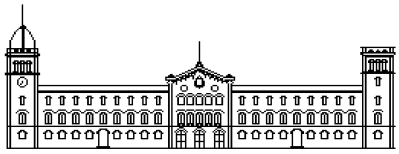

Treball fi de carrera

ENGINYERIA TÈCNICA EN INFORMÀTICA DE SISTEMES

Facultat de Matemàtiques
Universitat de Barcelona

IMPLEMENTACIÓ DE L'OPERADOR SURF EN OPENCL

Andrés Pardo Ginés

Co-Directors: Sergio Escalera
Manel Martínez Torres
Realitzat a: Departament de Matemàtica Aplicada i Anàlisi. UB

Barcelona, 17 de Setembre de 2010

Agraïments

Els primers agraïments són pels meus codirectors de projecte: Sergio Escalera i Manel Martínez. El primer sobretot per mantenir el contacte setmanal i resoldre els meus dubtes sobre la part de visió per computador. Al segon per la seva implicació en la implementació i tota la part d'OpenCL. Moltes gràcies als dos, especialment per la seva paciència i comprensió quan han aparegut problemes de tota mena que anaven endarrerint la planificació original.

També agraeixo molt la paciència i el suport mostrat per la meva família, xicota i amics de dins i fora de la facultat durant la realització d'aquest projecte que ha arribat a ser molt frustrant de vegades. Moltes gràcies a tots.

Resum

Creació d'una implemtanció OpenCL de l'algorisme de visió per computador SURF amb l'objectiu de poder fer-se servir per arquitectures AMD i Nvidia. Després de diferents intents d'implementació col·locant el codi de la versió OpenCV com a base va optar-se per crear una versió C++ que va servir com a base de la versió OpenCL final. Dissortadament els resultats per AMD no són correctes, però si ho són i a més a més satisfactoris per Nvidia.

Resumen

Creación de una implementación OpenCL del algoritmo de visión por computador SURF con el objetivo de ser usable tanto para arquitecturas AMD como Nvidia. Después de diferentes intentos de implementación poniendo el código de la versión OpenCV como base se optó por crear una versión C++ que sirvió como base de la versión OpenCL final. Desafortunadament los resultados no son correctos para las GPUs de AMD, pero son satisfactorios para las de Nvidia.

Abstract

The creatión of an implementation of the SURF operator in OpenCL aiming for a compatibility of Nvidia and AMD current products. After some failed attemps of implementations based on the OpenCV one, a OpenCL implementation based on a intermedium C++ layer worked. Unfortunately the results for the AMD products were incorrect, gratefully the Nvidia ones were good.

Índex

1. Introducció

En aquest capítol s'introdueixen tant el problema a resoldre com l'algoritme i la tecnologia utilitzades per aconseguir-ho. Primer, s'introdueix el concepte de reconeixement de punts d'interès tant important a la Visió artificial actual. Després es parla de l'algoritme Speeded Up Robust Features escollit per a ser implementat en la nova tecnologia, OpenCL, que és l'últim tema d'aquest capítol.

1.1 Introducció al reconeixement de punts d'interès

El reconeixement de punts d'interès representa el primer pas en qualsevol algoritme de detecció i descripció. Al ésser el pas inicial la informació obtinguda serveix com a base per a tot el procés i per tant volem que sigui lo més acurada possible. Per tant, és interessant que aquests punts siguin lo més impermeables possibles a transformacions afins, soroll i demés deformadors de la informació. Filtrar o depurar els punts d'interès trobats serveix per reduir la quantitat de falsos positius. Un cop localitzats i triats els punts es poden fer servir com a tals o trobar regions o cantonades.

Es fan servir moltes tècniques per a la detecció de punts d'interès: *Harris-Laplace, Hessian-Laplace, Laplacian of Gaussians, Difference of Gaussians*, etc. Com es veurà més endavant, en l'algoritme SURF que s'ha implementat en aquest treball s'utilitza una aproximació de la matriu hessiana anomenada "*Fast-Hessian*" com a eina per detectar els punts d'interès.

Aplicacions pràctiques del reconeixement de punts d'interès. Exemples d'aplicacions de diferents algoritmes, no només de SURF.

Reconeixement d'objectes i qualsevol aplicació que requereixi reconèixer una imatge i comparar-la amb altres.

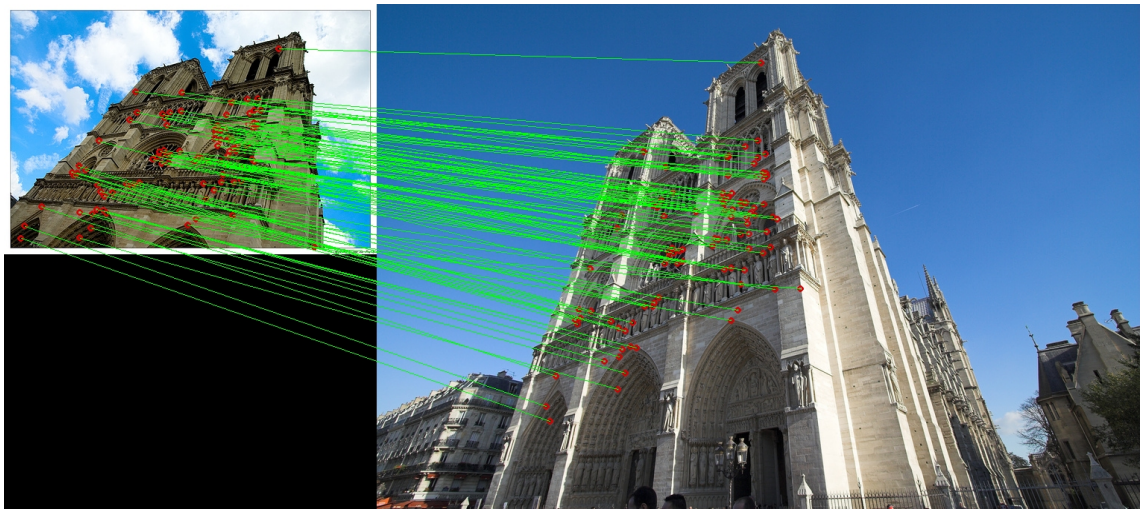

Figura 1: Mostra de la impermeabilitat i robustesa de SIFT davant de transformacions afins

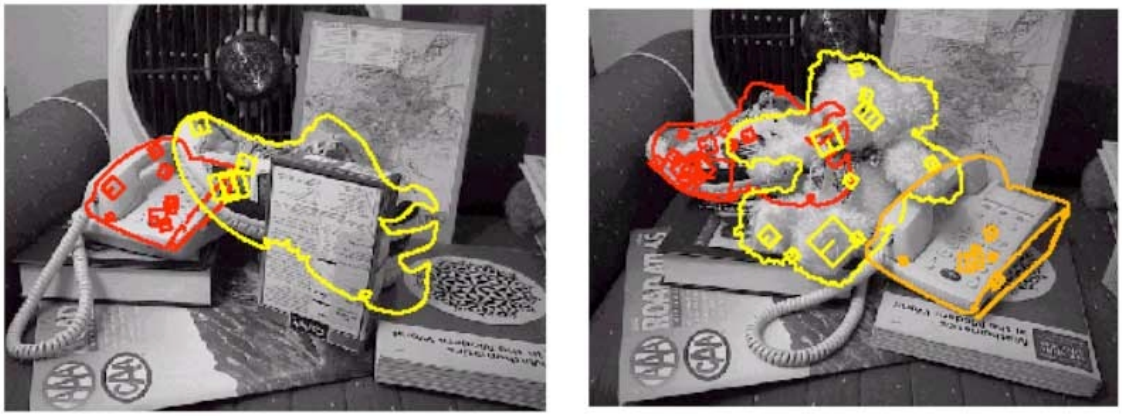

Figura 2: Mostra el reconeixement d'objectes d'una base d'aprenentatge en un ambient d'oclusió parcial. SIFT

Automatitzar del procés d'*stitching* aconseguint una imatge panoràmica o d'alta resolució combinant fotografies que presenten solapament parcial.

Figura 3: S*titching* usant SURF+RANSAC

Altres aplicacions: reconeixement de gestos per tele-medicina, millora de les interfícies home màquina, traducció de la llengua de signes, etc.

1.2 Introducció a SURF

Speeded Up Robust Features, d'ara endavant SURF, és un algoritme robust de detecció de punts d'interès i càlcul de descriptors d'imatges creat el 2006 per Herbert Bay, Tinne Tuytelaars i Luc Van Gool basant-se parcialment en l'algoritme Scale-invariant feature transform, SIFT d'ara endavant, creat per David Lowe a l'any 1999. En aquest treball s'ha estudiat la versió revisada de la publicació que va aparèixer durant 2008.

SIFT durant aquests anys es transforma en una solució molt popular, però cal accelerar-ne l'execució ja que aquesta mena d'algoritmes requereixen molt recursos i en aplicacions en temps real uns 30 fotogrames per segon per aconseguir els resultats desitjats. Diferents propostes ho han anat aconseguit, com Fast approximated SIFT, però cap sense empitjorar la qualitat final del descriptor. SURF proposa reduir el cost computacional, però mantenint la qualitat aconseguida per SIFT.

Això ho aconsegueix realitzant algunes aproximacions a càlculs que pertanyen a les tres parts d'aquest tipus d'algoritmes: la detecció dels punts d'interès, la creació d'un descriptor basat en ells i la comparació amb els d'altres imatges.

A més a més, en alguns dels camps d'aplicació de SURF, com ara la robòtica, l'espai físic disponible per la màquina que executarà el codi no permet la cabuda d'un ordinador estàndard. Forçant solucions basades en dispositius menys potents, però més portables.

Aquesta necessitat d'eficiència i portabilitat i el fet de que la implementació original fos de codi tancat a fet proliferar gran quantitat d'implementacions de codi obert amb diferents objectius. Hi ha versions per Java, Android o iOS basades en OpenSURF que és una de les primeres implementacions obertes de l'algorisme. N'hi ha d'altres centrades en obtenir una bona eficiència com les versions de Matlab o OpenCV. Aquesta cerca per l'eficiència màxima s'ha encaminat tant en el *multithreading* per CPUs com en l'ús de GPUs. Centrant-nos ja en aquestes últimes, podem trobar fins a 3 versions diferents de SURF en CUDA i altres solucions anteriors per gràfiques Nvidia, però aquesta és la primera realitzada en OpenCL, a priori compatible amb arquitectures d'AMD-ATI com d'Nvidia.

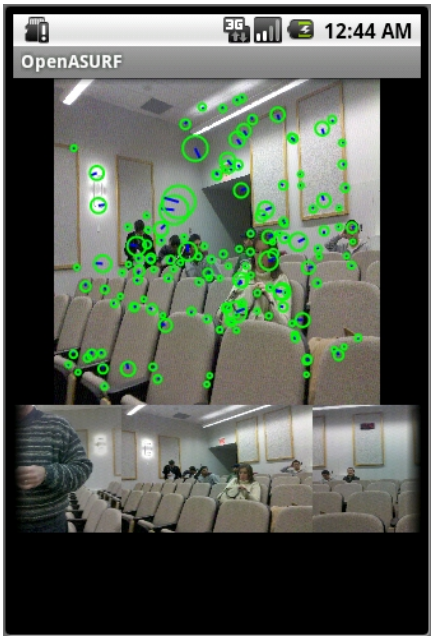

Figura 4: OpenSURF per Android

D'aquest treball per gràfiques Nvidia destaca GPU-boosted online image matching referència [1]. Un SURF GPU que utilitza OpenGL, les eines NVParse i el seu llenguatge Cg shading laguange per implementar un *matching* de dues imatges amb uns mil punts d'interès en 20ms. El seu treball i la manera d'apropar-se a l'*state of the art* del moment ha servit com a base per la feina realitzada en aquest projecte.

