda informació sobre la profunditat dels objectes (és a dir la coordenada Z en window space) d'una escena 3D.

Aquest tipus de buffer, per tant, és de gran ajuda a l'hora de representar correctament una escena 3D, ja que d'aquesta manera ens assegurem que els polígons que estan més a prop oclueixin aquells que estiguin més lluny.

Una representació del Z buffer de forma visual, podria ser per exemple la següent:



En el cas, per exemple, del motor gràfic OpenGL, aquest buffer és per exemple com el color buffer, en el sentit que guarda informació per fragment i té la mateixa amplada i llargada. En el cas d'OpenGL, els valors del Z buffer són entre 0 (màxim aprop) i 1 (màxim lluny). En OpenGL poden activar el test de Z buffer mitjançant la següent crida:

```
glEnable(GL_DEPTH_TEST);
```

Cal destacar que, avui dia, moltes GPU modernes, per termes d'eficiència incorporen una característica anomenada “early depth testing”, la qual permet fer el depth test abans que el fragment shader s'executi.

e:

Quan fem la crida a un kernel, la majoria de vegades només hi declarem dos paràmetres, els quals són, la dimensió del grid (en blocs) i dels blocs (en threads) respectivament. Però realment podem arribar a passar 4 paràmetres (els dos últims, per tant, són opcionals). Quins són aquests dos últims dos paràmetres?

Llegint la documentació oficial de CUDA, podem veure que aquests dos últims paràmetres són:

- **par3:** especifica el nombre de bytes que s'assigna de forma dinàmica a la shared memory per bloc, a part de l'assignació estàtica a la memòria shared. Si no especificuem aquest paràmetre, per default s'assignarà com a 0. Cal destacar que no podem assignar memòria dinàmica infinita a la shared memory per bloc, hi ha un límit, el qual el podem calcular com: limit assignació dinàmica a memòria shared per bloc = limit total memoria shared per bloc - quantitat memòria estàtica assignada a memòria shared per bloc.

El límit total de memòria shared per bloc el podem saber fàcilment fent una consulta a la comanda deviceQuery. Per exemple, per a una rtx 3070 Max-Q (laptop), aquest límit és:

Total amount of constant memory:	65536 bytes
----------------------------------	-------------

- **par4:** es tracta del stream amb el qual s'associarà l'execució del kernel. Recordem que a l'hora de fer crides CUDA, podem tenir diversos streams,

els quals ens serveixen per fer crides paral·leles en diversos streams. Útil per quan tenim més d'una GPU.

**f:**

Aquesta es tracta d'una pregunta molt relativa. A què ens referim amb més potent? Avui dia una GPU està dissenyada per fer molts tipus de coses, i podem avaluar la seva potència de formes diferents.

Si, mirem la majoria de benchmarks que avaluem les GPUs, veurem que aquests acostumen a avaluar el rendiment a partir d'un test gràfic 3D, i llavors comparen les GPU segons quina a fet més frames en un temps donat, o quina a extret el major framerate/temps.

Si mirem dos de les millors pàgines web que es consideren dels millors benchmarks de GPUs (les quals, personalment trobo que són: 3Dmark ([ul.com](#)) i [passmark](#)) veurem que pel que fa a la puntuació 3D, la gpu més potent és la RTX 3090 Ti. A la pàgina PassMark, podem observar també que pel que fa a la puntuació 2D, la millor segueix essent la 3090 Ti. No obstant això canviarà, com ja és habitual, en un futur no molt llunyà, ja que ja s'han filtrat les [especificacions de la rtx 4090](#), la qual és de la generació Lovelace, successora a Ampere (de la qual forma part la 3090Ti).

Deixant de banda el que acabem de dir, però, també podem avaluar el rendiment d'una altre forma. Podem avaluar el rendiment, per exemple, pel que fa al nombre d'instruccions en coma flotant, on aquí, com és d'esperar, lideren les targetes gràfiques de la gamma Tesla d'Nvidia. És important destacar que, pel que fa a això aquí sen's dubte lidera la GPU: NVIDIA H100 SXM5. Es tracta d'una GPU la qual es va llançar al 22 de març d'aquest any, i forma part de l'arquitectura Hopper (l'equivalent a Lovelace, però per a targetes destinades a servidors).

**g:**

La comanda `__syncthreads()` s'encarrega de fer esperar al thread que està avaluant fins que tots els threads del mateix bloc del que forma part també hagin

arribat a aquesta comanda. Serveix, per tant, per sincronitzar tots els threads d'un bloc.

**h:**

Per exemple, de la manera següent:

```
__shared__ float SharedMemory[1024];
```

D'aquesta manera hem fet l'allocació. Ara, els threads del bloc, ja podran escriure/consultar els valors d'aquest vector a la shared Memory anomenat: SharedMemory (es pot donar-li el nom que es vulgui).

**i:**

- Vertex shader.
- Hull shader.
- Domain shader.
- Geometry shader.
- Pixel shader.

**j:**

Clipping és una etapa del pipeline gràfic que determina quins primitius descartar i quins passar a la següent etapa.

## Annex

- **Pregunta 1:**

- [link.](#)

- **Pregunta 2:**

- Transparències teoria.
- [Cuda documentation.](#)
- [Nvidia Nsight Compute.](#)

- **Pregunta 3:**

- [link.](#)

- Pregunta 4:

- [link.](#)

- Pregunta 5:

- [link.](#)
- [link.](#)
- [link.](#)
- [link.](#)

- Pregunta 6:

- [link.](#)
- [link.](#)
- [link.](#)
- [link.](#)
- [link.](#)

- Pregunta 7:

- [link.](#)

- [link.](#)
- [link.](#)
- [link.](#)
- **Pregunta 8:**
  - **apartat a:**
    - [link.](#)
    - [link.](#)
    - [link.](#)
    - [link.](#)
    - [link.](#)
    - [link.](#)
  - **Apartat b:**
    - [link.](#)
    - [link.](#)
    - [link.](#)
  - **Apartat c:**
    - [link.](#)
    - [link.](#)
    - [link.](#)
  - **Apartat d:**
    - [link.](#)
    - [link.](#)
  - **Apartat e:**
    - [link.](#)
    - [link.](#)
  - **Apartat f:**
    - [link.](#)
    - [link.](#)
    - [link.](#)
    - [link.](#)
  - **Apartat g:**
    - [link.](#)

- [link.](#)
- **Apartat i:**
- [link.](#)
- **Apartat j:**
- [link.](#)