Arquitectures basades en computació gràfica (GPU)

Francesc Guim
Ivan Rodero

PID_00215404

UOC
Universitat Oberta de Catalunya

Els textos i imatges publicats en aquesta obra estan subjectes –llevat que s'indiqui el contrari– a una llicència de Reconeixement-NoComercial-SenseObraDerivada (BY-NC-ND) v.3.0 Espanya de Creative Commons. Podeu copiar-los, distribuir-los i transmetre'ls públicament sempre que en citeu l'autor i la font (FUOC. Fundació per a la Universitat Oberta de Catalunya), no en feu un ús comercial i no en feu obra derivada. La llicència completa es pot consultar a http://creativecommons.org/licenses/by-nc-nd/3.0/es/legalcode.ca

Índex

Introducció ... 5

Objectius ... 6

1. Introducció a la computació gràfica ... 7
- 1.1. *Pipeline* bàsic ... 7
- 1.2. Etapes programables del *pipeline* ... 9
- 1.3. Interfícies de programació del *pipeline* gràfic ... 11
- 1.4. Utilització del *pipeline* gràfic per a computació general ... 14

2. Arquitectures orientades al processament gràfic ... 15
- 2.1. Context i motivació ... 15
- 2.2. Visió històrica ... 16
- 2.3. Característiques bàsiques dels sistemes basats en GPU ... 18
- 2.4. Arquitectura Nvidia GeForce ... 20
- 2.5. Concepte d'arquitectura unificada ... 25

3. Arquitectures orientades a computació de propòsit general sobre GPU (GPGPU) ... 27
- 3.1. Arquitectura Nvidia ... 29
- 3.2. Arquitectura AMD (ATI) ... 35
  - 3.2.1. Arquitectura AMD CU ... 37
- 3.3. Arquitectura Intel Larrabee ... 39

4. Models de programació per a GPGPU ... 44
- 4.1. CUDA ... 44
  - 4.1.1. Arquitectura compatible amb CUDA ... 44
  - 4.1.2. Entorn de programació ... 45
  - 4.1.3. Model de memòria ... 49
  - 4.1.4. Definició de *kernels* ... 53
  - 4.1.5. Organització de fluxos ... 54
- 4.2. OpenCL ... 57
  - 4.2.1. Model de paral·lelisme a nivell de dades ... 57
  - 4.2.2. Arquitectura conceptual ... 59
  - 4.2.3. Model de memòria ... 59
  - 4.2.4. Gestió de *kernels* i de dispositius ... 61

5. Arquitectures *many-core*: el cas de l'Intel Xeon Phi ... 64
- 5.1. Història dels Xeon Phi ... 64
- 5.2. Presentació dels Xeon KNC i KNF ... 66
- 5.3. Arquitectura i sistema d'interconnexió ... 68

5.4. Nuclis dels KNC 72
5.5. Sistema de coherència 74
5.6. Protocol de coherència 77
5.7. Conclusions 79

Resum 81

Activitats 83

Bibliografia 86

Introducció

En aquest mòdul didàctic estudiarem les arquitectures basades en computació gràfica (GPU), que és una de les tendències en computació paral·lela amb més creixement en els darrers anys, i que gaudeix de gran popularitat, entre d'altres qüestions, a causa de la bona relació entre les prestacions que ofereixen i el seu cost.

Primer repassarem els fonaments de la computació gràfica per tal d'entendre els orígens i les característiques bàsiques de les arquitectures basades en computació gràfica. Estudiarem el *pipeline* gràfic i la seva evolució des d'arquitectures amb elements especialitzats fins als *pipelines* programables i l'arquitectura unificada que permet la programació d'aplicacions de propòsit general amb GPU. Veurem les diferències principals entre CPU i GPU i algunes de les arquitectures GPU més representatives, fent èmfasi en com poden ser programats certs elements.

Un cop presentades les arquitectures orientades a computació gràfica ens centrarem en les arquitectures unificades modernes que estan focalitzades en la programació d'aplicacions de propòsit general. Analitzarem les característiques de les principals arquitectures relacionades, com són les de Nvidia, AMD i Intel. Finalment, estudiarem CUDA i OpenCL, que són els principals models de programació per a aplicacions de propòsit general sobre GPU.

Objectius

Els materials didàctics d'aquest mòdul contenen les eines necessàries per a assolir els objectius següents:

1. Conèixer els fonaments de la computació gràfica i les característiques bàsiques del *pipeline* gràfic.

2. Entendre les diferències i similituds entre les arquitectures CPU i GPU i conèixer les característiques de les arquitectures gràfiques modernes.

3. Entendre la creixent importància de la programació massivament paral·lela i les motivacions de la computació de propòsit general per a GPU (GPGPU).

4. Entendre les diferències entre arquitectures orientades a gràfics i arquitectures orientades a la computació de propòsit general.

5. Identificar la necessitat i font de paral·lelisme de dades de les aplicacions per a GPU.

6. Aprendre els conceptes fonamentals per a programar dispositius GPU i els conceptes bàsics de CUDA i OpenCL.

1. Introducció a la computació gràfica

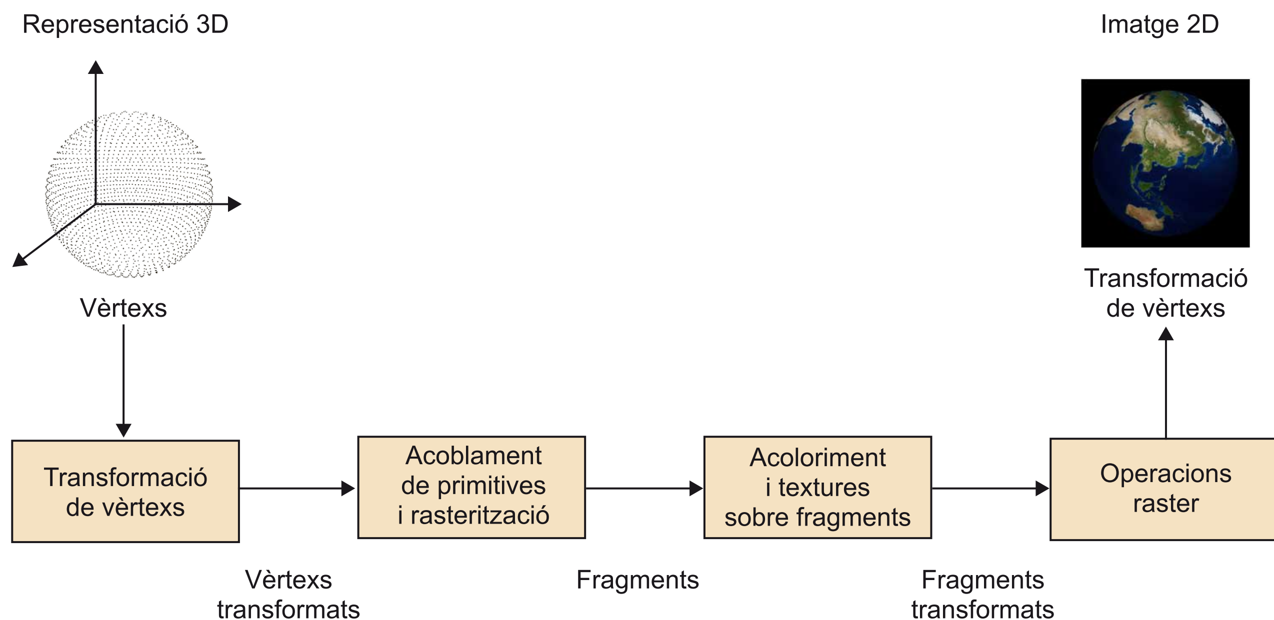
En aquest apartat estudiarem els fonaments més bàsics de la computació gràfica com a pas previ a estudiar les característiques de les arquitectures orientades al processament gràfic.

1.1. *Pipeline* bàsic

El processadors gràfics tradicionalment funcionen mitjançant un *pipeline* de processament format per etapes molt especialitzades en les funcions que desenvolupen, i que s'executen en un ordre preestablert. Cadascuna de les etapes del *pipeline* rep la sortida de l'etapa anterior i proporciona la seva sortida a l'etapa següent. A causa de la implementació mitjançant una estructura de *pipeline*, el processador gràfic pot executar diverses operacions en paral·lel. Com que aquest *pipeline* és específic per a la gestió de gràfics normalment s'anomena *pipeline gràfic* o *pipeline* de renderització. L'operació de renderització consisteix a projectar una representació en 3 dimensions a una imatge en 2 dimensions (que és l'objectiu d'un processador gràfic).

L'entrada del processador gràfic és una seqüència de vèrtexs agrupats en primitives geomètriques (polígons, línies i punts), que són tractades seqüencialment per mitjà de quatre etapes bàsiques, tal com mostra el *pipeline* simplificat de la figura 1.

Figura 1. *Pipeline* simplificat d'un processador gràfic



L'etapa de **transformació de vèrtexs** consisteix en l'execució d'una seqüència d'operacions matemàtiques sobre els vèrtexs d'entrada basant-se en la representació 3D proporcionada. Algunes d'aquestes operacions són l'actualització

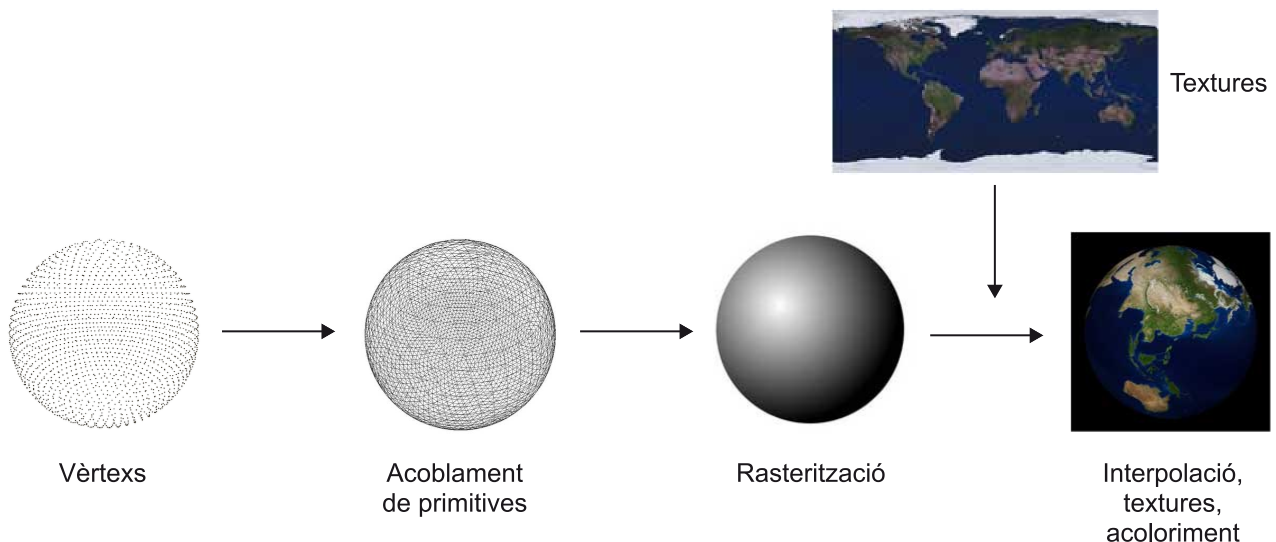
de la posició o la rotació dels objectes representats, la generació de coordinades per a poder aplicar textures o l'assignació de color als vèrtexs. La sortida d'aquesta fase és un conjunt de vèrtexs actualitzats, un per a cada vèrtex d'entrada.

En l'etapa d'**acoblament de primitives i rasterització** els vèrtexs transformats s'agrupen en primitives geomètriques basant-se en la informació rebuda juntament amb la seqüència inicial de vèrtexs. Com a resultat s'obté una seqüència de triangles, línies i punts. Aquests punts són posteriorment processats en una etapa anomenada *rasterització*. La rasterització és un procés que determina el conjunt de píxels afectats per una primitiva determinada. Els resultats de la rasterització són conjunts de localitzacions de píxels i conjunts de fragments. Un fragment té associada una localització i també informació relativa al seu color i brillantor o bé a coordinades de textura. Com a norma general podem dir que d'un conjunt de 3 o més vèrtexs s'obté un fragment.

En l'etapa d'**aplicació de textures i acoloriment** el conjunt de fragments és processat mitjançant operacions d'interpolació (predir valors a partir de la informació coneguda), operacions matemàtiques, d'aplicació de textures i de determinació del color final de cada fragment. Com a resultat d'aquesta etapa s'obté un fragment actualitzat (acolorit) per a cada fragment d'entrada.

En les últimes etapes del *pipeline* es fan operacions anomenades ***raster***, que s'encarreguen d'analitzar els fragments mitjançant un conjunt de tests relacionats amb aspectes gràfics. Aquests tests determinen els valors finals que prendran els píxels. Si algun d'aquests tests falla es descarta el píxel corresponent, i si tots són correctes, finalment el píxel s'escriu en memòria (*framebuffer*). La figura 2 mostra les diferents funcionalitats del *pipeline* gràfic

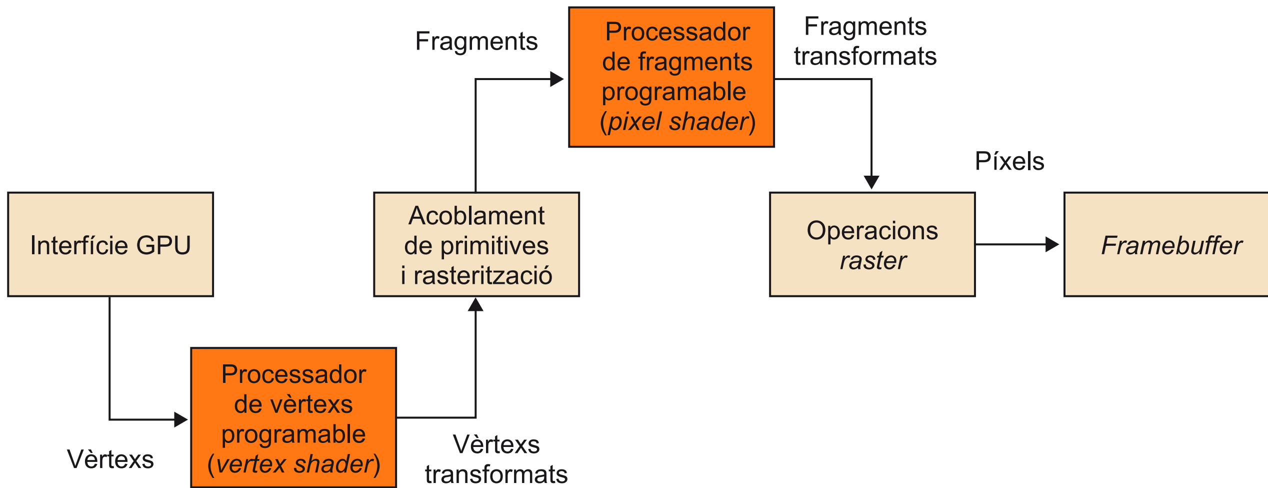
Figura 2. Resum de les funcionalitats del *pipeline* gràfic



1.2. Etapes programables del *pipeline*

Tot i que la tendència és proporcionar un nombre més elevat d'unitats programables en els processadors gràfics, les fases que típicament en permeten la programació són les de transformació de vèrtexs i transformació de fragments. Tal com mostra la figura 3, aquestes dues fases permeten programació mitjançant el processador de vèrtexs (*vertex shader*) i el processador de fragments (*pixel shader*).

Figura 3. *Pipeline* gràfic programable



Les etapes de procés de vèrtexs i de fragments poden ser programades mitjançant els processadors corresponents.

El funcionament d'un processador de vèrtexs programable és, en essència, molt similar al d'un processador de fragments. El primer pas consisteix en la càrrega dels atributs associats als vèrtexs per analitzar. Aquests es poden carregar en registres interns del processador de vèrtexs mateix. Hi ha tres tipus de registres:

- Registres d'atributs de vèrtexs, només de lectura, amb informació relativa a cadascun dels vèrtexs.
- Registres temporals, de lectura/escriptura, utilitzats en càlculs provisionals.
- Registres de sortida, on s'emmagatzemen els nous atributs dels vèrtexs transformats que, a continuació passaran al processador de fragments.

Un cop els atributs s'han carregat, el processador executa de manera seqüencial cadascuna de les instruccions que componen el programa. Aquestes instruccions es troben en zones reservades de la memòria de vídeo.

Un dels principals inconvenients d'aquests tipus de processadors programables és la limitació del conjunt d'instruccions que són capaços d'executar. Les operacions que els processadors de vèrtexs han de poder fer inclouen bàsicament:

- Operacions matemàtiques en coma flotant sobre vectors (ADD, MULT, mínim, màxim, etc.).

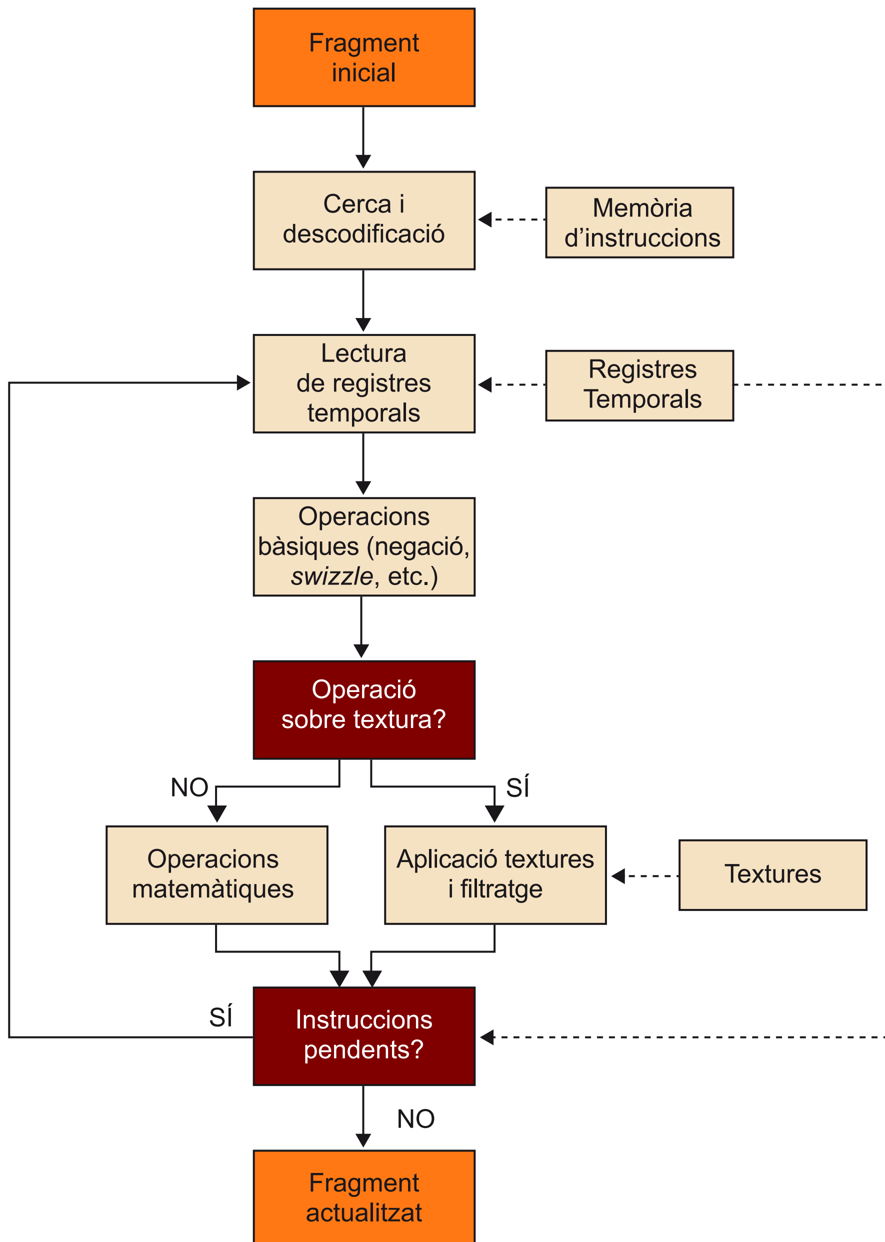
- Operacions amb maquinari per a la negació de vectors i *swizzling* (indexació de valors quan es carreguen de memòria).

- Exponencials, logarítmiques i trigonomètriques.

Els processadors gràfics moderns suporten també operacions de control de flux que permeten la implementació de bucles i construccions condicionals. Aquest tipus de processadors gràfics disposen de processadors de vèrtexs totalment programables que funcionen o bé en modalitat SIMD o bé en modalitat MIMD sobre els vèrtexs d'entrada.

Els processadors de fragments programables requereixen moltes de les operacions matemàtiques que exigeixen els processadors de vèrtexs però afegeixen operacions sobre textures. Aquest tipus d'operacions faciliten l'accés a imatges (textures) mitjançant l'ús d'un conjunt de coordinades, per a retornar a continuació la mostra llegida després d'un procés de filtratge. Aquest tipus de processadors només funcionen en modalitat SIMD sobre els elements d'entrada. Una altra característica d'aquests processadors és que poden accedir en modalitat de lectura a altres posicions de la textura (que correspondran a un flux amb dades d'entrada diferents). La figura 4 mostra el funcionament esquemàtic d'un d'aquests processadors.

Figura 4. Esquema del funcionament d'un processador de fragments



1.3. Interfícies de programació del *pipeline* gràfic

Per tal de programar el *pipeline* d'un processador gràfic de manera efectiva, els programadors necessiten biblioteques gràfiques que ofereixin les funcionalitats bàsiques per a especificar els objectes i les operacions necessàries per a produir aplicacions interactives amb gràfics en 3 dimensions.

OpenGL és una de les interfícies més populars. OpenGL està dissenyada de manera completament independent al maquinari per tal de permetre implementacions en diverses plataformes. Això fa que sigui molt portable però no inclou instruccions per a la gestió de finestres, gestió d'esdeveniments d'usuari, etc. Les operacions que es poden fer amb OpenGL són principalment les següents (i normalment també en l'ordre següent):

- Modelar figures a partir de primitives bàsiques, creant descripcions geomètriques dels objectes (punts, línies, polígons, fotografies i mapes de bits).

- Situar els objectes a l'espai tridimensional de l'escena i seleccionar la perspectiva des de la qual la volem observar.

- Calcular el color de tots els objectes. El color es pot assignar explícitament per a cada píxel o bé es pot calcular a partir de les condicions d'il·luminació o a partir de les textures.

- Convertir la descripció matemàtica dels objectes i la informació de color associada en un conjunt de píxels que es mostraran per pantalla.

A part d'aquestes operacions bàsiques, OpenGL també desenvolupa altres operacions més complexes com, per exemple, l'eliminació de parts d'objectes que queden ocultes darrere d'altres objectes de l'escena.

Atesa la versatilitat d'OpenGL, un programa en OpenGL pot arribar a ser força complex. En termes generals l'estructura bàsica d'un programa en OpenGL consta de les parts següents:

- Inicialitzar certs estats que controlen el procés de renderització.

- Especificar quins objectes s'han de visualitzar mitjançant la seva geometria i les seves propietats externes.

El codi 1.1 mostra un programa molt senzill en OpenGL. En concret, el codi de l'exemple genera un quadre blanc sobre un fons negre. Com que es treballa en 2 dimensions, no s'utilitza cap operació per a situar la perspectiva de l'observador a l'espai 3D. La primera funció obre una finestra en la pantalla. Aquesta funció no pertany realment a OpenGL i, per tant, la implementació depèn del gestor de finestres concret que s'utilitzi. Les funcions `glCleanColor` estableixen el color actual i `glClear` esborra la pantalla amb el color indicat prèviament amb `glCleanColor`. La funció `glColor3f` estableix quin color s'utilitzarà per a dibuixar objectes a partir d'aquell moment. En l'exemple es tracta del color blanc `(1.0, 1.0, 1.0)`. A continuació mitjançant `glOrtho` s'especifica el sistema de coordinades 3D que es vol utilitzar per a dibuixar l'escena i com es fa el mapatge a pantalla. Després de les funcions `glBegin` i `glEnd` es defineix la geometria de l'objecte. En l'exemple es defineixen dos objectes (que apareixeran a l'escena) mitjançant `glVertex2f`. En aquest cas s'utilitzen coordinades en dues dimensions, ja que la figura que es vol representar és un pla. Finalment la funció `glFlush` assegura que les instruccions anteriors s'executin.

```
#include <GL/gl.h>
#include <GL/glu.h>
void main ()
{
  OpenMainWindow ();
  glClearColor (0.0, 0.0, 0.0, 0.0);
  glClear (GL_COLOR_BUFFER_BIT);
  glColor3f (1.0, 1.0, 1.0);
  glOrtho (-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
  glBegin (GL_POLYGON);
    glVertex2f (-0.5, -0.5);
    glVertex2f (-0.5, 0.5);
    glVertex2f ( 0.5, 0.5);
    glVertex2f ( 0.5, -0.5);
  glEnd ();
  glFlush ();
}
```

Codi 1.1. Exemple de codi en OpenGL.

L'alternativa directa a OpenGL és Direct3D, que va ser presentat el 1995 i finalment va esdevenir el principal competidor d'OpenGL. Direct3D ofereix un conjunt de serveis gràfics 3D en temps real que s'encarrega de tota la renderització basada en programari-maquinari de tot el *pipeline* gràfic (transformacions, il·luminació i rasterització) i de l'accés transparent al dispositiu.

Direct3D és completament escalable, i permet que tot o una part del *pipeline* gràfic es pugui accelerar per maquinari. Direct3D també exposa les capacitats més complexes de les GPU com ara *z-buffering*, *anti-aliasing*, *alfa blending*, *mipmapping*, efectes atmosfèric i aplicació de textures mitjançant correcció de perspectiva. La integració amb altres tecnologies de DirectX permeten a Direct3D tenir altres característiques, com ara de relacionades amb vídeo, i proporcionar capacitats de gràfics 2D i 3D a aplicacions multimèdia.

Direct3D també té un nivell de complexitat força elevat. A mode d'exemple, el codi 1.2 mostra com podem definir un quadrat amb Direct3D. Com que Direct3D no disposa de cap primitiva per a quadrats com en el cas d'OpenGL, s'ha de definir amb una seqüència de dos triangles. Així doncs, els 3 primers vèrtexs formen un triangle i després la resta de vèrtexs afegeix un altre triangle format per aquest vèrtex i els 2 vèrtexs anteriors. Per tant, per a dibuixar un quadrat necessitem definir 4 vèrtexs, que corresponen a 2 triangles.

```
Vertex vertexs_quadrat[] ={
   // x, y, z, rhw, color
  { 250.0f, 200.0f, 0.5f, 1.0f, D3DCOLOR_XRGB(255,255,0) },
  { 250.0f, 50.0f, 0.5f, 1.0f, D3DCOLOR_XRGB(255,0,255) },
  { 400.0f, 200.0f, 0.5f, 1.0f, D3DCOLOR_XRGB(0,255,255) },
  { 400.0f, 50.0f, 0.5f, 1.0f, D3DCOLOR_XRGB(255,255,255) }
};
```

Codi 1.2. Exemple de codi en Direct3D per a dibuixar un quadrat.

Altres alternatives d'interfícies de programació del *pipeline* gràfic són SDL (*Simple Direct Media Layer*), Allegro i Render Ware.

1.4. Utilització del *pipeline* gràfic per a computació general

Més endavant veurem que podem utilitzar els processadors gràfics per a fer computació general. A l'hora d'adaptar les funcionalitats dels processadors gràfics a la computació general, sembla que la millor opció és utilitzar els processadors de fragments, per tres motius:

1) En un processador gràfic normalment hi ha més processadors de fragments que processadors de vèrtexs. Aquest fet ha estat cert fins a l'aparició de les arquitectures unificades, les quals fusionen tots dos tipus de processadors en un únic model de processador capaç de tractar tant vèrtexs com fragments.

2) Els processadors de fragments suporten operacions de lectura de dades procedents de textures (tot i que els darrers processadors gràfics també tenen aquesta capacitat en l'etapa de processament sobre processadors de vèrtexs). Les textures tenen un paper determinant a l'hora de treballar amb conjunts de dades (vectors i matrius) en processadors gràfics.

3) Quan es processa un fragment, el resultat s'emmagatzema directament en memòria, i per tant es pot reaprofitar directament per a ser processat un altre cop com un nou flux de dades. En canvi, en el cas dels processadors de vèrtexs, el resultat obtingut de la computació ha de passar encara per etapes de rasterització i processament de fragment abans d'arribar a la memòria, cosa que fa més complicat utilitzar-lo per a computació de propòsit general.

L'única manera en què els processadors de fragments poden accedir a la memòria és mitjançant les textures. La unitat de textures que hi ha a qualsevol processador gràfic fa el paper d'interfície només de lectura en memòria. Quan el processador gràfic genera una imatge hi pot haver dues opcions:

- Escriure la imatge en memòria (*framebuffer*), de manera que la imatge es mostri per pantalla.

- Escriure la imatge en memòria de textura. Aquesta tècnica s'anomena *render-to-buffer* i resulta imprescindible en computació general (tal com veurem més endavant), ja que és l'únic mecanisme per a implementar de manera senzilla una realimentació entre les dades de sortida d'un procés amb l'entrada del procés posterior sense passar per la memòria principal del sistema (que implicaria una transferència de les dades força costosa).

Tot i disposar d'interfícies tant de lectura com d'escriptura en memòria, cal tenir en compte que els processadors de fragments poden llegir sobre memòria un nombre de vegades il·limitat durant l'execució d'un programa, però només poden fer una única escriptura en finalitzar l'execució. Per tant, serà molt difícil utilitzar el *pipeline* tradicional programable per a executar programes de propòsit general de manera eficient.

2. Arquitectures orientades al processament gràfic

En aquest apartat estudiarem les motivacions i els factors d'èxit del desenvolupament d'arquitectures basades en computació gràfica, les característiques bàsiques d'aquestes arquitectures i el cas particular de l'arquitectura Nvidia orientada a gràfics com a cas d'ús. Finalment analitzarem les possibilitats i limitacions per a poder ser utilitzades per a computació de propòsit general.