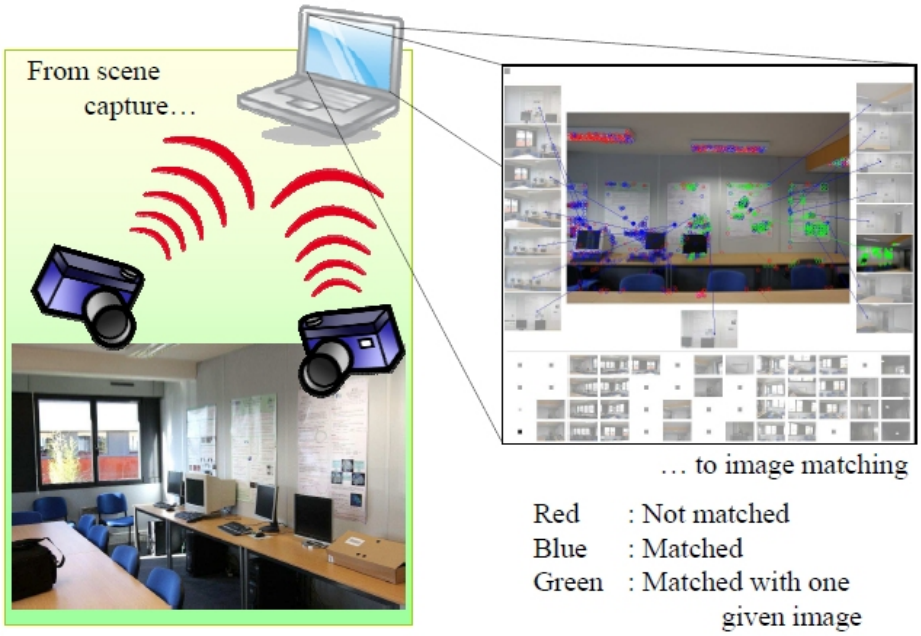

Figura 5: Sistema portàtil utilitzat pels creadors de GPU-boosted online image matching

1.3 Introducció a OpenCL

OpenCL és un *framework* d'estàndard obert per al desenvolupament i execució de codi d'interès general aprofitant el paral·lelisme que ofereixen actualment CPUs, GPUs i altres processadors. OpenCL consta tant d'una API per a definir i coordinar el paral·lelisme com d'un llenguatge propi basat en C99 per escriure els *kernels* que s'executen en aquests processadors.

OpenCL va ser desenvolupat en un estadi inicial per Apple, conserva els drets sobre la marca, però es presenta com una plataforma oberta, gratuïta, multi-plataforma i neutral. Tot això s'aconsegueix gràcies al Khronos Group. Khronos Group és una consorci sense ànim de lucre format per diferents empreses de la indústria que s'encarrega del desenvolupament tant d'OpenGL, com de COLLADA i ara també del d'OpenCL. Empreses com Intel, Arm, Samsung, AMD, Texas Instruments, Nvidia, Ericsson, IBM, Motorola, Nokia o Blizzard són membres d'aquest consorci.

OpenCL permet diferents maneres d'apropar-se al paral·lelisme, encara que no tots el fabricants de components ho fan actualment, i a part de GPU i CPU per separat permet altres modes com CPU+GPU o GPU+GPU. OpenCL permet totes aquestes possibilitats tant per dispositius portàtils com de sobretaula.

OpenCL pot ser cridat des de qualsevol llenguatge de programació que pugui interactuar amb l'*stub* de C. Python i C++ ja disponibles de forma pública.

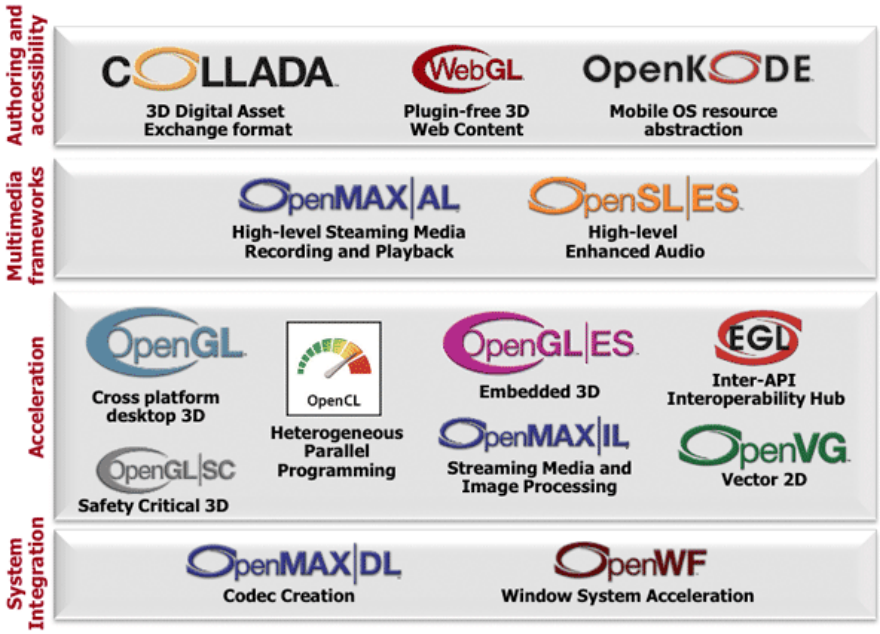

Figura 6: Tecnologies gestionades pel Khronos Group

2. OpenCL

En aquest apartat s'explica l'origen i desenvolupament d'OpenCL per després descriure l'estàndard i suport actual del mateix.

2.1 Antecedents

General-purpose computing on graphics processing units, GPGPU d'ara endavant, és un terme que es refereix a tractar càlculs computacionals no lligats a la infografia mitjançant la targeta gràfica o Graphic Processing Unit, GPU d'ara endavant, i no la CPU com tradicional s'havia fet. Aprofitar per altres computacions les arquitectures SIMD de les GPU actuals s'ha transformat en un punt important per fabricants i desenvolupadors de software.

Aquesta és una tendència cada cop més assentada i que ha anat accelerant-se paral·lelament a l'evolució en els últims anys de la API DirectX de Microsoft i de les arquitectures que la suporten. Arribant al punt actual on és molt comú que la GPU s'encarregui de la codificació/descodificació i post-processament d'àudio i vídeo en la majoria dels sistemes operatius actuals.

El desenvolupament de llibreries que aprofitant aquestes possibilitats va iniciar-se a l'any 2006 quan GPUs amb arquitectura de processadors de *shaders* unificats va proliferar amb l'arribada de la primera iteració de la serie GeForce 8 d'Nvidia, la G80. Aquesta arquitectura, que era la primera que suportava la versió 10 de DirectX, va servir com a base per a dos de les innovacions més importants del desenvolupament del GPGPU: Nvidia Tesla i CUDA.

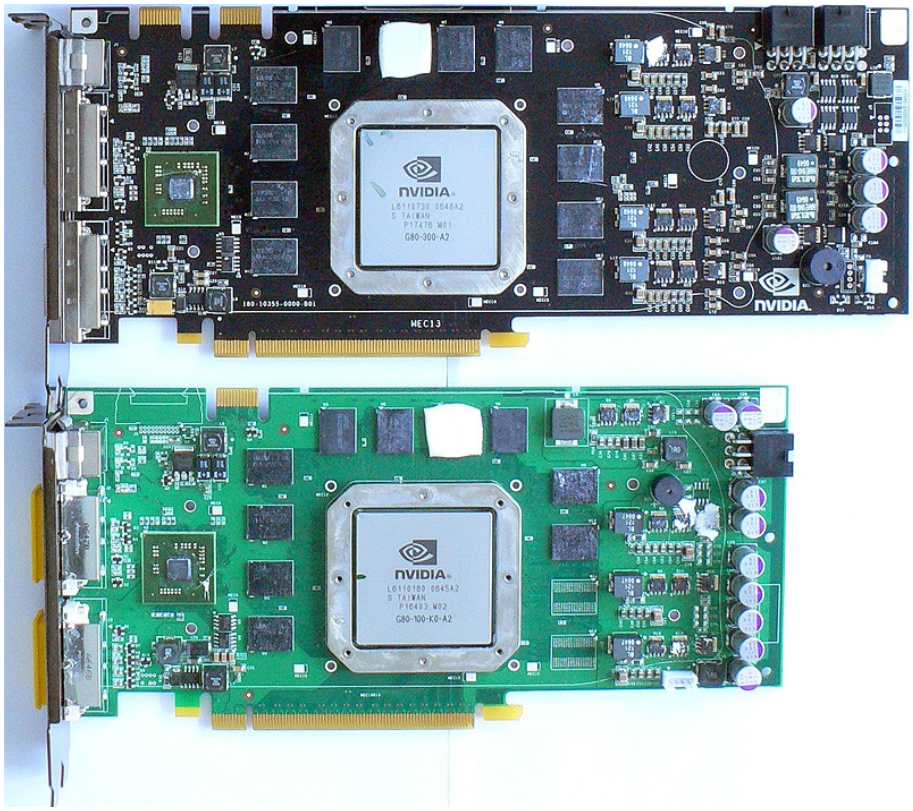

Figura 7: Circuit impresos de dos iteracions de l'arquitectura G80

Nvidia Tesla és la branca de GPUs d'Nvidia enfocada al GPGPU. Basades en les versions més potents de les diferents generacions d'arquitectures de GPUs i enfocades a les operacions en coma flotant per a càlculs científics o financers, no disposen ni de sortida de vídeo. La seva primera versió va aparèixer a finals de 2007 i estava basada efectivament en G80.

CUDA és la API d'alt i baix nivell que juntament amb el llenguatge C for CUDA permet aprofitar les capacitats GPGPU de les diferents arquitectures CUDA que va llançant Nvidia. El primer SDK va ser llançat el febrer 2007 i d'ençà llavors ha anat millorant i guanyant usuaris. Gràcies a la facilitat per portar el codi i la gran quantitat de *wrappers* per altres populars llenguatges de programació CUDA s'ha transformat en una eina molt usada en camps com la simulació d'efectes físics, la dinàmica de fluids i altres camps científics.

La resposta d'ATI a tot això, que acabava de ser comprada per AMD al 2006, va arribar quasi a l'hora que la proposta d'Nvidia. AMD FireStream o Ati FireStream, ha rebut ambdós noms, és exactament la mateixa mena de producte que Nvidia Tesla enfocat també al *high-performance computing*.

El projecte inicial de l'API que havia d'acompanyar FireStream s'anomenava Close to Metal, però després d'una reprogramació parcial amb el corresponent endarreriment va aparèixer durant el Desembre de 2007 com a Stream Computing SDK.

Per la seva part, Intel va anunciar Larrabee al SIGGRAPH 2008. Larrabee és un processador GPGPU a mig camí entre una CPU multicore x86 i una GPU SIMD moderna, però usant el joc d'instruccions x86, tenint poc hardware gràfic específic i uns nuclis molt senzills basats en la primera revisió dels Pentium original, P54C. Segons alguns *benchmarks* fets públics per la mateixa Intel durant 2008 escalant el número de nuclis s'obtenia un increment de rendiment quasi lineal. Dissortadament a finals de 2009, després d'uns quants endarreriments i rumors de tot tipus Intel va anunciar que Larrabee no arribaria al mercat com a GPU, però que següents iteracions de la tecnologia podrien fer-ho.

El 16 de Juny de 2008 el Khronos Compute Working Group va ser format per membres de la indústria per continuar amb la creació d'OpenCL i un estàndard clar. El 8 de Decembre es va fer pública l'especificació final d'OpenCL 1.0.

Amb el llançament de Mac OS X Snow Leopard a l'Agost de 2009 també es llança OpenCL 1.0. Poc després AMD decideix dona suport a OpenCL en el seu Stream SDK, a l'Octubre IBM llença la seva primera implementació d'OpenCL i al Desembre del mateix any Nvidia anuncia que donarà suport OpenCL en el seu GPU Computing Toollkit.

La més recent aportació al GPGPU és Microsoft DirectCompute. Disponible per GPUs AMD i Nvidia és una alternativa a OpenCL. DirectCompute va aparèixer a finals de 2009 com a part de la versió 11 de DirectX, encara que també és compatible amb arquitectures DirectX 10, i encara no ha rebut l'atenció ni l'ús de massa desenvolupadors.

2.2 OpenCL Models

En aquest apartat es defineixen les diferents capes conceptuals que formen l'API d'OpenCL.

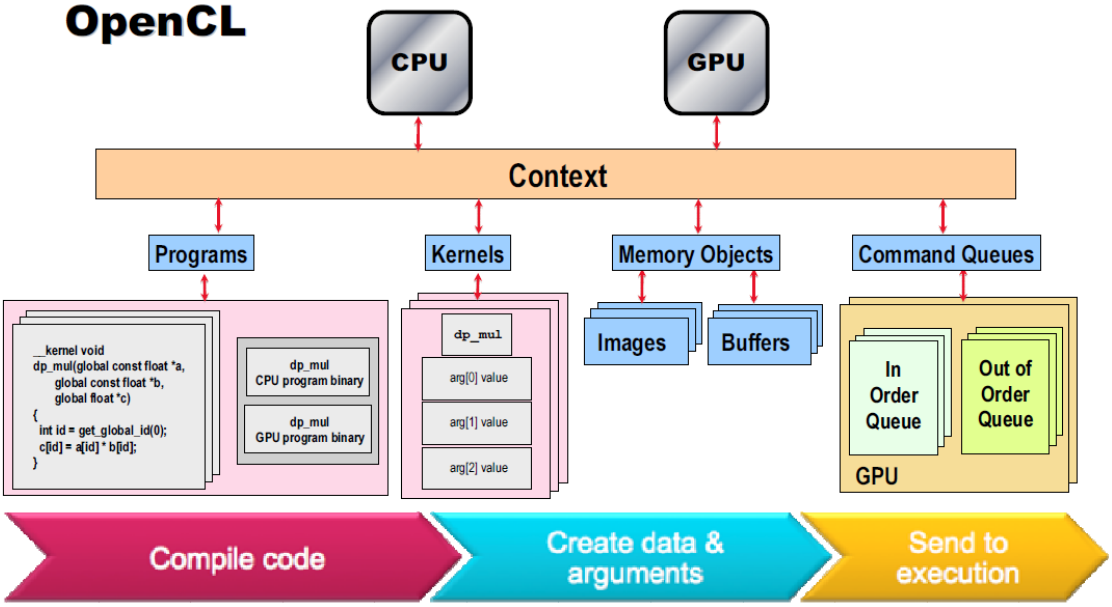

Figura 8: Esquema general d'OpenCL

2.2.1 Platform Model

Tenim un amfitrió o *host* que és el nostre ordinador connectat a diversos *compute devices*, processadors com ara CPUs o GPUs, i aquests *compute devices* estan dividits en un o més *compute unit*, un nucli d'un multiprocessador, i a l'hora aquests compute *devices* estan dividits en un o més *processing element*.

