Paral·lelització amb GPUs d'una aplicació per a química computacional

Albert Herrero Rosello
Màster en Enginyeria Informàtica
Computació d'altes prestacions

Ivan Rodero Castro
Josep Jorba Esteve

25 de juny de 2016

Fitxa del treball final

Títol:	Paral·lelització amb GPUs d'una aplicació per a química computacional
Nom de l'autor:	Albert Herrero Rosello
Nom del consultor/a:	Ivan Rodero Castro
Nom del PRA:	Josep Jorba Esteve
Data de lliurament(mm/aaaa):	06/2016
Titulació o programa:	Màster en Enginyeria Informàtica
Idioma del treball:	Català
Paraules clau	computació, opencl

Resum

El propòsit d'aquesta memòria és explicar el procediment que s'ha seguit en la paral·lelització del software científic PELE que es fa servir en el departament *Life Sciences* del BSC (*Barcelona Supercomputing Center*) per la recerca de nous medicaments. PELE és un software científic que es dedica a busca les contribucions d'energia en el buit i en el medi aquós entres els àtoms de molècules d'un sistema. El software combina algoritmes de predicció d'estructures de proteïnes i tècniques de simulació *Monte Carlo* per tal d'explorar totes les contribucions d'energia.

La primera part del treball explica les tècniques i algoritmes que s'han fet servir en la paral·lelització de l'energia solvent del sistema. L'energia solvent és l'energia que es produeix en transferir una molècula des del buit fins al medi aquós. En aquesta part es presenta la versió seqüencial de l'algoritme i tres versions paral·leles implementades amb *OpenCL*.

La segona part del treball explica el procediment seguit en la paral·lelitació de la funció que calcula l'energia *nonbonding* del sistema. L'energia *nonbonding* és la suma de les energies que interaccionen entre els àtoms d'una molècula en el buit. En aquesta part es presenta la versió seqüencial de la funció i quatre versions paral·leles implementades amb *OpenCL*.

Abstract

The purpose of this report is to explain the procedure in the parallelization of scientist software PELE that is used in the department *Life Sciences* of BSC (*Barcelona Supercomputing Center*) for search new medicines. PELE is a scientific software dedicated to seeking contributions of energy in the vacuum and the aqueous amongst the atoms of molecules of a system. The software combines algorithms for predicting protein structures and simulation techniques *Monte Carlo* to explore all the energy contributions.

The first part of the work explains the techniques and algorithms that have been used in parallelization of solvent energy of the system. The solvent energy is the energy that occurs when transferring one molecule from the vacuum to the aqueous. This section presents the sequential version of the algorithm and three parallel versions implemented with *OpenCL*.

The second part of the work explains the followed procedure in the parallelization of the function that calculates the nonbonding energy in the system. Nonbonding energy is the sum of the energies that interact with the atoms of a molecule in a vacuum. This section presents the sequential version of the function and four parallel versions implemented with *OpenCL*.

Agraïments

Vull dedicar aquest document a totes les persones que m'han ajudat i m'han donat suport durant el desenvolupament del treball. En especial als meus familiars que m'ha donat ànims en tot moment per continuar endavant i acabar la feina. Als amics que han aguantat explicacions tècniques sobre el treball que estava fent sens entendre molt be el que els deia. Als amics que m'han donat forces per tirar endavant en els moments difícils. A Enric Gibert que ha supervisat la feina i m'ha donat diferents punts de vista per encarar els problemes. A Pedro Riera que m'ha ajudat a entendre els conceptes teòrics en els que es fonamenta tot el treball. A Victor Guallar que és la persona que em va posar amb contacte amb l'empresa Pharmacelera. Al BSC per oferir-me l'oportunitat de treballar amb un software científic d'aquestes característiques. A totes ells perquè, sense el seu recolzament, hagués sigut una tasca molt difícil de realitzar. Per tot això, gracies companys. I com sempre dic: en aquesta vida no es té res materialment però, es té tot potencialment.

Índex

Índex de figures

Índex de taules

Capítol 1

Descripció del projecte

1.1 Introducció

La química computacional o química digital és una branca de la química que fa servir ordinadors per ajudar a resoldre problemes químics. Es fan servir els resultats de la química teòrica, incorporant-los en algun software per fer els càlculs de les estructures i les propietats de molècules i cosos sòlids. La química computacional és àmpliament utilitzada en el disseny de nous medicaments i materials.

En aquest sentit, la idea que hi ha al darrere d'aquest treball és la d'accelerar un software científic dedicat a la recerca i disseny de nous medicaments. El software en qüestió es diu *PELE* (acrònim de Protein Energy Landscape Exploration) i és usat pel departament *Life Sciences* del *BSC* (Barcelona Supercomputing Center, apartat 1.2.1).

PELE combina un enfocament de simulació estocàstica (*Montecarlo*) amb algoritmes de predicció d'estructures de proteïnes i és capaç de reproduir amb exactitud processos que necessiten molt temps de calcul amb nomes unes quantes hores de còmput en *CPU*.

La figura 1.1 mostra l'algoritme heurístic per al mètode d'exploració que es basa amb tres passes: un pertorbació inicial, un mostreig de cadena lateral i una minimització final.

1 **Localized perturbation**. Després d'un calcul d'energia per a una estructura inicial, el procediment comença amb la generació d'una pertorbació en el sistema. En estudis de difusió de *lligands*[i], la pertorbació comença amb una translació aleatòria i una rotació del *lligand*. La pertorbació pot incloure tot l'esquelet proteic mitjançant la realització d'una minimització on el carbonis alfa són impulsats a una nova posició resultant d'un petit desplaçament en una baixa freqüència anisotròpica de mode normal (ANM[ii]).

2 **Side chain sampling**. L'algoritme procedeix mitjançant la col·locació de totes les cadenes laterals locals al *lligand* amb una optimització de les cadenes laterals de la biblioteca de *rotámers*[iii] en una resolució dels *rotámers* de 10 graus. L'algoritme de cadena lateral utilitza filtrat estèric i un mètode d'agrupació per reduir el nombre de *rotámers* a minimitzar.

[i] En bioquímica i farmacologia, un lligand és una molècula que s'uneix amb una proteïna per servir a un propòsit biològic.

[ii] El model de xarxa anisotròpic (ANM) és una simple però potent eina feta per a l'anàlisi de mode normal de proteïnes. Es tracta essencialment d'un model elàstic de xarxa per als àtoms Cα (*alpha carbon atoms*) amb una funció de pas per a la dependència de les constants de força en la distancia entre particules.

[iii] Generalment els rotámers es defineixen com conformacions de cadenes laterals de baixa energia. L'ús d'una biblioteca de rotámers permet a qualsevol persona determinar o modelar una estructura per tractar les conformacions de cadenes lateral més probables.

3 **Minimization**. L'últim pas involucra la minimització d'una regió definida per l'usuari amb el mètode de truncat de newton (Truncaded Newton minimizer).

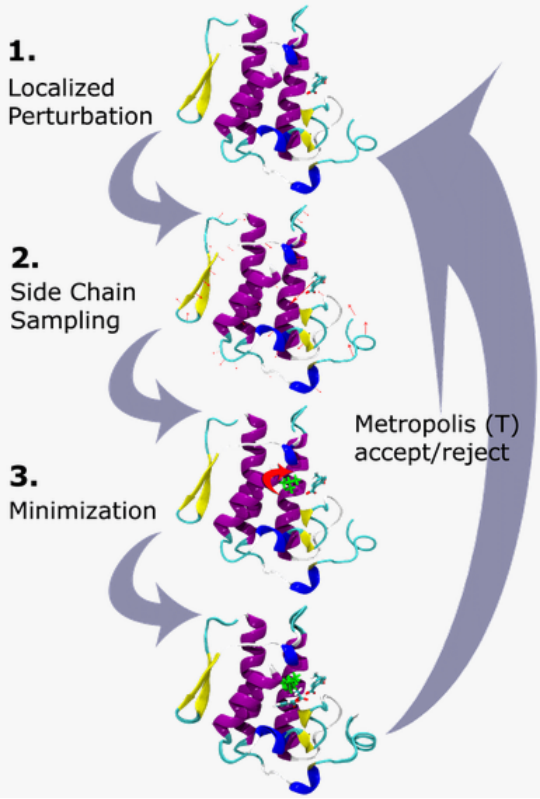


Figura 1.1: Algoritme heurístic d'exploració.

Aquestes tres passes componen un moviment que és acceptat o rebutjat en base al criteri de *Metropolis*[iv].

Per tant, el que es va voler fer des del *BSC* va ser accelerar l'algorisme fent ús de la capacitat de comput que ofereixen les *GPUs* actuals. Per fer-ho es van constituir dos equips que han treballat amb les tecnologies *CUDA* i *OpenCL*.

El primer equip ha estat format per un grup de recerca del BSC i un estudiant de màster de la FIB. El segon equip ha estat format per Enric Gibert de l'empresa *Pharmacelera* (supervisor de la feina) i per l'estudiant de màster en enginyeria informàtica de la UOC, Albert Herrero (autor d'aquest document).

[iv] El propòsit de l'algorisme de Metropolis és generar una col·lecció d'estats d'acord amb una distribució desitjada. Per aconseguir això, l'algorisme utilitza un procés de Markov que cerca asimptòticament una distribució estacionària única. L'enfocament de l'algorisme es basa en dos etapes: una distribució de proposta i una distribució de rebuig o acceptació. La distribució proposada és la probabilitat condicional de proposar un estat x $\rightarrow$ x' donat, i la distribució d'acceptació és la probabilitat condicional per acceptar la proposta per a l'estat x'. L'algoritme es resumeix en cinc passes: triar un estat inicial a l'atzar; triar un estat x' a l'atzar d'acord amb la distribució proposada; acceptar l'estat x' d'acord amb la distribució d'acceptació (si no s'accepta, la transició no té lloc, i es continua amb l'estat inicial); repetir l'algorisme des de la segona passa fins que es generen T estats diferents; es guarda l'estat.

1.2 Actors

En el moment d'elegir el meu treball final de màster tenia molt clar la temàtica que volia fer i on el volia fer. En aquest sentit em vaig adreçar al cap del departament *Life Sciences* del *BSC* per explicar-li les meves intensions i per preguntar-li si tenia projectes per oferir-me. D'aquella primera reunió em vaig posa amb contacte amb diferents professors de la *UPC*, investigadors del *BSC* i amb director d'operacions de l'empresa *Pharmacelera*.

Després d'un seguit de reunions amb cadascun dels contactes vaig decidir fer el treball amb l'empresa *Pharmacelera*. Llavors, aquest treball es tracta d'un projecte realitzat en l'empresa *Pharmacelara* per al *BSC*.

1.2.1 Barcelona Supercomputing Center

El centre de supercomputació de Barcelona és el centre pioner de la supercomputació en Espanya. La seva naturalesa és doble: per una part és un centre d'investigació format per més de tres cents científics, i per l'altra és un centre de serveis de supercomputació per a tota la comunitat científica.

El centre és l'encarregat de gestionar el *Marenostrum*, un dels superordinadors més potents d'Europa, situat a la capella Torre Girona. La missió principal del BSC és investigar, desenvolupar i gestionar tecnologia de la informació amb l'objectiu de facilitar el progrés científic. Amb aquest objectiu, ha pres especial dedicació en àrees com Ciències de la Computació, Ciències de la Vida, Ciències de la Terra i aplicacions computacionals en Ciències i Enginyeria.

Pel que fa al departament de ciències de la vida, integra un programa d'investigació amb diferents investigacions independents liderades per científics d'alt nivell que treballen en diferents aspectes de la biologia computacional, que van des de la bioinformàtica per a la genòmica a la bioquímica computacional. El programa del departament es va originar com un esforç de col·laboració amb altres institucions com *ICREA*[v], *IRB-PCB-UB*[vi] i l'*Institut Nacional de Bioinformàtica*[vii].

El treball, doncs, és objecte d'estudi en la línia d'investigació d'*Electrònica i Modelatge de Proteïnes Atòmiques* en el departament de ciències de la vida. Aquesta línia té com objectiu explorar les respostes químiques i físiques als canvis de configuracions locals i globals que es poden aconseguir mitjançant l'acoblament d'una descripció de la mecànica quàntica del procés reactiu amb tècniques avançades de mostreig. Amb aquesta finalitat s'estudien diversos protocols i algoritmes de simulació, els quals combinen tècniques de mostreig optimitzades amb mètodes *QM/MM*[viii] híbrids i correccions de dinàmica semiclàssica. Els principals objectius es centren en dues àrees:

- La primera àrea es basa en l'aplicació dels mètodes descrits, especialment en interaccions proteïna-substrat i dinàmica de conformació de proteïnes.

[v] Institució Catalana de Recerca i Estudis Avançats, és una fundació finançada pel govern català i està governada pel seu patronat. ICREA va néixer com a resposta a la necessitat de noves fórmules de contractació que permetessin competir en condicions d'igualtat amb altres sistemes de recerca, orientada a la contractació exclusiva del personal científic i acadèmic més extraordinari i amb més talent.

[vi] És un centre d'investigació de classe mundial dedicada a la comprensió de qüestions fonamentals sobre la salut humana i la malaltia. Va ser fundada l'octubre de 2005 pel Govern de Catalunya i la Universitat de Barcelona, i està situat al Parc Científic de Barcelona.

[vii] És una plataforma institucional fundada el 2003 pels majors grups de recerca en bioinformàtica a Espanya. La seva missió primordial és difondre i donar suport a la bioinformàtica per a laboratoris, institucions de recerca i empreses de tot Espanya.

[viii] L'enfocament hibrid *QM/MM* (mecànica quàntica/mecànica molecular) és un mètode de simulació molecular que combina els avantatges de l'enfocament de mecànica quàntica (precisió) i de mecànica molecular (velocitat) per l'estudi dels processos químics en les proteïnes.

- La segona àrea involucra el desenvolupament de nous components metodològics, centrant-se en l'obtenció de la dinàmica de proteïnes per mitjà d'un esquema *Montecarlo*. Aquesta àrea es centra principalment en el desenvolupament continu de l'eina *PELE*.

1.2.2 Pharmacelera

Pharmacelera és una empresa dinàmica i innovadora que proporciona solucions de *hardware* i *software* per al disseny de fàrmacs assistit per ordinador, amb productes com PharmScreen i PharmQSAR. Els productes desenvolupats per Pharmacelera trenquen l'equilibri entre la precisió i el temps (costos computacionals) en les primeres etapes de descobriment de fàrmacs mitjançant el desenvolupament de nous models moleculars i mitjançant l'ús d'acceleradors de maquinari per executar aquests models de manera eficient.

L'empresa està formada per un equip multidisciplinari de professionals amb una trajectòria professional excepcional en els seus respectius camps. Pharmacelera està fortament compromesa amb la innovació per tal de crear un conjunt atractiu de productes i tecnologies que poden ajudar a la indústria farmacèutica en la recerca de les millors molècules candidates en les seves investigacions.

Així, doncs, el treball realitzat s'ha desenvolupant en l'empresa *Pharmacelara* sota la supervisió d'*Enric Gibert*. Enric és el director d'operacions de Pharmacelera i és un enginyer informàtic amb una àmplia experiència en el desenvolupament de programari, en investigació i en innovació gràcies a la seva trajectòria laboral durant anys en els laboratoris d'*Intel*.

Durant la seva estància en Intel, Enric va treballar en la millora del disseny dels processadors de nova generació, en el disseny de solucions *hardware* i *software* i en tecnologies de compilació per millorar la capacitat de programació en plataformes de computació. Ara utilitza tot el coneixement adquirit durant anys per innovar en el camp de la bioinformàtica, amb especial èmfasis en el disseny de fàrmacs assistit per ordinador.

1.3 Estructura de la memòria

En aquest apartat es dóna una visió global del document perquè sigui fàcil de consultar i es pugui saber on trobar cada cosa.

- *Descripció*. En aquest bloc es descriu el treball realitzar. Conté una breu introducció al problema tractat, una posada en context que ens situa en el marc de treball en que s'ha realitzat la feina, la justificació de perquè es va decidir realitzar el treball, l'abast, els objectius i la planificació.
- *Tecnologies*. En aquest bloc es descriuen les tecnologies usades en el desenvolupament del treball.
- *Solvent*. Aquest bloc redacta els passos seguits en la paral·lelització del codi solvent i presenta els resultats obtinguts.
- *Energia Nonbonding*. Aquest bloc redacta els passos seguits en la paral·lelització del codi de la funció d'energia i presenta els resultats obtinguts.
- *Anàlisis econòmic*. En aquest bloc es fa una una breu valoració dels costos que a suposat fer aquest treball: programari usat, hardware i recursos humans.
- *Valoracions i treballs futurs*. Aquí s'exposen les conclusions globals del treball i es presenten futures línies de millorar.

- *Apèndix.* Bloc que conté el codi al que es fa referència en els diferents aparats del document. Els apartats descrits en l'apèndix complementen la part principal del treball.

1.4 Justificació del projecte

Un dels principals motiu pels quals vaig decidir participar en aquest projecte és perquè em donava una gran oportunitat de fer un canvi de 360 graus en la meva carrera professional. Actualment porto 6 anys treballant en empreses de consultoria on vaig començat com a becari, seguint com a programador junior, programador senior y finalment com a analista programador amb Java.

Encara que actualment estic en una posició laboral favorable i estable, sempre he volgut treballar i involucrar-me amb temes de caire acadèmic i d'investigació en l'àrea del *high computing*. En aquest sentint vaig començar a interessar-me en el camp durant els estudis d'Enginyeria Tècnica en Informàtica de Sistemes que vaig cursar en la FIB. En l'acabament dels estudis vaig fer en el treball final de carrera un estudi de rendiment del sistema de cues *Sun Grid Engine* que feia servir el clúster de la UPC. Durant aquest treball vaig començar a veure temes relacionats amb l'alta computació i vaig tenir els meus primers contactes amb les tecnologies MPI i Openmp.

A partir d'aquí vaig decidir continuar estudiant fent el Màster en Enginyeria Informàtica de la UOC i va ser amb l'assignatura de 'Computació d'altes prestacions' on vaig veure que volia continuar per aquest camí.

Aquest treball m'ha ajudant a aprofundir en la tecnologia OpenCL i en veure el potencial que es pot treure de les GPUs. A més a més, gràcies aquest treball he pogut conèixer gent molt qualificada de la que he après molt i he rebut una oferta per realitzar un doctorat industrial entre l'empresa Pharmacelera y la UAB el curs vinent.

1.5 Abast

Quan es parla d'abast es refereix a la suma o col·lecció de productes, serveis i resultats que s'entreguen com a part del projecte, és a dir, el que es farà i el que no es farà. D'aquesta manera es poden definir dos tipus d'abast: abast del projecte i abast del producte.

L'abast del projecte fa referència al conjunt d'entregues que s'han fet durant l'execució de tot el projecte i es divideix en:

- Una entrega inicial que conté una descripció i una planificació del projecte.
- Dos entregues parcials que contenen part del producte.
- Una entrega final que conté el producte, una memòria i una presentació.

L'abast del producte fa referència al conjunt de tasques realitzades en l'execució del projecte i es divideix en:

- Paral·lelitzar el codi *Solvent*.
- Paral·lelitzar el codi *Energia*.
- Presentar els resultats obtinguts de les paral·lelitzacions.
- Descriure possibles millores.

1.6 Objectius

Els principals objectius que es buscaven en la realització d'aquest treball són:

- Augmentar la velocitat d'execució del software *PELE*.
- Millorar el coneixement que es tenia de la tecnologia OpenCL.
- Millorar les aptituds de recerca que es tenien.
- Integrar els coneixements que s'han anat adquirint durant el màster i explotar principalment els coneixements apresos en l'assignatura 'Computació d'altes prestacions'.

1.7 Planificació

La planificació del projecte va ser, d'entrada, una aproximació de la feina que s'havia de realitzar ajustada al temps de dedicació per hores/crèdit acadèmic. En aquest sentit però, no es van tenir en compte els inconvenient ni les dificultats que han anat sorgint durant el desenvolupament, fet que ha obligat a canviar la planificació inicial del treball eliminant el bloc d'anàlisis del codi amb l'eina *CodeXL*, que s'havia planificat en la primera entrega, i augmentant el temps de dedicació en el bloc de l'energia i en l'escriptura i correcció de la memòria. La nova planificació es descriu mitjançant la taula de fites i el seu diagrama de Gant associat.

