# Implementació en CUDA del Distance Field

Roger Puig

Gener 2011

**Resum**

El *Distance Field* en 3D és una representació on, per a un conjunt discret de punts dins d'un volum, es coneix la distància des de cada punt al punt més proper de la superfície de qualsevol dels objectes del domini. Es proposa una implementació per a calcular el *Distance Field* en 3D basada en l'arquitectura CUDA. En el plantejament proposat, es calcula la distància Euclídea per a cada punt i per totes les cares. El codi està preparat per a calcular el *signed distance field* però en la implementació final no s'ha inclòs. El signe es determina utilitzant el mètode *angleweighted pseudonormals*. També es proposa una segona implementació que permet obtenir un rendiment més satisfactori en temps a costa d'obtenir el DF només a les regions properes a la superfície. Finalment es compara el rendiment d'una implementació i l'altra i com varia el rendiment en cada una d'elles a mida que es van aplicant optimitzacions.

# Índex


**1 Introducció** **7**
- 1.1 Objectius . . . 7
- 1.2 Metodologia . . . 7
- 1.3 Eines . . . 7

**2 Del *Color Graphics Adapter* a CUDA** **9**
- 2.1 Els primers anys de les targetes gràfiques . . . 9
- 2.2 L'acceleració 3D . . . 9
- 2.3 El pipeline gràfic programable . . . 10
- 2.4 GPGPU . . . 10
- 2.5 Els processadors gràfics unificats . . . 11
- 2.6 L'arquitectura SM . . . 11

**3 CUDA** **12**
- 3.1 Model d'execució de CUDA . . . 13
  - 3.1.1 Estructura d'un programa en CUDA . . . 13
  - 3.1.2 Kernels . . . 14
  - 3.1.3 Jerarquia de threads . . . 14
  - 3.1.4 Jerarquia de memòria . . . 15
  - 3.1.5 Sincronització i escalabilitat . . . 17
- 3.2 Assignació de *threads* . . . 18
- 3.3 Planificació de threads . . . 19
- 3.4 Gestió de recursos i escalat transparent . . . 19
  - 3.4.1 *Compute capability* . . . 19
  - 3.4.2 Relació entre el hardware i l'execució de CUDA . . . 19

**4 El Distance Field** **22**
- 4.1 Definició del problema . . . 22
- 4.2 Càlcul de la distància . . . 22
- 4.3 Càlcul del signe . . . 22
- 4.4 Visió general de l'algoritme . . . 23

**5 Estat de l'art** **24**
- 5.1 Implementacions en GPU:*DiFi:Fast 3D Distance Field Computation Using Graphics Hardware* . . . 24
- 5.2 Implementació en CPU: *Signed Distance Computation Using the Angle Weighted Pseudonormal* . . . 24
- 5.3 Implementacions en CUDA: *CUDA-base Signed Distance Field Calculation for Adaptive Grids* . . . 25

**6 Implementació força bruta amb *tiling* en CUDA** **27**
- 6.1 Visió general de la implementació . . . 27
- 6.2 Preparant les dades d'entrada . . . 28
- 6.3 Càlcul de la distància . . . 28
- 6.4 Càlcul del signe . . . 29
- 6.5 Optimitzacions . . . 29
  - 6.5.1 La importància del format de les dades d'entrada . . . 29
  - 6.5.2 *Shared memory* . . . 30

6.5.3 Memoització . . . 30
6.6 Altres detalls de la implementació . . . 31
6.6.1 Memòria compartida . . . 31
6.6.2 Organització de *threads* i *blocks* . . . 31
6.6.3 Sincronització de *threads* . . . 31

**7 Millora amb *bricking*** **32**

**8 Altres implementacions** **34**
8.1 CPU . . . 34
8.2 Força bruta amb recorregut per cares seqüencial . . . 34

**9 Representació del DF i el model gràficament** **35**

**10 Resultats** **36**
10.1 Nombre de *threads* llençats per model i volum . . . 37
10.2 Força bruta: CUDA . . . 37
10.2.1 Proves mida de block . . . 37
10.2.2 Proves temps en dimensió màxima . . . 37
10.2.3 *Coalescing* en *compute capability 1.1* . . . 38
10.2.4 Efecte memoització . . . 39
10.2.5 Temps d'execució i error introduït per les funcions de distància . . . 39
10.3 Seqüencial vs *tiling* . . . 40
10.4 Força bruta: CPU vs CUDA . . . 40
10.5 Bricking CUDA . . . 41
10.5.1 Cost del recorregut en les cares . . . 41
10.5.2 Precisió perduda utilitzant diferents factors de subdivisió i overlapping, temps de triatge i temps de *kernel* . . . 41

**11 Anàlisi dels resultats** **45**
11.1 DiFi vs la nostra implementació *bricking* CUDA . . . 45
11.2 Bærentzen & Aanæs vs la nostra implementació en CUDA . . . 45
11.3 Diferència de temps d'execució entre seqüencial i *tiling* . . . 45

**12 Conclusions** **47**

**13 Treball futur** **48**

**14 Planificació i cost del projecte** **49**

**Referències** **50**

**A Manual d'usuari** **51**
A.1 Inici per la línia de comandes . . . 51
A.2 Ús de la interfície . . . 51

**B Kernels de CUDA** **53**
B.1 Kernel tiling . . . 53
B.2 Kernel seqüencial . . . 55

**C Funcions de CUDA** **55**
C.1 get3DPoint . . . 55
C.2 getDistanceSimple . . . 56
C.3 getDistance . . . 56

**D Shaders** **59**

# Índex de figures

1 Pipeline fix. Font [5] . . . 9
2 Pipeline programable . . . 10
3 Diferència entre processador no unificat i unificat. Font: nVIDIA . . . 11
4 Arquitectura CUDA. Font: nVIDIA . . . 12
5 Flops teòrics de GPU i CPU. Font: nVIDIA . . . 12
6 Esquema de l'arquitectura CPU i GPU. Font: nVIDIA . . . 13
7 Típica execució d'un programa en CUDA . . . 14
8 Exemple de kernel de CUDA . . . 15
9 Extensions de CUDA per a declarar funcions . . . 15
10 Jerarquia de *threads* en CUDA . . . 16
11 Exemple d'execució d'un kernel . . . 16
12 Visió general de la jerarquia de memòria. Font: nvIDIA . . . 17
13 Escalat dinàmic en CUDA . . . 18
14 Compute capability de la GPU nVIDIA GeForce GTX 280 . . . 20
15 Regions d'un triangle per a calcular la distància d'un punt. . . . 23
16 Angles incidents $\alpha_1$ , $\alpha_2$, $\alpha_3$ d'un punt $x \in M$ . . . 23
17 Algoritme amb el bucle interior enfocat als punts del volum . . . 24
18 Algoritme amb el bucle exterior enfocat als triangles de la malla . . . 24
19 Resultats de Park et al. . . . 26
20 Esquema de l'algoritme 1 . . . 27
21 Esquema de l'algoritme 2 . . . 28
22 Esquema de redistribució de les dades d'entrada per afavorir *coalescing* . . . 30
23 Problema de l'algoritme amb *bricking* . . . 32
24 Solució a l'error del *bricking* . . . 33
25 Representació del DF en OpenGL . . . 35
26 Malles de prova . . . 36
27 Proves amb mida de *block* . . . 38
28 Proves amb temps en funció de la figura i el volum . . . 38
29 Gràfic comparatiu entre aplicar memoització o no . . . 39
30 Gràfic diferència de temps funció de distància . . . 40
31 Gràfic temps d'execució amb recorregut seqüencial per cares. . . . 41
32 Gràfics comparatius del temps d'execució de la implementació en seqüencial i la de *tiling* . . . 41
33 Cost de classificació de cares . . . 42
34 Pèrdues de dades en *bricking* . . . 43
35 Pèrdues de precisió en *bricking* . . . 43
36 Diagrama gantt de la planificació del projecte . . . 49

# Índex de taules


1 Jerarquia de memòria de CUDA . . . . . 15
2 Resultats de la implementació DiFi . . . . . 25
3 Resultats de la implementació de Bæretzen i Aanæsi . . . . . 25
4 Complexitat de les malles de prova . . . . . 36
5 Punts a mostrejar per model i mida del volum . . . . . 37
6 Proves de *coalescing* en *compute capability* 1.1 . . . . . 39
7 Error introduït per la funció de distància. . . . . 40
8 Comparació en temps de CPU i CUDA . . . . . 42
9 Errors i temps en bricking amb horse i volum de 64 . . . . . 42
10 Comparació resultats *bricking* CUDA vs DiFi . . . . . 45
11 Costos del projecte. . . . . 50

# 1 Introducció

## 1.1 Objectius

L'objectiu principal del projecte és aprendre i familiaritzar-se amb el model de programació CUDA i entendre el seu funcionament i les seves limitacions. Per això ens hem basat en un algoritme del càlcul del DF. Es començarà implementant una versió bàsica del codi i s'anirà especialitzant per a l'arquitectura CUDA i s'observarà com canvia el rendiment a cada iteració. El resultat final s'ha de poder representar en una aplicació que permeti observar els resultats gràficament.

## 1.2 Metodologia

La metodologia seguida és la següent:

- **Recerca:** Estudi de l'arquitectura de les targetes gràfiques i les restriccions que imposen i mètodes existents per a adreçar problemes computacionals.
- **Diagnosi:** S'estudien les característiques del problema i els objectius que es volen complir.
- **Anàlisi:** S'analitzen els objectius que es volen assolir i es proposen solucions al problema.
- **Proposta:** Es pren una proposta i es dissenya com s'integrarà en el projecte.
- **Implementació:** S'implementa la proposta escollida.
- **Resultats:** S'observen els resultats obtinguts i si no s'assoleixen els objectius es prova una altra proposta.

Cal notar que la idea original era la implementació d'un algoritme diferent al SDF però a la fase d'anàlisi és va observar que no era viable continuar endavant i es va buscar una alternativa arribant al DF.

## 1.3 Eines

Per tal d'implementar el projecte, s'han requerit unes eines hardware i software. S'ha tingut accés a dos ordinadors amb capacitat per a executar CUDA i OpenGL. Un d'ells, el que ha servit per a desenvolupar i debugar i l'altre per a fer proves. També es faran proves amb el de desenvolupament per a comprovar que el programa i la tecnologia escalen correctament. El PC de desenvolupament té les següents especificacions:

- **CPU:** Intel Core 2 Duo T7300 2.00 GHz
- **RAM:** 2GB
- **Targeta gràfica:** nVIDIA GeForce 8600M GS

Les especificacions hardware del PC de proves són les següents:

- **CPU:** Intel Core 2 Quad Q9550 2.83 GHz
- **RAM:** 8GB
- **Targeta gràfica:** nVIDIA GeForce GTX 280

Per a poder desenvolupar en CUDA, s'han d'instal·lar els controladors de la targeta gràfica i l'entorn de desenvolupament que permet compilar i executar codi. El software utilitzat per al desenvolupament del projecte és el següent:

- **Sistema operatiu:** Ubuntu 10.04
- **Compilador C++:** gcc 4.3.4
- **IDE:** Qt Creator 4.6.2
- **GPU driver:** nVIDIA device driver 260.24
- **CUDA driver:** 3.20
- **CUDA runtime:** 3.20
- **CUDA SDK:** nVIDIA CUDA SDK 3.2.9

# 2 Del *Color Graphics Adapter* a CUDA

En aquesta secció s'explicarà el camí que ha seguit l'arquitectura hardware de les targetes gràfiques partint de les primeres controladores de pantalla, utilitzades només per a mostrar colors a la pantalla, fins a arribar a la última generació de GPUs que permeten utilitzar el seu hardware com a processadors massivament paral·lels.

## 2.1 Els primers anys de les targetes gràfiques

El primer processador gràfic va ser introduït el 1981 per IBM[8]. El seu ordinador personal IBM PC contenia l'anomenat CGA (*Color Graphics Adapter*). La màxima resolució que permetia mostrar aquesta controladora era de 640x200 amb una profunditat de color de 4 bits (16 colors). L'única característica que tenia, era que contenia una memòria de 16Kbytes on s'emmagatzemava el contingut del que es mostrava a la pantalla. No contenia hardware específic per a rasteritzar. Això significa que tots els càlculs s'havien d'efectuar a la CPU per a després ser copiats de la memòria principal a la de la controladora. Uns anys més tard, Hercules va introduir la *Hercules Graphic Card*. Aquesta podia mostrar una resolució de 720x348 però amb només imatges monocromàtiques. Així fins a finals dels 90, les targetes gràfiques anaven evolucionant en termes de resolució i colors però els procediments que s'hi executaven eren els mateixos.

## 2.2 L'acceleració 3D

El primer adaptador que permetia l'acceleració 3D utilitzant hardware específic era la *Matrox Impression*. Fins al moment l'acceleració 3D només es podia trobar en estacions de treball. A partir d' aquest moment, van sorgir una sèrie d'adaptadors que utilitzaven el pipeline gràfic fixe. Aquest pipeline (figura 1) estava dissenyat exclusivament per a dibuixar triangles. El procés del

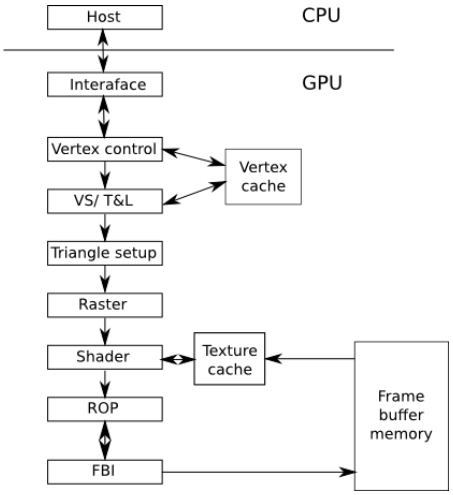


Figura 1: Pipeline fix. Font [5]

pipeline fixe és el següent. La CPU envia una sèrie de triangles a la GPU. Aquests, a l'etapa *Vertex Control*, són convertits de manera que el hardware ho entengui i són desats a la cache de vèrtex. L' etapa *Vertex Shading, transform & Lighting* s'encarrega de transformar els vèrtex i assignar-los uns valors, colors, normals, coordenades de textura o tangents. L' etapa *triangle*

*Setup* genera equacions que més tard són utilitzades per a interpolar les coordenades de textura assignades pel *VS/T&L*. L'etapa de *Raster*, decideix quins píxels pertanyen a quin triangle i a cada píxel i interpola els valors donats pels vèrtexs. És a l'etapa de *shading* que a cada píxel se li assigna un valor a partir de l'interpolació amb els vèrtexs. Durant aquesta era, el hardware permetia que aquestes funcions fixades fossin configurables i donava una mica de llibertat als programadors a l'hora de generar els seus gràfics.

## 2.3 El pipeline gràfic programable

Al 2001 nVIDIA va comercialitzar la seva GeForce 3[10]. La seva arquitectura permetia que els desenvolupadors d'aplicacions gràfiques programessin els seus propis shaders. Els shaders són petits programes que substitueixen a les etapes VS/T& L i *shading* de la figura 1. El pipeline fix es va convertir llavors en el pipeline programable figura 2. En les targetes més modernes, es permet fins i tot modificar la geometria dels models 3D.

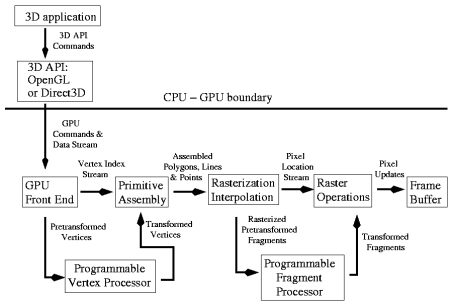


Figura 2: Pipeline programable

## 2.4 GPGPU

A mida que al hardware gràfic se li afegien més processadors, s'assemblaven cada cop més a CPUs d'alt rendiment paral·lel. Quan van aparèixer les primeres GPUs amb DirectX 9, alguns investigadors es van adonar que podien utilitzar-les per a resoldre problemes científics i d'enginyeria d'alt cost computacional, però les targetes, havien estat dissenyades per als requeriments de les API gràfiques.
Per a accedir a aquests recursos, els enginyers havien de transportar el problema a operacions gràfiques en les APIs OpenGL o DirectX per tal que poguessin ser executades. Per a executar vàries instàncies d'una funció, aquesta havia de ser escrita en un *pixel shader* i això permetia el càlcul en paral·lel que els científics necessitaven. El conjunt de variables d'entrada, s'havia d'emmagatzemar en una textura i enviada a la GPU com a part d'un triangle. Les dades de sortida, llavors eren obtingudes com un conjunt de píxels generats per les operacions *raster*.
El fet que el hardware de les GPU hagués estat dissenyat per a operacions gràfiques implicava que aquest mètode fos molt restrictiu per a una gran part d'aplicacions numèriques. A més el processador gràfic no tenia gaires capacitats per a llegir o escriure en memòria. Per a passar la sortida d'un programa com a entrada d'un altre, el framebuffer havia de ser convertit a textura. Un altre inconvenient d'aquesta metodologia era la inexistència de tipus definits per l'usuari en el llenguatge dels shaders, pel que els usuaris s'havien de conformar amb tipus de 1,2 o 4 components. Tot i aquests inconvenients, molts investigadors van optar per a utilitzar les GPUs com a

processadors de propòsit general, això va encunyar el terme GPGPU *General Purpose computing on GPU*

## 2.5 Els processadors gràfics unificats

L'arquitectura de les GPU de la generació anterior disposava d'unitats d'execució separades per al processament de *shaders* de fragment i de vèrtex. Aquesta divisió, tot i que tenia avantatges, també tenia un efecte negatiu en l'eficiència. Per exemple, en una escena amb un cost de *pixel shader* i poc de vèrtex, la unitat de *pixel shader* notava una manca de capacitat de procés, mentre la unitat de vertex estava ociosa, i viceversa. La única manera de solucionar aquest problema era unificant els processadors de *shaders* per tal de que la càrrega es pogués distribuir dinàmicament segons els requeriments de l'escena. Aquest exemple es pot veure il·lustrat a la figura 3. La videoconsola XBOX 360 conté el primer processador d'aquest tipus, el xip Xenos fabricat per ATI. Poc més tard nVIDIA presenta l'arquitectura G80, també amb processador gràfic unificat.