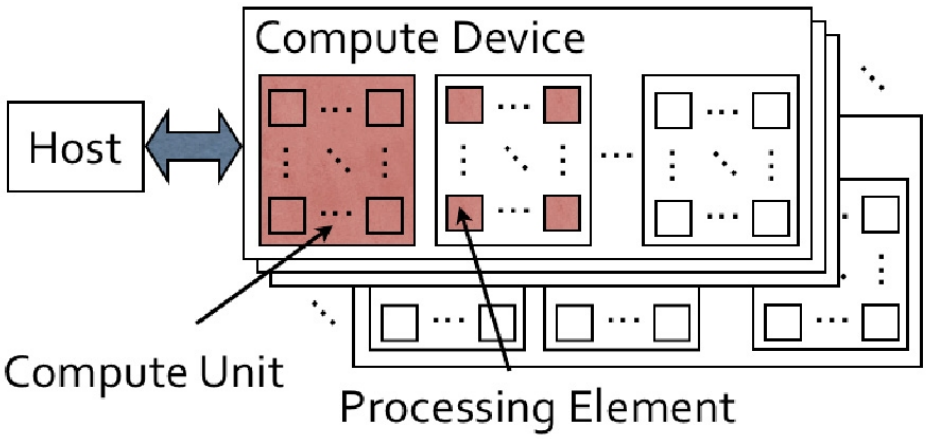

Figura 9: Estructura del Platform Model

2.2.2 Execution Model

El model d'execució consta de dues parts: els *kernels* que són les funcions que s'executen als compute devices i que fan possible el paral·lelisme i el *host program* que defineix i administra el context d'execució.

Al enviar un *kernel* a executar-se un espai d'índexs es crea i cada instància del *kernel* executant-se per cada punt de l'espai s'anomena *work-item*. Aquests ítems s'identifiquen per una ID global i sempre executant el mateix codi, encara que no tenen perquè fer-ho sobre les mateixes dades o en el mateix instant de temps ja que això pot escollir-se en funció de si cerquem un model de paral·lelisme de dades o de tasques. Per tant, cada instància del *kernel* que s'està executant a la GPU és un fil independent, això s'aconsegueix mitjançant control via hardware i degut a que els *processing elements* computen fils molt lleugers.

L'espai d'índexs s'anomena NDRange, on la N pot ser 1, 2 o 3, i ve definit per un vector d'enters tant llarg com calgui.

Per la seva part, els *work-items* s'organitzen *work-groups*. Aquests *work-groups* reben també una ID de la mateixa mida que la dimensionalitat del NDRange i de la Global ID d'un *work-item*, a més a més el fet de pertanyer a un grup dona als *work-items* una Local ID. Aquest fet propicia que un *work-item* pugui ser identificat mitjançant la seva Global ID o la seva Local ID més ID del seu grup.

Aquesta organització serveix per a que els *work-items* d'un *work-group* s'executin concurrentment en els *processing elements* d'un *compute unit* concret.

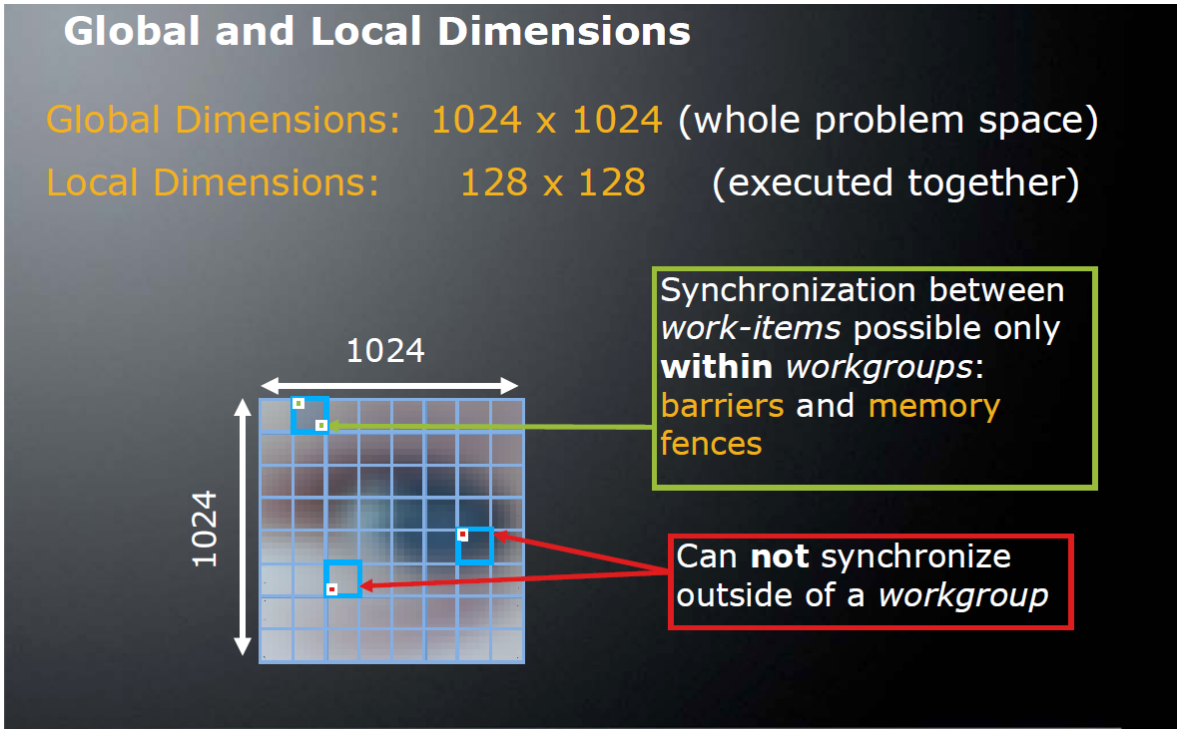

Figura 10: Dimensions locals i globals

El *host* defineix el context on els *kernels* seran executats. El context creat pel *host* mitjançant funcions de l'API d'OpenCL està format per: *devices, kernels, program objects i memory objects*. Els primers són els elements que usarà per executar els *kernels*, després venen les funcions *kernel* en si mateixes, el codi font i executable que implementen les funcions i per últim els objectes de la memòria visibles pels elements computacionals i el *host*. Per poder organitzar l'execució el host crea una cua de comandes on pot incloure comandes per manipular l'execució dels *kernels*, la creació i manipulació de tota mena dels objectes de la memòria i eines per a la sincronització que s'explicaran més endavant. Podem tenir més d'una cua en un mateix i context i aquestes poden funcionar *in-order* o *out-of-order*, esperant a que les comandes anteriors finalitzin o sense fer-ho i forçant les constriccions mitjançant sincronització.

Tipus de *Kernels*

***Kernels* OpenCL**: *kernels* escrits en OpenCL C i compilats amb el compilador inclòs a OpenCL. Compatible amb totes les implementacions d'OpenCL.

***Kernels* natius**: *kernels* escrits en altres llenguatges o provinents d'altres llibreries. Si la implementació és compatible amb ells s'encuen i fan servir la memòria compartida com si fossin *kernels* OpenCL.

2.2.3 Memory Model

La Jerarquia de memòria d'OpenCL defineix quatre nivells amb diferents restriccions d'accés per a cada element del Platform Model. Endreçades de forma descendent de la més interna i ràpida a la que ho és menys.

Memòria privada: només accessible per un únic *work-item.* Inaccessible pel Host.

Memòria Local: accessible exclusivament a nivell de *work-group*. Coneguda com a s*hared memory* a l'arquitectura CUDA i local data share a la família AMD.

Memòria Constant: regió de la memòria global creada i inicialitzada pel Host que no pot ser modificada pels *work-items*, però si llegida i que no canvia durant l'execució.

Memòria Global: memòria accessible per lectura i escriptura tant per qualsevol *work-item* de qualsevol *work-group* com pel Host.

La memòria s'ha d'administrar explícitament del Host a la Memòria Global i d'aquesta a la Local i després seguirà el camí invers.

AMD i Nvidia utilitzen arquitectures, i per tant jerarquies de memòria, diferents. Aquestes diferències existeixen per a cada generació de productes, encara que sortosament la tendència és a eliminar-les, i per tant si es vol treure el màxim profit s'ha de conèixer l'arquitectura sobre la que es treballa com sempre que es tracta amb optimitzacions basades en hardware.

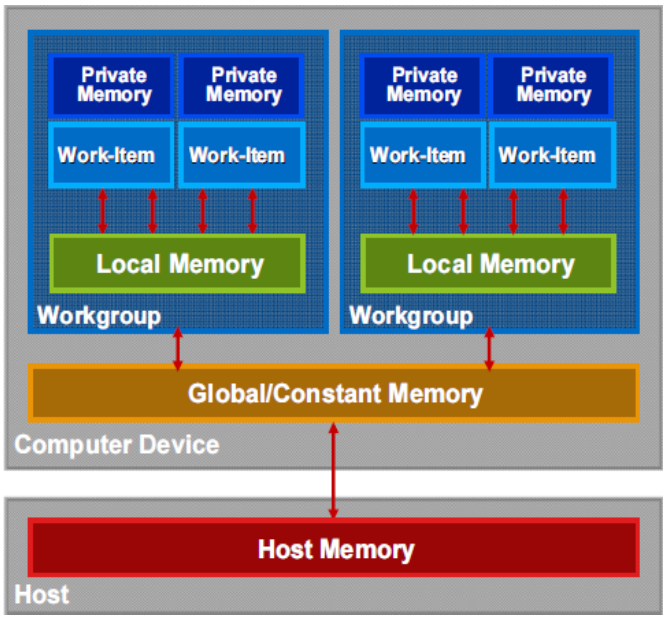

Figura 11: Model de memòria d'OpenCL

Memory objects

El objectes de memòria poden ser *buffer objects* o *image objects*. Els *buffers* emmagatzemen una colecció unidimensional d'elements i en canvi les imatges són bidimensional o tridimensional.

Buffer objects : els *buffers* s'emmagatzemen seqüencialment i poden ser accedits mitjançant un punter. En un *buffer* les dades s'emmagatzemen en el mateix format que usa el *kernel* per accedir-hi.

Image objects*:* les imatges s'emmagatzemen en un format especial, optimitzat i no lineal, per tant no es pot accedir directament als elements mitjançant punters com faríem normalment. Les imatges no s'enmmagatzemen en el mateix format que usa el *kernel* per accedir-hi, sempre són vectors de 4 components en el qual cada component pot ser float, unsinged integer o integer.

2.2.4 Programming Model

El model de programació d'OpenCL està enfocat a la computació de dades en paral·lel, model Single Instruction Multiple Data de la taxonomia de Flynn, però no requereix un aparellament u a u entre els processadors i les dades i de fet no restringeix el model de paral·lelisme i també accepta *task parallelism.*

El model ens permet fixar la distribució dels *work items* en *work groups* explícitament o deixar que OpenCL decideixi com dividirà el número de work items d'execució prèviament designat en grups.

Un altre factor important és la sincronització de processos. A nivell de *work group*, entre els elements que el formen, s'utilitza una barrera de grup: fins que tots els elements executin la comanda cap pot avançar a l'execució, per tant o tots els elements passen la barrera o cap ho fa. No es pot sincronitzar entre diferents *work groups*.

La sincronització a nivell de cua *Command-queue barrier* t'assegura que totes les comandes encuades prèviament s'han executat i que els elements següents podran accedir als possibles canvis realitzats a objectes de la memòria.