2.1. Context i motivació

Durant les darreres dècades un dels mètodes més rellevants per a millorar el rendiment dels computadors ha estat l'augment de la velocitat del rellotge del processador. No obstant això, els fabricants es van veure obligats a buscar alternatives a aquest mètode a causa de diversos factors, com ara:

- Limitacions fonamentals en la fabricació de circuits integrats com, per exemple, el límit físic d'integració de transistors.

- Restriccions d'energia i calor degudes, per exemple, als límits de la tecnologia CMOS i a l'elevada densitat de potència elèctrica.

- Limitacions en el nivell de paral·lelisme a nivell d'instrucció (*instruction level parallelism*). Això implica que fent el *pipeline* més i més gran es pot acabar obtenint pitjor rendiment.

La solució que la indústria va adoptar var ser el desenvolupament de processadors amb múltiples nuclis centrant-se en el rendiment d'execució d'aplicacions paral·leles en contra de programes seqüencials. Així doncs, en els darrers anys s'ha produït un canvi molt significatiu en la indústria de la computació paral·lela. Actualment gairebé tots els ordinadors de consum incorporen processadors multinucli. Des de la incorporació dels processadors multinucli en dispositius quotidians, des de processadors duals per a dispositius mòbils fins a processadors amb més d'una dotzena de nuclis per a servidors i estacions de treball, la computació paral·lela ha deixat de ser exclusiva de supercomputadors i sistemes d'altes prestacions. Així, aquests dispositius proporcionen funcionalitats més sofisticades que els seus predecessors mitjançant computació paral·lela.

Paral·lelament, en els darrers anys la demanda per parts dels usuaris de gran potència de càlcul en l'àmbit de la generació de gràfics tridimensionals ha provocat una ràpida evolució del maquinari dedicat a la computació gràfica o GPU (*graphics processing unit*).

2.2. Visió històrica

Al final dels anys vuitanta i principi dels noranta hi va haver un augment molt important de la popularitat dels sistemes operatius amb interfícies gràfiques, com ara el Microsoft Windows, que va començar a acaparar gran part del mercat. Al principi dels noranta es van començar a fer populars els dispositius acceleradors per a 2D orientats a computadors personals. Aquests acceleradors donaven suport al sistema operatiu gràfic fent operacions sobre mapes de bits directament amb maquinari.

A la vegada que es produïa aquesta evolució en la informàtica de masses, en el món de la computació professional l'empresa Silicon Graphics va dedicar molts esforços durant els anys vuitanta a desenvolupar solucions orientades a gràfics tridimensionals. Silicon Graphics va popularitzar l'ús de tecnologies per a 3D en diferents sectors com ara el governamental, de defensa i la visualització científica i tècnica, a més de proporcionar eines per a crear efectes cinematogràfics mai vistos fins aquell moment. El 1992, Silicon Graphics va obrir la interfície de programació del seu maquinari per mitjà de la biblioteca OpenGL amb l'objectiu que OpenGL es convertís en l'estàndard per a escriure aplicacions gràfiques 3D independentment de la plataforma utilitzada.

A mitjan anys noranta, la demanda de gràfics 3D per part dels usuaris va augmentar vertiginosament a partir de l'aparició de jocs immersius en primera persona, com ara Doom, Duke Nukem 3D o Quake, que apropaven a la indústria dels videojocs per a ordinadors personals entorns 3D cada cop més realistes. Al mateix temps, empreses com Nvidia, ATI Technologies i 3dfx Interactive van començar a comercialitzar acceleradors gràfics que eren suficientment econòmics per als mercats de gran consum. Aquests primers desenvolupaments van representar el principi d'una nova era de gràfics 3D que ha portat a una constant progressió en les prestacions i capacitat computacionals del processament gràfic.

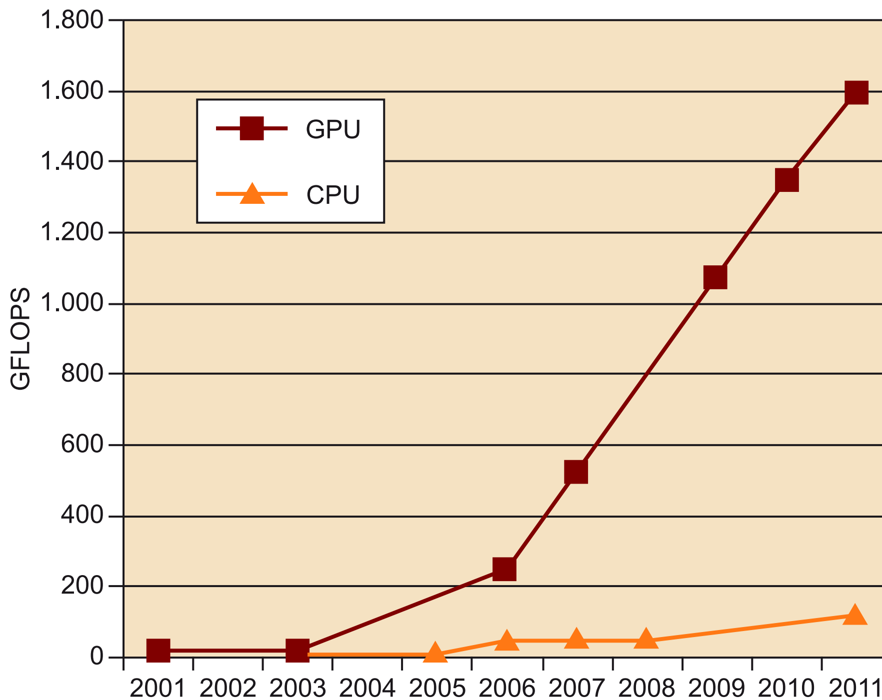
L'aparició de la Nvidia GeForce 256 va representar un impuls important per a obrir encara més el mercat del maquinari gràfic. Per primera vegada, un processador gràfic era capaç de fer operacions de transformació i il·luminació directament en el processador gràfic, i millorava així les possibilitats de desenvolupar aplicacions molt més interessants des d'un punt de vista visual. Com que les transformacions i la il·luminació ja eren part del *pipeline* d'OpenGL, la GeForce 256 va marcar el començament d'una progressió natural envers dispositius capaços d'implementar cada cop més etapes del *pipeline* gràfic directament en el processador gràfic.

Des del punt de vista de la computació paral·lela, l'aparició de la GeForce 3 de Nvidia l'any 2001 va representar segurament el canvi més significatiu en la tecnologia GPU fins aquell moment. La sèrie GeForce 3 va ser el primer xip de la indústria en implementar el que llavors era el nou estàndard DirectX 8.0. El maquinari compatible amb aquest estàndard disposava d'etapes programables

tant per al processament de vèrtexs com per al processament de fragments. Així doncs, per primer cop els desenvolupadors van tenir un cert control sobre els càlculs que es podien desenvolupar en les GPU.

Des del punt de vista arquitectural, les primeres generacions de GPU tenien una quantitat de nuclis força reduïda però ràpidament es va incrementar fins avui dia, en què parlem de dispositius de tipus *many-core* amb centenars de nuclis en un únic xip. Aquest augment de la quantitat de nuclis va fer que el 2003 hi hagués un salt important de la capacitat de càlcul en coma flotant de les GPU respecte de les CPU, tal com mostra la figura 5. Aquesta figura mostra l'evolució del rendiment en coma flotant (pic teòric) de la tecnologia basada en CPU (Intel) i GPU (Nvidia) durant l'última dècada. Es pot apreciar clarament que les GPU van molt més per davant que les CPU respecte de la millora de rendiment, especialment a partir de 2009, quan la relació era aproximadament de 10 a 1.

Figura 5. Comparativa de rendiment (pic teòric) entre tecnologies CPU i GPU



Les diferències tan grans entre el rendiment de CPU i GPU multinucli són degudes principalment a una qüestió de filosofia de disseny. Mentre que les GPU estan pensades per a explotar el paral·lelisme a nivell de dades amb paral·lelisme massiu i lògica prou simple, el disseny d'una CPU està optimitzat per a l'execució eficient de codi seqüencial. Les CPU utilitzen lògica de control sofisticada que permet paral·lelisme a nivell d'instrucció i fora d'ordre i utilitzen memòries cau força grans per tal de reduir el temps d'accés a les dades en memòria. També hi ha altres qüestions, com ara el consum elèctric o l'amplada de banda d'accés a memòria. Les GPU actuals tenen amplades de

banda a memòria entorn de 10 vegades més grans que les CPU, entre d'altres coses perquè les CPU han de satisfer requisits heretats dels sistemes operatius, de les aplicacions o dispositius d'entrada/sortida.

També hi ha hagut una evolució molt ràpida des del punt de vista de la programació de les GPU que ha fet canviar el propòsit d'aquests dispositius. Les GPU del principi dels anys 2000 utilitzaven unitats aritmètiques programables (*shaders*) per a retornar el color de cada píxel de la pantalla. Com que les operacions aritmètiques que s'aplicaven als colors d'entrada i textures les podia controlar completament el programador, els investigadors van observar que els colors d'entrada podien ser qualsevol tipus de dada. Així doncs, si les dades d'entrada eren dades numèriques que tenien algun significat més enllà d'un color, els programadors podien fer qualsevol dels càlculs que necessitessin sobre aquestes dades mitjançant els *shaders*.

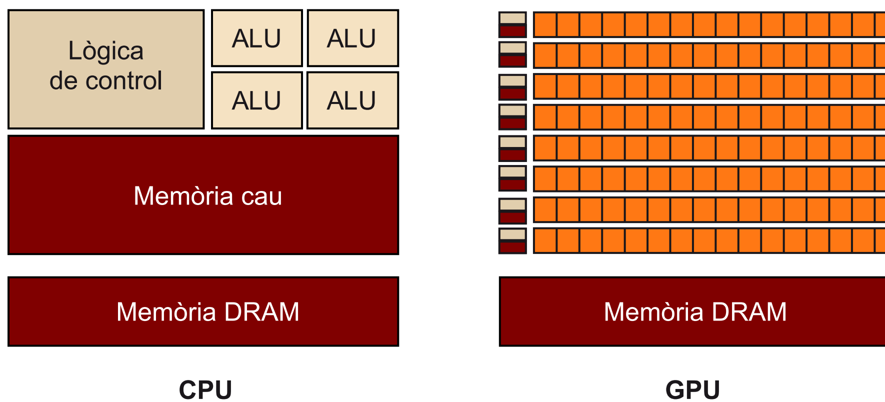
Tot i les limitacions que tenien els programadors per a desenvolupar aplicacions sobre GPU (per exemple, escriure resultats en qualsevol adreça de memòria) l'alt rendiment amb operacions aritmètiques va fer que es dediquessin molts esforços a desenvolupar interfícies i entorns de programació d'aplicacions de propòsit general per a GPU. Algunes d'aquestes interfícies de programació han tingut molta acceptació en diversos sectors, tot i que l'ús encara requereix una certa especialització.

2.3. Característiques bàsiques dels sistemes basats en GPU

Tal com s'ha comentat, la filosofia de disseny de les GPU està influenciada per la indústria del videojoc, que exerceix una gran pressió econòmica per a millorar la capacitat de fer una gran quantitat de càlculs en coma flotant per a processament gràfic. Aquesta demanda fa que els fabricants de GPU busquin maneres de maximitzar l'àrea del xip i la quantitat d'energia dedicada a càlculs en coma flotant. Per tal d'optimitzar el rendiment d'aquest tipus de càlculs s'ha optat per explotar un nombre massiu de fluxos d'execució. L'estratègia consisteix a explotar aquests fluxos de tal manera que, mentre que uns estan en espera per a l'accés a memòria, la resta pot seguir executant tasca pendent.

Tal com es mostra a la figura 6, en aquest model es requereix menys lògica de control per a cada flux d'execució. Al mateix temps, es disposa d'una petita memòria cau que permet que fluxos que comparteixen memòria tinguin l'amplada de banda suficient per a no haver d'anar-hi tots a la DRAM. En conseqüència, molta més àrea del xip es dedica al processament de dades en coma flotant.

Figura 6. Comparativa de la superfície dedicada típicament a computació, memòria i lògica de control per a CPU i GPU



Tal com veurem en detall més endavant, a més de disposar de molts fluxos d'execució disposats en nuclis més senzills que els de les CPU, altres característiques bàsiques de les arquitectures GPU són:

- Segueixen el model SIMD (una instrucció amb múltiples dades). Tots els nuclis executen alhora una mateixa instrucció; per tant, només es necessita descodificar la instrucció una única vegada per a tots els nuclis.

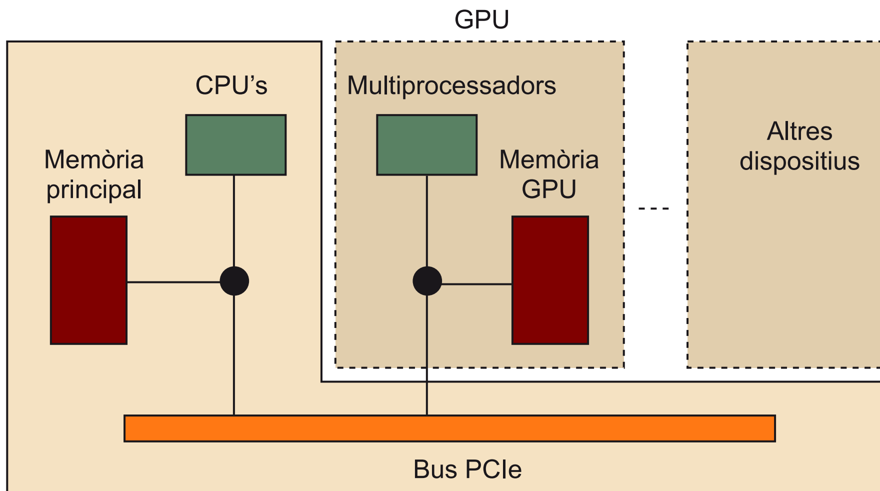
- La velocitat d'execució es basa en l'explotació de la localitat de les dades, tant la localitat temporal (quan accedim a una dada, és probable que es torni a utilitzar la mateixa dada en un futur proper) com la localitat espacial (quan accedim a una data, és molt probable que s'utilitzin dades adjacents a les ja utilitzades en un futur proper, i per això s'utilitzen memòries cau que desen diverses dades en una línia de la mida del bus).

- La memòria d'una GPU s'organitza en diversos tipus de memòria (local, global, constant i textura), les quals tenen diferents mides, temps d'accés i modes d'accés (per exemple, només lectura o lectura/escriptura).

- L'amplada de banda de la memòria és més gran.

En un sistema que disposa d'una o múltiples GPU, normalment les GPU són vistes com a dispositius externs a la CPU (que pot ser multinucli o fins i tot un multiprocessador), la qual es troba a la placa base del computador, tal com mostra la figura 7. La comunicació entre CPU i GPU es fa per mitjà d'un port dedicat. Actualment PCI Express o PCIe (*peripheral component interconnect express*) és l'estàndard per a fer aquesta comunicació.

Figura 7. Interconnexió entre CPU i GPU mitjançant PCIe



Un altre port de comunicació molt estès és l'AGP (*accelerated graphics port*), que es va desenvolupar durant l'última etapa de la passada dècada en resposta a la necessitat de velocitats de transferència més elevades entre CPU i GPU, a causa de la millora de les prestacions dels processadors gràfics. AGP és un port paral·lel de 32 bits amb accés directe al NorthBridge del sistema (el qual controla el funcionament del bus d'interconnexió de diversos elements crucials com ara CPU, memòria, etc.) i, per tant, permet utilitzar part de la memòria principal com a memòria de vídeo. La velocitat de transferència de dades varia entre 264 MB/s i 2GB/s (per a AGP 8x), en funció de la generació d'AGP.

Tot i l'augment en la velocitat de transferència de dades en les subseqüents generacions d'AGP, aquestes no són suficients per a dispositius gràfics d'última generació. Per aquest motiu el 2004 es va publicar l'estàndard PCIe. PCIe és un desenvolupament del port PCI que, a diferència d'AGP, utilitza una comunicació en sèrie en lloc de ser en paral·lel. Amb PCIe es poden arribar a velocitats de transferència de dades molt més elevades, que arriben a ser entorn d'alguns GB/s. Per exemple, en la versió 3.0 de PCIe el màxim teòric és de 16 GB/s direccionals i 32 GB/s bidireccionals.

A causa de l'organització de les GPU respecte a la CPU, cal tenir en compte que una GPU no pot accedir directament a la memòria principal i que una CPU no pot accedir directament a la memòria d'una GPU. Per tant, **caldrà copiar les dades entre CPU i GPU de manera explícita** (en tots dos sentits). Com a conseqüència, per exemple, no es pot fer servir `printf` en el codi que s'executa en una GPU i, en general, el procés de depuració en GPU acostuma a ser força feixuc.

2.4. Arquitectura Nvidia GeForce

Comercialment, Nvidia ofereix diferents productes, dividits en les tres famílies principals següents:

- **GeForce**: orientada al gran mercat de consum multimèdia (videojocs, edició de vídeo, fotografia digital, etc.).

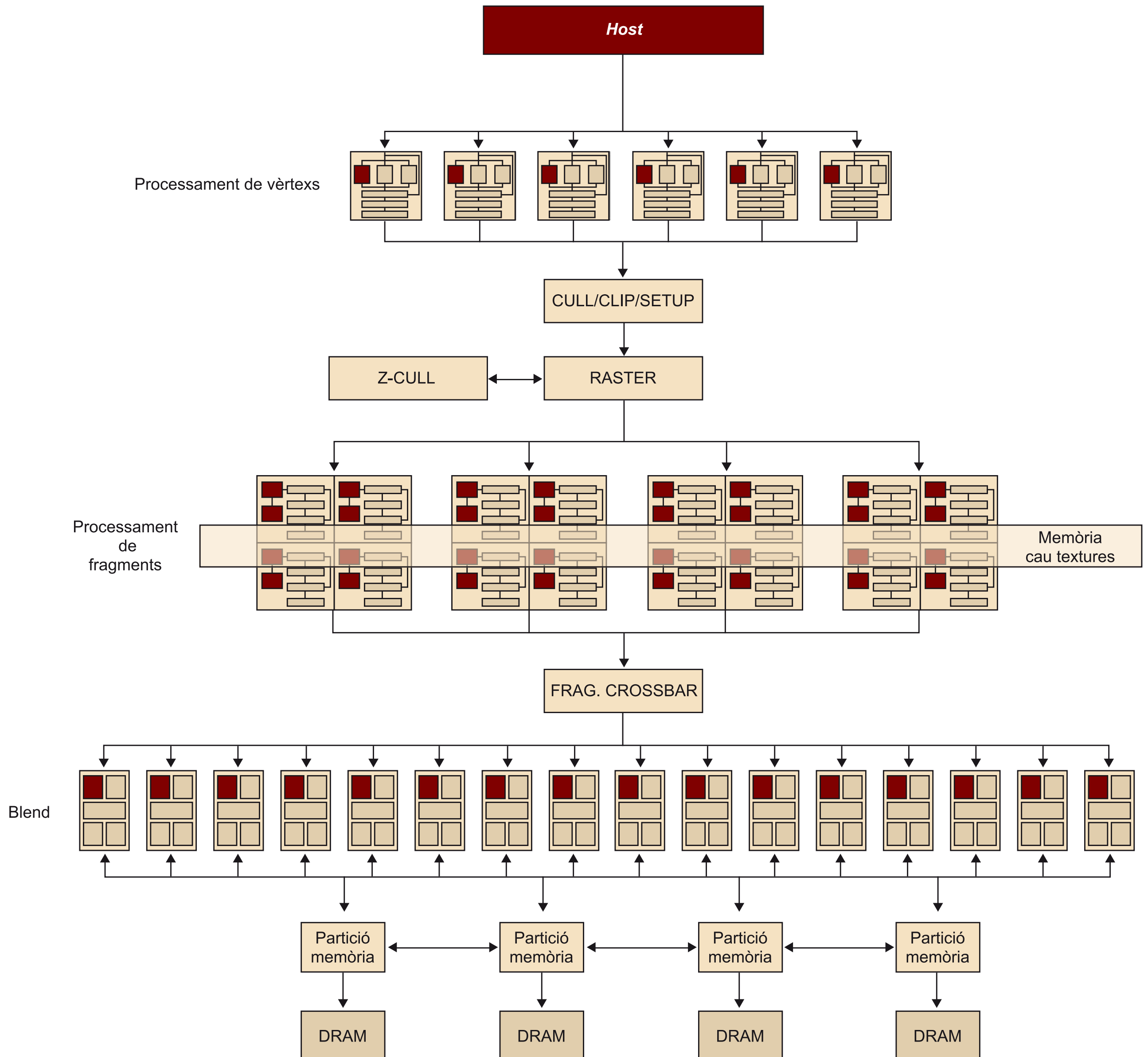
- **Quadro**: orientada a solucions professionals que requereixen models 3D, com ara els sectors de l'enginyeria o arquitectura.

- **Tesla**: orientada a la computació d'altes prestacions, com ara el processament d'informació sísmica, simulacions de bioquímica, models meteorològics i de canvi climàtic, computació financera o anàlisi de dades.

En aquest subapartat utilitzarem la sèrie Nvidia GeForce 6 com a cas d'ús de GPU pensada per a tractament de gràfics. Tot i no ser l'arquitectura més actual, ens servirà per a estudiar millor les diferències amb les arquitectures orientades a computació de propòsit general i a entendre millor l'evolució de les arquitectures de GPU. També podem trobar arquitectures semblants d'altres fabricants, com per exemple ATI (actualment AMD).

La figura 8 mostra de manera esquemàtica els blocs principals que formen l'arquitectura GeForce 6 de Nvidia.

Figura 8. Esquema de l'arquitectura de la GPU GeForce 6 de Nvidia

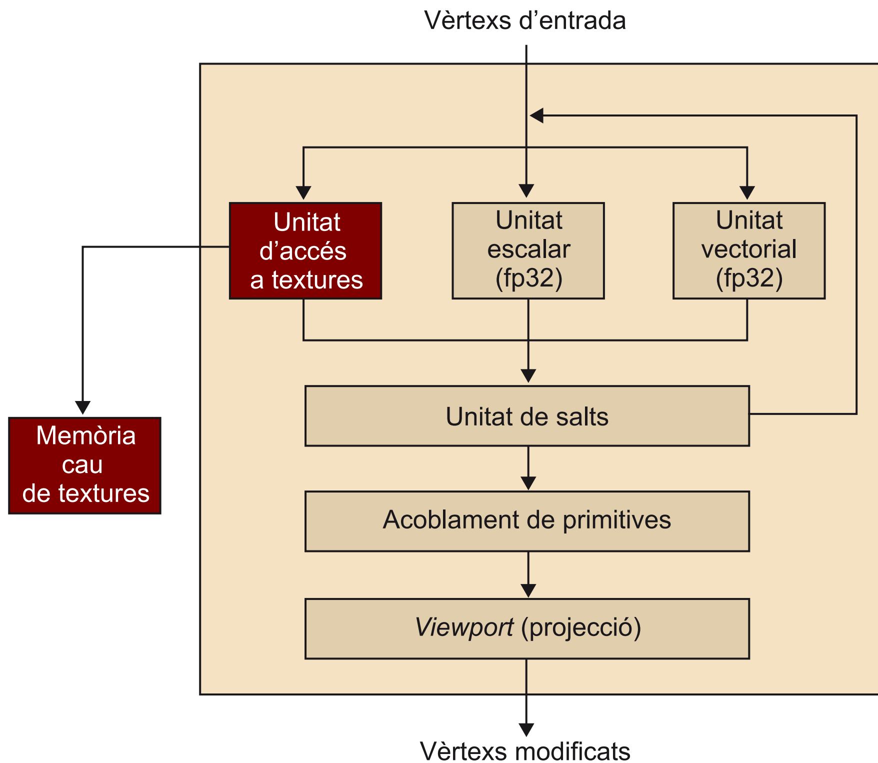


La CPU (*host* a la figura) envia a la unitat gràfica tres tipus de dades: instruccions, textures i vèrtexs. Els processadors de vèrtexs són els encarregats d'aplicar un programa específic que s'encarrega de fer les transformacions sobre cadascun dels vèrtexs d'entrada. La sèrie GeForce 6 va ser la primera que permetia que un programa executat en el processador de vèrtexs fos capaç de consultar dades de textura. Totes les operacions són fetes amb una precisió de 32 bits en coma flotant (fp32). El nombre de processadors de vèrtexs disponibles és variable en funció del model de processador, tot i que acostuma a ser entre dos i setze.

Com que els processadors de vèrtexs són capaços de fer lectures de la memòria de textures, cadascun té connexió a la memòria cau de textures, tal com mostra la figura 9. A més, hi ha una altra memòria cau (de vèrtexs) que emmagatzema dades relatives a vèrtexs abans i després d'haver estat processades pel processador de vèrtexs.

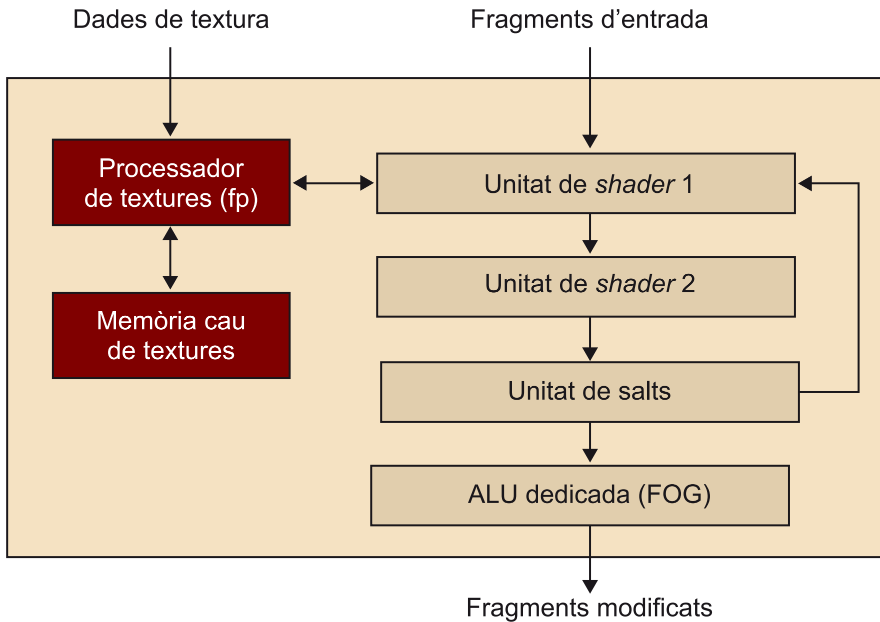
Els vèrtexs són agrupats seguidament en primitives (punts, línies o triangles). El bloc etiquetat en la figura com CULL/CLIP/SETUP fa operacions específiques per a cada primitiva, eliminant-les, transformant-les o preparant-les per a l'etapa de rasterització. El bloc funcional dedicat a la rasterització calcula quins píxels són afectats per cada primitiva, fent ús del bloc Z-CULL per a descartar píxels. Un cop fetes aquestes operacions els fragments poden ser vistos com a candidats a píxels, i poden ser transformats pel processador de fragments.

Figura 9. Esquema del processador de vèrtexs de la sèrie GeForce 6 de Nvidia



La figura 10 mostra l'arquitectura dels processadors de fragments típics de la sèrie GeForce 6 de Nvidia. Els processadors de fragments es divideixen en dues parts: la unitat de textura, que està dedicada al treball amb textures, i la unitat de processament de fragments, que opera, amb ajuda de la unitat de textures, sobre cadascun dels fragments d'entrada. Totes dues unitats operen de manera simultània per a aplicar un mateix programa (*shader*) a cadascun dels fragments de manera independent.

Figura 10. Esquema del processador de fragments de la sèrie GeForce 6 de Nvidia



De manera similar al que passaria amb els processadors de vèrtexs, les dades de textura es poden emmagatzemar en memòries cau en el xip mateix amb la finalitat de reduir l'amplada de banda a memòria, i augmentar així el rendiment del sistema.

El processador de fragments utilitza la unitat de textures per a carregar dades des de memòria (i, opcionalment, filtrar els fragments abans de ser rebuts pel processador de fragments mateix). La unitat de textures suporta gran quantitat de formats de dades i de tipus de filtratge, tot i que totes les dades són retornades al processador de fragments en format fp32 o fp16. Els processadors de fragments posseeixen dues unitats de processament que operen amb una precisió de 32 bits (unitats de *shader* en la figura). Els fragments circulen per les dues unitats de *shader* i per la unitat de salts abans de ser encaminats de nou cap a les unitats de *shader* per a continuar executant les operacions. Aquest reencaminament succeeix una vegada per cada cicle de rellotge. En general, és possible portar a terme un mínim de vuit operacions matemàtiques en el processador de fragments per cicle de rellotge, o quatre en el cas que es produeixi una lectura de dades de textura en la primera unitat de *shader*.

Per tal de reduir costos de fabricació, la memòria del sistema es divideix en quatre particions independents, cadascuna construïda a partir de memòries dinàmiques (DRAM). Totes les dades processades pel *pipeline* gràfic són emmagatzemades en memòria DRAM, mentre que les textures i les dades d'entrada (vèrtexs), es poden emmagatzemar tant en memòria DRAM com en la memòria principal del sistema. Aquestes quatre particions de memòria proporcionen un subsistema de memòria de força amplada (256 bits) i flexible, que assoleix velocitats de transferència properes als 35 GB/s (per a memòries DDR amb velocitat de rellotge de 550 Mhz, 256 bits per cicle de rellotge i 2 transferències per cicle).

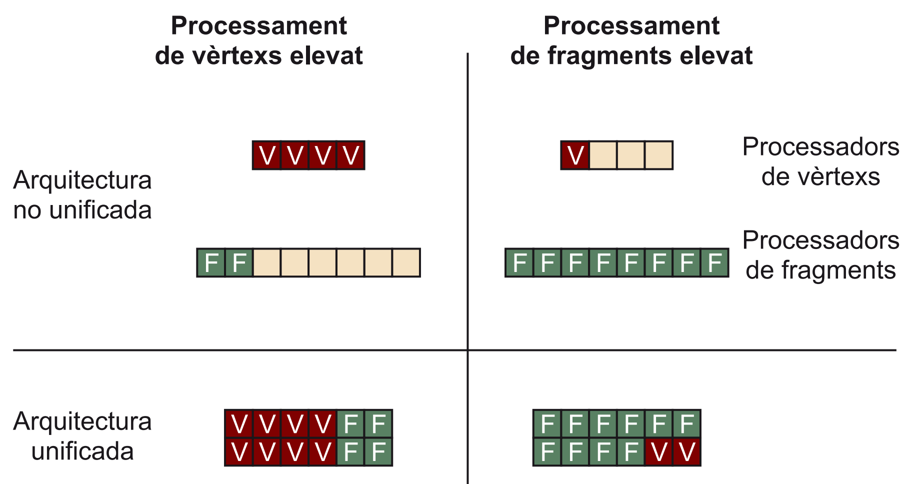
Per tant, comparant la implementació feta per aquesta sèrie de processadors (molt similar a altres sèries de la mateixa generació), és possible identificar quines unitats funcionals es corresponen amb cadascuna de les etapes del *pipeline* gràfic. Aquest tipus d'implementacions han estat les més esteses, fins a l'aparició de l'última generació de GPU, les quals estan basades en una arquitectura unificada i no fan cap diferenciació entre les diferents etapes del flux de processament a nivell de maquinari.

