Universitat Oberta de Catalunya

TFC - Arquitectura de Computadors i Sistemes Operatius

# Títol: Paral·lelització de la factorització LU utilitzant OpenCL

Consultor: Francesc Guim Bernat
Alumne: Javier Vegas Caballero

Versió del document 1.4

Data: 23 de Gener de 2012

Índex

Índex de Figures

# 1. Introducció

Als darrers anys els processadors gràfics (GPUs) han experimentat un increment de la seva potencia computacional. Aquesta evolució els ha portat a nivells superiors als que es poden trobar a les unitats de processament central (CPUs).

Els processadors gràfics han evolucionat la seva arquitectura fins a convertir-se en unitats preparades per a l'execució d'aplicacions paral·leles de caràcter general, deixant enrere el seu us exclusió per el processament de primitives gràfiques.

Aquesta evolució ha propiciat l'aparició d'entorns de computació heterogenis a on conviuen arquitectures CPU multi-core, i GPUs.

Davant d'aquesta situació ha sigut necessària l'aparició d'un estàndard que possibilités l'explotació del conjunt de recursos computacionals d'aquests nous entorns. L'OpenCL és la resposta, un estàndard obert que defineix una API i un llenguatge de programació per a la implementació d'aplicatius paral·lels.

# 2. Motivació

A la meva vida laboral he treballat sovint en projectes de programació paral·lela en entorn HPC (*High Performance Computing*) utilitzant clústers de càlcul de tipus *Beowulf* i darrerament amb maquinaris multiprocessadors i multi nuclis (també multi fils). Per aquests projectes he utilitzat principalment llibreries com la MPI (*Message Passing Interface*) que ofereix un conjunt d'operacions per realitzar programació paral·lela amb pas de missatges, la PVM (*Parallel Virtual Machine*) que permet usar una xarxa de computadors com si fos un processador paral·lel distribuït i la openMP (*Open Multi-Processing*) que és un altre interfície de programació d'aplicacions que proporciona una programació multiprocés amb memòria compartida en múltiples plataformes.

# 3. Objectius

Els objectius del projecte són l'estudi de l'estàndard OpenCL (Open Computing Language) i la seva avaluació desenvolupant la paral·lelització d'una factorització LU.

# 4. Descripció de les tasques

Les tasques a realitzar són les següents:

- Cerca de treballs relacionats: en aquesta tasca es buscaran treballs relacionats.
- Documentació: en aquesta tasca es buscarà i llegirà tota la informació necessària per implementar l'aplicatiu.

- Instal·lació de l'entorn de treball: en aquesta tasca s'instal·larà l'entorn de programació Xcode 4.0 sobre el meu portàtil un MacBook Pro executant Mac OS X 10.6 i totes les llibreries que calguin per fer l'aplicació, incloses totes les eines de *debugging* i *profiling*, com gDEBugger , NVIDIA Visual Profiler i Shark.

- Disseny: en aquesta tasca es farà el disseny de l'aplicació. Es farà el disseny de l'aplicació tenint en compte el model de programació paral·lela basat en les dades.
- Implementació: en aquesta tasca es realitzarà la implementació en llenguatge C i en llenguatge OpenCL.
- Avaluació de la implementació: en aquesta tasca es provarà l'aplicació que s'haurà creat en la tasca anterior. L'API de OpenCL permet l'execució tant en la CPU com en la GPU i per tant possibilitaran realitzar el *benchmarking*.
- Millores de la implementació: en aquesta tasca s'utilitzarà la informació obtinguda en l'apartat anterior per tal de millorar el *speedup*.

## 5. Planificació de les tasques i estimació econòmica

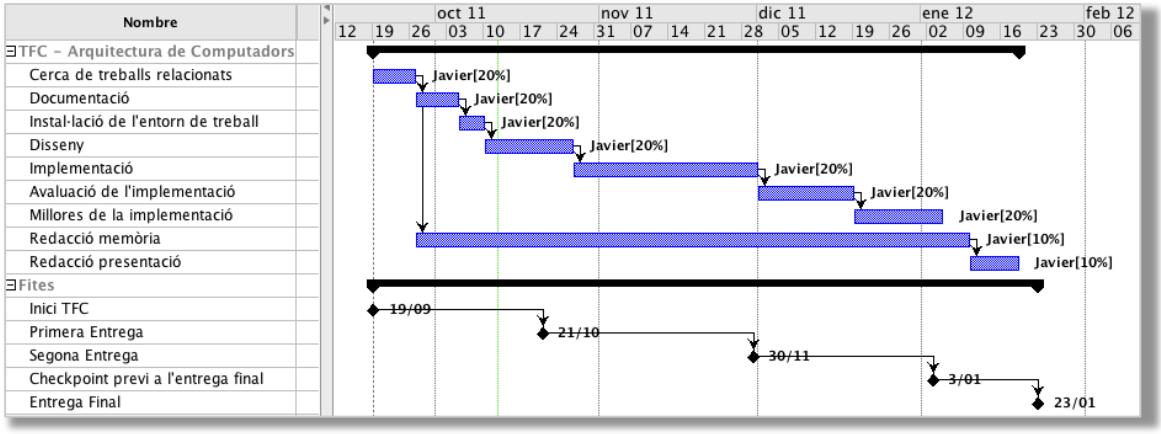
Per cada activitat, es defineixen el treball, la duració, la data d'inici, la data de finalització i el cost associat.

Les activitats a realitzar són les que es mostren a la taula següent:

| | | Nombre | Trabajo | Duración | Inicio | Terminado | Costo | Pred... | Nombres del Recurso |
|---|---|---|---|---|---|---|---|---|---|
| 1 | | **TFC - Arquitectura de Computado...** | **191 horas** | **81,75 d...** | **19/09/11 8:00** | **19/01/12 14:59** | **11459,98 €** | | |
| 2 | | Cerca de treballs relacionats | 10 horas | 6,25 days | 19/09/11 8:00 | 27/09/11 9:59 | 600,00 € | | Javier[20%] |
| 3 | | Documentació | 10 horas | 6,25 days | 27/09/11 9:59 | 5/10/11 11:59 | 600,00 € | 2 | Javier[20%] |
| 4 | | Instal·lació de l'entorn de treball | 5 horas | 3,125 d... | 5/10/11 11:59 | 10/10/11 13:59 | 300,00 € | 3 | Javier[20%] |
| 5 | | Disseny | 20 horas | 12,5 days | 10/10/11 13:59 | 27/10/11 8:59 | 1200,00 € | 4 | Javier[20%] |
| 6 | | Implementació | 40 horas | 25 days | 27/10/11 8:59 | 1/12/11 8:59 | 2400,00 € | 5 | Javier[20%] |
| 7 | | Avaluació de l'implementació | 20 horas | 12,5 days | 1/12/11 8:59 | 19/12/11 13:59 | 1200,00 € | 6 | Javier[20%] |
| 8 | | Millores de la implementació | 20 horas | 12,5 days | 19/12/11 13:59 | 5/01/12 8:59 | 1200,00 € | 7 | Javier[20%] |
| 9 | | Redacció memòria | 60 horas | 75 days | 27/09/11 9:59 | 10/01/12 9:59 | 3600,00 € | 2 | Javier[10%] |
| 10 | | Redacció presentació | 6 horas | 7,5 days | 10/01/12 9:59 | 19/01/12 14:59 | 360,00 € | 9 | Javier[10%] |
| 11 | | **Fites** | **0 horas** | **83 days** | **19/09/11 8:00** | **23/01/12 8:00** | **0,00 €** | | |
| 12 | | Inici TFC | 0 horas | 0 days | 19/09/11 8:00 | 19/09/11 8:00 | 0,00 € | | |
| 13 | | Primera Entrega | 0 horas | 0 days | 21/10/11 8:00 | 21/10/11 8:00 | 0,00 € | 12 | |
| 14 | | Segona Entrega | 0 horas | 0 days | 30/11/11 8:00 | 30/11/11 8:00 | 0,00 € | 13 | |
| 15 | | Checkpoint previ a l'entrega final | 0 horas | 0 days | 3/01/12 8:00 | 3/01/12 8:00 | 0,00 € | 14 | |
| 16 | | Entrega Final | 0 horas | 0 days | 23/01/12 8:00 | 23/01/12 8:00 | 0,00 € | 15 | |

He creat un calendari laboral tenint en compte la meva disponibilitat que és de 2.8 hores/dia però distribuiré la feina fent 2 hores/dia i el cap de setmana la part de redacció de la memòria 4 hores (0.8 x 5).

A continuació es mostra el diagrama de Gantt del projecte:




Respecte a la valoració econòmica he estimat un cost de 60 € per hora, en concepte de la meva retribució, incloent els costos fixes (consum energètic, cost ADSL).

El programari que utilitzaré és el que ja portava el meu portàtil Apple Macbook Pro, el sistema operatiu Mac OS X Snow Leopard amb el seu entorn de programació XCode 4, addicionalment utilitzaré Windows per això serà necessari les llicències de Windows 7 Professional i Visual Studio 2010 Professional que tampoc han suposat cap despesa ja que les he obtingut a través de UOC DotNetClub (programa Microsoft Academic Alliance facilitat per Microsoft Ibérica).

Com que el projecte té una durada de 191 hores, el cost final serà de 191 x 60 € = 11460 €.

# 6. OpenCL

## 6.1. Introducció

L'*Open Computing Language* (OpenCL) és un estàndard obert per a la programació d'entorns heterogenis on coexisteixen diferents recursos de computació: CPUs i GPUs.

L'objectiu de OpenCL és habilitar la creació d'aplicatius de propòsit general capaços d'utilitzar l'estàndard per explotar de forma eficient tots els recursos computacionals d'un mateix sistema. Per aconseguir-ho, OpenCL va més enllà de la definició d'un llenguatge i ofereix un complet *framework* així com el seu propi *runtime* per al desenvolupament de programari.

OpenCL es defineix de forma abstracta respecte al maquinari, és responsabilitat de cada fabricant proporcionar una implementació fidel al estàndard per fer que el seu maquinari sigui accessible des de l'OpenCL. Aquesta característica permet que l'elaboració d'aplicatius OpenCL sigui completament independent del maquinari que les executa, amb això s'aconsegueix un alt grau de portabilitat al codi desenvolupat.

## 6.2. Historia

OpenCL va donar els seus primer passos gràcies a l'empresa Apple que va crear l'especificació inicial. Posteriorment i juntament amb d'altres equips tècnics d'empreses com AMD, Intel i Nvidia es va elaborar la proposta inicial que va presentar per a la seva estandardització al Khronos Group.

El khronos Group va ser fundat al Gener de 2000 per companyies com Intel, Nvidia, AMD i Sun Microsystems (ara Oracle) entre d'altres, i la seva funció es la de crear estàndards oberts i lliures de royalties a l'àmbit de la computació paral·lela, la computació gràfica i multimèdia.

Un cop feta publica l'especificació inicial, els fabricants de maquinari iniciaren el desenvolupament d'implementacions que permetessin utilitzar els seus productes sota OpenCL. Per assegurar la qualitat de les implementacions, el Khronos Group va publicar tests d'avaluació que permetessin comprovar el funcionament de les implementacions.

Durant la segona meitat de l'any 2009 empreses com Intel, Nvidia i IBM va publicar els controladors de l'OpenCL 1.0.

Actualment el Khronos Group ha fet publica l'especificació de la versió 1.1 de OpenCL en la que s'incorporen noves funcionalitats a l'estàndard.

## 6.3. L'Arquitectura OpenCL

L'arquitectura d'OpenCL es mostra a la figura següent, en la que es poden trobar tots els elements que intervenen en l'execució d'aplicatius OpenCL distribuïts en capes ordenades del més alt al més baix nivell.

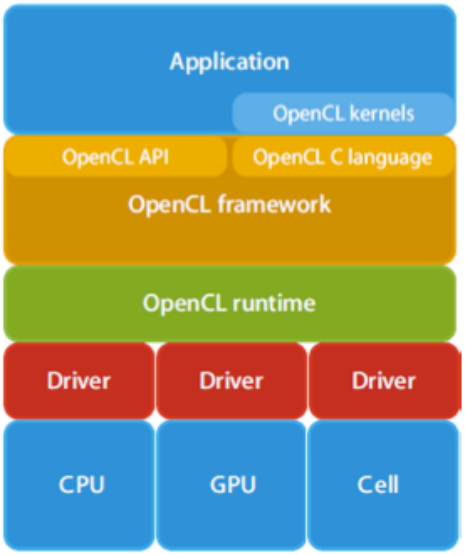



Figura 1: Arquitectura OpenCL

A continuació es realitzarà una descripció de cadascuna de les capes de l'arquitectura de l'OpenCL.

### 6.3.1. L'aplicatiu OpenCL

Una aplicació OpenCL es la que utilitza durant la seva execució les funcionalitats especifiques de l'OpenCL per portar a terme les tasques computacionals en les unitats de computació disponibles al sistema on s'executa.

Aquestes tasques computacionals es poden considerar com a nous subprogrames tals que la seva vida i execució queda lligada específicament al dispositiu al que han sigut encarregats. Aquests subprogrames reben el nom de kernels a l'àmbit de l'OpenCL.

La utilització de kernels fa que l'entorn d'execució d'una aplicació OpenCL no estigui composada únicament pel generat per l'aplicació inicial, sinó que també cal afegir els entorns d'execució que es generen als dispositius als quals s'executen els kernels. Aquest nous entorns d'execució són gestionats des de l'aplicació inicial mitjançant mecanismes proporcionats per l'OpenCL.

### 6.3.2. El *Framework* OpenCL

El framework d'OpenCL és l'encarregat de proporcionar els recursos necessaris per el desenvolupament d'aplicacions OpenCL. Esta format per l'API OpenCL, el llenguatge OpenCL C i el compilador OpenCL.

L'API OpenCL és un conjunt de funcions que permet controlar des de l'aplicació inicial l'execució de les tasques computacionals sobre els dispositius disponibles. Mitjançant les crides a l'API s'aconsegueix governar tot el procés d'execució, des d'obtenir inicialment els dispositius disponibles, fins a preparar l'entorn, executar el kernel i recuperar els resultats obtinguts. L'API també inclou mètodes per a la sincronització de kernels així com els mètodes per a obtenir informació del *profiling*.

El llenguatge OpenCL C és el llenguatge de programació utilitzat per implementar els kernels. Esta basat al llenguatge C99 amb els canvis necessaris per a la programació d'aplicacions sota el model de l'OpenCL.

El compilador OpenCL s'encarrega de processar els kernels implementats en OpenCL C i elaborar els binaris executables per als dispositius.

### 6.3.3. El *runtime* OpenCL

El *runtime* d'OpenCL s'encarrega d'executar totes les tasques que s'envien a través de la crida a les funcions de l'API d'OpenCL. El *runtime* intenta portar a terme l'execució aprofitant els recursos disponibles de la forma més eficient possible.

### 6.3.4. Els drivers OpenCL

El darrer nivell de l'execució de les aplicacions OpenCL trobem els dispositius a on s'executen els kernels. Aquests dispositius son controlats mitjançant els divers proporcionats per els respectius fabricants. Per a que el dispositiu sigui accessible des de l'OpenCL, el driver ha de suportar-ho proporcionant una implementació de l'estàndard seguint l'especificació oficial de OpenCL.

## 6.4. Els models OpenCL

Una vegada descrits tots el elements que formen l'arquitectura, per aconseguir explotar els conceptes clau a l'entorn del desenvolupament d'aplicacions OpenCL es seguirà el mateix esquema que apareix a l'especificació oficial ja que es considera el més adequat. Aquest esquema consisteix en classificar en models tots el elements que defineixen l'OpenCL. Els models són els següents:

- Model de plataforma. Descriu la visió global que utilitza OpenCL per a esquematitzar les plataformes computacions heterogènies sobre las que opera.
- Model d'execució. Descriu com es porta a terme l'execució dels kernels als dispositius OpenCL, així com els elements que cal definir per la mateixa.

- Model de memòria. Descriu la jerarquia de memòria utilitzada a l'execució dels kernels.
- Models de programació. Descriu les tècniques de programació que suporta OpenCL per a la implementació dels kernels.

### 6.4.1. Model de plataforma

L'esquema que defineix OpenCL com model conceptual per a la plataforma sobre la que s'executen les aplicacions està format per un host connectat a un o varis dispositius que suporten OpenCL.