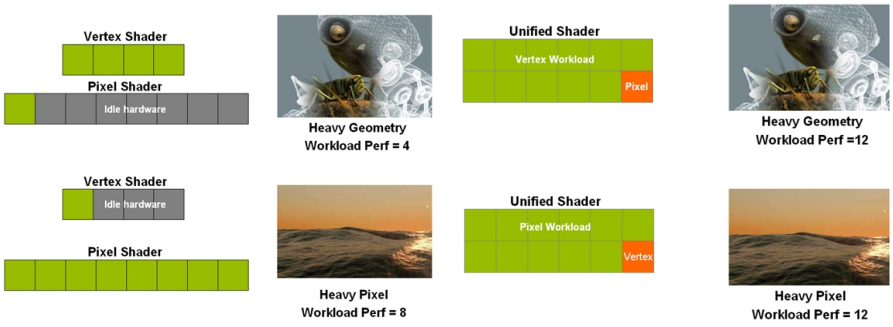


Figura 3: Diferència entre processador no unificat i unificat. Font: nVIDIA

## 2.6 L'arquitectura SM

La figura 4 mostra un exemple de l'arquitectura d'una GPU actual. El cor del xip són 128 SPs (*streaming processors*), que a la figura estan mostrats com a quadrats verds. Aquests, estan agrupats en 8 blocs anomenats SM (*Streaming Multiprocessors*). Cada bloc conté dos *shader processors*, (cada un amb 8 SPs) i tots 8 comparteixen lògica de control, caché d'instruccions i tenen accés a qualsevol de les sis cachés L2 i a qualsevol dels 6 *arrays* de registres de propòsit general. A l'arquitectura G80, a part d'aquestes característiques, cada SP disposa d'una unitat MAD(*multiply-add*), una de suma i d'unitats especials per a fer operacions en coma flotant com l'arrel quadrada i altres funcions. Cada SM G80 pot executar fins a 768 threads, el que suma aproximadament 12000 *threads*, i a l'arquitectura GT200 30000 *threads*. Segons nVIDIA la capacitat de computació és de 520 gigaflops.

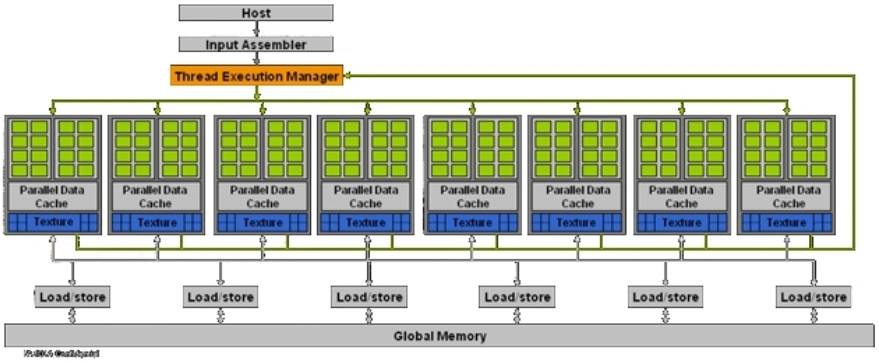


Figura 4: Arquitectura CUDA. Font: nVIDIA

# 3 CUDA

Com el títol de l'article a Dr. Dobbs indica[4] CUDA és el *super-computing for the masses* ja que, utilitzant hardware que es pot trobar a la majoria d'ordinadors personals. CUDA permet reescriure programes al seu model de programació i obtenir un rendiment dos ordres de magnitud superior al que es pot obtenir amb una CPU.
Quina és la diferencia entre CUDA i el GPGPU 2.4? CUDA és un acrònim per *Compute Unifided Device Architecture*. CUDA permet als programadors accedir al potencial de les targetes gràfiques sense necessitat de transportar els seus problemes a les APIs gràfiques OpenGL i DirectX. Així el programador pot obtenir rendiments superiors programant en C/C++, Fortran, Java o Python. En aquesta secció s'explica quines són les raons que fan que CUDA pugui obtenir molt més rendiment que una CPU en problemes altament paral·lelitzables i com s'aconsegueix. La figura 5 mostra el rendiment teòric que es pot aconseguir amb les GPU i les CPU actuals. La raó que hi

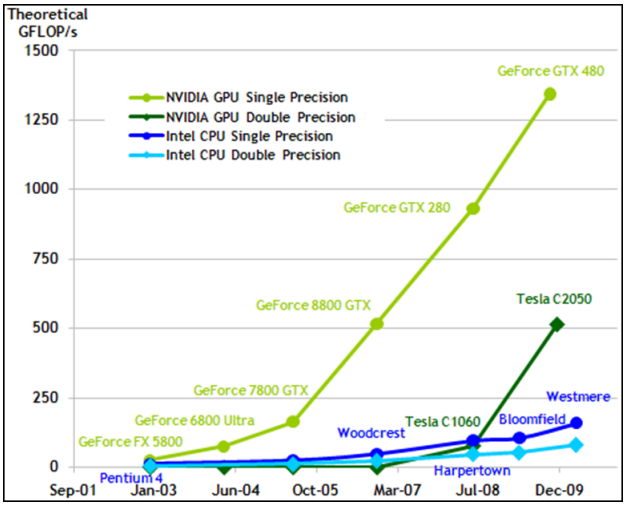


Figura 5: Flops teòrics de GPU i CPU. Font: nVIDIA

hagi aquesta diferència tan gran entre les operacions en coma flotat que pot obtenir una CPU i una GPU està en el propòsit del hardware gràfic. Aquesta inverteix més transistors en el càlcul de dades que en el control de flux o les memòries cau. La figura 6 mostra un esquema general de les dues arquitectures

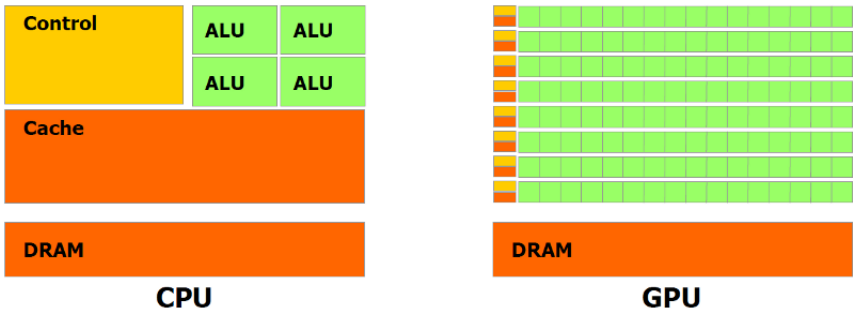


Figura 6: Esquema de l'arquitectura CPU i GPU. Font: nVIDIA

## 3.1 Model d'execució de CUDA

CUDA és una extensió del llenguatge de programació C, per tant, algú que està familiaritzat amb aquest llenguatge no ha de tenir problemes a l'hora de programar en CUDA. La unitat bàsica d'execució d'un programa en CUDA és el *thread*. Aquest s'agrupa en *blocks* que al mateix temps estan agrupats en *grids*. La figura 10 mostra d'una manera més visual com és aquesta organització.

### 3.1.1 Estructura d'un programa en CUDA

L'estructura d'un programa CUDA consisteix en una o més fases que són executades en el *host* o CPU i en la *device* o GPU. Les fases que tenen poc o gens paral·lelisme estan implementades en CPU. La porció de programa més paral·lelitzable s'implementa en codi *device*. El codi de device està escrit en ANSI C estès amb etiquetes per a identificar les funcions *kernel* i les estructures de dades associades. Els *kernels*, típicament generen un gran nombre de *threads* per a explotar el paral·lelisme entre les dades. Per exemple, en un problema de multiplicació de matrius, cada *thread* es pot ocupar de calcular un element de la matriu de sortida.
Cal notar que els *threads* CUDA són molt més lleugers que els de CPU. Es pot assumir que aquests *threads* triguen molt pocs cicles a generar-se i planejar-se, gràcies a l'eficiència del hardware, en contraposició als de CPU que típicament requereixen milers de cicles. La típica execució d'un programa en CUDA es pot observar en la figura 7. El programa comença amb l'execució en el *host*. Quan un kernel és invocat, el focus de l'execució es mou a la GPU, on un gran nombre de *threads* son generats per a aprofitar l'abundant paral·lelisme de dades. Tots els *threads* generats per un kernel durant una invocació s'anomenen col·lectivament *grid*. Com s'ha dit abans CUDA és una extensió d'ANSI C que introdueix algunes extensions. Per exemple, __global__ davant de la declaració d'una funció indica que la funció es un kernel i pot ser invocat des del *host* com a part d'un *grid*.
En general CUDA té tres paraules clau per a identificar les funcions. La figura 9 conté un resum d'aquests identificadors. Les funciones definides amb __device__ son porcions que només poden

ser invocades en GPU i executades en aquesta. La paraula __host__ davant d'una funció indica que només pot ser executada en el *host*. __device__ i __host__ no son exclusives entre elles.

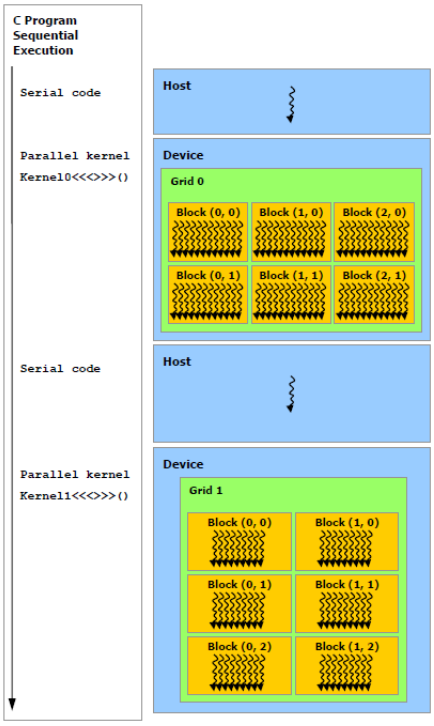


Figura 7: Típica execució d'un programa en CUDA

### 3.1.2 Kernels

En CUDA, un *kernel* especifica el codi que serà executat per tots els *threads* durant la fase d'execució en GPU. Com que tots els *threads* executen el mateix codi, el llenguatge CUDA, es un exemple del model de programació single-program, multiple-data (SPMD). La figura 8 mostra un exemple de *kernel* que calcula la suma entre dos vectors. Per a indicar que una funció és un *kernel*, a la definició ha de contenir la paraula clau __global__ . Una altra extensió notable de C, que es pot observar a la figura 8, és la paraula `threadIdx`, que indica l'índex del thread. Donat que tots els *threads* executen el mateix *kernel*, és necessari un mecanisme que permeti que els *threads* es puguin diferenciar entre ells i que es puguin adreçar a la porció de dades que els pertoca processar.

### 3.1.3 Jerarquia de threads

Quan un kernel és invocat, aquest s'executa com a part d'una *grid*. Els *threads* pertanyents a un *grid* estan organitzats en una jerarquia de dos nivells. Al nivell més alt, un *grid* està dividit en *thread blocks*. Tots els *blocks* que pertanyin a un *grid*, tenen el mateix nombre de threads. Cada *block* esta identificat per la tupla de dos dimensions `blockIdx`. Cada *block*, per sota, esta organitzat en grups de com a màxim 512 threads. Per conveniència l'identificador de thread

```
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

Figura 8: Exemple de kernel de CUDA

| etiqueta | executat a: | invocat des de: |
|---|---|---|
| __device__ | *device* | *device* |
| __global__ | *device* | *host* |
| __host__ | *host* | *host* |

Figura 9: Extensions de CUDA per a declarar funcions

`threadIdx` és un vector de 3 components per tal que cada *thread* pugui ser identificat per un índex d'1, 2 o 3 dimensions. Això proporciona un mode natural d'accedir a vectors, matrius o volums en memòria.

Quan el codi *host* invoca un kernel, s'assignen les mides del *grid* i del *thread block* via els paràmetres de configuració d'execució. És necessari declarar dos variables de tipus dim3. La figura 11 mostra un exemple d'execució de kernel amb 8 *blocks* de 128 *threads* cada un.

### 3.1.4 Jerarquia de memòria

En CUDA, el *host* i el *device* tenen dos espais separats de memòria. Per a poder executar un *kernel* en el *device*, el programador necessita reservar memòria en el *device* i transferir les dades pertinents del *host* al *device*. Similarment, després de l'execució, les dades resultants necessiten ser transferides de tornada cap a la memòria principal i alliberar l'espai reservat al *device* si no es necessita més. La figura 12 mostra una visió general de la jerarquia de memòria de les GPU. Tot seguit s'expliquen las particularitats de de cada tipus de memòria. També es pot veure la taula resum 1 per a una idea general.

**Registres** En general, l'accés a un registre consumeix zero cicles extra per instrucció però poden haver-hi retards, per read-after-write o conflictes en els bancs de memòria. La latència d'un read-after-write es aproximadament de 24 cicles però gràcies al mecanisme que s'explica en la secció 3.3, aquesta latència es pot amagar.

| Memòria | *Location on/off chip* | *Cached* | *Access* | *Scope & Lifetime* | |
|---|---|---|---|---|---|
| Register | On | n/a | R/W | *thread* | *thread* |
| Local | Off | $ | R/W | *thread* | *thread* |
| Shared | On | n/a | R/W | *Block* | *Block* |
| Global | Off | $ | R/W | *Grid* + *host* | *host* allocation |
| Constant | Off | Yes | R | *Grid* + *host* | *host* allocation |
| Texture | Off | Yes | R | *Grid* + *host* | Host allocation |

Taula 1: Jerarquia de memòria de CUDA

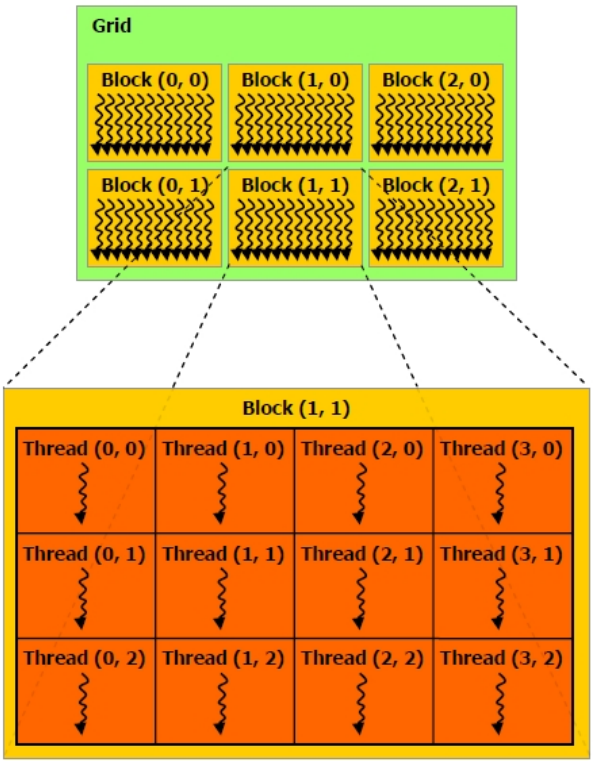


Figura 10: Jerarquia de *threads* en CUDA