1.7.1 Taula de fites

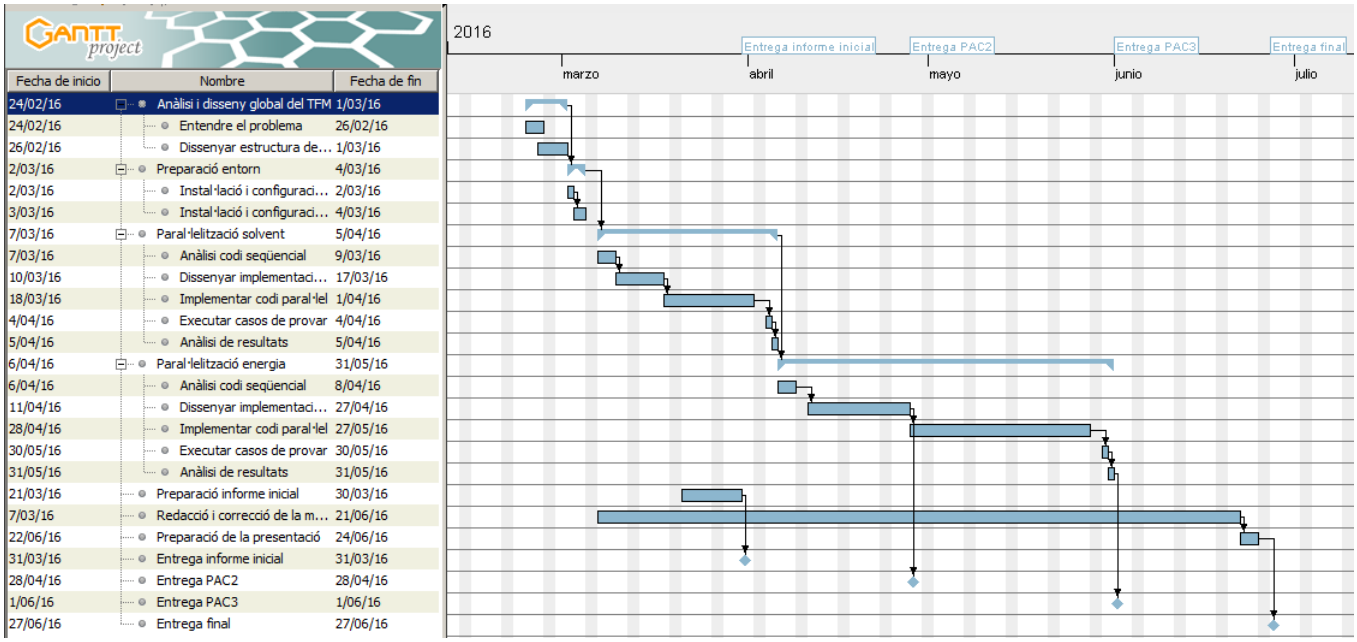
Es presenta un desglòs de les tasques realitzades en la següent taula de fites:

Tasca	**Data Inici**	**Data Fi**	**Jornades**	**Hores**
Anàlisi i disseny global del TFM	**24/02/16**	**01/03/16**	**2**	**16**
Entendre el problema	24/02/16	26/02/16	1	8
Dissenyar estructura del treball	26/02/16	01/03/16	1	8
Preparació entorn	**02/03/16**	**04/03/16**	**1**	**8**
Instal·lació i configuració sistema operatiu	02/03/16	02/03/16	0,5	4
Instal·lació i configuració de OpenCL	03/03/16	04/03/16	0,5	4
Paral·lelització solvent	**07/03/16**	**05/04/16**	**8**	**64**
Anàlisi codi seqüencial	07/03/16	09/03/16	1	8
Dissenyar implementació amb OpenCL	10/03/16	17/03/16	2	16
Implementar codi paral·lel	18/03/16	01/04/16	4	32
Executar casos de provar	04/04/16	04/04/16	0,5	4
Anàlisi de resultats	05/04/16	05/04/16	0,5	4
Paral·lelització energia	**06/04/16**	**30/05/16**	**18,5**	**148**
Anàlisi codi seqüencial	06/04/16	08/04/16	1	8
Dissenyar implementació amb OpenCL	11/04/16	25/04/16	6	48
Implementar codi paral·lel	26/04/16	20/05/16	10,5	84
Executar casos de provar	23/05/16	23/05/16	0,5	4
Anàlisi de resultats	24/05/16	24/05/16	0,5	4
Preparació informe inicial	**21/03/16**	**30/03/16**	**2,5**	**20**
Redacció i correcció de la memòria	**07/03/16**	**21/06/16**	**9**	**108**
Preparació de la presentació	**23/06/16**	**25/06/16**	**2**	**16**
TOTAL	**24/02/2016**	**25/06/2016**	**43**	**380**

Taula 1.1: Taula de fites.

1.7.2 Diagrama de Gantt

El següent diagrama de Gantt representa l'avanç del projecte a traves del temps. El diagrama representa cadascuna de les tasques de la taula de fites.



Fecha de inicio	Nombre	Fecha de fin
24/02/16	Anàlisi i disseny global del TFM	1/03/16
24/02/16	Entendre el problema	26/02/16
26/02/16	Dissenyar estructura de...	1/03/16
2/03/16	Preparació entorn	4/03/16
2/03/16	Instal·lació i configuraci...	2/03/16
3/03/16	Instal·lació i configuraci...	4/03/16
7/03/16	Paral·lelització solvent	5/04/16
7/03/16	Anàlisi codi seqüencial	9/03/16
10/03/16	Dissenyar implementaci...	17/03/16
18/03/16	Implementar codi paral·lel	1/04/16
4/04/16	Executar casos de provar	4/04/16
5/04/16	Anàlisi de resultats	5/04/16
6/04/16	Paral·lelització energia	31/05/16
6/04/16	Anàlisi codi seqüencial	8/04/16
11/04/16	Dissenyar implementaci...	27/04/16
28/04/16	Implementar codi paral·lel	27/05/16
30/05/16	Executar casos de provar	30/05/16
31/05/16	Anàlisi de resultats	31/05/16
21/03/16	Preparació informe inicial	30/03/16
7/03/16	Redacció i correcció de la m...	21/06/16
22/06/16	Preparació de la presentació	24/06/16
31/03/16	Entrega informe inicial	31/03/16
28/04/16	Entrega PAC2	28/04/16
1/06/16	Entrega PAC3	1/06/16
27/06/16	Entrega final	27/06/16

Figura 1.2: Diagrama de Gantt.

Capítol 2

Tecnologies

2.1 PELE

L'obtenció d'un coneixement a nivell molecular de bioquímica de proteïnes i processos biofísics és una tasca complexa que requereix la caracterització de reordenaments conformacionals des de fa molt de temps. Els mètodes de dinàmica molecular, en els quals l'evolució del sistema es projectat com una sèrie d'instantànies resultants de la integració de les equacions clàssiques del moviment, s'han utilitzat amb gran èxit per modelar la reordenació conformacional en biomacromolècules. Aquestes simulacions, però, són sensibles a la mida del sistema i al temps total de propagació, el que requereix recursos computacionals significatius que arriben a vegades a l'ordre de centenars de nanosegons de temps d'execució.

Amb l'ús d'algoritmes de mostreig d'estructura de proteïnes especialitzades, com ara models electrostàtics simplificats, múltiples dinàmiques de rèplica i combinacions de dinàmica molecular i Motecarlo, s'ha desenvolupat un nou enfocament per explorar la dinàmica de les proteïnes. El programa s'anomena *PELE*: *Protein Energy Landscape Exploration*. El mètode utilitza una pertorbació localitzada acoblada a una cadena de predicció i minimització d'algoritmes. El procediment inclou la possibilitat de realitzar càlculs mixtos de mecànica quàntica/mecànica molecular (QM/MM) per actualitzar les càrregues dels lligands complexos o per obtenir estimacions ràpides d'una reacció bioquímica.

PELE, doncs, és un nou enfocament que permet explorar les interaccions energètiques en una proteïna. Aquest procediment incorpora moviments Montecarlo, llibreries de rotàmers de cadenes laterals optimitzades, minimització amb el mètode de truncament de Newton, i proves d'acceptació de Metropolis. El mètode s'utilitza per generar i propagar els canvis en un sistema mitjançant la generació d'una sèrie de mínims locals similars que es combinen en una trajectòria.

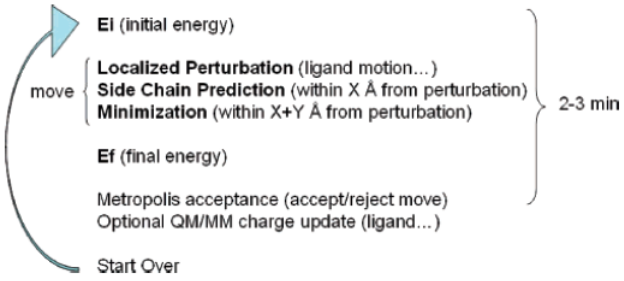


Figura 2.1: Procediment de mostreig usat en PELE.

El procediment de mostreig usat en PELE es basa amb tres passes: un pertorbació inicial, un mostreig de cadena lateral i una minimització final.

- *Local Perturbation.* El procediment usat en la figura 2.1 comença amb la generació d'una pertorbació local del lligand. Molts lligands poden ser tractats com a cossos rígids, llavors per aquest casos sols es necessiten tres graus de llibertat en les rotacions i les translacions. Els lligands flexibles, en canvi, no es poden descriure com a simples unitats rígides. Llavors, la pertorbació d'aquests lligands flexibles inclou graus addicionals de llibertat a partir dels angles diedres d'enllaços rotatius, mentre que les distàncies d'enllaç i angles de flexió es mantenen fixos. En aquest punt, l'usuari ha de proporcionar una llista d'enllaços rotatius per al lligand. A continuació s'apliquen una sèrie de filtres per determinar si hi ha algun contacte estèric entre el lligand i l'esquelet proteic de la proteïna i, en el cas de lligands flexibles, entre el lligand i el mateix. Finalment, si es troba algun tipus de contacte, es rebutja la pertorbació.

- *Side-Chain Sampling.* L'algoritme procedeix mitjançant la col·locació òptima de totes les cadenes laterals locals a l'àtom pertorbat en el primer pas, amb una biblioteca de *rotámers* de cadenes laterals optimitzades amb una resolució dels rotámers de 10 graus. L'algoritme de cadena lateral utilitza filtrat estèric i un mètode d'agrupació per reduir el nombre de *rotámers* a minimitzar. El temps típic que es requereix per a un mostreig de cadenes laterals de entre 20 i 30 residus és de 20 a 40 segons aproximadament.

- *Minimization.* L'últim pas en cada moviment implica la minimització d'una regió que inclou, com a mínim, tots els residus locals als àtoms que participen en els passos 1 i 2. L'algoritme de truncament de minimització de Newton utilitza un protocol multiescalat de 1 a 2 ordres de magnitud més eficient que els enfocaments convencionals. Amb la minimització es pretén generar la resposta a la pertorbació inicial i possibilitar la reordenació de les cadenes laterals en les primeres passes. Aquest enfocament està dissenyat sobre suposició que les cadenes laterals actuen com a sensors de proteïnes que responen als moviments dels lligands, a les interaccions entre proteïnes i a l'activitat bioquímica.

Aquests tres passos componen un moviment que és acceptat (definint un nou mínim) o rebutjat sobre la base d'una criteri Metropolis per a una temperatura donada

$$\begin{array}{c} \triangle V < 0 \\ exp(-\triangle V/K_B T) < R \end{array} \tag{2.1}$$

és a dir, per una disminució de la superfície de potencial, $\triangle V < 0$, o mitjançant el compliment del segon criteri, on K_B és la constant de Boltzmann [i], T és la temperatura escollida per a la simulació i R és un nombre aleatori entre [0, 1].

Tot aquest procediment de mostreig que es fa servir per explorar les internacions energètiques en una proteïna queda resumit en la formula matemàtica descrita en l'equació 2.2. Aquesta formula és el nucli que implementa el software PELE que fan servir en el BSC.

$$\begin{aligned} E_{nb} &= \sum_{bounds} K_r(r - r_{eq})^2 + \sum_{angles} K_\theta(\theta - \theta_{eq})^2 + \sum_{anglei} \sum_{j} k^i_{j,1}[1 + k_{j,2} cos(n_j \phi_i)] \\ &+ \sum_{i<j} [scale_{coulomb} \frac{q^s_i q^s_j}{\varepsilon_{in(ij)} r_{ij}} + \epsilon^s_i \epsilon^s_j ((\frac{\sigma^s_{ij}}{r_{ij}})^{12} - (\frac{\sigma^s_{ij}}{r_{ij}})^6)] f_{ij} \end{aligned}$$

[i] La constant de Boltzmann és la constant física que relaciona la temperatura absoluta i l'energia. S'anomena així en honor al físic austríac Ludwig Boltzmann.

$$
\begin{aligned}
&+ \quad \sum_{i<j}\left(\frac{1}{\varepsilon_{in(ij)}} - \frac{e^{-kf_{GB}}}{\varepsilon_{solv}}\right)\frac{q_i^s q_j^s}{f_{GB}} + \frac{1}{2}\sum_i\left(\frac{1}{\varepsilon_{in(ii)}} - \frac{e^{-k\alpha_i}}{\varepsilon_{solv}}\right)\frac{(q_i^s)^2}{\alpha_i} - \triangle G_{solv,penalty} \\
&+ \quad \sum_i C_{\gamma,i} SASA_i + C_{\alpha,i} S(a/\alpha_i) \\
&+ \quad E_{constraints}
\end{aligned} \tag{2.2}
$$

on,

- El primer terme és la suma de tots els enllaços a la molècula.
- El segon terme és la suma de tots els angles (formats per dos enllaços consecutius) en la molècula.
- El tercer terme és la suma de tots els angles de torsió (també alguns angles de torsió impropis) en la molècula. Per a cada angle de torsió, hi ha una suma de fins a 4 termes, cadascun en funció d'un cosinus relacionat amb l'angle de torsió ϕ_i.
- El quart terme és la suma de totes les interaccions entre parelles d'àtoms (sempre que els àtoms no estiguin connectats per un enllaç, o per dos enllaços de distància; aquestes són les interaccions 1-2 i 1-3, que no es consideren). Inclou una part electrostàtica i una part *van der Waals*. Depenent del camp de força usat, hi ha diferents valors per a $scale_{coulomb}$. Els paràmetres q, σ, ϵ són escalats segons el camp de força utilitzat per adaptar-se a la funció de camp de força específica i les unitats d'energia desitjades (kcal/mol). $f_{i,j}$ és 1.0 per a totes les interaccions excepte per les interaccions 1-4 on pren el valor 0.5.
- El cinquè terme és l'energia de solvatació polar electrostàtica en relació amb les mateixes interaccions que en el quart terme. En alguns models, K és zero, així que el terme Debye $e^{Kf_{GB}}$ és 1. També, depenent del model del solvent, $\epsilon_{in(ij)}$ és diferent de 1.
- El sisè terme és un complement del cinquè, i correspon a la mateixa energia de solvatació polar. Es calcula per a cada àtom en el sistema.
- El setè terme, el *solvent penalty*, depèn del model utilitzat; és una funció polinòmica que depèn dels paràmetres atòmics i la distància interatòmica. Només s'utilitza en el model solvent SGB.
- Els vuitè terme és l'energia de solvatació no polar, i té contribucions de tots els àtoms en el sistema. Aquí apareix la fórmula del model SGB; per al model OBC la formula és similar (encara que computacionalment més ràpida, ja que no necessita $SASA_i$, que és l'àrea exposada d'un àtom, i és en general tan costós de calcular com α_i).
- Els novè i últim terme és per a les constants afegides per l'usuari en diferents fases de la simulació; aquest té forma harmònica (com els termes enllaçats), i és lineal amb el nombre d'àtoms.

2.2 OpenCL

OpenCL és un *framework* de programació paral·lela de software lliure dissenyat per habilitar a les GPUs, i a altres processadors, treballa de manera conjunta amb la CPU i proporcionar, així, més poder de còmput. Com a estàndard, la primera versió d'OpenCL data del 8 de Desembre del 2008 i fou desenvolupada pel grup Khronos.

Amb el *framework* d'OpenCL s'habilita l'ús de les GPUs per a ser usades com a processadors de caràcter general. Juntament amb les CPUs es proporciona un model de programació heterogeni que permet desenvolupar algoritmes que exprimeixen tots els recursos dels que es

disposa. El principal benefici que hi ha en l'ús d'OpenCL és l'acceleració substancial en el processament paral·lel. OpenCL pren tots els recursos computacionals, tals com els processadors multicore, com a unitats de comput i corresponentment assigna diferents nivells de memòria, prenen així avantatge de tots els recursos disponibles en el sistema.

2.2.1 Anatomia de OpenCL

El *framework* de desenvolupament d'OpenCL es divideix en tres parts:

- Especificació del llenguatge.
- API de la capa de plataforma.
- API de temps d'execució.

2.2.1.1 Especificació del llenguatge

L'especifiacació del llenguatge descriu la sintaxis i la interfície de programació per a escriure *kernels* (apartat 2.2.2.2) que es puguin executar en els acceleradors suportats (GPUs, CPUs multicore). El llenguatge de programació OpenCL està basat amb l'especificació de la ISO C99 afegint algunes extensions i restriccions.

2.2.1.2 La capa de plataforma

La capa de plataforma permet als desenvolupadors l'accés a les rutines de software per poder cerca dispositius que suportin OpenCL. Aquesta capa també permet als desenvolupadors usar el concepte del context de *device* per poder seleccionar i inicialitzar dispositius, enviar tasques als *devices* i transferir dades.

2.2.1.3 API en temps d'execució

El framework d'OpenCL fa servir diferents contexts per gestionar els dispositius, per crear objectes com ara cues, objectes de memòria, objectes de kernel, així com permetre l'execució dels kernels en un o més dispositius especificats en el context.

2.2.2 Arquitectura d'OpenCL

2.2.2.1 El model de plataforma

El model de plataforma d'OpenCL està definit com un *host* connectat a un o més dispositius OpenCL. La figura 2.2 mostra el model de plataforma format per un host i múltiples dispositius, on cada dispositiu està format per múltiples unitats de comput i on cada unitat de comput disposa de múltiples elements de processament.

Un host és qualsevol ordinador amb una CPU que executa un sistema operatiu estàndard. Els dispositius d'OpenCL poden ser GPU, DSP[ii], o CPUs multicore. Un dispositiu OpenCL consisteix en una col·lecció d'una o més unitats de comput (cores). Una unitat de comput es composa per un o més elements de processament. Els elements de processament executen

[ii] Un DSP o processador digital de senyals és un sistema basat en un processador o microprocessador que posseeix un conjunt d'instruccions, un hardware y un software optimitzats per a aplicacions que necessitin operacions numèriques a molt alta velocitat. Degut això és especialment útil per al processament i representació de senyals analògiques en temps real.

les instruccions com *SIMD*[iii] (Single Instruction, Multiple Data) o *SPMD*[iv] (Single Program, Multiple Data). Les instruccions *SPMD* típicament són executades en dispositius de propòsit general com les CPUs, mentre que les instruccions *SIMD* requereixen d'un processador de vector com ara les GPUs.

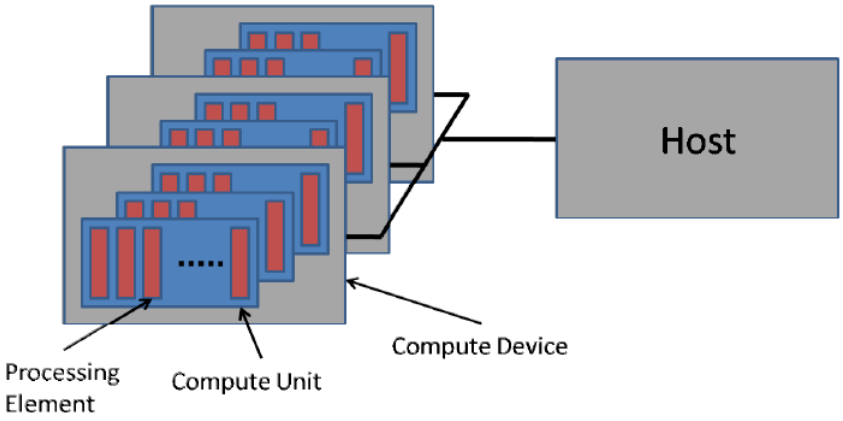


Figura 2.2: Model de plataforma de OpenCL.

La figura 2.3 il·lustra l'arquitectura d'una ATI *Radeon*^{TM} HD 5870 GPU. EL dispositiu està construït amb 20 unitats SIMD, el que es tradueix a 20 unitats de comput en OpenCL.

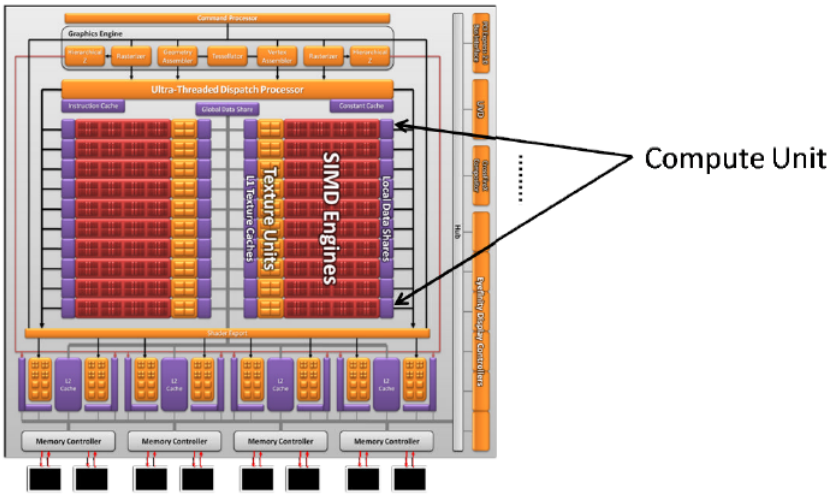


Figura 2.3: Arquitectura d'una ATI *Radeon*^{TM} HD 5870 GPU.

Cada unitat de comput conte 16 cores i cada core emmagatzema cinc elements de processament. Llavors, cada unitat de comput de la ATI *Radeon*^{TM} HD 5870 GPU té 80 (16/5) elements de processament. Les figures 2.4 i 2.5 mostren com està formada una unitat de comput i com està format cada core.

[iii] En computació, SIMD (de l'anglès Single Instruction, Multiple Data) és una tècnica que es fa servir per aconseguir paral·lelisme a nivell de dades. Amb SIMD s'aplica una mateixa operació sobre un conjunt de dades més o menys gran.