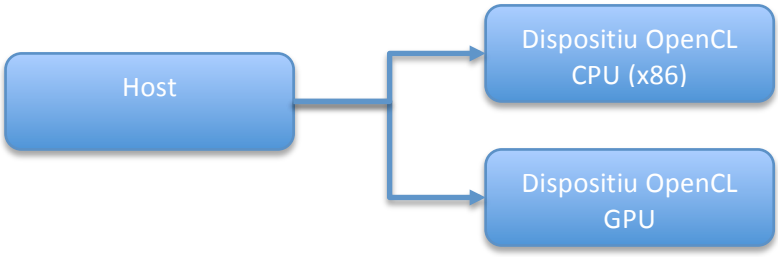



Figura 2: Interacció host - dispositiu

El host és l'encarregat d'iniciar l'execució de l'aplicació OpenCL. La comunicació entre el host i els dispositius als que es realitza mitjançant el enviament de comandes a través dels quals s'aconsegueix executar les tasques de preparació, execució i obtenció del resultats dels kernels.

OpenCL defineix a més una esquematització dels dispositius, de tal forma que els dispositius estan formats per unitats de computació que alhora es divideixen en elements de processament. Aquests darrers són els encarregats de portar a terme l'execució dels kernels.

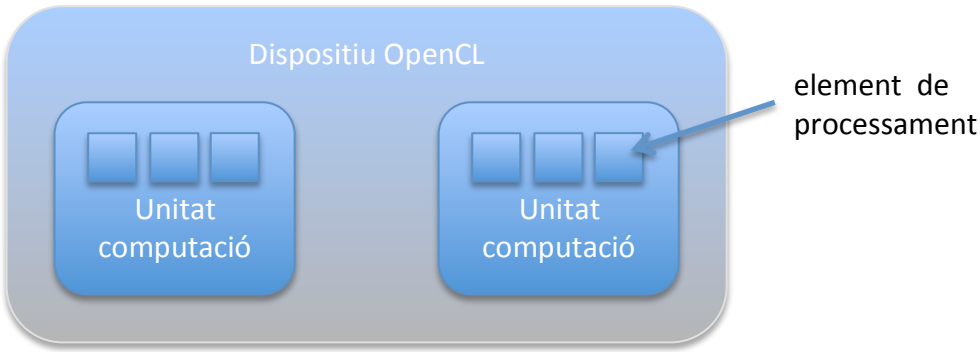



Figura 3: Estructura d'un dispositiu OpenCL

### 6.4.2. Model d'execució

L'execució de les aplicacions OpenCL es realitza en dues parts. Per un costat existeix una aplicació inicial que s'executa al host. Aquesta s'encarrega d'inicialitzar tots els elements necessaris per establir la comunicació amb els dispositius OpenCL, així com realitzar l'enviament de les tasques als dispositius. Per altre banda, es realitza la execució, ordenada des del host, dels kernels sobre els dispositius als que es tenen accés.

#### 6.4.2.1. Elements de l'OpenCL (host)

Per a l'execució de kernels OpenCL cal inicialitzar una sèrie d'estructures que permetin l'enviament de comandes cap als dispositius accessibles. Aquesta inicialització es realitza mitjançant crides a l'API de OpenCL. Els elements necessaris són el següents:

- El context OpenCL

A l'inici de l'aplicació i mitjançant l'OpenCL només es té accés a la informació relativa als dispositius connectats al sistema. La informació dels dispositius es distribueix en plataformes. Una plataforma es correspon amb una implementació d'OpenCL i inclou tots els dispositius que caldran ser accedits a través d'aquesta.

Un context OpenCL es crea sobre una plataforma i conté un subconjunt dels dispositius disponibles. Totes les crides posteriors es realitzaran sobre el context creat i solament tindran accés els dispositius que el context inclogui.

Una mateixa aplicació pot tenir varis contextos possibilitant l'execució de kernels OpenCL en dispositius de diferents plataformes.

- Les cues de comandes

A l'OpenCL tota interacció entre el host i els dispositius es realitza mitjançant comandes. Per iniciar l'enviament als dispositius inclosos a la definició d'un context cal crear una cua de comandes per a cada dispositiu al qual es vulguin enviar.

Una cua de comandes pot rebre els següents tipus de comandes:

- Comandes d'execució de kernel. S'utilitzen per iniciar l'execució d'un kernel al dispositiu associat a la cua de comandes.
- Comandes de memòria. S'utilitzen per a interactuar amb la memòria del dispositiu associat a la cua de comandes.
- Comandes de sincronització. S'utilitzen per controlar l'ordre d'execució de les comandes.

L'execució de comandes als dispositius es realitza de forma asíncrona respecte a l'aplicació host. Cada vegada que s'enviï una comanda es genera una estructura

d'event que permet a l'aplicació host conèixer l'estat d'execució de la comanda enviada.

Internament les cues de comandes es poden configurar per a controlar el comportament de les comandes una vegada han estat enviades. Existeixen dues configuracions:

**Execució en ordre**. Totes les comandes s'executen al mateix ordre en el que es van rebre.

**Execució fora d'ordre**. Totes les comandes inicien la seva execució, si es possible, al arribar a la cua de comandes. Pot succeir que quan una comanda enviada més tard finalitzi abans que una altre que va ser enviada abans.

- Els kernels

Encara que els kernels OpenCL van lligats als dispositius que els executen, és a l'aplicació host a on han de ser creats prèviament a ser enviats a la seva execució. Els kernels es poden crear de dues formes:

Mitjançant el codi font. L'aplicació Host llegirà el codi font del kernel OpenCL. Una vegada llegit invocarà al compilador OpenCL per tal de generar un executable binari que podrà ser enviat per a la seva execució al dispositiu.

Mitjançant un binari. Quan es compila el codi font d'un kernel, OpenCL dona la possibilitat de emmagatzemar al disc el binari generat pel compilador. Aquest binari pot ser recuperat posteriorment evitant haver de recompilar de nou el codi font.

A més a més del kernels escrits en OpenCL C, OpenCL defineix el que denomina com kernels natius. Un kernel natiu és una funció definida al codi de l'aplicació de host. La possibilitat d'utilitzar aquest tipus de kernels queda restringida solament a aquells dispositius que ho suportin.

#### 6.4.2.2. Procés d'execució d'un kernel

La forma en que es porta a terme l'execució d'un kernel és molt important per desenvolupar aplicacions OpenCL.

Quan un kernel s'executa es defineix un espai indexat de N dimensions a on N pot ser 1, 2 o 3. Per a cada punt d'aquest espai es crea una instancia del kernel a executar. Aquestes instancies es denominen com work-items. Cadascun s'executarà de forma independent.

Dins de l'espai indexat, els work-items es poden agrupar en grups que reben el nom de work-groups. La utilització de work-groups permet regular la granularitat amb la que es divideix l'espai indexat.

Des de el codi d'un kernel es poden identificar la posició dins de l'espai indexat que ocupa el work-item que s'està executant. Per això als work-items s'assignen dos tipus d'identificadors:

**Identificador global**. Identifica de forma absoluta la posició del work-item. Té un valor per a cada dimensió de l'espai indexat.

**Identificador local**. Identifica de forma absoluta la posició del work-item dins del work-group al que pertany. Per obtenir la posició absoluta dins de l'espai indexat a cada work-group se'ls assigna també un identificador. Per calcular la posició global ($g_x$,$g_y$), per exemple en dos dimensions, es pot fer:

$$g_x = w_x \cdot L_x + l_x \qquad g_y = w_y \cdot L_y + l_y$$

a on ($w_x$,$w_y$) és l'identificador al work-group, ($l_x$,$l_y$) és l'identificador local i $L_x$,$L_y$ són les dimensions del work-group.

### 6.4.3. Model de la memòria

En una aplicació OpenCL es defineixen dos espais de memòria diferents:

Espai de memòria de l'aplicació. És la memòria que habitualment utilitzen les aplicacions durant la seva execució utilitzin o no les funcionalitats de l'OpenCL.

Espai de memòria de dispositiu. És la memòria a la que té accés un work-item durant l'execució d'un kernel. Aquest espai es troba definit al dispositiu a on s'està executant.

L'espai de memòria disponible al dispositiu per que un work-item realitzi l'execució d'un kernel es presenta a la figura que es mostra a continuació:

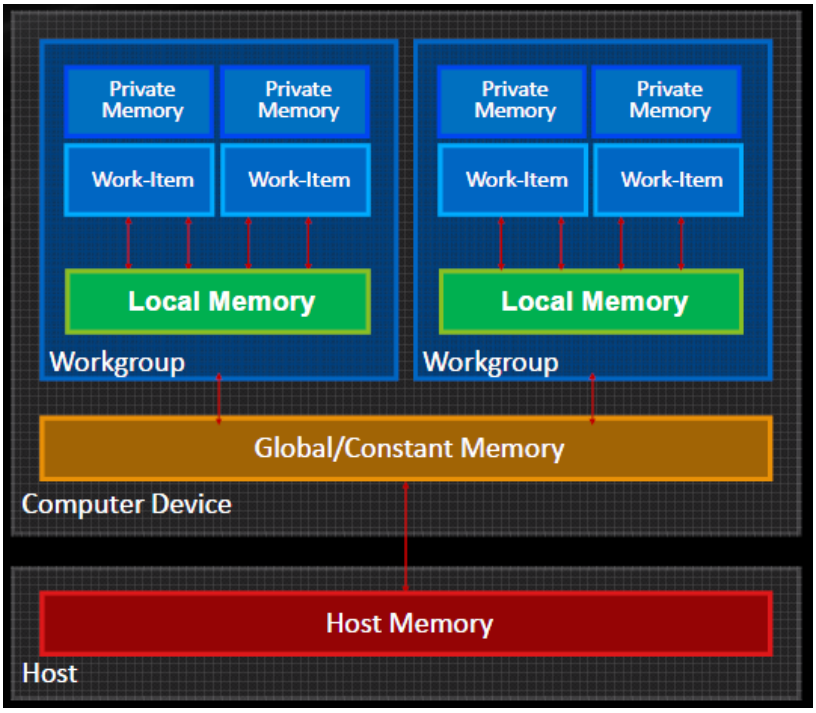



Figura 4: Model de memòria

Tal com es mostra a la figura, es distingeixen quatre regions de memòria:

**Memòria Global**. Regió de memòria que permet lectures i escriptures realitzades per tots els work-items en execució. En aquells dispositius que ho suportin, els accessos a la memòria global poden ser accelerats utilitzant tècniques de memòria cau.

**Memòria Constant**. Regió de la memòria global que roman constant durant l'execució del kernel.

**Memòria Local**. Regió de memòria que permet lectures i escriptures pels work-items que comparteixen un determinat work-group. Per tant, work-items de diferents work-groups no tenen accés a la mateixa memòria local. La memòria local pot ser implementada en els dispositius com una regió específica de la memòria global o bé utilitzar regions de memòria dedicades exclusivament a la seva utilització com a memòria local.

**Memòria privada**. Regió de memòria exclusiva pels work-items. Tot el definit en memòria privada per un work-item és accessible únicament per ell mateix.

La utilització de la memòria de dispositiu es realitza de forma explícita des del codi de l'aplicació. Els objectius que persegueix són els següents:

**Reserva**. Reservar l'espai de memòria necessari per a les dades amb els quals el kernel necessitarà treballar durant la seva execució.

**Inicialització**. Inicialitzar l'espai de memòria reservat amb els valors d'entrada amb els quals es vol executar el kernel. Habitualment aquesta inicialització es realitza mitjançant la transferència de dades entre l'espai de memòria de l'aplicació i l'espai de memòria del dispositiu.

**Recuperació**. Recuperar les dades de sortida que el kernel ha elaborat durant la seva execució. La recuperació es realitza mitjançant la transferència de dades entre l'espai de memòria del dispositiu i l'espai de memòria de l'aplicació.

La API d'OpenCL proporciona crides específiques per al compliment dels objectius descrits anteriorment. En concret proporciona dues modalitats de treball sobre la memòria de dispositiu:

**Modalitat de transferència**. Mitjançant comandes OpenCL s'executen transferències de blocs de memòria entre l'espai de memòria de l'aplicació i l'espai de memòria de dispositiu i viceversa.

**Modalitat de mapejat**. Mitjançant comandes OpenCL es mapeja la memòria de dispositiu sobre una regió de la memòria de l'aplicació. A partir d'aquest moment l'aplicació realitza totes les operacions d'inicialització i recuperació sobre la regió mapejada. Una vegada finalitzades totes les operacions i també mitjançant comandos OpenCL, es desfà el mapatge permetent que els canvis arribin a la memòria de dispositiu.

### 6.4.4. Model de programació

Tenint en compte el model d'execució que segueixen els kernels, OpenCL defineix dos models de programació:

**Data parallel**. El model de programació data parallel consisteix a executar una seqüència d'instruccions sobre els diferents elements que formen un bloc de dades. El model d'execució per work-items dels kernels encaixa perfectament amb aquest model ja que es pot fer correspondre a cada work-item amb cada bloc de dades sobre els quals es vol operar. Encara així, OpenCL no requereix que es realitzi una correspondència un a un entre els work-items i els elements del bloc de dades tenint el desenvolupador la possibilitat de fer la correspondència com ho desitgi.

**Task parallel**. El model task parallel consisteix a executar una sola instància dels kernels de forma independent a l'espai indexat que es defineixi. És a dir, els dispositius es converteixen en processadors paral·lels de tasques independents a nivell de dispositiu.

## 6.5. El llenguatge C OpenCL

El llenguatge OpenCL C és el llenguatge definit per OpenCL per a l'escriptura dels kernels que s'executen en els dispositius OpenCL. Està format per un subconjunt del llenguatge C ISO/IEC 9899:1999 també conegut com C99. A aquesta base se li han afegit diverses extensions per adaptar-ho a l'ús que se li dóna en OpenCL.

Com a principals característiques del OpenCL C destaquen:

**Identificador d'entrada de kernel**. En el codi d'un programa OpenCL poden existir diverses rutines. Per saber quin és la que s'executarà com a punt de partida s'utilitza l'etiqueta _kernel que es col·loca just abans de la definició de la capçalera. Dins d'un mateix arxiu de codi font poden existir diverses capçaleres marcades per l'etiqueta _kernel. És el host qui haurà d'especificar el nom de la funció entre les marcades que actuarà com a punt d'entrada del programa quan crea el kernel mitjançant les crides a la API.

**Recursivitat no permesa**. La recursivitat dins dels programes OpenCL no està permesa.

**Vectors de tipus bàsics predefinits.** OpenCL C defineix vectors de tipus bàsics predefinits per ser usats com a tipus directament. Així doncs, per exemple, si volem utilitzar en el nostre codi un vector de 4 elements de tipus float OpenCL predefineix el tipus float4 per ser utilitzat directament sense necessitat de declarar un array de tipus float amb 4 posicions. L'exemple es pot aplicar a la resta de tipus bàsics i amb dimensions potencia de 2 fins al màxim que suporti el dispositiu.

**Aritmètica de tipus vector**. A part de la definició de tipus vector, OpenCL també defineix el comportament d'aquests tipus amb els operadors bàsics unaris i binaris. El comportament està definit tant entre elements de tipus vector, que habitualment es

resol executant l'operador component a component, com també entre tipus vector i tipus bàsic que es resol executant l'operació components a component amb el valor de l'operand de tipus bàsic.

**Identificadors d'espai de memòria**. OpenCL dóna la possibilitat de controlar on es declaren les variables o sobre quin espai de memòria apunten els punters que es declaren mitjançant identificadors d'espai de memòria. Existeixen quatre identificadors i són: _global, _constant, _local, _private.

**Funcions predefinides**. OpenCL C té predefinides una sèries de funcions que permeten facilitar les següents tasques:

- Funcions per work-items. Donen informació sobre els identificadors del work-item actual, tant l'identificador global com el local, així com també serveixen per informar del nombre total de grups i work-items en execució.
- Funcions matemàtiques. Es correspondrien a les funcions que podríem trobar en el Math.h de la llibreria estàndard de C. Trobem des de funcions cosinus, sinus, fins a potències i arrels quadrades passant per funcions d'arrodoniment.
- Funcions geomètriques. Permeten executar operacions sobre els tipus de vectors. Trobem funcions per normalitzar un vector, per realitzar productes vectorials i escalars, etc.
- Funcions comunes i relacionals. Trobem funcions per al pas entre graus i radiants així com també funcions que permeten executar comparacions entre valors.