```
dim3 block(64,2);
dim3 grid(4,2);
vecAdd<<<block,grid>>>(A,B,C);
```

Figura 11: Exemple d'execució d'un kernel

**Memòria compartida** Degut a la seva localització al xip, l'espai de memòria compartida, és molt més ràpid que la memòria local i global. Per a obtenir un bon ample de banda, la memòria compartida està dividida en mòduls de la mateixa mida anomenats *banks*, els quals poden ser accedits simultàniament. Qualsevol lectura o escriptura d'$n$ adreces a $n$ bancs diferents, pot ser servida simultàniament, donant un rendiment final que és $n$ vegades més gran que l' ample de banda d'un sol mòdul. Tot i així, si dues adreces de memòria cauen al mateix banc, hi ha un conflicte de banc i l'accés ha de ser serialitzat. El hardware s'encarrega de separar els accessos que generen conflictes en tants accessos com calgui, disminuint el throughput per un factor igual al nombre d'accessos a memòria separats.

**Memòria global** L'accés a memòria global es fa mitjançant transaccions de 32, 64, o 128 bytes. Quan els *warp* executen accessos a memòria global, uneix (*coalesces*) els accessos a memòria del *warp* en una o més d'aquestes transaccions. Les lectures o escriptures a memòria global de dades

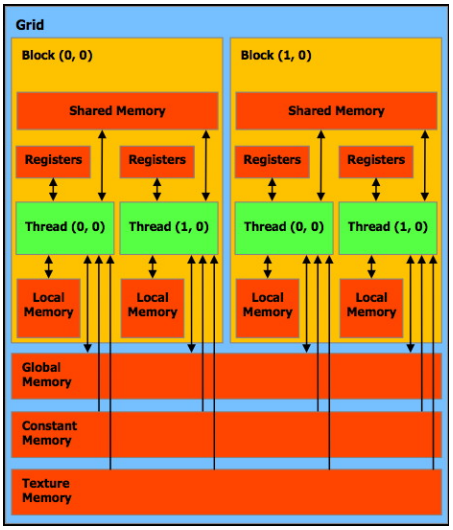


Figura 12: Visió general de la jerarquia de memòria. Font: nvIDIA

de mida 1, 2, 4, 8 o 16 bytes i que estan alineades a memòria només utilitzen un accés. Si les dades a les que es vol accedir no estan alineades, es fan 2 o més transaccions. És recomanable per tant, que els tipus de dades emmagatzemats a memòria, compleixin aquestes característiques. En *devices* de *compute capability* 1.0 o 1.1, el *thread* $k$ hauria d'accedir a la posició $k$. No cal que tots els *threads* facin accessos a memòria. Per a *compute capabilities* superiors, la unió de transaccions a memòria no necessita de cap patró d'accés.

**Local** La memòria local, s'anomena local per que el seu àmbit és local al thread. No per la seva localització física, en realitat, es troba a la memòria global. La memòria local només s'utilitza per a variables automàtiques. D'utilitzar o no aquesta memòria se n'encarrega el compilador.

**Memòria de constants** Hi ha un total de 64 kB de memòria de constants a cada device. L'espai de memòria de constants té una *cache* pel que una lectura a memòria de constants costa tant com una lectura cost a memòria global en cas de miss o només el cost de lectura en cache. Per a tots els *threads* d'un *warp*, llegir de memòria de constants, és tan ràpid com llegir els registres mentre tots els *threads* llegeixin la mateixa adreça. Els accessos a diferents adreces dins dels *half-warps* són serialitzats per tant el cost augmenta linealment amb el nombre d'adreces diferents per *half-warp*.

**Memòria de textures** La memòria de textures, té una *cache* igual que la de constants. Per tant un miss en lectura, costa un accés a memòria de dispositiu. La *cache* de textures, està optimitzada per localitat espacial en 2D, per tant, *threads* dins del mateix *warp* que llegeixen posicions de textura properes obtindran millor rendiment.

### 3.1.5 Sincronització i escalabilitat

CUDA permet que tots els *threads* dins d'un mateix *block* coordinin les seves activitats utilitzant la barrera de sincronització `__syncthreads()`. Quan un kernel crida a `__syncthreads()`, el

*thread* que executa la funció es mantindrà en aquella instrucció fins que tots els *threads* dins del *block*, arribin a aquella instrucció. Això assegura que tots els *threads* dins del *block*, han completat una fase de l'execució del *kernel* abans de passar a la següent. Cal tenir en compte que `__syncthreads()` ha de poder ser executat per tots els *threads* dins d'un *block*. Si la crida esta continguda en un `if`, o tots els *threads* dins del *block* executen el `__syncthreads()` o cap ho fa. En el cas d'un `if-else`, si cada branca conté una crida a `__syncthreads()` i dos *threads* dins d'un *block* prenen camins diferents en l'execució, farà que s'esperin a continuar l'execució indefinidament. El fet que existeixi un mecanisme de sincronització imposa restriccions d'execució en els *threads* dins d'un *block*. Aquests *threads* s'han d'executar propers temporalment per a evitar llargs temps d'espera. El motor de runtime de CUDA permet satisfer aquesta restricció assignant recursos d'execució a tots els *threads* dins d'un *block* com a unitat. Això vol dir que quan al *thread* d'un *block* se li assigna un recurs d'execució, a la resta dels *threads* del *block* se'ls hi assigna el mateix recurs, assegurant la proximitat temporal del *threads* dins d'un *block* i prevé llargs temps d'espera durant la sincronització. Al no existir sincronització fora del *block*, permet que els *blocks* s'executin en qualsevol ordre entre ells per que no s'han d'esperar entre ells. Aquesta flexibilitat permet l'escalat que es mostra a la figura 13. Un mateix programa es pot executar en dos dispositius amb diferents capacitats de processament i el resultat obtingut serà el mateix en les dues execucions sense que el programador hagi de preocupar-se de com escalarà el programa.

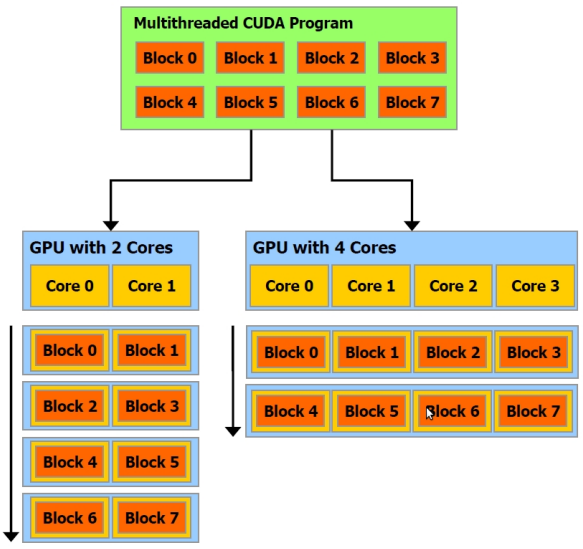


Figura 13: Escalat dinàmic en CUDA

## 3.2 Assignació de *threads*

Quan un kernel és llençat, el runtime de CUDA genera la corresponent *grid* de threads. Aquests *threads* són assignats a recursos d'execució *block* a *block*. En la generació de GPUs GT200, per exemple, els processadors contenen 30 SMs cada un d'aquests SM conté 8 SPs. A cada SP se li pot assignar un *block* en execució mentre hi hagi recursos suficients per a satisfer a tots els *blocks*. En una situació on una quantitat suficient d'un o mes recursos (veure secció 3.4.1), el

runtime de CUDA limita el nombre de *blocks* assignat a l'SM fins que d'utilització de recursos estigui per sota del límit. Amb 240 SPs en una targeta GT200, es poden assignar fins a 240 *blocks* als SP. La majoria de grids poden contenir més de 240 *blocks*, llavors el runtime manté una llista dels *blocks* que manquen per a executar i va assignant-los a mida que *blocks* anteriors han acabat l'execució.

## 3.3 Planificació de threads

En les implementacions de l'arquitectura GT200, una vegada el *block* ha estat assignat a un SM, es divideix en unitats de 32 *threads* anomenades *warp*. *Threads* individuals que formen part d'un mateix warp, comencen a la mateixa adreça de programa però cada un té el seu propi comptador de programa i estat de registres. Cada warp conté *threads* de `threadId` consecutius. Cada warp executa les instruccions comunes en paral·lel, per tant, quan tots els *threads* dins d'un warp tenen el mateix path d'execució s'aconsegueix el màxim rendiment. En canvi, si els *threads* dins d'un warp divergeixen en la seva execució, s'executa cada una de les branques en sèrie fins que tots els *threads* convergeixen un altre cop al mateix warp. El concepte de warp pren sentit quan s'introdueix les operacions d'alta latència com poden ser els accessos a memòria global. Quan una instrucció executada pels *threads* d'un warp ha d'esperar el resultat d'una instrucció anterior, aquest warp no s'executa. Un altre warp resident que no ha d'esperar és seleccionat per a l'execució. Aquest mecanisme s'anomena *latency hiding*.

## 3.4 Gestió de recursos i escalat transparent

### 3.4.1 *Compute capability*

Les GPU amb capacitat per a executar CUDA estan agrupades en grups segons la versió de l'arquitectura (G80, GT200, GF100) i la implementació del hardware. Aquesta classificació s'anomena *compute capability*. Cada *device* està definida per un *major revision* i *minor revision*. Les *devices* amb el mateix nombre *major revision*, tenen la mateixa versió de l'arquitectura. El valor de *minor revision*, indica millores en el *core* de l'arquitectura i la inclusió de noves característiques o funcionalitats. La figura 14 mostra la *compute capability* de la GPU on s'han fet les proves.

### 3.4.2 Relació entre el hardware i l'execució de CUDA

Els recursos d'execució en un SM són els registres, els *slots* de *thread block* i els *slots* de *thread*. Aquests recursos són dinàmicament particionats i assignats als threads per a la seva execució.

***Threads* per SM** S'ha vist en la secció 2.6 que l'arquitectura G80 té una capacitat de *thread slots* de 768 i l'arquitectura GT200 de 1024. Aquests *thread slots*, són particionats i assignats als *thrad blocks* en temps d'execució. Si cada *thread block* consisteix en 256 *threads*, els 1024 *threads* són particionats i assignats a 4 *blocks*. En aquest cas, cada SM podrà acomodar fins a 4 *blocks* degut a la limitació de threads per SM. Suposant un altre cas en que cada *block* contingués 128 threads, els 1024 *threads slots*, seran particionats i assignats a 8 *thread blocks*.

***Blocks* per SM** En l'exemple anterior s'ha plantejat una infrautilització dels blocks per SM per a aconseguir un ús ideal dels *thread slots*. Ara es plantejarà un cas on per tal d'acomodar el màxim de blocks s'infrautilitzen els *thread slots*. Suposem que tenim un *grid* amb 64 *threads* per *block*. Els 1024 *thread slots* dels que disposa l'SM poden ser dividits en 16 *blocks* però, degut a

```
Device 0: "GeForce GTX 280"
 CUDA Driver Version:                           3.20
 CUDA Runtime Version:                          3.20
 CUDA Capability Major/Minor version number:    1.3
 Total amount of global memory:                 1073414144 bytes
 Multiprocessors x Cores/MP = Cores:            30 (MP) x 8 (Cores/MP) = 240 (Cores)
 Total amount of constant memory:               65536 bytes
 Total amount of shared memory per block:       16384 bytes
 Total number of registers available per block: 16384
 Warp size:                                     32
 Maximum number of threads per block:           512
 Maximum sizes of each dimension of a block:    512 x 512 x 64
 Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
 Maximum memory pitch:                          2147483647 bytes
 Texture alignment:                             256 bytes
 Clock rate:                                    1.30 GHz
 Concurrent copy and execution:                 Yes
 Run time limit on kernels:                     Yes
 Integrated:                                    No
 Support host page-locked memory mapping:       Yes
 Compute mode:                                  Default (multiple host threads can use this device simultar
 Concurrent kernel execution:                   No
 Device has ECC support enabled:                No
 Device is using TCC driver mode:               No
```

Figura 14: Compute capability de la GPU nVIDIA GeForce GTX 280

que cada SM només disposa de 8 *block slots*, només 8 blocks podran ser assignats al SM causant que només s'utilitzin 512 dels 1024 *thread slots*.

**Registres per thread** Un altre recurs del que disposa CUDA són els registres. En l'arquitectura G80, els *devices*, tenen 8192 registres a cada SM. Aquests s'utilitzen per a guardar dades, generades pel programador i el compilador, i reduir la latència d'accés. Segons la implementació dels *kernels*, es requeriran més o menys registres. Particionant dinàmicament els registres entre els *blocks*, l'SM podrà assignar m és blocks si l'ús de registre per *thread* és baix o menys *blocks* si aquest és alt. Suposem que s'invoca una *grid* que està distribuïda en *blocks* de 256 *threads* i el compilador ens diu que cada *thread* necessita 10 registres per a la seva execució. La necessitat de registres per cada *block*, serà de 10x256 = 2560 registres. Si el device on s'executa aquest programa té un *compute capability* 1.1, el nombre de registres del que disposa és de 8192 per SM. El nombre de registres que es necessitaran per a executar 3 *blocks* es de 3*2560 = 7680 valor que està per sota de 8192 i es podran assignar 3 *blocks* al SM. El programador decideix aplicar una optimització i al compilar el *kernel*, després de l'optimització, l'ús de registres és d'11. Cada *block* de 256 *threads* necessitarà llavors 11x256=2816 registres. L'SM al assignar els recursos als *blocks* veurà que 2816*3 = 8448 registres són masses per a la seva capacitat i el nombre de *blocks* que s'assignaran a l'SM serà 2. L'ús de l'SM s'ha reduït en un 33%, esperem que l' optimització valgui la pena. Això explica que l'aplicació d'optimitzacions, pot aconseguir millora en el temps d'execució de *treads* individualment, però una pèrdua en el paral·lelisme de *threads*.

**Memòria compartida per *block*** Les *devices* tenen un límit de memòria compartida per SM en concret per a la sèrie GT200 són 16384 bytes. La gestió del nombre de blocks que es podran

assignar per SM, és similar a l'explicada anteriorment. Suposem un *kernel* que necessita 32 bytes per *thread*. Si s' agrupen els *threads* en *blocks* de 128. El màxim nombre de *blocks* que es podran assignar al SM és $\lfloor maxSharedSize/(blockDim * memPerThread) \rfloor$. En el cas de l'exemple s'hi assignaran 4 *blocks*.

# 4 El Distance Field

La primera aparició en la literatura del Distance Field, va ser al 1966 en un paper de processament d'imatge de Rosenfeld i Pfaltz [9]. Van proposar un mètode per a obtenir la mínima representació de l'estructura d'una imatge en escala de grisos. Els primers treballs amb l'SDF van ser utilitzats per al processament de la imatge. Avui en dia, un grup relativament gran de disciplines, fan ús de l'SDF: la física, visió per computador, gràfics i la robòtica en són un exemple.

## 4.1 Definició del problema

Donada $M$ una malla de triangles. La malla $M$ és un conjunt de triangles $T_i$ on $i \in [1, N]$ i $N$ és el nombre de triangles de la malla.

Assumint que tenim la malla de triangles $M$ i un punt $p$ definit en l'espai de 3 dimensions, la mínima distància des d'un punt $p$ a $M$ queda definida per l'equació 1

$$dist_M(p) = \inf_{x \in M} \|x - p\| \tag{1}$$

Si es vol diferenciar si el punt, és interior o exterior al sòlid que defineix la malla tancada, és necessari definir què és interior i què exterior. Es defineix $S$ com el conjunt de punts interiors a la malla. Els punts de l'univers tindran signe negatiu si pertanyen a $S$ i signe positiu en cas contrari. Afegint el càlcul del signe l'equació 1 esdevé l'equació 2

$$dist_M(p) = sgn(p) \inf_{x \in \delta S} \|x - p\| \tag{2}$$

on

$$sgn(p) = \begin{cases} 1 & si\, p \in S \\ -1 & altrament \end{cases} \tag{3}$$

## 4.2 Càlcul de la distància

La distància d'un punt a un triangle, es calcula utilitzant un anàlisi de casos. Quan el punt $p$ es projecta al pla que conté el triangle, el punt $p'$, cau en una de les set regions que es mostren a la figura 15. Si el punt $p'$ cau a les regions R6, R5 i R7, vol dir que la la part més propera al triangle és un vèrtex i es calcula la distància aquests. Si les regions son R2, R3 o R4, es calcula la distància a les arestes e1, e2, e3. I finalment si la projecció cau a la regió R1, la distància es calcula com la diferència entre $p$ i $p'$. Aquesta és la descripció del problema a grans trets. H. Eberly descriu matemàticament el procés per a fer aquest càlcul en el seu llibre [3].

## 4.3 Càlcul del signe

Els objectes representats com a malles de triangles, no estan formats per superfícies contínues, per tant, no tenen la normal definida per a tots els punts de la superfície (la superfícies és discontínua als vèrtexs i a les arestes). Tot i així es poden definir uns vectors als vèrtexs que posseeixen algunes de les propietats de les normals, aquests vectors s'anomenen pseudonormals. Hi ha una gran varietat de pseudonormals però per al cas de determinar si un punt és dins o fora d'una malla, les *angle weighted pseudonormals* proposades per Thürmer [12] tenen les propietats necessàries. Són J. Andreas i Henrik[7] els que van proposar i demostrar que les *angle weighted pseudonormals* es poden utilitzar per a resoldre el problema de trobar si un punt és interior o exterior a una malla de triangles. La pseudonormal en un punt està definida per l' equació 4 on $\alpha_i$ és l'angle de la cara $i$ incident al vèrtex i $n_i$ la normal de la cara. Veure figura 16.

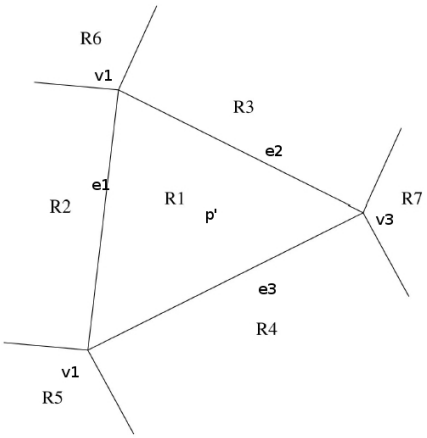


Figura 15: Regions d'un triangle per a calcular la distància d'un punt.

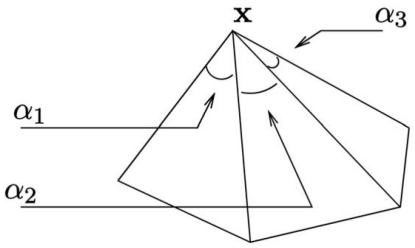


Figura 16: Angles incidents $\alpha_1$ , $\alpha_2$, $\alpha_3$ d'un punt $x \in M$

$$n_\alpha = \frac{\sum_i \alpha_i n_i}{\| \sum_i \alpha_i n_i \|} \tag{4}$$

Per saber si un punt p és interior o exterior a la malla, Andreas i Henrik proven que les equacions són certes per a un punt $p$ i un punt $c$ proper a $p$ i pertanyent a la malla.

$$n_\alpha(p-c) > 0 \,\text{si}\, p \,\text{exterior}$$
$$n_\alpha(p-c) < 0 \,\text{si}\, p \,\text{interior}$$
$$n_\alpha(p-c) > 0 \,\text{si}\, p \,\text{a la superfície}$$

## 4.4 Visió general de l'algoritme

La idea de l'algoritme és escollir un grup discret de punts dins del volum que envolta la malla i per a cada punt obtenir la distància més propera a la superfície de la malla. Aquest problema es pot enfocar de dues maneres que el bucle exterior siguin els punts del volum i l' interior les cares a consultar (figura 4.4) o la inversa (figura 4.4).