[iv] En computació, SPMD (de l'anglès Single Program, Multiple Data) és una tècnica que s'usa per a aconseguir paral·lelisme. Les tasques es separen i s'executen simultàniament en múltiples processadors amb diferents entrades per obtenir resultats amb major rapides.

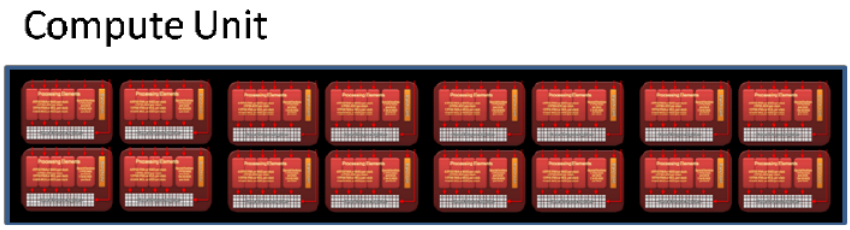


Figura 2.4: Unitat de comput de ATI $Radeon^{TM}$ HD 5870 GPU.

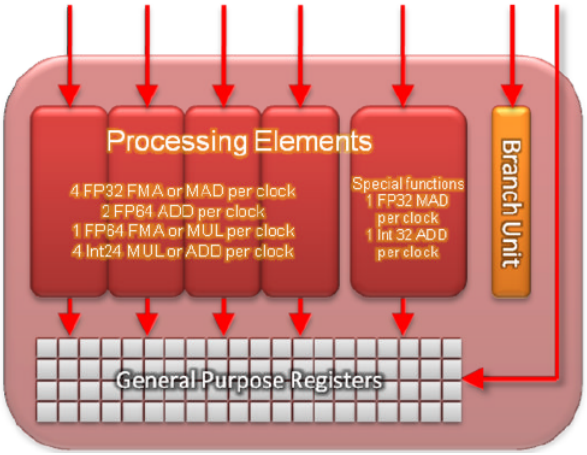


Figura 2.5: Core amb cinc elements de processament.

2.2.2.2 Model d'Execució

El model d'execució d'OpenCL disposa de dos components: els kernels i els programes host. Els kernels són la unitat bàsica de codi que s'executa en un o més dispositius OpenCL. Els kernels són similars a una funció amb C. Els programes host s'executen en el sistema operatiu del host, defineixen els contexts dels dispositius i posen en cua instàncies d'execució de kernels usant cues de comandes. Els kernes es posen en cua de manera ordenada, però es poden executar amb ordre o no.

Kernels .
OpenCl explota la computació paral·lela en dispositius de còmput a partir de definir el problema en un espai N-dimensional. Quan un kernel es posa en cua per a l'execució, des del programa host es defineix un espai d'índexs. Cada element independent de l'execució en aquest espai d'índex s'anomena *work-item*. Cada *work-item* executa la mateixa funció de kernel amb diferents dades. Quan un kernel es posa en cua s'ha de definir un espai d'índexs que indica el nombre total de *work-item* que es necessiten per a l'execució. L'espai d'índex N-dimensional pot ser *N=1,2 o 3*. Per a processar un array lineal de dades es faria servir un valor de N=1; per a processar una imatge es faria servir un valor de N=2, i per a processar un volum 3D es faria servir un valor de N=3.

Per exemple, per processar una imatge de 1024x1024 píxels es podria programar d'aquesta manera: l'espai global d'índex seria un espai de dues dimensions de 1024 per 1024 que consistiria en l'execució d'un kernel per píxel amb un total de 1,048,576 execucions. Dins d'aquest espai d'índex, cada work-item té associat un únic ID global. El work-item per al pixel x=30,

y=22 tindrà un ID global de (30,22).

OpenCL també permet agrupa work-items dins de work-groups, tal com es pot veure en la figura 2.6. La mida de cada work-group està definida pel seu propi espai d'índex local. Tots els work-items dins d'un mateix work-group s'executen junts dins del mateix dispositiu. La raó per a executar-se en un mateix dispositiu és per permetre als work-items compartir memòria local i sincronitzar-se. La sincronització sols és possible entre els work-items dins d'un mateix work-group.

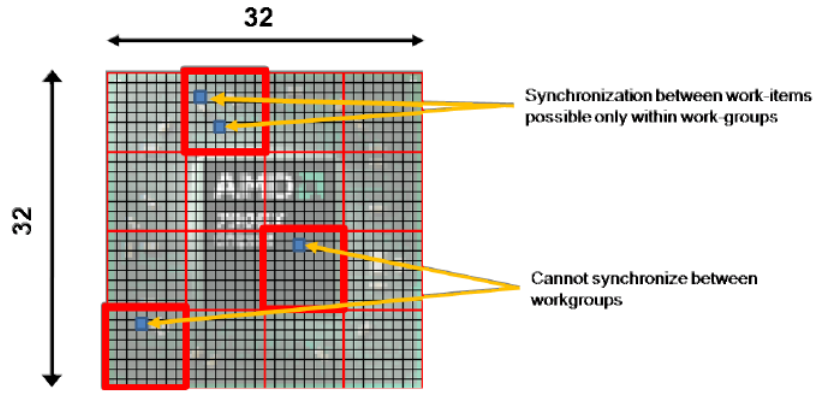


Figura 2.6: Agrupació de work-items dins de work-groups.

La figura 2.7 mostra un exemple d'una imatge en dos dimensions de mida 1024 (32x32). L'espai d'índex està dividit en 16 work-groups. El work-group remarcat té un ID de (3,1) i una mida local de 64 (8x8). El work-item remarcat té un ID local de (4,2), però també es pot accedir a ell a partir del ID global (28,10).

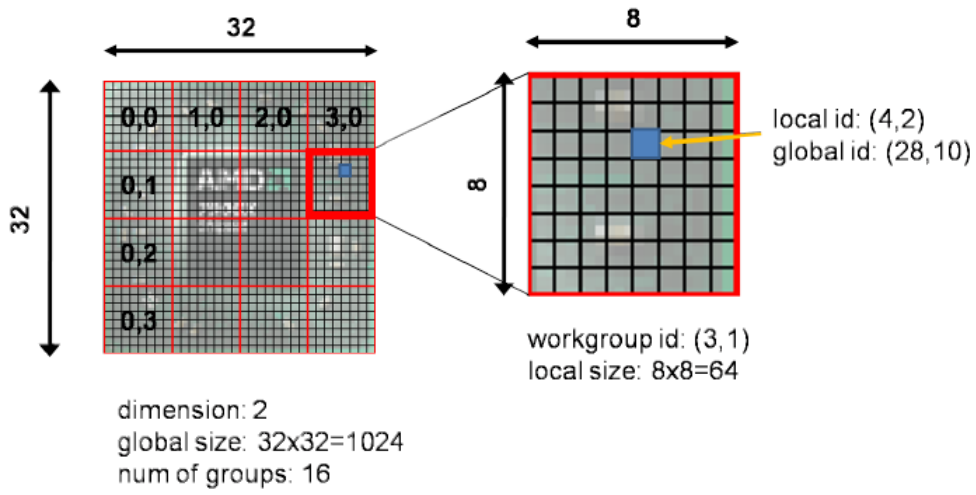


Figura 2.7: Exemple de work-group.

Programa host .
El programa host es responsable de gestionar l'execució dels kernels en els dispositius OpenCL a través de l'ús del context. Usant la API d'OpenCL, el host pot crear i manipular el context per incloure els següents recursos:

- **Devices** - Un conjunt de dispositius OpenCL usat pel host per executar els kernels.
- **Program objects** - El codi del programa que implementa un kernel o una col·lecció de kernels.
- **Kernels** - Les funcions especifiques d'OpenCL que s'executen en un dispositiu.
- **Memory objects** - Un conjunt de buffers de memòria o mapes de memòria comuns entre el host i els dispositius.

Després de crear el context es creen les cues de comandes per a gestionar l'execució dels kernels en els dispositius OpenCL que s'han associat amb el context. Les cues de comandes accepten tres tipus de comandes:

- **Kernel execution commands** - són les comandes per executar un kernel en un dispositiu OpenCL.
- **Memory commands** - serveixen per a transferir objectes de memòria entre l'espai de memòria del host i l'espai de memòria del dispositiu.
- **Synchronization commands** - defineixen l'ordre que s'executen les comandes.

Les comandes s'envien a la cua de manera ordenada però es poden executar en mode ordenadament o desordenat. Quan les comandes s'executen amb el mode ordenat, aquestes són executades en base al criteri FIFO (First in - First out), mentre que si són executades amb mode desordenat, aquestes s'executen en base als criteris de sincronització especificats.

2.2.2.3 Model de memòria

Com que l'espai de memòria comú no està disponible entre el host i els dispositius OpenCL, el model de memòria d'OpenCL defineix quatre regions de memòria accessible pels work-items quan s'executa un kernel. La figura 2.8 mostra les regions de memòria accessibles pel host i els dispositius de comput.

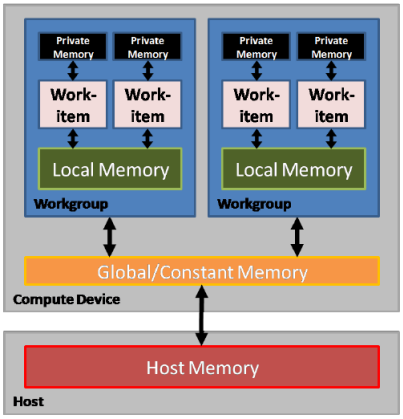


Figura 2.8: Model de memòria OpenCL.

La memòria global és una regió de memòria en que tots els work-items i els work-groups tenen accés de lectura i escriptura tant en dispositius de comput com en el host. Aquesta regió de memòria pot ser assignada solament pel host durant el temps d'execució.

La memòria constant és una regió de la memòria global que es manté constant durant l'execució del kernel. Els work-items sols tenen accés de lectura, mentre que el host té accés de lectura i escriptura.

La memòria local és una regió de memòria usada per a compartir dades entre els work-items en un work-group. Tots els work-items d'un mateix work-group tenen accés de lectura i escriptura.

La memòria privada és una regió que solament és accessible per un work-item.

En la majoria dels casos la memòria del host i la dels dispositius és independent l'una de l'altra. Llavors, la gestió de la memòria ha de ser explicita per permetre compartir dades entre el host i els dispositius. Això significa que les dades s'han de moure explícitament des de la memòria del host cap a la memòria global i cap a la memòria local i viceversa. Aquesta gestió es fa posant comandes de lectura/escriptura a la cua de comandes. Aquestes comandes poden ser bloquejant i no bloquejant. Que siguin bloquejant significa que les comandes de memòria del host esperaran fins que la transacció de memòria hagi acabat per continuar, mentre que no bloquejant significa que el host simplement posa la comanda en la cua i continua sense esperar que la transacció hagi finalitzat.

2.3 Entorn de treball

Tot el treball s'ha dut a terme en una *worksation* que em va oferir el BSC. La worksation disposava de les següents característiques tècniques:

- 16 GB de memòria física.
- Processador Intel(R) Xeon(R) CPU E3-1226 v3 @ 3.30GHz
- Sistema operatiu Ubuntu 14.04.3 LTS
- Targeta gràfica AMD FirePro W5100 4GB GDDR5

La figura 2.9 mostra les especificacions tècniques de la targeta AMD FirePro W5100 4GB GDDR5 obtingudes a partir de la API de OpenCL.

A nivell de software s'ha fet servir:

- El IDE de programació *Netbeans* per a programar tot el codi.
- El programa de calcul *Libre Office Cal* per a crear taules i realitzar càlculs.
- El software *GanttProject* per a crear el diagrama de gantt i per portar un seguiment del treball.
- L'editor de text *LaTeX* per a redactar la memòria.

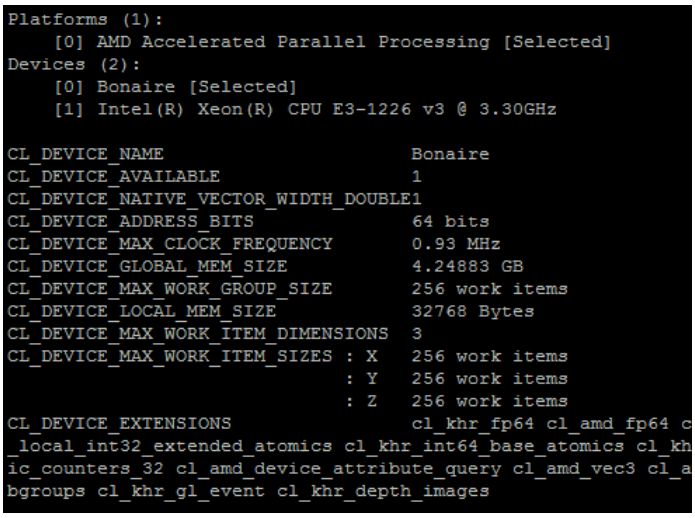


Figura 2.9: Característiques AMD FirePro W5100 4GB GDDR5 .

Capítol 3

Paral·lelització del Solvent

3.1 Descripció del solvent

Els models solvents implícits donen, per a moltes aplicacions, una precisió raonable i un camí computacionalment efectiu per a descriure l'electrostàtica d'un solvent aquós. El desenvolupament del model *OBC Solvent* que implica aquesta part del treball, es basa en la modificació del model *Generalized Born*[i] (GB) per a millorar la seva precisió en el càlcul de la part de la polarització solvent de canvis d'energia lliure en transicions conformacionals a gran escala.

Disposar d'una descripció precisa d'un entorn aquós és essencial per a obtenir simulacions bimoleculars precises, però pot convertir-se amb una tasca molt costosa a nivell computacional. Una alternativa seria l'ús d'un model electrostàtic, el qual reemplaçaria les molècules discretes de l'aigua per un mitjà continu infinit amb les propietats dielèctriques de l'aigua.

Els models continus tenen moltes avantatges sobre la representació explicita de l'aigua, especialment en simulacions de dinàmica molecular (MD):

- El cost computacional associat amb l'ús d'aquests models en simulacions és generalment més petit que el cost de representat l'aigua de manera explicita.
- Els models descriuen la resposta instantània del solvent dielèctric, el qual elimina la necessitat de llargs equilibraments de l'aigua que són típics i necessaris en les simulacions explicites de l'aigua.
- Degut a l'absència de viscositat associada amb l'entorn explicit de l'aigua, la molècula pot explorar més ràpidament tot l'espai conformacional disponible.
- Els models continus corresponen a la solvatació en un volum infinit del solvent, evitant d'aquesta manera possibles artefactes de les interaccions de rèpliques que es produeixen en els sistemes periòdics usats en els càlculs explícits de l'aigua.
- Com que es tenen en compte els diferents graus de llibertat del solvent, l'estimació de les energies de les estructures solvatades és molt més senzill que amb els models d'aigua explícits.

L'energia total d'una molècula solvatada es pot escriure com $E_{tot} = E_{vac} + \Delta G_{solv}$, on E_{vac} representa l'energia de la molècula en el buit i ΔG_{solv} és l'energia lliure de transferir la molècula des del buit fins al solvent.

[i] El model generalitzat de Born és una aproximació a l'equació exacta de Poisson-Boltzmann. Es basa amb el modelatge del solut com un conjunt d'esferes en que la seva constant dielèctrica interna difereix del solvent extern.

Pel que fa al cas que ens interessa, ens tenim de fixar amb l'energia lliure de transferir la molècula des del buit fins al solvent (ΔG_{solv}). Llavors, podem assumir que el calcul de l'energia lliure es pot descompondre amb parts *electrostàtica* i *no electrostàtica*:

$$\Delta G_{solv} = \Delta G_{el} + \Delta G_{surf} \tag{3.1}$$

on ΔG_{surf} és l'energia lliure de solvatar una molècula on totes les seves cargues han sigut eliminades, i ΔG_{el} és l'energia lliure d'esborrar totes les cargues en el buit més la suma de l'energia de connectar-se de nou en un entorn solvent.

L'expressió descrita a sobre és la base de l'esquema *PB/SA* (Poisson-Boltzmann/Accessible surface). Amb aquest esquema, ΔG_{el} es computa mitjançant el mètode numeric *Poisson-Boltzmann*[ii] (PB), i ΔG_{surf} s'agafa com un valor proporcional a tota l'àrea accessible (SA) per la molècula, amb una proporcionalitat constant derivada d'energies experimentals de petites molècules no polars.

Amb el mètode *OBC Solvent*, el que es fa és obtenir un algoritme computacionalment eficient que ajudi a realitzar el calcul de ΔG_{el}. Per fer el calcul de ΔG_{el} es fa servir el mètode *Generalized Born (GB)* com aproximació mitjançant la formula

$$\Delta G_{el} \approx \Delta G_{GB} = -\frac{1}{2}\sum_{ij}\frac{q_i q_j}{f^{GB}(r_{ij}, R_i, R_j)}(1 - \frac{e^{-K f_{ij}^{GB}}}{\epsilon_w}) \tag{3.2}$$

on cada àtom d'una molècula es representa com una esfera de radi ρ_i amb una carga q_i en el centre, r_{ij} és la distancia entre dos àtoms i i j, R_i són coneguts com el *Radi efectiu de Born* dels àtoms i i j, i f^{GB} és una funció amb paràmetres d'entrada r_{ij} i R_i. Una elecció comuna de la funció f^{GB} és

$$f^{GB} = [r_{ij}^2 + R_i R_j exp(\frac{-r_{ij}^2}{4R_i R_j})]^{\frac{1}{2}} \tag{3.3}$$

L'algoritme del *OBC Solvent* és una implementació que millora l'algorisme del model *GB*, el qual anomenem GB^{HCT}. El model GB^{HCT} serveix per a obtenir el calcul del paràmetre R_i de l'equació anterior, el qual es resumeix com

$$R_i^{-1} = \widetilde{\rho}_i^{-1} - I \tag{3.4}$$

on

$$I = \frac{1}{4\pi}\int_{VDW}\theta(|\overrightarrow{r}| - \widetilde{\rho}_i)\frac{1}{r^4}d^3\overrightarrow{r} \tag{3.5}$$

i

$$\widetilde{\rho}_i = \rho_i - 0.09\dot{A} \tag{3.6}$$

El que fa l'algorisme *OBC Solvent*, que anomenem GB^{OBC}, és millorar el calcul del valor R_i mitjançant l'equació

$$R_i^{-1} = \widetilde{\rho}_i^{-1} - \rho_i^{-1} tanh(\alpha\Psi - \beta\Psi^2 - \gamma\Psi^3) \tag{3.7}$$

on

[ii] L'equació de Poisson-Boltzmann és una equació diferencial que descriu interaccions electrostàtiques entre molècules en solucions iòniques. La equació es pot fer servir com a fonament matemàtic del model de la Doble Capa Elèctrica Interracial de Gouy-Chaoman, prposada inicialment per L.G.Gout el 1910 i acabada per Chapman el 1913.

$$\Psi = I\widetilde{\rho}_i \tag{3.8}$$

i α, β i γ són tractats com paràmetres que poden variar per tal d'ajustar la optimització. Aquests paràmetres, en la implementació del *OBC Solvent*, prenen els següents valors:

$$GB^{OBC} : \alpha = 1.0, \beta = 0.8, \gamma = 4.85 \tag{3.9}$$

3.2 Implementació del solvent

Després d'una breu explicació teòrica sobre l'algorisme *OBC Solvent*, es passa a presentar la seva implementació amb el llenguatge de programació C++.

Es presenten quatre versions de l'algorisme:

- **Versió seqüencial**: es tracta de la versió seqüencial de l'algorisme que em va proporcionar el *BSC* i la qual s'ha accelerat amb *OpenCL*.
- **Versió OpenCL**: la versió OpenCL del codi està formada per tres implementacions diferents:
  - *Inner loop*: es tracta d'una primera implementació amb *OpenCL* que paral·lelitza la part interna d'una doble iteració feta amb el bucle *for* de C++.
  - *Outter loop*: es tracta d'una segona implementació amb *OpenCL* que paral·lelitza la part externa d'una doble iteració feta amb el bucle *for* de C++.
  - *Pair loop*: es tracta d'una tercera implementació amb *OpenCl* que paral·lelitza la part interna i externa d'una doble iteració feta amb un bucle *for* de C++.

Per tal d'unificar totes les versions amb un sol codi, s'ha modificat la versió seqüencial que em va proporciona el *BSC* per tal d'obtenir un codi que permeti executar les diferents versions implementades a partir d'un paràmetre d'entrada des de la línia de comandes.

3.2.1 Versió seqüencial

Des del *BSC* se'm va entregar un codi aïllat que representa una extracció de la part del codi original de *PELE* que es fa servir per a calcular els *radis efectius de born*. Juntament amb el codi se'm va entregar una serie de fitxers d'entrada que contenen paràmetres de configuració i varis llistats d'àtoms (les seves estructures) per tal de poder realitzar una sèrie de tests i comprovar que els resultats obtinguts són els mateixos que els s'obtenen amb el codi original de *PELE*.