2.5. Concepte d'arquitectura unificada

L'arquitectura de la sèrie GeForce 6 estudiada anteriorment es podria definir com una arquitectura dividida a nivell de *shaders* o processadors programables. Això vol dir que disposa d'un maquinari especialitzat per a executar programes que operen sobre vèrtexs, i un altre dedicat exclusivament a l'execució sobre fragments. Tot i que el maquinari dedicat es pot adaptar força bé a la seva funció, hi ha certs inconvenients que fan que s'hagi optat per arquitectures totalment diferents a l'hora de desenvolupar una nova generació de processadors gràfics, basats en una arquitectura unificada.

Les arquitectures anteriors (no unificades) tenien problemes importants quan la càrrega de treball de processament de vèrtexs i de processament de fragments de les aplicacions gràfiques que executaven no estava balancejada. Com que normalment les GPU tenen menys unitat de processament de vèrtexs que de fragments, el problema s'agreujava quan la quantitat de treball sobre vèrtexs era predominant, ja que en aquest cas les unitats de vèrtexs quedaven totalment ocupades, mentre que moltes unitats de fragments podien quedar desaprofitades. De la mateixa manera, quan la càrrega és principalment sobre fragments també es pot produir un desaprofitament de recursos. La figura 11 mostra l'execució de dues aplicacions no balancejades i com l'arquitectura unificada pot oferir una solució més eficient.

Figura 11. Comparativa de l'assignació de processadors d'una GPU al processament de vèrtexs i de fragments en arquitectures unificades i no unificades, per a diferents tipus d'aplicacions



L'arquitectura unificada permet executar més computació simultània i millorar la utilització dels recursos.

La solució que es va desenvolupar a partir de la sèrie G80 de Nvidia o l'R600 de ATI (actualment AMD) va ser crear arquitectures unificades a nivell de *shaders*. En aquest tipus d'arquitectures, no hi ha la divisió a nivell de maquinari entre processadors de vèrtexs i processadors de fragments. Qualsevol unitat de processament que les forma (anomenades també ***stream processors***), és capaç de treballar tant a nivell de vèrtex com a nivell de fragment, sense estar especialitzat en un tipus en concret.

Aquest canvi en l'arquitectura també comporta un canvi important en el *pipeline* gràfic. Amb arquitectures unificades no hi ha parts específiques del xip associades a una etapa concreta del *pipeline*, sinó que un únic tipus d'unitat és l'encarregada de fer totes les operacions, sigui quina sigui la seva naturalesa. Un dels avantatges d'aquest tipus d'arquitectures és el balanceig implícit de la càrrega computacional. El conjunt de processadors es pot assignar a una tasca o una altra, depenent de la càrrega que el programa exigeixi a escala d'un determinat tipus de processament. Així doncs, l'arquitectura unificada pot solucionar el problema de balanceig de la càrrega i assignació d'unitats de processament a cada etapa del *pipeline* gràfic però, com a contrapartida, els processadors que componen la GPU són més complexos, ja que són més genèrics.

Aquest tipus d'arquitectures ofereixen un potencial molt més gran per a fer computació de propòsit general. En l'apartat següent veurem com podem programar aplicacions de propòsit general en aquest tipus de dispositius i també estudiarem diverses arquitectures GPU orientades a la computació de propòsit general.

3. Arquitectures orientades a computació de propòsit general sobre GPU (GPGPU)

En aquest apartat estudiarem com els processadors gràfics poden ser utilitzats per a executar aplicacions que tradicionalment són executades en CPU, veurem quins són els tipus d'aplicacions adequades per a computació gràfica i, finalment, estudiarem algunes de les principals arquitectures GPU orientades a computació de propòsit general.

Tal com hem vist anteriorment, la capacitat de càlcul d'una GPU actual és més elevada que la de les CPU més avançades per a fer certes tasques. Això ha fet que aquest tipus de dispositius estiguin esdevenint molt populars per al còmput d'algoritmes de propòsit general i no solament per a la generació de gràfics. La computació de propòsit general sobre GPU es coneix popularment com a GPGPU (*general-purpose computing on graphics processing unit*).

A més del nivell de paral·lelisme massiu i l'alt rendiment que proporcionen les plataformes GPU, cal destacar que aquests dispositius ofereixen una molt bona relació entre el seu preu i les seves prestacions, factors essencials perquè els fabricants hagin apostat fortament per aquesta tecnologia. Tot i això, les GPU proporcionen rendiments molt elevats només per a certes aplicacions a causa de les seves característiques arquitecturals i de funcionament. De manera general podem dir que les aplicacions que poden aprofitar millor les capacitats de les GPU són aquelles que compleixen les dues condicions següents:

- Treballen sobre vectors de dades grans.
- Tenen un paral·lelisme de gra fi tipus SIMD.

Hi ha diversos dominis en què la introducció de la GPGPU ha proporcionat una gran millora en termes de *speedup* de l'execució de les aplicacions associades. Entre les aplicacions que es poden adaptar eficientment a GPU, podem destacar l'àlgebra lineal, el processament d'imatges, algoritmes d'ordenació i cerca, processament de consultes sobre bases de dades, anàlisi de finances, mecànica de fluids computacional, predicció meteorològica, etc.

Un dels principals inconvenients a l'hora de treballar amb GPU és la dificultat per al programador per a transformar programes dissenyats per a CPU tradicionals a programes que puguin ser executats de manera eficient en una GPU. Per aquest motiu s'han desenvolupat models de programació, ja siguin de propietat (CUDA) o oberts (OpenCL), que proporcionen al programador un nivell d'abstracció més proper a la programació per a CPU, que li simplifiquen considerablement la seva tasca.

Tot i que els programadors poden veure reduïda la complexitat de la programació de GPGPU mitjançant aquestes abstraccions, analitzarem els principals fonaments de la programació GPGPU abans de veure les arquitectures sobre les quals s'executaran els programes desenvolupats. El principal argument del model GPGPU és la utilització del processador de fragments (o *pixel shader*) com a unitat de còmput. També cal tenir en compte que l'entrada/sortida és limitada: es poden fer lectures arbitràriament però hi ha restriccions per a les escriptures (per exemple, en les textures).

Per tal de comprendre com les aplicacions de propòsit general es poden executar en una GPU, podem fer una sèrie d'analogies entre GPU i CPU. Entre aquestes podem destacar les que estudiarem a continuació.

Hi ha dues estructures de dades fonamentals en les GPU per a representar conjunts d'elements del mateix tipus: les textures i els vectors de vèrtexs. Com que els processadors de fragments són els més utilitzats, podem fer un símil entre els vectors de dades en CPU i les textures en GPU. La memòria de textures és l'única memòria accessible de manera aleatòria des de programes de fragments o de vèrtexs. Qualsevol vèrtex, fragment o flux al qual s'hagi d'accedir de manera aleatòria s'ha de transformar primer a textura. Les textures poden ser llegides o escrites tant per la CPU com per la GPU. Podem fer un símil entre la lectura de memòria en CPU i la lectura de textures en GPU. En el cas de la GPU, l'escriptura es fa mitjançant el procés de renderització sobre una textura o bé copiant les dades des del *framebuffer* a la memòria de textura. Des del punt de vista de les estructures de dades, les textures són declarades com a conjunts de dades organitzats en una, dues o tres dimensions, i s'accedeix a cadascun dels seus elements mitjançant adreces d'una, dues o tres dimensions, respectivament. La manera més habitual de fer la transformació entre vectors (o matrius) i textures és mitjançant la creació de textures bidimensionals.

En la majoria d'aplicacions, especialment en les científiques, el problema s'acostuma a dividir en diferents etapes, les entrades de les quals depenen de les sortides d'etapes anteriors. Aquestes també es poden veure com les diferents iteracions dels bucles. Si parlem de fluxos de dades que són tractats per una GPU, cada nucli ha de processar un flux complet abans que el nucli següent es pugui començar a executar amb les dades resultants de l'execució anterior. La implementació d'aquesta retroalimentació de dades entre etapes del programa és trivial en la CPU, ja que qualsevol adreça de memòria pot ser llegida o escrita en qualsevol punt del programa. La tècnica *render-to-texture* és la que permet l'ús de procediments similars en GPU, escrivint resultats de l'execució d'un programa en memòria perquè puguin estar disponibles com a entrades per a etapes posteriors. Així doncs, podem fer un símil entre l'escriptura a memòria en CPU i el *render-to-texture* en GPU.

En GPU, la computació es fa normalment per mitjà d'un flux de processador de fragments, que s'haurà d'executar en les unitats funcionals de la GPU corresponents. Així doncs podem fer un símil entre el programa en CPU i la rasterit-

zació (o programa del *shader*) en GPU. Per començar la computació, es crea un conjunt de vèrtexs amb els quals podem alimentar el processadors de vèrtexs. L'etapa de rasterització determinarà quins píxels del flux de dades es veuen afectats per les primitives generades a partir d'aquests vèrtexs, i es genera un fragment per a cadascun. Per exemple, imaginem que el nostre objectiu és el d'operar sobre cadascun dels elements d'una matriu de N files i M columnes. En aquest cas, els processadors de fragments hauran de fer una (la mateixa) operació sobre cadascun dels N x M elements que composen la matriu.

A continuació utilitzarem la suma de matrius per a exemplificar les similituds entre GPU i CPU a l'hora de programar una aplicació de propòsit general. En aquest exemple es prenen com a operands d'entrada dues matrius de valors reals, A i B, per a obtenir una tercera matriu C. Els elements de la matriu C seran la suma dels elements corresponents de les matrius d'entrada. Una implementació en CPU crea tres matrius en memòria, recorrent cadascun dels elements de les matrius d'entrada, calculant per a cada parella la suma i posant el resultat en una tercera matriu, tal com mostra el codi 3.1.

```
float *A, B, C;
  int i, j;
  for (i=0; i<M; i++){
    for (j=0; j<N; j++){
      C[i][j] = A[i][j] + B[i][j];
    }
  }
```

Codi 3.1. Implementació de la suma de matrius per a CPU

El procediment en GPU és una mica més complex. Primer és necessari definir tres textures en memòria de vídeo, que actuaran de la mateixa manera que les matrius definides en memòria principal en el cas de les CPU. Cada nucli o programa per executar en la GPU correspon a aquelles operacions que es fan en el bucle més intern de la implementació per a CPU. Cal tenir en compte, però, algunes limitacions addicionals de les GPU, com el fet que no podrem escriure el resultat de l'operació de la suma directament a la textura corresponent a la matriu C. Caldrà, doncs, retornar el resultat de la suma i crear un flux addicional que recollirà aquest resultat i l'enviarà a la textura de destinació (mitjançant la tècnica *render-to-texture*). Finalment, les dades emmagatzemades en la textura corresponent a la matriu C es transfereix un altre cop cap a la memòria central perquè el programa que va invocar l'execució del nucli a la GPU pugui continuar l'execució.

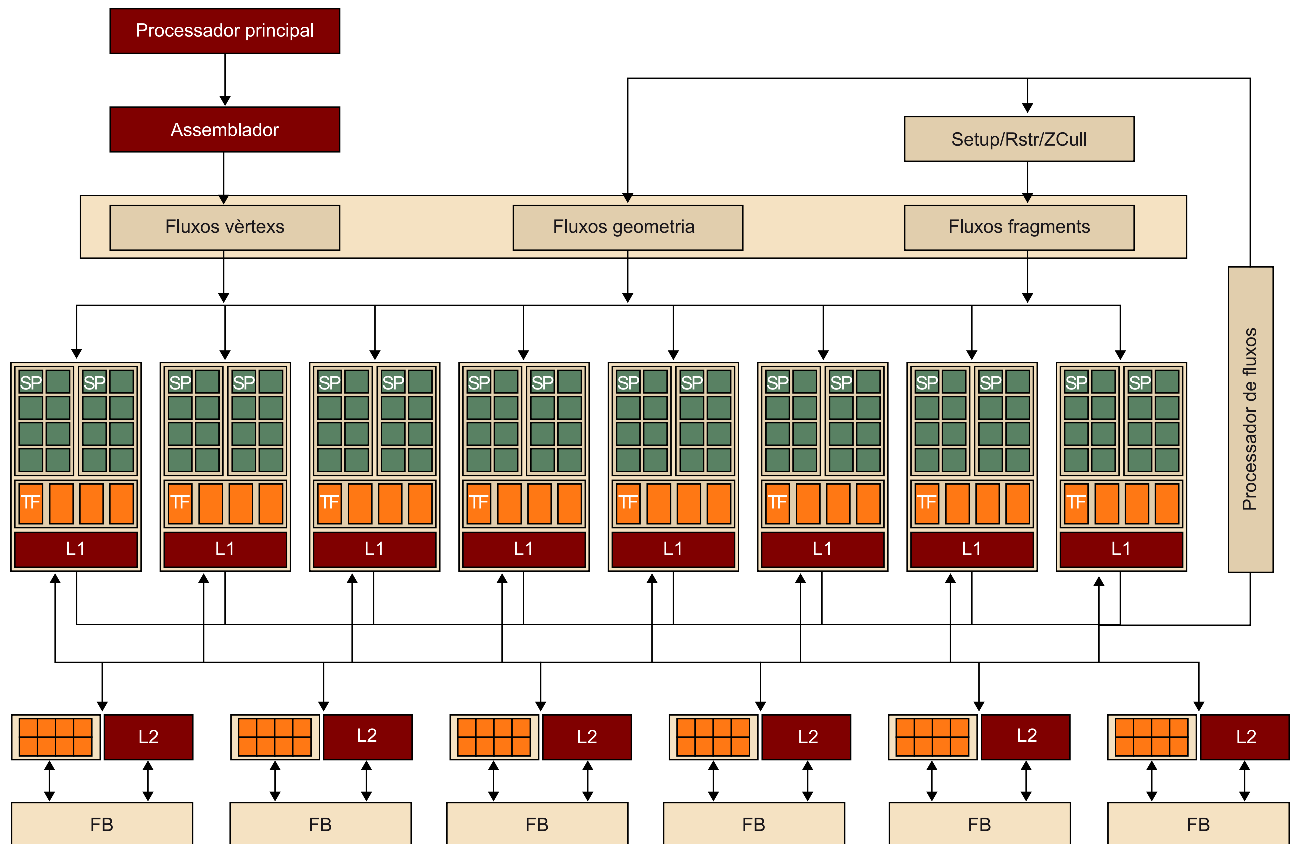
3.1. Arquitectura Nvidia

Nvidia va presentar al final del 2006 una nova línia de maquinari orientat a la computació general d'altres prestacions anomenada *Tesla*, que es va començar a desenvolupar a mitjan 2002. Tesla ofereix un maquinari d'altes prestacions (en forma, per exemple, de blocs de processadors gràfics) sense cap tipus d'orientació a aplicacions gràfiques. Tot i que no és la implementació de Tesla més actual, en aquest apartat ens centrarem en l'arquitectura G80, ja que va

representar un salt tecnològic important pel fet d'implementar una arquitectura unificada i, tal com es veurà al pròxim apartat, va venir amb el model de programació CUDA.

L'arquitectura G80 de Nvidia es defineix com una arquitectura totalment unificada, sense diferenciació a nivell de maquinari entre les diferents etapes que formen el *pipeline* gràfic, totalment orientada a l'execució massiva de fluxos, ajustant-se a l'estàndard IEEE 754. La figura 12 mostra l'esquema complet de l'arquitectura G80.

Figura 12. Arquitectura (unificada) de la sèrie G80 de Nvidia

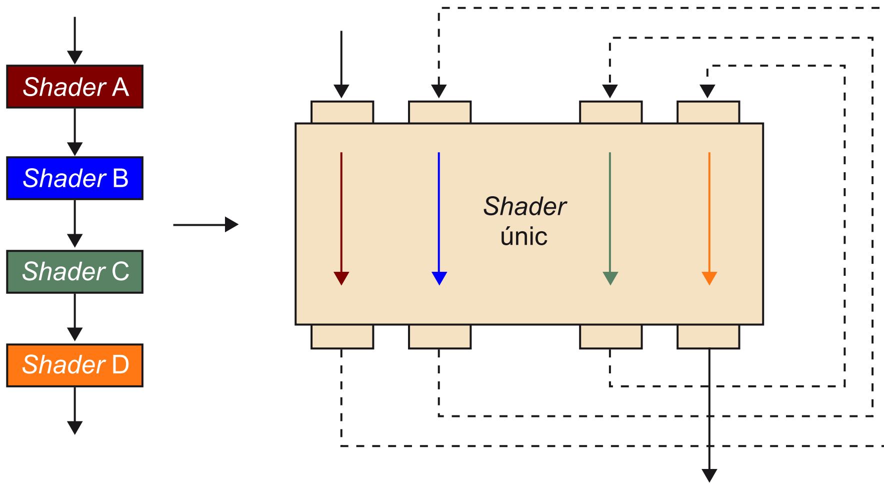


L'aparició de l'arquitectura G80 va representar una millora de la capacitat de processament gràfic i un increment de les prestacions respecte a la generació anterior de GPU, però la clau amb vista a l'àmbit de la computació general va ser la millora de la capacitat de càlcul en coma flotant. També es van afegir en el *pipeline* per tal de complir les característiques definides per Microsoft en DirectX 10.

Gràcies a l'arquitectura unificada, el nombre d'etapes del *pipeline* es redueix de manera significativa, i passa d'un model seqüencial a un model cíclic, com mostra la figura 13. El *pipeline* clàssic utilitza diferents tipus de *shaders* per mitjà dels quals les dades es processen seqüencialment. En canvi, en l'arquitectura unificada només hi ha una única unitat de *shaders* no especialitzats que processen les dades d'entrada (en forma de vèrtexs) per passos. La sortida d'un pas realimenta els *shaders* que poden executar un conjunt diferent d'instruccions,

emulant d'aquesta manera el *pipeline* clàssic, fins que les dades han passat per totes les etapes del *pipeline* i són encaminades cap a la sortida de la unitat de processament.

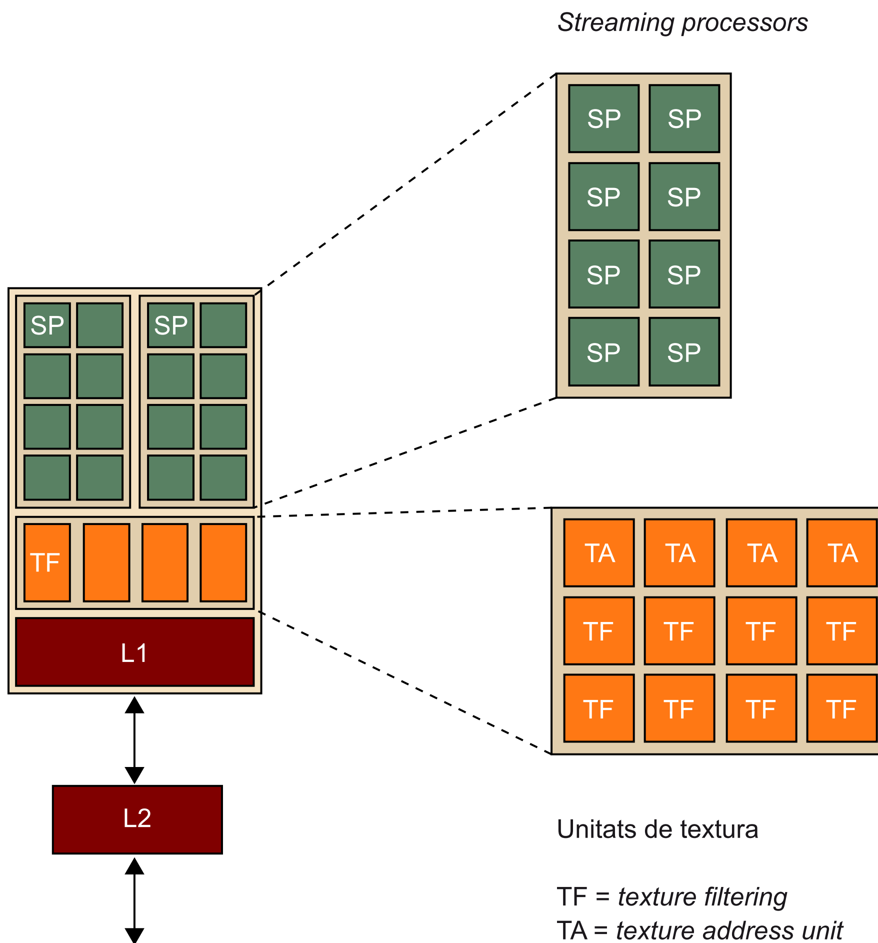
Figura 13. Comparació entre el *pipeline* seqüencial (esquerra) i el cíclic de l'arquitectura unificada (dreta)



El funcionament bàsic en l'arquitectura G80 per executar un programa consisteix en dues etapes diferenciades. En la primera etapa, les dades d'entrada es processen mitjançant maquinari especialitzat. Aquest s'encarrega de distribuir les dades de tal manera que es puguin utilitzar el màxim nombre d'unitats funcionals per tal d'obtenir la màxima capacitat de càlcul durant l'execució del programa. En la segona etapa, un controlador global de fluxos s'encarrega de controlar l'execució dels fluxos que fan els càlculs de manera coordinada. També determina en cada moment quins fluxos i de quin tipus (de vèrtexs, de fragments o de geometria) seran enviats a cada unitat de processament que componen la GPU. Les unitats de processament tenen un planificador de fluxos que s'encarrega de decidir la gestió interna dels fluxos i de les dades.

El nucli de processament dels *shaders* està format per 8 blocs de processament. Cadascun d'aquests blocs està format per 16 unitats de processament principal, anomenades *streaming processors* (SP). Tal com s'ha comentat, cada bloc té un planificador de fluxos però també memòria cau de nivell 1 (cau L1) i unitats d'accés i filtratge de textures pròpies. Així doncs, cada grup de 16 SP agrupats dins d'un mateix bloc comparteix unitats d'accés a textures i memòria cau L1. La figura 14 mostra de manera esquemàtica l'estructura dels blocs que formen l'arquitectura G80 de Nvidia.

Figura 14. *Streaming processors* i unitats de textura que componen un bloc en l'arquitectura G80 de Nvidia



Cada SP està dissenyat per a portar a terme operacions matemàtiques o bé adreçament de dades en memòria, i la transferència posterior. Cada processador és una ALU que opera sobre dades escalars de 32 bits de precisió (estàndard IEEE 754). Això contrasta amb el suport vectorial que oferien les arquitectures anteriors en els processadors de fragments.

Internament, cada bloc està organitzat en dos grups de 8 SP. El planificador intern planifica una mateixa instrucció sobre els dos subconjunts d'SP que formen el bloc, i cadascun pren un cert nombre de cicles de rellotge, que estarà definit pel tipus de flux que s'estigui executant en la unitat en aquell instant. A més de tenir accés al seu conjunt de registres propi, els blocs també poden accedir tant al conjunt de registres global com a dues zones de memòria addicionals, només de lectura, anomenades *memòria cau global de constants* i *memòria cau global de textures*.

Els blocs poden processar fluxos de tres tipus: de vèrtexs, de geometria i de fragments, de manera independent i en un mateix cicle de rellotge. També tenen unitats que permeten l'execució de fluxos dedicats exclusivament a la

transferència de dades en memòria: unitats TF (*texture filtering*) i TA (*texture addressing*), amb l'objectiu de solapar el màxim possible les transferències de dades a memòria amb la computació.

A més de poder accedir al seu conjunt propi de registres dedicat, al conjunt de registres global, a la memòria cau de constants i a la memòria cau de textures, els blocs poden compartir informació amb la resta de blocs per mitjà del segon nivell de memòria cau (L2), tot i que únicament en mode lectura. Per a compartir dades en mode lectura/escriptura és necessari l'ús de la memòria DRAM de vídeo, amb la penalització conseqüent que representa en el temps d'execució. En concret, els blocs tenen els tipus següents de memòria:

- Un conjunt de registres de 32 bits locals a cada bloc.
- Una memòria cau de dades paral·lela o memòria compartida, comuna per a tots els blocs, i que implementa l'espai de memòria compartida.
- Una memòria només de lectura anomenada *memòria cau de constants*, compartida per tots els blocs, que accelera les lectures a l'espai de memòria de constants.
- Una memòria només de lectura anomenada *memòria cau de textures*, compartida per tots els blocs, que accelera les lectures a l'espai de memòria de textures.

La memòria principal es divideix en sis particions, cadascuna de les quals proporciona una interfície de 64 bits, i s'aconsegueix així una interfície combinada de 384 bits. El tipus de memòria més utilitzat és GDDR. Amb aquesta configuració s'aconsegueixen velocitats de transferència de dades entre memòria i processador molt elevades. Això és una de les claus en la computació amb GPU, ja que l'accés a memòria és un dels principals colls d'ampolla.

L'amplada de banda de comunicació amb la CPU és de 8 GB/s: una aplicació CUDA pot transferir dades des de la memòria del sistema a 4GB/s a la vegada que pot enviar dades cap a la memòria del sistema també a 4GB/s. Aquesta amplada de banda és molt més reduïda que l'amplada de banda a memòria, la qual cosa pot semblar una limitació, però no ho és tant, ja que l'amplada de banda del bus PCI Express és comparable a la de la CPU a la memòria del sistema.

A més de les característiques exposades anteriorment cal tenir en compte altres millores respecte a arquitectures anteriors, com ara:

- Conjunt unificat d'instruccions.
- Més registres i constants utilitzables des d'un mateix *shader*.
- Nombre il·limitat d'instruccions per als *shaders*.
- Menys canvis d'estat, amb menys intervenció de la CPU.
- Possibilitat de recirculació de les dades entre diferents nivells del *pipeline*.
- Control del flux dinàmic tant a nivell de *shaders* com de vèrtexs i de píxels.

- Introducció d'operacions específiques sobre enters.

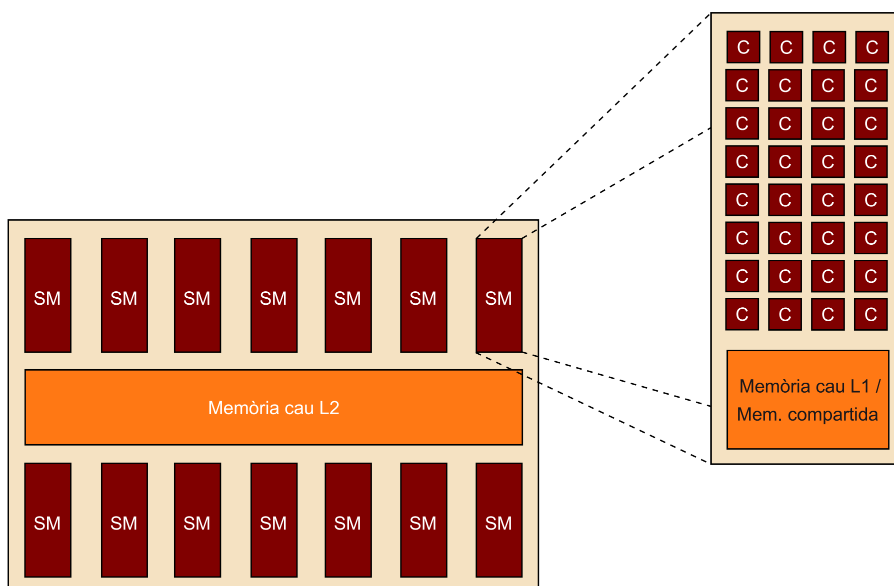
Implementacions de Tesla posteriors a la sèrie G80 ofereixen encara més prestacions, com ara un nombre més elevat d'SP, més memòria o amplada de banda. La taula 1 mostra les principals característiques d'algunes de les implementacions dins d'aquesta família de Nvidia.

Taula 1. Comparativa de diversos models de la família Tesla de Nvidia

Model	Arquitectura	Rellotge (MHz)	Nuclis		Memòria				Rendiment - pic teòric (GFLOP)
			Nombre SP	Rellotge (MHz)	Tipus	Mida (MB)	Rellotge (MHz)	Amplada de banda (GB/s)	
C870	**G80**	600	128	1.350	GDDR3	1.536	1.600	76	518
C1060	**GT200**	602	240	1.300	GDDR3	4.096	1.600	102	933
C2070	**GF100**	575	448	1.150	GDDR5	6.144	3.000	144	1.288
M2090	**GF110**	650	512	1.300	GDDR5	6.144	3.700	177	1.664

Les últimes generacions de Nvidia implementen l'arquitectura Fermi, que proporciona solucions cada cap més massives a nivell de paral·lelisme, tant des del punt de vista del nombre d'SP com de fluxos que es poden executar en paral·lel. La figura 15 mostra un esquema simplificat de l'arquitectura Fermi. En aquesta veiem com el dispositiu està format per un conjunt de *streaming multiprocessors* (SM) que comparteixen memòria cau L2. Cadascun dels SM està compost per 32 nuclis, cadascun dels quals pot executar una instrucció entera o en punt flotant per cicle de rellotge. A part de la memòria cau també inclou una interfície amb el processador principal, un planificador de fluxos i múltiples interfícies de DRAM.