# 7. Factorització LU

## 7.1. Descripció del problema

La factorització LU és un mètode per descompondre una matriu A (no singular ) en dos matrius una triangular inferior L i una altre triangular superior U .

$$A = L \cdot U$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \cdot \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

La factorització LU s'utilitza per resoldre sistemes d'equacions de forma eficient i per trobar la matriu inversa.

## 7.2. Resolució matemàtica

Per a la resolució he escollit un procediment directe d'eliminació Gaussiana sense pivot (vàlid únicament per a matrius denses).

Per demostrar el procediment partirem del següent:

Fem que $A_1$=A i $A_1$=L·U,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & & & & \\ a_{31} & & A' & & \\ \vdots & & & & \\ a_{n1} & & & & \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ l_{21} & & & & \\ l_{31} & & L_2 & & \\ \vdots & & & & \\ l_{n1} & & & & \end{bmatrix} \cdot \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ 0 & & & & \\ 0 & & U_2 & & \\ \vdots & & & & \\ 0 & & & & \end{bmatrix}$$

La matriu A' té dimensió n-1 x n-1 resultat d'eliminar la primer columna i la primera fila de la matriu $A_1$. Operant amb les submatrius podem obtenir les següents igualtats:

$(1)\ a_{11} = u_{11}$

$(2)[a_{12} \quad a_{13} \quad \dots \quad a_{1n}] = [u_{12} \quad u_{13} \quad \dots \quad u_{1n}]$

$(3)[a_{21} \quad a_{31} \quad \dots \quad a_{n1}]^T = u_{11}[l_{21} \quad l_{31} \quad \dots \quad l_{n1}]^T$

$$(4)\ A' = \begin{bmatrix} l_{21} \\ l_{31} \\ \vdots \\ l_{n1} \end{bmatrix} [u_{12} \quad u_{13} \quad \dots \quad u_{1n}] + L_2U_2$$

Resolvent tenim,

$u_{11} = a_{11}$

$l_{i1} = {a_{i1}}/{u_{11}}$, i=2,...,n

$u_{1j} = a_{1j}$ , j=2,...,n

Així doncs completem una iteració fent,

$$A_2 = L_2U_2 = A' - \begin{bmatrix} l_{21} \\ l_{31} \\ \vdots \\ l_{n1} \end{bmatrix} [u_{12} \quad u_{13} \quad \dots \quad u_{1n}]$$

$a_{ij}^{(2)} = a_{ij}^{(1)} - l_{i1}u_{1j}$, i,j=2,...,n

de forma anàloga, per a la iteració k,

$a_{ij}^{(2)} = a_{ij}^{(1)} - l_{ik}u_{kj}$, k=1,2,...,n-1

# 8. Implementació

## 8.1. Versió seqüencial (CPU)

La implementació per a la CPU s'obté de la traducció de l'algorisme principal al llenguatge C. Aquesta versió servirà com a referencia respecte a la versió executada sobre la GPU.

A continuació es mostra un extracte de la implementació realitzada, a l'apèndix 1 es pot consultar la resta del codi.

```
⋮

/* Per a cada iteració i */
for(i=0;i<n;i++)
{
        /* Assignem a Lii=1 */
        L[i*n+i]=1.0f;

        /* Per a cada fila j de la matriu A */
        for(j=i+1;j<n;j++)
        {
                L[j*n+i]=U[j*n+i]/U[i*n+i];

                /* Per a cada columna k de la matriu A */
                for(k=i;k<n;k++)
                        U[j*n+k]-=(L[j*n+i]*U[i*n+k]);
        }
}

⋮
```

Comentaris:

- La implementació fa servir una indexació de matrius com a vectors unidimensionals de forma que cada element de la matriu s'accedeix fent el càlcul de l'index següent:
  A(i,j)=A[i*n+j] on n és la dimensió de la matriu (quadrada).
- Per el correcte funcionament de la implementació cal que la matriu U s'inicialitzi amb els elements de la matriu A.

## 8.2. Versió paral·lela (GPU)

### 8.2.1. Estructura d'una aplicació OpenCL

L'aplicació s'inicia obtenint la informació de les plataformes disponibles en el sistema on s'executa. De les plataformes obtingudes se selecciona la primera disponible i es crea un context sobre ella.

Per crear el context, s'especifica en les seves propietats l'identificador de la plataforma sobre la qual es vol crear. També s'especifica que volem incloure en el nostre context tots els dispositius de la plataforma de tipus GPU.

```
⋮

    /* Get Platform/Device Information */
    ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
    if (ret != CL_SUCCESS)
    {
        printf("Error: Failed to get device info!\n");
        return EXIT_FAILURE;
    }
    /* Connect to a GPU compute device */
    ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id,
                    &ret_num_devices);
    if (ret != CL_SUCCESS)
    {
        printf("Error: Failed to create a device group!\n");
        return EXIT_FAILURE;
    }

    /* Create OpenCL Context */
    context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);
    if (!context)
    {
        printf("Error: Failed to create a compute context!\n");
        return EXIT_FAILURE;
    }


⋮
```

Una vegada obtingut el context d'execució s'ha de crear una cua de comandes sobre algun dels dispositius inclosos en ell. Demanem a OpenCL que construeixi una cua de comandes associada a un d'ells.

```
⋮

    /* Create command queue */
    command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
    if (!command_queue)
    {
        printf("Error: Failed to create a command queue!\n");
        return EXIT_FAILURE;
    }

⋮
```

Abans d'utilitzar la cua de comandes per preparar l'entorn d'execució del kernel, llegirem el seu codi font i ho compilarem per deixar-ho preparat per a la seva execució.

```
⋮

    /* Create kernel program from source file*/
    program1 = clCreateProgramWithSource(context, 1, (const char **)&source_str1,
                                         (const size_t *)&source_size1, &ret);
    if (!program1 || ret != CL_SUCCESS)
    {
        printf("Error: Failed to create compute program!\n");
        return EXIT_FAILURE;
    }

    /* Build the program executable */
    ret = clBuildProgram(program1, 1, &device_id, NULL, NULL, NULL);
    if (ret != CL_SUCCESS)
    {
        size_t len;
        char buffer[2048];

        printf("Error: Failed to build program executable!\n");
        clGetProgramBuildInfo(program1, device_id, CL_PROGRAM_BUILD_LOG,
                              sizeof(buffer), buffer, &len);
        printf("%s\n", buffer);
        return EXIT_FAILURE;
    }

    kernel1 = clCreateKernel(program1, KERNEL_NAME1, &ret);
    if (!kernel1 || ret != CL_SUCCESS)
    {
            printf("Error: Failed to create compute kernel!\n");
            return EXIT_FAILURE;
     }


⋮
```

La reserva de memòria es realitza mitjançant comandos per a la creació de buffers de memòria. Aquestes comandes, a més, permeten inicialitzar la regió de memòria del dispositiu a partir d'una regió de memòria del host. En el nostre cas inicialitzarem la memòria del dispositiu amb els vectors inicialitzats prèviament en la memòria de host.

Una vegada la memòria aquesta creada i inicialitzada correctament en el dispositiu, els espais reservats s'enllacen amb els paràmetres d'entrada del kernel.

```
    /* Memory allocation */
    A = (float *)malloc(ndim*ndim*sizeof(float));

    /* Create Input/Output Buffer Object */
    AUbuff = clCreateBuffer(context, CL_MEM_READ_WRITE, ndim*ndim*sizeof(float),
                            NULL, &ret);
    if (!AUbuff)
    {
        printf("Error: Failed to allocate destination array!\n");
        return EXIT_FAILURE;
    }

  /* Copy input/output data to the memory buffer */
    ret = clEnqueueWriteBuffer(command_queue, AUbuff, CL_TRUE, 0,
                            ndim*ndim*sizeof(float), A, 0, NULL, NULL);
    if (ret != CL_SUCCESS)
    {
        printf("Error: Failed to write to source array!\n");
        return EXIT_FAILURE;
    }
```

L'assignació dels paràmetres d'entrada del kernel es realitza mitjançant l'índex que ocupa en la definició de la capçalera del mètode indicat com a punt d'entrada.

A partir d'aquest moment l'entorn d'execució del kernel està llest, és el moment d'executar-ho. En enviar la comanda d'execució a la cua de comandes indiquem les dimensions de l'espai indexat amb les quals s'executarà el kernel.

```
    /* Set OpenCL kernel arguments */
    ret|= clSetKernelArg(kernel1, 0, sizeof(cl_mem), (void *)&AUbuff);
    ret|= clSetKernelArg(kernel1, 1, sizeof(cl_mem), (void *)&Lbuff);
    ret|= clSetKernelArg(kernel1, 2, sizeof(int), &ndim);
    ret|= clSetKernelArg(kernel1, 3, sizeof(int), &i);
    if (ret != CL_SUCCESS)
    {
        printf("Error: Failed to set kernel arguments!\n");
        return EXIT_FAILURE;
    }

    /* Execute OpenCL kernel */
    ret = clEnqueueNDRangeKernel(command_queue, kernel1, 1, NULL, global1,
                                 local1, 0, NULL, NULL);
    if (ret!= CL_SUCCESS)
    {
        printf("Error: Failed to execute kernel!\n");
        return EXIT_FAILURE;
    }


⋮
```

### 8.2.2. Versió inicial

El primer pas i possiblement el més complicat alhora de paral·lelitzar és detectar els punts susceptibles de ser processats en paral·lel.
L'algorisme de la factorització LU utilitzat a la versió seqüencial no és totalment paral·lelitzable ja que es tracta d'un procés iteratiu. Així doncs cada pas o iteració de l'algorisme utilitza valors de les matriu U d'iteracions anteriors.
Aquesta primera versió el que fa es cridar dins del bucle principal, que recorre els elements de la diagonal, a una primera funció per calcular el valor corresponents als elements d'una columna de la matriu triangular inferior L i a continuació cridar a una altre funció que s'encarrega de fer la reducció de les files sota l'element de la diagonal $A_{ii}$ utilitzant els elements de la columna de la matriu $L_{ji}$ prèviament calculats per obtenir les files corresponents de la matriu U. Aquests dues funcions s'implementen en OpenCL com a kernels, i per tant s'executen en paral·lel.

Llavors tenim que per a cada element de la diagonal de la matriu A, calculem seqüencialment els valors de L i després els utilitzem per calcular els valors de la matriu U. Això fa que no tingui un rendiment molt bo ja que no es processa en parallel el càlcul iteratiu de les matrius L i U.

A cada iteració la mida de la matriu és més petita i per tant és van reduint el nombre de work-items (threads) executats a cada kernel.

Els primer kernel implementat per calcular la matriu L és el següent:

**kernel 1**

**Fitxer: Pivot.cl**

```
__kernel void Pivot(__global float* AU,__global float* L, const int n, const int
i)
{

    int j=get_global_id(0)+i;

    if(j>n-1)
        return;

    L[j*n+i]=native_divide(AU[j*n+i],AU[i*n+i]);


}
```

Aquest primer kernel està indexat en una sola dimensió ja que a cada iteració i, només processarà els elements d'un vector columna de la matriu L.

El segon kernel utilitzat per calcular cada iteració de la matriu U és el següent:

**kernel 2**

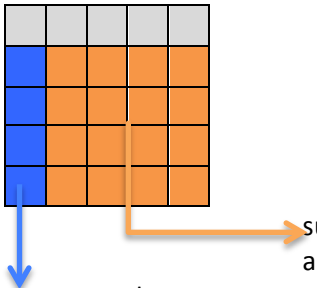
**Fitxer: ForwardElimination.cl**

```
__kernel void ForwardElimination(__global float* AU, __global float* L, const int
n, const int i)
{

    int j=get_global_id(0)+(i+1);
    int k=get_global_id(1)+i;

    if(j>n-1 || k>n-1)
        return;

    AU[j*n+k]-=(L[j*n+i]*AU[i*n+k]);
```

En aquest segon kernel l'indexat és bidimensional ja que es processen tots els elements de la submatriu de la matriu A al mateix temps.

El procés iteratiu es mostra amb un exemple on A es una matriu 5 x 5 en el següent diagrama:

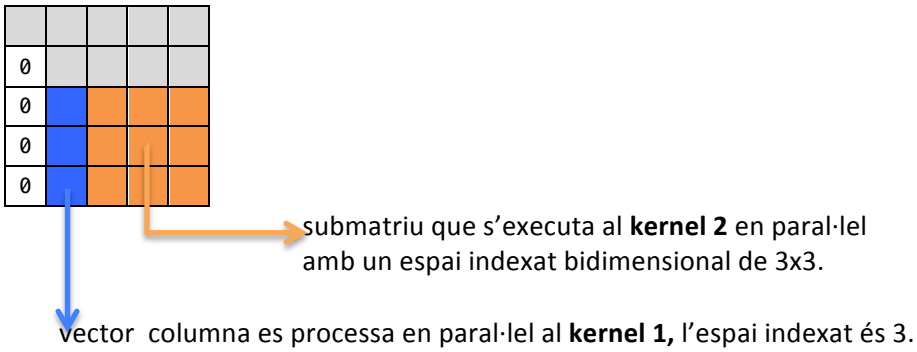
iteració i=0



submatriu que s'executa al **kernel 2** en paral·lel amb un espai indexat de 4x4.

vector columna es processa en paral·lel al **kernel 1,** l'espai indexat és 4.

iteració i=1




iteració i=2

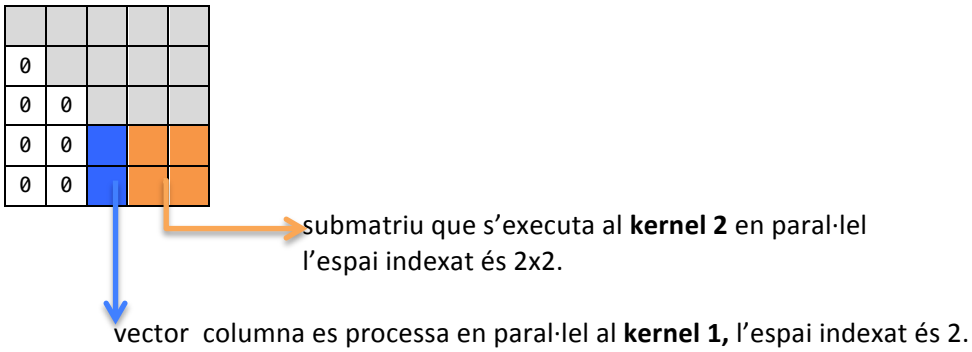
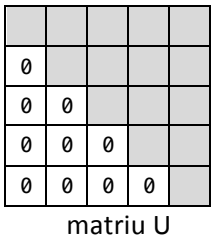
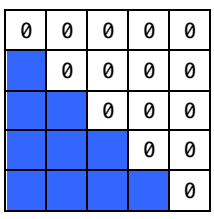



Figura 5. Diagrama del procés iteratiu

Aquesta darrera figura mostra les iteracions per obtenir la matriu U, el algorisme executarà les iteracions que calguin cridant als kernels 1 i 2 recorrent la diagonal obtenint zeros als elements per sota de la mateixa.
Finalment com a resultat de l'execució seqüencial dels dos kernels obtindrem la factorització de la matriu A.



matriu U



matriu L

Figura 6. Matrius resultat de la factorització

A l'apèndix 2 es pot consultar tota la implementació.

### 8.2.3. Versió millorada

Un punt clau per millorar el rendiment és minimitzar els accessos a la memòria global, per això a la versió millorada de l'algorisme, s'ha modificat per utilitzar-la.

Una altre punt important és reduir el nombre de kernels per augmentar la carrega dels work-items (threads), a la versió anterior, les operacions que executaven els work-items era molt petita i això penalitza el temps d'execució.

A la nova versió doncs és modifica el següent:

- En lloc d'executar dos kernels per iteració es passa a treballar amb nomes un.
- El nou kernel només treballa amb un espai d'indexat unidimensional, això vol dir que cada work-item processarà una fila sencera de la matriu.
- L'ús de memòria local.