El paquet entregat conté el següent llistat de classes:

- **Atom**: classe que modela l'estructura d'un àtom. Els àtoms són l'estructura de dades mínima de *PELE*. Aquesta classe actua com un paquet que emmagatzema tots els atributs d'un àtom. La versió que s'entrega és uns versió simplificada per a poder realitzar les diferents proves dels càlculs dels *Radis de Born*.
- **AtomSet**: classe que modela un conjunt d'àtoms, com pot ser una proteïna. La versió entregada és una versió simplificada per poder realitzar les proves de càlcul dels *Radis de Born*.
- **ObcAlphaSasaUpdater**: classe que calcula el valor *alpha* (inversa del radi de born) d'un àtom. Aquesta és la classe principal que conté la doble iteració de cost logarítmic $O(N^2)$ que s'ha de paralitzar. La classe conté el mètode *updateAlphas* que és una extracció exacta del codi original de *PELE*. L'algoritme usat en el mètode *updateAlphas* segueix el model GB^{OBC} explicat en el punt 3.1.

- **PeleBuildException**: classe que hereta de *std:runtime_error* i que controla qualsevol excepció que pugui aparèixer en el programa.
- **SolventTemplateReader**: classe que llegeix i guarda la informació relativa als àtoms que hi ha en els fitxers de configuració. La informació emmagatzemada es usada per les altres classes del programa.
- **StringTools**: classe que conté un conjunt d'utilitats.

El diagrama de classes associat al programa seqüencial que se'ns va entregar és el següent:

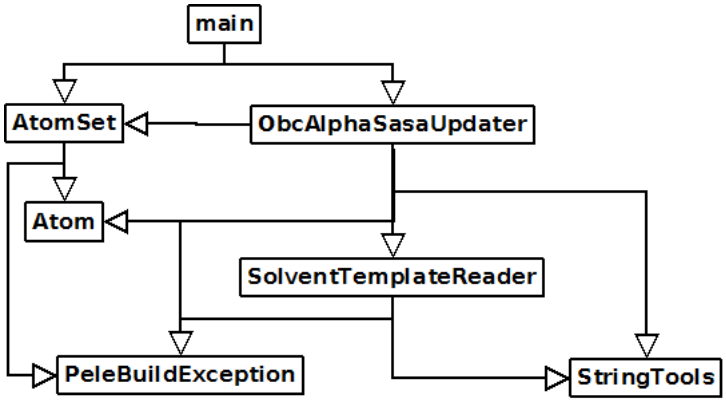


Figura 3.1: Diagrama classes seqüencial

Si es desitja es pot veure el contingut dels fitxers '*.h*' que defineixen l'estructura de les classes en l'apèndix A.1.

Juntament amb el conjunt de classes se'ns va entrega el fitxer *main.cpp* que defineix el punt d'entrada al programa i mostra com es llegeixen les dades per a poder executar els testos. La implementació del fitxer s'adjunta en l'entrega del treball.

A part dels fitxers de codi, també tenim els fitxers de dades de test que ens permetran comparar els resultats obtinguts amb els que s'obtenen des de *PELE*. Els fitxers mostren l'estructura de 3 proteïnes de diferent mida:

- Hexapeptide de 96 àtoms amb 6 residus. L'estructura està en el fitxer *plop_isr_out.pdb*. Les dades per poder executar els càlculs estan en el fitxer *plop_isr_out_amber99sb_input.txt*.
- Proteïna *1AAR* que conté 1231 àtmos amb 76 residus. L'estructura de la proteïna està en el fitxer *1AAR_76_res_min.pdb*. Les dades per poder executar els càlculs estan en el fitxer *1AAR_76_res_min_amber99sb_input.txt*.
- Proteïna *3QLQ* que conté 14132 àtoms amb 948 residus. L'estructura de la proteïna està en el fitxer *3QLQ_237_res_min.pdb*. Les dades per poder executar els càlculs estan en el fitxer *3QLQ_237_res_min_amber99sb_input.txt*.
- Proteïna *3UTG* que conté 31316 àtoms amb 2000 residus. L'estructura de la proteïna està en el fitxer *3UTG_506_res_impact_min.pdb*. Les dades per poder executar els càlculs estan en el fitxer *3UTG_506_res_impact_min_amber99sb_input.txt*.

A més a més, també s'adjunta el fintxer *solventParamsHTCOBC.txt* que conté els paràmetres *OBC* necessaris per poder realitzar els càlculs.

Tots els càlculs dels fitxers de referència s'han fet utilitzant el camp de força AMBER99sb (AMBER99[iii] amb correccions sobre l'esquelet proteic).

En l'apèndix A.2 es presenta el codi de la funció *updateAlphas()* que implementa la formula presentada en l'equació Eq.(3.7).

Si analitzem el codi de *updateAlphas()* podem veure com es fa ús dels valors α, β i γ definits en la Eq. (3.9) (línies 3-5). També s'observa la implementació de Eq. (3.7) (línies 11-15), on el valor Ψ es calcula amb el mètode *computeOtherAtomsContributionToIthAtom* i es guarda en la variable *sum*.

El valor obtingut de *atomsBornRadiiOffset[i]* (línia 15) fa referència al valor $\widetilde{\rho}_i^{-1}$ de Eq. (3.7) i es calcula en la funció *setAtomsObcProperties()* de la classe a partir del valor ρ_i^{-1} que s'obté com a paràmetre d'entrada en la lectura del fitxer *solventParamsHTCOBC.txt*. El valor que s'obté amb $atoms[i] -> gbr$ (línia 16) fa referència a ρ_i^{-1}.

En l'apèndix A.3 es presenta la funció *computeOtherAtomsContributionToIthAtom* que realitza el calcul del paràmetre Ψ. Si mirem el codi veiem que no es correspon amb Eq. (3.5). Això és així perquè la idea de la integral és obtenir una aproximació de la densitat d'energia sobre tota la regió molecular, i aquest calcul s'obté amb l'expressió que involucra la suma sobre parelles d'àtoms.

Analitzant els dos mètodes (*updateAlphas()* i *computeOtherAtomsContributionToIthAtom*) podem veure que es fan parelles d'àtoms, de manera que la complexitat algorítmica és de $O(N^2)$, motiu pell qual des del *BSC* es va decidir paral·lelitzar el codi per tal de aconseguir una millora en els temps dels càlculs.

3.2.2 Versions OpenCL

Les següents versions del codi implementen la paral·lelització del codi GB^{OBC} amb *OpenCL*. A l'hora d'implementar-les s'ha intentat seguir el mateix estil de programació que el codi original, i l'estructura dels fitxers i el nom de les variable s'han modificat poc.

La següent taula mostra els fitxers que té la versió original i els fitxers que s'han incorporat en la versió paral·lela amb *OpenCL*:

Fitxers Originals (.cpp i .h)	Fitxers nous
Atom	basic
AtomSet	cmdparser
ObcAlphaSasaUpdater	oclobject
PeleBuildException	SolventMain
SolventTemplateReader	obc_solvent_kernels.cl
stringTools	
Main	

Taula 3.1: Fitxers del programa

[iii] AMBER (de l'anglès Assisted Model Building with Energy Refinement) és una familia de camps de força de dinàmica molecular desenvolupat originalment pel grup Peter Kollman's de la Universitat de Califòrnia, San Francisco. AMBER és també el nom del paquet de programari de dinàmica molecular que simula aquests camps de força.

Els fitxers *basic*, *cmdparser* i *oclobject* són tres fitxers que proveeix *Intel* com a *open source* per a facilitar l'ús de les abstraccions de *OpenCL*.

En les implementacions amb *OpenCL* la major part del codi original del fitxer *main.cpp* s'-ha mogut cap a la classe *SolventMain*, la qual executa les diferents versions paral·leles (inner loop, outter loop i pair loop), la versió seqüencial i compara els seus temps d'execució. Els fitxers *.cpp* i *.h* configuren els *Kernels* per a cadascuna de les versions. Els kernels es troben implementats en el fitxer *obc_solvent_kernels.cl* i el seu punt d'entrada des del host es troba en el fitxer *ObcAlphaSasaUpdater*.

Amb la introducció dels nous fitxers de codi, el diagrama de classes del programa ens queda:

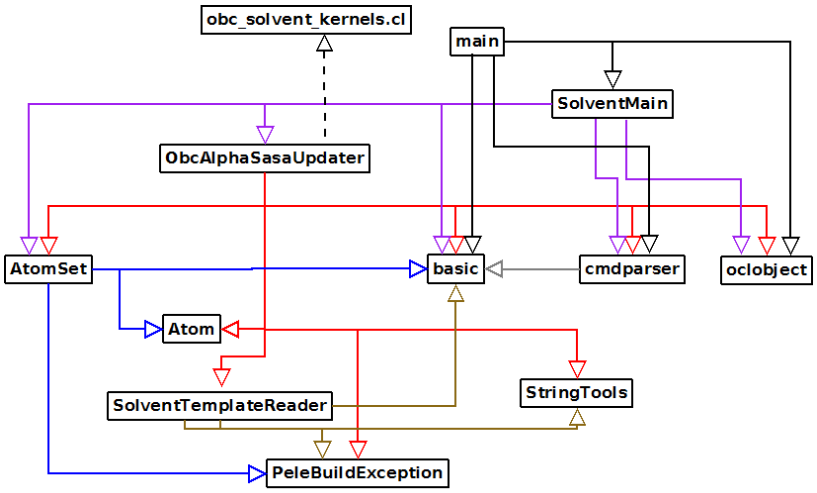


Figura 3.2: Diagrama classes OpenCL

A l'hora d'implementar el codi s'ha assumit que el *host* (CPU) i el *device* (hardware accelerador) no comparteixen memòria, com és el cas de la gran majoria de les configuracions de targetes gràfiques d'avui dia. Per tant, les dues entitats es comuniquen entre si a través de buffers. La següent figura resumeix els buffers de comunicació:

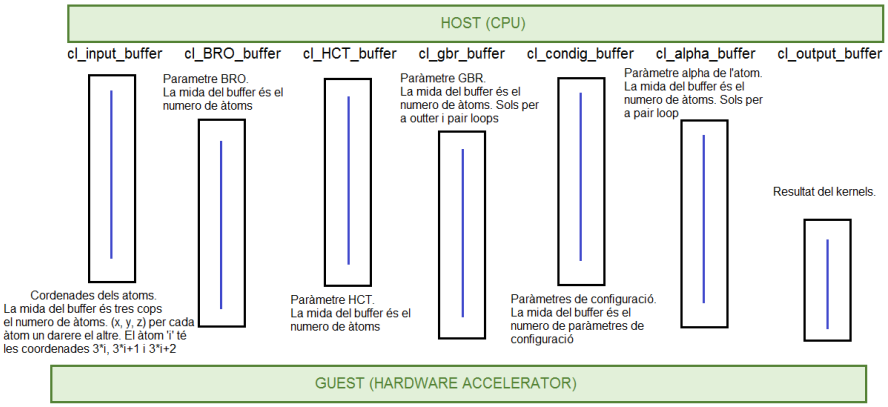


Figura 3.3: Buffers OpenCL

En l'apèndix A.4 es presenta l'estructura del fitxer *.h* que defineix a la classe *SolventMain*. No es presenten els fixes *.h* de les classes *basic*, *cmdparser* i *oclobject* perquè són implementacions obtingudes des de *Intel* i no s'han tingut de modificar en cap moment. Si es desitja veure els fitxers de definició o els fitxers d'implementació de les classes es pot fer des del paquet de codi que s'entrega.

3.2.2.1 Inner Loop

La primera versió paral·lelitza el bucle que hi ha en el mètode *ComputeOtherAtomsContributionToIthAtom*. En altres paraules, el mètode *updateAlphas* s'ha replicat com *updateAlphasInnerLoopCL* que és molt similar al original i que s'executa en la CPU. No obstant, aquest mètode crida a *computeOtherAtomsContributionToIthAtomCL*, el qual executa la contribució de cada parella d'àtoms en un *kernel* de manera paral·lela. El kernel s'anomena *OBCInnerLoopKernel*.

Si s'observa el codi de l'apèndix A.5 es pot veure que les diferencies que hi han amb el de la versió seqüencial són la crida al mètode *computeOtherAtomsContributionToIthAtomCL* (línia 10) i el bloc (línies 11-23) que calcula el temps total d'execució del *kernel*.

En l'apèndix A.6 es presenta el codi de la nova implementació del mètode *computeOtherAtomsContributionToIthAtom* que executa la crida al *kernel OBCInnerLoopKernel*. El més important del codi són les línies 24-30 en que es sumen totes les contribucions de cada parella d'àtoms *atomIndex-j*. El calcul d'aquestes contribucions es fa en paral·lel en el kernel i es guarda en el buffer de sortida cl_output_buffer.

La idea que hi ha al darrera d'aquesta implementació és la d'executar el bucle exterior *numAtoms* vegades i executar amb paral·lel les contribucions de les *numAtoms - 1* parelles amb l'àtom actual. Per fer-ho es declara un *Range* de kernel amb *global size* igual al número d'àtoms i amb *local size* el màxim permès per la targeta gràfica (256 en la nostra targeta). Amb això s'aconsegueix reduir el cost logarítmic $O(N^2)$ de la versió seqüencial a *O(N)*.

En l'apèndix A.7 es pot veure el codi del kernel que executa el bucle interior. L'important del kernel es la crida al mètode *ComputeContributionFromAtom* (línia 8) que és l'encarregada d'implementar el codi del bucle interior. El codi del mètode *ComputeContributionFromAtom* es pot veure en l'apèndix A.8. Si mirem el codi es pot veure que el mètode *ComputeContributionFromAtom* executa exactament el mateix que en la funció *computeOtherAtomsContributionToIthAtom* del codi seqüencial.

3.2.2.2 Outter Loop

La segona versió paral·lelitza el bucle que hi ha en el mètode *updateAlphas* (A.2). El mètode *updateAlphasOutterLoopCL* és el punt d'entrada cap a la paral·lelització del codi que es troba en el *kernel OBCOutterLoopKernel*.

En l'apèndix A.9 es presenta el codi de la funció *updateAlphasOutterLoopCL*. L'important del codi, com en el cas del mètode *computeOtherAtomsContributionToIthAtomCL*, són les línies 16-18 on l'atribut *alpha* de l'àtom pren el valor del buffer de sortida del kernel on es guarda el resultat.

La idea que hi ha darrera d'aquesta implementació és executar el bucle exterior en paral·lel en el kernel, de manera que cada thread executara el bulce intern. Igual que en el cas del *inner loop*, per fer-ho es declara un *Range* de kernel amb *global size* igual al número d'àtoms i amb *local size* el màxim permès per la targeta gràfica (256 en la nostra targeta). Amb això s'aconsegueix reduir el cost logarítmic $O(N^2)$ de la versió seqüencial a *O(N)*.

En l'apèndix A.10 es pot veure la implementació del kernel *OBCOutterLoopKernel*. El més important del kernel són les línies 12-15, on es veu que cada thread calcula la contribució de l'àtom actual (atom_index) contra tots els demés.

3.2.2.3 Pair Loop

La tercera versió paral·lelitza els dos bucles i és més complexe que la de les versions *inner* i *outter*. El mètode *updateAlphasPairLoopCL* és el punt d'entrada cap a la paral·lelització del codi que es troba en els *kernels* *OBCPairLoopKernel* i *updateAlphas_kernel*. En l'apèndix A.11 es pot veure la implementació del mètode *updateAlphasPairLoopCL*. El que destaca del mètode són les línies 8-18 on es llencen dos kernels, i les línies 21-24 on és recupera el valor *alpha* que s'ha calculat.

La idea que hi ha al darrera d'aquesta implementació és fer que el primer kernel sigui l'encarregat de calcular les contribucions de cada àtom amb els demés, és a dir, sigui l'encarregat de calcular el bucle interior de manera paral·lela. Un cop es tenen les contribucions de cada àtom amb la resta s'executa el segon kernel que s'encarrega de realitzar els càlculs del bucle extern. A diferència dels casos anteriors on el *Range* del kernel tenia una mida global de *numAtoms*, i on cada thread era vist com un àtom, en aquest cas es té que la mida global és de *numAtoms * localSize*, on cada grup equival a un àtom.

En el primer kernel (apèndix A.12) el que s'aconsegueix és calcular les contribucions de cada àtom amb els demès de manera paral·lela. En aquest cas es té que cada àtom és un grup de mida *localSize*, de tal manera que es fan servir els *localSize* threads del grup per calcular les contribucions que té l'àtom amb els demés àtoms. Una manera fàcil de dir-ho és que el thread '*j*' del grup '*i*' calcula les contribucions de l'àtom '*i*' amb els àtoms '*i+2*', '*i+4*' i '*i+6*' mentre que el thread '*j+1*' del grup '*i*' calcula les contribucions de l'àtom '*i*' amb els àtoms '*i+1*', '*i+3*' i '*i+5*'.

Per a poder entendre millor aquest algorisme es presenta el següent exemple pràctic on es veu el seu funcionament. Imaginem l'escenari:

- Disposem d'un conjunt de 8 àtoms
- El nostre *localSize* és de mida 2.
- El nostre *globalSize* és de mida *localSize * numAtoms*

Amb aquesta configuració obtenim un *range* amb 8 grups (un per àtom) de mida 2, és a dir, un total de 16 threads. Si executem la primera part de l'algorisme de l'apèndix A.12 (línies 1-31) obtenim el següent resultat:

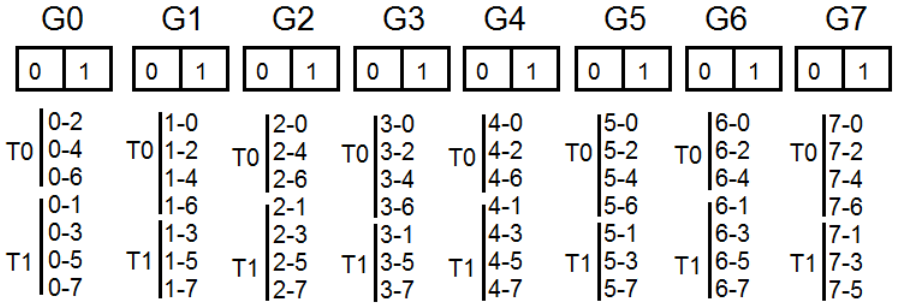


Figura 3.4: Exemple pràctic

En la figura es poden veure 8 grups (GX on $0 <= X < 8$) amb dos threads cada grup. En aquest cas, si executem l'algorisme del primer kernel obtenim que cada thread exectuta les contribucions indicades i les guardara en la variable *sum*. Per exemple, en el cas del grup 0, el thread 0 calcularà les contribucions entre els àtoms 0-2, 0-4 i 0-6 mentre que el thread 1 calcularà les contribucions dels àtoms 0-1, 0-3, 0-5 i 0-7.

Un cop els threads han acabat el calcul de totes les contribucions tenim que cada grup GY (on $0 <= Y < numAtoms$) té la contribució del àtom Y repartida entre tots els threads del grup. Ara, doncs, el que fa falta és fer la suma total de les contribucions de cada thread dins d'un grup. Això s'aconsegueix fen un *reduce* (línies 32-48). La tècnica usada per fer la reducció consisteix en anar dividint la mida del grup entre dos i sumar les dues parts fins que s'aconsegueix acumula la suma total. Per fer-ho però, és necessari disposa d'un array en memòria local on guardar la contribució calculada per cada thread (línies 4 i 36).

Finalment es guarda la contribució total de cada àtom en l'array '*contributions*' alocatada en memòria global. Aquest array de sortida del primer kernel és un array d'entrada al segon kernel.

En el segon kernel (apèndix A.13) s'executa el codi del bucle exterior. Si s'observa el codi es pot veure que és fidel al de la versió seqüencial. Com a punt a destacar està la línia 6 on s'agafen les contribucions de cada àtom del array '*contributions*' que s'havien calculat en el primer kernel.

3.3 Resultats

En aquest punt es presenten els resultats obtinguts de l'execució de la versió seqüencial i les versions OpenCL implementades. Per tal d'obtenir uns resultats significatius s'ha seguit el següent criteri en el moment d'agafar els temps del calcul:

- S'han fet 5 execucions de cada versió de l'algorisme.
- S'ha eliminat el mínim i el màxim temps obtinguts.
- S'ha fet la mitjana aritmètica dels tres temps restants.

Es presenten dos modes d'execució de les versions, la primera amb el flag *O0* i la segona amb el flag *O3*. Aquests flags són flags de compilació que té el compilador de *C++*. Amb el segon flag sé li diu al compilador que optimitzi el codi que s'executa en la part del host.

La següent taula presenta els temps obtinguts en ambdós modes d'execució per a la versió seqüencial i les versions implementades amb OpenCL.

			Temps d'execució (s)	
Input	**Nº Àtoms**	**Versió**	**O0**	**O3**
Mitjà	4358	Inner	0,6567356667	0,6100453333
		Outter	0,1183426667	0,104402
		Pair	0,0947346667	0,0805280333
		Seqüencial	1,7215766667	1,12518
Gran	31314	Inner	14,4947	12,6188333333
		Outter	3,9166866667	3,8601866667
		Pair	3,6617133333	3,6148233333
		Seqüencial	83,7464666667	52,0911666667

Taula 3.2: Taula temps OBC Solvent.

Analitzant la taula es poden veure varies coses: per una banda es veu que -independentment del flag de compilació i de la mida de l'imput- totes les versions amb OpenCL són més ràpides que la versió seqüencial; d'altra banda també es pot veure que amb el flag de compilació *O3* totes les execucions són més ràpides que les del flag *O0*, cosa que ja s'esperava, ja que el flag *O3* indica al compilador que optimitzi el codi en la part del host; per últim, es pot veure que per a ambdós imputs i ambdós flags de compilació la versió més ràpida és la *Pair Loop*.

La següent taula mostra els *speedups* resultants de dividir el temps seqüencial amb els temps obtinguts en les versions de OpenCL.