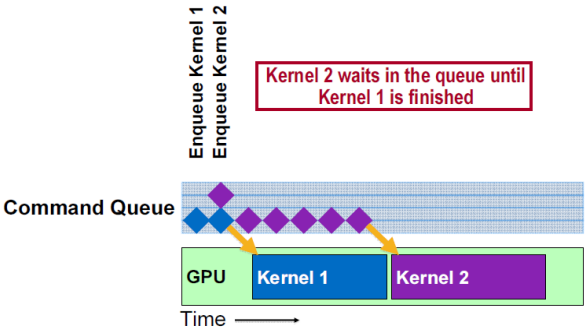

Figura 12: Exemple de *Command-queue barrier*

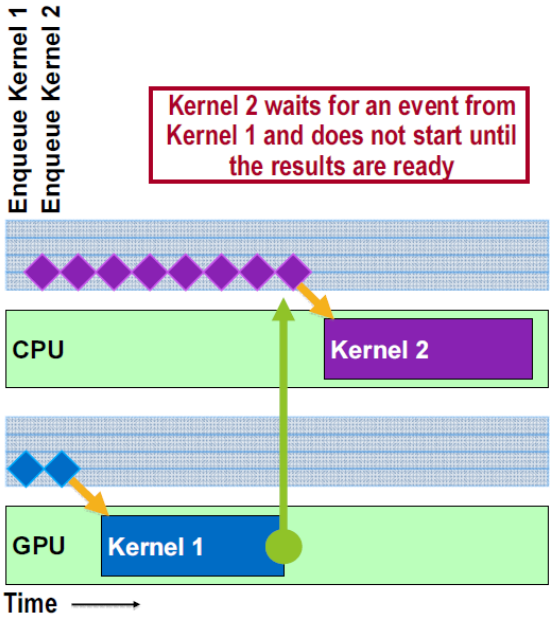

Figura 13: Exemple de *Waiting on an event*

La sincronització entre diferents cues s'anomena *Waiting on an event.* OpenCL llança events per cada comanda realitzada, informant de quins objectes s'han modificat i de quina comanda ho ha fet, per tant les comandes que esperen aquest fet també podran accedir als possibles canvis un cop notificats.

2.3 Suport actual OpenCL

Nvidia

Nvidia porta donant compatibilitat amb OpenCL d'ençà que va aparèixer CUDA i l'arquitectura G80. Per tant, 3 generacions i múltiples revisions de cadascuna d'elles després la llista de productes Nvidia amb capacitat OpenCL/CUDA és molt llarga. Sortosament, unes taules de compatibilitat ajudaran a aclarir les capacitats de cadascuna de les diferents GPUs.

<table>
<tr><th></th><th colspan="5">Compute Capability</th></tr>
<tr><th>Feature Support
(Unlisted features are supported for all compute capabilities)</th><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.0</th></tr>
<tr><td>Integer atomic functions operating on 32-bit words in global memory (Section B.10)</td><td>No</td><td colspan="4">yes</td></tr>
<tr><td>Integer atomic functions operating on 64-bit words in global memory (Section B.10)</td><td colspan="2" rowspan="3">No</td><td colspan="3" rowspan="3">Yes</td></tr>
<tr><td>Integer atomic functions operating on 32-bit words in shared memory (Section B.10)</td></tr>
<tr><td>Warp vote functions (Section B.11)</td></tr>
<tr><td>Double-precision floating-point numbers</td><td colspan="3">No</td><td colspan="2">Yes</td></tr>
<tr><td>Floating-point atomic addition operating on 32-bit words in global and shared memory (Section B.10)</td><td colspan="4" rowspan="4">No</td><td rowspan="4">Yes</td></tr>
<tr><td>__ballot() (Section B.11)</td></tr>
<tr><td>__threadfence_system() (Section B.5)</td></tr>
<tr><td>__syncthreads_count(),
__syncthreads_and(),
__syncthreads_or() (Section B.6)</td></tr>
</table>

Figura 14: Capacitats en funció del perfil suportat

Compute capability (version)	GPUs
1.0	G80
1.1	G86, G84, G98, G96, G96b, G94, G94b, G92, G92b
1.2	GT218, GT216, GT215
1.3	GT200, GT200b
2.0	GF100
2.1	GF104, GF106, GF108

Figura 15: Perfil suportat en funció de l'arquitectura

AMD

La primera arquitectura compatible amb OpenCL d'AMD Graphics Product Group, antiga ATI Technologies Inc, va ser R700. Qualsevol de les diferents iteracions i modificacions d'aquesta arquitectura són compatibles: FireStream 92XX, Radeon 4XXX, Mobility Radeon 4XXX, etc. Al haver estat dissenyada abans que existís l'estàndard d'OpenCL aquesta compatibilitat és molt reduïda i no arriba ni a ser-ho amb la base d'OpenCL 1.0. No suporta cap mena de Image, 2dImage_t o 3dImage_t, ni les operacions atòmiques de cl_khr_global_int32_base_atomics, la memòria local està emulada i resideix realment en la memòria global, etc. Per tant, si bé es compatible per arribar a aprofitar aquesta possibilitat s'ha de programar tenint en compte únicament aquesta arquitectura en ment, complicant molt la feina i allunyant-se molt de l'estàndard. Per això, aquestes gràfiques surten tan mal parades en els *benchmarks* genèrics d'OpenCL.

En canvi, la família de GPUs R800 o Evergreen és la primera d'AMD en suportar l'estàndard OpenCL 1.0 complert. De fet, les gràfiques Radeon de gama alta o FirePro basades en Cypress tenen *double-precision floating-point operations*.

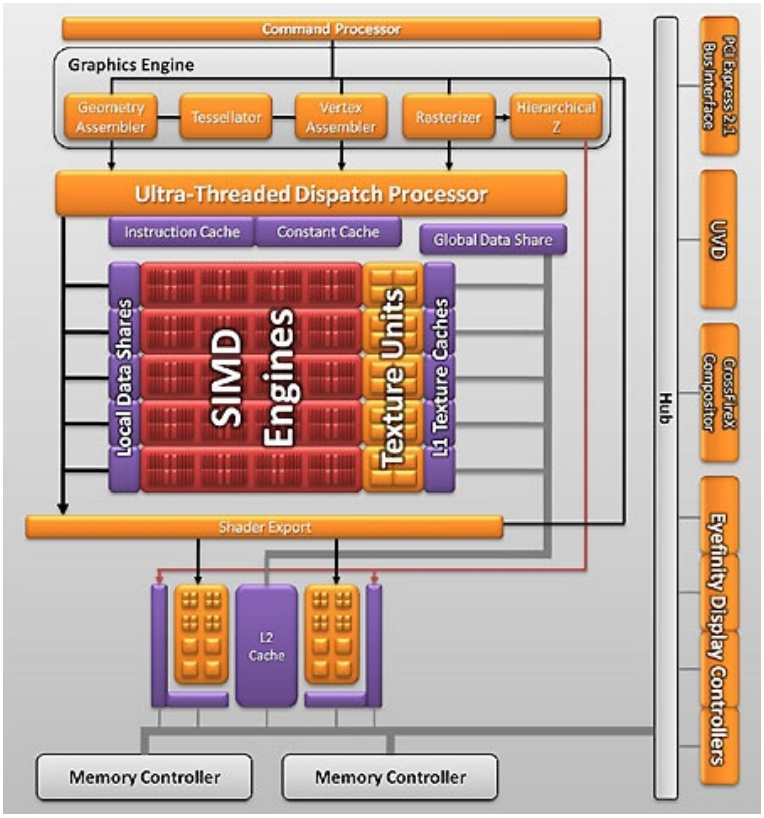

Figura 16: Diagrama de blocs de la Radeon 5570. Gama baixa de la família Evergreen

Qualsevol CPU x86 amb el joc d'instruccions SSE3 o subsegüents pot usar-se per a executar codi OpenCL utilitzant l'AMD Stream SDK.

Exemples de GPUs actuals

La llista de GPUs mostrada a continuació només inclou models actualment disponibles. A finals d'Octubre apareixerà la gama mitja-alta de la nova generació d'AMD, Southern Islands o R900,

anomenades Radeon 6770 i Radeon 6750 i una mica després apareixeran els productes Tesla basats en l'arquitectura GF100 d'Nvidia, però com cap d'aquests productes és al mercat i la majoria de les seves especificacions no estan contrastades no seran inclosos en aquesta llista no exhaustiva.

Totes les GPU són de sobretaula i no hi ha cap amb més d'una GPU com podrien ser una Tesla C1075 o una Radeon 5970.

Nvidia GeForce GTX 285

Arquitectura	GT200b
Shaders	240:80:32
Bandwidth (GB/s)	159.0

Nvidia GeForce GTX 480

Arquitectura	GF100
Shaders	480:60:48
Bandwidth (GB/s)	177.4

AMD Radeon 5870

Arquitectura	Cypress XT - RV870
Shaders	1600:80:32
Bandwidth (GB/s)	153.6

Nvidia Tesla S1060

Arquitectura	GT200
Thread Processors	240
Bandwidth (GB/s)	102.4

FireStream 9370

Arquitectura	Cypress XT - RV870
Shaders	1600:80:32
Bandwidth (GB/s)	147.2

Estructura dels shaders. Quantitat Unified Shaders: Quantitat Texture Mapping Unit: Quantitat Render Output Unit.

Totes aquestes GPUs estan capacitades per a realitzar *double-precision floating-point operations*.

Intel

Intel per la seva part no ofereix compatibilitat actualment amb OpenCL, però a una recent presentació a Alemanya ha explicat el seus plans de futur. OpenCL arribarà a les CPU Intel.

A finals d'aquest any apareixeran les primeres unitats de la nova plataforma basada en l'arquitectura Larrabee anomenada Knights Corner. Competència de Tesla i FireStream amb nuclis x86 natius i que seran compatibles amb OpenCL.

Figura 17: Imatge del primer model de Knights Corner: Knights Ferry

Les GPU integrades Intel, conegudes normalment com Intel Graphics Media Accelerator o Intel GMA, segueixen evolucionant i la nova generació que encara no té nom oficial, però es coneguda com Sandy Bridge IGP al venir integrada al processador d'aquesta arquitectura, si bé sembla tenir un rendiment similar a una Radeon 5450 encara no serà compatible amb OpenCL. Per tant, l'OpenCL a les GPU Intel encara és farà esperar.

3. SURF-OpenCL

En aquest capítol descriurem el funcionament de SURF i el desenvolupament de la versió OpenCL basat en el codi de la implementació d'OpenCV. Per arribar a aquest punt passarem per una versió en C++.

3.1 Algoritme SURF

Speeded Up Robust Features

La primera part de l'algorisme consisteix en la detecció de *keypoints* o punts d'interès.

3.1.1 Detecció dels *keypoints*

Imatge Integral

Primer i degut al seu ús reiterat definim el concepte d'imatge integral. L'imatge integral de x és la suma de tots els píxels de l'imatge d'entrada dins de la regió rectangular formada per l'origen i el punt x. Un cop calculada podem calcular la suma de les intensitats de qualsevol àrea rectangular en només 3 operacions, fet que simplifica i alleugereix l'aplicació de filtres de gran mida com es veurà més endavant.

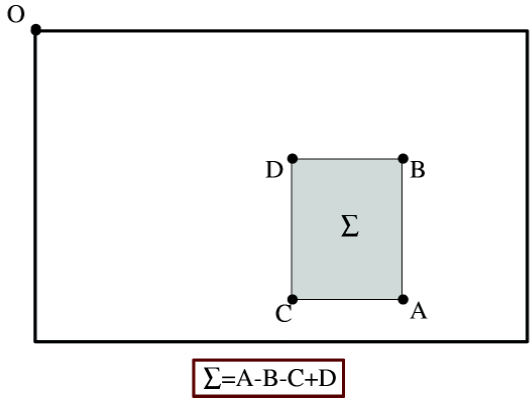

Figura 18: Suma d'intensitats en només 3 operacions i 4 accessos de memòria

Per la seva part, la matriu hessiana dona bons resultats com a detectora, sent ràpida i mantenint la precisió. Serveix per a detectar *blob-like structures*, regions, on el determinant és màxim. També és farà servir per a la selecció d'escales com es veurà després.

Punts d'interès basats en la Matriu Hessiana

La matriu Hessiana del punt x de coordenades (x,y) de l'imatge I a l'escala sigma es defineix com a

$$\mathcal{H}(\mathrm{x}, \sigma) = \begin{bmatrix} L_{xx}(\mathrm{x}, \sigma) & L_{xy}(\mathrm{x}, \sigma) \\ L_{xy}(\mathrm{x}, \sigma) & L_{yy}(\mathrm{x}, \sigma) \end{bmatrix}$$

Figura 19: Definició de la matriu hessiana

on Lxx és la convolució de la derivada segona de la gaussiana amb l'imatge I al punt x, similarment per Lxy i Lyy. Això ens mostra el *rate of change* del gradient d'intensitat idealment cercant *zero-crossing* que ens puguin informar del màxim local del gradient.

