Parallel Rendering & Encoding

Programació Paral·lela

Enginyeria Informàtica – UB

Blai Ras Jimenez

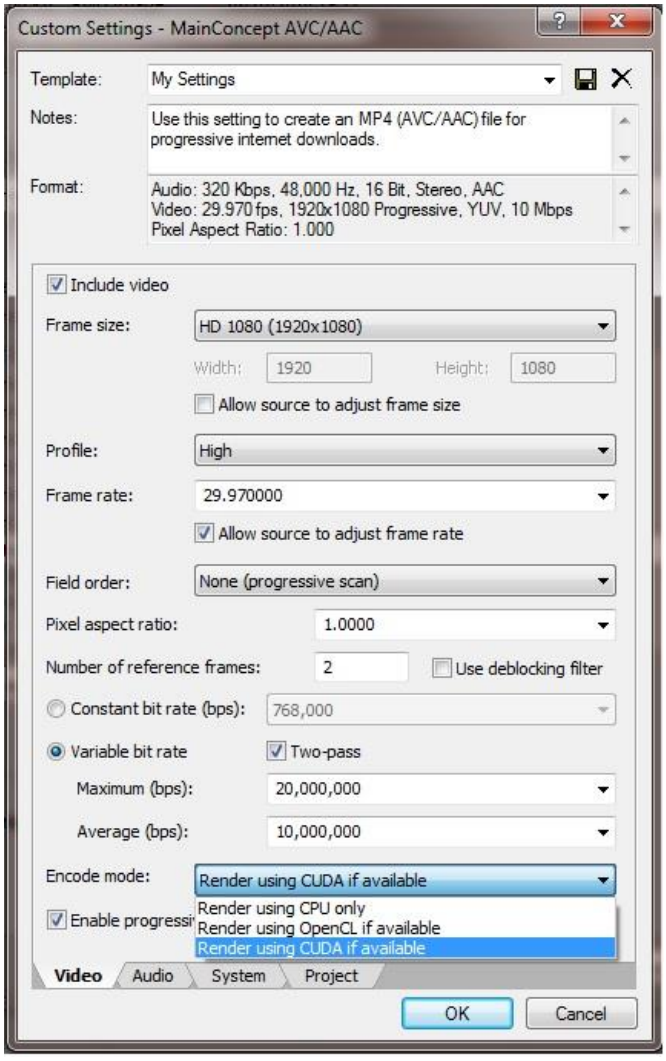
Índex

Introducció

La setmana passada m'ha germana em va demanar ajuda per a realitzar un vídeo que havien gravat ella i els seus amics pel comiat d'un company de feina. Concretament, volia utilitzar el meu ordinador, el qual és més potent que el seu i té un programa d'edició de vídeo instal·lat.

Un cop m'ha germana havia acabat d'editar al seu gust el vídeo en qüestió, només s'havia de renderitzar. M'ha germana, que no té molts coneixements en "multimèdia", va quedar al·lucinada quan va veure un temps de renderitzat de més d'una hora.

Com que no disposava de tanta estona, em va tocar a mi investigar com escurçar aquest temps. Jo pràcticament mai he fet servir aquest programa, però per la meva sorpresa, en un submenú del menú de renderitzat, vaig trobar una opció sorprenent: realitzar *l'encoding* fent ús de CUDA:



Il·lustració 1: Opcions d'encoding amb còdec MainConcept

A partir d'això, m'he decidit a investigar com es realitza aquesta codificació des del punt de vista de la computació en paral·lel. D'aquesta manera puc veure CUDA o *OpenMP* des d'un punt de vista diferent al impartit a teoria i aprofundir una mica més en el seu ús.

Què és

Tal i com el seu nom indica, la renderització en paral·lel és la distribució de la feina de representar un contingut multimèdia ja sigui entre els diferents nuclis d'un ordinador (renderització en CPU) o fent ús de la targeta gràfica (renderització en GPU).

Ara bé, per què es necessari distribuir la feina? Bé, la majoria de vegades renderitzar un vídeo o una escena és una tasca de molt alt cost computacional, i la correcta programació en paral·lel o la bona distribució dels recursos de la CPU ens pot ajudar a disminuir el seu temps de processat o inclús a fer possible escenes amb milers d'elements gràfics.