Figura 15. Esquema simplificat de l'arquitectura Fermi



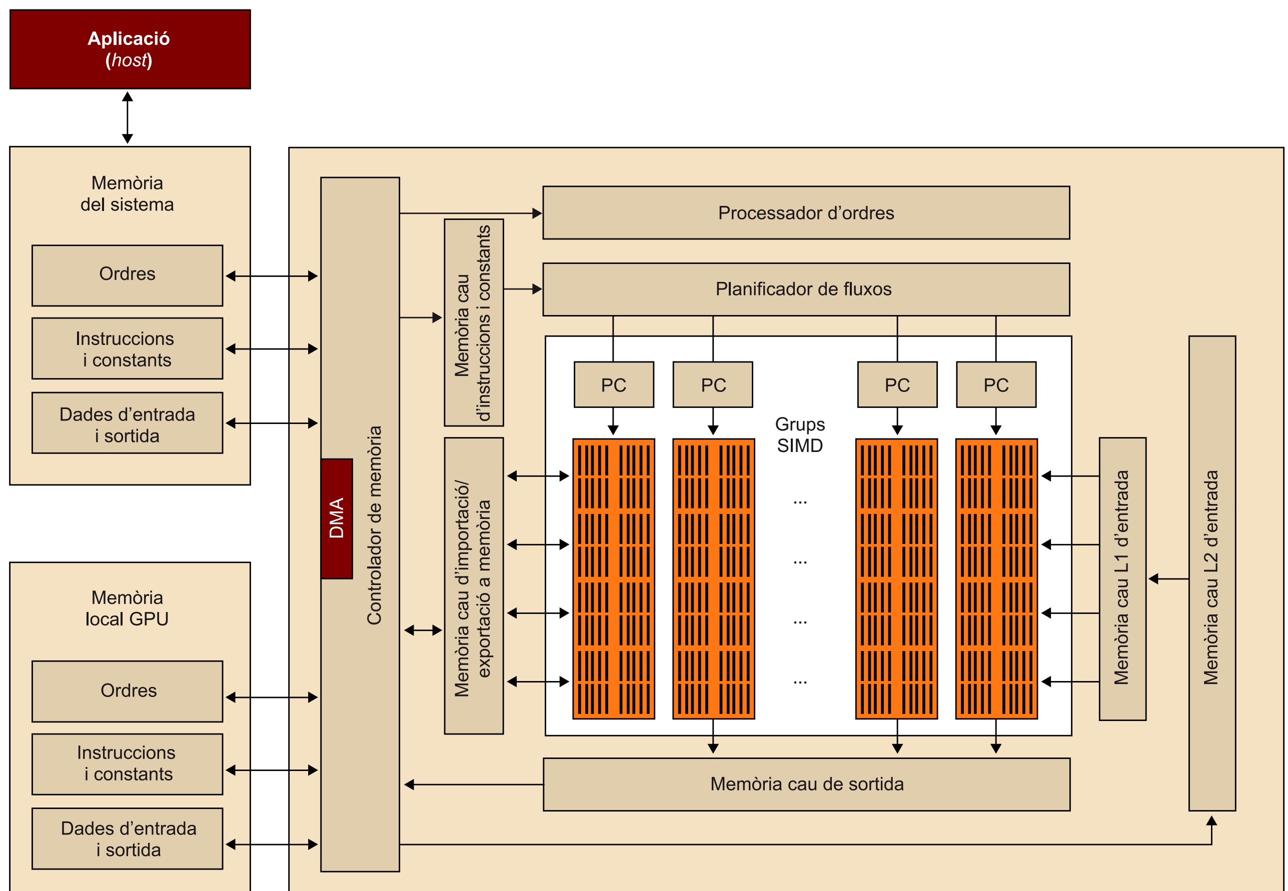
Fermi proporciona un nivell d'abstracció en què la GPU està composta d'un conjunt uniforme d'unitats computacionals amb només uns pocs elements que li donen suport. Així doncs, l'objectiu principal d'aquest disseny és el de dedicar la major part de la superfície del xip i del corrent elèctric a l'aplicació en qüestió i maximitzar així el rendiment en coma flotant.

3.2. Arquitectura AMD (ATI)

De manera similar a la tecnologia de Nvidia, la tecnologia GPU desenvolupada per AMD/ATI ha evolucionat molt ràpidament en la darrera dècada envers la computació GPGPU. En aquest subapartat estudiarem les principals característiques de l'arquitectura de la sèrie R600 d'AMD (que és l'equivalent a la Nvidia G80) i l'arquitectura CU (implementada per la família Evergreen d'AMD), que és un desenvolupament nou i està associada al model de programació OpenCL.

La sèrie R600 d'AMD implementa una arquitectura unificada com en el cas de la Nvidia G80. També inclou un *shader* de geometria que permet fer operacions de creació de geometria en la GPU i així no haver de sobrecarregar la CPU. La sèrie R600 incorpora 320 SP, que són molts més que els 128 de la G80. Això no vol dir, però, que sigui molt més potent, ja que els SP de les dues arquitectures funcionen de manera força diferent. La figura 16 mostra l'esquema de l'arquitectura de la sèrie R600 d'AMD.

Figura 16. Diagrama de blocs de l'arquitectura de la sèrie R600 d'AMD



Els SP en l'arquitectura R600 estan organitzats en grups de 5 (grups SIMD), dels quals només un pot fer algunes operacions més complexes (per exemple operacions en coma flotant de 32 bits), mentre que els altres 4 estan dedicats només a fer operacions simples amb enters. Aquesta arquitectura SIMD hibrida també es coneix com a VLIW-5D. Com que els SP estan organitzats d'aquesta manera, l'arquitectura R600 només pot executar 64 fluxos, quan podríem pensar que en podria suportar fins a 320. Des d'una perspectiva optimista aquests 64 fluxos podrien executar 5 instruccions en cada cicle de rellotge, però cal tenir en compte que cadascuna d'aquestes instruccions ha de ser completament independent de les altres. Així doncs, l'arquitectura R600 deixa gran responsabilitat al planificador de fluxos que, a més, ha de gestionar una quantitat molt gran de fluxos. El planificador de fluxos s'encarrega de seleccionar on cal executar cada flux mitjançant una sèrie d'àrbitres, dos per a cada matriu de 16 grups SIMD. Una altra tasca que desenvolupa el planificador de fluxos és determinar la urgència de l'execució d'un flux. Això vol dir que el planificador de fluxos pot assignar a un flux un grup SIMD ocupat, mantenir les dades que aquest grup SIMD estava utilitzant i, un cop el flux prioritari ha finalitzat, continuar executant el flux original amb tota normalitat.

Les diferències entre els SP de les arquitectures Nvidia i AMD també inclouen la freqüència de rellotge a la qual treballen. Mentre que els SP de la implementació de Nvidia utilitzen un domini específic de rellotge que funciona a una velocitat més elevada que la resta dels elements, en la implementació d'AMD els SP utilitzen la mateixa freqüència de rellotge que la resta d'elements, i no tenen el concepte de domini de rellotge.

La taula 2 mostra les principals característiques de les arquitectures AMD orientades a GPGPU. Noteu que el nombre d'SP és molt més elevat que en les Nvidia, però també hi ha diferències importants respecte a la freqüència de rellotge i amplada de banda a memòria.

Taula 2. Comparativa de diversos models de GPU AMD

Model	Arquitectura (família)	Nuclis		Memòria				Rendiment - pic teòric (GFLOP)
		Nombre SP	Rellotge (MHz)	Tipus	Mida (MB)	Rellotge (MHz)	Amplada de banda (GB/s)	
R580	**X1000**	48	600	GDDR3	1.024	650	83	375
RV670	**R600**	320	800	GDDR3	2.048	800	51	512
RV770	**R700**	800	750	GDDR5	2.048	850	108	1.200
Cypress (RV870)	**Evergreen**	1.600	825	GDDR5	4.096	1.150	147	2.640

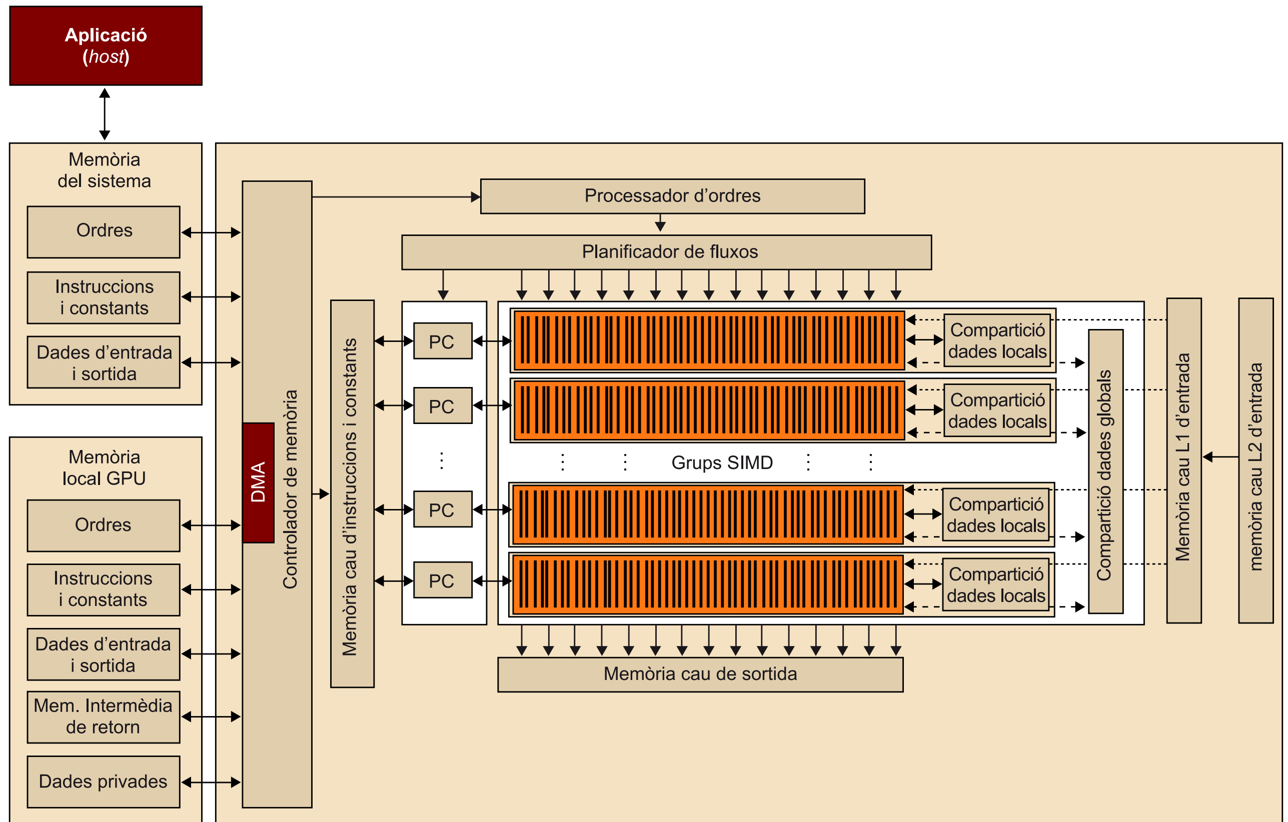
3.2.1. Arquitectura AMD CU

L'adquisició d'ATI per part d'AMD el 2006 (que va implicar la supressió de la marca ATI dels seus productes) va ser una peça important dintre de l'estratègia d'AMD per desenvolupar una generació de processadors que integressin capacitats per a computació de propòsit general i de funcions gràfiques en un mateix xip. Així doncs, AMD va fer grans esforços en desenvolupar una nova arquitectura que substituís la basada en VLIW-5D per una altra de més homogènia. Aquesta arquitectura es va anomenar CU (*Compute Unit*) i té com a objectiu simplificar el model de programació per tal d'encoratjar els programadors a la utilització de GPGPU. Tal com veurem en el pròxim apartat, aquesta arquitectura ve amb model de programació OpenCL.

L'arquitectura CU és una arquitectura orientada a la computació multiflux intensiu però sense deixar d'oferir gran rendiment per a la computació gràfica. Està basada en un disseny escalar (similar a Nvidia Fermi) amb administració de recursos fora d'ordre i enfocats a l'execució simultània de diferents tipus d'instruccions independents (gràfiques, textures, vídeo i còmput general). Així doncs, ofereix més potència, latències més petites i, sobretot molta més flexibilitat als programadors. El nou nucli de *shaders* unificat abandona el disseny vectorial VLIW i, en el seu lloc, s'utilitza el nou CU, que està format per 4 unitats SIMD escalars.

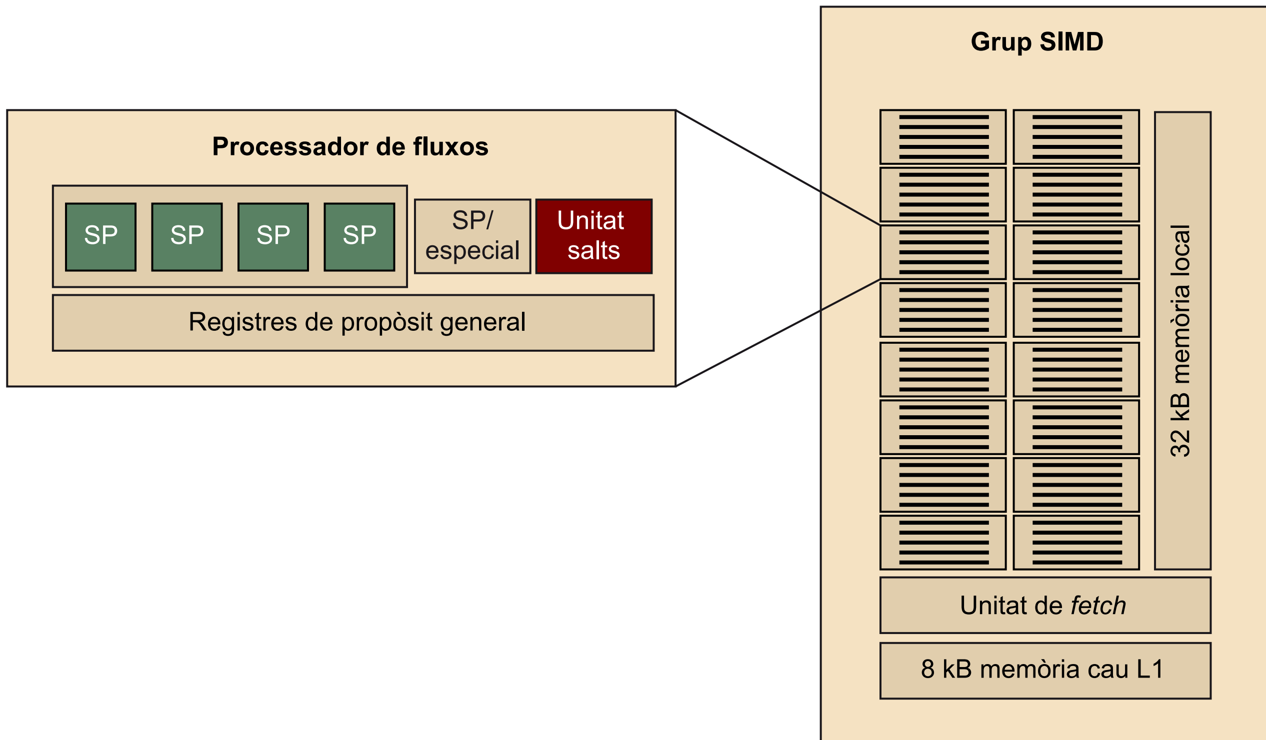
Una implementació d'aquest tipus d'arquitectura es troba en la família Evergreen. La figura 17 mostra un diagrama de blocs de l'arquitectura Evergreen. Aquesta està composta per un conjunt de grups SIMD els quals, al mateix temps, estan compostos de 16 unitats de fluxos, cadascuna amb 5 nuclis, tal com mostra el detall de la figura 18.

Figura 17. Diagrama de blocs de l'arquitectura Evergreen d'AMD



Tenint en compte l'exemple d'arquitectura Evergreen de la taula 2, podem comptar 20 grups SIMD, cadascun amb 16 unitats de fluxos, que estan formats per 5 nuclis. Tot plegat suma un total de 16.000 nuclis, la gestió eficient dels quals és un repte important tant des del punt de vista del maquinari com a l'hora de programar les aplicacions (les quals han de proporcionar el nivell de paral·lelisme suficient).

Figura 18. Diagrama de blocs dels grups SIMD de l'arquitectura Evergreen d'AMD



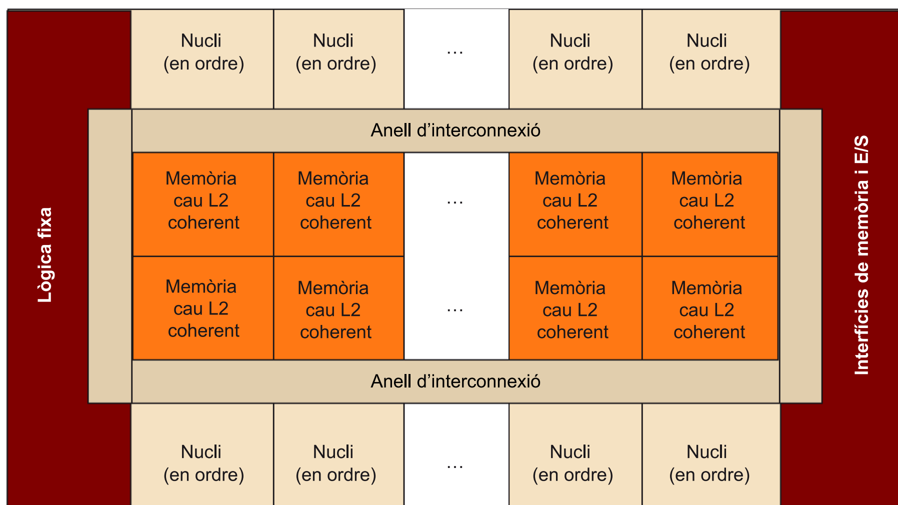
Si considerem el cas anterior, amb un grup de 4 nuclis podríem executar, per exemple, en un únic cicle de rellotge:

- 4 operacions MUL+ADD (fusionades) de 32 bits en coma flotant.
- 2 operacions MUL o ADD de 64 bits de doble precisió.
- 1 operació MUL+ADD (fusionades) de 64 bits de doble precisió.
- 4 operacions MUL o ADD+ enteres de 24 bits.

3.3. Arquitectura Intel Larrabee

Larrabee és una arquitectura altament paral·lela basada en nuclis que implementen una versió estesa de x86 amb operacions vectorials (SIMD) i algunes instruccions escalars especialitzades. La figura 19 mostra un esquema d'aquesta arquitectura. Una de les principals característiques és que la memòria cau de nivell 2 (L2) és coherent entre tots els nuclis.

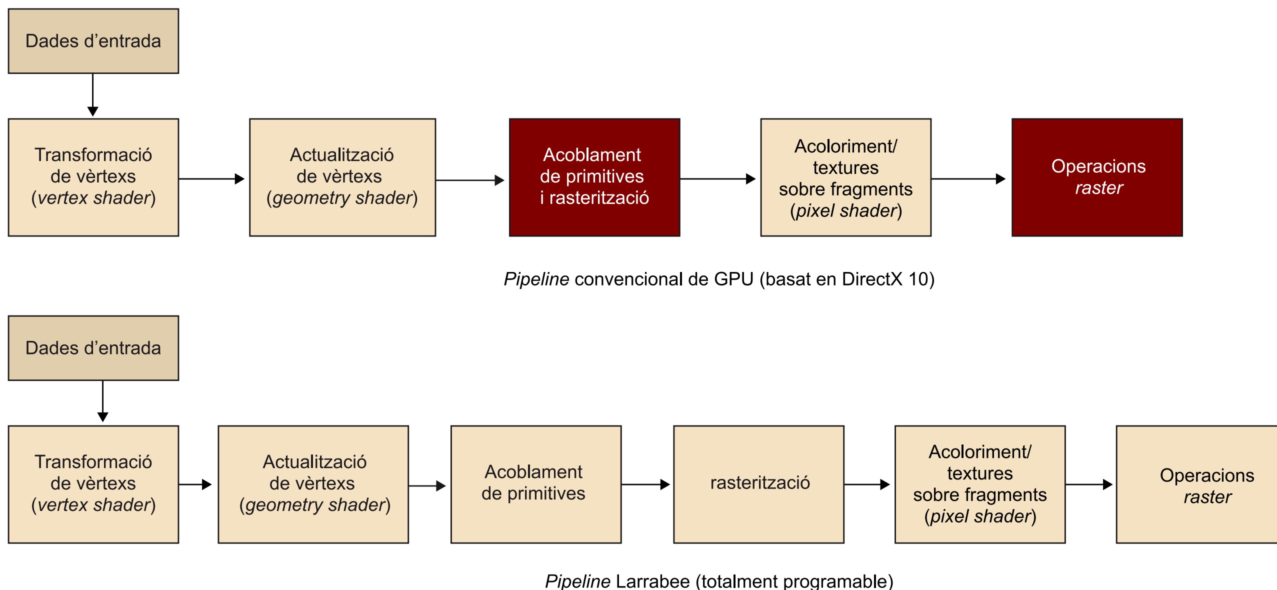
Figura 19. Esquema de l'arquitectura Larrabee



La quantitat de nuclis i la quantitat i tipus de coprocessadors i blocs d'E/S són dependents de la implementació.

El fet de disposar de coherència de memòria cau i d'una arquitectura tipus x86 fa que aquesta arquitectura sigui més flexible que altres arquitectures basades en GPU. Entre d'altres qüestions, el *pipeline* gràfic de l'arquitectura Larrabee és totalment programable, tal com es mostra a la figura 20.

Figura 20. Comparativa entre el *pipeline* convencional de GPU i el de l'arquitectura Larrabee



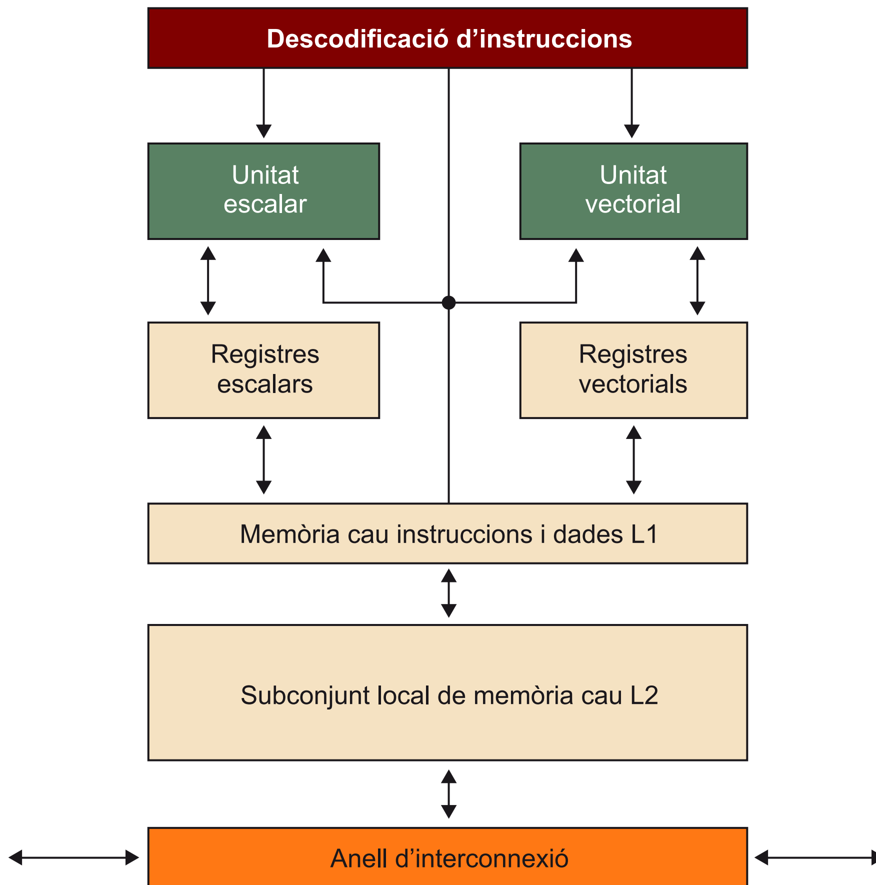
Els elements no enfosquits de la figura són les unitats programables.

Els diferents nuclis es comuniquen mitjançant una xarxa d'interconnexió d'una alta amplada de banda amb lògica fixa, interfícies d'entrada/sortida (E/S) i d'alguna altra lògica d'E/S segons la implementació concreta (per exemple, una implementació com a GPU incorporaria suport per a bus PCIe).

La figura 21 mostra un esquema d'un nucli amb la interconnexió de xarxa del xip mateix associada i el seu subconjunt de memòria cau L2. El descodificador d'instruccions suporta el conjunt d'instruccions x86 del processador Pentium estàndard i algunes altres instruccions noves. Per tal de simplificar el disseny, les unitats escalar i vectorial utilitzen diferents conjunts de registres. Les dades

que es transfereixen entre aquestes unitats s'escriuen a memòria i després es llegeixen un altre cop de la memòria cau L1. La memòria cau L1 permet accessos molt ràpids des de les unitats escalar i vectorial; per tant, d'alguna manera la memòria cau L1 es pot veure com una espècie de conjunt de registres.

Figura 21. Diagrama de bloc dels nuclis de l'arquitectura Larrabee



La memòria cau L2 es divideix en diferents subconjunts independents, un per a cada nucli. Cada nucli té un camí d'accés ràpid i directe al seu subconjunt local de la memòria cau L2. Les dades que es llegeixen en un nucli s'emmagatzemen en el seu subconjunt de memòria cau L2 i s'hi pot accedir a la vegada que altres nuclis accedeixen al seu subconjunt de memòria cau L2. Les dades que escriu un nucli s'emmagatzemen a la memòria cau L2 i, si és necessari, també s'actualitzen altres subconjunts de memòria cau L2 per mantenir-ne la consistència. La xarxa en forma d'anell garanteix la coherència per a dades compartides. La mida de la memòria cau L2 és força considerable (per exemple, entorn de 256 kB) cosa que fa que el processament gràfic es pugui aplicar a un conjunt de dades prou gran. Amb vista al processament de propòsit general, aquesta característica és especialment positiva, ja que facilita la implementació eficient de certes aplicacions.

El *pipeline* escalar de l'arquitectura Larrabee es deriva del processador Pentium P54C, el qual disposa d'un *pipeline* d'execució força senzill. Tot i això, l'arquitectura Larrabee té algunes característiques més avançades que el P54C, com per exemple el suport per a multiflux, extensions de 64 bits i un *prefetc-*

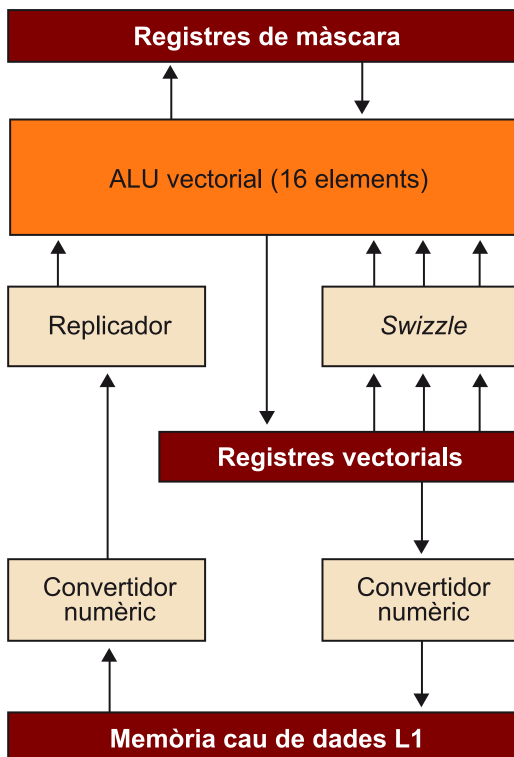
hing més sofisticat. Els nuclis també disposen d'algunes instruccions escalars afegides, però són compatibles amb tot el conjunt d'instruccions del Pentium x86; per tant, poden executar codis existents, com per exemple nuclis de sistemes operatius i aplicacions.

També s'afegeixen algunes instruccions i modalitats noves per a permetre el control de la memòria cau de manera explícita. Per exemple, inclou instruccions específiques per a fer *prefetching* de dades a la memòria cau L1 o L2 i també modalitats per a reduir la prioritat d'una línia de memòria cau.

Sincronitzar l'accés de diversos fluxos d'un mateix nucli a memòria compartida no és massa costós. Els fluxos d'un mateix nucli comparteixen la mateixa memòria cau local L1; per tant, només cal fer una simple lectura a un semàfor atòmic per a implementar la sincronització. Sincronitzar l'accés entre diferents nuclis és una mica més costós, ja que requereix *locks* entre nuclis (la qual cosa és un problema típic en el disseny de multiprocessadors). Cal tenir en compte que Larrabee suporta 4 fluxos d'execució amb conjunts de registres independents per a cada flux.

L'arquitectura Larrabee disposa d'una unitat vectorial (VPU, *vector processing unit*) d'elements que executa instruccions tant amb enters com en coma flotant de precisió simple i doble. Aquesta unitat vectorial juntament amb els seus registres ocupa aproximadament un terç de la superfície del processador però proporciona millor rendiment amb enters i coma flotant. La figura 22 mostra el diagrama de blocs de la VPU amb la memòria cau L1.

Figura 22. Diagrama de bloc de la unitat vectorial de l'arquitectura Larrabee



Les instruccions vectorials permeten fins a tres operands d'origen, dels quals un pot venir directament de la memòria cau L1. Tal com hem vist anteriorment, si les dades ja són a la memòria cau, llavors la memòria cau L1 és com un conjunt de registres addicionals.

L'etapa següent consisteix a alinear les dades dels registres i memòria amb les línies corresponents de la VPU (*swizzle*). Aquesta és una operació típica tant en el procés de dades gràfiques com no gràfiques per tal d'augmentar l'eficiència de la memòria cau. També implementa conversió numèrica mitjançant els mòduls corresponents. La VPU disposa d'un ampli ventall d'instruccions tant enteres com en coma flotant. Un exemple és l'operació aritmètica estàndard conjunta de multiplicació i suma (MUL+ADD).

L'arquitectura Larrabee utilitza un anell bidireccional que permet comunicar elements com ara els nuclis, la memòria cau L2 i altres blocs lògics entre ells dins d'un mateix xip. Quan s'utilitzen més de 16 nuclis llavors s'utilitzen diversos anells més petits. L'amplada de dades de l'anell és de 512 bits per sentit.