Llavors s'usa un determinant aproximat de la derivada segona de la gaussiana que gràcies a l´´ús d'imatges integrals pot ser avaluat molt ràpidament.

$$\det(\mathcal{H}_{\text{approx}}) = D_{xx}D_{yy} - (wD_{xy})^2.$$

Figura 20: Determinant aproximat de la matriu hessiana

Aquests determinants aproximats depenen del pes w de la resposta del filtre que cal per mantenir la conservació de l'energia entre l'original i l'aproximat. Tècnicament s'hauria d'anar re-calculant, però en els experiments no es van notar canvis importants i per això és constant com a 0.9 per a totes les mides de filtre. La resposta als filtres es normalitza respecte la seva mida. Cal recordar que el temps de càlcul és independent de la mida del filtre degut novament a l'imatge integral.

Els determinants aproximats de la hessiana representen la resposta *blob* al punt x de l'imatge I. Aquestes respostes es desen en un mapa de respostes de filtre per les diferents escales calculades, que és d'on s'extreuen el màxims locals.

Creació de l'espai d'escales

Per què ens cal detectar els *keypoints* en diferents escales? Doncs perquè volem comparar diferents imatges i per tant els punts d'interès poden estar a diferents escales i per això cal trobar-los en totes les imatges a totes les escales.

Gràcies, de nou, a l'us d'imatges integrals i de *box filters* podem aplicar els filtres de qualsevol mida sempre sobre l'imatge original al mateix cost evitant l'aliàsing que provoca la reducció de l'imatge original. L'anàlisi de l'espai per totes les escales s'aconsegueix aplicant filtres cada vegada més grans, començant per una màscara de 9 x 9.

Aquest espai d'escales és dividit en octaves. Una octava és la resposta als filtres aplicats a l'imatge d'entrada amb filtres de màscares cada cop més grans. El factor d'escalat d'una octava és 2, com s'explica més endavant.

Les octaves contenen un número constant de subdivisions per representar les diferents escales. El

factor d'escalat és 2 i depèn de la mida de la resposta positiva o negativa de la derivada parcial segona en la direcció de la derivació, x o y, que està fixada en un terç. Per tant, en el cas inicial del filtre 9 x 9 és 3. Això provoca que l'increment sigui de 6 píxels per banda, fent que el següent filtre sigui 15 x 15 i els subsegüents 21 x 21 i 27 x 27 per la primera octava, mantenint la mida senar i per tant el píxel central. Al aplicar una supressió tridimensional dels valors no màxims el primer i últim mapa de les respostes hessianes no poden contenir aquest màxims i per tant es fan servir només per comparacions. Per més informació referencia [2].

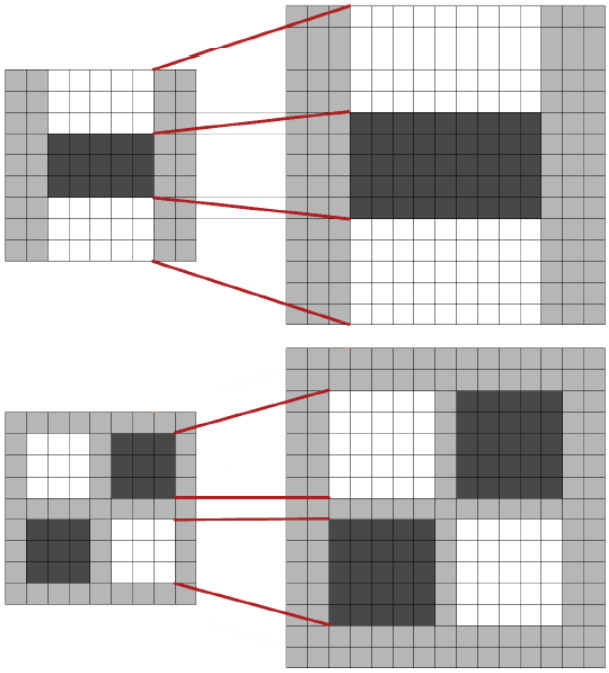

Figura 21: Màscara de 9 x 9 al costat de la de 15 x 15, mostrant la conservació del punt central

Això s'aplica per a cadascuna de les escales on l'augment dels intervals per cada extracció dels punts d'interès es dobla per a cada nova octava com es veu a cada imatge fins que el filtre aplicat sigui major que l'imatge d'entrada. En cada ampliació trobant cada vegada menys punts d'interès.

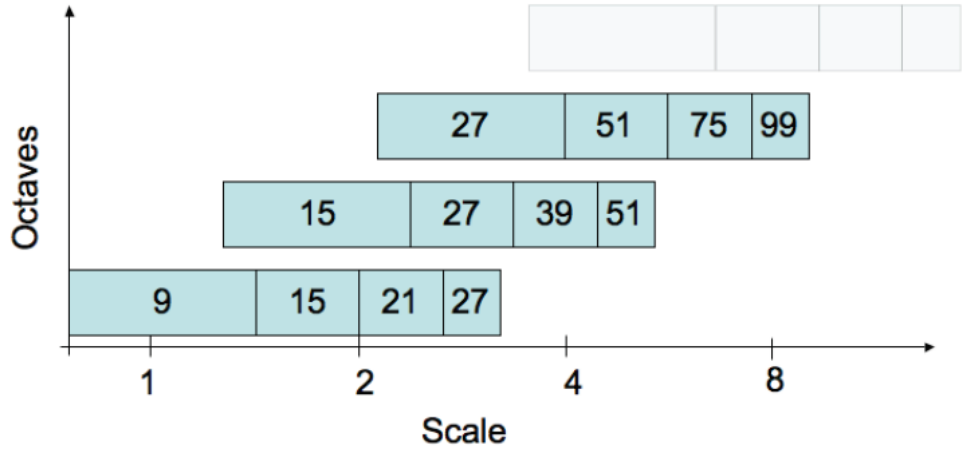

Figura 22: Mides dels filtres per a les tres primeres octaves.

Després de passar per la supressió 3 x 3 x 3 abans descrita el màxim del determinant de la matriu hessiana s'interpola fent servir el mètode de Brown referència [3]. Ja tenim punts d'interès.

3.1.2 Descriptor

El descriptor descriu la distribució de la intensitat dins el sector del punt d'interès.

Amb aquesta idea s'usa un vector de 64 amb les respostes en direcció x i y al *Haar wavelet.* La meitat que les 128 dimensions del SIFT cosa que incrementa la velocitat de nou gràcies a l'ús d'imatges integrals. També millora la comparació i a més incrementa la robustesa dels resultats.

Orientació dels punts d'interès

Primer calculem la resposta al *Haar wavelet* en les direccions x i y en un radi circular 6s, on s és l'escala a la que el *keypoint* fou trobat. La mida del wavelet també depèn de l'escala i és designat per 4s. Tornam a aprofitar-nos de l'imatge integral per obtenir les respostes x i y en tan sols 6 operacions per a qualsevol escala.

Una vegada les respostes són calculades i passen per un filtre gaussià 2s, per suavitzar, centrat al punt d'interès, aquestes respostes es representen com a punts en l'espai bidimensional. L'orientació dominant s'estima sumant totes les respostes dins d'una finestra corredissa de mida pi terços. Les respostes vertical i horitzontal es sumen per separat trobant un vector d'orientació local. El més gran de tots sobre la finestra corredissa defineix l'orientació del punt d'interès.

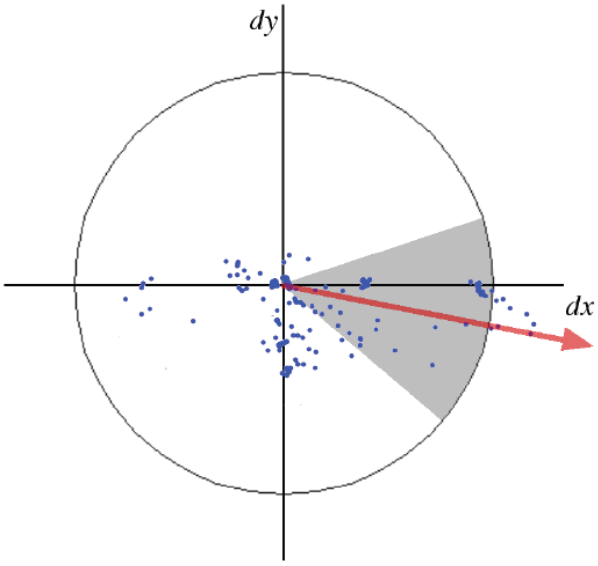

Figura 23: Exemple del mètode de la finestra corredissa mostrant una orientació local

Descriptor basat en la suma de respostes al *Haar wavelet*

Per extreure el descriptor primer cal construir unes regions quadrades de mida 20s centrades al punt d'interès i que a més a més segueixin la seva orientació. Dividim aquestes regions ens àrees de mida 4 x 4 i per cadascuna d'elles apliquem un *Haar wavelet* de mida 5s. Anomenarem aquestes respostes dx en el cas de l'horitzontal i dy per la vertical, orientacions definides en funció de l'orientació del punt d'interès. Posteriorment s'aplica un altre filtre gaussià, aquest cop de mida 3.3s, centrat al punt d'interès. Ara es sumen les respostes en dx i dy de cada subregió 4 x 4 i també dels valors absoluts, denominats |dx| i |dy|, per aportar informació sobre la polaritat dels canvis d'intensitat, obtenint d'aquesta manera un vector de quatre components que defineix l'estructura de la intensitat per aquesta subdivisió. Concatenant això per cadascuna de les subregions obtenim un vector de dimensió 64 que ens defineix la resposta per a tota la regió.

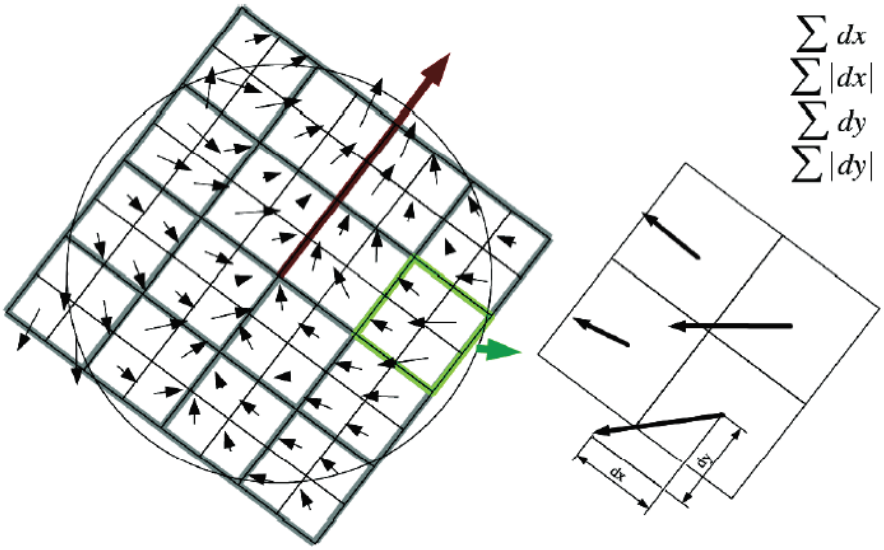

Figura 24: Exemple d'orientacions per les regions 4 x 4 i per la regió que les conté

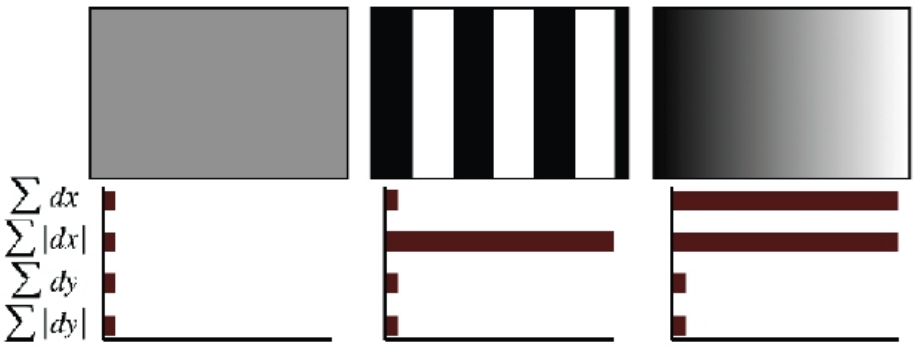

Figura 25: Exemples de respostes al *Haar wavelet* i de com les diferents combinacions d'aquestes defineixen l'estructura de la intensitat a l'imatge.

La resta de l'algorisme es refereix a la part de *matching* que no ha estat implementada en aquest projecte i per tant s'eludeix en aquest capítol.

Després d'explicar punt a punt l'algorisme és fàcil veure que consta de 6 fases:

- Càlcul d'imatge integral.
- Càlcul de la matriu de determinants.
- Cerca del màxim.
- Optimització dels punts trobats.
- Càlcul d'Orientacions.
- Càlcul del descriptor.