El nou kernel implementat és el següent:

```
__kernel void ludecomposition(__global float* A, const int n, const int i,
__local float* Ai)
{

    int k;
    float ratio;

    int j=get_global_id(0);
    int jloc=get_local_id(0);
    int nloc=get_local_size(0);

    if(j>n-1)
        return;

    ratio=native_divide(A[j*n+i],A[i*n+i]);

    for(k=i+jloc;k<n;k=k+nloc)
        Ai[k]=A[i*n+k];

    barrier(CLK_LOCAL_MEM_FENCE);

    if(j<=i)
        return;

    for(k=i;k<n;k++)
        A[j*n+k]-=(ratio*Ai[k]);

    A[j*n+i]=ratio;

}
```

Respecte a l'ús de la memòria local té varis inconvenients el primer de tots és la seva limitació a 16Kbytes. Si una dada tipus float ocupa 4bytes tenim un màxim de 4096 elements que al meu cas correspondrien als elements d'una fila i per tant la matriu més gran que puc processar és de 4096 x 4096 elements. L'altre limitació de l'OpenCL és que només es pot sincronitzar la memòria local dels work-items dins un mateix work-group (utilitzant la comanda barrier).

L'estratègia seguida per treballar amb memòria local és la següent:

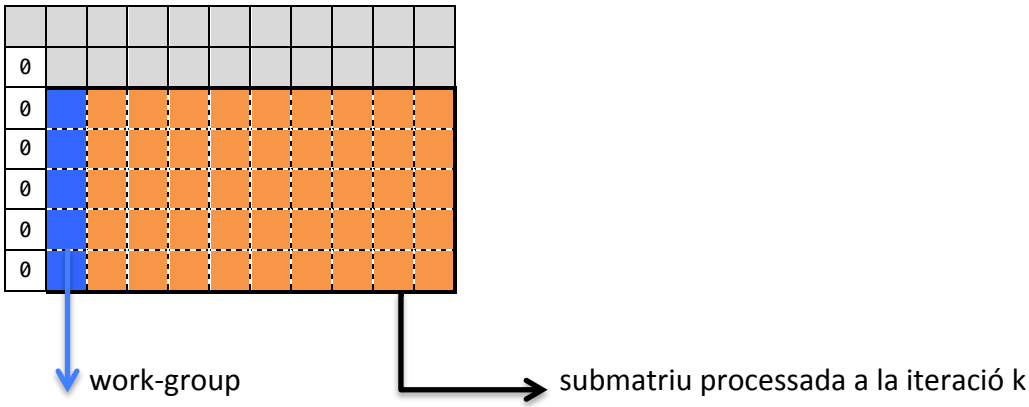



Figura 7. Iteració k

Cada work-item del work-group de mida 5 (5 files) omple 5 elements del vector Ai, així de forma col·laborativa tots el work-items treballen conjuntament per copiar els elements d'una fila al vector Ai que és el que utilitza memòria local.

Com a resultat obtindrem que la resta de càlculs es realitzaran utilitzant accessos a Ai d'aquesta forma eliminen accessos a memòria global i al seu lloc utilitzarem accessos a memòria local.

A l'apèndix 3 es pot consultar tota la implementació.

# 9. Benchmarking

## 9.1. Entorn de desenvolupament

El desenvolupament del codi l'he realitzat en un portàtil Apple MacBook Pro que disposa d'una targeta gràfica Nvidia 320M. El sistema operatiu és Mac OS X 10.6.8 que incorpora un driver que suporta la versió OpenCL 1.0.

Les característiques tècniques que OpenCL retorna de la targeta gràfica Nvidia 320M són les següents:

```
=== 1 OpenCL device(s) found on platform:
  DEVICE_NAME = GeForce 320M
  DEVICE_VENDOR = NVIDIA
  DEVICE_VERSION = OpenCL 1.0
  DRIVER_VERSION = CLH 1.0
  DEVICE_MAX_COMPUTE_UNITS = 6
  DEVICE_MAX_CLOCK_FREQUENCY = 950
```

```
DEVICE_GLOBAL_MEM_SIZE = 268435456
DEVICE_LOCAL_MEM_SIZE = 16384
CL_KERNEL_WORK_GROUP_SIZE = 512
```

L'entorn de desenvolupament utilitzat és el Xcode 4.0[1].

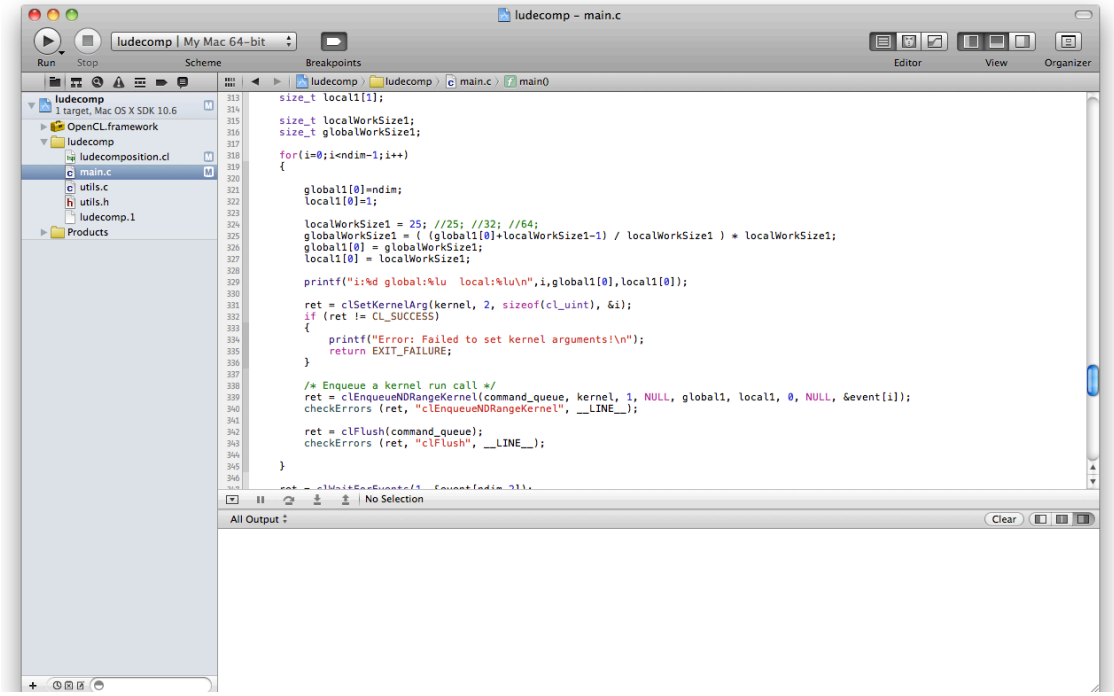



Figura 8. Captura de pantalla de l'entorn de desenvolupament

## 9.2. Entorn de proves

Les proves s'han realitzat a un servidor linux de la UOC, on he disposat de una targeta Tesla C1060 i una CPU Xeon E5504. El sistema operatiu del servidor és Fedora 13.

Les característiques tècniques de la targeta Nvidia Tesla C1060 són les següents:

```
=== 2 OpenCL device(s) found on platform:

  DEVICE_NAME = Tesla C1060
  DEVICE_VENDOR = NVIDIA Corporation
  DEVICE_VERSION = OpenCL 1.0 CUDA
  DRIVER_VERSION = 270.41.19
  DEVICE_MAX_COMPUTE_UNITS = 30
  DEVICE_MAX_CLOCK_FREQUENCY = 1296
  DEVICE_GLOBAL_MEM_SIZE = 4294770688
```

## 9.3. Resultats

Per obtenir el rendiment de les versions implementades s'ha utilitzat per mesurar el temps de càlcul diferents APIs. La versió de desenvolupament compilada en Mac OS X

[1] http://developer.apple.com/technologies/tools/features.html

utilitza la crida `mach_absolute_time()` que retorna el nombre de tics des de l'inici de sistema operatiu. Per altre banda per a la versió de proves i donat que disposava d'un servidor Linux la crida utilitzada ha estat `clock()`.

```
clock_t start = clock();
// execució OpenCL …
clock_t end = clock() ;
double elapsed_time = (end-start)/(double)CLOCKS_PER_SEC;
printf("Elapsed time: %lf seconds \n",elapsed_time);
```

També s'ha utilitzat les eines que proporciona OpenCL per fer mesurar els temps de càlcul fent servir la funció `clGetEventProfilingInfo`.

Els temps de càlculs mostrats al diagrama són el resultats de fer un promig de 10 execucions per a una mateixa mida de la matriu. De totes formes els temps no han variat gaire a les repeticions de les execucions.

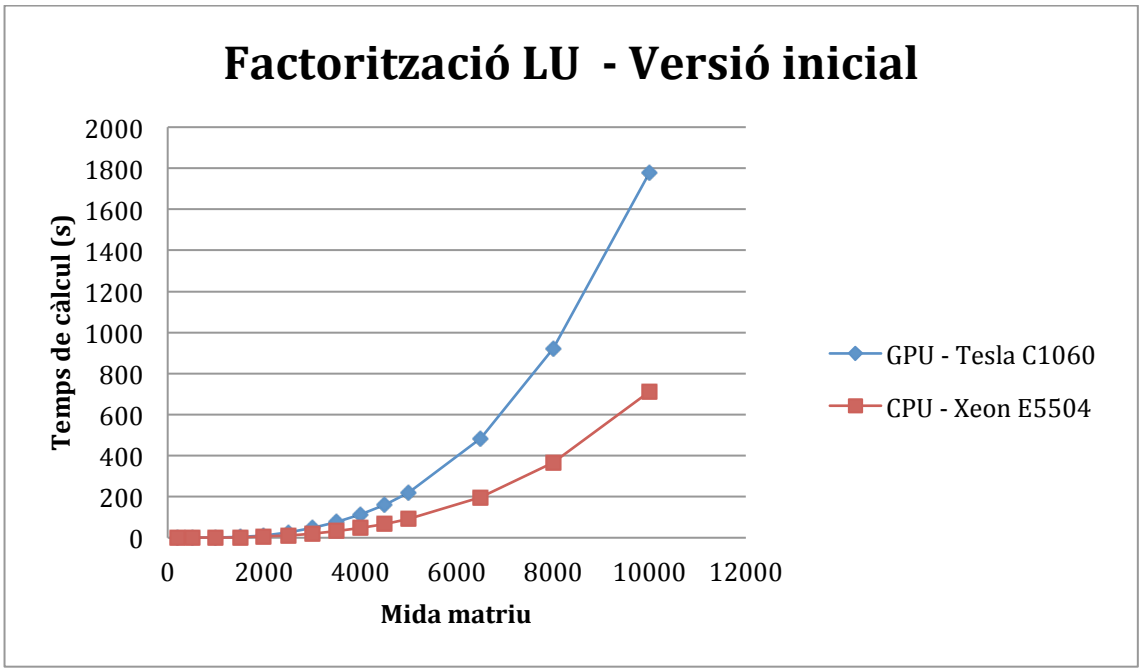



Figura 9. Temps d'execució GPU vs CPU

La figura anterior mostra els resultats dels temps d'execució de la versió inicial de l'algorisme i la seva relació amb la mida de la matriu, en referencia amb la versió de la CPU.

A continuació es mostra el diagrama amb els resultats dels temps d'execució de la versió millorada de l'algorisme i la seva relació amb la mida de la matriu.

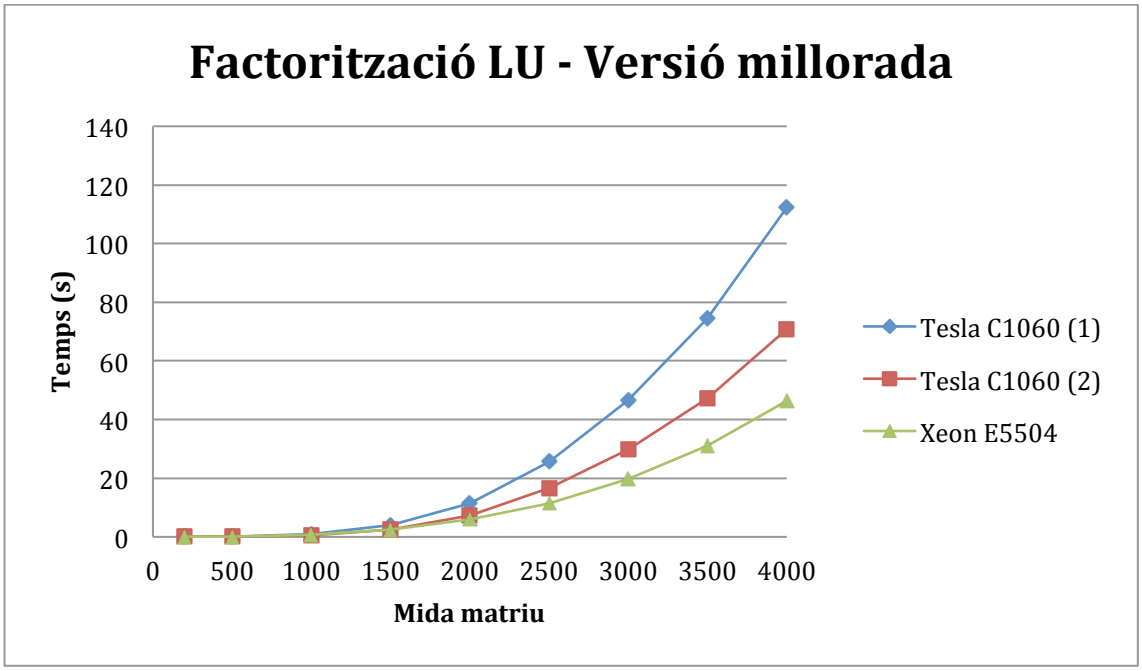



Figura 10. Temps d'execució GPU vs CPU (versió millorada)

La versió (1) correspon amb la versió inicial i la (2) amb la versió millorada.
Al gràfic de la Figura 11 es mostra el resultat del temps d'execució en percentatge respecte a la versió de referència (CPU).

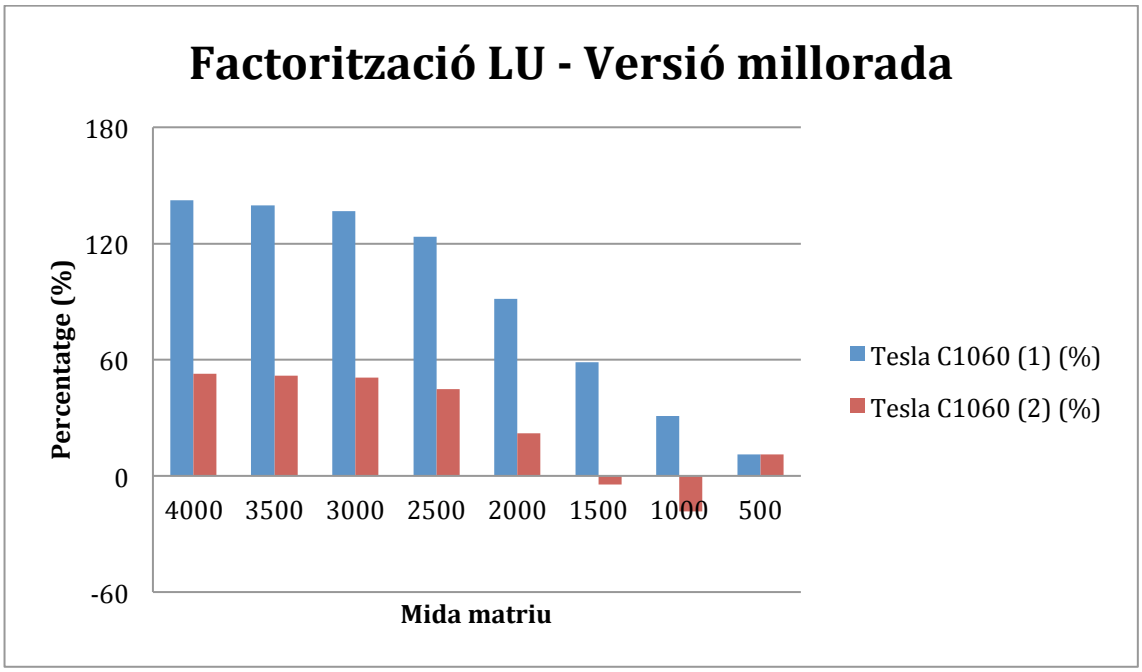



Figura 11. Temps d'execució GPU vs CPU en percentatge

Com es pot veure a les figures el temps de càlcul tant a la versió inicial com a la versió millorada no son inferiors a la versió de la CPU. El principals problemes de la implementació sobre la GPU són els accessos a memòria penalitzen ja que la versió de la CPU treballa amb la RAM que és molt més rapida. També la crida seqüencial als dos

kernels encara que cadascun s'executi en paral·lel no poden arribar al rendiment que proporciona el codi optimitzat pel compilador de la versió per a la CPU.