La memòria cau L2 està dissenyada per a proporcionar a cada nucli una amplada de banda molt gran a adreces de memòria que altres nuclis no poden escriure, mitjançant el subconjunt local de memòria cau L2 del nucli. Cada nucli pot accedir al seu subconjunt de la memòria cau L2 en paral·lel sense comunicar-se amb altres nuclis. Tot i així, abans d'assignar una nova línia a la memòria cau L2 s'utilitza l'anell per a comprovar que no hi hagi compartició de dades per tal de mantenir la coherència d'aquestes. L'anell d'interconnexió també proporciona a la memòria cau L2 l'accés a memòria.

4. Models de programació per a GPGPU

En apartats anteriors hem vist que hi ha interfícies d'usuari per a la programació gràfica, com ara OpenGL o Direct3D, i que les GPU poden ser utilitzades per a computació de propòsit general, tot i que amb certes limitacions. Ateses les característiques i l'evolució de les GPU, els programadors cada vegada necessiten fer front a una varietat de plataformes de computació massivament paral·leles i la utilització de models de programació estàndard és crucial.

En aquest apartat estudiarem els principals models de programació per a GPU orientats a aplicacions de propòsit general, específicament CUDA i OpenCL, que són els estàndards actuals per a GPGPU.

4.1. CUDA

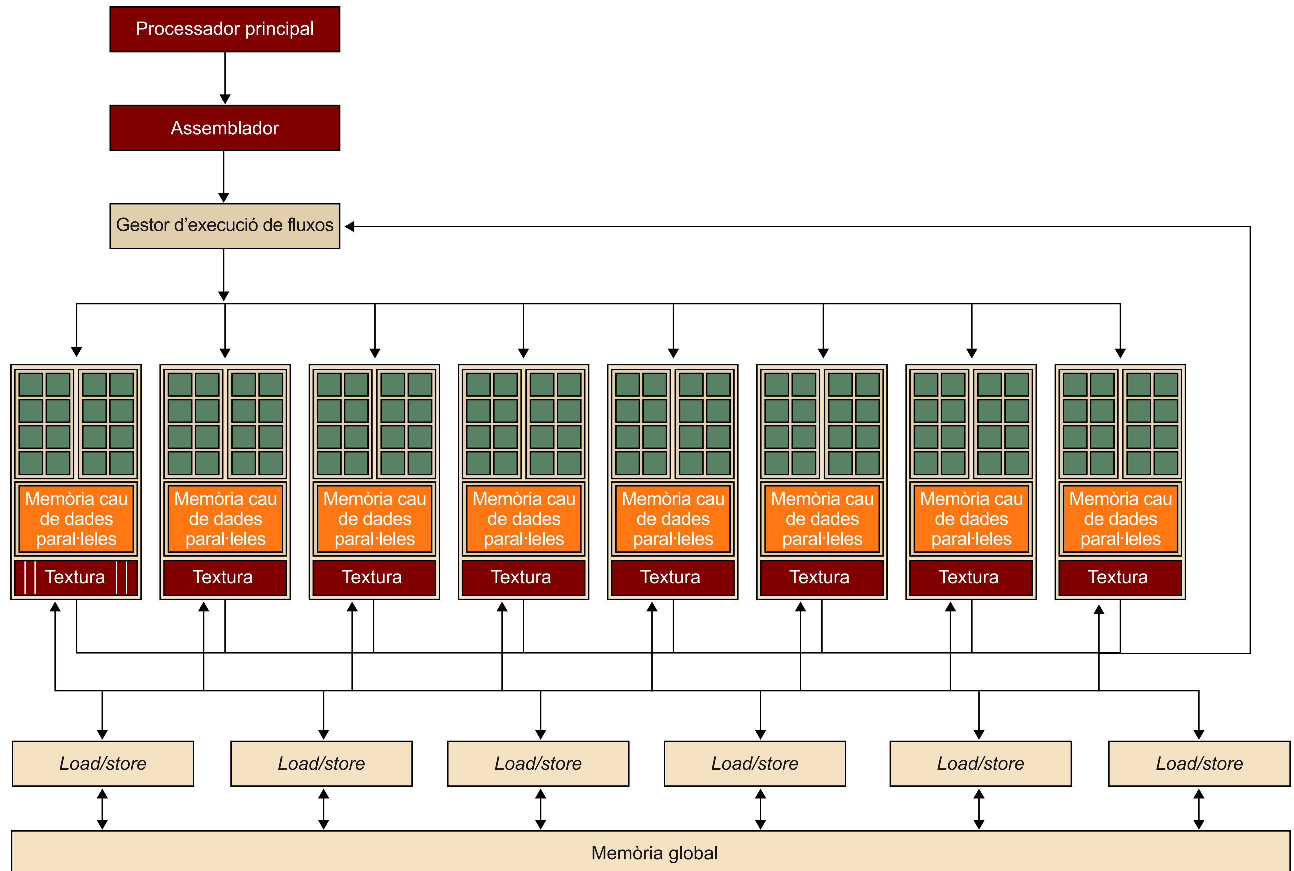
CUDA (*compute unified device architecture*) és una especificació inicialment de propietat desenvolupada per Nvidia com a plataforma per als seus productes GPU. CUDA inclou les especificacions de l'arquitectura i un model de programació associat. En aquest apartat estudiarem el model de programació CUDA; no obstant això, parlarem de dispositius compatibles amb CUDA per al cas d'aquelles GPU que implementen l'arquitectura i especificacions definides en CUDA.

4.1.1. Arquitectura compatible amb CUDA

La figura 23 mostra l'arquitectura d'una GPU genèrica compatible amb CUDA. Aquesta arquitectura està organitzada en una sèrie de multiprocessadors o *streaming multiprocessors* (SM), els quals tenen una quantitat elevada de fluxos d'execució. En la figura, dos SM formen un bloc, tot i que el nombre d'SM per bloc depèn de la implementació concreta del dispositiu. A més, cada SM de la figura té un nombre de *streaming processors* (SP), que comparteixen la lògica de control i memòria cau d'instruccions.

La GPU té una memòria DRAM de tipus GDDR (*graphics double data rate*), la qual està indicada en la figura 23 com a memòria global. Aquesta memòria GDDR es diferencia de la memòria DRAM de la placa base del computador en el fet que essencialment s'utilitza per a gràfics (*framebuffer*). Per a aplicacions gràfiques aquesta manté les imatges de vídeo i la informació de les textures. En canvi, per a càlculs de propòsit general aquesta funciona com a memòria externa amb molta amplada de banda però amb una latència una mica més elevada que la memòria típica del sistema. Tot i així, per a aplicacions massivament paral·leles l'amplada de banda més gran compensa la latència més elevada.

Figura 23. Esquema de l'arquitectura d'una GPU genèrica compatible amb CUDA



Noteu que aquesta arquitectura genèrica és molt similar a la G80 descrita anteriorment ja que la G80 la implementa. L'arquitectura G80 suporta fins a 768 fluxos per SM, la qual cosa suma un total de 12.000 fluxos en un únic xip.

L'arquitectura GT200, posterior a la G80, té 240 SP i supera el TFlop de pic teòric de rendiment. Com que els SP són massivament paral·lels es poden arribar a utilitzar encara més fluxos per aplicació que la G80. La GT200 suporta 1.024 fluxos per SM i en total suma entorn de 30.000 fluxos per xip. Per tant, la tendència mostra clarament que el nivell de paral·lelisme suportat per les GPU està augmentant ràpidament. Serà molt important, doncs, intentar explotar aquest nivell tan elevat de paral·lelisme quan es desenvolupin aplicacions de propòsit general per a GPU.

4.1.2. Entorn de programació

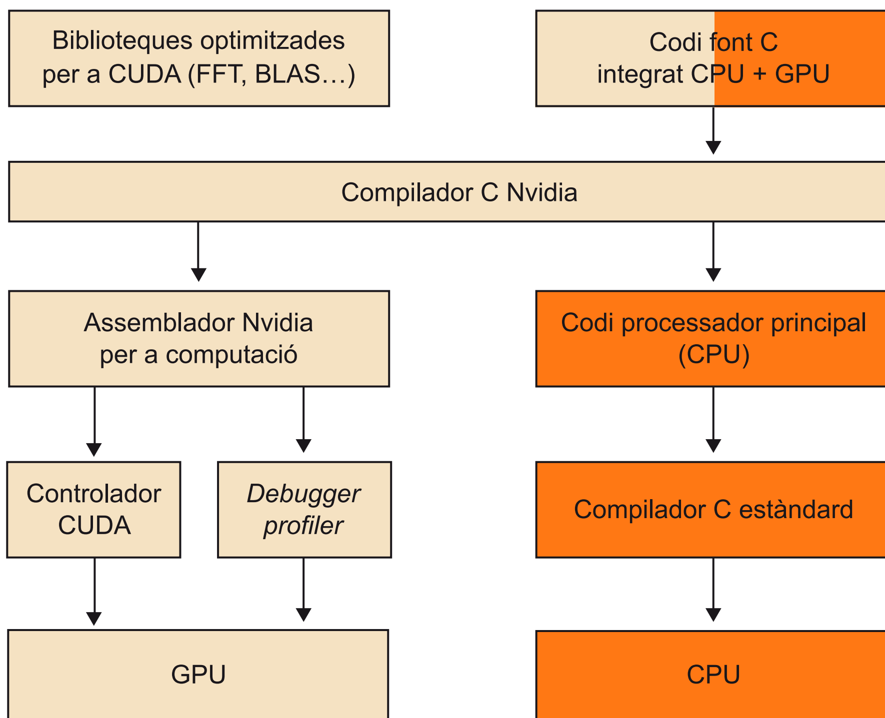
CUDA es va desenvolupar per tal d'augmentar la productivitat en el desenvolupament d'aplicacions de propòsit general per a GPU. Des del punt de vista del programador el sistema està compost per un processador principal (*host*), que és una CPU tradicional, com per exemple un processador d'arquitectura Intel, i d'un o més dispositius (*devices*), que són GPU.

CUDA pertany al model SIMD; per tant, està pensat per a explotar el paral·lelisme a nivell de dades. Això vol dir que un conjunt d'operacions aritmètiques es poden executar sobre un conjunt de dades de manera simultània. Afortunadament, moltes aplicacions tenen parts amb un nivell molt elevat de paral·lelisme a nivell de dades.

Un programa en CUDA consisteix en una o més fases que poden ser executades o bé en el processador principal (CPU) o bé en el dispositiu GPU. Les fases en les quals hi ha molt poc o gens de paral·lelisme a nivell de dades s'implementen en el codi que s'executarà en el processador principal, i les fases amb un nivell de paral·lelisme a nivell de dades elevat s'implementen en el codi que s'executarà al dispositiu.

Tal com mostra la figura 24, el compilador de NVIDIA (`nvcc`) s'encarrega de proporcionar la part del codi corresponent al processador principal i al dispositiu durant el procés de compilació. El codi corresponent al processador principal és simplement codi ANSI C que es compila mitjançant el compilador de C estàndard del processador principal, com si fos un programa per a CPU convencional. El codi corresponent al dispositiu també és ANSI C però amb extensions que inclouen paraules clau per a poder definir funcions que tracten les dades en paral·lel. Aquestes funcions s'anomenen ***kernels***.

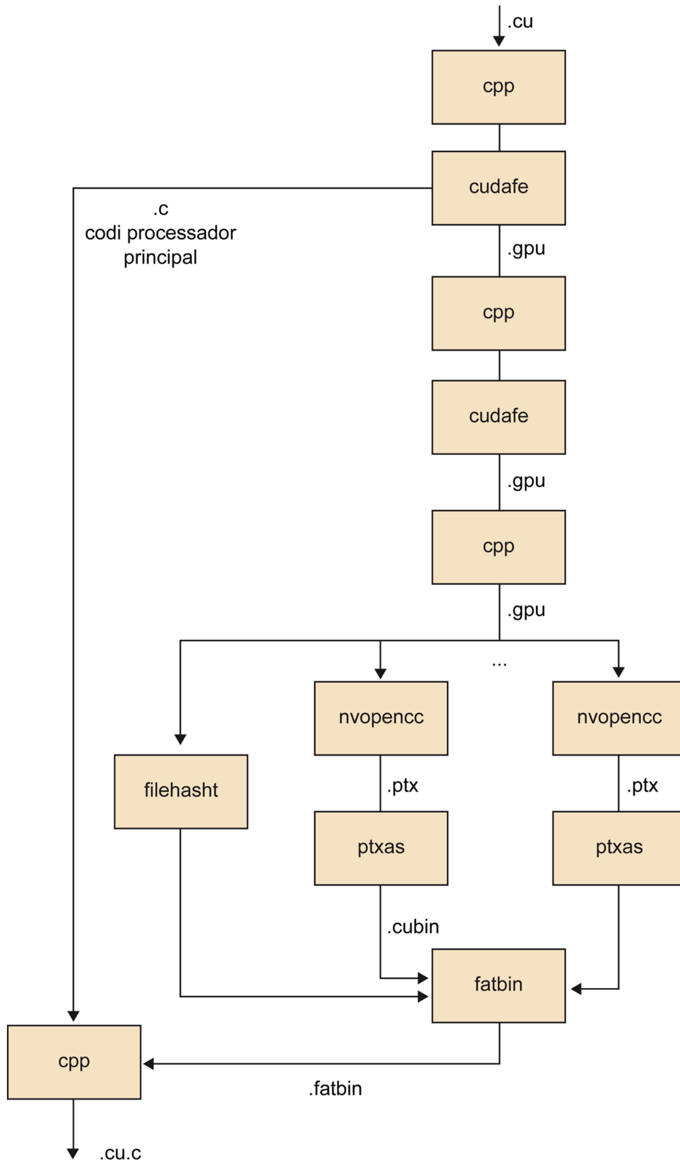
Figura 24. Esquema de blocs de l'entorn de compilació de CUDA



El codi del dispositiu es torna a compilar amb `nvcc` i llavors ja es pot executar al dispositiu GPU. En situacions en què no hi ha cap dispositiu disponible o bé el *kernel* és més apropiat per a una CPU, també es poden executar els *kernels* en una CPU convencional mitjançant eines d'emulació que proporciona la

plataforma CUDA. La figura 25 mostra totes les etapes del procés de compilació i la taula 3 descriu com el compilador `nvcc` interpreta els diferents tipus de fitxers d'entrada.

Figura 25. Passos en la compilació de codi CUDA, des de *.cu* fins a *.cu.c*



Taula 3. Interpretació de `nvcc` dels fitxers d'entrada

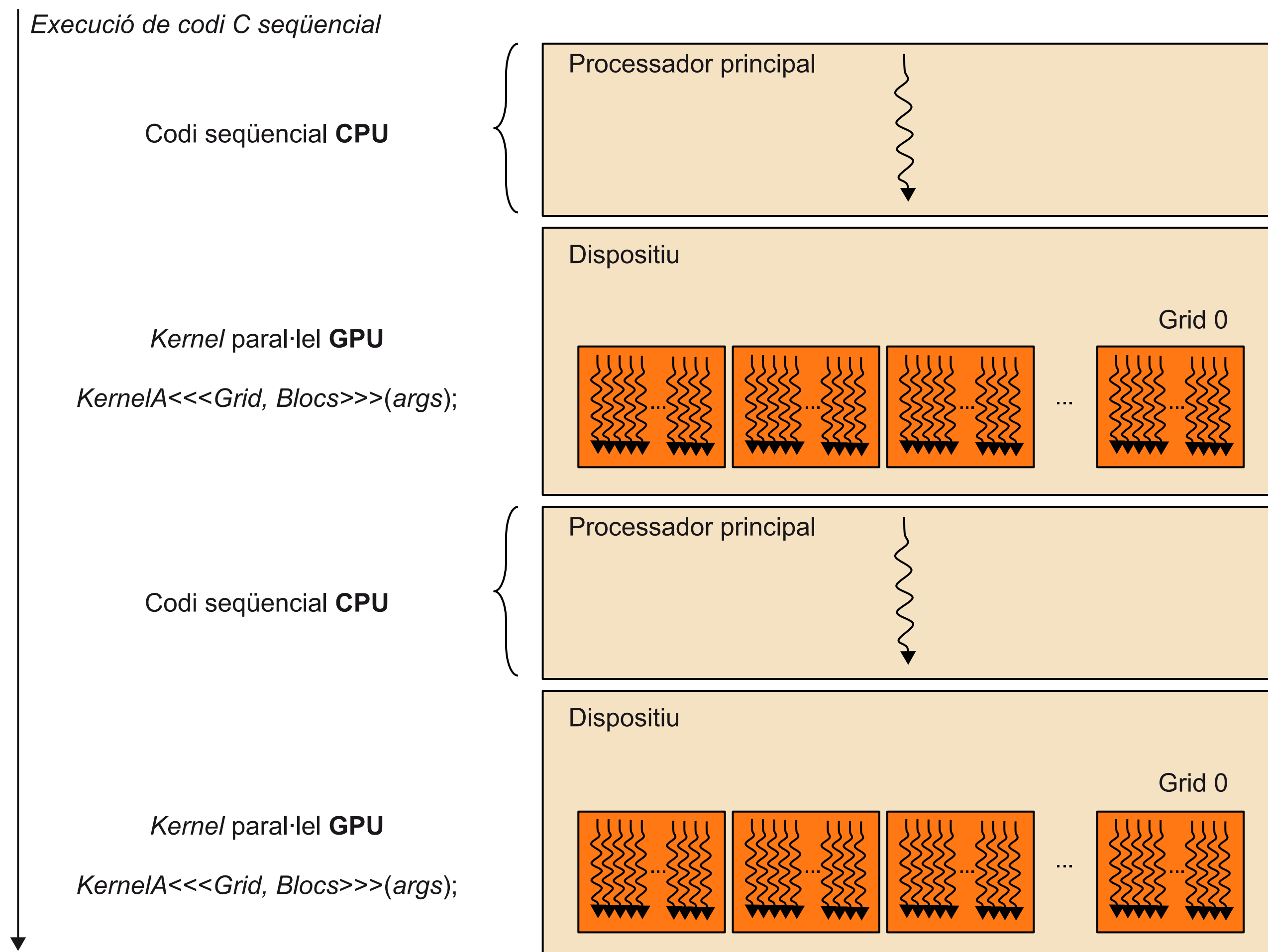
.cu	Codi font CUDA que conté tant el codi del processador principal com les funcions del dispositiu
.cup	Codi font CUDA preprocessat que conté tant el codi del processador principal com les funcions del dispositiu
.c	Fitxer de codi font C
.cc, .cxx, .cpp	Fitxer de codi font C++
.gpu	Fitxer intermedi *gpu*
.ptx	Fitxer assemblador intermedi *ptx*
.o, .obj	Fitxer d'objecte

.a, .lib	Fitxer de biblioteca
.res	Fitxer de recurs
.so	Fitxer d'objecte compartit

Per tal d'explotar el paral·lelisme a nivell de dades, els *kernels* han de generar una quantitat de fluxos d'execució força elevada (per exemple, entorn de desenes o centenars de milers de fluxos). Hem de tenir en compte que **els fluxos de CUDA són molt més lleugers que els fluxos de CPU**. De fet, podrem assumir que per a generar i planificar aquests fluxos només necessitarem uns pocs cicles de rellotge a causa del suport de maquinari, en contrast dels fluxos convencionals per a CPU, que normalment requereixen milers de cicles.

L'execució d'un programa típic CUDA es mostra en la figura 26. L'execució comença amb execució al processador principal (CPU). Quan s'invoca un *kernel*, l'execució es mou cap al dispositiu (GPU), on es generen un nombre molt elevat de fluxos. El conjunt de tots aquest fluxos que es generen quan s'invoca un *kernel* s'anomena *grid*. En la figura 26 es mostren dos *grids* de fluxos. La definició i organització d'aquests *grids* els estudiarem més endavant. Un *kernel* finalitza quan tots els seus fluxos finalitzen l'execució al *grid* corresponent. Un cop finalitzat el *kernel*, l'execució del programa continua al processador principal fins que s'invoca un altre *kernel*.

Figura 26. Etapes en l'execució d'un programa típic CUDA



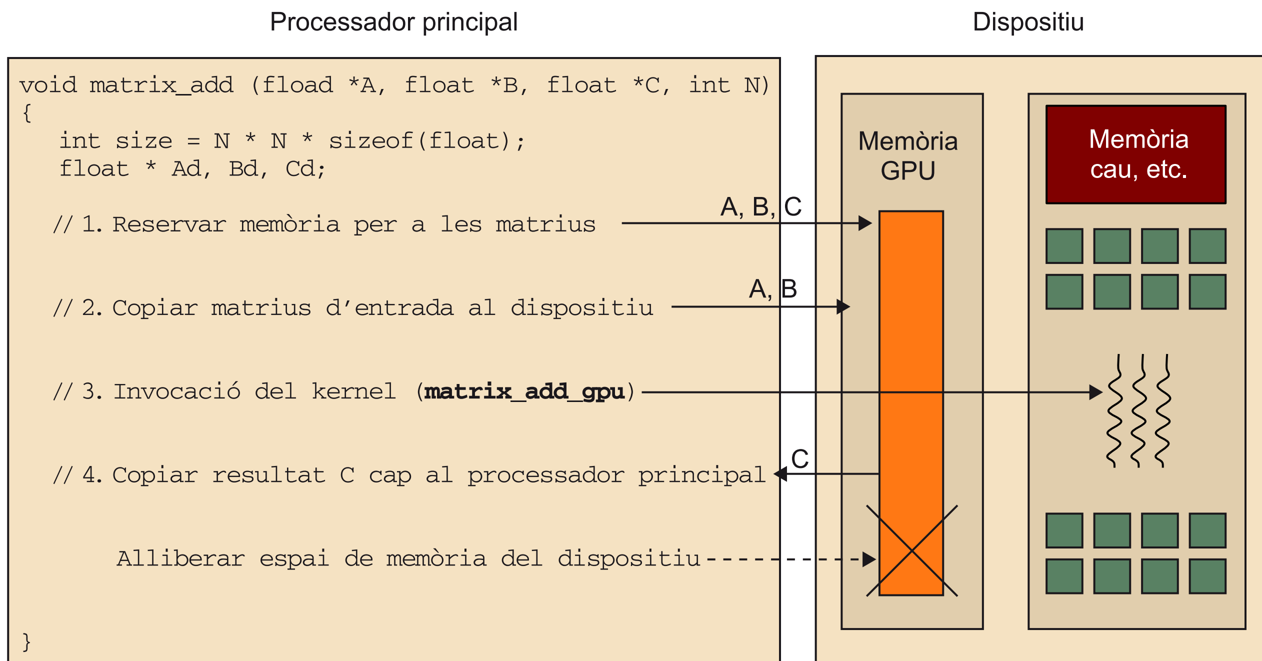
4.1.3. Model de memòria

És important remarcar que **la memòria del processador principal i del dispositiu són espais de memòria completament separats**. Això reflecteix la realitat que els dispositius són típicament targetes que tenen la seva pròpia memòria DRAM. Per tal d'executar un *kernel* al dispositiu GPU normalment cal seguir els passos següents:

- Reservar memòria al dispositiu (pas 1 de la figura 27).
- Transferir les dades necessàries des del processador principal a l'espai de memòria assignat al dispositiu (pas 2 de la figura 27).
- Invocar l'execució del *kernel* en qüestió (pas 3 de la figura 27).
- Transferir les dades amb els resultats des del dispositiu cap al processador principal i alliberar la memòria del dispositiu (si ja no és necessària), un cop finalitzada l'execució del *kernel* (pas 4 de la figura 27).

L'entorn CUDA proporciona una interfície de programació que simplifica aquestes tasques al programador. Per exemple, només cal especificar quines dades cal transferir des del processador principal al dispositiu i viceversa.

Figura 27. Esquema dels passos per a l'execució d'un *kernel* en una GPU



Durant tot aquest apartat utilitzarem el mateix programa d'exemple que fa la suma de dues matrius. El codi 4.1 mostra el codi corresponent a la implementació seqüencial en C estàndard de la suma de matrius per a arquitectura de tipus CPU. Noteu que aquesta implementació és lleugerament diferent a la del codi 3.1.

```
void matrix_add_cpu (fload *A, float *B, float *C, int N)
{
  int i, j, index;
  for (i=0; i<N; i++){
      for (j=0; j<N; j++){
        index = i+j*N;
        C[index] = C[index] + B[index];
      }
    }
  }

int main(){
    matrix_add_cpu(a, b, c, N);
}
```

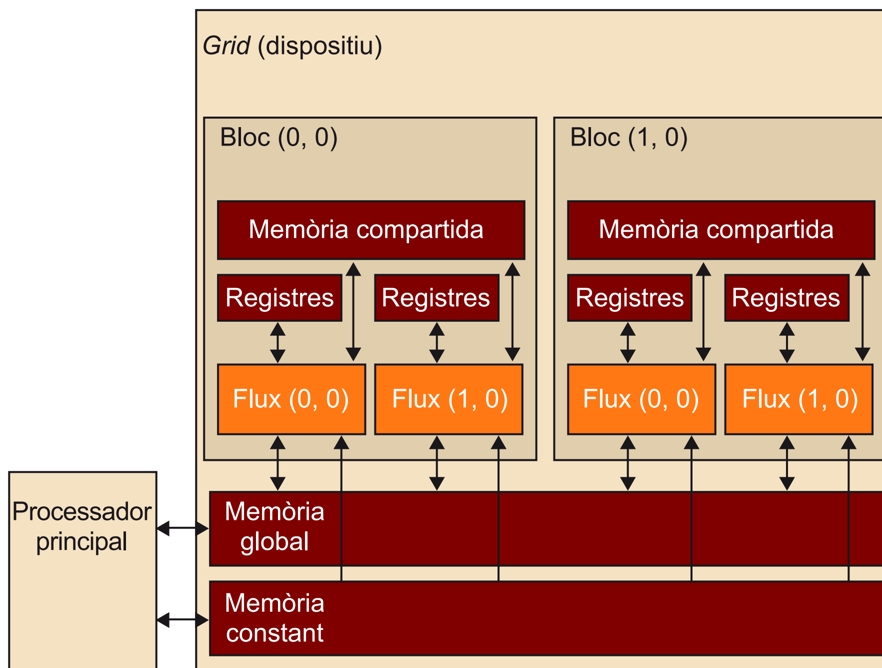
Codi 4.1. Implementació seqüencial en C estàndard de la suma de matrius per a arquitectura tipus CPU.

La figura 28 mostra el model de memòria de CUDA exposat al programador en termes d'assignació, transferència i utilització dels diferents tipus de memòria del dispositiu. A la part inferior de la figura es troben les memòries de tipus global i constant. Aquestes memòries són aquelles a les quals el processador

principal pot transferir dades de manera bidireccional, per *grid*. Des del punt de vista del dispositiu es poden accedir a diferents tipus de memòria amb els modes següents:

- Accés de lectura/escriptura a la memòria global, per *grid*.
- Accés només de lectura a la memòria constant, per *grid*.
- Accés de lectura/escriptura als registres, per flux.
- Accés de lectura/escriptura a la memòria local, per flux.
- Accés de lectura/escriptura a la memòria compartida, per bloc.
- Accés de lectura/escriptura.

Figura 28. Model de memòria de CUDA



La interfície de programació per a assignar i alliberar memòria global del dispositiu consisteix en dues funcions bàsiques: `cudaMalugar()` i `cudaFree()`. La funció `cudaMalugar()` es pot cridar des del codi del processador principal per a assignar un espai de memòria global per a un objecte. Com haureu observat, aquesta funció és molt similar a la funció `malugar()` de la biblioteca de C estàndard, ja que CUDA és una extensió del llenguatge C i pretén mantenir les interfícies com més similars millor a les originals.

La funció `cudaMalugar()` té dos paràmetres. El primer paràmetre és l'adreça del punter cap a l'objecte un cop s'hagi assignat l'espai de memòria. Aquest punter és genèric i no depèn de cap tipus d'objecte; per tant, s'haurà de fer *cast* a tipus `(void **)`. El segon paràmetre és la mida de l'objecte que es vol assignar en bytes. La funció `cudaFree()` allibera l'espai de memòria de l'objecte indicat com a paràmetre de la memòria global del dispositiu. El codi 4.2 mos-

tra un exemple de com podem utilitzar aquestes dues funcions. Després de fer el `cudaMalugar()`, `Matriu` apunta a una regió de la memòria global del dispositiu que se li ha assignat.

```
float *Matriu;

  int mida = AMPLADA * LLARGÀRIA * sizeof(float);
  cudaMalugar((void **) &Matriu, mida);
  ...
  cudaFree(Matriu);
```

Codi 4.2. Exemple d'utilització de les crides d'assignació i alliberament de memòria del dispositiu en CUDA.

Un cop que un programa ha assignat memòria global del dispositiu per als objectes o estructures de dades del programa, es poden transferir les dades que caldran per a la computació des del processador principal cap al dispositiu. Això es fa mitjançant la funció `cudaMemcpy()`, que permet transferir dades entre memòries. Cal tenir en compte que la transferència és asíncrona.

La funció `cudaMemcpy()` té quatre paràmetres. El primer és un punter a l'adreça de destinació on s'han de copiar les dades. El segon paràmetre apunta a les dades que s'han de copiar. El tercer paràmetre especifica el nombre de bytes que s'han de copiar. Finalment, el quart paràmetre indica el tipus de memòria involucrat en la còpia, que pot ser un dels següents:

- `cudaMemcpyHostToHost`: de la memòria del processador principal cap a la memòria del mateix processador principal.
- `cudaMemcpyHostToDevice`: de la memòria del processador principal cap a la memòria del dispositiu.
- `cudaMemcpyDeviceToHost`: de la memòria del dispositiu cap a la memòria del processador principal.
- `cudaMemcpyDeviceToDevice`: de la memòria del dispositiu cap a la memòria del dispositiu.

Cal tenir en compte que aquesta funció es pot utilitzar per a copiar dades de la memòria d'un mateix dispositiu però no entre diferents dispositius. El codi 4.3 mostra un exemple de transferència de dades entre processador principal i dispositiu basat en l'exemple de la figura 27.

```
void matrix_add (fload *A, float *B, float *C, int N)
{
  int size = N * N * sizeof(float);
  float * Ad, Bd, Cd;

  // 1. Reservar memòria per a les matrius
    cudaMalugar((void **) &Ad, size);
    cudaMalugar((void **) &Bd, size);
    cudaMalugar((void **) &Cd, size);
  // 2. Copiar matrius d'entrada al dispositiu
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
    ...
    // 4. Copiar resultat C cap al processador principal
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);
    ...
```

Codi 4.3. Exemple de transferència de dades entre processador principal i dispositiu.