			Speedup	
Input	**Nº Àtoms**	**Versió**	**O0**	**O3**
Mitjà	4358	Inner	2,6214149072	1,8444203054
		Outter	14,5473878117	10,7773797437
		Pair	18,1726154452	13,9725255098
Gran	31314	Inner	5,7777302508	4,1280493442
		Outter	21,3819674112	13,4944683159
		Pair	22,8708418828	14,4104322295

Taula 3.3: Taula speedups OBC Solvent.

Analitzant la taula podem veure que per a les versions *Outter* i *Pair* s'han obtingut uns speedups superiors a **10x**. Per al cas del BSC, l'*speedup* que els interessa és el que s'obté amb l'imput mitjà de 4358 àtoms. Aixi, doncs, per aquest cas, podem veure que per a la versió *Pair* s'ha obtingut un speedup aproximat de **14x** de millora en el rendiment quan es compila amb el flag *O3*.

Capítol 4

Paral·lelització de la funció d'energia

4.1 Energia nonbonding

En química i bioquímica les molècules s'atreuen unes a altres sense formar enllaços químics. Quan aquesta atracció és més forta que l'energia cinètica mitjana de la molècula, com succeeix per a la majoria de molècules, les molècules s'agreguen i formen una fase condensada (líquid, sòlid, etc.).

La majoria de les forces intermoleculars, o *forces nonbonded*, són electrostàtiques, el que significa que els potencials electrostàtics al voltant d'una molècula són una bona mesura de les forces *nonbonded* i es poden utilitzar per a estudiar les interaccions *nonbonded* entre molècules.

Tot i que la majoria de *forces nonbonded* són electrostàtiques, als químics els agrada distingir entre diferents tipus de forces d'acord amb els tipus de càrregues involucrades. Les *forces nonbonded* més fortes ocorren quan les dues molècules contenen càrrega permanent, o càrrega parcial, és a dir, àtoms capaços de generar grans potencials electrostàtics. Les forces d'aquests tipus s'anomenen forces ió-ió, ió-dipol i dipol-dipol[i].

Les forces electrostàtiques són proporcionals a la càrrega, per tant, les forces d'ió-ió normalment són les més fortes en aquest grup i les forces dipol-dipol són les més febles. No obstant, a vegades hi ha variacions grans i inesperades degut a que el potencial també es veu afectat pel radi atòmic i per àtoms veïns.

Una força important és la que actua entre un ió (o dipol) i una molècula no polar polaritzable (això significa que el núvol d'electrons d'una molècula canvia de forma en resposta a forces electrostàtiques). L'enfocament d'un ió (o dipol) carregat permanentment indueix a canvis temporals en la densitat del núvol d'electrons d'una molècula polaritzable, fet què condueix a forces anomenades ió-induïdes i dipol-induïdes.

Un altre tipus de força, coneguda com la *força de Van der Waals*[ii] o *força de dispersió de London*, és la que es crea cada cop que dues molècules s'aproximen, independentment de la seva

[i] *Dipol* significa qualsevol molècula neutra que conté àtoms parcialment carregats capaços de generar grans potencials.

[ii] En fisicoquímica, les forces de Van der Waals o interaccions de Van der Waals, són les forces d'atracció o repulsió entre molècules (o entre parts d'una mateixa molècula) diferents a aquelles degudes a un enllaç intramolecular (enllaç iònic, enllaç metàl·lic i enllaç covalent de tipus reticular) o a la interacció electrostàtica de ions amb altres o amb molècules neutres.

polaritat. Les forces de *Van der Waals* són forces electrostàtiques dèbils creades per canvis momentanis en la distribució de la càrrega d'una molècula i que no poden ser avaluades usant els potencials elèctrics. Aquestes són les forces més dèbils de totes les forces *nonbonded* i sols actuen en un rang de distància molt curta, de manera que sovint queden emmascares per altres forces.

En aquest sentit, el software *PELE* té la missió de calcular l'energia *nonbonded* entre parells d'àtoms en el sistema. Per fer-ho però, *PELE* fa servir com a optimització el següent criteri: els àtoms que estan separats a una gran distància tenen una interacció molt petita i són ignorats; aquest criteri es gestiona mitjançant una llista de parells *nonbonding* entre àtoms propers que tinguin un interacció significativa en el sistema.

En resum, el software *PELE* es dedica a trobar els valor mínims per una funció donada f(). Aquesta funció té moltes variables d'entrada, per tant no és representable gràficament, ja que cada variable afegeix una nova dimensió a la gràfica. La funció que minimitza PELE no és arbitraria: en el seu lloc, el que es vol minimitzar és l'energia del sistema. L'input de la funció d'energia són els àtoms del sistema, més concretament les seves posicions i les seves càrregues, i l'output, l'energia del sistema. La funció d'energia és en realitat la suma de tres factors independents entre ells:

```
EnergiaTotal(a1, a2, a3, ..., an) =
                      EnergiaCovalent(a1, a2, a3, ..., an)
                    + EnergiaNonBonding(a1, a2, a3, ..., an)
                    + EnergiaSolvent
```

Per tant, sempre que es fa un càlcul energètic, s'avaluen els tres termes anteriors per tal d'obtenir l'energia total dels sistema. D'aquests tres termes, el d'energia covalent i energia del solvent són ràpids de calcular, mentre que el terme de *nonbonding* n'és el més lent. El terme de *nonbonding* consisteix en fer, per a cada parella d'àtoms, la suma de les seves energies de Lennard-Jones[iii] i Coulomb[iv]. Degut a la seva natura tots-contra-tots, avaluar aquest terme és un algorisme $O(N^2)$. També, quan s'avalua aquest terme, es té en compte si els àtoms estan al solvent o al buit, i per tant el càlcul i el resultat són diferents en cada cas.

La següent formula és la usada per la funció d'energia de *PELE* e inclou una part que correspon a l'energia en el buit i un altra corresponent a l'energia solvent. S'ha de notar que, en el buit, la segona part de la formula (l'energia solvent) és zero, i $\epsilon_{in(ij)}$ és 1.0.

$$E_{nb} = \sum_{i<j}[scale_{coulomb}\frac{q_i^s q_j^s}{\varepsilon_{in(ij)} r_{ij}} + \epsilon_i^s \epsilon_j^s((\frac{\sigma_{ij}^s}{r_{ij}})^{12} - (\frac{\sigma_{ij}^s}{r_{ij}})^6)]f_{ij}$$
$$+ \sum_{i<j}(\frac{1}{\varepsilon_{in(ij)}} - \frac{e^{-kf_{GB}}}{\varepsilon_{solv}})\frac{q_i^s q_j^s}{f_{GB}} + \frac{1}{2}\sum_i(\frac{1}{\varepsilon_{in(ii)}} - \frac{e^{-k\alpha_i}}{\varepsilon_{solv}})\frac{(q_i^s)^2}{\alpha_i} \quad (4.1)$$

$$f_{ij} = 0 \quad \text{1-2 pairs, 1-3 pairs} \quad (4.2)$$

[iii] Un parell d'àtoms o molècules estan subjectes a dos forces diferents en el límit d'una gran i una petita separació: una força atractiva actua a grans distàncies y una força repulsiva actua en petites distancies (el resultat de la sobreposició dels orbitals electrònics, conegut com la repulsió de Pauli). El potencial de Lennard-Jones (tambe conegut com el potencial L-J, el potencial 6-12 o, amb menor freqüència, com el potencial 12-6) és un model matemàtic senzill per representar aquest comportament.

[iv] La llei de Coulom diu que la magnitud de cadascuna de les forces elèctriques amb què interactuen dues càrregues puntuals en repòs és directament proporcional al producte de la magnitud d'ambdues càrregues i inversament proporcional al quadrat de la distància que les separa i té la direcció de la línia que les uneix. La força és de repulsió si les càrregues són d'igual signe, i d'atracció si són de signe contrari.

$$f_{ij} = 0.5 \quad \text{1-4 pairs} \tag{4.3}$$

$$f_{ij} = 1.0 \quad \text{all other pairs} \tag{4.4}$$

$$f_{GB} = \sqrt{r_{ij}^2 + \alpha_{ij}^2 e^{-D}} \tag{4.5}$$

$$\alpha_{ij} = \sqrt{\alpha_i \alpha_j} \tag{4.6}$$

$$D = \frac{r_{ij}^2}{(2\alpha_{ij})^2} \tag{4.7}$$

$$K = \sqrt{\frac{2N_A e^2 I}{\epsilon_{solv}\epsilon_0 RT}} \tag{4.8}$$

Els paràmetres ϵ_i^s i σ_{ij}^s es donen com a paràmetres de la funció. En la part de *PELE* encarregada del calcul de l'energia *nonbonded*, els parells 1-4 no es consideren, llavors $f_{ij} = 1.0$ i $scale_{coulomb} = 1.0$. Les càrregues q_i^s també es donen com a paràmetres de la funció. A continuació es presenta una breu descripció de cada paràmetre de l'equació de l'energia nonbonded:

- r_{ij}: és la distancia entre el parell d'àtoms i i j (en $\dot{A}$).
- q_i^s: càrrega escalada de l'àtom i (en unitats internes).
- q_j^s: càrrega escalada de l'àtom j (en unitats internes).
- ϵ_{ii}: valor epsilon escalat de l'àtom i (en sqrt(kcal/mol)).
- ϵ_{jj}: valor epsilon escalat de l'àtom j (en sqrt(kcal/mol)).
- σ_{ii}: valor sigma escalat de l'àtom i (en $\dot{A}$).
- σ_{jj}: valor sigma escalat de l'àtom j (en $\dot{A}$).
- K: pren el valor zero si l'aproximació *Debye* no està activa. Altrament, segueix una formula basada en la força iònica.
- ϵ_0: constant de permitivitat del buit.
- ϵ_{solv}: constant de permitivitat relativa del solvent: 80.0.
- $\epsilon_{in(ij)}$: constant de permitivitat relativa de la proteïna, depèn dels àtoms i i j, solament esta actiu en el model solvent *VDGBNP* (en aquest cas, és el màxim de la constant dielèctrica dels àtoms i i j). Per a tots els altres models el seu valor és de 1.0.
- f_{GB}: una funció del radi de *Born* per a tots els àtoms del sistema (veure apartat 3.1).
- α_i: radi de *Born* de l'àtom i (en $\dot{A}$).

De la fórmula presentada per al calcul d'energia s'ha de tenir en compte que l'últim terme, el que està multiplicat per 1/2, no es calcula en el codi de la funció d'energia que se'ns va entregar des del *BSC*. No obstant, aquest terme es presenta en el document per tal de proporcionar la formula completa que fa servir el software *PELE* original.

4.2 Implementació de la funció d'energia nonbonding

Després d'una breu explicació teòrica sobre l'energia *nonbonding*, es passa a presentar la seva implementació amb el llenguatge de programació C++.

Es presenten cinc versions de l'algorisme:

- **Versió seqüencial**: es tracta de la versió seqüencial de l'algorisme que se'm va proporcionar el *BSC* i la qual s'ha accelerat amb *OpenCL*.
- **Versió OpenCL**: la versió OpenCL del codi està formada per quatre implementacions diferents:
  - *Versió inicial*: es tracta de la primera implementació amb *OpenCL* que paral·lelitza la funció de l'energia en el codi. Aquesta versió és una traducció directa de la versió seqüencial de l'algoritme a un *kernel* de OpenCL. En aquesta versió però, es dóna el cas que els testos per a comprovar el correcte funcionament del codi no es passen amb èxit. Això és degut a la naturalesa intrínseca del propi codi seqüencial que, tal com s'explicara en el següent punt, guarda en una mateixa posició de memòria X la suma/resta del valor que hi ha en la posició més un valor procedent d'un altre calcul. En la versió seqüencial no hi ha cap problema perquè aquestes sumes/restes es fan una darrera de l'altra però, amb la versió amb OpenCL es té que diferents *threads* poden llegir i escriure simultàniament en una mateixa posició de memòria, fent que el resultat sigui incorrecte.
  - *Segona versió*: es tracta d'una segona implementació amb *OpenCL* que paral·lelitza la funció d'energia *nonbonding* en el codi. En aquesta versió, per tal de solucionar el problema de lectura/escriptura en memòria per varis *threads* simultanis, s'ha fet servir una tècnica basada amb operacions atòmiques. Amb l'ús de les operacions atòmiques s'ha solucionat el problema que es tenia en la primera versió del codi, de manera que els resultats dels diferents tests són correctes. No obstant, els temps de càlculs obtinguts han sigut molt dolents a causa de les operacions atòmiques.
  - *Tercera versió*: es tracta d'una tercera implementació amb *OpenCL* que paral·lelitza la funció d'energia *nonbonding* en el codi. En aquesta versió, per tal d'intentar millorar els temps de calcul, s'ha fet ús d'algoritmes d'ordenació. En aquest cas s'ha implementat l'algoritme *Bitonic sort* amb OpenCL per a ordenar les dades tractades i intentar millorar els temps de calcul.
  - *Quarta versió*: es tracta d'una quarta implementació amb *OpenCL* que paral·lelitza la funció d'energia *nonbonding* en el codi. En aquest cas, donat que en cap de les altres implementacions s'ha obtingut una millora significativa dels temps de calcul, s'ha repensat l'algorisme per tal d'obtenir una solució que involucra càlculs addicionals en el *host* que fan que els temps de calcul en el *device* siguin més bons. Es tracta de la versió que millors resultats ens ha donat.

Per tal d'unificar les versions seqüencial i OpenCL amb un sol codi, s'ha modificat la versió seqüencial que ens proporciona el *BSC* per tal d'obtenir un codi que permeti executar la versió seqüencial i la versió OpenCL (la quarta) a partir d'un paràmetre d'entrada des de la línia de comandes.

4.2.1 Versió seqüencial

Des del *BSC* se'ns va entregar un codi aïllat que representa una extracció exacta de la part del codi original de *PELE* que es fa servir per a calcular l'energia *nonbonding*. Juntament amb el codi se'ns va entregar un a serie de fitxers d'entrada que contenen paràmetres de configuració, varis llistats d'àtoms (les seves estructures) i les llistes de paralles *nonbonding* entre àtoms. La versió entregada ha sigut adaptada amb un conjunt de funcions de test per testejar el correcte comportament de la implementació.

En el main del programa l'únic que es fa és instanciar la classe de test i executar-la. S'executen funcions de test que criden al càlcul del terme de *nonbonding* i comproven que el resultat és correcte. Cada funció imprimeix "OK"per consola si el resultat és correcte, i "ERROR", juntament amb una llista d'errors trobats, si no ho és.

Pel que fa a l'estructura del programa, aquesta s'assembla molt a la estructura real de PELE: s'instancia una classe que implementa la interfície *NonBondingTermCalculator* i es crida al seu mètode de càlcul d'energia. Realment, aquests mètodes l'únic que fan és delegar la cridada a un objecte intern, "*implementation*", que és qui realment implementa el càlcul en la tecnologia desitjada -seqüencial, OpenMP, CUDA, OpenCL...- D'aquesta manera s'aïlla la implementació, depenent del hardware, i aquesta s'ofereix a la resta del codi a través d'una interfície intermitja.

A continuació es presenta el llistat de classes que implementen la versió seqüencial del programa:

- **TestNonBondingTermCalculator**: classe encarregada d'executar els diferents tests.
- **NonBondingEnergyValues**: classe que emmagatzema els diferents termes de l'energia *nonbonding*.
- **EnergyValues**: classe que empaqueta tots els termes d'energia.
- **NonBondPair**: classe que empaqueta els parells *nonbonding* juntament amb el seu valor sigma2 i la seva càrrega.
- **ToolsForEnergyTests**: classe que conté un conjunt de funcions per comprovar els resultats que s'obtenen.
- **NonBondingTermCalculator**: classe que computa l'energia *nonbonding*. Es tracta d'una classe abstracta que s'usa com a interfície per a totes les classes derivades.
- **NonBondingTermSgbCalculator**: classe que implementa la interfície *NonBondingTermCalculator*.
- **NonBondingTermImplementation**: interfície per a totes les implementacions de l'energia *nonbonding*. Aquesta és la interfície que s'ha implementant per a realitzar als càlculs amb OpenCL.
- **NonbondingTermSerialSgb**: classe que implementa la interfície *NonBondingTermImplementation*. Es tracta de la implementació seqüencial del programa.
- **ReadFileException**: classe que hereta de std:runtime_error i que controla les excepcions que es produeixen en la lectura dels fitxers de configuració.
- **ConstantDielectricEnergyCalculator**: classe que inicialitza les dielèctriques per als càlculs.
- **SolventEnergyCalculator**: classe que guarda les contribucions d'energia *Still*, *self* i *non-polar*. Es tracta de termes de la fórmula.

- **SolventEnergyParams**: classe que inicialitzar els paràmetres de l'energia sovent.
- **SolventTypes**: fitxer '.h' que enumera els diferents tipus de solvent en el sistema.
- **VariableDielectricEnergyCalculator**: classe que hereta de la classe *SolventEnergyCalculator*.
- **PotentialParameterization**: fitxer '.h' que enumera els diferents tipus de potencials en el sistema.
- **PhysConsts**: fitxer '.h' que conté diferents constants físiques usades en el sistema.
- **Assertion**, **MathTools**, **StringTools**, **TestTools**, **Utils**: diferents classes d'utilitats per alocatar memòria, executar els testos, tractar cadenes de text (strings), etc...

La funció *calcEnergyGradient* (apèndix B.1) de la classe *NonbondingTermSerialSgb* és l'encarregada de calcular l'energia. Dues coses cal destacar de la funció:

- Per una banda es pot observar que sols hi ha un bucle '*for*', la qual cosa pot fer pensar, de manera errònia, que el cost logarítmic de l'algorisme és $O(N)$. En aquest punt s'ha de veure que la funció rep com a paràmetre d'entrada el vector '*nonBondingInteractions*' de parelles d'àtoms. Aquest vector conté totes les parelles d'àtoms que tenen un pes significant en el calcul de l'energia. Si es recorda, en el punt 4.1 es deia que *PELE* fa servir un criteri d'optimització a base de llistes de parelles de *nonbonding*. Llavors, es té que el vector d'entrada és una llista de parelles *nonbonding* de tots els àtoms que tenen un pes significant en el sistema i, per tant, el cost logarítmic real de recorre el vector és $O(N^2)$.
- D'altra banda destaca la funció *GradientHessianPairUpdater_updateGradient* de la línia 37 (apèndix B.2). Aquesta funció no destaca per la seva complexitat, ja que tal com es veu és molt senzilla. No obstant, cal remarcar-la per entendre correctament les implementacions amb OpenCL. Com em vist, la funció principal és un bucle que itera per totes les parelles d'àtoms de la llista d'entrada per tal de calcular l'energia *nonbonding* del sistema. Ara, si mirem les línies 4-9 de la funció *GradientHessianPairUpdater_updateGradient*, podem veure que es fa una suma o una resta a les posiciones $a->ix$ i $b->ix$ de l'array *grad*. El valor '*ix*' és un index a l'array de coordenades dels àtoms que en aquest cas es fa servir com a index per guardar el valor de la suma o resta en l'array *grad* del sistema. El que pot passar doncs, és que varis àtoms tinguin el mateix valor en l'índex '*ix*', de manera que diferents iteracions de la funció principal guardaran el valor de la suma o resta en la mateixa posició. Per a la implementació amb seqüencial això no és un problema però, per a les versions amb OpenCL si, ja que l'array *grad* és guarda en memòria global, de manera que varis *threads* poden realitzar lectures i escriptures simultàniament, fet que provoca que el resultat final sigui incorrecte.

4.3 Versions OpenCL

Les següents versions del codi implementen la paral·lelització de la funció d'energia amb *OpenCL*. A l'hora d'implementar-les s'ha intentat seguir el mateix estil de programació que el codi original i, igual que en el solvent, l'estructura dels fitxers i el nom de les variables s'ha modificat poc.

La següent taula mostra els fitxers que té la versió seqüencial i els fitxers que s'han incorporat amb *OpenCL*:

Fitxers Originals (.cpp i .h)	Fitxers nous
TestNonBondingTermCalculator	basic
PotentialParameterizations	CmdParser
Assertion	CmdParserSO
MathTools	oclobject
physicsConstants	OpenCLSingleton
stringTools	Nonbonding
TestTools	NonbondingTermOpenclSgb
Utils	ObcEnergyGradient
ReadFileException	obc_nonbonding_kernels.cl
FakeComplex	
MinimalAtom	
ConstantDielectricEnergyCalculator	
SolventEnergyCalculator	
SolventEnergyParams	
SolventTypes	
VariableDielectricEnergyCalculator	
EnergyValues	
NonBondingEnergyValues	
NonBondPair	
ToolsForEnergyTests	
NonBondingTermCalculator	
NonBondingTermSgbCalculator	
NonBondingTermImplementation	
NonbondingTermSerialSgb	
Main	

Taula 4.1: Fitxers del programa *nonbonding* OpenCL.

Com en el cas del *Sovent*, els fitxers *basic*, *cmdparser*, *cmdparserso* i *oclobject* són fitxers que proveeix Intel com a open source per a facilitar l'ús de les abstraccions de *OpenCL*.

En quan a la resta de classes introduïdes:

- **OpenCLSingleton**:
  Aquesta classe segueix el patró de disseny *Singleton* i emmagatzema tots els objectes relacionats amb *OpenCL* (kernels i objectes bàsics) i un objecte *CmdParserSO* que guarda tots els paràmetres llegits per línia de comandes. Al tractar-se d'una classe *singleton*, un cop instanciada, es pot accedir als seus atributs des de qualsevol lloc del sistema.