No ens confonguem: renderitzar és el procés de "crear" una o varies escenes de manera automàtica seguint un *pipeline* gràfic concret. En el context d'un vídeo, per exemple, seria el procés de "portar a la realitat" cada *frame* que em realitzat en el programa d'edició.

En canvi, codificar o *encoding* és el procés d'agafar un vídeo "descomprimit" i portar-lo a un format comprimit, un *stream* d'informació de varis formats com podem ser l'h264 o *mpeg-intra*. Aquests formats usen un embolcall o contenidor que ja ens són més familiars, com l'MP4 en el cas de l'H264.

Per tant, quan editem un vídeo i en conseqüència en creem un de nou, el que estem fent és primer renderitzar el vídeo (aplicar els canvis) i després codificar-lo per a que sigui reproduïble. En aquest treball intento estudiar com es realitza la paral·lelització d'aquests dos processos ampliant els coneixements adquirits a classe.

Com funciona

Renderització

A l'assignatura de Gràfics del tercer curs de la carrera és la primera vegada que se'ns introdueix el procés de renderització d'una escena. De fet, en ella aprenem quin és el *pipeline* de transformació i quins són tots els elements d'una escena. A més a més, veiem algoritmes de *shading* que operen en GPU. De totes maneres, mai entrem en detall en com es reparteix la feina en aquesta targeta gràfica.

Quan es tracta de renderitzar en paral·lel hi ha 3 mètodes principals a implementar:

Sort-first rendering

Conegut també com *Tilesort* o *Frame distribution*, aquest mètode divideix la pantalla en regions (*tiles*) les quals cadascuna té el seu propi *pipeline* gràfic. D'aquesta manera, diferents *tiles* poden ser renderitzats en paral·lel.

Per a poder implementar aquest algoritme es crea una xarxa de render-nodes. Aquests estan comandats per un *Tilesort* SPU (*Stream Procession Unit*). Aquest és l'encarregat de repartir la feina mitjançant crides de comandes *d'OpenGL*.

En poques paraules, cada node renderitza un tile de la pantalla general i quan tots han acabat es forma l'escena final. Aquesta divisió pot ser *creada* amb certs paràmetres:

- Amb seguiment d'estat: el *Tilesort* SPU manté un estat dels seus nodes i realitza un seguiment del seu estat. Implica una baixada de rendiment.
- "*Lazy Send*": s'envien les diferents *tiles* a renderitzar quan fan falta. Del contrari, s'envien totes de cop.

A la vegada, es pot usar *Tilesort* amb múltiples *SPU's* de manera que cadascuna d'elles té múltiples nodes. De totes maneres, s'incrementa la complexitat en la manera de sincronitzar els *threads* significativament.

Sort-first és un algoritme eficaç a l'hora de repartir la feina equitativament però s'ha d'anar en compte amb *l'overhead* de *tiles* quan el punt de vista (*frustum*) va canviant. Aquest equilibri de la càrrega de *tiles* de manera dinàmica implica més cost computacional.

Sort-last rendering

Conegut també com *Object distribution*, aquest mètode particiona l'escena per objectes en comptes de *tiles*. Dit d'una altre manera, es renderitzen tots els objectes de l'escena en clústers diferents i de manera paral·lela. Quan cadascun ha acabat, es forma l'escena final.

La complexitat del *sort-last rendering* està precisament en aquest *merge* de l'escena final, i es que cadascun dels objectes està a un pla o coordenades diferents. Hi ha dos mètodes per a realitzar-lo: *Z-testing* o *alpha blending*. No entrarem en detall a veure cadascun, simplement ambdós giren al voltant de la profunditat (eix z) per a decidir quin objecte va sobre quin altre. He llegit que aquest *alpha blending* usa semàfors un cop ha decidit quin objecte va primer.