A més dels mecanismes d'assignació de memòria i transferència de dades, CUDA també suporta diferents tipus de variables. Els diferents tipus de variables que utilitzen els diversos tipus de memòria són utilitzats en diversos àmbits i tindran cicles de vida diferents, tal com resumeix la taula 4.

Taula 4. Diferents tipus de variables en CUDA

Declaració variables	Tipus de memòria	Àmbit	Cicle de vida
Per defecte (diferents vectors)	Registre	Flux	*Kernel*
Vectors per defecte	Local	Flux	*Kernel*
`__device__, __shared__, int SharedVar;`	Compartida	Bloc	*Kernel*
`__device__, int GlobalVar;`	Global	*Grid*	Aplicació
`__device__, __constant__, int ConstVar;`	Constant	*Grid*	Aplicació

4.1.4. Definició de *kernels*

El codi que s'executa en el dispositiu (*kernel*) és la funció que executen els diferents fluxos durant la fase paral·lela, cadascun en el rang de dades que li correspon. Cal recordar que CUDA segueix el model SPMD (*single-program multiple-data*) i, per tant, tots els fluxos executen el mateix codi. El codi 4.4 mostra la funció o *kernel* de la suma de matrius i la seva crida.

```
__global__ matrix_add_gpu (fload *A, float *B, float *C, int N)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if (i<N && j<N){
    C[index] = A[index] + B[index];
  }
}

int main(){
  dim3 dimBlock(blocksize, blocksize);
  dim3 dimGrid(N/dimBlock.x, N/dimBlock.y);
  matrix_add_gpu<<<dimGrid, dimBlock>>>(a, b, c, N);
}
```

Codi 4.4. Implementació de la suma de matrius en CUDA.

Podem observar la utilització de la paraula clau específica de CUDA `__global__` davant de la declaració de `matrix_add_gpu()`. Aquesta paraula clau indica que aquesta funció és un *kernel* i que cal cridar-lo des del processador principal per tal de generar el *grid* de fluxos que executarà el *kernel* en el dispositiu. A més de `__global__` hi ha dues paraules clau més que es poden utilitzar davant de la declaració d'una funció:

1) La paraula clau `__device__` indica que la funció declarada és una funció CUDA de dispositiu. Una funció de dispositiu s'executa únicament en un dispositiu CUDA i només es pot cridar des d'un *kernel* o des d'una altra funció de dispositiu. Aquestes funcions no poden tenir ni crides recursives ni crides indirectes a funcions mitjançant punters.

2) La paraula clau `__host__` indica que la funció és una funció de processador principal, és a dir, una funció simple de C que s'executa en el processador principal i, per tant, que pot ser cridada des de qualsevol funció de processador principal. Per defecte totes les funcions en un programa CUDA són funcions de processador principal, si és que no s'especifica cap paraula clau en la definició de la funció.

Les paraules clau `__host__` i `__device__` es poden utilitzar simultàniament en la declaració d'una funció. Aquesta combinació fa que el compilador generi dues versions de la mateixa funció: una que s'executa al processador principal i que només es pot cridar des d'una funció de processador principal, i una altra que s'executa en el dispositiu i que només es pot cridar des del dispositiu o funció de *kernel*.

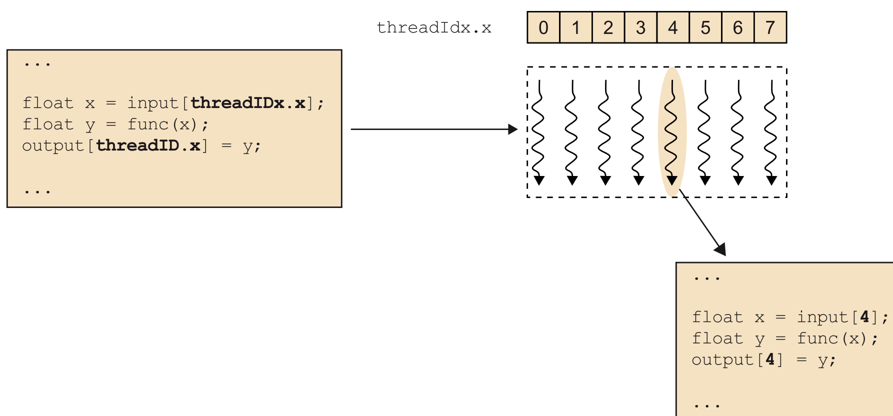
4.1.5. Organització de fluxos

En CUDA, un *kernel* s'executa mitjançant un conjunt de fluxos (per exemple, un vector o una matriu de fluxos). Com que tots els fluxos executen el mateix *kernel* (model SIMT, *single instruction multiple threads*) es necessita un mecanisme que permeti diferenciar-los, i així poder assignar la part corresponent

de les dades a cada flux d'execució. CUDA incorpora paraules clau per a fer referència a l'índex d'un flux (per exemple, `threadIdx.x` i `threadIDx.y` si tenim en compte dues dimensions).

La figura 29 mostra com el *kernel* fa referència a l'identificador de flux i que durant l'execució en cadascun del fluxos l'identificador se substitueix pel valor que li correspon. Per tant, les variables `threadIdx.x` i `threadIDx.y` tindran diferents valors per a cadascun dels fluxos d'execució. Noteu que les coordinades reflecteixen l'organització multidimensional dels fluxos d'execució, tot i que l'exemple de la figura 29 només fa referència a una dimensió (`threadIdx.x`).

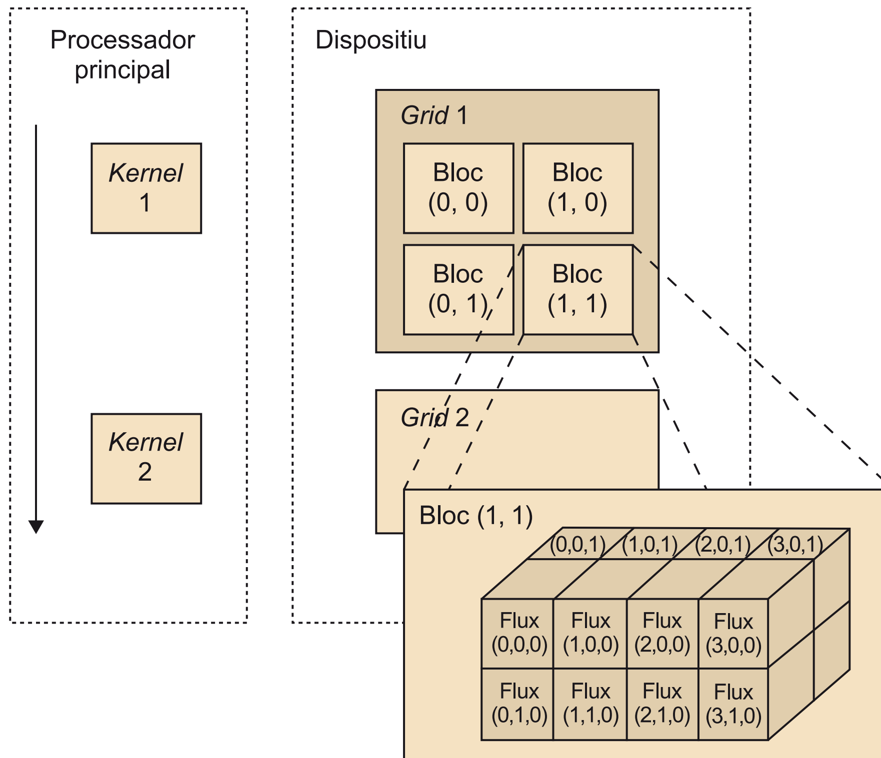
Figura 29. Execució d'un *kernel* CUDA en un vector de fluxos



Normalment els *grid* que s'utilitzen en CUDA estan formats per molts fluxos (entorn de milers o fins i tot de milions de fluxos). Els fluxos d'un *grid* estan organitzats en una jerarquia de dos nivells, com es pot veure en la figura 30. També es pot observar com el `kernel 1` crea el `Grid 1` per a l'execució. El nivell superior d'un *grid* consisteix en un o més blocs de fluxos. Tots els blocs d'un *grid* tenen el mateix nombre de fluxos i han d'estar organitzats de la mateixa manera. A la figura 30 el `Grid 1` està compost de 4 blocs i està organitzat com una matriu de 2 x 2 blocs.

Cada bloc d'un *grid* té una coordinada única en un espai de dues dimensions mitjançant les paraules clau `blockIdx.x` i `blockIdx.y`. Els blocs s'organitzen en un espai de tres dimensions amb un màxim de 512 fluxos d'execució. Les coordinades dels fluxos d'execució en un bloc s'identifiquen amb tres índexs (`threadIdx.x`, `threadIdx.y` i `threadIdx.z`), tot i que no totes les aplicacions necessiten utilitzar les tres dimensions de cada bloc de fluxos. A la figura 30 cada bloc està organitzat en un espai de 4 x 2 x 2 fluxos d'execució.

Figura 30. Exemple d'organització dels fluxos d'un *grid* en CUDA



Quan el processador principal fa una crida a un *kernel* s'han d'especificar les dimensions del *grid* i dels blocs de fluxos mitjançant paràmetres de configuració. El primer paràmetre especifica les dimensions del *grid* en termes de nombre de blocs i el segon especifica les dimensions de cada bloc en termes de nombre de fluxos. Tots dos paràmetres són de tipus `dim3`, que essencialment és una estructura de C amb tres camps (`x, y, z`) de tipus enter sense signe. Com que els *grids* són grups de blocs en dues dimensions, el tercer camp de paràmetres de configuració del *grid* s'ignora (qualsevol valor serà vàlid). El codi 4.5 mostra un exemple en què dues variables d'estructures de tipus `dim3` defineixen el *grid* i els blocs de l'exemple de la figura 30.

```
// Configuració de les dimensions de grid i blocs
dim3 dimGrid(2, 2, 1);
dim3 dimBlock(4, 2, 2);

// Invocació del kernel (suma de matrius)
matrix_add_gpu<<<dimGrid, dimBlock>>>(a, b, c, N);
```

Codi 4.5. Exemple de definició d'un *grid* i blocs.

CUDA també ofereix un mecanisme per a sincronitzar els fluxos d'un mateix bloc mitjançant la funció de tipus *barrier* `__syncthreads()`. Quan es crida la funció `__syncthreads()`, el flux que l'executa quedarà bloquejat fins que tots els fluxos del seu bloc arribin a aquest mateix punt. Això serveix per a assegurar que tots els fluxos d'un bloc han completat una fase abans de passar a la següent.

4.2. OpenCL

OpenCL és una interfície estàndard, oberta, lliure i multiplataforma per a la programació paral·lela. La principal motivació per al desenvolupament d'OpenCL va ser la necessitat de simplificar la tasca de programació portable i eficient de la creixent quantitat de plataformes heterogènies, com ara CPU multinucli, GPU o fins i tot sistemes encastats. OpenCL va ser concebuda per Apple, tot i que la va acabar desenvolupant el grup Khronos, que és el mateix que va impulsar OpenGL i n'és responsable.

OpenCL consisteix en tres parts: l'especificació d'un llenguatge multiplataforma, una interfície a escala d'entorn de computació i una interfície per a coordinar la computació paral·lela entre processadors heterogenis. OpenCL utilitza un subconjunt de C99 amb extensions per al paral·lelisme i utilitza l'estàndard de representació numèrica IEEE 754 per tal de garantir la interoperabilitat entre plataformes.

Hi ha moltes similituds entre OpenCL i CUDA, tot i que OpenCL té un model de gestió de recursos més complex, ja que suporta múltiples plataformes i portabilitat entre diferents fabricants. OpenCL suporta models de paral·lelisme tant a nivell de dades com a nivell de tasques. En aquest subapartat ens centrarem en el model de paral·lelisme a nivell de dades, el qual és equivalent al de CUDA.

4.2.1. Model de paral·lelisme a nivell de dades

De la mateixa manera que en CUDA, un programa en OpenCL està format per dues parts: els *kernels* que s'executen en un o més dispositius i un programa en el processador principal que invoca i controla l'execució dels *kernels*. Quan es fa la invocació d'un *kernel*, el codi s'executa en tasques elementals (*work items*) que corresponen als fluxos de CUDA. Les tasques elementals i les dades associades a cada tasca elemental es defineixen a partir del rang d'un espai d'índexs de dimensió N (NDRanges). Les tasques elementals formen grups de tasques (*work groups*), que corresponen al blocs de CUDA. Les tasques elementals tenen un identificador global que és únic. A més, els grups de tasques elementals s'identifiquen dins del rang de dimensió N i, per a cada grup, cadascuna de les tasques elementals té un identificador local, que anirà des de 0 fins a la mida del grup-1. Per tant, la combinació de l'identificador del grup i de l'identificador local dintre del grup també identifica de manera única una tasca elemental. La taula 5 resumeix algunes de les equivalències entre OpenCL i CUDA.

Taula 5. Algunes correspondències entre OpenCL i CUDA

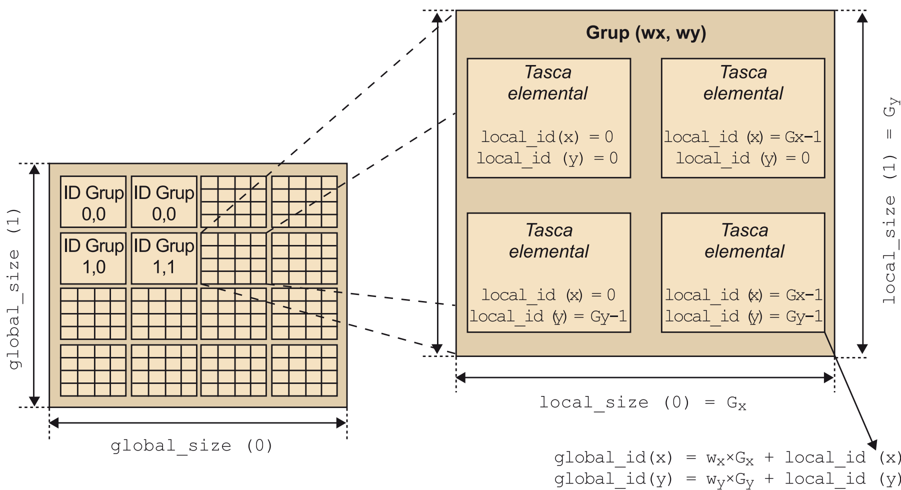
OpenCL	CUDA
Kernel	*Kernel*
Programa processador principal	Programa processador principal

OpenCL	CUDA
NDRange (rang de dimensió *N*)	*Grid*
Tasca elemental (*work item*)	Flux
Grup de tasques (*work group*)	Bloc
get_global_id(0);	blockIdx.x * blockDim.x + threadIdx.x
get_local_id(0);	threadIdx.x
get_global_size(0);	gridDim.x*blockDim.x
get_local_size(0);	blockDim.x

La figura 31 mostra el model de paral·lelisme a nivell de dades d'OpenCL. El rang de dimensió *N* (equivalent al *grid* en CUDA) conté les tasques elementals. En l'exemple de la figura, el *kernel* utilitza un rang de dues dimensions, mentre que en CUDA cada flux té un valor de `blockIdx` i `threadIdx` que es combinen per a obtenir i identificar el flux, en OpenCL disposem d'interfícies per a identificar les tasques elementals de les dues maneres que hem vist. Per una banda, la funció `get_global_id()`, donada una dimensió, retorna l'identificador únic de tasca elemental en la dimensió especificada. En l'exemple de la figura, les crides `get_global_id(0)` i `get_global_id(1)` retornen l'índex de les tasques elementals en les dimensions *X* i *Y*, respectivament. Per l'altra banda, la funció `get_local_id()`, donada una dimensió, retorna l'identificador de la tasca elemental dins del seu grup en la dimensió especificada. Per exemple, `get_local_id(0)` és equivalent a `threadIdx.x` en CUDA.

OpenCL també disposa de les funcions `get_global_size()` i `get_local_size()` que, donada una dimensió, retornen la quantitat total de tasques elementals i la quantitat de tasques elementals dintre d'un grup en la dimensió especificada, respectivament. Per exemple, `get_global_size(0)` retorna la quantitat de tasques elementals, que és equivalent a `gridDim.x*blockDim.x` en CUDA.

Figura 31. Exemple de rang de dimensió *N* en què es poden observar les tasques elementals, els grups que formen i identificadors associats

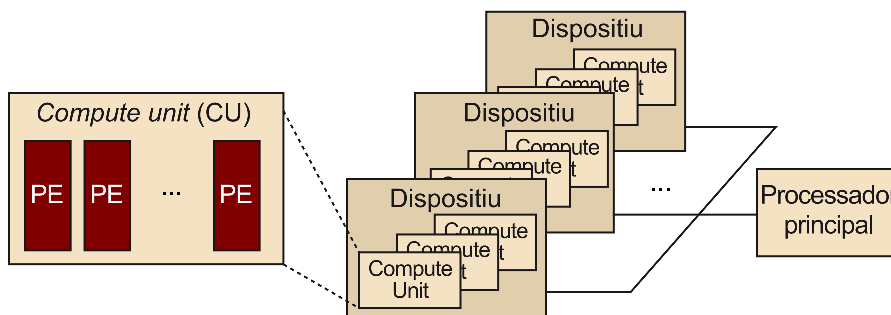


Les tasques elementals dintre del mateix grup es poden sincronitzar entre elles utilitzant *barriers*, que són equivalents a `__syncthreads()` de CUDA. En canvi, les tasques elementals de diferents grups no es poden sincronitzar entre elles, excepte si és per la terminació del *kernel* o la invocació d'un de nou.

4.2.2. Arquitectura conceptual

La figura 32 mostra l'arquitectura conceptual d'OpenCL, la qual està formada per un processador principal (típicament una CPU que executa el programa principal) connectat a un o més dispositius OpenCL. Un dispositiu OpenCL està compost per una o més unitats de còmput o *compute units* (CU), que corresponen als SM de CUDA. Finalment, una CU està formada per un o més elements de processament o *processing elements* (PE), que corresponen als SP de CUDA. L'execució del programa s'acabarà fent als PE.

Figura 32. Arquitectura conceptual d'OpenCL



4.2.3. Model de memòria

En un dispositiu OpenCL hi ha disponible una jerarquia de memòria que inclou diversos tipus de memòria diferents: global, constant, local i privada.

- La memòria global és la que poden utilitzar totes les unitats de càlcul d'un dispositiu.

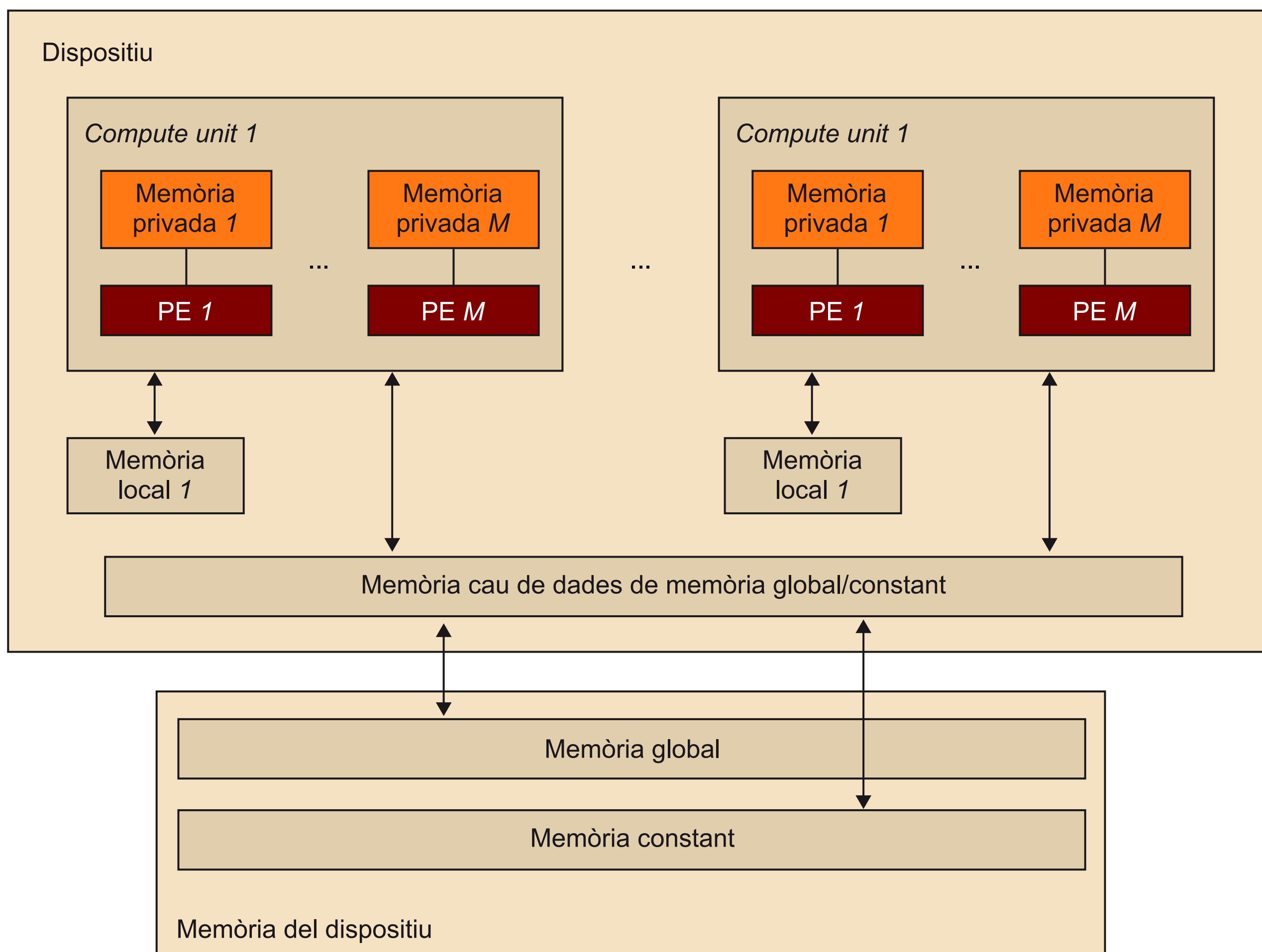
- La memòria constant és la memòria que es pot utilitzar per a emmagatzemar dades constants per a accés només de lectura de totes les unitats de càlcul d'un dispositiu durant l'execució d'un *kernel*. El processador principal és responsable d'assignar i iniciar els objectes de memòria que resideixen en l'espai de memòria.

- La memòria local és la memòria que es pot utilitzar per a les tasques elementals d'un grup.

- La memòria privada és la que pot utilitzar únicament una unitat de càlcul única. Això és similar als registres en una única unitat de càlcul o un únic nucli d'una CPU.

Així doncs, tant la memòria global com la constant corresponen als tipus de memòria amb el mateix nom de CUDA, la memòria local correspon a la memòria compartida de CUDA i la memòria privada correspon a la memòria local de CUDA.

Figura 33. Arquitectura i jerarquia de memòria d'un dispositiu OpenCL



Per tal de crear i manegar objectes en memòria d'un dispositiu, l'aplicació que s'executa en el processador principal utilitza la interfície d'OpenCL, ja que els espais de memòria del processador principal i dels dispositius són principalment independents l'un de l'altre. La interacció entre l'espai de memòria del processador principal i dels dispositius pot ser de dos tipus: copiant dades explícitament o bé mapant/desmapant regions d'un objecte OpenCL a memòria. Per a copiar dades explícitament, el processador principal envia la instrucció per a transferir dades entre la memòria de l'objecte i la memòria del processador principal. Aquestes instruccions de transferència poden ser o bé bloquejants o bé no bloquejants. Quan s'utilitza una crida bloquejant es pot accedir a les dades des del processador principal amb seguretat, un cop la crida ha finalitzat. En canvi, per a fer una transferència no bloquejant, la crida a la funció d'OpenCL finalitza immediatament i es treu de la cua, tot i que la memòria des del processador principal no és segura per a ser utilitzada. La interacció mapant/desmapant objectes en memòria permet que el processador principal pugui tenir al seu espai de memòria una regió corresponent a objectes OpenCL. Les instruccions de mapatge/desmapatge també poden ser bloquejants o no bloquejants.

4.2.4. Gestió de *kernels* i de dispositius

Els *kernels* d'OpenCL tenen la mateixa estructura que els de CUDA. Per tant, són funcions que es declaren començant amb la paraula clau `__kernel`, la qual és equivalent a `__global` de CUDA. El codi 4.6 mostra la implementació del nostre exemple de suma de matrius en OpenCL. Fixeu-vos que els arguments del *kernel* corresponents a les matrius estan declarats com a `__global`, ja que es troben a la memòria global i les dues matrius d'entrada estan també declarades com a `const`, ja que només caldrà fer accessos en modalitat de lectura. En la implementació del cos del *kernel* s'utilitza `get_global_id()` en les dues dimensions de la matriu per tal de definir l'índex associat. Aquest índex s'utilitza per a seleccionar les dades que corresponen a cadascuna de les tasques elementals que instanciï el *kernel*.

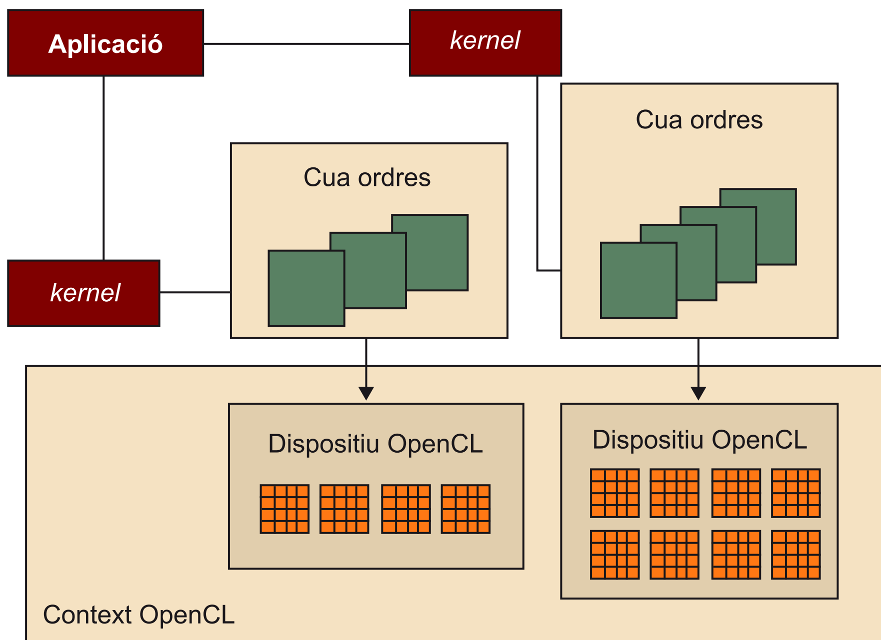
```
__kernel void matrix_add_opencl ( __global const float *A,
                                  __global const float *B,
                                  __global float *C,
                                  int N) {
  int i = get_global_id(0);
  int j = get_global_id(1);
  int index = i + j*N;
  if (i<N && j<N){
      C[index] = A[index] + B[index];
  }
}
```

Codi 4.6. Implementació del *kernel* de suma de matrius en OpenCL.

El model de gestió de dispositius d'OpenCL és molt més sofisticat que el de CUDA, ja que OpenCL permet abstraure diferents plataformes de maquinari. Els dispositius es gestionen mitjançant contextos. Tal com mostra la figura 34, per tal de gestionar un o més dispositius (2 en l'exemple de la figura), primer cal

crear un context que contingui els dispositius mitjançant o bé la funció `clCreateContext()` o bé la funció `clCreateContextFromType()`. Normalment cal indicar a les funcions `CreateContext` la quantitat i el tipus de dispositius del sistema mitjançant la funció `clGetDeviceIDs()`. Per a executar qualsevol tasca en un dispositiu primer cal crear una cua d'instruccions per al dispositiu mitjançant la funció `clCreateCommandQueue()`. Un cop s'ha creat una cua per al dispositiu, el codi que s'executa al processador principal hi pot inserir un *kernel* i els paràmetres de configuració associats. Un cop el dispositiu estigui disponible per a executar el *kernel* següent de la cua, aquest *kernel* s'elimina de la cua i passa a ser executat.

Figura 34. Gestió de dispositius en OpenCL mitjançant contextos



El codi 4.7 mostra el codi corresponent al processador principal per tal d'executar la suma de matrius mitjançant el *kernel* del codi 4.6. Suposem que la definició i inicialització de variables s'ha fet correctament i que la funció del *kernel* `matrix_add_opencl()` està disponible. En el primer pas es crea un context i, un cop s'han obtingut els dispositius disponibles, es crea una cua d'instruccions que utilitzarà el primer dispositiu disponible dels presents en el sistema. En el segon pas es defineixen els objectes que ens caldran en memòria (les tres matrius *A*, *B* i *C*). Les matrius *A* i *B* es defineixen de lectura, la matriu *C* es defineix d'escriptura. Noteu que en la definició de les matrius *A* i *B* s'utilitza l'opció `CL_MEM_COPY_HOST_PTR`, que indica que les dades de les matrius *A* i *B* es copiaran dels punters especificats (`srcA` i `srcB`). En el tercer pas es defineix el *kernel* que s'executarà posteriorment mitjançant `clCreateKernel` i s'especifiquen també els arguments de la funció `matrix_add_opencl`. A continuació s'envia el *kernel* a la cua d'instruccions prèviament definida i, finalment, es llegeixen els resultats de l'espai de memòria corresponent a la matriu *C* per mitjà del punter `dstC`. En aquest exemple no hem entrat en detall en

molts dels paràmetres de les diferents funcions de la interfície d'OpenCL; per tant, es recomana repassar-ne l'especificació, que està disponible a la pàgina web http://www.khronos.org/opencl/.

```
main(){
  // Inicialització de variables, etc.
  (...)

  // 1. Creació del context i cua al dispositiu
  cl_context context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);
  // Per a obtenir la llista de dispositius GPU associats al context
  size_t cb;
  clGetContextInfo( context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
  cl_device_id *devices = malloc(cb);
  clGetContextInfo( context, CL_CONTEXT_DEVICES, cb, devices, NULL);
  cl_cmd_queue cmd_queue = clCreateCommandQueue(context, devices[0], 0 , NULL);

  // 2. Definició dels objectes en memòria (matrius A, B i C)
  cl_mem memobjs[3];
  memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
  sizeof(cl_float)*n, srcA, NULL);
  memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
  sizeof(cl_float)*n, srcB, NULL);
  memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(cl_float)*n, NULL, NULL);

  // 3. Definició del kernel i arguments
  cl_program program = clCreateProgramWithSource(context, 1, &program_source, NULL, NULL);
  cl_int err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
  cl_kernel kernel = clCreateKernel(program, "matrix_add_opencl", NULL);
  err = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobjs[0]);
  err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&memobjs[1]);
  err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&memobjs[2]);
  err |= clSetKernelArg(kernel, 3, sizeof(int), (void *)&N);

  // 4. Invocació del kernel
  size_t global_work_size[1] = n;
  err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL, global_work_size,
  NULL, 0, NULL, NULL);

   // 5. Lectura dels resultats (matriu C)
  err = clEnqueueReadBuffer(context, memobjs[2], CL_TRUE, 0, n*sizeof(cl_float),
    dstC, 0, NULL, NULL);
  (...)
```

Codi 4.7. Codi del processador principal per a l'execució de la suma de matrius en OpenCL.