## 10. Possibles millores

Una possible millora per realitzar la factorització LU es treballar per blocs, si fem el següent:

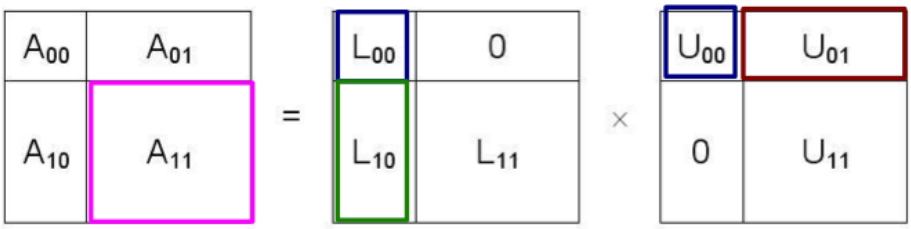



Figura 12. Factorització per blocs

On A és una matriu M x N. I fixem la mida del bloc a r llavors tindrem que $A_{00}$ és una submatriu r x r. Si multipliquem les matrius obtenim el següent:

(1) $L_{00}U_{00}=A_{00}$

(2) $L_{10}U_{00}=A_{10}$

(3) $L_{00}U_{01}=A_{01}$

(4) $L_{10}U_{01}+ L_{11}U_{11}=A_{11}$

on $A_{01}$ és r x (N-r), $A_{10}$ és (M-r) x r i $A_{01}$ és (M-r) x (N-r).

I ara podem resoldre (1) amb una LU sense blocs obtindrem $L_{00}$ i $U_{00}$. A continuació podrem resoldre el sistema triangular inferior (2) i obtindrem $L_{10}$.

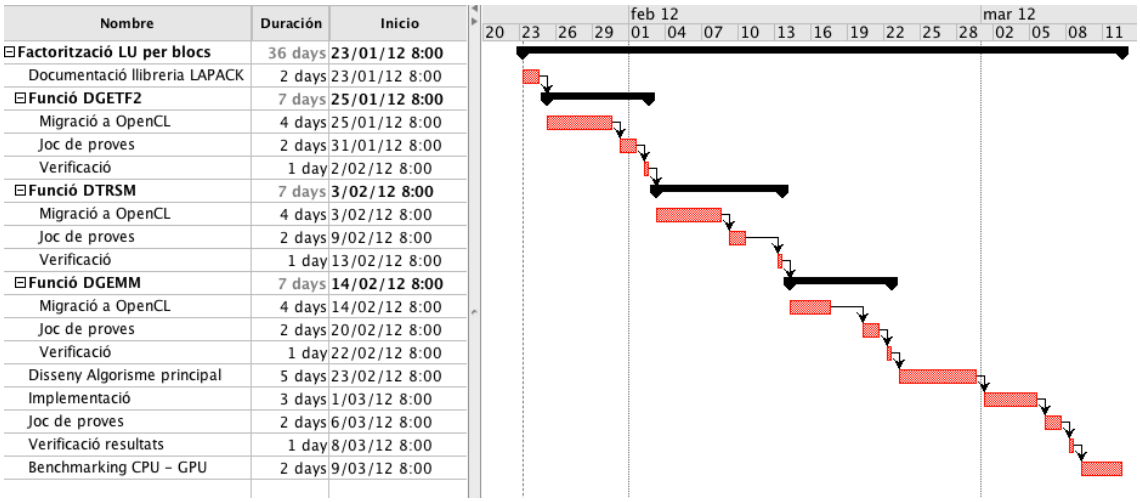
Si fem a partir de (3) $U_{01}=L_{00}^{-1}A_{01}$ i reorganitzem els termes de (4) obtindrem:

(5) $A_{11}'=A_{11}-L_{10}U_{01}=L_{11}U_{11}$

Per tant ara només cal tornar a repetir els passos anterior per trobar un nova factorització LU sobre $A_{11}'$. Així iterativament trobarem la factorització de la matriu A.

La implementació d'aquest procediment a OpenCL és força complexa donat que per a resoldre eficientment caldria disposar d'una versió de la llibreria LAPACK sobre OpenCL. En aquest cas l'equació (1) que és una factorització LU es pot resoldre amb una crida a la funció DGETF2 la (2) i la (3) amb crides a la funció DTRSM que resol sistemes triangulars. I finalment trobaríem $A_{11}'$ cridant a la funció DGEMM.

A continuació es mostra una possible planificació amb les tasques a realitzar per implementar una versió de la factorització en blocs.



| Nombre | Duración | Inicio |
|---|---|---|
| **Factorització LU per blocs** | 36 days | **23/01/12 8:00** |
| Documentació llibreria LAPACK | 2 days | 23/01/12 8:00 |
| **Funció DGETF2** | 7 days | **25/01/12 8:00** |
| Migració a OpenCL | 4 days | 25/01/12 8:00 |
| Joc de proves | 2 days | 31/01/12 8:00 |
| Verificació | 1 day | 2/02/12 8:00 |
| **Funció DTRSM** | 7 days | **3/02/12 8:00** |
| Migració a OpenCL | 4 days | 3/02/12 8:00 |
| Joc de proves | 2 days | 9/02/12 8:00 |
| Verificació | 1 day | 13/02/12 8:00 |
| **Funció DGEMM** | 7 days | **14/02/12 8:00** |
| Migració a OpenCL | 4 days | 14/02/12 8:00 |
| Joc de proves | 2 days | 20/02/12 8:00 |
| Verificació | 1 day | 22/02/12 8:00 |
| Disseny Algorisme principal | 5 days | 23/02/12 8:00 |
| Implementació | 3 days | 1/03/12 8:00 |
| Joc de proves | 2 days | 6/03/12 8:00 |
| Verificació resultats | 1 day | 8/03/12 8:00 |
| Benchmarking CPU - GPU | 2 days | 9/03/12 8:00 |



Figura 13. Planificació de la factorització per blocs

## 11. Conclusions

Les principals conclusions a les quals he arribat com a resultat de l'elaboració d'aquest projecte són les següents:

- No tots els algorismes són directament paral·lelitzables, el cas tractat de la factorització és un exemple ja que n'hi han diverses formes de resoldre-ho però no totes son adients per ser implementades sobre el llenguatge OpenCL.

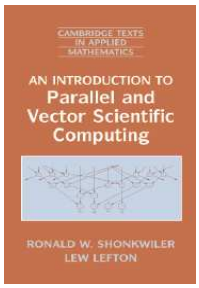
- He trobat serioses deficiències a l'entorn de desenvolupament utilitzat (plataforma Mac OS X) més concretament respecte a l'entorn de depuració (*debugging*). La possibilitat de depurar fent l'execució pas a pas dins d'un kernel han sigut nul·la. Tot això ha fet que la part de depuració sigui més costosa de lo inicialment planificat.

- Com a resultat de l'estudi del llenguatge OpenCL i de les proves realitzades durant l'aprenentatge he comprovat l'augment de rendiment que es pot proporcionar la correcte utilització de la GPU com a recurs de computació.

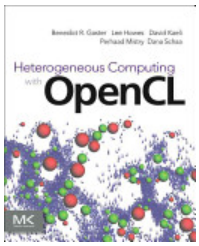
- Les implementacions d'OpenCL existents no son completament homogènies.

# 12. Bibliografia

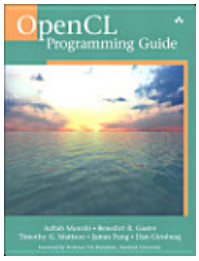
Llibres:




Ronald W. Shonkwiler and Lew Lefton. An Introduction to Parallel and Vector Scientific Computation. Cambridge University Press (2006)




Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry & Dana Schaa. Heterogeneous Computing with OpenCL. Elsevier Science & Technology (2011).




Aaftab Munshi, Benedict R Gaster, Timothy G. Mattson, James Fung, Dan Ginsburg. OpenCL Programming Guide. Addison-Wesley Professional (2011)

Articles:

[1] Khronos, khronos.org, "OpenCL specification 1.0", [Online] http://www.khronos.org/registry/cl/specs/opencl-1.0.pdf
[2] http://www.macresearch.org/tutorial_performance_and_time
[3] Apunts Mètodes numèrics. Wladimiro Díaz Villanueva. Departament d'Informàtica. Universitat de València. http://www.uv.es/diaz/mn/fmn.html
[4] E.E.Santos, M.Muraleetharan. Analysis and Implementation of Parallel LU Decomposition with Different Data Layouts. June 2000
[5] LU Example. Barcelona Supercomputings Center. http://www.bsc.es/plantillaH.php?cat_id=426
[6] Stephen Ng. Factoring Matrices Gaussian Elimination and LU Factorization. UC Davis.

[7] Weisstein, Eric W. "LU Decomposition." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/LUDecomposition.html
[8] Luis Miguel de la Cruz. Computación Científica en Paralelo. DGSCA-UNAM. http://mmc2.geofisica.unam.mx/luiggi/CCP/03print.pdf
[9] Michael T. Heath. Parallel Numerical Algorithms. Chapter 6 – LU Factorization. University of Illinois at Urbana-Champaign. http://www.cse.illinois.edu/courses/cs554/notes/06_lu.pdf
[10] Miquel Àngel Fontana Torroba. Paral·lelització de la factorització LU. PFC. FIB. Universitat Politècnica Catalunya.
[11] Simon McIntosh-Smith. Optimising OpenCL performance. Department of Computer Science. University of Bristol. http://www.cs.bris.ac.uk/home/simonm/workshops/OpenCL_lecture3.pdf
[12] Introduction to GPU computing with OpenCL. http://developer.download.nvidia.com/CUDA/training/NVIDIA_GPU_Computing_Webinars_Introduction_To_OpenCL.pdf
[13] OpenCL Programming Guide for Mac OS X. Apple. http://developer.apple.com/library/mac/documentation/Performance/Conceptual/OpenCL_MacProgGuide/OpenCL_MacProgGuide.pdf
[14] Codeproject. Part 1: OpenCL™ – Portable Parallelism. http://www.codeproject.com/KB/showcase/Portable-Parallelism.aspx
[15] Codeproject. Part 2: OpenCL™ – Memory Spaces. http://www.codeproject.com/KB/showcase/Memory-Spaces.aspx
[16] Codeproject. Part 3: Work-Groups and Synchronization. http://www.codeproject.com/KB/showcase/Work-Groups-Sync.aspx
[17] LU Matrix Decomposition in parallel with CUDA. http://www.noctua-blog.com/index.php/2011/04/21/lu-matrix-decomposition-in-parallel-with-cuda/
[18] Cristian González Sánchez. Implementación de un Raytracer para la evaluación de OpenCL. PFC. Universitat Politècnica de Catalunya. http://upcommons.upc.edu/pfc/handle/2099.1/11641
[19] Jack Dongarra. Derivation of a Block Algorithm for LU Factorization. http://www.netlib.org/utk/papers/siam-review93/node13.html
[20] Javier Cuenca. Programación Paralela y Computación de Altas Prestaciones Algoritmos Matriciales por Bloques. Dpto. de Ingeniería y Tecnología de Computadores. Universidad de Murcia. http://dis.um.es/~domingo/apuntes/PPCAP/1011/PPCAP_1011_AlgMatBlo.pdf

# Apèndix 1

```
//
//  main.c
//  LU decomposition (CPU)
//
//  Autor: Javier Vegas Caballero
//  Versió de referencia
//  UOC TFC


#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <unistd.h>
#include <time.h>
#include <math.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <string.h>
#include <stdint.h>

#ifdef __APPLE__
#include <stdint.h>
#include <mach/mach_time.h>
#endif

void save_matrix_csv(const char *filename, float *M, int n)
{
    FILE *fp;
    int i,j;

    fp = fopen(filename,"w");
    if (!fp) {
        fprintf(stderr, "Failed to save matrix.\n");
        return;
    }

    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            fprintf(fp,"%lf;",M[i*n+j]);
        }
        fprintf(fp,"\n");
    }

    fclose(fp);
}

void display_matrix(char t, float *M, int n)
{
    int i,j;

    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            printf("%c[%d][%d]:%8.2lf ",t,i,j,M[i*n+j]);
        }
        printf("\n");
    }

}
```

```
void create_random_matrix(float *M, int n)
{
    int i,j;

    srand((unsigned int)time(0));

    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            M[i*n+j]=10.0f*((float)rand()/(float)RAND_MAX)+
               10.0f*((float)rand()/(float)RAND_MAX);

}


int main (int argc, char * const argv[])
{
    int i, j, k, n=3;
    float *A;
    float *U;
    float *L;


    /* Parsing arguments with getopt */
    int pflag = 0;
    int vflag = 0;
    int tflag = 0;
    int cflag = 0;
    int opt;

    opterr = 0;

    while ((opt = getopt (argc, argv, "cn:pvt")) != -1)
        switch (opt)
    {
        case 'n': /* number of rows and columns */
            n = atoi(optarg);
            break;

        case 'p': /* print stdout A,L,U */
            pflag = 1;
            break;

        case 'v': /* save csv file A,L,U */
            vflag = 1;
            break;

        case 'c': /* check results */
            cflag = 1;
            break;

        case 't': /* test data */
            tflag = 1;
            n=3;
            break;

        default:
            abort();
    }

    /* Memory allocation */
    A = (float *)malloc(n*n*sizeof(float));
    L = (float *)calloc(n*n,sizeof(float));
    U = (float *)malloc(n*n*sizeof(float));

    /* Initialize input data */
    if(!tflag)
    {
        /* Create some random input data */
        printf("Creating random input data [%d x %d]...\n", n, n);

        create_random_matrix(A,n);

        printf("Random input data created [%d x %d]\n", n, n);
    }
    else
```

```
    {
        printf("Creating test input data [%d x %d]...\n", n, n);

        A[0]=60;A[1]=30;A[2]=20;
        A[3]=30;A[4]=20;A[5]=15;
        A[6]=20;A[7]=15;A[8]=12;

        printf("Test input data created [%d x %d]\n", n, n);

    }

    /* Init U with A */
    memcpy(U,A,n*n*sizeof(float));

    if(vflag)
        save_matrix_csv("A.csv", A, n);

    if(pflag)
        display_matrix('A',A,n);


    /* Compute LU */

    printf("Performing LU decomposition [%d x %d]...\n", n, n);
#ifdef __APPLE__
    uint64_t         t1, t2;
    t1 = mach_absolute_time();
#else
    clock_t start = clock();
#endif


    for(i=0;i<n;i++)
    {
        L[i*n+i]=1.0f;
        for(j=i+1;j<n;j++)
        {
            L[j*n+i]=U[j*n+i]/U[i*n+i];
            for(k=i;k<n;k++)
                U[j*n+k]-=(L[j*n+i]*U[i*n+k]);
        }
    }

#ifdef __APPLE__
    t2 = mach_absolute_time();

    /* Calculate execution time */

    struct mach_timebase_info info;
    mach_timebase_info(&info);
    double t = 1e-9 * (t2 - t1) * info.numer / info.denom;
    printf("Elapsed Time:  %.2f s\n", t);

#else

    clock_t end = clock() ;
    double elapsed_time = (end-start)/(double)CLOCKS_PER_SEC;
    printf("Elapsed time: %lf seconds \n",elapsed_time);

#endif

    /* Display Results */
    if(pflag)
    {
        display_matrix('L',L,n);
        display_matrix('U',U,n);
    }

    /* Save Results */
    if(vflag)
    {
        save_matrix_csv("L.csv",L,n);
        save_matrix_csv("U.csv",U,n);
    }
```

```
    /* Check Results */
    if(cflag)
    {
        float aij;
        float error=0.0f;

        printf("Verifying results: A = L * U ...\n");

        for(i=0;i<n;i++)
        {
            for(j=0;j<n;j++)
            {
                aij=0.0f;
                for(k=0;k<n;k++)
                {
                    aij+=L[i*n+k]*U[k*n+j];
                }
                error+=fabs(A[i*n+j]-aij);

            }

        }

        printf("Total error: %lf\n",error/(n*n));

    }


    return 0;
}
```