Sort-middle rendering

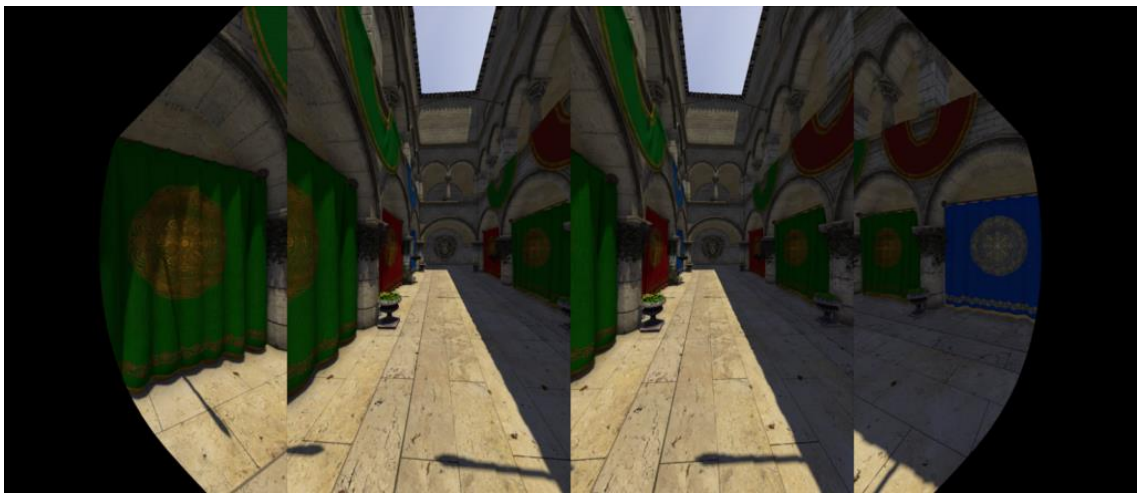
Tal i com el seu nom indica, aquest mètode és un híbrid entre el *sort-last* i el *sort-first*: alguns *frames* es paral·lelitzen fent ús del *sort-first* i d'altres utilitzen *sort-last rendering*.

La gràcia d'aquest mètode està en examinar l'escena. Si sabem que té molt *d'overlay* d'objectes, intentarem usar *sort-first rendering*. D'aquesta manera, evitem alts costos computacionals al extreure els valors z. Del contrari, potser convé més usar *sort-last rendering*.

Multi-view rendering

Aquest mètode intenta processar un *frame* sencer des de punts de vista diferents de manera paral·lela.

Alguns beneficis de moure el *frustum* son la millora de *l'anti-aliasing* (línies rectes i ben retallades) o una millor percepció del camp de profunditat. El seu principal problema és la quantitat de *frames* que escollim a processar a la vegada: més *frames* implica més temps de processat.



Il·lustració 2: Múltiples POV en una plataforma de Realitat Virtual

Fins aquí els diferents mètodes de renderització en paral·lel. Continuarem amb ells a l'apartat d'usos i aplicacions on entendrem millor el seu funcionament amb exemples d'ús d'avui en dia. A continuació entrem a l'apartat *d'encoding*.

Encoding

Tal i com he comentat anteriorment, el procés de codificació ve després del de renderitzar. Codificar un contingut multimèdia significa adaptar-lo o transformar-lo fins obtenir un *stream* d'informació intel·ligible per un SO. Aquest procés implica una compressió i per tant inevitablement una pèrdua de qualitat.

Per a paral·lelitzar aquest *encoding* podem fer ús de la CPU o la GPU d'un ordinador. Hi ha forces eines per a fer-ho, però jo hem centraré en un mètode de cadascuna d'elles: *OpenMP* per a CPU i NVIDIA NVENC per a GPU, les eines més properes a la teoria de l'assignatura.

Encoding amb *OpenMP*

OpenMP és una API que conté un conjunt de directives o *pragmas* per aconseguir paral·lelitzar parts de codi amb compartició de memòria. Opera a CPU, per tant, ha de córrer en un ordinador *multi-core*. És compatible amb els llenguatges C, C++ i Fortran.

OpenMP és un estàndard senzill i fàcil de manegar. Tot i que és compatible amb certes llibreries, la paral·lelització d'un procés de codificació d'un contingut multimèdia depèn de nosaltres. De totes maneres, no hi ha més remei que triar entre repartir funcions que fem a 'cada' *frame* (estimació/compensació de moviment, filtratges, codificació de l'entropía...) o repartir el tractat d'aquests *frames* (anar dividint un *frame* en trossos, blocs...).