5. Arquitectures *many-core*: el cas de l'Intel Xeon Phi

En aquesta secció ens centrarem en arquitectures *many-core* utilitzant una de les famílies de processadors d'Intel, Many Integrated Core d'Intel, com a fil conductor. Com es veurà durant les properes subseccions, aquests processadors es caracteritzen per donar accés a un nombre molt elevat de fils d'execució. D'aquesta manera aplicacions que són altament paral·leles poden treure un rendiment força més elevat respecte d'altres arquitectures disponibles en el mercat. Per altra banda, un aspecte que les fa molt atractives respecte de l'ús de GPU (que també donen accés a un nombre molt elevat de fils d'execució) és que manté una visió coherent de l'espai de memòria, mentre que la majoria de GPU actuals no ho fan.

Primerament es presentarà l'evolució històrica (fins al moment d'escriure aquest material) de la família Xeon Phi i els seus orígens. A continuació es presentaran les característiques més important dels dos processadors que s'han presentat fins ara (Knights Ferry i Knights Corner). Finalment es donaran un conjunt de referències recomanades per a aprofundir en les arquitectures Xeon Phi.

5.1. Història dels Xeon Phi

L'any 2010, la companyia de processador Intel va anunciar el primer dels processador Intel que inclouria un gran nombre de nuclis integrats. Aquest nombre de nuclis seria molt més elevat que el nombre dels processadors dissenyats fins aleshores. Aquest concepte, ja conegut en la literatura de computació d'altes prestacions, és conegut com a *many integrated cores* (molts nuclis integrats).

Aquesta família de processadors, anomenada Intel Xeon Phi, heretava el disseny conceptual del projecte Larrabee. L'objectiu d'aquest era dissenyar una GPU (*graphical processing unit* o targeta gràfica) en què els còmputs es fessin en unitats de procés de propòsit general (x86). El projecte en qüestió no va sortir a la llum. No obstant això, tota la recerca feta es va emprar per a transformar aquest sistema multinucli en un processador de propòsit general que donés accés a un nombre molt elevat de fils d'execució. Per a més detalls sobre el projecte Larrabee es recomana la lectura de l'article de Selier, Carmean i Sprangle (2008). Un dels altres avantatges més importants d'aquestes arquitectures era el baix consum energètic respecte dels competidors d'altres companyies.

Des del 2010, Intel ha anunciat tres nous processadors dins de la família dels Xeon Phi: Knights Ferry (KNF), Knights Corner (KNC) i Knights Landing (KNL).

El primer de tots, KNF, va ser un prototip que tan sols es va lliurar a centres de recerca o centres de supercomputació per a començar a fer proves i avaluacions de rendiments amb aquesta nova família de processadors. D'aquesta no se'n va fer cap venda comercial. L'objectiu era que els potencials usuaris finals d'aquest comencessin a veure com calia adaptar les aplicacions (si era necessari) i fer projeccions de quin rendiment en traurien. Com a característiques més importants cal destacar que estava compost per trenta-dos nuclis amb quatre fils per nucli, estava construït sobre un procés de 45 nm i venia amb 2 GB de memòria integrada. Aquest era el primer dels processadors amb objectius comercials que Intel llançava amb 128 fils d'execució.

El segon, KNC, va ser la versió ja comercial de KNF i es va presentar l'any 2012. Aquesta arquitectura es fonamentava en el mateix disseny que KNF. No obstant això, donava accés a un nombre més elevat de nuclis (cinquanta nuclis en la primera versió), incorporava una memòria integrada molt més gran (fins a 16 GBS) i usava un procés de producció més eficient (22 nm). Usant aquesta nova arquitectura, centres de computació com el Texas Advanced Computing Center o el Centre Guangzhou van construir supercomputadors que es van col·locar en les primeres posicions del Top 500 (Top500) i del Green Top 500 (500). El primer va construir Stampede (TACC), sistema format per un total de 102,400 nuclis i capaç de 9.5 petaflopts. El segon va construir el computador anomenat Thiane-2 (Top500, China's Tianhe-2 Supercomputer Takes No. 1 Ranking on 41st TOP500). Aquest és capaç d'arribar a un rendiment màxim de 33.8 petaflops, i en el moment del llançament va ocupar la primera posició dels computadors més potents del món (Top 500).

Finalment, el darrer de tots, KNL es va anunciar al final del 2013. No obstant això, en el moment d'escriure aquest document, la informació que Intel havia fet pública era molt escassa. El més destacable era que aquesta arquitectura seria una canvi important respecte de la darrera generació (KNC). Cal remarcar que KNC havia estat una evolució de KNF.

El disseny intern de KNL probablement serà força diferent del que es veurà durant la propera subsecció. Com a característica més remarcable, aquest processador durà integrat un sistema de memòria anomenat *high memory bandwidth*. Aquest, a part de donar accés a una quantitat de memòria més gran que els seus predecessors, permetrà accedir-hi mitjançant una amplada de banda molt més gran. Això és especialment important tenint en compte que moltes de les aplicacions HPC es troben limitades, per una banda, per la quantitat de memòria que poden fer servir, però, per una altra banda (i en alguns casos més important), per la quantitat de dades per segon a la qual el processador pot seguir.

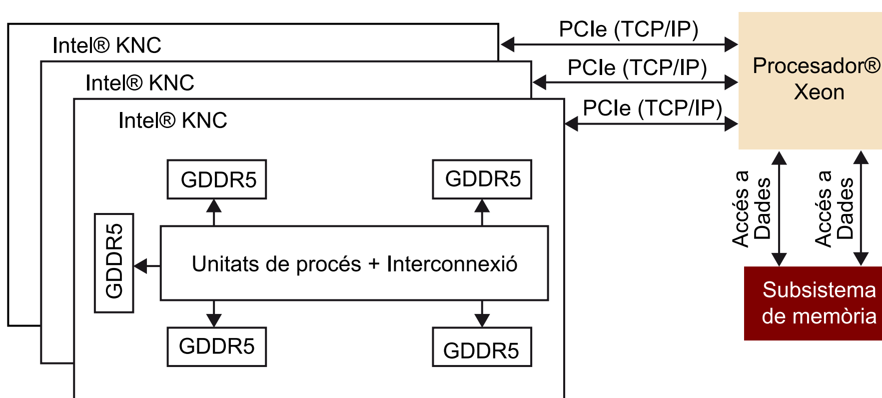
Com a exercici de lectura i recerca, es recomana que un cop acabada la lectura d'aquesta secció se cerquin en la Xarxa les novetats d'aquesta família de processadors: nous models, prestacions, etc. Cal tenir present que estem parlant

d'un mercat molt recent i en evolució constant. Per tant, el que es cobreix pot no incloure novetats més recents o canvis importants incorporats en noves generacions.

5.2. Presentació dels Xeon KNC i KNF

Com ja s'ha comentat en la darrera subsecció, l'arquitectura d'un KNC és una extensió comercial de la de KNF que augmenta la quantitat de nuclis i de memòria integrada. En aquesta subsecció es cobreixen les característiques arquitectòniques comunes a totes dues arquitectures. Com ja s'ha esmentat, en el moment d'escriure d'aquesta documentació, l'arquitectura dels Xeon KNL no s'havia fet pública. Per tant, no se'n discutiran els detalls.

Figura 35. Visió global d'un KNC



La figura 35 mostra una visió global de com un sistema de computació pot ser construït emprant KNC (a partir d'aquest punt s'emprarà KNC en referència a KNC o KNF indistintament, en cas contrari s'especificarà). Com es pot observar, els KNC es troben connectats a un processador Xeon tradicional (com podria ser un Sandy Bridge Corporation, Sandy Bridge Server Products) o Haswell (Corporation, Haswell Server Products) mitjançant PCIexpress. Això és degut al fet que tant KNC com KNF es van dissenyar com a coprocessadors. Aquests no són capaços d'arrancar un sistema operatiu convencional (per exemple, Linux o Windows). Per tant, necessiten un processador complet que permeti arrancar un sistema operatiu i permeti a l'usuari interactuar amb el sistema (executar aplicacions, gestionar fitxers, etc.).

De la mateixa manera que es fa amb els ordenadors convencionals i les targetes gràfiques (GPU), els KNC es col·loquen als PCIexpress del computador. Quan s'executa una aplicació, aquesta s'instancia en el processador principal (anomenat també *host*). El fil principal de l'aplicació s'executa en aquest. No obstant això, usant un conjunt de biblioteques que Intel facilita (*Corporation, Intel® Xeon Phi™ Coprocessor Developer's Quick Start Guide*, 2013), l'aplicació és capaç de moure dades al KNC i d'instanciar-hi fils d'execució.

```
float reduction(float *data, int size)
{
    float ret = 0.f;
    #pragma offload target(mic) in(data:length(size))
    for (int i=0; i<size; ++i)
    {
        ret += data[i];
    }
    return ret;
}
```

Codi 4.8. Exemple d'*offload*

El codi 4.8 mostra un exemple de com el programador pot especificar quina part del codi es vol executar en el *host* i quina s'executa en el KNC. Com es pot observar, per defecte, el codi s'executarà en el processador principal (*host*). Ara bé, si se'n vol executar una part en el coprocessador cal especificar-ho usant el que en el món de la programació es coneix com a *pragmas*. Aquests permeten especificar al compilador i a les biblioteques quin conjunt de línies es volen executar fora i com s'han d'executar. En l'exemple anterior es demana d'executar el bucle en el KNC i s'especifica que cal transferir-hi el contingut apuntat per la variable *data* que té una mida *size*.

Tal com es pot observar en la figura 35, es poden construir sistemes compostos per un sol processador *host* i tants coprocessadors KNC com entrades PCI-express tingui la placa base que s'està emprant. En cap cas es poden comunicar dos KNC connectats a la mateixa placa. Sempre s'ha de fer per mitjà del processador principal. Aquest és l'encarregat d'interaccionar amb tots els KNC que formen part del sistema.

Una de les novetats importants del més recent dels Xeon Phi, KNL, és que aquest sí que serà *bootable*. És a dir, aquest serà capaç d'arrancar un sistema operatiu complet i executar aplicacions per si mateix. Aquesta és una propietat molt interessant, ja que es podran construir sistemes d'altes prestacions sense haver d'usar forçosament processadors de servidor convencionals. Aquests sistemes podran estar compostos únicament per processadors KNL.

Durant els darrers paràgrafs s'ha esmentat que les dades es transmeten del processador principal al coprocessador. Això té tres implicacions importants. La primera és que els KNC han de tenir memòria pròpia per a poder emmagatzemar aquestes dades. Tal com es pot observar, els KNC empren prou memòria GDDR5 (Jedec) per a emmagatzemar dades. Les diferents versions disponibles en el mercat comprenen versions des de 6 GB fins a 16 GB les més potents.

La segona implicació important és que el *host* i el KNC no comparteixen espai de memòria virtual. És a dir, les variables que s'instancien el codi executat en el *host* no seran visibles per les parts de codi executades dins el KNC i viceversa. Si es volen compartir dades serà necessari especificar dins el codi quines variables es volen moure del *host* cap al KNC i de les KNC cap al *host*.

La tercera implicació pot afectar l'eficiència de les aplicacions que es dissenyin. És a dir, cal minimitzar, optimitzar al màxim les comunicacions que es fan del *host* al KNC. Cal estudiar quina és la millor manera de moure les dades (per exemple: moure-les totes de cop, a trossos, etc.). Cal tenir present que l'amplada de banda que un KNC pot donar de memòria principal és de 352 GB/s en la versió de 16 GBS (*pcwire*, 2012), i el que pot donar el PCIe pot ser de 8 GB/s en un PCIexpress x16 (anandtech). Per tant, el subsistema de memòria d'un KNC donarà una amplada de banda quaranta vegades més ràpida que el que dóna el PCIexpress.

5.3. Arquitectura i sistema d'interconnexió

En la subsecció anterior s'han donat les característiques més importants del funcionament d'un sistema de computació construït amb processadors de la MIC. Com s'ha vist, la comunicació entre diferents nodes KNC i el processador principal es fa amb una connexió PCIexpress. Ara bé, com està dissenyat internament un KNC? Com estan interconnectats els diferents nuclis, la memòria interna d'aquest i com aquests es connecten amb el món anterior ?

La figura 36 mostra l'arquitectura interna d'un KNC. Com es pot observar, aquest conté quatre tipus diferents d'agents:

- L'agent PCIe és l'encarregat de comunicar-se amb el *host*. Cada vegada que un dels nuclis accedeix a l'espai de memòria que es troba mapat al *host*, la petició del nucli acaba arribant al *host* per aquest agent. De la mateixa manera, cada vegada que el *host* vol enviar dades o peticions al KNC, ho fa usant aquest agent.

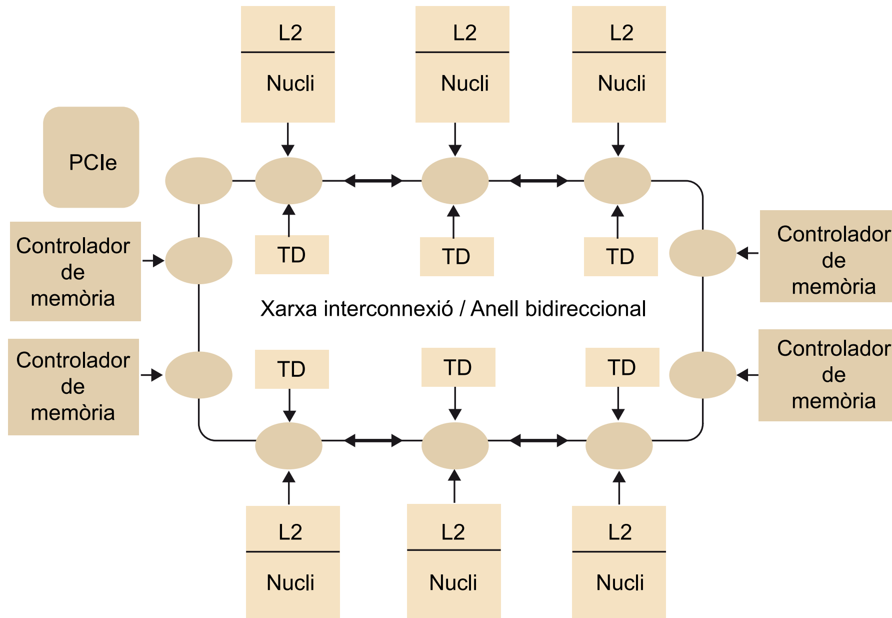
- L'agent Nucli és l'encarregat de dur a terme els càlculs i accions dels diferents fils que l'aplicació ha instanciat. Com ja s'ha esmentat anteriorment, cada nucli té quatre fils d'execució i conté dos nivells de memòria cau (més endavant se'n presenten més detalls).

- L'agent TD o *tag directory* és l'agent encarregat de mantenir la coherència de memòria. Cada vegada que un nucli vol accedir a una adreça de memòria ho ha de demanar al TD, ja que és l'encarregat de gestionar la coherència de l'adreça en qüestió. Cada adreça de l'espai de memòria és gestionada per un i tan sols un TD. Com es veurà més endavant, quan aquest rep una petició per a accedir a una dada ha de controlar que cap altre nucli no la tingui abans de demanar-la a memòria. En cas contrari haurà de notificar-ho al nucli que la tingui perquè aquest actualitzi el seu estat.

- Finalment, el darrer agent és el controlador de memòria. Aquest agent és l'encarregat de satisfer les peticions d'accés a la memòria principal que rep del TD i que han estat originades per un dels nuclis. Quan aquest rep una petició de lectura o escriptura, el controlador tradueix aquesta petició a les tecles d'ordre específiques que el dispositiu de GDDR5 entén. Per a més

informació de com un controlador de memòria interacciona amb un dispositiu GDDR5 es recomana la lectura de la introducció a aquesta tecnologia (Jedec).

Figura 36. Arquitectura interna d'un KNC



Els diferents agents es troben connectats per un anell bidireccional. Aquest és el responsable de realitzar la comunicació entre els diferents nuclis, TD, controladors de memòria i agent PCIexpress necessària per a poder mantenir la coherència de l'espai de memòria i comunicar-se amb el *host*. És important remarcar que tot i semblar una peça senzilla de maquinari, aquesta és una peça altament complexa que ha de tenir en compte molts aspectes importants: mecanismes de balanceig, mecanismes de descongestió en cas de molta càrrega, mecanismes per a evitar bloquejos (també anomenats *deadlocks*) etc.

Aquest anell bidireccional ha de ser capaç de transmetre informació de tres tipus diferents:

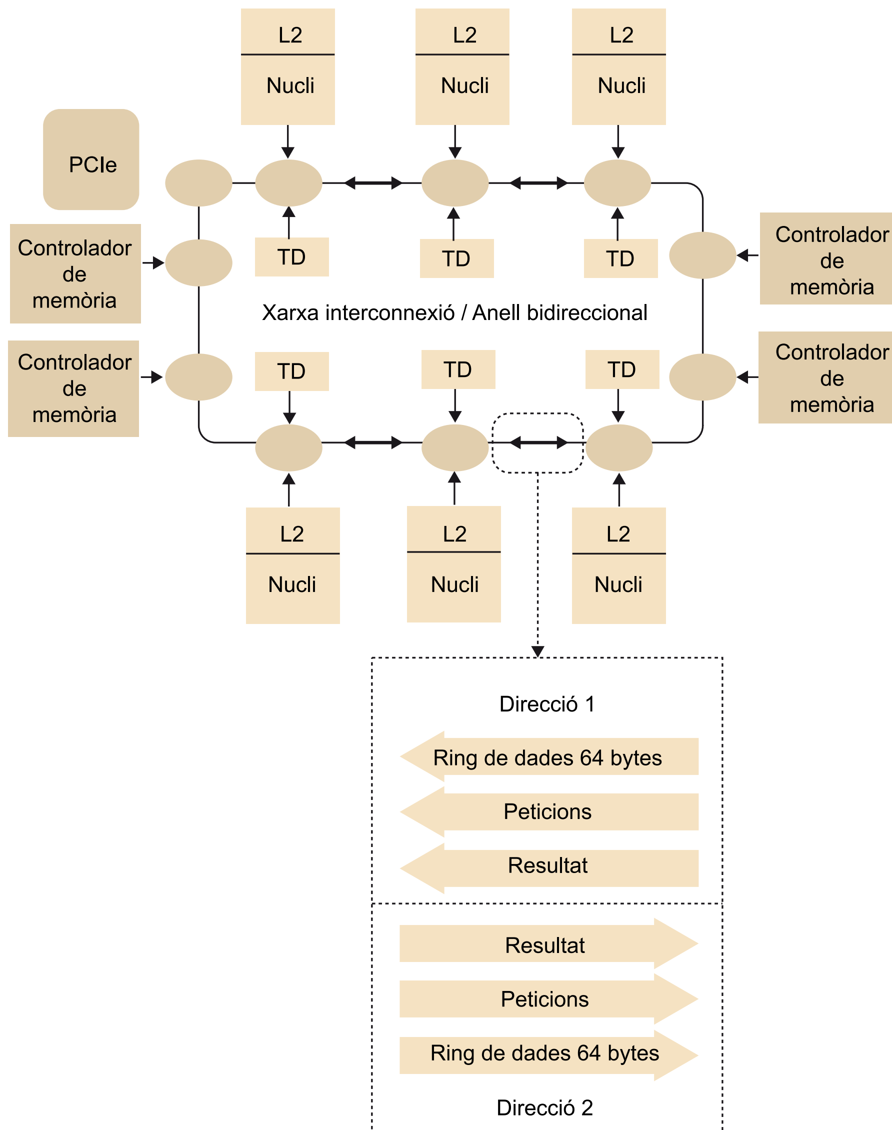
- Peticions entre els diferents agents. Per exemple la petició d'una línia de memòria en exclusiva d'un nucli a un TD; o bé una petició de lectura d'una línia de memòria d'un TD a un controlador de memòria.

- Dades que s'envien entre els diferents agents generades com a conseqüència d'una petició que s'ha iniciat anteriorment. Per exemple: després d'una de lectura d'una línia de memòria per part d'un nucli, algun dels controladors de memòria acabarà enviant les dades demanades al nucli que ha enviat la petició.

- Respostes de confirmació o resultat que, com en el cas de les dades, han estat generades com a conseqüència d'una petició. Per exemple: quan un TD rep una petició de lectura d'una línia de memòria, aquest ha de res-

pondre al nucli que la transacció s'està processant correctament i ha de dir en quin estat es retornarà la línia en qüestió (exclusiva o compartida).

Figura 37. Anell del *ring* bidireccional



Tal com es mostra en la figura 37, l'anell bidireccional està format per tres subanells diferents (tres en cada direcció). El primer de tots, anomenat AD, és el responsable de transportar les peticions. El segon de tots anomenat, BL (nom abreujat de ***bl****ock*), és el responsable de transportar les dades. I finalment el tercer, anomenat ACK (nom abreujat de ***ack****nowledgement*), és el responsable de transmetre confirmacions i resultats de les peticions.

Cadascun d'aquests subanells té una mida força diferent. El més ample de tots és el de BL. Aquest té una amplada total de 512 bits (mida de la línia de 64 bytes de memòria) més la capçalera (necessària per a incorporar informació de ruta: destinatari, origen, identificador de transacció, etc.). El següent, l'AD, és menys ample que aquest primer. Aquest només ha de transportar el tipus de petició, l'adreça física de la línia a la qual fa referència (48 bits) i, com en

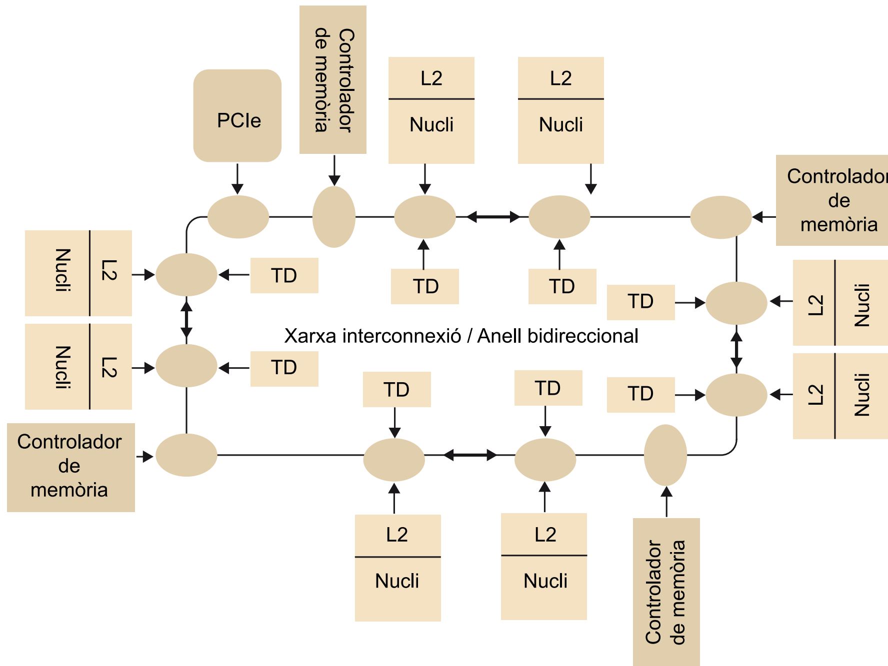
el cas anterior, la informació de ruta. Finalment, l'anell d'AK és el que menys amplada necessita. Aquest tan sols ha de dur la resposta, l'identificador de la transacció a la que fa referència i la informació de ruta.

És important remarcar que el disseny d'aquest anell bidireccional té com a objectiu donar prou amplada de banda perquè es puguin satisfer totes les transferències entre nuclis i l'amplada de banda que cadascun dels dispositius de GDDR5 dóna. És molt important tenir en compte que si cadascun dels dispositiu pot donar un total de 80 GB/s de lectura més escriptura a memòria, el disseny del sistema ha de ser capaç de suportar tota l'amplada de banda que tots els controladors poden donar (en la versió més potent fins a 350 GB/s). En cas contrari, no s'estaria usant tota la potència que el sistema de memòria dóna. Això no seria acceptable tenint en compte que aquest és un dels components més cars d'un sistema.

Tal com mostra la figura 38, els KNC tenen un total de quatre controladors de memòria. Cada controlador dóna accés a una quarta part de la memòria total disponible. Per exemple, en cas de tenir la versió amb 8 GB, cada controlador donarà accés a dispositius de GDDR5 de 2 GB amb una amplada de banda de 80 GB/s. Tal com s'ha esmentat, el disseny de la xarxa d'interconnexió haurà de ser capaç de poder saturar els quatre controladors.

En aquest apartat no s'aprofundirà en els detalls d'aquest disseny. En qualsevol cas, si el lector hi està interessat, es recomana la lectura del mòdul de xarxes d'interconnexió i ampliar els coneixements mitjançant lectures de treballs relacionats disponibles en la Xarxa. Com ja s'ha esmentat, un disseny d'un sistema de comunicació d'aquestes característiques requereix tenir en compte molts factors decisius a l'hora de treure'n el rendiment esperat.

Figura 38. Arquitectura d'un KNC



5.4. Nuclis dels KNC

Tal com ja s'ha esmentat anteriorment, la primera generació de Xeon Phi incorporava un nombre més limitat de nuclis respecte de KNC (un total de trenta-dos nuclis). No obstant això, l'arquitectura de tots dos era exactament la mateixa. En aquesta secció es presentaran les característiques més generals d'aquest i les seves prestacions.

Abans de discutir les prestacions d'aquests nuclis, cal remarcar que la segona generació, els processadors KNC, ofereix un ventall més obert de configuracions. Com es pot observar en la taula 6, l'opció més simple incorpora un total de cinquanta-set nuclis que poden córrer a una freqüència d'1.1 GHz, una memòria amb una amplada de banda màxima de 240 GB/s amb una capacitat de 6 GB i un rendiment màxim (en doble precisió) de 1003 gigaflops. Per altra banda, l'opció més agressiva d'aquest model incorpora un total de seixanta-un nuclis que poden córrer a 1.2 GHz oferint un rendiment màxim de 1208 gigaflops. Aquest també inclou una memòria amb una amplada de banda màxima de 352 GB/s i capacitat de 16 GBS.

Taula 6. Opcions de KNC disponibles (finals de 2013).

Model	Consum (Wats)	Nombre nuclis	Frequencia (GHZ)	Rendiment màxim (GFLops)	Ample de banda memòria	Capacitat memòria
3120P	300	57	1.1	1003	240	6
3120A	300	57	1.1	1003	240	6
5110P	225	60	1.053	1011	320	8

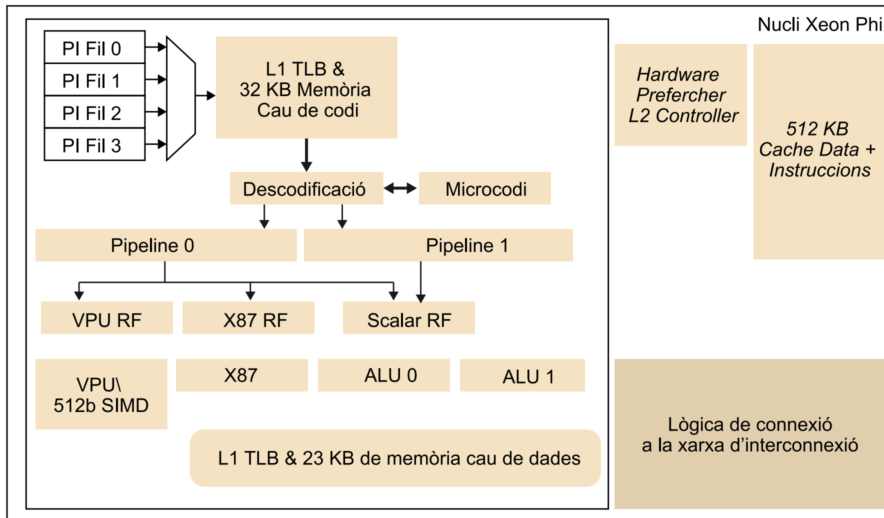
Model	Consum (Wats)	Nombre nuclis	Frequencia (GHZ)	Rendiment màxim (GFLops)	Ample de banda memòria	Capacitat memòria
5120D	245	60	1.053	1011	352	8
7110P	300	61	1.238	1208	352	16
7120X	300	61	1.238	1208	352	16

La diferència fonamental entre cadascuna de les opcions que hem mostrat en el punt anterior pel que fa als nuclis és a la freqüència a què aquests s'executen i la quantitat de memòria a la qual donen accés. La seva arquitectura interna és exactament la mateixa.

La figura 39 mostra els elements fonamentals que en defineixen el disseny. Com es pot observar en la part superior, aquest conté una estructura que emmagatzema la informació necessària per a poder executar quatre fils d'execució. De manera simbòlica, només es mostra el punter d'instrucció (PI), no obstant això, aquest emmagatzema altres dades necessàries per a poder gestionar el seu flux d'execució. Com ja s'ha vist en altres mòduls, això permet que les aplicacions puguin instanciar fins a quatre *hardware threads* o fils per cadascun dels nuclis.