# Apèndix 2

```
//
//  main.c
//  LU decomposition (OpenCL)
//
//  Autor: Javier Vegas Caballero
//  Versió inicial
//  UOC TFC


#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <unistd.h>
#include <time.h>
#include <math.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <string.h>

#ifdef __APPLE__
#include <OpenCL/opencl.h>
#include <stdint.h>
#include <mach/mach_time.h>
#else
#include <CL/cl.h>
#endif

#define KERNEL_FILE1 "Pivot.cl"
#define KERNEL_NAME1 "Pivot"

#define KERNEL_FILE2 "ForwardElimination.cl"
#define KERNEL_NAME2 "ForwardElimination"

#define MAX_SOURCE_SIZE (0x100000)

#define CL_CHECK(_expr)                                                         \
    do {                                                                        \
        cl_int _err = _expr;                                                    \
        if (_err == CL_SUCCESS)                                                 \
            break;                                                              \
        fprintf(stderr, "OpenCL Error: '%s' returned %d!\n", #_expr, (int)_err);   \
        abort();                                                                \
        } while (0)

#define CL_CHECK_ERR(_expr)                                                     \
    ({                                                                          \
        cl_int _err = CL_INVALID_VALUE;                                         \
        typeof(_expr) _ret = _expr;                                             \
        if (_err != CL_SUCCESS) {                                               \
            fprintf(stderr, "OpenCL Error: '%s' returned %d!\n", #_expr, (int)_err);\
            abort();                                                            \
        }                                                                       \
        _ret;                                                                   \
    })


void checkErrors(cl_int status, char *label, int line)
{
    switch (status)
    {
        case CL_SUCCESS:
            return;
        case CL_BUILD_PROGRAM_FAILURE:
            fprintf(stderr, "OpenCL error (at %s, line %d): CL_BUILD_PROGRAM_FAILURE\n",
               label, line);
            break;
        case CL_COMPILER_NOT_AVAILABLE:
            fprintf(stderr, "OpenCL error (at %s, line %d): CL_COMPILER_NOT_AVAILABLE\n",
               label, line);
            break;
```

```
case CL_DEVICE_NOT_AVAILABLE:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_DEVICE_NOT_AVAILABLE\n",
       label, line);
    break;
case CL_DEVICE_NOT_FOUND:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_DEVICE_NOT_FOUND\n",
       label, line);
    break;
case CL_IMAGE_FORMAT_MISMATCH:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_IMAGE_FORMAT_MISMATCH\n",
       label, line);
    break;
case CL_IMAGE_FORMAT_NOT_SUPPORTED:
    fprintf(stderr,
       "OpenCL error (at %s, line %d): CL_IMAGE_FORMAT_NOT_SUPPORTED\n",
       label, line);
    break;
case CL_INVALID_ARG_INDEX:
    fprintf(stderr,
       "OpenCL error (at %s, line %d): CL_INVALID_ARG_INDEX\n",
       label, line);
    break;
case CL_INVALID_ARG_SIZE:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_ARG_SIZE\n",
       label, line);
    break;
case CL_INVALID_ARG_VALUE:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_ARG_VALUE\n",
       label, line);
    break;
case CL_INVALID_BINARY:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_BINARY\n",
       label, line);
    break;
case CL_INVALID_BUFFER_SIZE:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_BUFFER_SIZE\n",
       label, line);
    break;
case CL_INVALID_BUILD_OPTIONS:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_BUILD_OPTIONS\n",
       label, line);
    break;
case CL_INVALID_COMMAND_QUEUE:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_COMMAND_QUEUE\n",
       label, line);
    break;
case CL_INVALID_CONTEXT:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_CONTEXT\n",
       label, line);
    break;
case CL_INVALID_DEVICE:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_DEVICE\n",
       label, line);
    break;
case CL_INVALID_DEVICE_TYPE:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_DEVICE_TYPE\n",
       label, line);
    break;
case CL_INVALID_EVENT:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_EVENT\n",
       label, line);
    break;
case CL_INVALID_EVENT_WAIT_LIST:
    fprintf(stderr,
       "OpenCL error (at %s, line %d): CL_INVALID_EVENT_WAIT_LIST\n",
       label, line);
    break;
case CL_INVALID_GL_OBJECT:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_GL_OBJECT\n",
       label, line);
    break;
case CL_INVALID_GLOBAL_OFFSET:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_GLOBAL_OFFSET\n",
       label, line);
    break;
```

```
case CL_INVALID_HOST_PTR:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_HOST_PTR\n",
       label, line);
    break;
case CL_INVALID_IMAGE_FORMAT_DESCRIPTOR:
    fprintf(stderr,
       "OpenCL error (at %s, line %d): CL_INVALID_IMAGE_FORMAT_DESCRIPTOR\n",
       label, line);
    break;
case CL_INVALID_IMAGE_SIZE:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_IMAGE_SIZE\n",
       label, line);
    break;
case CL_INVALID_KERNEL_NAME:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_KERNEL_NAME\n",
       label, line);
    break;
case CL_INVALID_KERNEL:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_KERNEL\n",
       label, line);
    break;
case CL_INVALID_KERNEL_ARGS:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_KERNEL_ARGS\n",
       label, line);
    break;
case CL_INVALID_KERNEL_DEFINITION:
    fprintf(stderr,
       "OpenCL error (at %s, line %d): CL_INVALID_KERNEL_DEFINITION\n",
       label, line);
    break;
case CL_INVALID_MEM_OBJECT:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_MEM_OBJECT\n",
       label, line);
    break;
case CL_INVALID_OPERATION:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_OPERATION\n",
       label, line);
    break;
case CL_INVALID_PLATFORM:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_PLATFORM\n",
       label, line);
    break;
case CL_INVALID_PROGRAM:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_PROGRAM\n",
       label, line);
    break;
case CL_INVALID_PROGRAM_EXECUTABLE:
    fprintf(stderr,
       "OpenCL error (at %s, line %d): CL_INVALID_PROGRAM_EXECUTABLE\n",
       label, line);
    break;
case CL_INVALID_QUEUE_PROPERTIES:
    fprintf(stderr,
       "OpenCL error (at %s, line %d): CL_INVALID_QUEUE_PROPERTIES\n",
       label, line);
    break;
case CL_INVALID_SAMPLER:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_SAMPLER\n",
       label, line);
    break;
case CL_INVALID_VALUE:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_VALUE\n",
       label, line);
    break;
case CL_INVALID_WORK_DIMENSION:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_WORK_DIMENSION\n",
       label, line);
    break;
case CL_INVALID_WORK_GROUP_SIZE:
    fprintf(stderr,
       "OpenCL error (at %s, line %d): CL_INVALID_WORK_GROUP_SIZE\n",
       label, line);
    break;
```

```
        case CL_INVALID_WORK_ITEM_SIZE:
            fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_WORK_ITEM_SIZE\n",
               label, line);
            break;
        case CL_MAP_FAILURE:
            fprintf(stderr, "OpenCL error (at %s, line %d): CL_MAP_FAILURE\n",
               label, line);
            break;
        case CL_MEM_OBJECT_ALLOCATION_FAILURE:
            fprintf(stderr,
              "OpenCL error (at %s, line %d): CL_MEM_OBJECT_ALLOCATION_FAILURE\n",
               label, line);
            break;
        case CL_MEM_COPY_OVERLAP:
            fprintf(stderr, "OpenCL error (at %s, line %d): CL_MEM_COPY_OVERLAP\n",
               label, line);
            break;
        case CL_OUT_OF_HOST_MEMORY:
            fprintf(stderr, "OpenCL error (at %s, line %d): CL_OUT_OF_HOST_MEMORY\n",
               label, line);
            break;
        case CL_OUT_OF_RESOURCES:
            fprintf(stderr, "OpenCL error (at %s, line %d): CL_OUT_OF_RESOURCES\n",
               label, line);
            break;
        case CL_PROFILING_INFO_NOT_AVAILABLE:
            fprintf(stderr,
              "OpenCL error (at %s, line %d): CL_PROFILING_INFO_NOT_AVAILABLE\n",
               label, line);
            break;
    }
    exit(status);
}



static char *
load_program_source(const char *filename, size_t *size)
{
    FILE *fp;
    char *source;

    fp = fopen(filename, "r");
    if (!fp) {
        fprintf(stderr, "Failed to load kernel.\n");
        exit(1);
    }
    source = (char *)malloc(MAX_SOURCE_SIZE);
    *size = fread( source, 1, MAX_SOURCE_SIZE, fp );
    fclose( fp );
    return source;
}

void save_matrix_csv(const char *filename, float *M, int n)
{
    FILE *fp;
    int i,j;

    fp = fopen(filename,"w");
    if (!fp) {
        fprintf(stderr, "Failed to save matrix.\n");
        return;
    }

    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            fprintf(fp,"%lf;",M[i*n+j]);
        }
        fprintf(fp,"\n");
    }

    fclose(fp);
}
```

```
void display_matrix(char t, float *M, int n)
{
    int i,j;

    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            printf("%c[%d][%d]:%8.2lf ",t,i,j,M[i*n+j]);
        }
        printf("\n");
    }
     printf("\n");

}

void create_random_matrix(float *M, int n)
{
    int i,j;

    srand((unsigned int)time(0));

    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
        {

M[i*n+j]=10.0f*((float)rand()/(float)RAND_MAX)+10.0f*((float)rand()/(float)RAND_MAX);

        }

}

int main (int argc, char **argv)
{

    cl_platform_id platform_id = NULL;
    cl_device_id device_id = NULL;
    cl_context context = NULL;
    cl_command_queue command_queue = NULL;
    cl_mem AUbuff = NULL;
    cl_mem Lbuff  = NULL;
    cl_program program1 = NULL, program2 = NULL;
    cl_kernel *kernel1, *kernel2 = NULL;
    cl_uint ret_num_devices;
    cl_uint ret_num_platforms;
    cl_int ret;


    int i, j, k, ndim=0;
    float *A;
    float *U;
    float *L;


    /* Parsing arguments with getopt */
    int pflag = 0;
    int vflag = 0;
    int tflag = 0;
    int cflag = 0;
    int opt;

    opterr = 0;

    while ((opt = getopt (argc, argv, "cn:pvt")) != -1)
        switch (opt)
    {
        case 'n': /* number of rows and columns */
            ndim = atoi(optarg);
            break;

        case 'p': /* print stdout A,L,U */
            pflag = 1;
            break;
```

```
        case 'v': /* save csv file A,L,U */
            vflag = 1;
            break;

        case 'c': /* check results */
            cflag = 1;
            break;

        case 't': /* test data */
            tflag = 1;
            ndim=3;
            break;

        default:
            abort();
    }

    /* Memory allocation */
    A = (float *)malloc(ndim*ndim*sizeof(float));
    L = (float *)calloc(ndim*ndim,sizeof(float));
    U = (float *)malloc(ndim*ndim*sizeof(float));

    kernel1 = (cl_kernel *)malloc(ndim*sizeof(cl_kernel));
    kernel2 = (cl_kernel *)malloc(ndim*sizeof(cl_kernel));

    /* Load kernel source file */
    char *source_str1, *source_str2;
    size_t source_size1, source_size2;

    source_str1=load_program_source(KERNEL_FILE1, &source_size1);
    source_str2=load_program_source(KERNEL_FILE2, &source_size2);

    /* Initialize input data */
    if(!tflag)
    {
        /* Create some random input data */
        printf("Creating random input data [%d x %d]...\n", ndim, ndim);

        create_random_matrix(A,ndim);

        printf("Random input data created [%d x %d]\n", ndim, ndim);
    }
    else
    {
        printf("Creating test input data [%d x %d]...\n", ndim, ndim);

        A[0]=60;A[1]=30;A[2]=20;
        A[3]=30;A[4]=20;A[5]=15;
        A[6]=20;A[7]=15;A[8]=12;

        printf("Test input data created [%d x %d]\n", ndim, ndim);

    }

    if(vflag)
        save_matrix_csv("A.csv", A, ndim);

    if(pflag)
      display_matrix('A',A,ndim);


    /* Get Platform/Device Information */
    ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
    if (ret != CL_SUCCESS)
    {
        printf("Error: Failed to get device info!\n");