Aquesta última organització segueix una arquitectura jeràrquica: grup de *frames* (GOP), després *frames*, després trossos de *frames*, blocs, etc... En conseqüència, poden designar *threads* amb més o menys CPU segons la informació que tractin. A això se li anomena tenir bona escalabilitat. En canvi, repartir funcions implica anar creant *threads* fins que la funció sigui indivisible.

A nivell de *load balance*, les dues organitzacions depenen de com de bons siguin els algoritmes que gestionen aquesta càrrega. Així doncs, ens decantem per paral·lelitzar el tractat de *frames* i no tant les funcions que els hi apliquem.

Una de les maneres d'explotar el paral·lelisme de *frames* és identificant-los per *frames* P, B o I.

- ***I-Frame (intra):*** una imatge completa, pot ser codificada o descodificada independentment.
- ***P-Frame (predicted):*** *frame* que només conté els canvis del *frame* anterior.
- ***B-Frame (bi-directional):*** conté els canvis dels dos *frames* anteriors i posteriors.

Una seqüència de vídeo convencional pot tenir una estructura del tipus "IBBPBBP...", és a dir, per a cada *P-frame* tenim dos *B-frames* enmig. En conseqüència, codificarem prioritàriament els I-*frames* o els P-*frames* primer i després els B-*frames*. De totes maneres, mentre codifico en paral·lel els B-*frames*, puc anar calculant ja el següent P-*frame*!

Una segona manera d'explotar el paral·lelisme de *frames* es particionar-los. Agafem cada *frame* i el trenquem en trossos molt petits. Cada tros és independent dels altres, per tant, podem paral·lelitzar la seva codificació sense seguir cap ordre. El gran desavantatge d'aquest procediment és l'increment de *bit-rate*.

Segons he pogut entendre, al particionar trenquem la dependència entre *macroblocs* de *frames*. Quan un *macroblock* trenca la seva relació amb altres *macroblocks*, és més difícil de comprimir, augmentant així el *bit-rate*. Si es disminueix la qualitat resultant, aquest increment de *bit-rate* és menor.

Un dels còdecs més populars avui en dia és l'H264. Aquest té una llibreria anomenada x264 que realitza la codificació a la CPU. No sé amb exactitud si opera en paral·lel (segurament), però sí que he trobat modificacions d'aquesta llibreria per realitzar (suposo que a més alt nivell) la paral·lelització de la codificació.

Encoding a la GPU

La codificació d'un vídeo pot suposar el tractat de milers i milers de *frames*. Avui en dia no només codifiquem vídeos: ¿què passa si s'ha de codificar un contingut que s'emet en *streaming* durant 8 hores? ¿Tenim *cores* suficients? ¿A quin cost?

Aquí és on entra la GPU: milers de *cores* operant en paral·lel amb una petita memòria volàtil. Hi ha més d'una eina per a codificar multimèdia a la GPU, però la més famosa és l'Nvidia NVENC.

NVENC és una característica implementada en les targetes gràfiques Nvidia GeForce 600 cap endavant que permet realitzar l'encoding d'un vídeo. Usa els milers de GPGPU's (*General Purpose Graphics Processing Units*) per a paral·lelitzar la feina de codificar tots els *frames* del vídeo o *stream* en qüestió.