```
for all p ∈ volume do
  curr_min = ∞
  for all face ∈ M do
    if dist(face, p) < curr_min then
      curr_min = dist(face, p)
    end if
  end for
  dist[p] = cur_min
end for
```

Figura 17: Algoritme amb el bucle interior enfocat als punts del volum

```
for all face ∈ M do
  curr_min = ∞
  for all p ∈ volume do
    if dist(face, p) < curr_min then
      curr_min = dist(face, p)
    end if
  end for
  dist[p] = cur_min
end for
```

Figura 18: Algoritme amb el bucle exterior enfocat als triangles de la malla

# 5 Estat de l'art

Aquesta secció pretén mostrar les implementacions que existeixen actualment per al càlcul de l'SDF. Llegir els articles descrits a continuació ha permès obtenir idees per a la implementació

## 5.1 Implementacions en GPU:*DiFi:Fast 3D Distance Field Computation Using Graphics Hardware*

Sud et al. van publicar al congrés EUROGRAPHICS 2004 el seu article *DiFi:Fast 3D Distance Field Computation Using Graphics Hardware*[11]. En aquest article expliquen com utilitzar el hardware de rasterització i les apis gràfiques per a obtenir l'SDF d'un volum 3D discretitzat. La tècnica utilitzada consisteix en fer llesques del model i renderitzar-les a la GPU. Apliquen una optimització per a evitar pintar cares que no aportaran informació al resultat final. Obtenen resultats millors a altres implementacions existents fins al moment. Els resultats que van obtenir es mostren a la taula 2.

## 5.2 Implementació en CPU: *Signed Distance Computation Using the Angle Weighted Pseudonormal*

J. Andreas Bæretzen i Henrik Aanæs van publicar al 2002 *Generating Signed Distance Fields From Triangle Meshes* [2] i al 2005 *Signed Distance Computation Using the Angle Weighted Pseudonormal* [7] uns articles on proposen un mètode per a calcular l'SDF en una malla de triangles. En el primer article, es fa una proposta de l'algoritme demostrant teòricament que el seu algoritme es pot utilitzar per a calcular el signed distance field per a malles $C^0$ tancades. És en l'article del 2005 quan es realitza una implementació en la que aprofiten una jerarquia de caixes

| Model | Polys | Resolution | CSC | HAVOC | HAVOC+CSC | DiFi |
|---|---|---|---|---|---|---|
| Rotor | 4736 | 4x128x128 | 59.22 | 6.33 | 3.98 | 0.61 |
| Rotor | 4736 | 8x254x254 | 424.89 | 18.73 | 12.12 | 1.16 |
| Triceratops | 5660 | 128x56x42 | 127.81 | 2.11 | 1.10 | 0.79 |
| Triceratops | 5660 | 254x111x84 | 990.48 | 6.33 | 3.65 | 1.92 |
| Hugo | 17000 | 73x45x128 | X | 30.55 | 19.24 | 4.22 |
| Hugo | 17000 | 145x90x254 | X | 108.84 | 75.85 | 8.63 |
| Head | 21764 | 78x105x128 | 201.12 | 37.89 | 16.76 | 4.98 |
| Shell | 22598 | 254x252x252 | X | 162.97 | 95.31 | 7.79 |
| Cassini | 93234 | 186x254x188 | X | 356.03 | 298.55 | 29.86 |
| Dragon | 108926 | 57x90x128 | X | 171.13 | 95.69 | 24.76 |

Taula 2: Resultats de la implementació DiFi

| | | Hierarchy construction | | | | | | Distance Computation | | | | | | Ray Casting | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | No sign comp. | | With sign comp. | | Overhead | | No sign comp. | | With sign comp. | | Overhead | | | |
| Model | Polygons | OBB | AABB | OBB | AABB | OBB | AABB | OBB | AABB | OBB | AABB | OBB | AABB | OBB | AABB |
| Cube | 12 | 3.5e-4 | 1.2e-4 | 4.2e-4 | 1.9e-4 | 21% | 53% | 13.79 | 13.98 | 14.15 | 14.52 | 3% | 4% | 9.96 | 54.76 |
| Man | 3330 | 0.16 | 0.07 | 0.18 | 0.09 | 16% | 35% | 81.76 | 83.55 | 80.97 | 86.01 | -1% | 3% | 10.01 | 50.8 |
| Simple | 4096 | 0.18 | 0.08 | 0.21 | 0.11 | 17% | 37% | 62.43 | 110.5 | 62.73 | 114.2 | 1% | 3% | 30.12 | 172.5 |
| Pieta | 6976 | 0.37 | 0.16 | 0.43 | 0.22 | 18% | 39% | 88.21 | 89.39 | 88.66 | 91.99 | 1% | 3% | 24.65 | 114.3 |
| Femur | 7798 | 0.42 | 0.18 | 0.5 | 0.25 | 18% | 36% | 103.4 | 134.5 | 103.8 | 138.3 | 0% | 3% | 11.5 | 54.54 |
| Bunny | 11316 | 0.59 | 0.26 | 0.7 | 0.36 | 19% | 38% | 81.77 | 117.9 | 81.14 | 121.6 | -1% | 3% | 40.74 | 202.3 |
| Horse | 60680 | 3.69 | 1.57 | 4.36 | 2.21 | 18% | 41% | 138.7 | 226.9 | 137.7 | 235.3 | -1% | 4% | 19.2 | 93 |
| Dragon | 276680 | 19.01 | 8.62 | 22.63 | 12.39 | 19% | 44% | 148.3 | 225.1 | 146.9 | 231.8 | -1% | 3% | 40.75 | 180.2 |

Taula 3: Resultats de la implementació de Bæretzen i Aanæsi

englobants i donen resultats en temps. Per a obtenir els resultats de les seves proves han utilitzat els models dragon i horse que s'utilitzaran per a comparar el rendiment d'una implementació en CPU eficient contra la implementació força bruta que s'ha implementat en CUDA.

### 5.3 Implementacions en CUDA: *CUDA-base Signed Distance Field Calculation for Adaptive Grids*

La única implementació en CUDA que s'ha pogut trobar per a solucionar el problema del càlcul de l'SDF, és la proposta de Park et al.[6] En lloc d'utilitzar una malla de punts regular per a obtenir l'SDF de l'objecte, proposen generar un arbre adaptatiu. Aquest arbre, permet obtenir més nivell de detall a les regions properes a la superfície i menys detall a les llunyanes. La construcció de l'arbre adaptatiu es genera a la CPU mentre que la GPU la utilitzen per a, donat un punt $p$ i una malla $M$, obtenir la distància mínima de $p$ a $M$. En resum, el procés per al qual han utilitzat CUDA, és per fer un accés a memòria calcular la distància i retornar el mínim valor amb un procés de reducció. Els resultats que mostren no són gaire precisos ja que no es mostra cap taula amb dades sinó gràfics de temps que només es poden observar qualitativament. Segons el gràfic de la figura 19 obtenen un speedup de 60x respecte la implementació en CPU del mateix algoritme. Tot i així el temps d'execució augmenta exponencialment amb els nivells de l'arbre generat.
Aquesta implementació no es pot comparar amb la d'aquest projecte per que el format de les

dades que retornen està en forma d'arbre

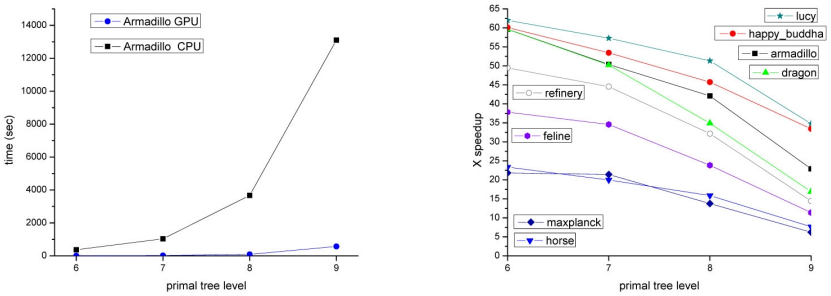


Figura 19: Resultats de Park et al.

# 6 Implementació força bruta amb *tiling* en CUDA

La primera implementació a provar és la versió força bruta de l'algoritme. S'anomena força bruta per que no hi ha cap mecanisme per a descartar cares per tant és un algoritme de tots contra tots. *Tiling* indica que l'accés i el càlcul dels punts es fa a trossos de mida *blockDim*. Per simplificar, en el document es referirà a aquesta implementació com a *tiling* CUDA Donat un volum que conté N vòxels i una malla amb $N$ triangles, el cost d' aquest algoritme és de $N * M$. Aprofitant el poder de paral·lelisme de CUDA, es pretén reduir temps d' execució de les implementacions utilitzades com a referència a la secció 5 Donat que el càlcul de l'SDF es pot entendre com una relació de molts a molts 4.4, el problema es pot enfocar de vàries maneres. Assignar a cada *thread* de CUDA una cara i que aquesta iteri sobre tots el punts del volum i desi la seva distància si és menor a la que ja hi havia abans en aquest punt. L'altra opció és assignar un *thread* per punt i que aquest visiti totes les cares i es desi la menor distància trobada. S'ha escollit una variant del segon. La següent secció pretén descriure quin ha sigut el mapeig del problema a CUDA i quines optimitzacions s'han aplicat per a millorar el seu rendiment,

## 6.1 Visió general de la implementació

Es podria imaginar el problema com llençar $N$ *threads*, un per cada punt del volum, i que cada un iteri per les $F$ cares del model. El *thread i* es queda amb la mínima distància trobada i abans d'acabar desa a la posició de memòria adient el valor trobat. Aquest enfoc (figura 20) aprofita el paral·lelisme de calcular varis punts simultàniament però necessita que cada *thread* porti de memòria global cada una de les cares, repetint així la feina que fan els seus companys de *block*.

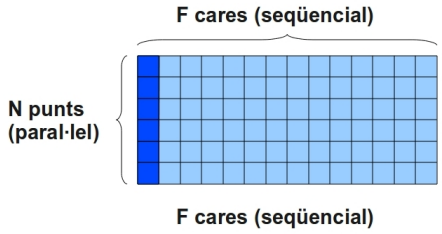


Figura 20: Esquema de l'algoritme 1

La solució proposada consisteix en llençar *blocks* de *blockDim threads*. Cada *thread i* s'ocupa de portar la cara $i$ del model de memòria a registres. En aquest moment, el *thread i* pot calcular la distància a la cara i desar-la a memòria si és la distància mínima fins la moment. Aprofitant que el *thread i* s'ha portat de memòria la cara $i$ pot estalviar-li la feina als *threads* amb els que comparteix *block* i evitar que aquest hagin de llegir totes les cares de memòria. Seguint aquest plantejament, els *thread i*, pot saber quines son les posicions dels *threads* amb els que comparteix *block* i calcular la distància per a cada un dels punts del *block* a la cara que ha llegit. Amb un accés a memòria per thread és suficient per a calcular les distàncies de *blockDim threads*.

Donat que en general els models tindran més cares que *threads* hi ha en un *block*, s'ha d'aplicar *tiling*, i iterar per conjunts de cares de mida `blockDim`. La figura 21 mostrar un esquema d'aquesta modificació.

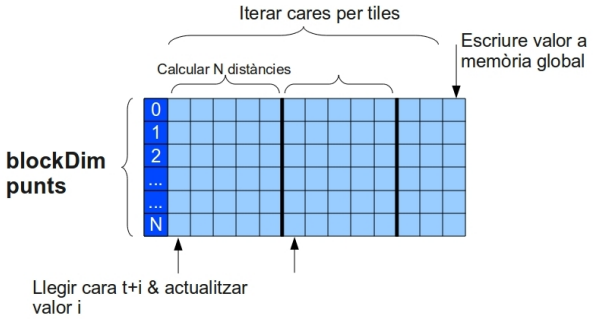


Figura 21: Esquema de l'algoritme 2

## 6.2 Preparant les dades d'entrada

Com a paràmetres d'entrada es necessiten les cares i la mida del volum per a fer els càlculs pertinents. No cal enviar a cada *thread* quina és la posició en el volum que li pertoca ja que sabent l'identificador del *thread* i la mida del volum, es pot calcular fàcilment(veure C.1). Una representació comuna de les malles de triangles és el format obj. Aquest conté una llista de cares i una llista de vèrtexs. La llista de cares conté apuntadors als vèrtexs que formen cada cara. Aquesta distribució pot originar que dos vèrtexs que són propers, estiguin en posicions de memòria llunyanes. Com que cada cara conté 3 vèrtexs, s'ha optat per a enviar a CUDA, una llista de vèrtexs. D'aquesta manera només cal un accés a memòria per vèrtex i vèrtexs pertanyents a una mateixa cara estan en posicions de memòria consecutiva. Per a calcular el signe del DF, són necessàries les pseudonormals del vèrtex. El càlcul d'aquestes s'ha efectuat en CPU per a després enviar-les a la GPU. S'ha decidit no *des-indexarles* com en el cas de les cares ja els *threads* només s'hi accedirà un cop per kernel després de saber quina és la cara amb distància mínima.

## 6.3 Càlcul de la distància

Per a calcular la distància dels punts del volum a la malla, els vèrtexs es transformen i es porten a la base on es troba el volum de punts. Això s'aconsegueix fent una translació i un escalat que a la pràctica és una suma i un producte. En aquest punt, el *thread i thread* ja té les coordenades del triangle en la base en la que es troba le volum. Com que es llença un *thread* per a cada punt del volum, coneixent el `threadIdx` del *thread* i la mida del volum de punts, es poden obtenir les coordenades del punt en qüestió (veure funció get3DPoint) i calcular la distància mínima a la cara *i*. S'ha observat que aquesta funció s'executa en el bucle intern dels *kernels* i que això pot impactar molt en el rendiment de l'execució. Per a solventar aquest problema, el *thread i* fa el càlcul de les seves coordenades en 3 dimensions i les escriu a la *shared memory*, d'aquesta manera, els *threads* companys de *block* quan hagin de calcular la distància al punt *i* accediran a la memòria compartida evitant una crida a funció amb dues divisions, dos mòduls, un producte d'enters, i una conversió a float.

S'han implementat dues versions per a calcular la distància des d'un punt a un triangle. La primera només calcula la distància als vèrtexs del triangle, això es converteix en 3 operacions en coma flotant i el càlcul del mínim de tres valors. La segona versió és la implementació que proposa Eberly en el seu llibre [3] amb algunes modificacions per adaptar els valors d'entrada als tipus que s'ha utilitzat en el projecte. En l'apartat de resultats es podrà veure quin és l'error que

introdueix utilitzar la funció més senzilla i com varia el rendiment segons la funció de distància utilitzada. Un altre punt important a tenir en compte és que es calcula la distància al quadrat, i en el moment que s'han trobat les distàncies mínimes i s'han d'escriure a memòria es calcula l'arrel quadrada de la distància al quadrat. Això permet estalviar el càlcul d'arrels quadrades en el bucle més intern del *kernel* i obtenir una millora de rendiment considerable.

## 6.4 Càlcul del signe

Per a calcular el signe, s'utilitza l'algoritme descrit a la secció 4.2 amb algunes modificiacions. Les pseudonormals s'envien en format indexat. S'ha decidit fer-ho així per que només s'hi accedirà un cop per *thread* no com en el cas de les cares que s'han de visitar $F/blockDim$. El signe es calcularà una vegada s'hagi obtingut la mínima distància de totes les cares al punt. Això vol dir just abans d'escriure a memòria el valor corresponent. Per tal de calcular el signe és necessari que el *thread* sàpiga quina és la cara amb la que s'ha trobat la distància mínima. A la fase del càlcul de distància es comprova si la actual és menor que la anterior. En cas de ser cert, s'escriu a la *shared memory*. S'ha afegit una instrucció que també desa l'índex de la cara a la memòria compartida. D'aquesta manera, el *thread* encarregat d'escriure el valor de sortida, llegint l'índex de la cara, pot cridar a la funció de càlcul de signe i obtenir-lo