3.2 Desenvolupament versió OpenCL

Procés i idea original

El procés de desenvolupament s'ha vist afectat per moltes dificultats tècniques, endarrerint greument la planificació inicial. Al final el procés s'ha composat de cinc fases:

- Preparació del sistema operatiu per poder compilar i executar tot el codi necessari pel desenvolupament de la mateixa. Instal·lació dels *drivers* propietaris d'AMD, de l'Stream SDK amb l'ICD, d'OpenCV i de qualsevol dependència necessària.

- Llegir la versió revisada al 2008 del paper de SURF.

- Llegir i aconseguir desxifrar el codi de la versió OpenCV afegint-hi comentaris tant sobre el codi mateix com sobre les funcions pròpies d'OpenCV.

- Realitzar versió en C++ desfent les funcions d'OpenCV i fent-la més entenedora i senzilla en general.

- Realitzar la versió OpenCL basada en l'anterior i que aprofiti en tot lo possible el paral·lelisme de la GPU.

L'idea inicial era implementar el detector de keypoints de SURF en OpenCL, que fos compatible amb Nvidia i AMD i que acceleres el procés tot lo possible sense entrar en optimitzacions per arquitectures concretes. A posteriori, s'ha acabat incloent també la part del descriptor i si ve funciona en GPUs d'ambdues companyies els resultats a la banda d'AMD no són correctes i ha estat impossible arreglar-ho. Sortosament, la part del detector de punts d'interès funciona correctament en totes les gràfiques.

Inicialització codi OpenCL

Per poder executar codi OpenCL primer cal realitzar tota la inicialització. Tota l'estructura bàsica és la mateixa per a qualsevol programa. Primer es crea el p*latform*, després el *context* on on s'executarà el codi i que serà, per exemple, CPU o GPU.

Ara és el torn de la cua al context desitjat. Ara tota crear tots els objectes de memòria i anar-los encuant a la cua del context amb els *flags* i opcions correctes.

Ara toca crear el programa i fer la compilació del mateix amb program.build . Ara toca crear els 5 *kernels* i col·locar-los al programa que acabem de crear. Ara toca col·locar els arguments, si cal, i executar els *kernels* amb NDRange més adient per cadascun d'elles.

Després cal copiar els resultats del *device* al *host*. Amb `clEnqueueReadBuffer` o `queue.enqueueReadBuffer` en el nostre cas.

Després s'allibera la memòria dels objectes.

```
// SI VA LA GPU, GPU, SI NO CPU
            try {
                context = cl::Context(CL_DEVICE_TYPE_GPU, cps );
            } catch (cl::Error &err ) {
                context = cl::Context(CL_DEVICE_TYPE_CPU, cps );
            }


            isGPUcapable = context.getInfo<CL_CONTEXT_DEVICES>().size() != 0;
            std::cout << "Using: " << context.getInfo<CL_CONTEXT_DEVICES>()[0].getInfo<CL_DEVICE_NAME>() << std::endl;

// CREEM UNA CUA DEL DEVICE 0
            queue    = cl::CommandQueue(context, context.getInfo<CL_CONTEXT_DEVICES>()[0] );

// CREEM BUFFERS I ENVIEM LES DADES DE LES CONSTANTS
            imgCL    = cl::Image2D( context, CL_MEM_READ_ONLY, cl::ImageFormat(CL_R, CL_UNSIGNED_INT8), width,  height);

            sumCLA   = cl::Image2D( context, CL_MEM_READ_WRITE, cl::ImageFormat(CL_R, CL_UNSIGNED_INT32), swidth,  sheight);
            sumCL    = cl::Image2D( context, CL_MEM_READ_WRITE, cl::ImageFormat(CL_R, CL_UNSIGNED_INT32), swidth,  sheight);

            kPtsCL     = cl::Buffer( context, CL_MEM_READ_WRITE,   swidth*sheight*sizeof(float) );

            dxDetCL  = cl::Buffer( context, CL_MEM_READ_ONLY, (NLAY+2)*3*5*sizeof(float) );
            dyDetCL  = cl::Buffer( context, CL_MEM_READ_ONLY, (NLAY+2)*3*5*sizeof(float) );
            dxyDetCL = cl::Buffer( context, CL_MEM_READ_ONLY, (NLAY+2)*4*5*sizeof(float) );
            queue.enqueueWriteBuffer( dxDetCL,  true, 0, (NLAY+2)*3*5*sizeof(float), (void*)DX_DET);
            queue.enqueueWriteBuffer( dyDetCL,  true, 0, (NLAY+2)*3*5*sizeof(float), (void*)DY_DET);
            queue.enqueueWriteBuffer( dxyDetCL, true, 0, (NLAY+2)*4*5*sizeof(float), (void*)DXY_DET);

            dxOriCL  = cl::Buffer( context, CL_MEM_READ_ONLY, (MAX_IMAGE_DIM/2)*2*5*sizeof(float) );
            dyOriCL  = cl::Buffer( context, CL_MEM_READ_ONLY, (MAX_IMAGE_DIM/2)*2*5*sizeof(float) );
            queue.enqueueWriteBuffer( dxOriCL,  true, 0, (MAX_IMAGE_DIM/2)*2*5*sizeof(float), (void *)DX_ORI );
            queue.enqueueWriteBuffer( dyOriCL,  true, 0, (MAX_IMAGE_DIM/2)*2*5*sizeof(float), (void *)DY_ORI );

            aptxCL   = cl::Buffer( context, CL_MEM_READ_ONLY, NANGLES*sizeof(int) );
            aptyCL   = cl::Buffer( context, CL_MEM_READ_ONLY, NANGLES*sizeof(int) );
            aptwCL   = cl::Buffer( context, CL_MEM_READ_ONLY, NANGLES*sizeof(float) );
            queue.enqueueWriteBuffer( aptxCL,  true, 0, NANGLES*sizeof(int),   (void *)APTX );
            queue.enqueueWriteBuffer( aptyCL,  true, 0, NANGLES*sizeof(int),   (void *)APTY );
            queue.enqueueWriteBuffer( aptwCL,  true, 0, NANGLES*sizeof(float), (void *)APTW );

            dwCL     = cl::Buffer( context, CL_MEM_READ_ONLY,     400*sizeof(float) );
            queue.enqueueWriteBuffer( dwCL,  true, 0, 400*sizeof(float), (void *)DW );
```

Figura 26: Inicialització tal com és al codi OpenCL de SURF.

El codi consta de 5 *kernels* i 5 funcions auxiliars, encara que fastAtan2 pot ésser substituïda per ata2pi de la llibreria d'OpenCL. Els dos primers *kernels* són per calcular l'imatge integral, el tercer per trobar els *keypoints*, el quart per calcular les seves orientacions i el cinquè per a calcular els descriptor.

Per entendre com funcionen, s'ha de tenir en compte que el codi dels *kernels* s'executa a la vegada i no seqüencialment com explica aquest exemple sobre NDRange.

Exemple per entendre NDRange

```
work_dim = {n}
```

Codi normal

```
Codi:
for (int i=0; i < n; i++)
{
     //instruccions a realitzar
}
```

Codi OpenCL

```
Codi:
kernel ( /*arguments*/ )
{
    i = get_global_id(0);
   //instruccions a realitzar
}

EnqueueNDRange(el_teu_kernel, work_items, etc)


work_dim = {n, p}
```

Codi normal

```
Codi:
for (int i=0; i < n; i++)
{
   for (int j=0; j < p; j++)
   {
     //instruccions a realitzar
   }
}
```

Codi OpenCL

```
Codi:
kernel ( /*arguments*/ )
{
    i = get_global_id(0);
    j = get_global_id(1);
   //instruccions a realitzar
}

EnqueueNDRange(el_teu_kernel, work_items, etc.)
```

Kernels

La part més important de qualsevol codi OpenCL són els *kernels*.

Els *kernels* integralA i integralB calculen l'imatge integral en dues passades, una horitzontal i l'altre vertical. Aquesta solució dista de ser ideal ja que la implementació clàssica és més ràpida que aquesta d'ambdues passades fins i tot a la GTX 470. El NDRange és unidimensional en aquests dos *kernels* i les seves xifres es van fixar en 128 i 32.

El tercer kernel findKeyPoints conté un NDRange bidimensional 32, 4 i es dedica a trobar els punts d'interès.

```
            int l0 = get_local_id(0);
            int l1 = get_local_id(1);
            if( l0 < 15 ) ((local float4 *)hdx) [l0] = ((global float4 *)DX_DET )[l0];
            if( l0 < 15 ) ((local float4 *)hdy) [l0] = ((global float4 *)DY_DET )[l0];
            if( l0 < 20 ) ((local float4 *)hdxy)[l0] = ((global float4 *)DXY_DET)[l0];
            mem_fence( CLK_LOCAL_MEM_FENCE );

// CREEM UNA CACHE PER NO CALCULAR 2 VEGADES EL MATEIX DETERMINANT
            local float cache[64*8*4];
            for(int x=0; x<2*8*4; x++) ((local float *)cache)[x*32+l0]= -100.f;
            mem_fence( CLK_LOCAL_MEM_FENCE );

            for( int layer = 1; layer <= NLAY; layer++ ) {

                int margin = (SIZE(layer+1)>>1)+1;
                // Ignore pixels without a 3x3 neighbourhood in the layer above

                if( x < margin ) continue;
                if( y < margin ) continue;

                if( x >= WIDTH -margin  ) continue;
                if( y >= HEIGHT-margin ) continue;

                float N9[3*9];
                float val = N9[13] = calcDet( sum, hdx, hdy, hdxy, x, y, layer );

                if( val > hessianThreshold ) {

                    float mx=-1E10f;
                    for( int nz = 0; nz < 3 && mx<val; nz++ )
                        for( int ny = 0; ny < 3 && mx<val; ny++ )
                            for( int nx = 0; nx < 3 && mx<val; nx++ )
                                if( nx != 1 || ny != 1 || nz != 1 )
                                mx = max( mx, N9[nz*9+ny*3+nx] = calcDet( sum, hdx, hdy, hdxy ,x+nx-1, y+ny-1, layer+nz-1) );
```

Figura 27: Part del *kernel* findKeyPoints després de l'inicialització.

El *kernel* calcOri funciona com s'ha descrit al desenvolupament de SURF.

```
local float hdx[2*5*64];
local float hdy[2*5*64];
for(int x=0; x<2*5; x++) hdx[l*10+x] = DX_ORI[(grWavSize>>1)*5*2+x];
for(int x=0; x<2*5; x++) hdy[l*10+x] = DY_ORI[(grWavSize>>1)*5*2+x];
barrier( CLK_LOCAL_MEM_FENCE );

for( int i = 0; i < 64; i ++ ) {
    X[i]=0;
    Y[i]=0;
}
barrier( CLK_LOCAL_MEM_FENCE );

for( int k = 0; k < NANGLES; k++ ) {

    float x = round( px + APTX[k]*s - ((float)(grWavSize-1))/2 );
    float y = round( py + APTY[k]*s - ((float)(grWavSize-1))/2 );

    float cx = calcHaarPattern( sum, x, y, &hdx[l*10], 2 ) * APTW[k];
    float cy = calcHaarPattern( sum, x, y, &hdy[l*10], 2 ) * APTW[k];

    int angle = round(fastAtan2(cy, cx))*(64.f/360.f);
    X[angle] += cx;
    Y[angle] += cy;
    barrier( CLK_LOCAL_MEM_FENCE );
}
```

Figura 28: Nucli de l'execució de calcOri

El cinquè *kernel* anomenat calcDesc l'NDRange és unidimensional i permet que el codi funcioni de la forma

La mida del NDRange ve fixada pel número de dimensions del descriptor, 64 en aquest cas i de ser una implementació de SIFT OpenCL 128.

```
size_t d = get_global_id(0);
if( d < nPts ) {

        Càlcul de les subregions
        Construcció del vector[64] del descriptor
        Normalització del vector
}
```

nPs és un dels arguments del *kernel.*

Cal recordar que fixar la mida del NDRange és moltes vegades un procés de proba i error.

Al capítol següent s'expliquen alguns dels detalls per modificar el codi i poder activar altres opcions. Per fer servir aquestes parts només cal navegar per la part inicial de de surf.h i descomentar les línies comentades i comentar les línies següents que són les que s'executen normalment.

4. Resultats

Capítol on es mostren les imatges utilitzades, com interpretar la sortida, els resultats experimentals i d'altres detalls.

4.1 Dades i Mètode

Tots els tests s'han realitzat per a les versions OpenCV, SURF-C++ i SURF-GPU, i sempre per una imatge petita i una altre de gran encara que també n'hi ha molts per la mitjana.