        return EXIT_FAILURE;
    }
    /* Connect to a GPU compute device */
    ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id,
              &ret_num_devices);
    if (ret != CL_SUCCESS)
    {
        printf("Error: Failed to create a device group!\n");
        return EXIT_FAILURE;
```

```
}

/* Show OpenCL device information */
printf("=== %d OpenCL device(s) found on platform:\n", ret_num_devices);
  {
    char buffer[10240];
    cl_uint buf_uint;
    cl_ulong buf_ulong;

     size_t arr_tsize[3];
     size_t ret_size;

    CL_CHECK(clGetDeviceInfo(device_id, CL_DEVICE_NAME, sizeof(buffer), buffer,
           NULL));
    printf("  DEVICE_NAME = %s\n", buffer);
    CL_CHECK(clGetDeviceInfo(device_id, CL_DEVICE_VENDOR, sizeof(buffer), buffer,
           NULL));
    printf("  DEVICE_VENDOR = %s\n", buffer);
    CL_CHECK(clGetDeviceInfo(device_id, CL_DEVICE_VERSION, sizeof(buffer), buffer,
           NULL));
    printf("  DEVICE_VERSION = %s\n", buffer);
    CL_CHECK(clGetDeviceInfo(device_id, CL_DRIVER_VERSION, sizeof(buffer), buffer,
           NULL));
    printf("  DRIVER_VERSION = %s\n", buffer);
    CL_CHECK(clGetDeviceInfo(device_id, CL_DEVICE_MAX_COMPUTE_UNITS,
           sizeof(buf_uint), &buf_uint, NULL));
    printf("  DEVICE_MAX_COMPUTE_UNITS = %u\n", (unsigned int)buf_uint);
   CL_CHECK(clGetDeviceInfo(device_id, CL_DEVICE_MAX_WORK_GROUP_SIZE,
           sizeof(size_t), arr_tsize, &ret_size));
    printf("  DEVICE_MAX_WORK_GROUP_SIZE = %lu\n", (unsigned long)arr_tsize[0]);

   clGetDeviceInfo(device_id,CL_DEVICE_MAX_WORK_ITEM_SIZES,3*sizeof(size_t),
           arr_tsize,&ret_size);

   printf("\tMax Work-Item Sizes:\t(%ld,%ld,%ld)\n",
           arr_tsize[0],arr_tsize[1],arr_tsize[2]);


   CL_CHECK(clGetDeviceInfo(device_id, CL_DEVICE_MAX_CLOCK_FREQUENCY,
           sizeof(buf_uint), &buf_uint, NULL));
    printf("  DEVICE_MAX_CLOCK_FREQUENCY = %u\n", (unsigned int)buf_uint);
    CL_CHECK(clGetDeviceInfo(device_id, CL_DEVICE_GLOBAL_MEM_SIZE,
           sizeof(buf_ulong), &buf_ulong, NULL));
    printf("  DEVICE_GLOBAL_MEM_SIZE = %llu\n", (unsigned long long)buf_ulong);

  }


/* Create OpenCL Context */
context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);
if (!context)
{
    printf("Error: Failed to create a compute context!\n");
    return EXIT_FAILURE;
}

/* Create command queue */
command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
if (!command_queue)
{
    printf("Error: Failed to create a command queue!\n");
    return EXIT_FAILURE;
}

/* Create Input/Output Buffer Object */
AUbuff = clCreateBuffer(context, CL_MEM_READ_WRITE, ndim*ndim*sizeof(float),
                        NULL, &ret);
if (!AUbuff)
{
    printf("Error: Failed to allocate destination array!\n");
    return EXIT_FAILURE;
}
```

```
/* Create Output Buffer Object */
Lbuff  = clCreateBuffer(context, CL_MEM_READ_WRITE, ndim*ndim*sizeof(float),
                        NULL, &ret);
if (!Lbuff)
{
    printf("Error: Failed to allocate destination array!\n");
    return EXIT_FAILURE;
}

/* Copy input/output data to the memory buffer */
ret = clEnqueueWriteBuffer(command_queue, AUbuff, CL_TRUE, 0,
                        ndim*ndim*sizeof(float), A, 0, NULL, NULL);
if (ret != CL_SUCCESS)
{
    printf("Error: Failed to write to source array!\n");
    return EXIT_FAILURE;
}

/* Copy output data to the memory buffer */
ret = clEnqueueWriteBuffer(command_queue, Lbuff, CL_TRUE, 0, ndim*ndim*sizeof(float),
                        L, 0, NULL, NULL);
if (ret != CL_SUCCESS)
{
    printf("Error: Failed to write to source array!\n");
    return EXIT_FAILURE;
}

/* Create kernel program from source file*/
program1 = clCreateProgramWithSource(context, 1, (const char **)&source_str1,
                                (const size_t *)&source_size1, &ret);
if (!program1 || ret != CL_SUCCESS)
{
    printf("Error: Failed to create compute program!\n");
    return EXIT_FAILURE;
}

program2 = clCreateProgramWithSource(context, 1, (const char **)&source_str2,
   (const size_t *)&source_size2, &ret);
if (!program1 || ret != CL_SUCCESS)
{
    printf("Error: Failed to create compute program!\n");
    return EXIT_FAILURE;
}

/* Build the program executable */
ret = clBuildProgram(program1, 1, &device_id, NULL, NULL, NULL);
if (ret != CL_SUCCESS)
{
    size_t len;
    char buffer[2048];

    printf("Error: Failed to build program executable!\n");
    clGetProgramBuildInfo(program1, device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer),
                        buffer, &len);
    printf("%s\n", buffer);
    return EXIT_FAILURE;
}

ret = clBuildProgram(program2, 1, &device_id, NULL, NULL, NULL);
if (ret != CL_SUCCESS)
{
    size_t len;
    char buffer[2048];

    printf("Error: Failed to build program executable!\n");
    clGetProgramBuildInfo(program2, device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer),
                        buffer, &len);
    printf("%s\n", buffer);
    return EXIT_FAILURE;
}
```

```
/* Create data parallel OpenCL kernel */
for(i=0;i<ndim;i++)
{
    kernel1[i] = clCreateKernel(program1, KERNEL_NAME1, &ret);
    if (!kernel1[i] || ret != CL_SUCCESS)
    {
        printf("Error: Failed to create compute kernel!\n");
        return EXIT_FAILURE;
    }
}

for(i=0;i<ndim;i++)
{
    kernel2[i] = clCreateKernel(program2, KERNEL_NAME2, &ret);
    if (!kernel2[i] || ret != CL_SUCCESS)
    {
        printf("Error: Failed to create compute kernel!\n");
        return EXIT_FAILURE;
    }
}

/* Get Info kernel */
size_t group_size;
clGetKernelWorkGroupInfo(kernel1[0], device_id, CL_KERNEL_WORK_GROUP_SIZE,
                         sizeof(group_size), &group_size, NULL);
printf(" (1) CL_KERNEL_WORK_GROUP_SIZE = %lu\n", group_size);

clGetKernelWorkGroupInfo(kernel2[0], device_id, CL_KERNEL_WORK_GROUP_SIZE,
                         sizeof(group_size), &group_size, NULL);
printf(" (2) CL_KERNEL_WORK_GROUP_SIZE = %lu\n", group_size);



/* Set OpenCL kernel arguments */
ret=CL_SUCCESS;
for(i=0;i<ndim;i++)
{
    ret|= clSetKernelArg(kernel1[i], 0, sizeof(cl_mem), (void *)&AUbuff);
    ret|= clSetKernelArg(kernel1[i], 1, sizeof(cl_mem), (void *)&Lbuff);
    ret|= clSetKernelArg(kernel1[i], 2, sizeof(int), &ndim);
    ret|= clSetKernelArg(kernel1[i], 3, sizeof(int), &i);
    if (ret != CL_SUCCESS)
    {
        printf("Error: Failed to set kernel arguments!\n");
        return EXIT_FAILURE;
    }
}

ret=CL_SUCCESS;
for(i=0;i<ndim-1;i++)
{
    ret|= clSetKernelArg(kernel2[i], 0, sizeof(cl_mem), (void *)&AUbuff);
    ret|= clSetKernelArg(kernel2[i], 1, sizeof(cl_mem), (void *)&Lbuff);
    ret|= clSetKernelArg(kernel2[i], 2, sizeof(int), &ndim);
    ret|= clSetKernelArg(kernel2[i], 3, sizeof(int), &i);
    ret|= clSetKernelArg(kernel2[i], 4, sizeof(float)*ndim, NULL);
    if (ret != CL_SUCCESS)
    {
        printf("Error: Failed to set kernel arguments!\n");
        return EXIT_FAILURE;
    }
}

/* Determine the global dimensions for the execution */
size_t global1[1] = {ndim};
size_t local1[1] = {ndim};

size_t localWorkSize1;
size_t globalWorkSize1;

size_t global2[2] = {ndim,ndim};
size_t local2[2] = {ndim,ndim};

size_t localWorkSize2[2];
size_t globalWorkSize2[2];
```

```
#ifdef __APPLE__
    uint64_t         t1, t2;
    t1 = mach_absolute_time();
#endif

#ifndef __APPLE__
    clock_t start = clock();
#endif


    /* Start the timing loop and execute the kernel over several iterations */

    printf("Performing LU decomposition [%d x %d]...\n", ndim, ndim);

    for(i=0;i<ndim;i++)
    {

        ret|= clSetKernelArg(kernel1[0], 3, sizeof(int), &i);
        if (ret != CL_SUCCESS)
        {
            printf("Error: Failed to set kernel arguments!\n");
            return EXIT_FAILURE;
        }

        global1[0]=ndim-i;
        local1[0]=1;

        localWorkSize1 = 32;
        globalWorkSize1 = ( (global1[0]+localWorkSize1-1) / localWorkSize1 ) *
                           localWorkSize1;
        global1[0] = globalWorkSize1;
        local1[0] = localWorkSize1;

        /* Execute OpenCL kernel as data parallel */
        ret = clEnqueueNDRangeKernel(command_queue, kernel1[0], 1, NULL, global1, local1,
                                     0, NULL, NULL);
        checkErrors (ret, "clEnqueueNDRangeKernel", __LINE__);
        if (ret!= CL_SUCCESS)
        {
            printf("Error: Failed to execute kernel!\n");
            return EXIT_FAILURE;
        }
        ret = clFlush(command_queue);

        if(i<ndim-1)
        {

            global2[0]=ndim-1-i;
            global2[1]=ndim-i;

            localWorkSize2[0] = 32;
            localWorkSize2[1] = 1;

            globalWorkSize2[0] = ( (global2[0]+localWorkSize2[0]-1) /
                                    localWorkSize2[0] ) * localWorkSize2[0];
            globalWorkSize2[1] = ( (global2[1]+localWorkSize2[1]-1) /
                                    localWorkSize2[1] ) * localWorkSize2[1];
            global2[0] = globalWorkSize2[0];
            global2[1] = globalWorkSize2[1];
            local2[0] = localWorkSize2[0];
            local2[1] = 1;

            ret|= clSetKernelArg(kernel2[0], 3, sizeof(int), &i);
            if (ret != CL_SUCCESS)
            {
                printf("Error: Failed to set kernel arguments!\n");
                return EXIT_FAILURE;
            }

            ret = clEnqueueNDRangeKernel(command_queue, kernel2[0], 2, NULL, global2,
               local2, 0, NULL, NULL);
            checkErrors (ret, "clEnqueueNDRangeKernel", __LINE__);
```

```
            if (ret!= CL_SUCCESS)
            {
                printf("Error: Failed to execute kernel!\n");
                return EXIT_FAILURE;
            }
            ret = clFlush(command_queue);
        }

    }


#ifdef __APPLE__
    t2 = mach_absolute_time();

    /* Calculate execution time */

    struct mach_timebase_info info;
    mach_timebase_info(&info);
    double t = 1e-9 * (t2 - t1) * info.numer / info.denom;
    printf("Elapsed time:  %.2f s\n", t);

#endif

#ifndef __APPLE__
    clock_t end = clock() ;
    double elapsed_time = (end-start)/(double)CLOCKS_PER_SEC;
    printf("Elapsed time: %.2f s\n",elapsed_time);
#endif

    /* Transfer result to host */
    ret = clEnqueueReadBuffer(command_queue, AUbuff, CL_TRUE, 0, ndim*ndim*sizeof(float),
                              U, 0, NULL, NULL);
    if (ret)
    {
        printf("Error: Failed to read back results from the device!\n");
        return EXIT_FAILURE;
    }

    /* Transfer result to host */
    ret = clEnqueueReadBuffer(command_queue, Lbuff, CL_TRUE, 0, ndim*ndim*sizeof(float),
                              L, 0, NULL, NULL);
    if (ret)
    {
        printf("Error: Failed to read back results from the device!\n");
        return EXIT_FAILURE;
    }


    /* Display Results */
    if(pflag)
    {
        display_matrix('L',L,ndim);
        display_matrix('U',U,ndim);
    }

    /* Save Results */
    if(vflag)
    {
        save_matrix_csv("L.csv",L,ndim);
        save_matrix_csv("U.csv",U,ndim);
    }

    /* Check Results */
    if(cflag)
    {
        float aij;
        float error=0.0f;
        FILE * pFile;

        pFile = fopen ("M.csv","w");

        printf("Verifying results: A = L * U ...\n");
```

```
        for(i=0;i<ndim;i++)
        {
            for(j=0;j<ndim;j++)
            {
                aij=0.0f;
                for(k=0;k<ndim;k++)
                {
                    aij+=L[i*ndim+k]*U[k*ndim+j];
                }
                error+=fabs(A[i*ndim+j]-aij);
                fprintf(pFile,"%lf;",aij);

            }
            fprintf(pFile,"\n");
        }
        fclose(pFile);

        printf("Total error: %lf\n",error/(ndim*ndim));


    }

    /* Finalization */
    ret = clFlush(command_queue);
    ret = clFinish(command_queue);
    for(i=0;i<ndim;i++)
        ret = clReleaseKernel(kernel1[i]);
    for(i=0;i<ndim;i++)
        ret = clReleaseKernel(kernel2[i]);

    ret = clReleaseProgram(program1);
    ret = clReleaseProgram(program2);
    ret = clReleaseMemObject(AUbuff);
    ret = clReleaseMemObject(Lbuff);
    ret = clReleaseCommandQueue(command_queue);
    ret = clReleaseContext(context);

    free(source_str1);
    free(source_str2);


    free(A);
    free(U);
    free(L);

    return EXIT_SUCCESS;
}
```