- **NonbondingTermOpenclSgb**:
  Aquesta classe implementa la interfície *NonBondingTermImplementation* i és el punt d'entrada al calcul de l'energia *nonbonding* amb *OpenCL*.

- **Nonbonding**:
  Aquesta classe és l'encarregada de calcular les dimensions de la malla d'execució en el *device*, és a dir, calcula el número de grups i els *threads* que tindrà cada grup en el sistema. Un cop calculats instància un objecte *ObcEnergyGradient* i executa el *kernel/s* amb el mètode *executeEnergyGradient* de la classe *ObcEnergyGradient*.

- **ObcEnergyGradient**:
  Aquesta classe és l'encarregada de crear tots els buffers de comunicació entre el *host* i el *device* i d'executar el *kernel/s* en cada cas.

- **obc_nonbonding_kernels**:
  Aquest fitxer conté tots els *kernels* implementats. En aquest fitxers estan els kernels i funcions de totes les versions implementades. De fet, en l'entrega, sols es presenta la versió de codi C++ que millor resultats ens ha donat. No obstant, es presenten tots els kernels utilitzats en cada versió. Això és així perquè s'ha anat programat sobre el mateix codi modificant els fitxers en cada cas.

Com en el cas del *Solvent*, s'ha assumit que el host i el device no comparteixen memòria. Per tant, les dues entitats es comuniquen entre si a través de *buffers*. A continuació es presenta un llistat dels principals *buffers* que es fan servir:

- **cl_coord_buffer**: *buffer* que transmet les coordenades dels àtoms.

- **cl_dielectricConstantsMatrix_buffer**: *buffer* que transmet la matriu de constants dielèctriques guardada en la classe *SolventEnergyParams*.

- **cl_dielectricTypes_buffer**: *buffer* que transmet els tipus de dielèctrics que poden existir en el sistema.

- **cl_[ab]_a[xyz]_buffer**: es tracta de sis *buffers*, un per cada combinació d'àtom de la parella (àtoms *a* i *b*) i les seves coordenades *x*, *y* i *z*. Aquest *buffers* transmeten els índexs a les coordenades [XYZ] de l'array de coordenades dels àtoms per cada àtom.

- **cl_[ab]_epsilon_buffer**: es tracta de dos *buffers*, un per a l'àtom *a*, i un per l'àtom *b*. Aquest *buffers* transmeten els valors *epsilon* dels àtoms.

- **cl_[ab]_dielectric_buffer**: es tracta de dos *buffers*, un per a l'àtom *a*, i un per l'àtom *b*. Aquest *buffers* transmeten els valors *dielectric* dels àtoms.

- **cl_[ab]_alpha_buffer**: es tracta de dos *buffers*, un per a l'àtom *a*, i un per l'àtom *b*. Aquest *buffers* transmeten els valors *alpha* dels àtoms.

- **cl_charge_buffer**: aquest *buffer* transmeten els valors *charge* de cada parella d'àtoms.

- **cl_sigma_buffer**: aquest *buffer* transmeten els valors *sigma* de cada parella d'àtoms.

- **cl_params_buffer**: aquest *buffer* transmeten diferents valors de configuració per a realitzar diferents càlculs en el kernel/s.

- **cl_electrostatic_buffer**: aquest *buffer* es per guardar les sumes parcials de les contribucions d'energia *electrostàtica* que es calculen en el kernel.

- **cl_lennard_jones_buffer**: aquest *buffer* es per guardar les sumes parcials de les contribucions d'energia *lennard jones* que es calculen en el kernel.

- **cl_fbon_sgb_buffer**: aquest *buffer* es per guardar les sumes parcials de les contribucions d'energia *fbon_sgb* que es calculen en el kernel.

De manera addicional, en el cas que es vulgui calcular l'energia gradient del sistema -línia 27 de la funció d'energia (apèndix B.1)-, es fan servir servir els següents *buffers*:

- *cl_grad_buffer*
- *cl_a_vector_index_buffer*
- *cl_b_vector_index_buffer*
- *cl_a_number_index_buffer*
- *cl_b_number_index_buffer*
- *cl_a_prefix_sums_buffer*
- *cl_b_prefix_sums_buffer*
- *cl_gradient_x_buffer*
- *cl_gradient_y_buffer*
- *cl_gradient_z_buffer*

Aquests *buffers* transporten les dades per a l'última implementació que s'ha fet amb *OpenCL*. Es tracta de la implementació que millors resultats ha donat però, també es tracta de la més difícil d'implementar. Així doncs, es deixa l'explicació dels buffers per al punt on s'analitza l'algorisme implementat.

Un punt **molt** important a remarcar sobre les implementacions fetes amb OpenCL, i que ha condicionat el disseny que s'ha fet, és l'ús de les llistes de *nonbonding* que fa servir *PELE* i la manera que té de llegir-les. El problema que hi ha amb aquest mètode és que, quan es llegeix el fitxer amb la llista, es guarden les dades amb la classe *NonBondPair*, de manera que s'obté un vector d'objectes de la classe *NonBondPair*. Aquest fet suposa un problema a l'hora de transferir les dades al kernel, ja que OpenCL (en el moment de desenvolupar l'algoritme), no suportava la programació amb objectes. Per tant, per a poder transferir les dades al kernel s'ha d'executar un proces addicional de preparació on es recorre tot el vector de parelles i s'emmagatzemen les dades en el *buffer* de transferència corresponent. Aquest proces de preparació es troba en el constructor de la classe *ObcEnergyGradient*.

Des del *BSC* en són conscients d'aquesta problemàtica i estan treballant per trobar una solució al problema. Pel que fa aquest treball, tot i que s'ha pensat com fer-ho, aquesta és una tasca que s'escapa de l'abast i dels objectius planificats.

4.3.1 Versió inicial

La primera versió que s'ha fet amb OpenCL consisteix en una traducció directa del codi seqüencial al kernel OpenCL. Així doncs, el que s'ha fet és agafar el codi que hi ha en la classe *NonbondingTermSerialSgb* i s'ha passat al fitxer *obc_nonbonding_kernels.cl*. Es pot veure el codi del kernel en l'apèndix B.3.

Si s'analitza es pot veure que es una copia exacta de la funció que hi ha en la classe *NonbondingTermSerialSgb* de la versió seqüencial. En el fitxer *obc_nonbonding_kernels.cl* també estan totes les altres funcions que fa servir el kernel. Si mirem la implementació de la funció *GradientHessianPairUpdater_updateGradient* (apèndix B.4) de la línia 50, veiem que és igual que en el cas seqüencial.

Donada aquesta primera implementació és fàcil veure que quan els *threads* executen el codi

de la funció *GradientHessianPairUpdater_updateGradient*, alguns d'ells estaran llegint i escrivint simultàniament en la mateixa posició de memòria, de manera que el resultat final serà incorrecte.

Per tal de veure amb més claredat aquest cas ens podem imaginar el següent escenari:

- Imaginem que tenim 4 *thread*, on cadascun d'ells computa la contribució d'energia de les parelles 1-2, 1-3, 1-4, 1-5.

Amb aquest escenari és trivial veure que en les línies 6-8 de la funció *GradientHessianPairUpdater_updateGradient* els 4 *threads* estaran llegint i escrivint simultàniament en la mateixa posició de memòria global.

Un altre punt a tenir en compte en el kernel es troba entre les línies 62-84. Com el kernel s'executa un cop en cada *thread*, al final es té que cada *thread* té les seves pròpies contribucions de les energies *electrostatic*, *lennard jones* i *fnbon sgb*. Com el que ens interessa és disposar de la suma total, el que es fa és fer una reducció a nivell de grup i es guarden els resultats en els arrays de sortida del kernel. Al final es tindrà un array de sortida que tindrà les N contribucions de les energies (una per cada grup) que s'acabaran de sumar en el *host*.

4.3.2 Segona versió

Per tal de solucionar el problema de la lectura i escriptura simultània que donava la primera versió, es van pensar unes modificacions en la funció *GradientHessianPairUpdater_updateGradient*. Així doncs, el que es va fer és utilitzar una tècnica basada en operacions atòmiques per tal de bloquejar els *threads* mentre hi hagués algun altre realitzant una operació de lectura o escriptura en memòria.

Aquesta solució a primera vista semblava molt senzilla, simplement es tenien de fer servir les operacions atòmiques que ofereix la *API* de *OpenCL* per tal de bloquejar els *threads*. No obstant, a l'hora de fer servir les funcions de OpenCL, em vaig adonar que aquestes solament funcionen amb nombres enters i en el cas que s'havien d'aplicar no eren útils, ja que en el kernel es treballa amb nombres de tipus *float* o *double*.

Després d'analitzar, pensar e investigar com solucionar el problema, vaig arribar a la conclusió de crear les meves pròpies funcions atòmiques per treballar amb nombres de tipus *double* i *float*. Per fer-ho vaig fer servir el tipus de dades *union* que ofereix C. Aquest tipus de dades és una espècie de *struct* que permet tenir diferents representacions o formats d'un valor, de manera que vaig poder representar el valor float com si fos un enter i posteriorment vaig poder fer servir les operacions atòmiques de *OpenCL*.

En l'apèndix B.5 es pot veure la implementació de les funcions atòmiques. El que fan aquestes funcions és recuperar el valor que hi ha en una posició de memòria global i sumar o restar el valor desitjat. Un cop fet la suma o resta es mira si el valor que hi ha actualment en la posició de memòria continua essent el mateix que era en el moment d'entrar. Si és així vol dir que el valor no ha sigut modificat per cap altre *thread* i es pot guardar el nou valor. En cas contrari es recuperar el nou valor, es tornar a fer l'operació de suma o resta i es tornar a comprovar si el valor que hi ha en memòria coincideix o no amb el valor que s'havia recuperat.

Amb les operacions atòmiques implementades sols va ser necessari modificar la funció *GradientHessianPairUpdater_updateGradient* (B.6) per aconseguir que tots els casos de prova passessin correctament.

4.3.3 Tercera versió

En aquesta tercera versió del codi amb OpenCL s'intenta arreglar el problema que hi ha en el calcul del gradient, és a dir, s'intenta realitzar una versió alternativa a l'ús de les operacions atòmiques per tal de millorar els temps de calcul. Per fer això, aquesta versió es basa amb l'ús de l'algorisme *Bitonic Sort* (apèndix C.1) per ordenar les dades d'entrada.

El problema que hi ha és que les dades estan desordenades, és a dir, el vector de parelles d'àtoms està desordenat. Que el vector de parelles d'àtoms està desordenat significa que en la posició 0 del vector el primer àtom de la parella pot ser 1, en el segona posició del vector el primer àtom de la parella pot ser 5, en la tercera posició del vector el primer àtom de la parella pot ser 2, etc... De manera generar, això significa que en una posició X del vector el primer àtom de la parella pot ser qualsevol. Llavors, com que en el kernel cada *thread* opera sobre una parella, es té que la parella de la posició X i la parella de la posició Y poden tenir com a primer àtom el mateix, de manera que a l'hora d'escriure en l'array de memòria les dades quedaran inconsistents.

Per tal d'arreglar aquest problema es va pensar l'algorisme que descriu la següent imatge:

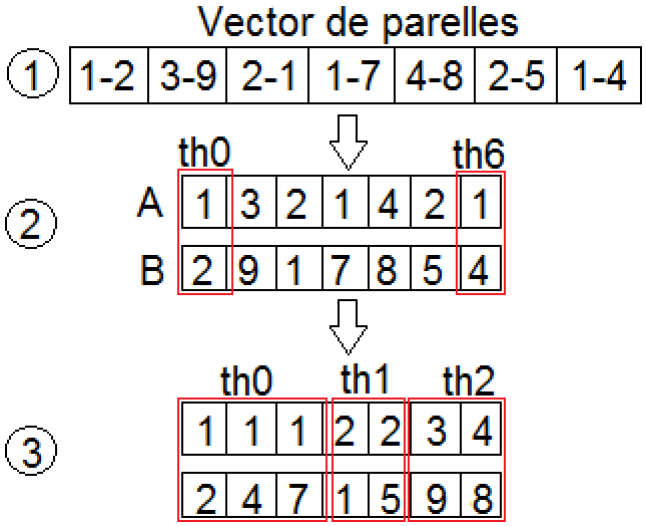


Figura 4.1: Implementació amb Bitonic Sort

- En el primer punt de la figura podem veure el vector de parelles d'entrada. Aquest correspon al vector *nonBondingInteractions*.
- El segon punt representa com li arriben les parelles al kernel. Si recordem, s'ha comentat que per a enviar les dades al kernel s'ha de fer un proces de preparació de dades per guardar-les en els buffers. Aquí, doncs, podem veure que les dades han sigut preparades i s'han obtingut dues arrays que identifiquen els àtoms de cada parella (en realitat els valors que identifiquen els àtoms en la figura són els índexs a les coordenades de cada àtom). També es pot observar que quan s'executi el kernel cada *thread* operara sobre una parella concreta.
- En el tercer punt podem veure les arrays dels àtoms ordenades a partir del primer àtom de la parella. Aquest és el punt on s'aplica l'algorisme d'ordenació de *Bitonic Sort*. Un cop les dades estan ordenades es té que cada *thread* opera sobre totes les parelles d'un àtom i **no** sobre una única parella com passava abans.

Al tenir les dades ordenades el que s'aconsegueix és que un sol *thread* treballi sobre totes les parelles d'un àtom concret, realitzant així tots els càlculs de totes les parelles de cada àtom. D'aquesta manera a l'hora de guardar els resultats en l'array de memòria solament hi haurà un únic *thread* llegint i escrivint en una mateixa posició de memòria.

En termes generals, la idea completa d'aquesta implementació és:

- Ordenar les parelles a partir del primer àtom de la parella.
- Amb els àtoms ordenats, calcular quantes vegades apareix cada àtom. El resultat es guarda en un array.
- Sabent el nombre de vegades que apareix cada àtom, calcular els offsets de cada àtom. Això serveix per a que cada *thread* pugui identificar a partir de quina posició de memòria pot començar a llegir dades. Per calcular-ho es fa servir l'algoritme de *prefix sums*E.1 o *scan*.
- Un cop es tenen les dades ordenades, el nombre d'aparicions de cada àtom i els offsets, es poden realitzar els càlculs.

Pel que fa a aquesta implementació, es té que no es va acabar d'integrar en el codi complet de PELE. El que es va fer és implementar el codi amb OpenCL de l'algorisme *Bitonic Sort* en un programa apart per tal de valorar si els temps d'ordenació de l'array eren prou ràpids com per a poder integrar els kernels de l'algorisme en la versió paral·lela del calcul de l'energia. Desafortunadament, els temps que es van obtenir per a l'ordenació donaven uns resultats més dolents que els temps que es triga en executar la versió seqüencial de l'algorisme, de manera que aquesta implementació es va tenir de descartar. S'adjunta el kernel que implementa l'algoritme de *Bitonic Sort* en l'entrega.

4.3.4 Quarta versió

Finalment es té la quarta versió de la implementació de la funció d'energia *nonbonding* amb *OpenCL*. Aquesta versió té algunes similituds amb la idea de la versió tres, però evitant l'ús d'un algoritme d'ordenació.

En aquest cas, el que es fa és fer ús de tres kernels per separat. Per una banda el primer kernel, el que ja teníem, ha sigut lleugerament modificat per tal d'adaptar-ho a la nova versió. D'altra banda, s'han implementat dos kernels nous que són els encarregats de computar les sumes i restes dels gradients i guardar el seu resultat en l'array *grad*.

Aquesta versió aprofita el bucle *for* de preparació de les dades per tal de crear dues estructures 2D dinàmiques que contindran, per a cada àtom de la parella, els índexs del array de parelles on apareixen. Això es pot veure en la següent figura 4.2.

- En el primer punt de la figura podem veure el vector de parelles d'entrada. Aquest correspon al vector *nonBondingInteractions*.
- En el segon punt es creen dues arrays de mida el nombre d'àtoms que hi ha en el sistema i es guarden els índexs de l'array *nonBondingInteractions* on apareix cada àtom de la parella.

Amb les arrays que s'obtenen podem veure, per exemple, que l'àtom 1 apareix en els índexs 0, 3 i 6 com a a primer àtom de la parella (array A).

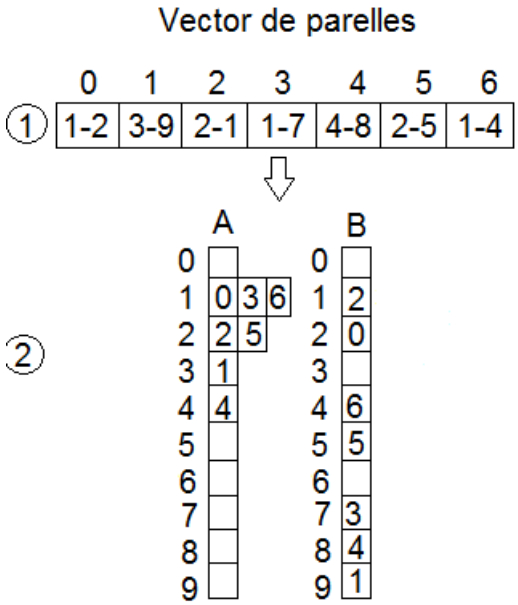


Figura 4.2: Creació de dos arrays dinàmiques 2D

El següent pas és calcular el nombre d'índexs on apareix cada àtom (figura 4.3). Per fer aquest calcul s'aprofita el bucle de preparació de dades com en el cas anterior.

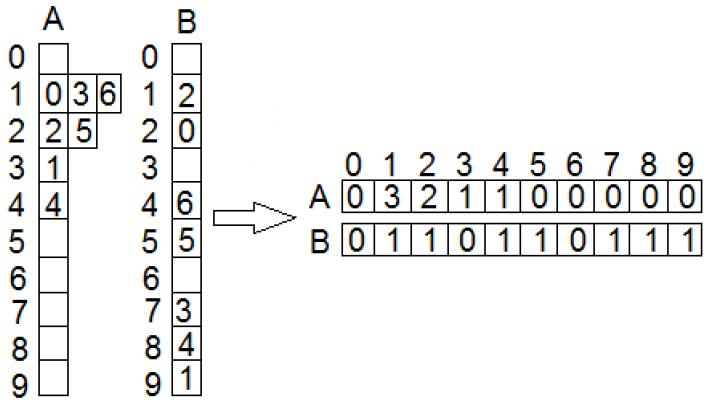


Figura 4.3: Suma de índexs on apareix cada àtom

Amb les arrays obtingudes podem saber que l'àtom 0 no apareix cap vegada en el sistema com a primer àtom de la parella, mentre que l'àtom 1 apareix tres vegades (en els índexs 0, 3 i 6) en el sistema com a primer àtom de la parella.

Com a últim pas es creen dues arrays (figura 4.4) d'una dimensió a partir de les arrays de dues dimensions calculades i es computen els offsets. D'aquesta manera s'obtenen dues arrays que contindran tots els índexs a cada àtom de manera ordenada i dues arrays que indicaran on comencen els índexs de cada àtom en l'array ordenada.

Per calcular les arrays dels offsets es fa servir l'algorisme de *prefix sums* o *scan* sobre els arrays que indiquen el nombre d'índexs de cada àtom.

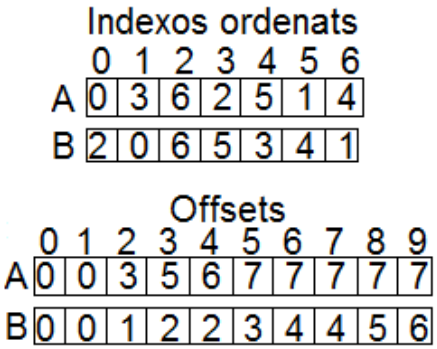


Figura 4.4: Arrays d'índexs ordenats i de offsets.

Amb les arrays obtingudes podem veure que els índexs on apareix l'àtom 1 com a primer àtom de la parella comencen en la posició 0 (array A offsets de la figura 4.4) de l'array de índex ordenat. Juntament amb l'array de nombre de vegades que apareixen el àtoms com a primer membre de la parella (figura 4.3), es pot veure que per a l'àtom 1 es començar en la posició 0 i s'han de llegir tres valors (0, 3 i 6).

Tots aquests nous vectors ens serveixen per realitzar els càlculs dels gradients de manera que no hi hagin lectures i escriptures simultanis en la mateixa posició de memòria. De fet, aquests vectors es transfereixen als nous kernels que s'han implementat. Els noms que prenen aquests vectors en el codi són:

- **cl_a_vector_index_buffer**: aquest buffer serveix per transmetre els índexs ordenats on apareixen cada àtom en l'array de parelles per al primer àtom de la parella.
- **cl_b_vector_index_buffer**: aquest buffer serveix per transmetre els índexs ordenats on apareixen cada àtom en l'array de parelles per al segon àtom de la parella.
- **cl_a_number_index_buffer**: aquest buffer transment el nombre de vegades que apareix cada àtom com a primer membre de la parella en l'array de parelles.
- **cl_b_number_index_buffer**: aquest buffer transment el nombre de vegades que apareix cada àtom com a segon membre de la parella en l'array de parelles.
- **cl_a_prefix_sums_buffer**: aquest buffer transmet els offsets per a l'array d'índexs ordenats dels àtoms primers de la parella.
- **cl_b_prefix_sums_buffer**: aquest buffer transmet els offsets per a l'array d'índexs ordenats dels àtoms segons de la parella.

Si recordem, aquests són els vectors que no s'havien explicat en el punt 4.3 del document.