A continuació podem veure que cadascun dels fils que s'executa s'extreu d'una memòria cau de primer nivell (només d'instruccions) de 32 kB. Com ja és sabut, les etapes d'un processador no treballen amb adreces virtuals, sinó físiques. Per aquest motiu, també en aquesta primera part trobem la TLB (*translation lookaside buffer*). Aquesta és una estructura que permet traduir les virtuals a físiques. Un cop la instrucció s'extreu de l'L1 d'instruccions i s'ha fet la TLB, les instruccions entren en la fase de descodificació. En aquesta fase, el nucli s'encarrega de dur a terme totes les comprovacions necessàries per a poder executar la instrucció (dependències, riscos estructurals, etc.) i de traduir les instruccions (anomenades *macroinstruccions*) en instruccions més simples, anomenades *microinstruccions*. Aquesta és una característica comuna a totes les arquitectures x86 (Corporation, Introduction to x64 Assembly).

Figura 39. Arquitectura d'un nucli de KNC



Les etapes següents ja són pròpiament les etapes de càlcul i d'accés a memòria en cas de ser necessari. Una de les característiques més remarcables d'aquesta arquitectura són les unitats vectorials que aquesta incorpora, anomenades *vector processing unit* (VPU). Aquestes unitats vectorials permeten executar setze operacions de precisió simple o vuit de doble precisió per cicle i permeten executar instruccions FMA. Les FMA (*fuse-multiply and add*) permeten sumar un element més un segon element multiplicat per un tercer element (A = A + B*C). Com que estem parlant d'una unitat vectorial, tant A com B com C són vectors. Per tant, les FMA permeten fer fins a trenta-dues operacions de precisió simple o bé setze de doble precisió per cicle. Aquestes operacions, molt emprades en el món de les aplicacions d'altes prestacions, permeten obtenir un bon rendiment de càlcul.

A part de la memòria cau d'instruccions de primer nivell, cada nucli conté una memòria de dades de primer nivell de 32 kB (juntament amb la seva TLB). Com a segon nivell de memòria, conté una L2 de 512 kB. En aquest cas l'L2 és unificada. És a dir, conté tant dades com instruccions.

Un dels elements interessants que un nucli KNC té és el *hardware prefetcher*. Aquest, tal com ja s'ha introduït en altres mòduls, és l'encarregat de demanar dades de memòria abans que un fil les demani. Això es fa mitjançant algorismes que miren de predir a quines adreces de memòria accedirà el fil en qüestió en un futur. D'aquesta manera, quan aquest les demani, ja estaran emmagatzemades en la memòria cau i no caldrà anar-les a buscar a la memòria principal (la qual cosa implicaria una quantitat de cicles molt superior).

5.5. Sistema de coherència

Els KNC implementen un sistema de coherència MESI (*modified, exclusive, shared i invalid*) que implementa un sistema GOLS (*globally owned locally shared*). Com ja s'ha discutit anteriorment, el fet de tenir un sistema de coherència permet assegurar que en tot moment la visió de memòria per part de tots els

nuclis és coherent. Per tant s'assegura que dos nuclis accedeixen a la línia de memòria @X, i aquests tindran un visió coherent d'aquesta (el mateix valor). També s'assegura que, si un nucli vol modificar el seu valor, cap altre nucli en tindrà una còpia que sigui diferent d'aquesta.

Taula 7. Definició de MESI

Estat	Definició	Propiteris de la linia	Estat respecte memòria
M	Modificada	Només un nucli és propiteri la línia	Modificada
E	Exclusiva	Només un nucli és propiteri la línia	No modificada
S	Compartida	Un conjunt de nuclis contenen la línia	Pot estar modificada o no
I	Invalida	Cap nucli conté la línia	-

La taula 7 mostra els diferents estats en els quals es pot trobar una línia de memòria en un moment determinat de l'execució dins d'una L1 o L2 d'un nucli. Pel fet de ser un sistema multinucli, una línia de memòria es pot trobar ubicada en diferents memòries cau de diferents nuclis. El sistema GOLS, definit en la taula 8, estén la definició de MESI afegint quatre estats més que es fan servir globalment en el sistema. Cada nucli guarda l'estat de la línia seguint un protocol MESI, i els TD, encarregats de mantenir una visió global coherent, implementen el protocol GOLS.

L'estat GOLS permet saber al TD que tot i que els diferents nuclis tenen una línia en estat compartit o S, aquesta havia estat prèviament modificada per un nucli (quan només aquesta la tenia). Per tant, quan el darrer nucli que contingui la línia la vulgui invalidar caldrà escriure-la en la memòria. Per exemple:

1) El nucli 1 demana la línia @Y al TD 1 i la modifica. Aquest passa d'estat E al nucli M.

2) El nucli 2 demana la línia @Y al TD 1. El TD primer enviarà una notificació al nucli 1 perquè passi en estat S i aquest li respondrà que la tenia modificada. Al final d'aquesta transacció, els nuclis 1 i 2 tindran la línia en estat S (per tant no la podran modificar) i el TD 1 tindrà apuntat que es troba en mode GOLS.

3) El nucli 1 invalida la línia i ho notifica al TD 1.

4) El nucli 2 invalida la línia i ho notifica al TD 1. Aquest cop, com que el nucli 2 és el darrer a tenir la línia i el TD sap que es trobava en mode GOLS, demanarà al nucli 2 que enviï les dades a memòria.

Taula 8. Definició de GOLS

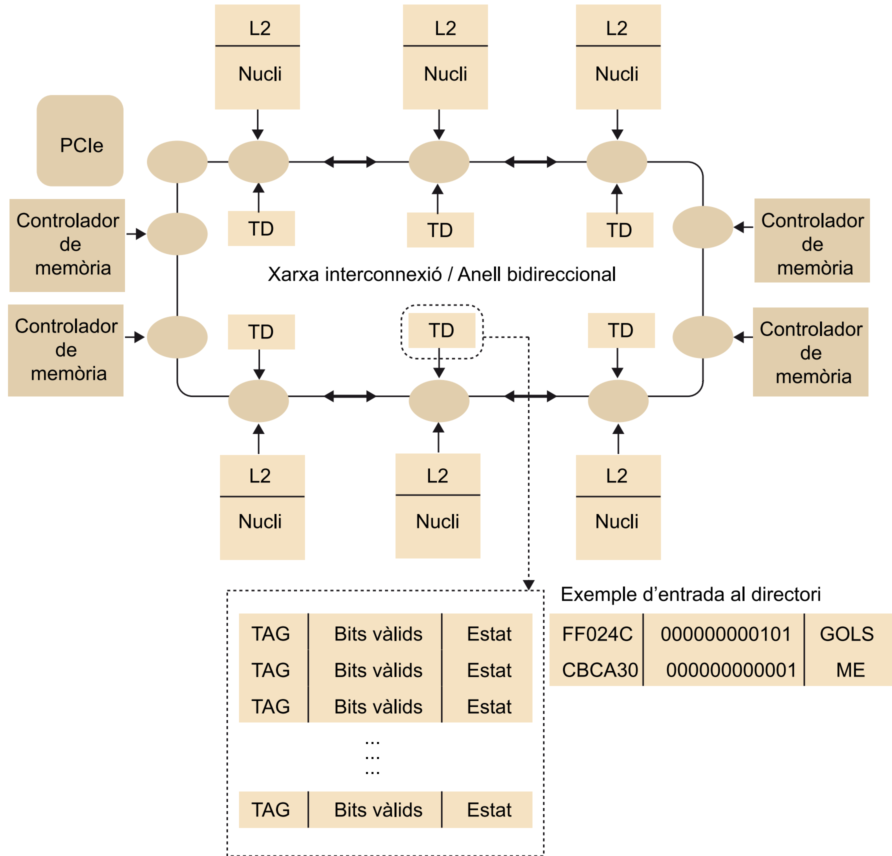
Estat	Definicio	Propiteris de la linia	Estat respecte memòria
GOLS	*GloballyownedLocallyShared*	Diferents nuclis la poden tenir	Modificada
GM/GE	*Globally Modified / Exclusive*	Només aquest nucli la té	Pot estar modificada o no
GS	*Globally Shared*	Diferents nuclis la poden tenir	No ha sigut modificada
GI	*Globally Invalid*	Cap nucli conté la línia	-

Al principi d'aquesta secció s'ha explicat que els TD són els agents encarregats de gestionar la coherència de memòria. Com ja s'ha esmentat, cada línia de memòria és gestionada per un i solament un TD. Cada nucli implementa un funció que li permet calcular quin és el TD que gestiona una adreça en qüestió td_id = f_calcul_td(@X). Tots els nuclis implementen la mateixa funció, d'aquesta manera el sistema s'assegura que les peticions sobre una adreça aniran sempre al mateix TD.

Cada TD conté una estructura que li permet seguir l'estat de cada línia i saber quin dels nuclis la tenen. La figura 35 n'és un exemple. Cada línia té associats, a part de l'estat, el que s'anomenen *els bits vàlids*. Cada bit correspon a un dels nuclis del sistema. Si un nucli té la línia, el bit corresponent estarà a 1. En aquest exemple, la línia amb TAG FF024C es troba en estat GOLS i la tenen els nuclis 0 i 2. Cal remarcar que el TD no guarda la adreça de la línia sencera, sinó el TAG (com ja s'ha discutit anteriorment en mòduls que introdueixen la gestió de memòries cau).

Els sistemes de coherència són una de les peces més importants a l'hora de determinar el rendiment que un sistema pot donar. Aquest és especialment important per a aplicacions en què els diferents fils comparteixen molt sovint dades o bé han d'accedir molt al sistema de memòria. Es recomana la lectura de l'article de Ramos Garea i Hoefler (2013). Aquest presenta una comparativa del sistema de coherència d'un KNC respecte d'un Sandy Bridge. No tan sols en presenta una descripció detallada, sinó que també en presenta un estudi de rendiment força interessant.

Figura 40. Implementació del directori del TD



5.6. Protocol de coherència

El KNC implementa un protocol de missatges entre els diferents agents per mantenir l'estat MESI i GOLS. Aquest protocol s'implementa sobre els tres subanells explicats en les darreres subseccions (AD, BL i AK). Com ja s'ha esmentat, les peticions es generen i s'envien per mitjà d'AD, les respostes per mitjà d'AK, i les dades per mitjà de BL.

El protocol que permet definir l'estat de coherència és complex i preveu moltes situacions i tipus de peticions diferents. Cal tenir present que no solament hi ha peticions d'accés a memòria. Hi podem trobar transaccions que demanen accés a zones de memòries no coherents, a la zona PCIexpress etc. Cada tipus de transacció ha de tenir associats uns conjunts de missatges i dependències perquè aquesta es dugui a terme.

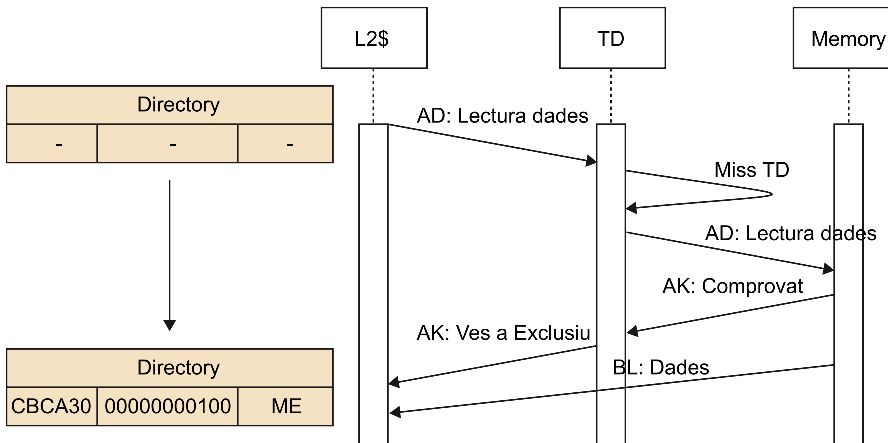
A tall d'exemple, a continuació es descriuen tres possibles transaccions en el cas d'una petició de lectura d'una línia de memòria per part d'un nucli.

Petició d'una línia de memòria no ubicada en cap nucli

El nucli demana la línia en qüestió al TD mitjançant l'anell d'AD. El TD busca en el seu directori i veu que aquesta no és dins de cap de les L2 de la resta de nuclis. A continuació, el mateix TD demana al controlador de memòria que

enviï les dades de la línia al nucli que l'ha demanat. El controlador enviarà una confirmació d'inici de transacció al directori. Aquest, quan la rep, comunica al nucli que la línia que ha demanat la té en mode exclusiu (mitjançant l'anell d'AK). En aquest punt, el director actualitza l'estat de la línia en ME i el nucli l'actualitzarà a Exclusiva un cop rebi, per l'anell de BL, les dades que rep de memòria. Com es pot veure, el TD ha activat el bit corresponent dels bits vàlids.

Figura 41. Petició de línia de memòria que no té cap altre nucli

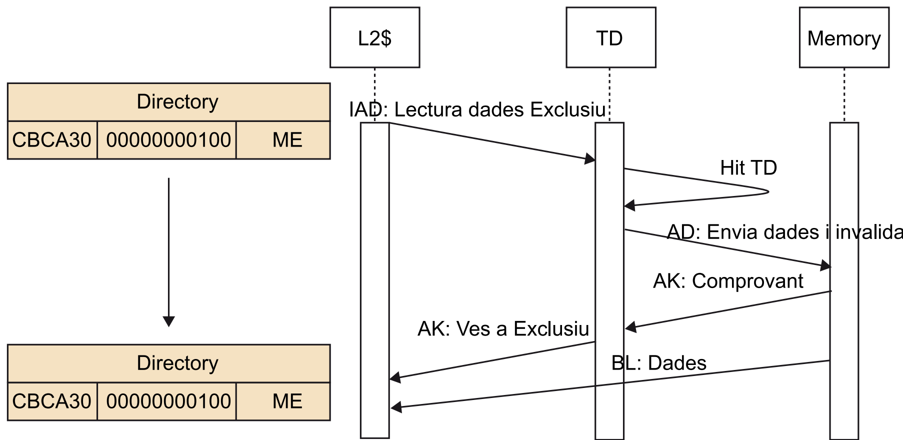


Petició d'una línia en estat exclusiu que ja té un altre nucli

A diferència de l'exemple anterior, en aquest cas s'assumeix que la línia ja estava ubicada en un altre nucli. Per altra banda, també s'assumeix que el nucli demana la línia en exclusiu. Això és necessari en tots el casos en què el nucli ha de modificar el contingut de la línia.

En aquest cas, quan el TD processa la petició del nucli, la cerca dins els TAGS és positiva. Com es pot veure en la figura 42, la cerca retorna que la línia ja està ubicada en el nucli 1 i que està en estat ME. Com que el nucli l'està demanant en exclusiva, el TD envia una petició al nucli que té la línia d'invalidar-la i enviar les dades al nucli que l'ha demanat. Com es pot veure, els bits vàlids són modificats de tal manera que el bit corresponent al primer nucli passa a ser zero i el bit del tercer nucli (el que ha demanat la línia) passa a ser 1.

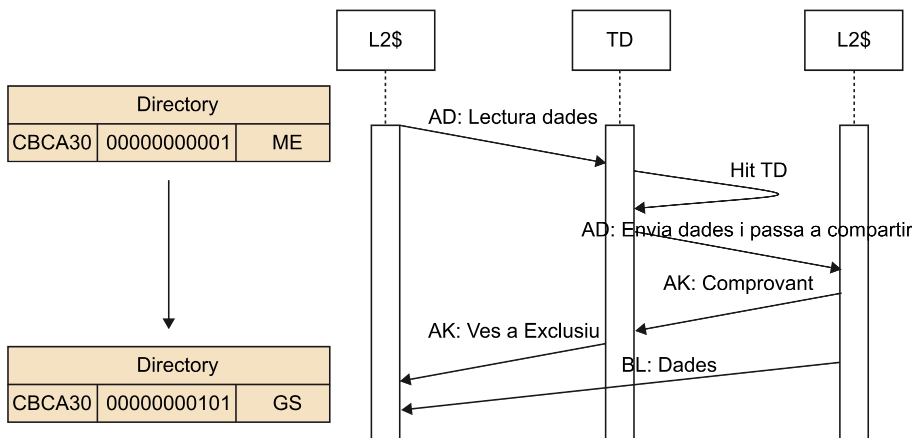
Figura 42. Petició d'una línia de memòria que ja té un altre nucli



Petició d'una línia que ja té un altre nucli

En aquest darrer exemple, a diferència del cas anterior, la línia no demana l'exclusivitat del nucli. Per tant, el TD demana al nucli que ja té la línia que la reenviï al nucli que l'ha demanat. En aquest cas, però, notifica al nucli que tenia la línia en estat exclusiu que passi a compartit o *shared*. Tal com es pot veure en la part esquerra de la figura 43, els bits vàlids passen de tenir el bit del nucli 1 a 1, i mantenen l'estat del bit del nucli 1 a 1. D'aquesta manera, el TD s'apunta que els dos nuclis tenen la línia en qüestió i que aquesta es troba en estat GS.

Figura 43



5.7. Conclusions

Durant aquesta secció s'han presentat les pinzellades generals i més representatives de les primeres arquitectures Many Integrated Core que Intel va treure al mercat sota la família anomenada Xeon Phi. Se n'han discutit les característiques generals com a sistema, arquitectura, coherència i nucli.

No obstant això, com ja s'ha esmentat, aquestes contenen una quantitat substancialment més elevada de detalls i definicions. És per aquest motiu que es recomana la lectura d'altres fonts que aprofundeixin més en els detalls esmentats en aquest document, com per exemple: *Corporation, Intel® Xeon Phi™ Coprocessor - the Architecture, The first Intel® Many Integrated Core (Intel® MIC) architecture product* (2013). També es recomana cercar en la Xarxa nous documents que donin més detalls d'aquesta família o bé que expliquin arquitectures de les quals encara no s'havien donat detalls a l'hora d'escriure aquesta documentació (per exemple: el processador Knights Landing).

En aquesta introducció a la família Xeon Phi, no s'han cobert detalls del seu model de programació. Tot i haver introduït el model general en la primera subsecció, Intel ha estès models de programació ja existents com, per exemple, OpenMP (OpenMP), Cilck (Corporation, Intel® Cilk™ Plus) o Intel-TBB (Corporation, Threading Building Blocks). Per altra banda, ha donat noves directives per poder treballar amb aquestes noves arquitectures. Aquestes extensions (majoritàriament accessibles mitjançant directives de tipus *pragma*) permeten, per una banda, interaccionar amb els KNC, per exemple, moure dades al MIC, crear o destruir fils, fer *map o reduce* de variables, etc.; i, per altra banda, permeten usar noves funcions proporcionades per aquesta arquitectura, per exemple (algunes ja disponibles en altres sistemes), *software prefetchs*, FMA, etc. És per aquest motiu que també es recomana la lectura de documents que donen més detalls del model de programació: *Corporation, Intel® Xeon Phi™ Coprocessor Developer's Quick Start Guide* (2013); Corporation, *An Overview of Programming for Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors*, Roth (2013).

Resum

En aquest mòdul hem estudiat com els dispositius gràfics han evolucionat des de dispositius especialitzats en la seva tasca per a la generació de gràfics fins a dispositius computacionals massivament paral·lels aptes per a la computació d'altres prestacions de propòsit general.

Hem vist que les arquitectures gràfiques estan formades per un *pipeline* gràfic, compost de diverses etapes, que fan diferents tipus d'operacions sobre les dades d'entrada. Hem destacat que la clau en l'evolució de les arquitectures gràfiques està en el fet de poder disposar de *shaders* programables que permeten al programador definir la seva funcionalitat.

Després de veure les característiques d'algunes arquitectures orientades a computació gràfica, hem estudiat les arquitectures unificades, les quals fusionen diferents funcionalitats en un únic model de processador capaç de tractar tant vèrtexs com fragments.

Un cop observades les limitacions de la programació sobre GPU, hem identificat el tipus d'aplicacions que poden treure profit de la utilització de GPU. També hem estudiat algunes arquitectures GPU unificades orientades a la computació de propòsit general. Hem vist que la principal característica d'aquest tipus d'arquitectura és la de disposar d'elements computacionals o *stream processors* (SP), que poden implementar qualsevol etapa del *pipeline* i proporcionen la flexibilitat suficient per a implementar aplicacions de propòsit general, tot i que amb certes limitacions.

Finalment, hem estudiat els dos models de programació principals per a computació de propòsit general sobre GPU (GPGPU): CUDA, que és una extensió de C inicialment concebuda per a arquitectures de propietat (Nvidia), i OpenCL, que és una interfície estàndard, oberta, lliure i multi-plataforma per a la programació paral·lela

Activitats

En aquest mòdul proposem les activitats següents per tal d'aprofundir en les arquitectures basades en computació gràfica i ajudar a complementar i aprofundir els conceptes introduïts.

1. Durant aquest mòdul hem utilitzat la suma de matrius com a exemple principal. Es demana que penseu i implementeu la multiplicació de matrius per la vostra banda utilitzant CUDA o OpenCL. Un cop tingueu la vostra versió pròpia compareu-la amb alguna de les nombroses solucions que es poden trobar fent una cerca a Internet o bé a la bibliografia.

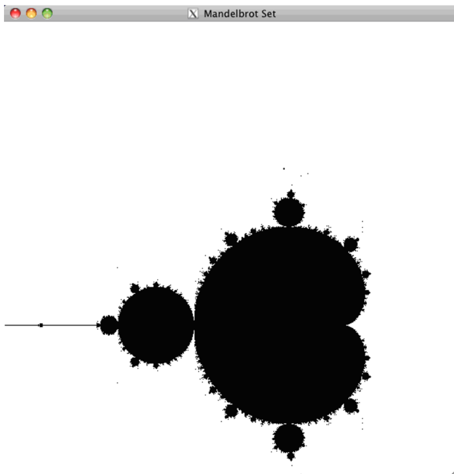
2. Exploreu el model de programació Microsoft Direct Compute, que és una alternativa a CUDA/OpenCL. Quins són els principals avantatges i inconvenients respecte a CUDA/OpenCL?

3. La majoria dels PC actuals disposen d'una GPU. Escriviu si en el vostre sistema personal en teniu alguna i quines són les característiques. Si teniu alguna GPU al vostre sistema, tingueu en compte les característiques amb vista a les activitats següents.

4. Exploreu com es poden implementar aplicacions que utilitzin més d'una GPU en el mateix sistema. Busqueu la manera de consultar les característiques del sistema compatible amb CUDA (per exemple, el nombre de dispositius disponibles en el sistema). Quin model seguiríeu en CUDA/OpenCL per a utilitzar diverses GPU distribuïdes en diferents nodes? (Noteu que en tractar-se de múltiples nodes us caldrà pas de missatges.)

5. Una de les característiques positives de les GPU és que poden arribar a oferir un rendiment en coma flotant molt elevat a un cost energètic força reduït. Busqueu a Internet el corrent elèctric que requereixen diverses arquitectures GPU i feu un estudi comparatiu respecte les CPU des d'un punt de vista de rendiment i consum elèctric. Quines condicions s'han de complir perquè surti a compte la utilització de GPU?

6. Es proposa programar una versió del conjunt Mandelbrot per a CUDA o OpenCL. El conjunt Mandelbrot resulta en una figura geomètrica de complexitat infinita (de natura fractal) obtinguda a partir d'una fórmula matemàtica i un petit algoritme recursiu, tal com mostra la figura següent.



El codi següent en C és una implementació seqüencial del conjunt Mandelbrot que resulta en la figura mostrada. Preneu aquesta implementació com a referència i tingueu en compte que per a compilar-la caldrà incloure les biblioteques `libm` i `libX11` i, per tant, una possible manera de compilació seria:

```
gcc -I/usr/include/X11 -omandel mandel.c -L/usr/lib -lX11 -lm
```

(Noteu que els directoris poden dependre de cada sistema en particular.)

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xos.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

#define         X_RESN    800     /* resolució eix de les X */
#define         Y_RESN    800     /* resolució eix de les Y */

typedef struct complextype {
        float real, imag;
} Compl;

int main ()
{
         Window           win;                    /* inicialització d'una finestra */
         unsigned int     width, height,          /* mida de finestra */
                          x, y,                   /* posició finestra */
                          border_width,           /* gruix marc en pixels */
                          display_width, display_height,  /* mida pantalla */
                          screen;
         char             *window_name = "Mandelbrot Set", *display_name = NULL;
         GC                       gc;
         unsigned long            valuemask = 0;
         XGCValues                values;
         Display                  *display;
         XSizeHints               size_hints;
         Pixmap                   bitmap;
         XPoint                   points[800];
         FILE                     *f            str[100];
         XSetWindowAttributes     attr[1];

/* variables Mandlebrot */
   int                                    i, j, k;
   Compl                                  z, c;
   float                                  lengthsq, temp;

/* connexió al Xserver */
         if (  (display = XOpenDisplay (display_name)) == NULL ) {
            fprintf (stderr, "drawon: cannot connect to X server %s\n", XDisplayName
            (display_name) );
            exit (-1);
         }

/* obtenir mida de la pantalla */
         screen = DefaultScreen (display);
         display_width = DisplayWidth (display, screen);
         display_height = DisplayHeight (display, screen);

/* establir mida de la finestra */
         width = X_RESN;
         height = Y_RESN;

/* establir posició de la finestra */
         x = 0;
         y = 0;

/* crear una finestra opaca */
         border_width = 4;
         win = XCreateSimpleWindow ( display, RootWindow (display, screen), x, y,
                 width, height, border_width, BlackPixel (display, screen),
                 WhitePixel(display, screen));
         size_hints.flags = USPosition|USSize;
         size_hints.x = x;
         size_hints.y = y;
         size_hints.width = width;
         size_hints.height = height;
         size_hints.min_width = 300;
         size_hints.min_height = 300;
         XSetNormalHints (display, win, &size_hints);
         XStoreName(display, win, window_name);

/* crear un context pels gràfics */
         gc = XCreateGC (display, win, valuemask, &values);
         XSetBackground (display, gc, WhitePixel (display, screen));
         XSetForeground (display, gc, BlackPixel (display, screen));
         XSetLineAttributes (display, gc, 1, LineSolid, CapRound, JoinRound);
         attr[0].backing_store = Always;
         attr[0].backing_planes = 1;
         attr[0].backing_pixel = BlackPixel(display, screen);
         XChangeWindowAttributes(display, win, CWBackingStore | CWBackingPlanes |
         CWBackingPixel, attr);
         XMapWindow (display, win);
         XSync(display, 0);

 /* Calcular i dibuixar els punts */
     for(i=0; i < X_RESN; i++)
     for(j=0; j < Y_RESN; j++) {

         z.real = z.imag = 0.0;
         c.real = ((float) j - 400.0)/200.0;  /* factors d'escalat per a finestra de
         800x800 */
         c.imag = ((float) i - 400.0)/200.0;
         k = 0;

          do  {                          /* iterar per definir el color pel pixel */
            temp = z.real*z.real - z.imag*z.imag + c.real;
            z.imag = 2.0*z.real*z.imag + c.imag;
            z.real = temp;
            lengthsq = z.real*z.real+z.imag*z.imag;
            k++;
          } while (lengthsq < 4.0 && k < 100);

         if (k == 100) XDrawPoint (display, win, gc, j, i);

         }

         XFlush (display);
         sleep (30);      /* Esperem uns quants segons per poder veure el resultat */
}
```

Un cop hàgiu fet la implementació podeu fer un petit estudi de rendiment utilitzant diferents configuracions (per exemple, diferents mides de bloc, distribució de fluxos, etc.). Noteu que per a escriure els resultats no podreu utilitzar el mateix mecanisme, ja que des dels dispositius

GPU no es pot accedir directament a l'espai de memòria de la finestra que es mostrarà per pantalla.

Un cop feu l'estudi de rendiment, el més probable és que no observeu escalabilitat respecte del nombre de nuclis utilitzats (*speedup*), especialment si considereu el temps d'execució seqüencial de referència el de l'execució en CPU. Això és perquè el problema és molt petit. A més, els dispositius GPU requereixen un cert temps d'inicialització, que podria dominar l'execució del programa. Penseu com faríeu el problema més gran per tal de poder observar una certa escalabilitat i fer mesures significatives.

La implementació proporcionada resulta en una imatge fractal en blanc i negre. Podeu fer una petita cerca a Internet i afegir-hi colors. També podeu proposar altres extensions, com ara la implementació amb múltiples GPU, ja siguin en un mateix sistema o distribuïdes, en diversos nodes.

Bibliografia

AMD (2008). *R600-Family Instruction Set Architecture. User Guide.*

AMD (2010). *ATI Stream Computing OpenCL. Programming Guide.*

AMD (2011). *Evergreen Family Instruction Set Architecture Instructions and Microcode. Reference Guide.*

Gaster, B.; Howes, L.; Kaeli, D. R.; Mistry, P.; Schaa, D. (2011). *Heterogeneous Computing with OpenCL*. Morgan Kaufmann.

Glaskowsky, P. (2009). *NVIDIA's Fermi: The First Complete GPU Computing Architecture.*

Green, S. (2005). "The OpenGL Framebuffer Object Extension". *Game Developers Conference (GDC).*

Kirk, D. B.; Hwu, W. W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann.

Munshi, A.; Gaster, B.; Mattson, T. G.; Fung, J.; Ginsburg, D. (2011). *OpenCL Programming Guide*. Addison-Wesley Professional.

Munshi, A. (2009). *The OpenCL Specification. Khronos OpenCL Working Group.*

Nvidia (2007). *Nvidia CUDA Compute Unified Device Architecture. Technical Report.*

Nvidia (2007). *The CUDA Compiler Driver NVCC.*

Nvidia (2008). *NVIDIA GeForce GTX 200 GPU Architectural Overview. Second-Generation Unified GPU Architecture for Visual Computing. Technical Brief.*

Owens, J. D.; Luebke, D.; Govindaraju, N.; Harris, M.; Krüger, J.; Lefohn, A.; Purcell, T. J. (2007). "A Survey of General-Purpose Computation on Graphics Hardware". *Computer Graphics Forum.*

Pharr, M.; Randima, F. (2005). *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional.

Randima, F.; Kilgard, M. J. (2003). *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley.

Sanders, J.; Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional.

Seiler, L.; Carmean, D.; Sprangle, E.; Forsyth, T.; Abrash, M.; Dubey, P.; Junkins, S.; Lake, A.; Sugerman, J.; Cavin, R.; Espasa, R.; Grochowski, E.; Juan, T.; Hanrahan, P. (2008). "Larrabee: A Many-Core x86 Architecture for Visual Computing". *ACM Trans. Graph.* (núm. 27, vol. 3, art. 18).