## 6.5 Optimitzacions

### 6.5.1 La importància del format de les dades d'entrada

Com s'ha comentat anteriorment a la secció 6.2 les dades d'entrada s'han adaptat per varis motius, aquí se'n mostren els detalls. La primera justificació és el nombre d'accessos a memòria. Utilitzant el mode de dades indexat, calen 6 accessos a memòria per a obtenir els vèrtex d'una cara, 3 per a obtenir els índexs de vèrtex i 3 més per a accedir als vèrtex en qüestió. Amb les modificacions fetes a les dades, es pot accedir directament als vèrtexs utilitzant només 3 accessos a memòria. Un altre avantatge que té aquesta representació és que les dades que hauran de visitar *threads* veïns, estan en posicions de memòria consecutives i es pot aprofitar el mecanisme que tenen les GPU per a portar dades. Per últim, l'accés a memòria global permet llegir valors de mida 1, 2, 4, 8 o 16 bytes. Qualsevol accés a variables d'aquesta mida es transforma en un accés. En canvi, si s'utilitzen variables de mida diferent, el compilador converteix aquestes lectures en diversos accessos a memòria. La mida d'una variable de tipus float3 és de 12 bytes, per tant en molts casos, les variables no estaran alineades en memòria a 16 bytes i caldran 2 accessos enlloc d'un per a llegir una variable. Amb tots aquests avantatges, l'únic inconvenient que presenta aquesta distribució de les dades és l'espai en memòria que ocupen els models una vegada carregats( veure la taula 10). Sigui F el nombre de cares i V el nombre de vèrtexs, l'equació que descriu l'espai que ocupa una malla en memòria es troba a l'equació 5.

$$F * sizeof(int) * 3 + V * sizeof(float) * 3 \tag{5}$$

En general a les malles de triangles es compleix aquesta propietat $2 * V = F + 4$. Substituint F a l'equació anterior i simplificant, s'obté l'equació 6 que descriu l'espai en memòria només en funció de les cares.

$$3 * sizeof(int) * (3 * (\frac{F+4}{2})) \tag{6}$$

Si no s'utilitza el mode d'accés indexat per cares i s'afegeix padding per a millorar l'alineament en memòria, l'equació d'ocupació de memòria d'una malla és $F*3*4*sizeof(int)$ bytes. Comparant un amb l'altre queda que el segon mode ocupa 8/3 parts més que l'indexat. El model més gran

que s'ha provat és el buddha que conté 1000000 cares aproximadament. Gràcies a que les GPU actuals disposen de com a mínim 1 GB de memòria global, una ocupació de 18 MB a 50 MB sobre 1024, significa un increment del 3%, el qual és un preu bastant baix a pagar per aconseguir un codi que fa la meitat d'accessos a memòria a més de ser alineats. Per provar l'impacte del *coalescing* en una GPU de *compute capability* 1.1, s'ha implementat una funció especial. Que retorna la llista de vèrtexs ordenats de manera que el thread $k$, llegeix la posició de memòria $k$. Aquest reordenament depèn de la mida del *block*. A la figura 22 es veu com s'han redistribuït les dades per tal d'afavorir l'alineament en lectura.

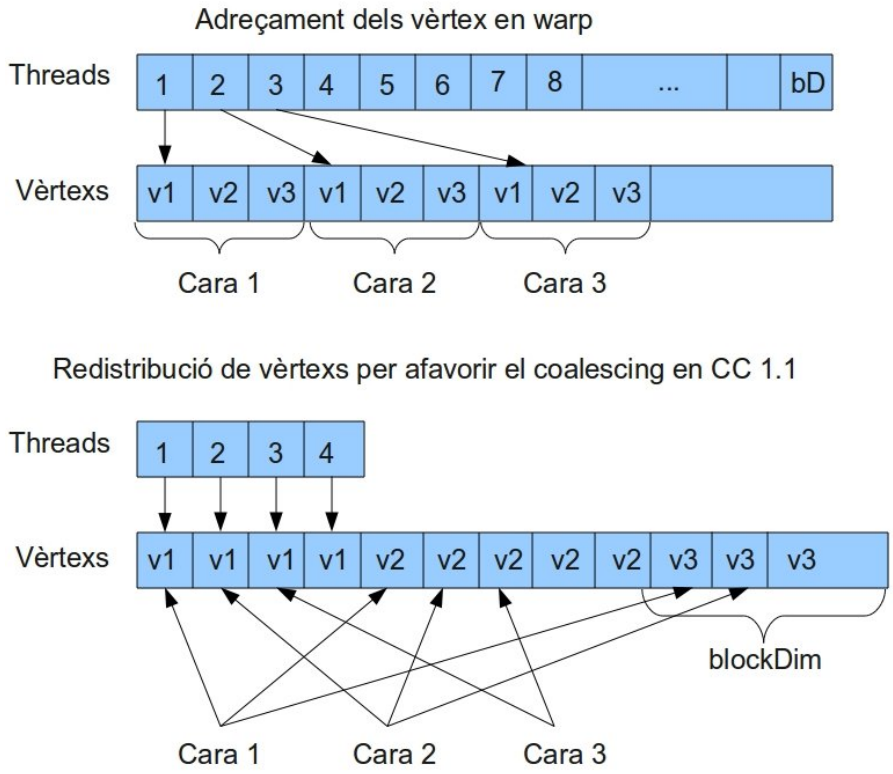


Figura 22: Esquema de redistribució de les dades d'entrada per afavorir *coalescing*

### 6.5.2 *Shared memory*

No té sentit programar un kernel en el que els *threads* interactuïn entre ells a través de la memòria global. Per aquesta raó, des del principi, la implementació s'ha pensat des del primer moment utilitzant la memòria compartida. A la secció 6.6.1 s'explica més detalladament l'ús que es fa de la *shared memory*.

### 6.5.3 Memoització

La funció `get3DPoint` calcula per un identificador de *thread*, quines son les coordenades en 3 dimensions sobre les que haurà de fer el càlcul de la distància. Per tal d'evitar que el bucle

interior dels *kernels* hagi de fer el càlcul de les coordenades dels seus veïns, a l'inici del kernel, cada *thread* calcula quina és la seva posició i la desa a la *shared memory* per a que els *threads* companys de *block* no hagin de calcular el punt.

## 6.6 Altres detalls de la implementació

### 6.6.1 Memòria compartida

Per a maximitzar el rendiment, la interacció entre *threads* del mateix *block* s'ha de fer utilitzant la memòria compartida enlloc de la global ja que el cost de lectura en aquesta memòria és d'un cicle versus els 400 o 600 cicles de la memòria global. Com també s'ha de conservar la cara per la que la distància és menor donat un punt, s'utilitzarà la memòria compartida de la mateixa manera que per les distàncies i una vegada s'hagi acabat el procés de trobar les distàncies mínimes, Per a permetre un bon escalat, la reserva de la memòria compartida es fa en funció dels *threads* per *block*. En total per a cada thread es necessitara un `float4` per a desar el punt en 3 dimensions que li pertoca al `thread`, un float per a la distància mínima i un int per a l'índex de la cara amb la que té la distància mínima. Donat un block de mida `blockDim` la quantitat de *shared memory* que s'aurà de reservar seran (sizeof(float4)+sizeof(float)+sizeof(int))*blockDim bytes. En l'arquitectura que s'ha utilitzat aquest valor és igual a 24*blockDim bytes.

### 6.6.2 Organització de *threads* i *blocks*

Donat que l'enfoc que s'ha dissenyat és assignar un *kernel* per a cada punt, el nombre de kernels de CUDA a invocar serà igual al nombre de valors que es volen obtenir dins del volum. Si es vol analitzar un volum de $N^3$ mostres en una GPU que té una limitació de mida de *block* de *maxBlockDim*, el *grid* resultant són $\frac{N^3}{maxBlockDim}$ *blocks* de mida *maxBlockDim*. Per exemple si es vol analitzar un model amb $32^3$ mostres, s'han de llençar 32768 *threads*. Donat que el màxim nombre de *threads* per *block* en les GPU de desenvolupament i de test és de 512. Els 32768 *threads* s'hauran d'agrupar en una *grid* de 64 *blocks* amb 512 *threads* cada un. D'aquesta manera però, no s'aconseguirà un ús efficient dels multiprocessadors ja que amb la limitació de 1024 *threads* per MP, només 2 *blocks* podran ser executats en aquell MP. S'ha decidit escollir una mida de *block* de 128 ja que és el que permet una màxima ocupació de *blocks*. Una altra raó per a reduir la mida de 512 a 128 és l'ús de la shared memory. Amb 512 *threads* per *block* es necessitaran 512*24 = 12288 bytes. Amb un límit de 16384 bytes per MP, només un block podrà ser assignat per MP reduint considerablement el rendiment. El codi del *kernel* es pot trobar a l'annex 53.

### 6.6.3 Sincronització de *threads*

Quan els *threads* d'un *warp* ja ha iterat per totes les cares i ha escrit les distàncies mínimes corresponents, ja no ha de fer més càlculs i poden portar les dades de *shared memory* a memòria global. No obstant, que el seu *warp* hagi arribat a aquell punt de l'execució no implica que la resta de *warps* del *block* ho hagin fet. Per a evitar escriure valors erronis a la memòria global, els warps s'han de sincronitzar en aquest punt i esperar a que totes les dades correctes estiguin a la memòria compartida per procedir a escriure els valors a memòria global.

# 7 Millora amb *bricking*

L'objectiu d'aquesta millora és reduir el nombre de cares sobre les que ha d'iterar cada *kernel* i aconseguir una millora en el temps d'execució. L'algoritme consisteix en classificar les cares A l'algoritme original se li afegeixen dues variables, el nombre de divisions i factor d'*overlapping*. Un valor de subdivisió igual a 1 dona el volum original. Amb el nombre de subdivisions es fa un proces en el volum i s'obtenen els *bricks*, aquests són porcions del volum, els quals la seva unió genera el volum inicial. Per a destriar les cares, s'itera sobre totes elles, i si estan contingudes dins d'un *brick* de classificació, es desen en un array per a després passar-lo com a paràmetre al *kernel*. Donada una malla amb $F$ cares, un volum de dimensió $N^3$, i un nombre de subdivisions $S$. Les cares per les que tenien que iterar, els threads era Igual a $F$. Amb En l'algoritme inicial, el nombre de cares per les que iteraven els *kernels* eren F. Suposem que una malla té els triangles uniformement repartits per tot el volum. El nombre de cares per les que haurà d'iterar el *kernel* és $(F/S^3)$ això implica un speed-up per thread de $S^3$x. Cal tenir en compte però que el recorregut per les cares, és costós. Aquest plantejament presenta un problema que esta il·lustrat a la figura 23. Donat un punt contingut dins d'un *brick*, la seva distància mínima a la superfície de l'objecte seran les cares incloses en el *brick*. Això pot originar errors. Per a solventar aquest problema s'introdueix la variable factor d'*overlapping*. Aquesta permet engrandir la mida del *brick de classificació*. Un *overlapping* major que 0, afegirà cares properes al *brick* i permetrà obtenir resultats més precisos. La figura 24 mostra com amb diferents valors d'*overlapping* el resultat final s'acosta més a l'ideal. Una altra característica, que es pot interpretar com a bona o dolenta, d'aquesta millora són els punts amb valors no assignats. Si un *brick* no conté cap cara, no es llançarà la grid de *kernels*, això suposa un estalvi en temps d'execució. L'inconvenient d'aquest fet és que tots els punts dins del *brick* no tindran cap valor assignat. En la majoria de casos només es necessiten els valors propers a la superfície del model per tant, això no és un inconvenient. En canvi si per l'ús que se li vol fer es necessita, caldrà incrementar el factor d'overlapping fins que tots els punts del volum tinguin valors assignats. Cal tenir en compte que si $overlap \geq subdiv - 1$ el nombre de cares a processar per cada kernel, serà el mateix que en l'algoritme original.

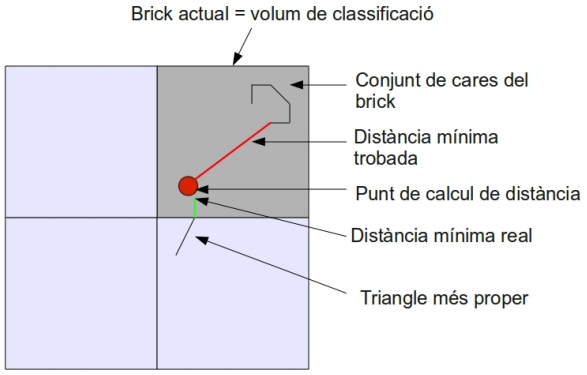


Figura 23: Problema de l'algoritme amb *bricking*

El procés per a eliminar cares, consisteix en definir un nombre de subdivisions del volum i un factor d'overlapping. Cada subdivió s'anomenarà *brick*, i per a cada *brick*, es llançarà un grid de threads amb les cares que estiguin incloses d'aquest brick Es llença un grid per a cada *brick*

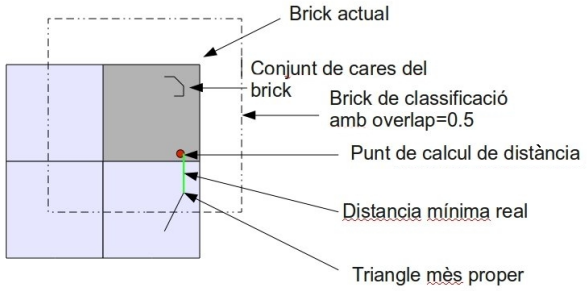


Figura 24: Solució a l'error del *bricking*

Akenine proposa en el seu article *Fast 3D Triangle-Box Overlap Testing*[1] un mètode efficient per veure si un triangle intersecta amb una caixa però s'ha optat per utilitzar un mètode més senzill. Com que els volums sobre els que es mirarà si el triangle està dins de la caixa, seran molt més grans que els triangles, en l'algoritme utilitzat, només es comprova si algun dels vèrtex està contingut dins de la caixa. Aquest algorisme té dos paràmetres configurables, un és el nombre de subdivisions en cada una de les dimensions i l'altre el factor d'overlapping.

# 8 Altres implementacions

## 8.1 CPU

Per provar com afecta el transformar d'un problema de cost $N^3 * M$ a CUDA, s'ha implementat una versió del càlcul del DF en CPU. Aquesta, correspon a l'algoritme descrit a la figura 4.4. No se li ha afegit cap optimització. Per a comparar amb una versió optimitzada en CPU, s'utilitzaran els resultats obtinguts per la implementació de Bæretzen i Henrik Aanæ [7].

## 8.2 Força bruta amb recorregut per cares seqüencial

Per últim i per provar que fent ús de la *shared memory* i reduint el nombre d'accessos a memòria s'aconsegueix més rendiment, s'ha implementat la versió del codi seqüencial descrita a la figura 20. Les funcions utilitzades per al càlcul de la distància i l'accés a memòria es fa de la mateixa manera. A aquesta implementació se l'anomenarà seqüencial CUDA. El codi del *kernel* es pot trobar a l'annex B.2.

# 9 Representació del DF i el model gràficament

Per a mostrar l'obtenció de l'SDF, s'ha utilitzat la llibreria gràfica OpenGL i el llenguatge de *shaders* GLSL. Per la implementació, els valors de l'SDF estaran continguts entre 0 i la mida màxima del volum. L'SDF s'envia al *shader* com a una matriu tridimensional de floats que al ser processat pel *shader* ho convertirà a colors segons el valor de la distància. El problema amb el que ens trobem és que OpenGL només pot llegir valors que van de 0 a 1 per tant, aquests, s'han de normalitzar. El procediment per a normalitzar els valors consisteix en trobar el valor màxim i mínim i dividir tots els elements de la matriu per la diferència del màxim i mínim. D'aquesta manera s'obtenen valors entre 0 i 1 que el *shader* pot interpretar. S'ha decidit que per a mostrar l'SDF el codi de colors, de més proper a més llunyà serà: vermell, groc, verd, blau. A l'annex D es troba la implementació dels *shaders*. La figura 25 mostra la representació gràfica del DF dels models armadillo, bunny i horse. Es pot observar que les regions més allunyades de la superfície són de color blau i les properes de vermell.

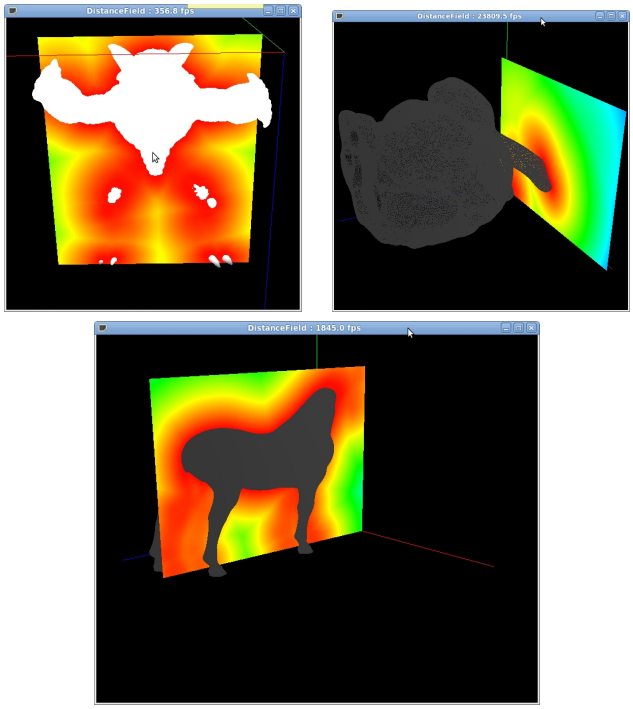


Figura 25: Representació del DF en OpenGL

# 10 Resultats

Aquestes proves s'utilitzaran per a comparar el rendiment de la implementació del projecte amb les existents. També s'utilitzaran per a comprovar si algunes de les suposicions fetes son certes i per a treure conclusions del treball. Les malles de triangles utilitzades per a realitzar els benchmark, es poden veure a la figura 26 i a la taula 10 es pot veure la seva complexitat i ocupació en memòria. Nota: Totes les dades de temps estan mostrades en ms. Per limitacions del driver de CUDA en les GPUs utilitzades per a les proves, les *grids* no poden excedir un temps d'execució de 10 segons, en cas contrari, el driver aborta l'execució del *kernel*.

| malla | # cares | # vertex | MB en memòria | MB redundància i padding |
|---|---|---|---|---|
| armadillo | 345944 | 172974 | 5.93 | 15.83 |
| dragon | 871414 | 437645 | 14.98 | 39.89 |
| bunny | 69451 | 35947 | 1.2 | 3.18 |
| horse | 96966 | 48485 | 1.66 | 4.43 |
| happy Buddha | 1087716 | 543652 | 18.66 | 49.79 |
| angel | 474048 | 237018 | 8.13 | 21.7 |
| sphere5k | 5120 | 2562 | 0.17 | 0.23 |

Taula 4: Complexitat de les malles de prova

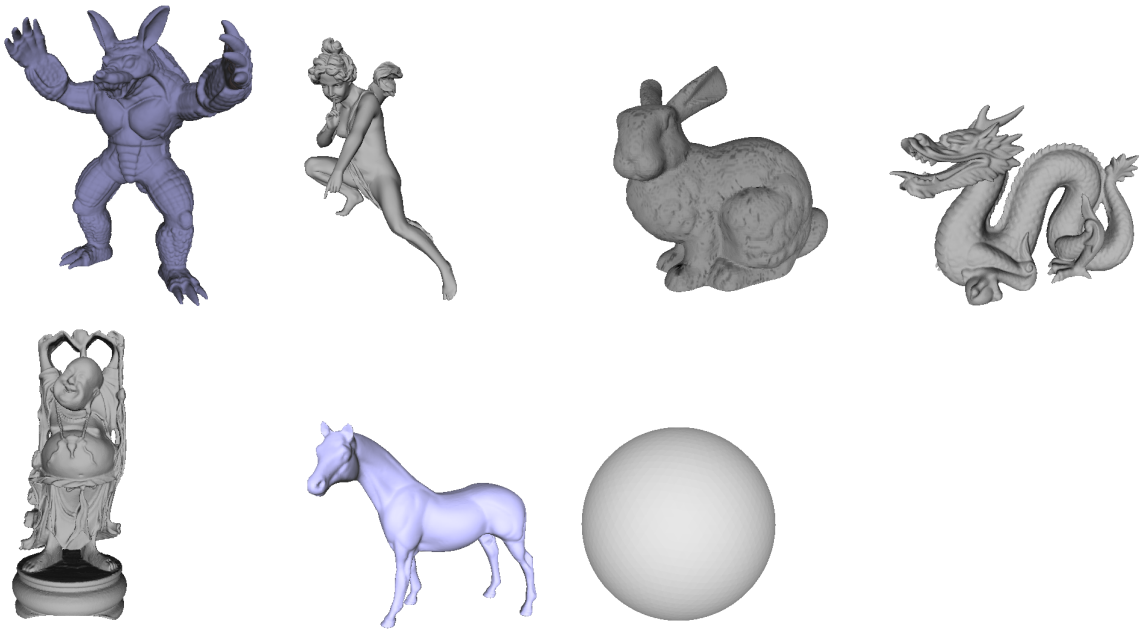

Figura 26: Malles de prova

## 10.1 Nombre de *threads* llençats per model i volum

El paràmetre d'entrada `maxvoldim` serveix per a definir el nombre de mostres que es prendran en la dimensió màxima de l'objecte a analitzar.