A part d'aquests vectors, també és necessari disposa dels següents:

- **cl_gradient_[xyz]_buffer**: es tracta de tres buffer de lectura i escriptura que es passen com a paràmetres d'entrada en els tres kernels. En el primer kernel cada *thread* guarda la contribució del gradient calculada per a les coordenades [XYZ] dels àtoms de la parella. En el segon kernel es fan servir els valors -obtinguts del primer kernel- per calcular la suma global del gradient en la coordenada [XYZ]. Finalment, en el tercer kernel, es fan servir els valors -obtinguts del segon kernel- per calcular la resta global del gradient en la coordenada [XYZ].

En els apèndixs B.7, B.8 i B.9 es poden veure els tres kernels usats en aquesta versió del codi. El primer d'ells és una lleugera modificació del kernel que ja es tenia implementat en les altres versions. Si s'analitza el codi, els canvis canvis significatius es troben en:

- S'han afegit tres arrays d'entrada noves: *gradientX*, *graditnY* i *gradientZ*.
- Entre les línies 50 i 56 es pot veure que s'ha eliminat la crida a la funció *GradientHessianPairUpdater_updateGradient* i en el seu lloc s'ha afegit el codi que calcula les contribucions del gradient en les tres coordenades i que guarda el resultat en les arrays de gradient que s'han afegit noves en els paràmetres d'entrada.

Pel que fa al segon i tercer kernel, es tracten de kernels senzills que realitzen la suma i resta de les contribucions dels gradients per tal de computar el gradient global. La idea que hi ha en aquests kernels és que, a partir de l'array d'índexs ordenats, dels arrays que conten el nombre d'índexs de cada àtom i de l'array de offsets, es pugui calcular el comput total de la contribució del gradient en el sistema.

Un punt a tenir en compte és que l'array d'índexs fa referencia als índexs on apareix un àtom en el vector de parelles inicial. El que pot veure's estrany en aquests kernels és que l'array d'índex es fa servir per accedir a l'array de gradients (línia 11). Aquest accés és correcte, ja que en el primer kernel cada *thread* guardara els seus valors del gradient en els arrays globals *gradient[XYZ]* i els índex en que es guarden els valors corresponen amb els índexs originals del vector de parells.

Per aquesta versió del algoritme s'adjunten dues implementacions dels kernels B.8 i B.9. En la primera d'elles es crea un *range* de mida global el nombre d'àtoms, de manera que cada thread sera l'encarregat de fer el comput de totes les parelles que tinguin com a primer àtom el que indica l'id global del thread. Amb aquest versió però, es té que cada thread ha d'accedir N vegades a memòria global (una per cada parella) per llegir les dades que ha de tractar, fet que podria relentiza el temps de calcul.

Amb aquesta última problemàtica s'ha intentat millorar el kernel fen ús de la localitat de les dades. Per fer-ho s'han creat *numAtoms* grups amb 128 thrads cada grup (també s'han fet proves amb 64 i 256) per tal que la lectura de les dades es realitzi en paral·lel amb tots el threads del grup i sigui el thread zero que faci els càlculs (suma o resta) de totes les contribucions que han sigut guardades en memòria local. A l'hora d'implementar aquesta versió es pensava que els resultats que s'obtindrien serien millors que amb la primera versió però, tal com es veurà en els resultats no ha sigut així.

4.4 Resultats

Com en el cas del solvent, per obtenir els temps obtinguts en la paral·lelització de l'energia nonbonding, s'han fet cinc execucions en cada cas on s'han eliminat els outlers (màxim i mínim) i s'ha fet la mitjana aritmètica dels tres temps restants.

En aquest cas també es presenten els temps obtinguts amb els flags de compilació *O0* i *O3*. No obstant, per a l'energia nonbonding, s'han fet servir cinc tests diferents en cada cas: tres sense gradient i dos amb gradient. Pel que fa als testos sense gradients els temps es presenten un sol cop, ja que són iguals per a tots els casos. Per al cas dels testos amb gradient es presenten els diferents temps obtinguts en cada implementació, ja que és amb aquests testos on es tenia el problema d'accés a memòria global i es aquí on s'han aplicat les diferents tècniques explicades.

En la taula 4.2 es mostren els temps obtinguts en la versió inicial (apartat 4.3.1) per a ambdós modes d'execució (flags *O0* i *O3*) en els cincs testos de prova.

		Temps d'execució (ms)	
Test	**Versió**	**O0**	**O3**
1	OpenCL	1,4756167	1,45742
	Seqüencial	32,2966667	13,5243333
2	OpenCL	1,4296033	1,4280033
	Seqüencial	44,8183333	22,6176667
3	OpenCL	1,57374	1,5629233
	Seqüencial	44,803	22,626
4	OpenCL	1,766494	1,876704
	Seqüencial	43,0216667	18,2056667
5	OpenCL	1,921506	1,933259
	Seqüencial	57,1713333	26,0556667

Taula 4.2: Taula temps de la versió inicial.

Analitzant la taula es pot veure que en tots els testos la versió OpenCL és la més ràpida. Per als primers tres testos -els que no calculen el gradient- aquest resultat és correcte però, per al cas dels dos últims testos -els que calculen gradient- aquest temps és idíl·lic, ja que els resultats obtinguts no són els correctes. També podem observar que en general quan s'executa amb el flag *O3* la velocitat de còmput augmenta fins a dues vegades més ràpid per als càlculs seqüencials i algunes centèsimes o mil·lèsimes en el cas de la versió amb OpenCL. Cal destacar però, d'aquesta última observació, que els temps amb el flag *O3* dels dos últims testos són més lents que amb el flag *O0*, fet que s'hauria d'analitzar amb més deteniment i que es deixa per a futurs treballs i millores de l'algoritme.

La taula 4.3 mostra els speedsups optinguts amb la paral·lelitzacio en OpenCL.

		Speedups	
Test	**Versió**	**O0**	**O3**
1	OpenCL	21,8868942727	9,2796402547
2	OpenCL	31,3501887552	15,8386655689
3	OpenCL	28,469124506	14,4767180834
4	OpenCL	24,3542670963	9,7008727535
5	OpenCL	29,753398272	13,4775871727

Taula 4.3: Taula d'speedups de la versió inicial.

Amb els speedups es pot veure la millora que s'obté en la versió OpenCL respecte la versió amb seqüencial. Per als tres primers casos, els que no calculen el gradient, en general la millora que s'obté és molt bona. Respecte als dos últims testos es poden veure uns speedups molt bons però, com ja s'ha dit, aquests resultats són totalment falsos perquè els resultats que s'obtenen són incorrectes.

En la taula 4.4 es presenten els resultats que s'obtenen en el moment d'aplicar la primera millora de les atòmiques (apartat 4.3.2). En aquesta versió i en les següents, no es presenten els resultats que s'obtenen dels tres primers testos, ja que són exactament els mateixos que s'han presentat en les taules 4.2 i 4.3.

		Temps d'execució (ms)	
Test	**Versió**	**O0**	**O3**
4	OpenCL	15,9884319	15,9600371
	Seqüencial	43,0216667	18,2056667
5	OpenCL	15,7699616	15,8374316
	Seqüencial	57,1713333	26,0556667

Taula 4.4: Taula temps per a la versió amb atòmiques.

Si s'analitzen els resultats de les versions OpenCL i es comparen amb els que s'han obtingut en la versió inicial (taula 4.2) es pot veure que els temps de còmput han augmentat fins a 15 vegades més degut a l'ús de les operacions atòmiques. La taula 4.5 mostra els speedsups que s'obtenen amb la versió amb atòmics.

		Speedup	
Test	**Versió**	**O0**	**O3**
4	OpenCL	2,6907996337	1,1407032819
5	OpenCL	3,6253311676	1,6451952159

Taula 4.5: Taula d'speedups de la versió amb atòmiques.

Si es comparen els speedups amb els de la taula 4.3 es pot veure que els temps han caigut dramàticament sense arribar aconseguir ni una millora de 2x en la velocitat. Amb aquesta versió de l'algorisme s'aconsegueix que els resultats finals dels testos siguin correctes, però a base de pagar un cost computacional molt alt.

En la taula 4.6 es mostren els temps que s'han obtingut amb la quarta implementació amb OpenCL (apartat 4.3.4).

		Temps d'execució (ms)	
Test	**Versió**	**O0**	**O3**
4	OpenCL	5,326864	4,460802
	Seqüencial	43,0216667	18,2056667
5	OpenCL	5,524259	4,716691
	Seqüencial	57,1713333	26,0556667

Taula 4.6: Taula temps per a la quarta versió.

Si comparem els temps obtinguts amb els temps que s'obtenien amb la versió amb atòmiques (taula 4.4) es pot veure que aquest es redueixen fins a tres vegades, de manera que amb aquesta ultima versió s'ha aconseguit que tots els testos passin satisfactòriament i amb una velocitat 3 vegades més ràpid que amb la versió amb atòmiques. En la taula 4.7 es poden veure els speedups que s'obtenen.

		Speedup	
Test	**Versió**	**O0**	**O3**
4	OpenCL	8,0763591299	4,0812541556
5	OpenCL	10,3491406359	5,5241411193

Taula 4.7: Taula d'speedups per a la quarta versió.

Comparant els speedsups amb els obtinguts de la versió amb atòmics (taula 4.5) es pot veure que amb aquesta versió s'obté una millora en el rendiment entre **4x** i **5x** amb el flag *O3*, la qual cosa és una millora molt important respecte la versió amb seqüencial.

En la taula 4.8 es mostren els temps que s'han obtingut amb la segona implementació de la quarta versió amb OpenCL (apartat 4.3.4).

		Temps d'execució (ms)	
Test	**Versió**	**O0**	**O3**
4	OpenCL	7,444481	6,50221
	Seqüencial	43,0216667	18,2056667
5	OpenCL	7,675617	6,852519
	Seqüencial	57,1713333	26,0556667

Taula 4.8: Taula temps per a la segona implementació de la quarta versió.

Si es comparen els tems amb els de la taula 4.6 es pot veure que aquest s'han incrementat una mica. El mateix passa si mirem la taula d'speedups.

		Speedup	
Test	**Versió**	**O0**	**O3**
4	OpenCL	5,7790001478	2,799919827
5	OpenCL	7,448434868	3,802348698

Taula 4.9: Taula d'speedups per a la segona implementació de la quarta versió.

Mirant les taules es pot veure que el rendiment ha baixat. Per a ser sincer no m'esperava aquest comportament, sinó tot el contrari. Pensava que al fer la lectura de les dades de manera paral·lela amb tots els threads del grup i després fer les sumes de les contribucions amb el thread 0 de cada grup, es podria aprofitar la localitat de les dades i els temps de comput en la GPU disminuirien però no ha sigut així. Per a saber exactament que ha passat seria necessari realitzar un estudi amb més profunditat d'aquest cas amb l'ajuda d'alguna eina per fer debuging i profiling dels kernels.

Capítol 5

Anàlisis Econòmic

5.1 Valoració econòmica del projecte

Es presenta un anàlisis de costos dividit en tres blocs:

- Costos de programari
- Costos de hardware
- Costos de desenvolupament

5.1.1 Costos de programari

Quan es parlar de costos de programari es fa referència a llicencies de programari que s'hagin pogut pagar per fer servir el software necessari en el desenvolupament del treball.

El software usat ha sigut:

Software	**Unitats**	**Euros**
Latex	1	0
LibreOffice Calc	1	0
Netbeans	1	0
GanttProject	1	0
Ubuntu Server 14.04.3 LTS	1	0
Total	-	**0**

Taula 5.1: Taula de costos de programari.

Tot el software usat és de codi lliure, per tant, els costos de programari pugen a un total de **0 euros**.

5.1.2 Costos de hardware

Quan es parla de costos de hardware es fa referència a tot el hardware necessari per al desenvolupament del treball. Com que no es disposen dels preus exactes del hardware, es mostra unes aproximacions del valor que pot tenir en base a cerques fetes per Internet.

Component	Unitats	Preu Euros
16 GB de memòria	1	80
Intel(R) Xeon(R) CPU E3-1226 v3 @ 3.30GHz	1	230
AMD FirePro W5100 4GB GDDR5	1	355
Total	-	**665 euros**

Taula 5.2: Taula de costos del hardware.

El preu aproximat de la workstation és de **665 euros**.

5.1.3 Costos de desenvolupament

Quan es parla de costos de desenvolupament es fa referència als costos humans que es fan servir per portar endavant un projecte. En el desenvolupament d'un projecte intervenen diferents rols: project managers, analistes, programadors seniors. Cadascun d'aquests rols té un preu brut a l'hora. Una taula aproximada del preu d'hora segons el rol pot ser:

Rol	Preu Euros/h
Project Manager(P.M)	50
Analista(A)	40
Programador Senior(P.S)	30

Taula 5.3: Taula dels preus de mà d'obra segons el rol

La següent taula mostra un desglòs dels preus de les tasques realitzades segons el rol necessari per portar-les a terme.

Concepte	A.	P.S.	P.M.	Euros
Anàlisis i disseny global del TFM	8	0	8	720
Preparació de l'entorn de treball	0	8	0	240
Preparació informe Previ	0	0	20	1000
Paral·lelització del solvent	28	36	0	2200
Paral·lelització de la funció d'energia	60	88	0	5040
Documentació memòria	20	38	50	4440
Preparar defensa	0	0	16	800
Total	-	-	-	**14440**

Taula 5.4: Preu de les tasques segons el rol.

Aproximadament el preu en els recursos humans el projecte ascendeix a uns **14440 euros**.

5.1.4 Preu total aproximat

El preu total del treball realitzat és la suma dels preus de software, hardware i de desenvolupament. Aproximadament, el preu total del treball ascendeix a uns **15105 euros**.

Capítol 6

Valoracions i treballs futurs

6.1 Valoració personal

En general estic content dels resultats que s'han obtingut en el treball. Per una banda he aconseguit tots els objectius planificats, i per l'altra he aconseguit donar un gir de 360 graus en la meva carrera professional.

Pel que fa als objectius que s'havien planificat en el treball, crec que és evident que s'han aconseguit tots sense excepció. S'ha aconseguit millorar el rendiment del software PELE fins a 14 vegades més ràpid en el cas del OBC Solvent, s'ha aconseguit una millora de entre 10 i 15 vegades més ràpid per als càlculs d'energia nonbonding sense gradient i una millora de entre 4 i 5 vegades més ràpid per als casos dels càlculs de l'energia nonbonding amb l'ús del gradient. D'altra banda he pogut millorar molt les meves aptituds en recerca i sobretot he millorat molt el meu coneixement en la tecnologia OpenCL i el paradigma de programació heterogeni.

Pel que fa a la meva carrera professional, gràcies a la decisió de fer aquest treball vaig començar un segon màster d'especialització en Enginyeria computacional i Matemàtica (màster que he estat cursant en paral·lel amb el desenvolupament del treball). Amb el treball realitzar i el segon màster se m'ha obert una porta d'entrada en aquest camp d'estudi, i és ara al setembre d'aquest any que començaré un doctorat industrial entre la UAB i l'empresa *Pharmacelera*. Amb aquest doctorat faig un gir de 360 graus en la meva carrera professional, deixant enrere la meva professió encarada més a caire de negoci i començant un nou camí de caire més acadèmic.

Val a dir però, que el desenvolupament del treball no ha sigut una tasca fàcil, sinó tot el contrari. Inicialment tenia planificat enfocar aquest treball en tres grans blocs: OBC Solvent, energia Nonbonding i anàlisis dels algoritmes amb CodeXL. No obstant, durant el transcurs del treball m'he topat amb moltes dificultats que no m'han permès realitzar el tercer bloc planificat. Entre les dificultats trobades, la més difícil ha sigut l'estudi -a un nivell molt bàsic- de tota la part teòrica que emmarca el treball, és a dir, tota la part teòrica relacionada amb la química i bioquímica que intervé en els càlculs d'energia en les molècules. Per a fer front aquest estudi he disposat de l'ajuda d'un tècnic de recerca que treballa en el BSC. Gràcies a ell he pogut entendre els conceptes bàsics que hi han darrere de tots els articles científics que he fet servir per dur a terme el treball.

La segona dificultat que m'he trobat el en transcurs del treball ha sigut en l'ús de la tecnologia OpenCL. Durant la programació vaig tenir problemes en la gestió de la memòria a nivell de GPU, problemes que venien donats pel meu desconeixement de la tecnologia. Afortunadament, gràcies a una serie de llibres que he fet servir com a base per a estudiar la tecnlogia vaig poder solucionar els problemes i vaig evitar que es tornessin a repetir.

Finalment, un altra dificultat important amb que em vaig topar, va ser en la paral·lelització de la part de l'energia Nonbonding. Quan vaig paral·lelitzar la funció d'energia vaig tenir el problema de la lectura i escriptura en memòria global que es produïa de manera simultània per varis threads. Per donar resposta a aquest problema vaig pensar d'utilitzar les operacions atòmiques però, els temps de comput que s'obtenien a penes milloraven. Per tant, em vaig tenir de posar a analitzar el codi i pensar com es podria solucionar aquest problema. Un primer enfocament va ser intentar fer ús d'algoritmes d'ordenació, de manera que vaig tenir de dedicar part del temps a estudiar el funcionament de l'algorisme *Bitonic Sort* i, encara que no es menciona en aquest document, de l'algoritme *Radix Sort*.

6.2 Futures línies de treball

Amb la feina realitzada i les últimes versions d'OpenCL s'obrin varies línies de treball per tal de millorar el que ja s'ha fet. Per una banda, la millora més important que es pot fer és l'*upgrade* de la versió d'OpenCL a la versió 2.2. Amb aquesta última versió es dóna la possibilitat de treballar amb el llenguatge C++ en els kernels, de manera que es pot fer servir la programació orientada a objectes. Amb aquest nou paradigma es podrien millorar els algoritmes implementats per que no es tingues de fer un tractament inicial de les dades per a poder-les enviar a la GPU.

Altres millores que s'han de fer són el canvi del model de dades d'entrada (per al cas de l'energia nonbonding). Si recordem en l'energia nonbonding es tenia una llista de parelles d'àtoms d'entrada que era necessari recórrer per obtenir les dades i enviar-les a la GPU. Amb un canvi de model de lectura de les dades es podrien construir les estructures de dades necessàries en el moment de la lectura.

Un altra línia de treball que es pot seguir és l'estudi dels algoritmes mitjançant l'eina CodeXL (o alguna de similar). Aquesta línia d'estudi en principi es volia afegir en aquest treball però, pels motius descrits en l'apartat 6.1 no ha sigut possible, de manera que és una feina que s'hauria de fer per tal d'analitzar el codi i veure els colls d'ampolla que puguin haver-hi.

Apèndix

Apèndix A

OBC Solvent

En l'apèndix A es fa referencia a explicacions addicionals que no tenen cabuda en el document principal del treball.