# Apèndix 3

```
//
//  main.c
//  LU decomposition (OpenCL)
//
//  Autor: Javier Vegas Caballero
//  Versió millorada
//  UOC TFC

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <unistd.h>
#include <time.h>
#include <math.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <string.h>

#ifdef __APPLE__
#include <OpenCL/opencl.h>
#include <stdint.h>
#include <mach/mach_time.h>
#else
#include <CL/cl.h>
#endif

#include "utils.h"

#define KERNEL_FILE "ludecomposition.cl"
#define KERNEL_NAME "ludecomposition"

int main (int argc, char **argv)
{

    cl_platform_id platform_id = NULL;
    cl_device_id device_id = NULL;
    cl_context context = NULL;
    cl_command_queue command_queue = NULL;
    cl_mem AUbuff = NULL;
    cl_mem Lbuff  = NULL;
    cl_program program = NULL;
    cl_kernel kernel;
    cl_uint ret_num_devices;
    cl_uint ret_num_platforms;
    cl_int ret;
    cl_event *event;


    int i, j, k, ndim=0;
    float *A;
    float *U;
    float *L;


    /* Parsing arguments with getopt */
    int pflag = 0;
    int vflag = 0;
    int tflag = 0;
    int cflag = 0;
    int opt;

    opterr = 0;

    while ((opt = getopt (argc, argv, "cn:pvt")) != -1)
        switch (opt)
    {
        case 'n': /* number of rows and columns */
            ndim = atoi(optarg);
            break;
```

```
        case 'p': /* print stdout A,L,U */
            pflag = 1;
            break;

        case 'v': /* save csv file A,L,U */
            vflag = 1;
            break;

        case 'c': /* check results */
            cflag = 1;
            break;

        case 't': /* test data */
            tflag = 1;
            ndim=3;
            break;

        default:
            abort();
    }

    /* Memory allocation */
    A = (float *)malloc(ndim*ndim*sizeof(float));
    L = (float *)calloc(ndim*ndim,sizeof(float));
    U = (float *)malloc(ndim*ndim*sizeof(float));

    event = (cl_event *)malloc(ndim*sizeof(cl_event));

    /* Load kernel source file */
    char *source_str;
    size_t source_size;

    source_str=load_program_source(KERNEL_FILE, &source_size);

    /* Initialize input data */

    for(i=0;i<ndim;i++)
        for(j=0;j<ndim;j++)
            if(i>=j)
                L[i*ndim+j]=1.0;


    if(!tflag)
    {
        /* Create some random input data */
        printf("Creating random input data [%d x %d]...\n", ndim, ndim);

        create_random_matrix(A,ndim);

        printf("Random input data created [%d x %d]\n", ndim, ndim);
    }
    else
    {
        printf("Creating test input data [%d x %d]...\n", ndim, ndim);

        A[0]=60;A[1]=30;A[2]=20;
        A[3]=30;A[4]=20;A[5]=15;
        A[6]=20;A[7]=15;A[8]=12;

        printf("Test input data created [%d x %d]\n", ndim, ndim);

    }


    if(vflag)
        save_matrix_csv("A.csv", A, ndim);

    if(pflag)
        display_matrix('A',A,ndim);


    /* Get Platform/Device Information */
    ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
    if (ret != CL_SUCCESS)
    {
```

```
        printf("Error: Failed to get device info!\n");
        return EXIT_FAILURE;
    }
    /* Connect to a GPU compute device */
    ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id,
&ret_num_devices);
    if (ret != CL_SUCCESS)
    {
        printf("Error: Failed to create a device group!\n");
        return EXIT_FAILURE;
    }

    /* Show OpenCL device information */
    printf("=== %d OpenCL device(s) found on platform:\n", ret_num_devices);
      {
       char buffer[10240];
       cl_uint buf_uint;
       cl_ulong buf_ulong;

       ret=clGetDeviceInfo(device_id, CL_DEVICE_NAME, sizeof(buffer), buffer, NULL);
        checkErrors (ret, "clGetDeviceInfo", __LINE__);
       printf("  DEVICE_NAME = %s\n", buffer);

       ret=clGetDeviceInfo(device_id, CL_DEVICE_VENDOR, sizeof(buffer), buffer, NULL);
        checkErrors (ret, "clGetDeviceInfo", __LINE__);
       printf("  DEVICE_VENDOR = %s\n", buffer);

       ret=clGetDeviceInfo(device_id, CL_DEVICE_VERSION, sizeof(buffer), buffer, NULL);
       checkErrors (ret, "clGetDeviceInfo", __LINE__);
        printf("  DEVICE_VERSION = %s\n", buffer);

       ret=clGetDeviceInfo(device_id, CL_DRIVER_VERSION, sizeof(buffer), buffer, NULL);
       checkErrors (ret, "clGetDeviceInfo", __LINE__);
        printf("  DRIVER_VERSION = %s\n", buffer);

       ret=clGetDeviceInfo(device_id, CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(buf_uint),
                           &buf_uint, NULL);
       checkErrors (ret, "clGetDeviceInfo", __LINE__);
        printf("  DEVICE_MAX_COMPUTE_UNITS = %u\n", (unsigned int)buf_uint);

        ret=clGetDeviceInfo(device_id, CL_DEVICE_MAX_CLOCK_FREQUENCY, sizeof(buf_uint),
                           &buf_uint, NULL);
       checkErrors (ret, "clGetDeviceInfo", __LINE__);
        printf("  DEVICE_MAX_CLOCK_FREQUENCY = %u\n", (unsigned int)buf_uint);

        ret=clGetDeviceInfo(device_id, CL_DEVICE_GLOBAL_MEM_SIZE, sizeof(buf_ulong),
                           &buf_ulong, NULL);
       checkErrors (ret, "clGetDeviceInfo", __LINE__);
        printf("  DEVICE_GLOBAL_MEM_SIZE = %llu\n", (unsigned long long)buf_ulong);

      }


    /* Create OpenCL Context */
    context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);
    if (!context)
    {
        printf("Error: Failed to create a compute context!\n");
        return EXIT_FAILURE;
    }

    /* Create command queue */
    command_queue = clCreateCommandQueue(context, device_id, CL_QUEUE_PROFILING_ENABLE,
                                         &ret);
    if (!command_queue)
    {
        printf("Error: Failed to create a command queue!\n");
        return EXIT_FAILURE;
    }

    /* Create Input/Output Buffer Object */
    AUbuff = clCreateBuffer(context, CL_MEM_READ_WRITE, ndim*ndim*sizeof(float), NULL,
              &ret);
    if (!AUbuff)
    {
```

```
    printf("Error: Failed to allocate destination array!\n");
    return EXIT_FAILURE;
}

/* Create Output Buffer Object */
Lbuff  = clCreateBuffer(context, CL_MEM_READ_WRITE, ndim*ndim*sizeof(float), NULL,
                        &ret);
if (!Lbuff)
{
    printf("Error: Failed to allocate destination array!\n");
    return EXIT_FAILURE;
}

/* Copy input/output data to the memory buffer */
ret = clEnqueueWriteBuffer(command_queue, AUbuff, CL_TRUE, 0,
                        ndim*ndim*sizeof(float), A, 0, NULL, NULL);
if (ret != CL_SUCCESS)
{
    printf("Error: Failed to write to source array!\n");
    return EXIT_FAILURE;
}

/* Copy output data to the memory buffer */
ret = clEnqueueWriteBuffer(command_queue, Lbuff, CL_TRUE, 0, ndim*ndim*sizeof(float),
                        L, 0, NULL, NULL);
if (ret != CL_SUCCESS)
{
    printf("Error: Failed to write to source array!\n");
    return EXIT_FAILURE;
}

/* Create kernel program from source file*/
program = clCreateProgramWithSource(context, 1, (const char **)&source_str,
                                    (const size_t *)&source_size, &ret);
if (!program || ret != CL_SUCCESS)
{
    printf("Error: Failed to create compute program!\n");
    return EXIT_FAILURE;
}

/* Build the program executable */
ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
if (ret != CL_SUCCESS)
{
    size_t len;
    char buffer[2048];

    printf("Error: Failed to build program executable!\n");
    clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer),
                        buffer, &len);
    printf("%s\n", buffer);
    return EXIT_FAILURE;
}

/* Create data parallel OpenCL kernel */
kernel = clCreateKernel(program, KERNEL_NAME, &ret);
if (!kernel || ret != CL_SUCCESS)
{
    printf("Error: Failed to create compute kernel!\n");
    return EXIT_FAILURE;
}


/* Get some kernel information */
size_t group_size;
clGetKernelWorkGroupInfo(kernel, device_id, CL_KERNEL_WORK_GROUP_SIZE,
                        sizeof(group_size), &group_size, NULL);
printf("  CL_KERNEL_WORK_GROUP_SIZE = %lu\n", group_size);

/* Set OpenCL kernel arguments */
ret=CL_SUCCESS;
ret|= clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&AUbuff);
ret|= clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&Lbuff);
ret|= clSetKernelArg(kernel, 2, sizeof(int), &ndim);
```

```
    if (ret != CL_SUCCESS)
    {
        printf("Error: Failed to set kernel arguments!\n");
        return EXIT_FAILURE;
    }



#ifdef __APPLE__
    uint64_t         t1, t2;
    t1 = mach_absolute_time();
#endif

    /* Start the timing loop and execute the kernel over several iterations */

    printf("Performing LU decomposition [%d x %d]...\n", ndim, ndim);
#ifndef __APPLE__
    clock_t start = clock();
#endif


    size_t global1[1];
    size_t local1[1];

    size_t localWorkSize1;
    size_t globalWorkSize1;

    for(i=0;i<ndim-1;i++)
    {

        global1[0]=ndim;
        local1[0]=1;

        localWorkSize1 = 25;
        globalWorkSize1 = ( (global1[0]+localWorkSize1-1) / localWorkSize1 ) *
                           localWorkSize1;
        global1[0] = globalWorkSize1;
        local1[0] = localWorkSize1;

       #ifdef _VERBOSE
        printf("i:%d global:%lu  local:%lu\n",i,global1[0],local1[0]);
       #endif

        ret|= clSetKernelArg(kernel, 3, sizeof(cl_uint), &i);
        ret|= clSetKernelArg(kernel, 4, sizeof(cl_float)*ndim, NULL);
        if (ret != CL_SUCCESS)
        {
            printf("Error: Failed to set kernel arguments!\n");
            return EXIT_FAILURE;
        }

        /* Enqueue a kernel run call */
        ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, global1, local1, 0,
                                     NULL, &event[i]);
        checkErrors (ret, "clEnqueueNDRangeKernel", __LINE__);

        ret = clFlush(command_queue);
        checkErrors (ret, "clFlush", __LINE__);

    }

    ret = clWaitForEvents(1, &event[ndim-2]);

    /* Profiling execution time */
    {
        long long start, end;
        double total;

        ret = clGetEventProfilingInfo(event[0], CL_PROFILING_COMMAND_START,
                                      sizeof(start), &start, NULL);
        if (ret != CL_SUCCESS)
            start = 0;

        ret = clGetEventProfilingInfo(event[i-1], CL_PROFILING_COMMAND_END,
```

```
                                        sizeof(end), &end, NULL);
        if (ret != CL_SUCCESS)
            end = 0;

        total = (double)(end - start) / 1e6; /* Convert nanoseconds to msecs */
        printf("Profiling: Total kernel time was %5.2f msecs.\n", total);
    }


    /* Transfer result to host */
    ret = clEnqueueReadBuffer(command_queue, AUbuff, CL_TRUE, 0, ndim*ndim*sizeof(float),
U, 0, NULL, NULL);
    if (ret)
    {
        printf("Error: Failed to read back results from the device!\n");
        return EXIT_FAILURE;
    }

    /* Transfer result to host */
    ret = clEnqueueReadBuffer(command_queue, Lbuff, CL_TRUE, 0, ndim*ndim*sizeof(float),
L, 0, NULL, NULL);
    if (ret)
    {
        printf("Error: Failed to read back results from the device!\n");
        return EXIT_FAILURE;
    }

#ifdef __APPLE__
    t2 = mach_absolute_time();

    /* Calculate execution time */

    struct mach_timebase_info info;
    mach_timebase_info(&info);
    double t = 1e-9 * (t2 - t1) * info.numer / info.denom;
    printf("Elapsed Time:  %.2f s\n", t);

#endif

#ifndef __APPLE__
    clock_t end = clock() ;
    double elapsed_time = (end-start)/(double)CLOCKS_PER_SEC;
    printf("Elapsed time: %lf seconds \n",elapsed_time);
#endif

    /* Display Results */
    if(pflag)
    {
        display_matrix('L',L,ndim);
        display_matrix('U',U,ndim);
    }

    /* Save Results */
    if(vflag)
    {
        save_matrix_csv("L.csv",L,ndim);
        save_matrix_csv("U.csv",U,ndim);
    }

    /* Check Results */
    if(cflag)
    {
        float aij;
        float error=0.0f;
        FILE * pFile;

        pFile = fopen ("M.csv","w");

        printf("Verifying results: A = L * U ...\n");

        for(i=0;i<ndim;i++)
        {
            for(j=0;j<ndim;j++)
            {
                aij=0.0f;
```

```
                for(k=0;k<ndim;k++)
                {
                    aij+=L[i*ndim+k]*U[k*ndim+j];
                }
                error+=fabs(A[i*ndim+j]-aij);
                fprintf(pFile,"%lf;",aij);

            }
            fprintf(pFile,"\n");
        }
        fclose(pFile);

        printf("Total error: %lf\n",error/(ndim*ndim));


    }

    /* Finalization */
    for(i=0;i<ndim-1;i++)
        clReleaseEvent(event[i]);

    ret = clFlush(command_queue);
    ret = clFinish(command_queue);
    ret = clReleaseMemObject(AUbuff);
    ret = clReleaseMemObject(Lbuff);
    ret = clReleaseKernel(kernel);
    ret = clReleaseProgram(program);
    ret = clReleaseCommandQueue(command_queue);
    ret = clReleaseContext(context);

    free(source_str);

    free(A);
    free(U);
    free(L);

    free(event);

    return EXIT_SUCCESS;
}


//
//  utils.c
//  LU decomposition (OpenCL)
//
//  Autor: Javier Vegas Caballero
//  Utilitats per inicialitzar matrius, tractament de l'error.
//  UOC TFC


#include "utils.h"


#define MAX_SOURCE_SIZE (0x100000)

void checkErrors(cl_int status, char *label, int line)
{
    switch (status)
    {
        case CL_SUCCESS:
            return;
        case CL_BUILD_PROGRAM_FAILURE:
            fprintf(stderr, "OpenCL error (at %s, line %d): CL_BUILD_PROGRAM_FAILURE\n",
                    label, line);
            break;
        case CL_COMPILER_NOT_AVAILABLE:
            fprintf(stderr, "OpenCL error (at %s, line %d): CL_COMPILER_NOT_AVAILABLE\n",
                    label, line);
            break;
        case CL_DEVICE_NOT_AVAILABLE:
            fprintf(stderr, "OpenCL error (at %s, line %d): CL_DEVICE_NOT_AVAILABLE\n",
                    label, line);
            break;
```

```
case CL_DEVICE_NOT_FOUND:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_DEVICE_NOT_FOUND\n",
                label, line);
    break;
case CL_IMAGE_FORMAT_MISMATCH:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_IMAGE_FORMAT_MISMATCH\n",
                label, line);
    break;
case CL_IMAGE_FORMAT_NOT_SUPPORTED:
    fprintf(stderr,
      "OpenCL error (at %s, line %d): CL_IMAGE_FORMAT_NOT_SUPPORTED\n", label,
      line);
    break;
case CL_INVALID_ARG_INDEX:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_ARG_INDEX\n",
      label, line);
    break;
case CL_INVALID_ARG_SIZE:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_ARG_SIZE\n",
      label, line);
    break;
case CL_INVALID_ARG_VALUE:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_ARG_VALUE\n",
      label, line);
    break;
case CL_INVALID_BINARY:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_BINARY\n", label,
      line);
    break;
case CL_INVALID_BUFFER_SIZE:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_BUFFER_SIZE\n",
      label, line);
    break;
case CL_INVALID_BUILD_OPTIONS:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_BUILD_OPTIONS\n",
      label, line);
    break;
case CL_INVALID_COMMAND_QUEUE:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_COMMAND_QUEUE\n",
      label, line);
    break;
case CL_INVALID_CONTEXT:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_CONTEXT\n", label,
      line);
    break;
case CL_INVALID_DEVICE:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_DEVICE\n", label,
      line);
    break;
case CL_INVALID_DEVICE_TYPE:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_DEVICE_TYPE\n",
      label, line);
    break;
case CL_INVALID_EVENT:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_EVENT\n", label,
      line);
    break;
case CL_INVALID_EVENT_WAIT_LIST:
    fprintf(stderr,
       "OpenCL error (at %s, line %d): CL_INVALID_EVENT_WAIT_LIST\n", label,
      line);
    break;
case CL_INVALID_GL_OBJECT:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_GL_OBJECT\n",
      label, line);
    break;
case CL_INVALID_GLOBAL_OFFSET:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_GLOBAL_OFFSET\n",
      label, line);
    break;
case CL_INVALID_HOST_PTR:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_HOST_PTR\n",
      label, line);
    break;
```

```
case CL_INVALID_IMAGE_FORMAT_DESCRIPTOR:
    fprintf(stderr,
      "OpenCL error (at %s, line %d): CL_INVALID_IMAGE_FORMAT_DESCRIPTOR\n",
      label, line);
    break;
case CL_INVALID_IMAGE_SIZE:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_IMAGE_SIZE\n",
      label, line);
    break;
case CL_INVALID_KERNEL_NAME:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_KERNEL_NAME\n",
      label, line);
    break;
case CL_INVALID_KERNEL:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_KERNEL\n",
      label, line);
    break;
case CL_INVALID_KERNEL_ARGS:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_KERNEL_ARGS\n",
      label, line);
    break;
case CL_INVALID_KERNEL_DEFINITION:
    fprintf(stderr, "OpenCL error (at %s, line %d):
      CL_INVALID_KERNEL_DEFINITION\n", label, line);
    break;
case CL_INVALID_MEM_OBJECT:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_MEM_OBJECT\n",
      label, line);
    break;
case CL_INVALID_OPERATION:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_OPERATION\n",
      label, line);
    break;
case CL_INVALID_PLATFORM:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_PLATFORM\n",
      label, line);
    break;
case CL_INVALID_PROGRAM:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_PROGRAM\n",
      label, line);
    break;
case CL_INVALID_PROGRAM_EXECUTABLE:
    fprintf(stderr,
      "OpenCL error (at %s, line %d): CL_INVALID_PROGRAM_EXECUTABLE\n",
      label, line);
    break;
case CL_INVALID_QUEUE_PROPERTIES:
    fprintf(stderr,
      "OpenCL error (at %s, line %d): CL_INVALID_QUEUE_PROPERTIES\n",
      label, line);
    break;
case CL_INVALID_SAMPLER:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_SAMPLER\n",
      label, line);
    break;
case CL_INVALID_VALUE:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_VALUE\n",
      label, line);
    break;
case CL_INVALID_WORK_DIMENSION:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_WORK_DIMENSION\n",
      label, line);
    break;
case CL_INVALID_WORK_GROUP_SIZE:
    fprintf(stderr,
       "OpenCL error (at %s, line %d): CL_INVALID_WORK_GROUP_SIZE\n",
      label, line);
    break;
case CL_INVALID_WORK_ITEM_SIZE:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_INVALID_WORK_ITEM_SIZE\n",
      label, line);
    break;
case CL_MAP_FAILURE:
    fprintf(stderr, "OpenCL error (at %s, line %d): CL_MAP_FAILURE\n",
      label, line);
```

```
            break;
        case CL_MEM_OBJECT_ALLOCATION_FAILURE:
            fprintf(stderr,
              "OpenCL error (at %s, line %d): CL_MEM_OBJECT_ALLOCATION_FAILURE\n",
              label, line);
            break;
        case CL_MEM_COPY_OVERLAP:
            fprintf(stderr, "OpenCL error (at %s, line %d): CL_MEM_COPY_OVERLAP\n",
              label, line);
            break;
        case CL_OUT_OF_HOST_MEMORY:
            fprintf(stderr, "OpenCL error (at %s, line %d): CL_OUT_OF_HOST_MEMORY\n",
              label, line);
            break;
        case CL_OUT_OF_RESOURCES:
            fprintf(stderr, "OpenCL error (at %s, line %d): CL_OUT_OF_RESOURCES\n",
              label, line);
            break;
        case CL_PROFILING_INFO_NOT_AVAILABLE:
            fprintf(stderr,
              "OpenCL error (at %s, line %d): CL_PROFILING_INFO_NOT_AVAILABLE\n",
              label, line);
            break;
    }
    exit(status);
}

char *load_program_source(const char *filename, size_t *size)
{
    FILE *fp;
    char *source;

    fp = fopen(filename, "r");
    if (!fp) {
        fprintf(stderr, "Failed to load kernel.\n");
        exit(1);
    }
    source = (char *)malloc(MAX_SOURCE_SIZE);
    *size = fread( source, 1, MAX_SOURCE_SIZE, fp );
    fclose( fp );
    return source;
}

void save_matrix_csv(const char *filename, float *M, int n)
{
    FILE *fp;
    int i,j;

    fp = fopen(filename,"w");
    if (!fp) {
        fprintf(stderr, "Failed to save matrix.\n");
        return;
    }

    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            fprintf(fp,"%lf;",M[i*n+j]);
        }
        fprintf(fp,"\n");
    }

    fclose(fp);
}

void display_matrix(char t, float *M, int n)
{
    int i,j;

    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            printf("%c[%d][%d]:%8.2lf ",t,i,j,M[i*n+j]);
```

```
        }
        printf("\n");
    }
    printf("\n");

}

void create_random_matrix(float *M, int n)
{
    int i,j;

    srand((unsigned int)time(0));

    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
        {
          M[i*n+j]=10.0f*((float)rand()/(float)RAND_MAX)+
               10.0f*((float)rand()/(float)RAND_MAX);

        }

}
```