| model | dimensio | punts |
|---|---|---|
| horse | 16 | 1792 |
| horse | 32 | 12960 |
| horse | 64 | 103680 |
| sphere | 16 | 4624 |
| sphere | 32 | 36992 |
| sphere | 64 | 295936 |
| angel | 16 | 1120 |
| angel | 32 | 8960 |
| angel | 64 | 67392 |

Taula 5: Punts a mostrejar per model i mida del volum

**Observacions:** En la implementació que s'ha fet es llencen tants *threads* com mostres es volen obtenir. A la taula, es veu que les proporcions de la figura influeixen en el nombre de threads que es necessiten llençar per a obtenir el seu DF. Si la caixa contenidora del model té forma aproximadament cúbica, com pot ser el cas de l'esfera, el nombre de threads necessitats, augmentara cúbicament en funció de la dimensió màxima. En canvi si el model s'assembla més a un cub aplanat el nombre de *threads* augmentarà quadràticament i si és de forma rectilínia, com l'angel, linialment. Aquestes dades poden servir per avaluar el cost en funció del volum i les proporcions de la caixa contenidora de l'objecte.

## 10.2 Força bruta: CUDA

### 10.2.1 Proves mida de block

Per a comprovar que la mida de *block* de 128, és el millor que es pot utilitzar, s'han executat proves amb 3 models amb mides de *block* múltiples de 32, des de 32 fins a 512. Cada prova s'ha executat 10 cops i el valor mostrat és la mitja de totes. La figura 27 mostra el gràfic per als models horse, bunny i angel.

**Observacions** Es pot veure que una mida de *block* de 128 *threads* aconsegueix la màxima ocupació del SM.

### 10.2.2 Proves temps en dimensió màxima

L'objectiu d'aquestes proves és veure el temps d'execució a mida que augmenta la mida del volum de mostres i com el temps per thread es veu afectat segons el nombre de cares que té el model. Es faran proves amb els models descrits anteriorment amb volums de dimensió màxima de 8, 16, 32, 64 i 128, 256. Els resultats d'aquestes proves es mostren a la figura 28. L'escala del gràfic és logarítmica.

**Observacions** Per a les figures horse, bunny i angel, el temps d'execució és el mateix per al càlcul amb volum 8 i 16 això és degut a que hi ha una infrautilització de la GPU.

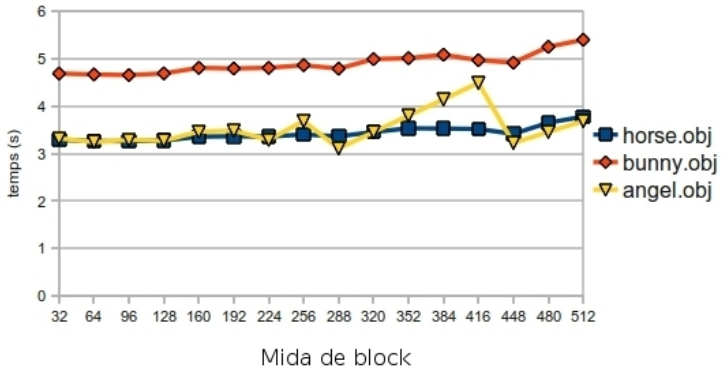


Figura 27: Proves amb mida de *block*

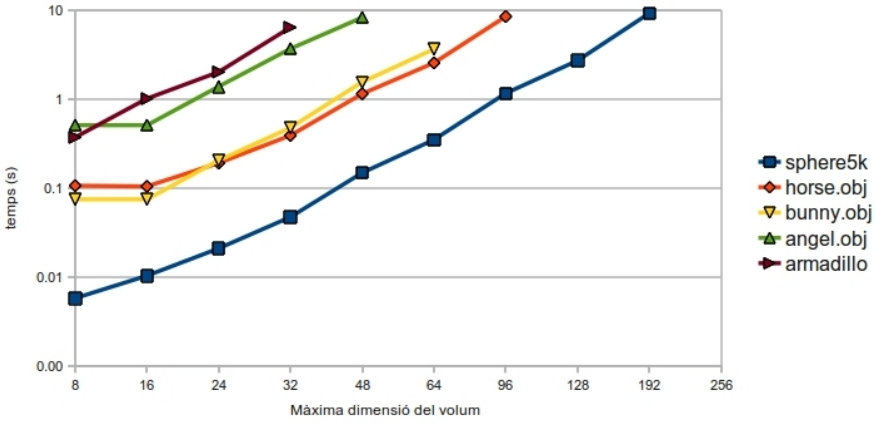


Figura 28: Proves amb temps en funció de la figura i el volum

### 10.2.3 *Coalescing* en *compute capability 1.1*

A diferència dels *devices* amb *compute capability* $<= 1.2$, les *compute capability* 1.1, per a obtenir les dades de memòria global eficientment, necessiten accedir a posicions de memòria consecutives. La mida del volum no pot ser molt gran ja que la GPU utilitzada per a aquesta prova és una 8600M GS i només disposa de 2 MP. La taula 6 mostra els resultats de temps, el model de l'esfera utilitzat té 80 cares.

**Observacions:** En el model de molt poques cares sphere, el coalescing ha afavorit el resultat final. En canvi al utilitzar un model de moltes cares, el mecanisme de *latency hiding* (veure secció 3.3) ha planificat els *warps* de manera que mentre uns estaven esperant resultats de memòria, planificava *warps* que ja tenien les dades preparades.

| model | coalescing | sense coalescing |
|---|---|---|
| sphere 64 | 100 | 161 |
| horse 32 | 5716 | 5717 |

Taula 6: Proves de *coalescing* en *compute capability* 1.1

### 10.2.4 Efecte memoització

L'objectiu d'aquesta prova (veure figura 29) és veure com ha afectat al rendiment de l'aplicació treure del bucle més intern una operació que conté dues divisions, dos mòduls, un producte d'enters, i una conversió a float.

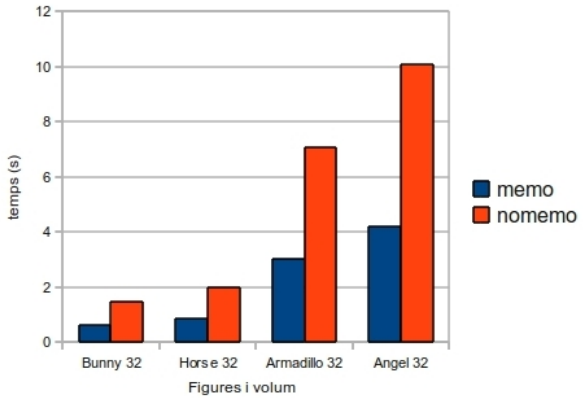


Figura 29: Gràfic comparatiu entre aplicar memoització o no

**Observacions** Com era d'esperar, doblar el nombre d'instruccions en el bucle més internet afecta negativament al rendiment del programa.

### 10.2.5 Temps d'execució i error introduït per les funcions de distància

S'han implementat dues funcions de distància una més costosa que l'altra, cal veure quin és el percentatge d'error d' una en front l'altre i valorar si pel cost en temps val la pena utilitzar la funció amb més precisió. Per a aquesta prova s'han utilitzat els models horse i bunny i s'han provat amb volums de 32 i 64 punts en la dimensió màxima. A la taula 7 es poden observar els valors del càlcul d'error i el gràfic 30 mostra els temps. Disteff, representa la funció de distància aproximada; distprec, la funció sense error. L'error està mostrat en diferència de distància del valor esperat al recollit.

**Observacions** S'ha vist que utilitzar la funció de distància òptima, no introdueix un error excessiu. En canvi el temps d'execució es redueix entre un 30% i un 50% així, es decideix deixar la implementació amb la funció eficient.

| model | dimensió | max err | min err | mig err | err % |
|---|---|---|---|---|---|
| bunny | 32 | 0.02 | $\approx 0$ | 0.002 | +2% |
| bunny | 64 | 0.05 | $\approx 0$ | 0.004 | +2% |
| horse | 32 | 0.03 | $\approx 0$ | 0.01 | +3% |
| horse | 64 | 0.03 | $\approx 0$ | 0.0014 | +3% |

Taula 7: Error introduït per la funció de distància.

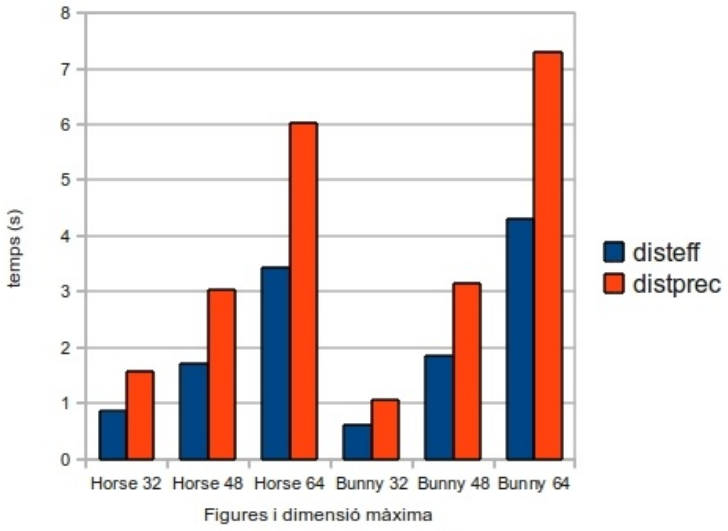


Figura 30: Gràfic diferència de temps funció de distància

## 10.3 Seqüencial vs *tiling*

Amb aquesta prova es comparará el rendiment de dues implementacions en CUDA diferents. L' escala de l'eix y es logarítmica.

**Observacions:** Al contrari del que es pensava, l'algoritme seqüencial ha obtingut millors resultats que el de tiling. A l'anàlisi de resultats s'intenta explicar quines són les causes.

## 10.4 Força bruta: CPU vs CUDA

En realitat una implementació i l'altre no s'haurien de comparar, ja que la versió en CPU no està gens optimitzada, però pot servir per a veure com un problema de cost computacional $N^3 * M$ pot ser transportat a CUDA per a obtenir millores espectaculars. A la figura 8 es veuen els resultats d'execució en cpu per al model horse amb diverses mides de volum.

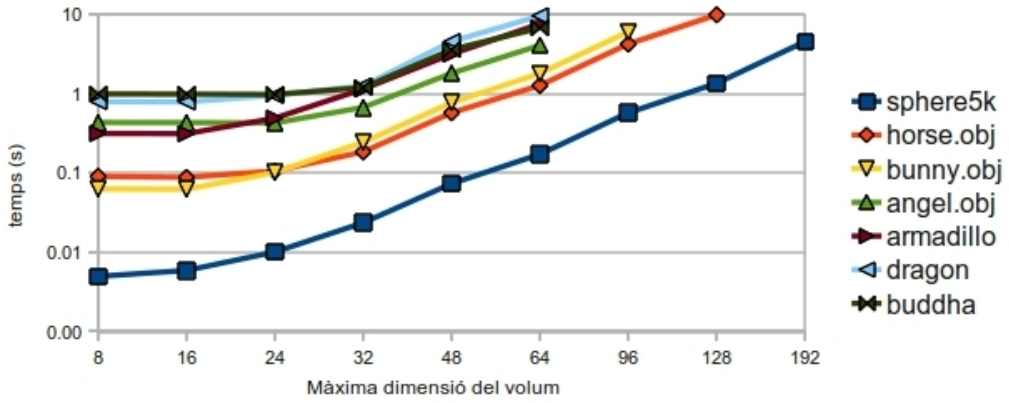


Figura 31: Gràfic temps d'execució amb recorregut seqüencial per cares.

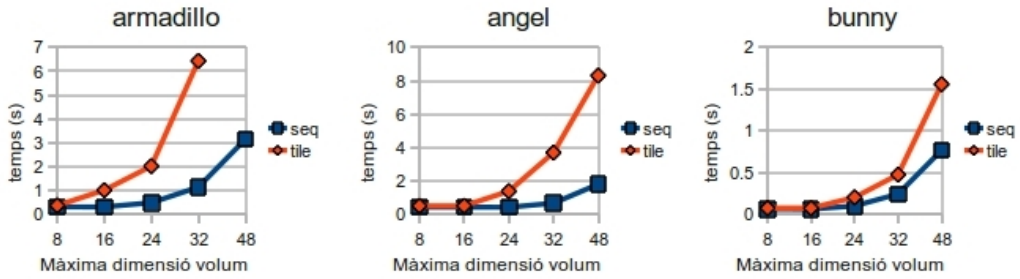


Figura 32: Gràfics comparatius del temps d'execució de la implementació en seqüencial i la de *tiling*

## 10.5 Bricking CUDA

### 10.5.1 Cost del recorregut en les cares

El cost de classificar les cares segons si estan inclosos en un *brick*, depèn del nombre de cares, del nombre de subdivisions. Es provarà amb models de moltes i poques cares. És d'esperar que el temps d'execució de l'algoritme sigui $S^3 * F * O$ on $S$ són el nombre de subdivisions, $F$ el nombre de cares i $O$ el valor de l'*overlapping*. L'overlapping però tindrà poc pes en el temps final ja que el costós de l'algoritme de classificació es el recorregut pels volums i les cares. Els gràfic es pot veure a la figura 33.

### 10.5.2 Precisió perduda utilitzant diferents factors de subdivisió i overlapping, temps de triatge i temps de *kernel*

Aquesta prova es fa per veure quin és el percentatge de mostres obtingudes segons la subdivisó i overlapping i quin és l'error que mostren. La mostra per calcular l'error són el conjunt de punts sobre els que s'ha obtingut la distància (dels que no es coneix, no s'utilitzen) i és la mitjana de la diferència entre el valor esperat i el calculat de tots ells. Els resultats esperats son una

| dimensió | CPU | CUDA | speed-up |
|---|---|---|---|
| 8 | 2.63 | 0.09 | 29.22x |
| 16 | 21.12 | 0.09 | 234x |
| 32 | 152.685 | 0.11 | 1388x |

Taula 8: Comparació en temps de CPU i CUDA

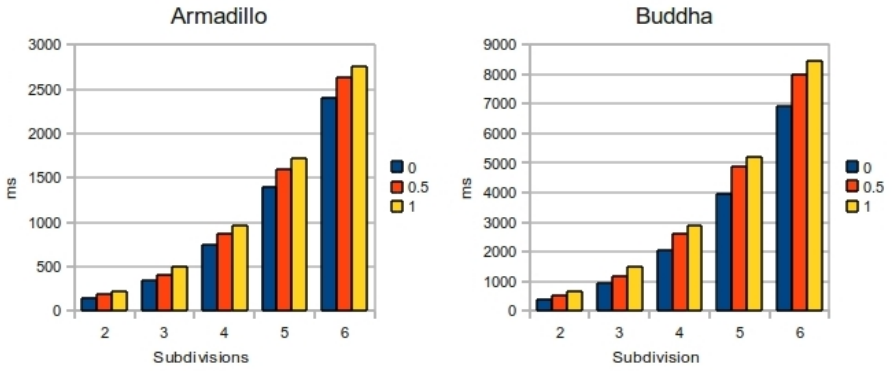


Figura 33: Cost de classificació de cares

disminució de l'error com més proper a *subdivisions* és *overlap*. Les proves de la taula 9 s'han executat amb el model horse i un volum de 64. Els temps estan indicats en milisegons.

| subdivisions | 1 | 2 | | 4 | | |
|---|---|---|---|---|---|---|
| overlapping | nobrick | 0 | 0.5 | 0 | 0.5 | 1 |
| %mostres | 100% | 79.34% | 79.34% | 38.31% | 66.85% | 71.31% |
| error mig | 0 | 0.71 | 0.24 | 1.4 | 0.73 | 0.25 |
| temps cares | 0 | 38.22 | 50.42 | 204.66 | 235.31 | 264.74 |
| temps kernel | 1282.51 | 205.51 | 915.49 | 157.78 | 1003.14 | 29965.33 |
| temps total | 1282.51 | 243.73 | 965.912 | 362.445 | 1238.45 | 30230.07 |

Taula 9: Errors i temps en bricking amb horse i volum de 64

La figura 34 mostra un model on a causa de les subdivisions, es perden valors, la part del pla colorejat que manca correspon als valors no calculats. Mentre que la figura 35 mostra els errors que apareixen als *bricks* on l'overlapping és baix i no es trien cares que estan properes però pertanyen a un altre *brick*. A la regió marcada en blau es pot veure una pèrdua de precisó respecte l'original.

**Observacions** Augmentar el factor d'*overlapping* afegeix una mica d'overhead en el triatge de les cares per el cost de copiar la cara a un array i com era d'esperar el cost es cúbic en funció del nombre de subdivisions. L'error disminueix a mida que l'*overlapping* augmenta, això és d'esperar ja que com més gran és l'*overlapping* més cares s'afegeixen al *brick* i més precisió es pot obtenir. També s'observa però que ambdós el temps d'execució i de *kernel* augmenten.

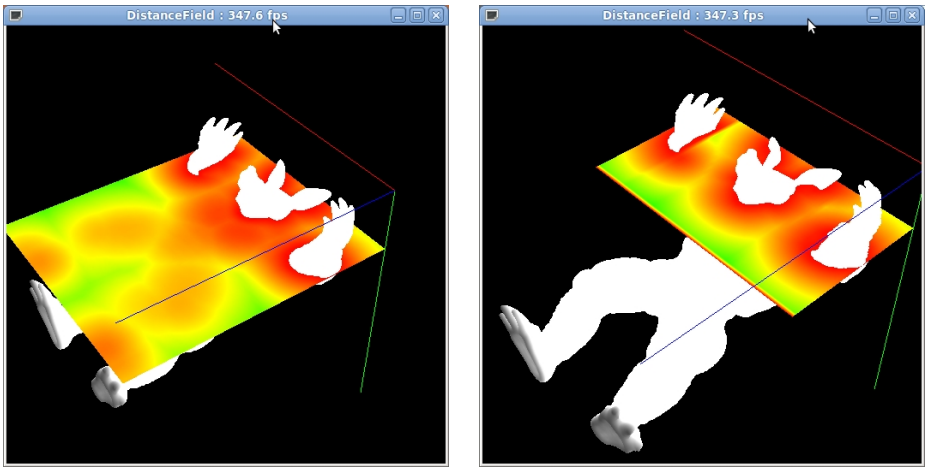


Figura 34: Pèrdues de dades en *bricking*

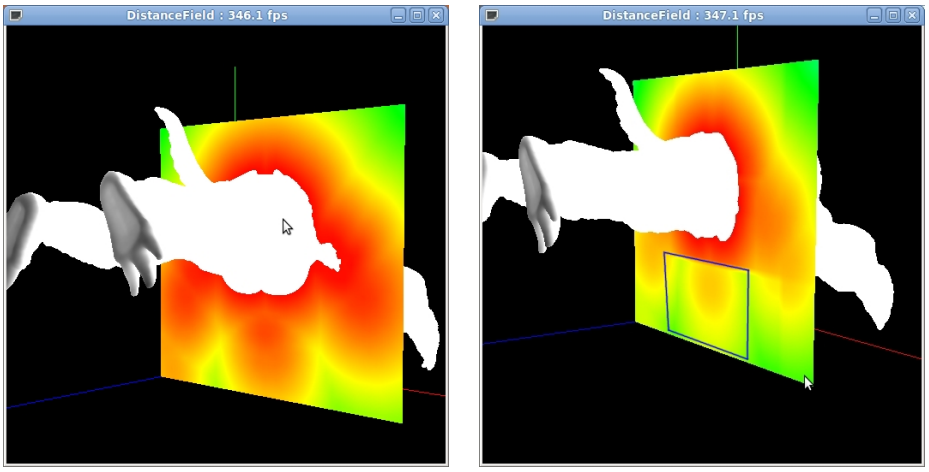


Figura 35: Pèrdues de precisió en *bricking*

El temps d'execució amb 2 i 4 subdivisions és el mateix per que no es descarten *bricks* en cap dels casos és d'esperar que el pas amb 4 subdivisions tingui més errors que el de 2 ja que tenen el mateix factor d'*overlapping* i els *bricks* en 4 subdivisions son més petits i trien menys cares.

# 11 Anàlisi dels resultats

## 11.1 DiFi vs la nostra implementació *bricking* CUDA

Els resultats de la implementació DiFi no es podran comparar gaire bé ja que, a part que el hardware que es va utilitzar a l'època està molt desfasat, no es disposen dels models que es van utilitzar a les proves. Per tal de comparar el temps, s'ha utilitzat el model horse utilitzant l'algoritme de *bricking* amb 6 subdivisions i un overlapping de 0.8. Les dades estan a la taula 10. Els models tenen un nombre de cares més o menys similar i per equiparar el nombre de mostres, s'ha augmentat la mida del volum. L'*speedup* obtingut és de quasi 5x. És necessari dir que segurament si les proves de DiFi fossin executades amb el hardware actual, s'obtindrien millors resultats.

| algoritme | model | cares | volum | temps |
|---|---|---|---|---|
| DiFi | Cassini | 93234 | 186x254x188 | 29.86 |
| *bricking* CUDA | horse | 96966 | 118x214x256 | 6.5 |

Taula 10: Comparació resultats *bricking* CUDA vs DiFi

## 11.2 Bærentzen & Aanæs vs la nostra implementació en CUDA

El rendiment de la implementació que s'ha fet en CUDA es pot comparar amb els resultats de la implementació de Bærentzen i Aanæs. En concret ells utilitzen un model d' un cavall de 60680 polígons. Les proves les han realitzat amb un volum de $128^3$. Veure la taula 3 per més detalls. La seva implementació del DF amb *axis aligned bounding box* triga 226.9 segons més 1.57 segons en la generació de la jerarquia de caixes per accelerar el procés. La meva implementació de força bruta amb recorregut de cares seqüencial sobre un model d'un cavall amb 96966 polígons amb un volum de 128, triga 9.84 segons. Això suposa un speedup de 23x respecte la seva implementació.

## 11.3 Diferència de temps d'execució entre seqüencial i *tiling*

S'ha dissenyat en primer lloc l'algoritme amb un recorregut per cares on es minimitzen el nombre d'accessos a memòria global i s'aprofita la memòria compartida al màxim per a evitar recalcular dades i per a que els *threads* puguin accedir a valors anteriorment calculats i, d'aquesta manera, reduir el nombre d'instruccions dins dels bucles. Secundàriament i per provar que el disseny efectuat era el correcte, s'ha implementat el *kernel* que ni fa ús de la memòria compartida ni minimitza de lluny el nombre d'accessos a memòria global.

Identificant les operacions en coma flotant de resta com a $r$, producte $p$, suma $s$ i accés a memòria d'un float $m$, es fa el següent anàlisi. Despreciant les instruccions externes als bucles, el *kernel* aplicant *tiling*, conté dos bucles aniudats. L'exterior és depenent del nombre de cares $F$ però al aplicar *tiling*, s'itera per ell $F/blockDim$ cops. La variable que controla el bucle interior és $blockDim$, per tant el cost en iteracions dels dos bucles per *thread* és $(F/blockDim) * blockDim$. El bucle interior s'encarrega d'obtenir la distància que costa, en operacions, $3*(4*r+3*p+2*s)$. El bucle exterior s'encarrega d'obtenir els vèrtex de memòria i transformar-los. Les operacions necessàries per a fer aquest procés són: $3 * 4 * (m + r + p)$. Com el *throughtput* de totes les operacions aritmètiques és el mateix les agrupem en $f$ (flop) i l'expressió que defineix el cost del

*kernel* queda com: $(F/blockDim) * (3 * 4 * (m + 2 * f) + blockDim * (3 * 9 * f))$. Simplificant i substituint blockDim per 128. L'expressió que defineix el cost queda finalment com:

$$(F/128) * (12 * m + 3480 * f) \tag{7}$$

En el cas del *kernel* amb recorregut seqüencial, només hi ha un bucle que itera per $F$ cares. I el cost d'accedir a memòria i transformar els vèrtexs i el del càlcul de la distància és el mateix. Per tant l'expressió del cost és $F * (3 * 4 * (m + 2 * f) + 3 * 9 * f)$ que simplificada queda

$$F * (12 * m + 51 * f) \tag{8}$$

Eliminant la variable $F$ i multiplicant 128 a l'expressió 8. L'expressió de l'esquerra correspon a la implementació amb *tiling* i la de la dreta a la *seqüencial*