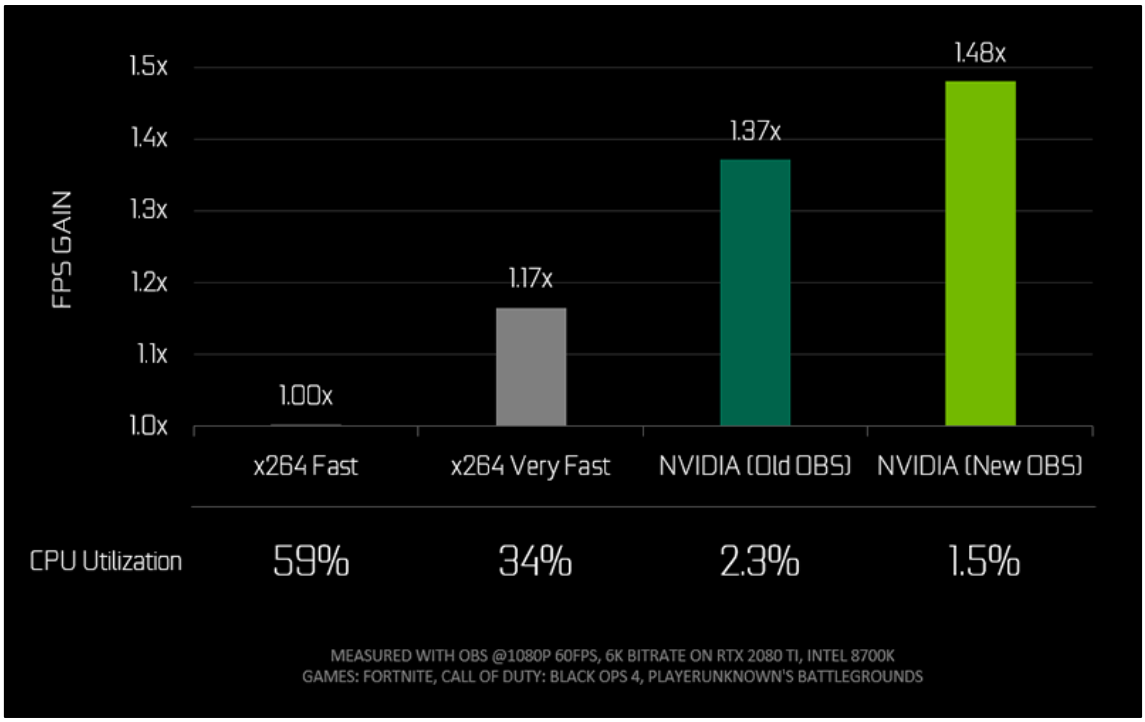
Al principi, al 2012, en les targetes gràfiques (Nvidia) d'arquitectura Kepler, NVENC era capaç de codificar amb el còdec H264 i aconseguir unes resolucions de 1920 x 1080 píxels. Avui en dia, amb l'arquitectura Turing, podem codificar amb el còdec H264 o H265 a resolució 8K i 30 FPS.

En un principi, NVENC no fa ús dels CUDA-Cores. Citant a la documentació del Nvidia Video Codec *SDK*:

- *"NVIDIA GPUs contain one or more hardware-based decoder and encoder(s) (separate from the CUDA cores) which provides [...]"*

De totes maneres, en arquitectures Maxwell, NVENC fa ús dels CUDA-Cores per aconseguir més *hardware-acceleration*. En arquitectures Pascal, s'introdueix al SDK el *Weighted Prediction feature*, possible gràcies als CUDA-Cores, també.

Malaurament no he trobat gran cosa sobre com funciona NVENC per dins, de manera que no he pogut esbrinar com paral·lelitza la informació. De totes maneres, sabem que la paral·lelització està allà, i, avui en dia, NVENC és la solució més ràpida a l'hora de codificar:



Il·lustració 3: Ús de la CPU en relació als FPS d'un joc i depenent d'on es faci la codificació

Ha arribat a un punt que fins i tot la gent que es dedica professionalment al *streaming* de videojocs usa dos ordinadors: un per jugar, amb la seva pròpia GPU, i un d'altre per codificar, usant també la seva pròpia GPU + NVENC.

A la imatge veiem les sigles OBS: *Open Broadcaster Software*, un dels milers de software de *vídeo streaming* compatible amb NVENC.

Usos i aplicacions

Si cités aquí totes les aplicacions o usos que hi ha actualment que usen alguns dels mètodes citats no acabaria mai, però si que destacaré els que em semblen més interessants o els més propers a mi.