Figura 29: Imatge petita box.png 324x223

Figura 30: Imatge de mida mitjana box_in_scene.png 512x384 píxels

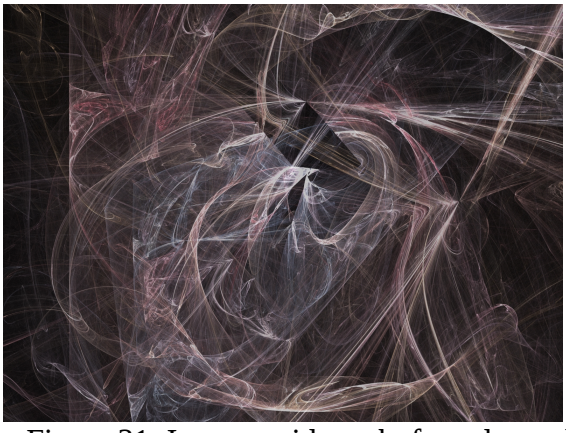

Figura 31: Imatge spiderweb_fractal.png de 1600x1200 píxels

Col·loquem el número d'iteracions del test en 20, realitzant per tant 60 vegades l'algorisme SURF entre les tres versions. La majoria de comparacions les farem entre la versió C++ i la OpenCL ja que la versió OpenCV normalment és molt més lenta.

4.2 Experiments i criteris de valor

Pels experiments treballarem amb 3 equips diferents amb GPUs de diferents arquitectures i preus, incloent un portàtil amb una gràfica integrada per veure com de viable és actualment OpenCL en aquesta mena de productes. Especificacions resumides dels equips de treball indicant quin sistema operatiu s'ha fet servir.

Equip Portàtil – Baixa Potència

Pentium Dual-Core T4400 2 nuclis a 2'2GHz
Nvidia 9100M G 256MB de la memòria DDR2 principal bus de 128bits 1 SIMD
2x2GB de DDR2
Windows 7 64bits

Equip AMD – Potència mitjana

AMD Phenom II X3 720 3 Nuclis a 2'8GHz
AMD Radeon 5670 512MB GDDR5 bus de 128bits 5 SIMD
2x2GB DDR3
Ubuntu 10.04 64bits

Equip Nvidia – Alta Potència

Intel Core i5 750 4 nuclis a 2'66GHz
Nvidia GTX 470 1280MB GDDR5 bus de 320bits 14 SIMD
4x2 GB DDR3
Ubuntu 10.04 64bits

Ara explicarem pas a pas com s'interpreta la sortida del programa per més endavant poder analitzar els resultats.

El programa primer ens indica de quina imatge es tracta, quines són les seves mides i també del nom de la GPU utilitzada.

Exemple, part 1:

Image: box.png
Size: 324, 223
Using: Redwood

A continuació apareixen els temps totals per a cada versió del programa tantes vegades seguides com iteracions s'hagin especificat en l'arxiu test.cpp.

Exemple, part 2:

OPENCV: 46.1ms
OPENCV: 39.0ms
OPENCV: 50.1ms
SURF-C++: 47.3ms
SURF-C++: 45.7ms
SURF-C++: 70.9ms
SURF-GPU: 117.3ms
SURF-GPU: 51.9ms
SURF-GPU: 47.4ms

Després apareix una taula amb els temps mínim, màxim i mitjà per a cada algoritme. En el temps mitjà no és tenen en compte els temps màxim i mínim.

Exemple, part 3:

SURF-CV: Min: 36.8ms Max: 55.2ms Avg: 44.4ms
SURF-C++: Min: 25.5ms Max: 70.9ms Avg: 35.9ms
SURF-GPU: Min: 22.6ms Max: 117.3ms Avg: 40.9ms

Ara arriba de l'anàlisi de la sortida. La quantitat de *keypoints* aconseguida per la versió OPENCV és considerada la correcta, per tant a l'apartat Comparable keypoints dels altres dos algorismes el percentatge va en funció dels extrets per la versió OpenCV. Això també és aplicat a les altres dades: error en distància euclídia, error en l'orientació i error en la creació del descriptor. Aquesta última és normal que romangui entre el 5% i el 10% degut a la simplificació de la funció resize en vers a la molt completa i complicada versió cvResize de la llibreria OpenCV.

Exemple, part 4:

SURF-CV:467keypoints

SURF-C++: 465 keypoints
Comparable keypoints: 465(99.6%)
Distance: 0.000 pixels
Orientation: 0.000%
Error 4.6%

SURF-GPU: 467 keypoints
Comparable keypoints: 467(100.0%)
Distance: 0.000 pixels
Orientation: 0.741%
Error 10.0%

En aquest últim apartat apareixen els temps que ha trigat cada iteració en desenvolupar diferents seccions del codi per les implementacions C++ i OpenCL. S'inclouen les inicialitzacions.

Exemple, part 5:

CPU1 calcIntegralImg	0.16	0.17	0.17
CPU2 calcKeyPoints	31.88	26.55	38.44
CPU3 calcOrientations	8.13	7.65	30.21
CPU4 calcDescriptors	7.14	11.32	2.07
GPU1 writeImage	64.84	0.67	4.92
GPU2 integralA	1.08	0.18	0.40
GPU3 integralB	1.00	0.22	14.91
GPU5 findKeyPts	42.68	18.19	19.91
GPU7 calcOri	3.26	3.11	3.10
GPU8 calcDesc	3.65	3.45	3.41
INIT1 compile	3732.46		
INIT2 getKernels	0.01		

Criteris

El criteri bàsic pels tests és la velocitat dels diferents càlculs, especialment la del total excloent-hi òbviament el temps de compilació dels *kernels*, sempre i quan els resultats siguin correctes i no presentin una quantitat d'errors massa elevada. Degut això, primer comentarem breument els resultats de l'equip AMD, que no calcula correctament del pas de l'orientació en endavant, i després passarem al de la resta d'equips que mostren resultats correctes. No hi haurà gràfiques per a cada comentari, però s'inclouran els fitxers amb els resultats amb el codi del projecte.

Els resultats per les diferents iteracions d'un algorisme en les mateixes condicions són òbviament iguals, però el temps d'execució va fluctuant, encara que la tendència és que romangui en un rang estable per tots els algoritmes en qualsevol plataforma.

Equip AMD

Gràfica que contempla el comput total encara que del càlcul de *keypoints* en endavant el resultat és incorrecte. La majoria de possibles canvis no presenten modificacions importants del rendiment i cap el millora notòriament.

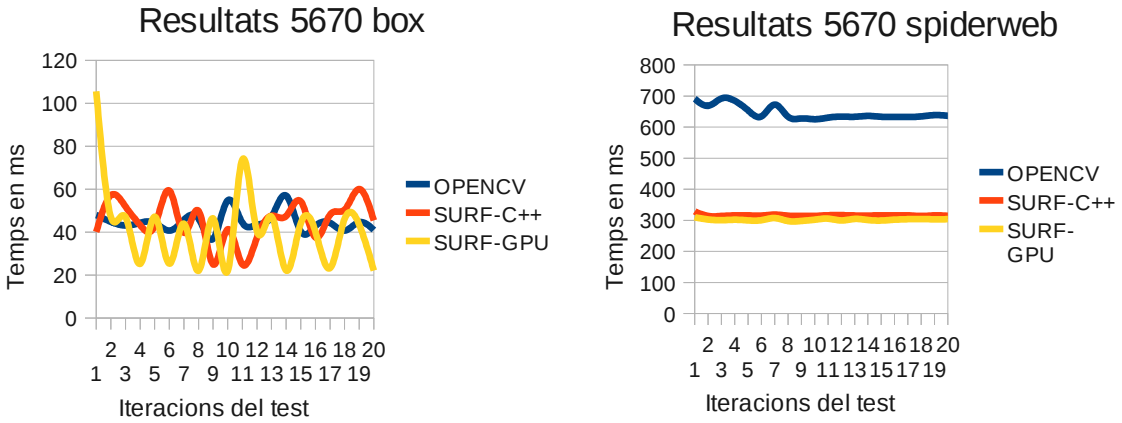

Figura 32: Resultats per box i spider de la Radeon 5670 comparada amb les altres versions

Equip Portàtil

Com es veurà els resultats de la versió OpenCL normal sobre aquesta gràfica són molt pobres, però era lo raonable tenint en compte que és una gràfica integrada per portàtils de gama baixa llançada al 2008. Encara que les gràfiques d'aquesta gama són tant senzilles que l'equivalent actual, GeForce 305M, només dobla en potència la 9100M G d'aquesta prova.

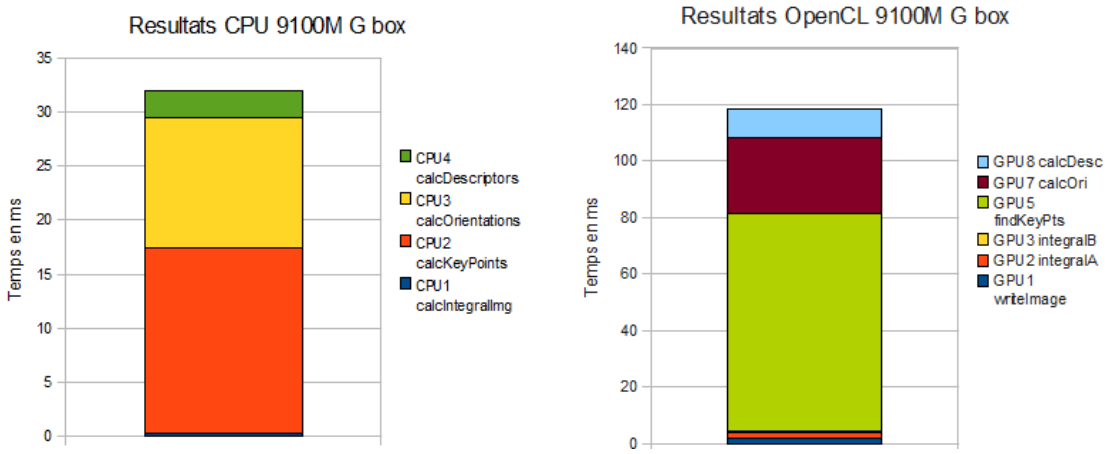

Figura 33: Resultats triplicant el temps de computació en CPU

Per l'imatge gran la diferència és la mateixa. Veiem com funciona sense memòria cau.

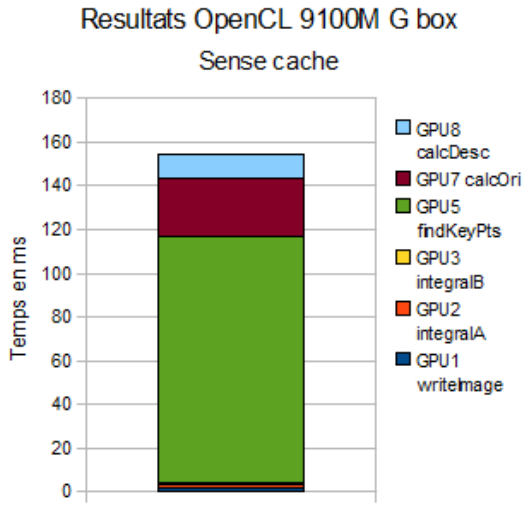

Figura 34: Resultats sense cache un 20% pitjors

De nou la diferència per l'imatge gran és la mateixa, sense memòria cau el rendiment baixa un 20%.

La memòria local presenta uns resultats una mica millor que els originals. Una còpia del determinant a la memòria local podrà millorar els resultats?

La còpia d'un en un fent servir float fa que vagi més del doble de lenta per l'imatge petita i per la gran no funciona ja que no entra en memòria. Els resultats de las versions de float2 i float4 milloren els resultats per l'imatge petita sense arribar als de la versió original amb cache. La resta de les modificacions no semblem produir millor resultats que l'original.

Equip Nvidia

La plataforma més potent d'Nvidia presenta uns resultats molt satisfactoris.

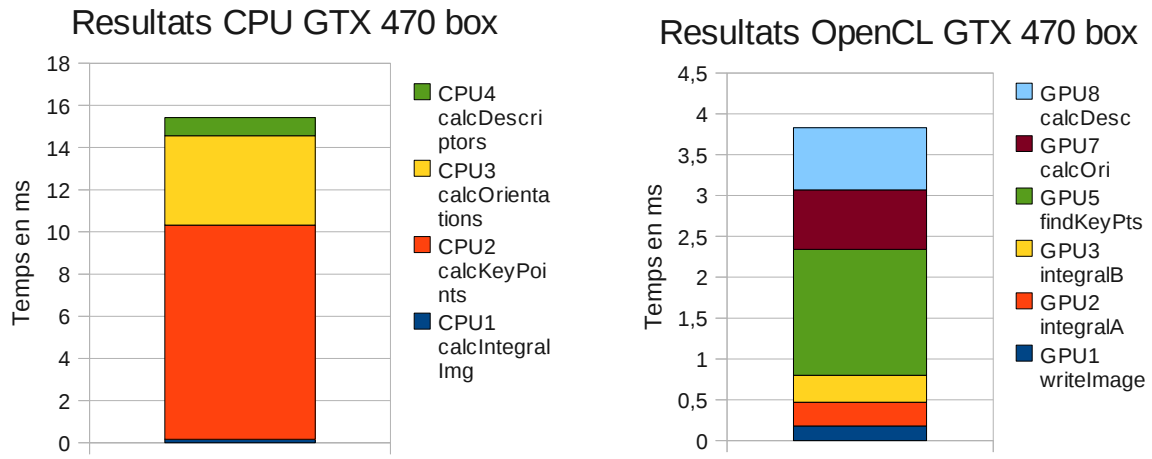

Figura 35: Resultats de la versió C++ i OpenCL per box

Com es pot veure a simple vista la versió OpenCL triga una mica més d'una cinquena part en computar l'imatge petita. Amb un error d'orientació del 0.127% i l'error del descriptor al 7.4% dins dels paràmetres normals per la nostra versió. La versió OpenCV triga 25ms en fer la mateixa feina.