$12 * m + 3480 * f << 1536 * m + 6528 * f$
$m + 290 * f << 128 * m + 544 * f$

Al fer el disseny de l'algoritme en CUDA es va passar per alt dos detalls. Un és el nombre de registres que utilitzen els *threads* (veure la secció 3.4.2). El compilador de CUDA, pot retornar el nombre de registres que utilitza cada *kernel* amb la opció `-ptxas-options=-v`. En el cas de l'algoritme amb recorregut per *tiles*, el nombre de registres usats és 27 i amb el recorregut seqüencial 21.
Contant que els *blocks* son de dimensió 128 el màxim nombre de *blocks* assignats a un MP serà $\lfloor maxblockPerRegister/registersPerBlock \rfloor$. En l'algoritme de *tiling* es poden assignar com a molt 4 *blocks* per MP 16384/128*27. En el cas del seqüencial, 5. Donat que la GTX disposa de 30 MP, si s'utilitzen 4 *cores* per MP, del total de la GPU se n'utilitzen 120. En el cas de l'algoritme seqüencial, l'ús de cores és de 150. Amb això es pot justificar un speedup del 25%.
La resta d'speedup que s'aconseguix s'ha de buscar en la gestió de memòria. Tot i que la memòria compartida permet accedir a les dades molt ràpidament si no hi ha conflicte, se'n fa un ús molt de lectura i escriptura en els bucles més interiors i per molt ràpida que sigui, no pot competir amb els registres de latència zero que utilitza l'algoritme en seqüencial.

L'altra implicació de la memòria en l'*speedup* és l'accés a memòria global. Quan els *threads* d'un mateix *warp* demanen la mateixa posició de memòria se'ls serveix en una sola transacció. I per últim, per evitar que els threads en el *warp* que ja han acabat els bucles escriguin a memòria global el contingut de la memòria compartida abans que els companys de *block* la hagin actualitzat, hi ha la instrucció `__syncthreads()` que bloqueja tots els *threads* del *block* fins que tots hi han arribat. Això implica que després del desbloqueig, 128 threads volen escriure al mateix temps a memòria. Per tant, es perd temps en l'espera i en la demanda en paral·lel d'escriptures a memòria.

# 12 Conclusions

Després d'estudiar i entendre el problema del càlcul del *distance field*, s'ha vist que presentava un paral·lelisme de dades considerable i s'ha fet una implementació sobre l'arquitectura CUDA. Per dur a terme la implementació del DF es va considerar que CUDA era una arquitectura sobre la que treballar ja que aquesta ofereix una capacitat de paral·lelisme superior a la d'una CPU. Un processador de gama alta actual, l'Intel Core i7 980 XE per exemple, pot assolir un pic teòric de rendiment de 107.55 GFlops en precisió simple. Mentre que els pics de la nVIDIA GTX 580 i la ATI Radeon HD 5970 són de 1581 i 4640 GFlops respectivament.

Una bona part del temps del projecte, s'ha dedicat a estudiar el funcionament de l'arquitectura CUDA per entendre quins són els factors més importants en l'execució. Adquirir aquest coneixement és bàsic per tal de poder utilitzar-los en la present implementació. Per a comprovar que les bones pràctiques proposades pels dissenyadors de CUDA s'havien aplicat correctament, s'ha implementat un versió que no seguís aquests patrons per a poder comparar-la i veure el canvi de rendiment. Al final s'ha vist que la versió que s'esperava menys eficient, ha resultat ser molt millor. La raó és que s'han volgut seguir les recomanacions però no s'han tingut en compte altres que afecten críticament al rendiment del programa.

S'ha implementat una millora per a reduir el temps d'execució implicant la CPU en el procés. Aquesta millora introdueix petits errors que segons l'ús que es vulgui fer del distance field no cal tenir-los en compte. Així combinant les capacitats de càlcul de propòsit general de la CPU i el poder de paral·lelisme de CUDA, es poden obtenir resultats més eficientment. La clau resideix en dedicar cada feina al processador que la pot fer millor.

Els resultats de les implementacions proposades s'han comparat amb altres existents i s'ha observat que en comparació amb els mètodes que fan ús de la CPU s'ha obtingut una millora de 23x. En el cas de les implementacions en GPU que apliquen optimitzacions de triatge de triangles, la millora és de 5x però si s'executés en el hardware actual, podríen obtenir millors resultats que en la implementació del projecte.

Finalment, s'ha observat que per a obtenir el màxim rendiment d'un programa en CUDA, un programador novell en la tecnologia, no només ha de pensar un disseny i implementar-lo seguint la guia de bones pràctiques, sinó també provar diversos dissenys i comparar-los ja que la documentació tècnica de CUDA és molt extensa i hi ha petits detalls que sempre es poden passar per alt. Trobar-se amb un resultat no esperat ha servit per preguntar-se perquè es produïa aquest resultat i investigar les raons per a poder trobar explicacions.

Ja ho deia Francis Bacon al segle XVI:
"El coneixement s'adquireix llegint la lletra petita d'un contracte; l'experiència, no llegint-la."

# 13 Treball futur

Completar la implementació amb el càlcul del signe. Per tal de millorar el rendiment es podríen fer varies optimitzacions. En l'algoritme de *bricking* s'ha vist a les proves que en alguns casos triga més temps el triatge de cares que el càlcul del DF. En la implementació present, abans de llençar els *kernels*, es fa el precàlcul de tots els *bricks*. Es podria provar d'aprofitar les capacitats d'enviament asíncron de la GPU per a calcular les cares del *brick i* a la CPU mentre la GPU està executant el *grid i-1*. D'aquesta manera, per exemple en el cas de la taula 9 es podria aconseguir un speedup de 2x, mentre que en altres casos no s'aconseguiria ni l'1 % de millora.

Donat que part del paral·lelisme que es pot aconseguir ve limitat pel nombre de registres usats en la meva implementació, cal reduïr l'us de registres. Per a això, s'han d'eliminar variables que s'utilitzen per a iterar o com a índexos. Els valors que emmagatzemaven, es continuen necessitant pel que el valor de les variables haurà de ser calculat cada cop que es necessiti. Això és una alternativa que s'ha de provar que haver de recalcular, no afecti negativament al temps d'execució

Per mostrar la textura amb colors, s'ha de fer un procés de normalitzat per a obtenir els valors en un rang de 0 a 1. Aquest procés es fa a la CPU. Al final del l'execució, abans d'escriure els valors a la memòria global, es podria fer un procés de reducció i obtenir els valors mínims i màxims. I just abans d'escriure a memòria global, normalitzar el valor. S'hauria de provar si unir-ho al *kernel* de càlcul de la distància és millor opció que fer-ho per separat en un altre *kernel*.

Segons la combinació de valors d'*overlapping* i subdivisions s'utilitzi, es perden mostres del DF. Si per a una aplicació no és important l'error però es necessiten tots els valors, es pot augmentar el valor de l'overlapping en *runtime*. Per evitar fer moltes iteracions, es podria comprovar si els *bricks* veïns, si no tenen valor, vol dir que 1, és un factor d'*overlapping* molt baix. L'algoritme de subdivisions que s'ha programat és molt senzill i potser amb una jerarquia de caixes en la CPU es pot accelerar el procés.

Caldria fer un anàlisi detallat de l'algoritme i definir un model que descrigui el temps d'execució en funció del nombre de cares, la mida del volum i les característiques de la GPU on es vol executar. D'aquesta manera es podria obtenir el cost en temps en funció de la GPU i permetre decidir quina GPU utilitzar si el temps d'execució és crític per l'ús que es vol fer del DF.

Per últim, manca estudiar l'arquitectura Fermi i aprofitar les noves característiques. D'entrada, s'aconseguira l'*speedup* proporcional al nombre de MP, però si s'aprofiten les *caches* i s'apliquen altres optimitzacions, aquest *speedup* es pot multiplicar.

# 14 Planificació i cost del projecte

Un cop acabat el projecte, cal comprovar si la planificació inicial s'ha seguit amb èxit, o si al contrari, hi ha hagut una desviació de temps i veure com això ha afectat al projecte. La figura 36 mostra el diagrama de gantt amb la planificació. A l' etapa de recerca de problemes, hi va haver un desfasament per que es va trobar un problema i es va estudiar però al fer l'anàlisi, es va veure que no era viable implementar-lo en CUDA. Això va endarrerir uns dies la resta del projecte. Per algun error en la implementació, no s'ha acabat afegint la funcionalitat del càlcul del signe i es deixa com a treball futur.

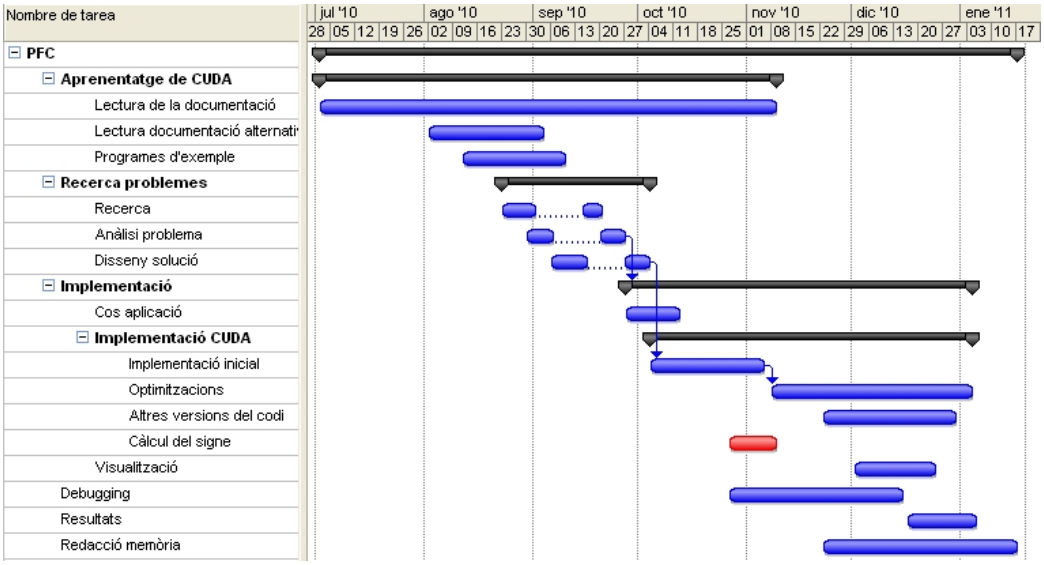


Figura 36: Diagrama gantt de la planificació del projecte

Considerant el projecte com a un producte que es pugui utilitzar per terceres persones, el cost del projecte es pot calcular com les hores invertides en desenvolupament mes el els recursos que s'han necessitat. Considerant que un enginyer informàtic desenvolupant aquest treball cobraria aproximadament uns 25 €per hora, havent realitzat un treball de 540 hores, el cost de programador resulta en 13.500 €. Els recursos que s'han realitzat per a desenvolupar el projecte han estat un ordinador portàtil per al desenvolupament i un sobretaula on el programa es pot executar. El sistema operatiu que s'ha utilitzat per a desenvolupar i fer les proves ha estat Ubuntu 10.04 i el software utilitzat durant el desenvolupament és gratuït. Contant que el programa que s'ha desenvolupat està pensat per a l'arquitectura GT200 i no s'aprofita cap característica concreta de Fermi, la targeta gràfica que pot donar millor rendiment és la nVIDIA Quadro FX 4800. La taula 14 mostra les despeses totals del projecte.

| Recurs | Cost |
|---|---|
| Treballador | 13.500 € |
| Portàtil | 1.300 € |
| PC sobretaula | 1400 € |
| 2 x Ubuntu 10.04 | 0 € |
| QT Creator | 0 € |
| Quadro FX 4800 | 1125 € |
| Total | 17325 |

Taula 11: Costos del projecte.

# Referències


[1] Tomas Akenine-moller, *Fast 3D triangle-box overlap testing*, October 24 2001.

[2] J. A. Bærentzen and H. Aanæs, *Computing discrete signed distance fields from triangle meshes*, Tech. report, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2002.

[3] David H. Eberly, *3d game engine design*, Morgan Kaufmann, pub-MORGAN-KAUFMANN:adr, 2001.

[4] Rob Farber, *Cuda, supercomputing for the masses*, Abril 2008, `http://www.drdobbs.com/high-performance-computing/207200659`.

[5] David B. Kirk and Wen mei W. Hwu, *Programming massively parallel processors: a hands-on approach*, Morgan Kaufmann Publishers, pub-MORGAN-KAUFMANN:adr, 2010.

[6] Taejung Park, Sung-Ho Lee, Jong-Hyun Kim, and Chang-Hun Kim, *CUDA-based signed distance field calculation for adaptive grids*, CIT, IEEE Computer Society, 2010, pp. 1202–1206.

[7] J. Andreas Bærentzen and Henrik Aanæs, *Signed distance computation using the angle weighted pseudonormal*, IEEE Trans. Vis. Comput. Graph **11** (2005), no. 3, 243–253.

[8] Ken Brodlie Rita Borgo, *State of the art report on gpu visualization*, April 2009, `http://www.viznet.ac.uk/reports/gpu/1`.

[9] A. Rosenfeld and J. Pfaltz, *Sequential operations in digital picture processing*, Journal of the ACM **13** (1966), no. 4, 471–494.

[10] Anand Lal Shimpi, *State of the art report on gpu visualization*, Febrer 2001, `http://www.anandtech.com/show/742/12`.

[11] Avneesh Sud, Miguel A. Otaduy, and Dinesh Manocha, *Difi: Fast 3D distance field computation using graphics hardware*, Comput. Graph. Forum **23** (2004), no. 3, 557–566.

[12] Grit Thürmer and Charles A. Wüthrich, *Computing vertex normals from polygonal facets*, Journal of Graphics Tools: JGT **3** (1998), no. 1, 43–46.

# A Manual d'usuari

## A.1 Inici per la línia de comandes

- –**file:** nom de l'arxiu *.obj que conté el model a analitzar. Ha d'estar a ./models
- –**maxvoldim:** nombre enter que indica la dimensió màxima del volum a analitzar
- –**mode:** algoritme a utilitzar per a calcular l'SDF. Admet els següents opcions `brick`, `eff`, `nomemoi`, `global`
- –**subdiv:** en el cass d'utilitzar l'algoritme de brick aquest paràmetre indica el nombre de subdivisions. Per defecte és 1, sense subdivisions.
- –**overlapping:** float, indica l'inflat de la caixa per a buscar cares dins del *brick*
- –**print:** imprimeix els valors del DF per la sortida estàndar
- –**nogl:** aquesta opció no obre un context OpenGL per a explorar l'objecte i l'SDF.

Les opcions que admet el paràmetre `-mode` son:

- **cpu:** Execució en mode CPU.
- **global:** Càrrega de dades de memòria global.
- **nomemoi:** No utilitza precàlcul dels punts.
- **eff:** Shared memory i memoització
- **coal:** Cares desades amb coalescing, shared memory i memoització. Només serveix amb *compute capability* 1.1.
- **pref:** Recorregut amb prefetching
- **brick:** Algoritme bricking.
- **seq:** Recorregut seqüencial per les cares.
- **seqbrick:** Recorregut seqüencial per les cares amb *bricking*.

Un exemple d'execució del programa seria: `./distanceField -file=horse.obj -maxvoldim=32 -subdiv=2 -overlapping=0.3 -mode=brick`

## A.2 Ús de la interfície

La interacció dins de l'interfície es amb l'ajut del mouse i el teclat. Els paràmetres de la càmera es modifiquen amb el mouse.

- **Rotació:** Clic esquerre + arrossegar.
- **Zoom:** Clic dret + arrossegar.
- **Pan:** Clic central + arrossegar.

La resta d'interacció es fa amb el teclat.

- **+:** Incrementa la posició del pla en l'eix actual.

- **-:** Incrementa la posició del pla en l'eix actual.
- **x:** Mostra el pla x.
- **y:** Mostra el pla y.
- **z:** Mostra el pla z.
- **o:** Mostra/no mostra l'objecte.
- **w:** Mode wireframe.

# B Kernels de CUDA

## B.1 Kernel tiling

El següent codi correspon al *kernel* amb recorregut per cares amb *tiling*.