1. Chromium
   El navegador Open-Source de Google usa àmpliament *Sort-last rendering*. Ho fa amb crides a OpenGL "thread-safe" mitjançant semàfors i barreres.
2. Realitat Virtual
   Amb el creixement exponencial de la fama del VR cada vegada veiem al mercat més pantalles compatibles amb aquesta tecnologia. Tot i així, necessitem primer una targeta gràfica compatible també!
   Un exemple d'elles és la gama de GPU's d'Nvidia d'arquitectura Turing, ja que implementen el *Multi-View rendering* explicat anteriorment. La Realitat Virtual implica grans camps de visió, múltiples *frustrums* que han de ser processats 'al moment', etc. El *Multi-View Rendering* ho fa possible!
3. Sony Vegas
   El misteri pel qual he realitzat aquest treball l'he resolt en mirant les *FAQ's* de Sony Vegas Pro 15 del 2016. *L'encoding* usant la GPU (CUDA) o a CPU (*OpenMP*) només és possible amb el còdec *MainConcept* AVC/AAC. Segons sembla, l'encoding amb CUDA no està suportat a partir de les GeForce 600... i que amb les GeForce 400-500 funciona bastant bé. Sembla un aspecte bastant desactualitzat. A mi m'apareix tot i tenir una 750 ja que posa (*if avaliable*), suposo.

Conclusions

Dos passos essencials per a crear un vídeo, escena o contingut multimèdia són el renderitzatge i el codificat. Aquests dos processos es realitzen en aquest ordre i comporten un alt cost computacional. Una solució per disminuir aquest alt cost és l'ús del paral·lelisme, tant a GPU com a CPU.

Renderitzar significa aplicar modificacions a un contingut multimèdia i representar-lo. Significa 'portar a la realitat', 'compilar'. Es possible realitzar un renderitzat de manera paral·lela que agilitza el procés del *pipeline* gràfic. Hem vist tres 4 mètodes, *sort-first*, *sort-last*, *middle-sort* i *multi-view rendering*.

Encoding o codificar significa comprimir, donar format a un *stream* d'informació per a que pugui ser reproduïble en els formats més utilitzats. Aquesta codificació es paral·lelitzable, tant a GPU com a CPU.

A nivell de CPU hem vist la seva implementació amb *OpenMP*, descomposant la informació per tasques o funcions que s'apliquen als *frames* o descomposant la pròpia informació de cada *frame*. Dins d'aquest últim grup hem vist dos explotacions diferents: a nivell de divisió de *frames* per seccions i blocs independents o a nivell d'anàlisis de *frames* segons tipus I, B o P.

A nivell de GPU hem girat al voltant de les GPU's d'Nvidia i la seva tecnologia NVENC, incidint sobretot amb la diferencia entre operar a CPU o a GPU.

Finalment hem vist alguns exemples de tecnologies conegudes que usen aquests conceptes.

Em sembla impressionant veure les possibilitats que han obert les GPU's al món de la multimèdia i com la programació en paral·lel ha donat les eines necessàries per dur-ho a terme. Tot i que des de fora pot semblar molt tancat i complex, hem vist eines obertes al públic i procediments fàcilment programables.

Referències

1. Nvidia Quadro P6000 – GPU Rendering
   http://www.pny.com/nvidia-quadro-p6000-gpu-rendering
2. Nvidia Developer Blog – Turing Multi-View Rendering in VRWorks
   https://devblogs.nvidia.com/turing-multi-view-rendering-vrworks/
3. A brief introduction to parallel rendering – Standford University
   http://graphics.stanford.edu/~mhouston/VisWorkshop04/ChromiumVis2004pt2.pdf
4. Wikipedia – Chromium (Web Browser)
   https://en.wikipedia.org/wiki/Chromium_(web_browser)
5. GeForce – Nvidia NVENC OBS Guide
   https://www.geforce.com/whats-new/guides/broadcasting-guide
6. Wikipedia – MainConcept (Codec)
   https://en.wikipedia.org/wiki/MainConcept
7. Nvidia Design Works – Nvidia Video Codec SDK
   https://developer.nvidia.com/nvidia-video-codec-sdk
8. NTown Film Entertainment - CUDA GPU Accelerated h264/h265/HEVC Video Encoding with ffmpeg
   https://ntown.at/de/knowledgebase/cuda-gpu-accelerated-h264-h265-hevc-video-encoding-with-ffmpeg/
9. Wikipedia – CUDA
   https://en.wikipedia.org/wiki/CUDA
10. Wikipedia – OpenMP
    https://en.wikipedia.org/wiki/OpenMP
11. Dr.Dobb's - Optimizing Video Encoding using Threads and Parallelism
    http://www.drdobbs.com/parallel/optimizing-video-encoding-using-threads/225600370?pgno=1