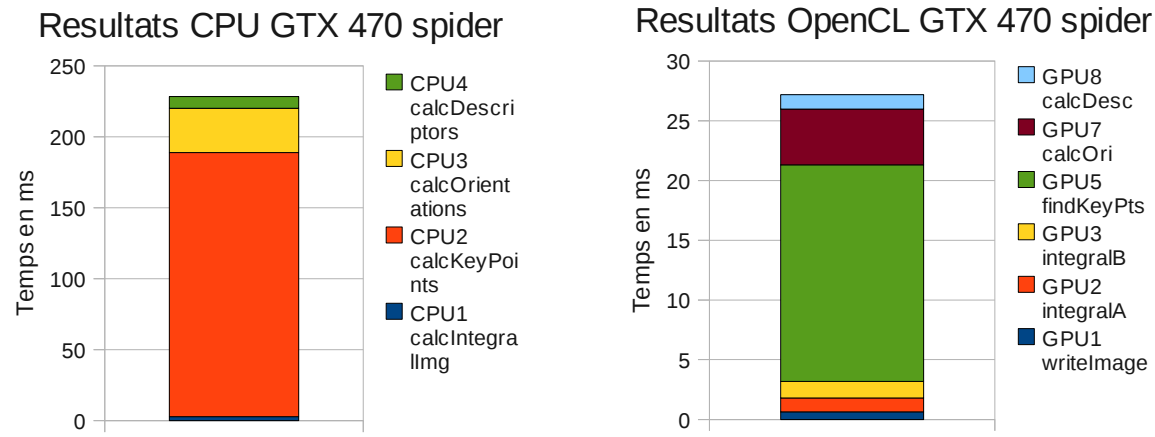

Figura 36: Resultats de la versió C++ i OpenCL per spider

En l'imatge gran els resultats són encara millors aconseguint una millora de casi un ordre de magnitud. La versió OpenCV triga de l'ordre de 480ms.

Si no s'utilitza la memòria cau els resultats empitjoren un 10%. La resta de versions no modifiquen els resultats positivament i tampoc els empitjoren més enllà d'un 15%.

5. Conclusions i treball futur

Els resultats de la GTX 470 són satisfactoris i més tenint en compte que no s'ha optimitzat pensant en una arquitectura ja que l'objectiu era poder executar el codi correctament en qualsevol arquitectura AMD o Nvidia. Dissortadament això no ha estat possible i ara mateix sembla que l'única manera d'aconseguir una implementació correcta i ràpida per AMD és fer-la pensant exclusivament en les seves característiques i per tant desaprofitant la compatibilitat d'OpenCL amb Nvidia. Això a un entorn d'experimentació no és un problema, però de cara a tenir a una aplicació de visió artificial que funcioni en qualsevol aparell amb una arquitectura actual és un problema.

La implementació OpenCL de SURF té sentit per GPUs de gama mitja-alta o alta d'Nvidia llançades de 2007 o 2008 en endavant tenint en compte les seves especificacions i les dels equips dels experiments realitzats.

Veient els productes anunciats de cara a finals d'any i inicis de 2011 sembla factible que portàtils amb gràfiques integrades puguin executar aquest SURF OpenCL més ràpid que les seves contrapartides a la CPU i en 2 anys més els *smartphones* arribaran a una potència similar. Per tant, queda un llarg camí per aquesta tecnologia en l'àmbit de la portabilitat.

L'ample de banda i la latència del bus d'interconnexió entre la CPU i la GPU pot produir, i sense canvis en el mateix produirà, un coll d'ampolla molt important fet que farà que en un futur proper el sistemes de connexió entre aquests dos elements canviïn. Possiblement això causarà canvis importants en aquesta mena d'implementacions.

Referències

[1] A. Chariot and R. Keriven. From GPU-boosted online image matching . CERTIS, Ecole des ponts, Paris-Est, France 2007.

[2] H. Bay. From Wide-baseline Point and Line Correspondences to 3D. PhD thesis, ETH Zurich, 2006.

[3] M. Brown and D. Lowe. Invariant features from interest point groups. In BMVC, 2002.

Annex A:

El contingut del cd-rom és a dins de la carpeta surf_definitiu. Allà tenim tot el material necessari per executar el codi, incloent un *makefile*. Per defecte, ./test agafa box.png com a objectiu, en cas de voler una altra imatge especificar després ./test imatge1.png.

Dins de la carpeta CL es troben els *headers* necessaris pel OpenCL C++. A la carpeta resultats hi ha resultats experimentals dels comentats al capítol 4.

Annex B:

Una forma fàcil i ràpida per saber si la nostra gràfica és compatible amb OpenCL és descarregar l'aplicació gratuïta GPU-Z de techPowerUp!

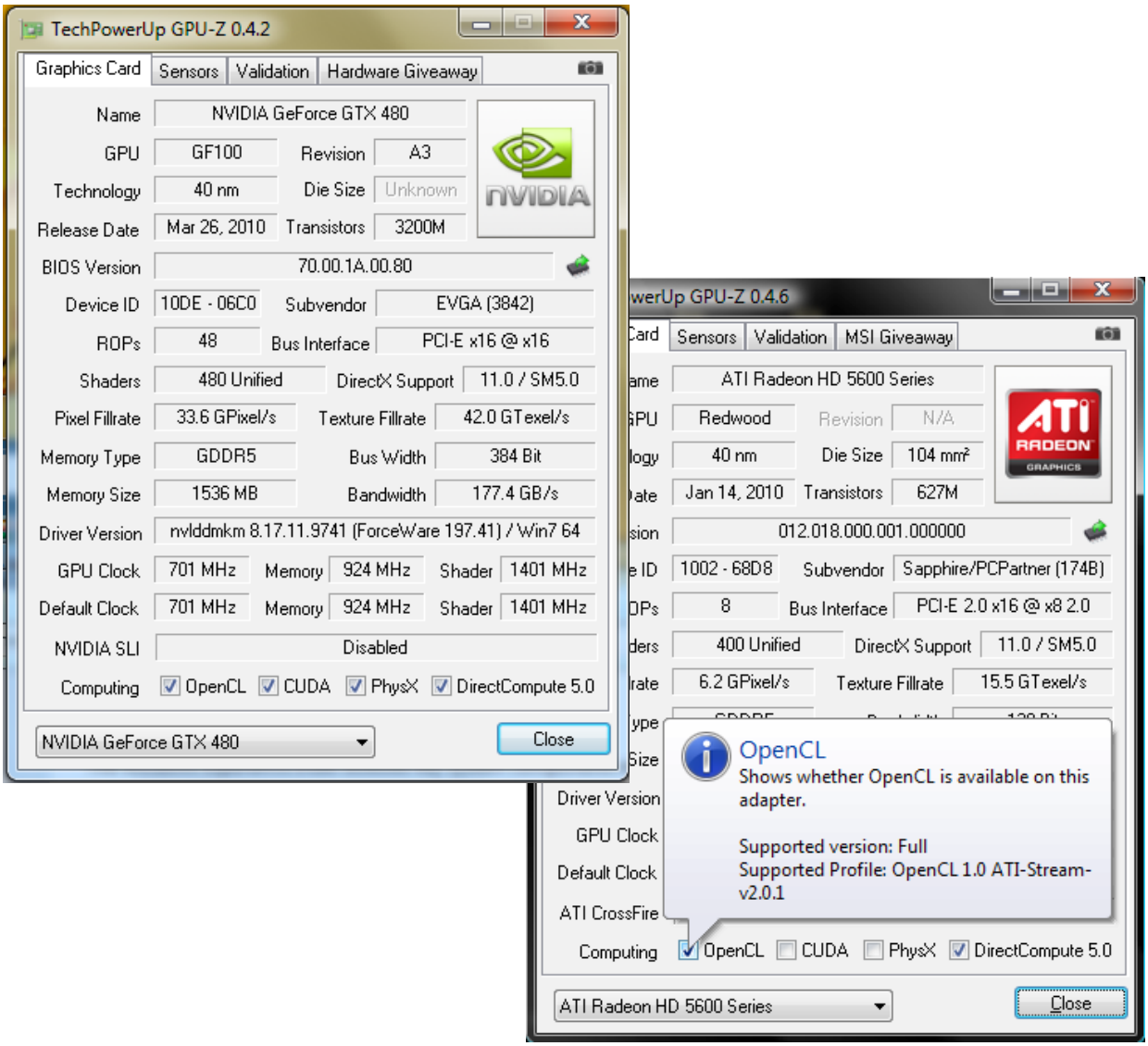

Figura 37: Exemples de GPUs actuals analitzades per GPU-Z

Per una altra banda tenim Geeks3D GPU Caps Viewer una eina enfocada a dona informació de tota mena sobre el suport i capacitat OpenGL/CUDA/OpenCL dels diferents processadors que conté la nostra màquina. Inclou links als últims drivers, diferents benchmarks per CPU i GPU i molta informació addicional.

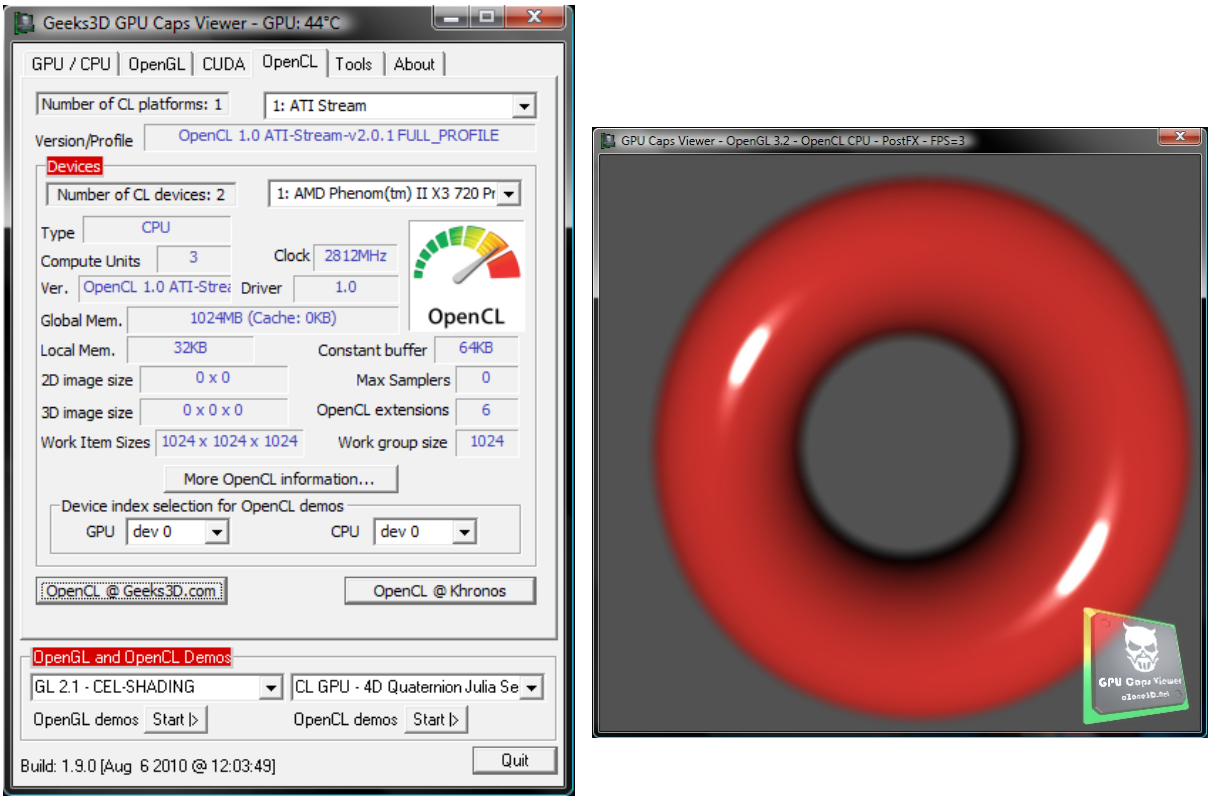

Figura 38: Captura on es mostra la pestanya d'OpenCL i una de les demos funcionant a la CPU