```
__global__ void ultimateKernel(float4* gpu_faces, float4* gpu_pseudo_normals,
int3* gpu_pseudo_normals_index, sdfParams params,  float* gpu_distance_field)
{
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    //dynamic allocation of shared memory
    float* shared_dist = (float*)shared;
    float4* points = (float4*)&shared_dist[blockDim.x];
    int* shared_face = (int*)&points[blockDim.x];

    //initialize the point coordinates to avoid calculation on each thread iteration
    points[tid] = get3DPoint(idx+c_params.grid_offset, make_int3(c_params.volume_size.x,
     c_params.volume_size.y, c_params.volume_size.z));

    //initialize return arrays to a very big value
    shared_dist[tid] = 999999.0;

    float4 scale = (c_params.volume_size - make_float4(1, 1, 1, 0)) * c_params.scale;

    float4 p0, p1,p2;
    //load face tiles grouped by blockdim until done
    int faceIndex = tid * 3;
    for (int f = 0; f < c_params.nFaces / blockDim.x; f++, faceIndex += blockDim.x*3,steps++)
    {
        //scale moves vertices to volume space
        p0 = (gpu_faces[faceIndex] - c_params.minb) * scale;
        p1 = (gpu_faces[faceIndex + 1] - c_params.minb) * scale;
        p2 = (gpu_faces[faceIndex + 2] - c_params.minb) * scale;
        int index = tid; // tid = [0..blockDim)
        for (int i = 0; i < blockDim.x; i++)
        {
            float dist = getDistanceEff(p0, p1, p2, points[index]);
            if (dist < shared_dist[index])
            {
                shared_dist[index] = dist;
                shared_face[index] = f*blockDim.x + threadIdx.x;
            }
            if (++index == blockDim.x)
                index = 0;
        }
    }

    if (faceIndex < c_params.nFaces*3)
```

```
    {
        p0 = (gpu_faces[faceIndex] - c_params.minb) * scale;
        p1 = (gpu_faces[faceIndex + 1] - c_params.minb) * scale;
        p2 = (gpu_faces[faceIndex + 2] - c_params.minb) * scale;
    }
    else
    { //if faceIndex out of bounds, dummy initialization
        p0 = make_float4(0.0,0.0,0.0,0.0);
        p1 = make_float4(0.0,0.0,0.0,0.0);
        p2 = make_float4(0.0,0.0,0.0,0.0);
    }
    int index = tid;
    for (int i = 0; i < blockDim.x; i++)
    {
        if (faceIndex < c_params.nFaces * 3)
        {
            float dist = getDistanceEff(p0, p1, p2, points[index]);
            //float dist = bookDistance(p0, p1, p2, points[index]);
            //shared memory version
            if (dist < shared_dist[index])
            {
                shared_dist[index] = dist;
                shared_face[index] = faceIndex/3 + threadIdx.x;
            }
        }
        if (++index==blockDim.x){
            index=0;
        }
    }
    //esperar per que potser alguns warp del bloc no han actualitzat la posicio a la shared
    __syncthreads();
    if (idx+c_params.grid_offset < c_params.volume_size.x * c_params.volume_size.y *
     c_params.volume_size.z)
    {
//sign calculation doesn't work
        /*if (c_params.sign){
            gpu_distance_field[idx+c_params.grid_offset] =
            sqrt(shared_dist[tid])*getSign(points[tid],shared_face[tid],
            gpu_faces,c_params.minb,scale, gpu_pseudo_normals, gpu_pseudo_normals_index);
        }
        else {
            gpu_distance_field[idx+c_params.grid_offset] = sqrt(shared_dist[tid]);
        }*/
        gpu_distance_field[idx+c_params.grid_offset] = sqrt(shared_dist[tid]);
    }
}
```

### B.2 Kernel seqüencial

Codi del *kernel* amb recorregut seqüencial per cares.

```
__global__ void ultimateKernelNo(float4* gpu_faces, float4* gpu_pseudo_normals,
int3* gpu_pseudo_normals_index, sdfParams params,  float* gpu_distance_field)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    float4 point  = get3DPoint(idx+c_params.grid_offset,
        make_int3(c_params.volume_size.x, c_params.volume_size.y, c_params.volume_size.z));

    float4 scale = (c_params.volume_size - make_float4(1, 1, 1, 0)) * c_params.scale;

    float4 p0, p1,p2;
    float min_dist =999999.0;
    int face=0;
    float dist;
    int f;
    for (f = 0; f < c_params.nFaces; f++ )
    {
        p0 = (gpu_faces[f*3] - c_params.minb) * scale;
        p1 = (gpu_faces[f*3+ 1] - c_params.minb) * scale;
        p2 = (gpu_faces[f*3+ 2] - c_params.minb) * scale;

        dist = getDistanceEff(p0, p1, p2, point);
        if (dist < min_dist)
        {
            min_dist = dist;
            face = f;
        }
    }

    if (idx+c_params.grid_offset < c_params.volume_size.x * c_params.volume_size.y *
     c_params.volume_size.z)
    {
        gpu_distance_field[idx+c_params.grid_offset] = sqrt(min_dist);
    }
}
```

# C Funcions de CUDA

## C.1 get3DPoint

```
__device__ float4 get3DPoint(int idx, int3 size)
{
    int wh = size.x*size.y;
    int idmwh = idx%wh;
    return make_float4(float(idmwh%x), float(idmwh/x), float(idx/wh),0);
}
```

## C.2 getDistanceSimple

Codi de la funció de distància amb pèrdues.

```
__device__ inline float getDistanceEff(float4 p0, float4 p1, float4 p2, float4 p3)
{
    float4 a = p3-p2;
    float4 b = p3-p1;
    float4 c = p3-p0;
    return min(sqrtlength(a), min(sqrtlength(b), sqrtlength(c)));
}
```

## C.3 getDistance

Codi de la funció de distància exacta. Veure la quantitat d'operacions que necessita i també el nombre de camins alternatius.

```
__device__ __host__ __inline__ float bookDistance(float4 p0,float4 p1,float4 p2,float4 p3)
{
    p0.w = 0;
    p1.w = 0;
    p2.w = 0;
    p3.w = 0;
    float4 e0 = p1-p0;
    float4 e1 = p2-p0;
    float4 e2 = p3-p0;
    float a = dot(e0,e0);
    float b = dot(e0,e1);
    float c = dot(e1,e1);
    float d = dot(e0,p0-p3);
    float e = dot(-e1,p0-p3);
    float fC = dot(e2,e2);
    float det = a*c-b*b;
    float s = b*e-c*d;
    float t = b*d-a*e;
    float fSqrDist;

    if ( s + t <= det ) {
        if ( s < 0.0 ) {
            if ( t < 0.0 ) { // region 4
                if ( d < 0.0 ) {
                    t = 0.0;
                    if ( -d >= a ) {
                        s = 1.0;
                        fSqrDist = a+2.0*d+fC;
                    }
                    else {
                        s = -d/a;
                        fSqrDist = d*s+fC;
                    }
                }
                else {
                    s = 0.0;
                    if ( e >= 0.0 ) {
```

```
                t = 0.0;
                fSqrDist = fC;
            }
            else if ( -e >= c ) {
                t = 1.0;
                fSqrDist = c+2.0*e+fC;
            }
            else {
                t = -e/c;
                fSqrDist = e*t+fC;
            }
        }
    }
    else { // region 3
        s = 0.0;
        if ( e >= 0.0 ) {
            t = 0.0;
            fSqrDist = fC;
        }
        else if ( -e >= c ) {
            t = 1;
            fSqrDist = c+2.0*e+fC;
        }
        else {
            t = -e/c;
            fSqrDist = e*t+fC;
        }
    }
}
else if ( t < 0.0 ) {  // region 5
    t = 0.0;
    if ( d >= 0.0 ) {
        s = 0.0;
        fSqrDist = fC;
    }
    else if ( -d >= a ) {
        s = 1.0;
        fSqrDist = a+2.0*d+fC;
    }
    else {
        s = -d/a;
        fSqrDist = d*s+fC;
    }
}
else {  // region 0
    // minimum at interior point
    float fInvDet = 1.0/det;
    s *= fInvDet;
    t *= fInvDet;
    fSqrDist = s*(a*s+b*t+2.0*d) +
        t*(b*s+c*t+2.0*e)+fC;
}
}
```

```
else {
    float fTmp0, fTmp1, fNumer, fDenom;
    if ( s < 0.0 ) { // region 2
        fTmp0 = b + d;
        fTmp1 = c + e;
        if ( fTmp1 > fTmp0 ) {
            fNumer = fTmp1 - fTmp0;
            fDenom = a-2.0*b+c;
            if ( fNumer >= fDenom ) {
                s = 1.0;
                t = 0.0;
                fSqrDist = a+2.0*d+fC;
            }
            else {
                s = fNumer/fDenom;
                t = 1.0 - s;
                fSqrDist = s*(a*s+b*t+2.0*d) +
                    t*(b*s+c*t+2.0*e)+fC;
            }
        }
        else {
            s = 0.0;
            if ( fTmp1 <= 0.0 ) {
                t = 1.0;
                fSqrDist = c+2.0*e+fC;
            }
            else if ( e >= 0.0 ) {
                t = 0.0;
                fSqrDist = fC;
            }
            else {
                t = -e/c;
                fSqrDist = e*t+fC;
            }
        }
    }
    else if ( t < 0.0 ) { // region 6
        fTmp0 = b + e;
        fTmp1 = a + d;
        if ( fTmp1 > fTmp0 ) {
            fNumer = fTmp1 - fTmp0;
            fDenom = a-2.0*b+c;
            if ( fNumer >= fDenom ) {
                t = 1.0;
                s = 0.0;
                fSqrDist = c+2.0*e+fC;
            }
            else {
                t = fNumer/fDenom;
                s = 1.0 - t;
                fSqrDist = s*(a*s+b*t+2.0*d) +
                    t*(b*s+c*t+2.0*e)+fC;
            }
```

```
            }
            else {
                t = 0.0;
                if ( fTmp1 <= 0.0 ) {
                    s = 1.0;
                    fSqrDist = a+2.0*d+fC;
                }
                else if ( d >= 0.0 ) {
                    s = 0.0;
                    fSqrDist = fC;
                }
                else {
                    s = -d/a;
                    fSqrDist = d*s+fC;
                }
            }
        }
        else {  // region 1
            fNumer = c + e - b - d;
            if ( fNumer <= 0.0 )
            {
                s = 0.0;
                t = 1.0;
                fSqrDist = c+2.0*e+fC;
            }
            else {
                fDenom = a-2.0*b+c;
                if ( fNumer >= fDenom ) {
                    s = 1.0;
                    t = 0.0;
                    fSqrDist = a+2.0*d+fC;
                }
                else {
                    s = fNumer/fDenom;
                    t = 1.0 - s;
                    fSqrDist = s*(a*s+b*t+2.0*d) + t*(b*s+c*t+2.0*e)+fC;
                }
            }
        }
    }
    return fSqrDist;
}
```

# D Shaders

```
uniform int axis;
uniform float axis_value;
varying vec3 dist_color;
void main()
{
    gl_Position    = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_FrontColor  = gl_Color;
```

```
    gl_TexCoord[0] = gl_MultiTexCoord0;
    dist_color = vec3(gl_Vertex.x, gl_Vertex.y, gl_Vertex.z);
};

varying vec3 dist_color;
uniform int axis;
uniform float axis_value;
uniform sampler3D distance_field;
vec3 obtainColor(float dist){
    if (dist < 0.25){
        return vec3(1.0, dist*4.0, 0.0);
    }
    else if (dist < 0.5) {
        return vec3(1.0-(dist-0.25)*4.0, 1.0, 0.0);
    }
    else if (dist < 0.75) {
        return vec3(0.0, 1.0, (dist-0.5)*4.0);
    }
    else{
        return vec3(0.0, 1.0-(dist-0.75)*4.0, 1.0);
    }
}
void main()
{
    float dist = texture3D(distance_field, dist_color).r;
    if (dist==0.0) {
        discard;
    }
    gl_FragColor = vec4(obtainColor(dist), 0.0);
};
```