A.1 Fitxers .h del codi OBC Solvent

- **Classe Atom**:

```
class Atom
{
  public:
    Atom();
    Atom(const Atom &other);

    virtual ~Atom();

    double getX() const {return this->coordinates[this->ix];};
    double getY() const {return this->coordinates[this->iy];};
    double getZ() const {return this->coordinates[this->iz];};

    std::string getTemplateName() const;

    double distance(Atom* a) const;
    //--------------
    // Attributes
    //--------------

    // Atom attributes as described by PDB documentation
    int serial;
    char altLoc;
    std::string resName;
    char chainID;
    int resSeq;
    char iCode;
    std::string name;
    std::string templateName;

    // Atom's solvent attributes
    double alpha; // Inverse of Born radius for GB
```

```
    double gbr;

    // Coordinate indexes inside coordinates array
    unsigned int ix;
    unsigned int iy;
    unsigned int iz;

    // The coordinates array
    double* coordinates;
}
```

- **Classe AtomSet**:

```
class AtomSet {
  public:
    std::vector<Atom*> & getAllAtoms();
    static boost::shared_ptr<AtomSet> readData(const std::string &dataFilename);
  private:
    std::vector<Atom*> atomPointers; // pointers to the atom objects in atoms.
    // atomPointers[i] contains the pointer to atoms[i].
    std::vector<Atom> atoms; // The atoms forming this set.
    std::vector<double> coordinates; // vector with all the coordinates of the
    // atoms in this atom set. For atom i in atoms, its coordinates are
    // (i-1)*3, (i-1)*3+1 (i-1)*3+2 (for x,y,z respectively).
}
```

- **Classe ObcAlphaSasaUpdater**:

```
class ObcAlphaSasaUpdater
{
  public:
    ObcAlphaSasaUpdater(AtomSet *atomSet, const std::string &solventDataPath);
    virtual ~ObcAlphaSasaUpdater();

    void updateAlphasAndSASAs(bool firstTime);

  private:
    AtomSet *atomSet;
    std::vector<double> atomsBornRadiiOffset;
    std::vector<double> atomsHCT;

    ObcAlphaSasaUpdater(const ObcAlphaSasaUpdater&);
    ObcAlphaSasaUpdater& operator=(const ObcAlphaSasaUpdater&);

    void updateAlphas();

    void loadSolventTemplate(const std::string &solventDataPath);

    void assignSolventValuesToAtoms(std::map<std::string, double> & atomGBR,
                                    std::map<std::string, double> & atomHCT);

    std::string createAtomSolventKey(Atom* atom);
```

```
    void checkAtomSolventId(const std::string & atomId,
                            std::map<std::string,double>& atomGBR,
                            std::map<std::string,double>& atomHCT);

    void setAtomsObcProperties(Atom * atom, unsigned int atomIndex,
                               std::map<std::string, double>& atomGBR,
                               std::map<std::string, double>& atomHCT);

    double computeOtherAtomsContributionToIthAtom(unsigned int i);
}
```

- **PeleBuildException**:

```
class PeleBuildException : public std::runtime_error
{
  public:
    PeleBuildException(const std::string & message, const std::string & location);
    virtual ~PeleBuildException() throw();
};
```

- **SolventTemplateReader**:

```
class SolventTemplateReader {
  public:
    SolventTemplateReader(const std::string &solventDataPath);
    virtual ~SolventTemplateReader();

    std::map<std::string, double> & getAtomToGBR();
    std::map<std::string, double> & getAtomToHCT();

  private:
    std::map<std::string, double> atomToGBR;
    std::map<std::string, double> atomToHCT;

    void read(const std::string & path);
    void processLine(const std::string& line);

};
```

- **StringTools**:

```
namespace StringTools{

  std::vector<std::string> split(const std::string &line,
                                 const std::string & delimiter=" ");
  void split(std::vector<std::string> & elements, const std::string & line,
             const std::string & delimiter=" ");

  std::string deleteWhiteSpaces(const std::string &str);
}
```

A.2 Funció *updateAlphas*

```
//////////////////////////////////////
/// It updates the values of the alphas
//////////////////////////////////////
1. void ObcAlphaSasaUpdater::updateAlphas()
2. {
3.   const double ALPHA = 1.0;
4.   const double BETA = 0.8;
5.   const double GAMMA = 4.85;

7.   vector<Atom*> & atoms = atomSet->getAllAtoms();
8.   unsigned int numAtoms = atoms.size();

9.   for(unsigned int i = 0; i < numAtoms; ++i)
10.  {
11.    double sum = computeOtherAtomsContributionToIthAtom(i);

12.    double squaredSum = sum*sum;
13.    double cubicSum = sum*squaredSum;

14.    double tangSum = tanh(ALPHA * sum - BETA * squaredSum + GAMMA * cubicSum);
15.    double bornRadii = 1.0 / atomsBornRadiiOffset[i] - tangSum / atoms[i]->gbr;

16.    atoms[i]->alpha = 1.0 / bornRadii;
17.  }
18.}
```

A.3 Funció computeOtherAtomsContributionToIthAtom

```
//////////////////////////////////////////////////////////////
/// It computes the contribution of the other atoms to the i-th Atom
/// \param atomIndex [In] Index of the ith atom
/// \return Contribution of other atoms to the ith Atom
//////////////////////////////////////////////////////////////
1. double ObcAlphaSasaUpdater::computeOtherAtomsContributionToIthAtom
                                                    (unsigned int atomIndex)
2. {
3.   double sum = 0.0;

4.   vector<Atom*> & atoms = atomSet->getAllAtoms();
5.   unsigned int numAtoms = atoms.size();

6.   for(unsigned int j = 0; j < numAtoms; ++j)
7.   {
8.     if(atomIndex != j)
9.     {
10.      double distance = atoms[atomIndex]->distance(atoms[j]);
11.      double sk = atomsBornRadiiOffset[j] * atomsHCT[j];
12.      double sk2 = sk*sk;

13.      double extraDistance = distance + sk;
```

```
14.      if(atomsBornRadiiOffset[atomIndex] < extraDistance)
15.      {
16.        double lik = 1.0 / max(atomsBornRadiiOffset[atomIndex], fabs(distance - sk));
17.        double uik = 1.0 / extraDistance;
18.        double lik2 = lik * lik;
19.        double uik2 = uik * uik;

20.        double term = lik - uik + 0.25 * distance * (uik2 - lik2)
21.                    + (0.5 / distance) * log(uik / lik)
22.                    + (0.25 * sk2/distance) * (lik2 - uik2);

23.        if(atomsBornRadiiOffset[atomIndex] < sk - distance)
24.        {
25.          term += 2.0 * (1.0 / atomsBornRadiiOffset[atomIndex] - lik);
26.        }

27.        sum += 0.5 * term;
28.      }
29.    }
30.  }

31.  return atomsBornRadiiOffset[atomIndex] * sum;
32.}
```

A.4 Fitxer .h de la class *SoventMain*

```
class Solvent {
  public:
    Solvent(CmdParserSO& cmd);
    virtual ~Solvent() throw();
    void ExecuteKernel(
            OpenCLBasic& ocl,
            OpenCLProgramOneKernel& first_executable,
            OpenCLProgramOneKernel& second_executable,
            CmdParserSO& cmd,
            float* p_time_device,
            float* p_time_host,
            float* p_time_read,
            clock_t *tStart,
            clock_t *tEnd);

    void ExecuteNative(
            const CmdParserSO& cmd,
            clock_t *tStart,
            clock_t *tEnd);

  private:
    void writeResultsToFile(const std::wstring &outputFilename,
                            const std::vector<Atom*> &atoms);
}
```

A.5 Fucnió *updateAlphasInnerLoopCL*

```
////////////////////////////////////////////////////////////
/// \remarks
/// It updates the values of the alphas - OpenCL version
/// inner loop parallelized
////////////////////////////////////////////////////////////
1. void ObcAlphaSasaUpdater::updateAlphasInnerLoopCL(
                    cl_command_queue cmd_queue/* command_queue */,
                    CmdParserSO& cmd,
                    cl_kernel        kernel/* kernel */,
                    cl_uint          work_dim/* work_dim */,
                    const size_t *   global_offset/* global_work_offset */,
                    const size_t *   global_size/* global_work_size */,
                    const size_t *   local_size/* local_work_size */,
                    cl_uint          num_events/* num_events_in_wait_list */,
                    const cl_event * event_wait_list/* event_wait_list */,
                    cl_event *       event/* event */)
2. {
3.  const fp_t ALPHA = ALPHA_CONST;
4.  const fp_t BETA  = BETA_CONST;
5.  const fp_t GAMMA = GAMMA_CONST;

6.  vector<Atom*> & atoms = atomSet->getAllAtoms();
7.  unsigned int numAtoms = atoms.size();

8.  for(unsigned int i = 0; i < numAtoms; ++i)
9.  {
10.   fp_t sum = computeOtherAtomsContributionToIthAtomCL(i, cmd_queue, kernel,
      work_dim, global_offset, global_size, local_size, num_events,
      event_wait_list, event);

      // accumulate time for all kernels
11.   if(cmd.ocl_profiling.getValue())
12.   {
13.     cl_int          err;
14.     cl_ulong start = 0;
15.     cl_ulong end = 0;
16.     err = clGetEventProfilingInfo(*event, CL_PROFILING_COMMAND_START,
17.     sizeof(cl_ulong), &start, NULL);
18.     SAMPLE_CHECK_ERRORS(err);
19.     err = clGetEventProfilingInfo(*event, CL_PROFILING_COMMAND_END,
20.     sizeof(cl_ulong), &end, NULL);
21.     SAMPLE_CHECK_ERRORS(err);
22.     extern_kernel_time += (float)(end - start)*1e-9f;
23.   }
    // ---

24.   fp_t squaredSum = sum*sum;
25.   fp_t cubicSum = sum*squaredSum;

26.   fp_t tangSum = tanh(ALPHA * sum - BETA * squaredSum + GAMMA * cubicSum);
27.   fp_t bornRadii = 1.0 / atomsBornRadiiOffset[i] - tangSum / atoms[i]->gbr;
```

```
28.   atoms[i]->alpha = 1.0 / bornRadii;
29. }
30.}
```

A.6 Funció *computeOtherAtomsContributionToIthAtomCL*

```
////////////////////////////////////////////////////////////
/// It computes the contribution of the other atoms to the i-th Atom
/// \param atomIndex [In] Index of the ith atom
/// \return Contribution of other atoms to the ith Atom
////////////////////////////////////////////////////////////
1. fp_t ObcAlphaSasaUpdater::computeOtherAtomsContributionToIthAtomCL(
                          unsigned int     atomIndex,
                          cl_command_queue cmd_queue/* command_queue */,
                          cl_kernel        kernel/* kernel */,
                          cl_uint          work_dim/* work_dim */,
                          const size_t *   global_offset/* global_work_offset */,
                          const size_t *   global_size/* global_work_size */,
                          const size_t *   local_size/* local_work_size */,
                          cl_uint          num_events/* num_events_in_wait_list */,
                          const cl_event * event_wait_list/* event_wait_list */,
                          cl_event *       event/* event */)
2.{
3.  fp_t sum = 0.0;
4.  fp_t kernel_sum = 0.0;

5.  vector<Atom*> & atoms = atomSet->getAllAtoms();
6.  unsigned int numAtoms = atoms.size();

7.  cl_int          err;

8.  this->p_config[1] = atomIndex;
9.  this->p_config[0] = numAtoms;

10.  // Copy config to config buffer
11. err = clEnqueueWriteBuffer(cmd_queue, this->cl_config_buffer, CL_TRUE,
12. 0, this->config_size, this->p_config, 0, NULL, NULL);
13. SAMPLE_CHECK_ERRORS(err);

14. // Execute kernel
15. err= clEnqueueNDRangeKernel(cmd_queue, kernel, work_dim, global_offset,
16. global_size, local_size,   num_events, event_wait_list, event);
17. SAMPLE_CHECK_ERRORS(err);
18. err = clWaitForEvents(1, event);
19. SAMPLE_CHECK_ERRORS(err);

20. // Copy results back from the message buffer
21. err = clEnqueueReadBuffer(cmd_queue, this->cl_output_buffer, CL_TRUE,
22. 0, this->size, this->p_output, 0, NULL, NULL);
23. SAMPLE_CHECK_ERRORS(err);
```

```
24. for(unsigned int j = 0; j < numAtoms; ++j)
25. {
26.   if(atomIndex != j)
27.   {
28.     kernel_sum += this->p_output[j];
29.   }
30. }

31. return this->p_BRO[atomIndex] * kernel_sum;
32.}
```

A.7 Kernel *OBCInnerLoopKernel*

```
__kernel void
OBCInnerLoopKernel( const __global FLOAT *input, const __global FLOAT *BRO,
                    const __global FLOAT *HCT, __global FLOAT *output,
                    const __global int *config)
{
1. //output[index] = input[index];
2.  int atom_index = config[1];
3.  int num_atoms = config[0];
4.  int j = get_global_id(0); // we use same varname as original code in
                              // computeOtherAtomsContributionToIthAtom
                              // 'j' is the index of the atom in the innermost
                              // loop

5.  FLOAT sum;

6.  if (atom_index >= num_atoms)
7.    return;

8.  sum = ComputeContributionFromAtom(atom_index, j, input, BRO, HCT);

9.  output[j] = sum;
}
```

A.8 Funció *ComputeContributionFromAtom*

```
// Routine that computes the contribution of atom 'j' to atom atom_index
FLOAT
ComputeContributionFromAtom(size_t atom_index, int j, const __global FLOAT *input,
                            const __global FLOAT *BRO, const __global FLOAT *HCT)
{
1.  FLOAT distance, sk, sk2, extraDistance, lik, uik, lik2, uik2, term;
2.  FLOAT sum = 0.0;

3.  if (atom_index != j)
4.  {
5.    distance = ComputeDistance(input, atom_index, j);

6.    sk = BRO[j] * HCT[j];
7.    sk2 = sk*sk;
```

```
8.    extraDistance = distance + sk;

9.    if(BRO[atom_index] < extraDistance)
10.    {
11.      lik = 1.0 / max(BRO[atom_index], fabs(distance - sk));
12.      uik = 1.0 / extraDistance;
13.      lik2 = lik * lik;
14.      uik2 = uik * uik;

15.      term = lik - uik + 0.25 * distance * (uik2 - lik2)
16.             + (0.5 / distance) * log(uik / lik)
               + (0.25 * sk2/distance) * (lik2 - uik2);

17.      if(BRO[atom_index] < sk - distance)
18.      {
19.        term += 2.0 * (1.0 / BRO[atom_index] - lik);
20.      }

21.      sum += 0.5 * term;
22.    }
23.  }

24.  return sum;
}
```

A.9 Funció *updateAlphasOutterLoopCL*

```
////////////////////////////////////////////////////////////
/// \remarks
/// It updates the values of the alphas - OpenCL version
/// outter loop parallelized
////////////////////////////////////////////////////////////
void ObcAlphaSasaUpdater::updateAlphasOutterLoopCL(
                    cl_command_queue cmd_queue/* command_queue */,
                    CmdParserSO& cmd,
                    cl_kernel        kernel/* kernel */,
                    cl_uint          work_dim/* work_dim */,
                    const size_t *   global_offset/* global_work_offset */,
                    const size_t *   global_size/* global_work_size */,
                    const size_t *   local_size/* local_work_size */,
                    cl_uint          num_events/* num_events_in_wait_list */,
                    const cl_event * event_wait_list/* event_wait_list */,
                    cl_event *       event/* event */)
{
1.  vector<Atom*> & atoms = atomSet->getAllAtoms();
2.  unsigned int numAtoms = atoms.size();

3.  cl_int          err;

4.  // Copy config to config buffer
5.  this->p_config[0] = numAtoms;
```

```
6.  err = clEnqueueWriteBuffer(cmd_queue, this->cl_config_buffer, CL_TRUE, 0,
                          this->config_size, this->p_config, 0, NULL, NULL);
7.  SAMPLE_CHECK_ERRORS(err);

8.  // Execute kernel
9.  err= clEnqueueNDRangeKernel(cmd_queue, kernel, work_dim, global_offset,
             global_size, local_size, num_events, event_wait_list, event);
10.  SAMPLE_CHECK_ERRORS(err);
11.  err = clWaitForEvents(1, event);
12.  SAMPLE_CHECK_ERRORS(err);

13.  // Copy results back from the message buffer and assign them to atoms
14.  err = clEnqueueReadBuffer(cmd_queue, this->cl_output_buffer, CL_TRUE, 0,
                            this->size, this->p_output, 0, NULL, NULL);
15.  SAMPLE_CHECK_ERRORS(err);

16.  for(unsigned int i = 0; i < numAtoms; ++i)
17.  {
       atoms[i]->alpha = this->p_output[i];
18.  }

19.  if(cmd.ocl_profiling.getValue())
20.  {
21.    cl_int          err;
22.    cl_ulong start = 0;
23.    cl_ulong end = 0;
24.    err = clGetEventProfilingInfo(*event, CL_PROFILING_COMMAND_START,
                                       sizeof(cl_ulong), &start, NULL);
25.    SAMPLE_CHECK_ERRORS(err);
26.    err = clGetEventProfilingInfo(*event, CL_PROFILING_COMMAND_END,
                                         sizeof(cl_ulong), &end, NULL);
27.    SAMPLE_CHECK_ERRORS(err);
28.    extern_kernel_time += (float)(end - start)*1e-9f;
29.  }
}
```

A.10 Kernel *OBCOutterLoopKernel*

```
// OutterLoop Kernel -- ENTRY POINT
__kernel void
OBCOutterLoopKernel( const __global FLOAT *input, const __global FLOAT *BRO,
                     const __global FLOAT *HCT, __global FLOAT *output,
                     const __global int *config, const __global FLOAT *GBR)
{
1.  int atom_index = (int)get_global_id(0);
2.  FLOAT sum, squaredSum, cubicSum, tangSum, bornRadii;
3.  int num_atoms = config[0];

4.  // Atom out of range -- just return
5.  if (atom_index >= num_atoms)
6.    return;
```

```
7.  // TODO: define these constants in a common file
8.  FLOAT ALPHA = 1.0;
9.  FLOAT BETA = 0.8;
10.  FLOAT GAMMA = 4.85;

11.  sum = 0;
12.  for(unsigned int j = 0; j < num_atoms; ++j)
13.  {
14.    sum += ComputeContributionFromAtom(atom_index, j, input, BRO, HCT);
15.  }

16.  sum = BRO[atom_index]*sum;

17.  squaredSum = sum*sum;
18.  cubicSum = sum*squaredSum;

19.  tangSum = tanh(ALPHA * sum - BETA * squaredSum + GAMMA * cubicSum);
20.  bornRadii = 1.0 / BRO[atom_index] - tangSum / GBR[atom_index];

21.  output[atom_index] = 1.0 / bornRadii;
}
```

A.11 Funció *updateAlphasPairLoopCL*

```
////////////////////////////////////////////////////////////
/// \remarks
/// It updates the values of the alphas - OpenCL version
/// pair loop parallelized
////////////////////////////////////////////////////////////
void ObcAlphaSasaUpdater::updateAlphasPairLoopCL(
                          cl_command_queue cmd_queue/* command_queue */,
                          CmdParserSO& cmd,
                          cl_kernel        first_kernel/* kernel */,
                          cl_kernel        second_kernel/* kernel */,
                          cl_uint          work_dim/* work_dim */,
                          const size_t *   global_offset/* global_work_offset */,
                          const size_t *   global_size/* global_work_size */,
                          const size_t *   local_size/* local_work_size */,
                          cl_uint          num_events/* num_events_in_wait_list */,
                          const cl_event * event_wait_list,/* event_wait_list */
                          cl_event * first_event,
                          cl_event * second_event)
{
1.  vector<Atom*> & atoms = atomSet->getAllAtoms();
2.  unsigned int numAtoms = atoms.size();

3.  cl_int          err;

    // Copy config to config buffer
4.  this->p_config[0] = numAtoms;
5.  this->p_config[1] = *local_size;
```

```
6.  err = clEnqueueWriteBuffer(cmd_queue, this->cl_config_buffer, CL_TRUE, 0,
                            this->config_size, this->p_config, 0, NULL, NULL);
7.  SAMPLE_CHECK_ERRORS(err);

    // Execute first_kernel
8.  err= clEnqueueNDRangeKernel(cmd_queue, first_kernel, work_dim, global_offset,
             global_size, local_size, num_events, event_wait_list, first_event);
9.  SAMPLE_CHECK_ERRORS(err);
10.  err = clWaitForEvents(1, first_event);
11.  SAMPLE_CHECK_ERRORS(err);

12.  size_t global_size_aux = (int)(ceil((double)numAtoms));
13.  if ( global_size_aux % *local_size != 0)
14.      global_size_aux = ((global_size_aux / *local_size) *   (*local_size))
                                                               + *local_size;

    // Execute second_kernel
15.  err= clEnqueueNDRangeKernel(cmd_queue, second_kernel, work_dim, global_offset,
        &global_size_aux, local_size, num_events, event_wait_list, second_event);
16.  SAMPLE_CHECK_ERRORS(err);
17.  err = clWaitForEvents(1, second_event);
18.  SAMPLE_CHECK_ERRORS(err);

    // Copy results back from the message buffer and assign them to atoms
19.  err = clEnqueueReadBuffer(cmd_queue, this->cl_alpha_buffer, CL_TRUE, 0,
                                    this->size, this->p_alpha, 0, NULL, NULL);
20.  SAMPLE_CHECK_ERRORS(err);

21.  for(unsigned int i = 0; i < numAtoms; ++i)
22.  {
23.    atoms[i]->alpha = this->p_alpha[i];
24.  }

25.  if(cmd.ocl_profiling.getValue())
26.  {
27.    cl_int          err;
28.    cl_ulong first_start = 0;
29.    cl_ulong first_end = 0;
30.    cl_ulong second_start = 0;
31.    cl_ulong second_end = 0;
32.    err = clGetEventProfilingInfo(*first_event, CL_PROFILING_COMMAND_START,
                                        sizeof(cl_ulong), &first_start, NULL);
33.    SAMPLE_CHECK_ERRORS(err);
34.    err = clGetEventProfilingInfo(*first_event, CL_PROFILING_COMMAND_END,
                                          sizeof(cl_ulong), &first_end, NULL);
35.    SAMPLE_CHECK_ERRORS(err);
36.    err = clGetEventProfilingInfo(*second_event, CL_PROFILING_COMMAND_START,
                                       sizeof(cl_ulong), &second_start, NULL);
37.    SAMPLE_CHECK_ERRORS(err);
38.    err = clGetEventProfilingInfo(*second_event, CL_PROFILING_COMMAND_END,
                                        sizeof(cl_ulong), &second_end, NULL);
39.    SAMPLE_CHECK_ERRORS(err);
```

```
40.    extern_kernel_time +=   ( ( (float)(first_end - first_start)*1e-9f )
                             + ( (float)(second_end - second_start)*1e-9f ) );
  }
}
```

A.12 Kernel *OBCPairLoopKernel*

```
__kernel
void OBCPairLoopKernel( const __global FLOAT *coordinates,
                        const __global FLOAT *BRO,
                        const __global FLOAT *HCT,
                        __global FLOAT *contributions,
                        const __global int *config )
{
1.  unsigned local_id = (int) get_local_id(0);
2.  unsigned group_size = (int) get_local_size(0);
3.  unsigned i = (int) get_group_id(0);

4.  __local FLOAT partialSums[256];

5.  FLOAT sum = 0.0;

6.  FLOAT bornRadiiOffset = BRO[i];

7.  for (unsigned int j = local_id;
8.       j < config[0];
9.       j += group_size)
10.  {
11.    if (i != j) {
12.      FLOAT distance = ComputeDistance(coordinates, i, j);
13.      FLOAT sk =  BRO[j] * HCT[j];
14.      FLOAT sk2 = sk*sk;

15.      FLOAT extraDistance = distance + sk;

16.      if (bornRadiiOffset < extraDistance)
17.      {
18.        FLOAT lik = 1.0 / max(bornRadiiOffset, fabs(distance - sk));
19.        FLOAT uik = 1.0 / extraDistance;
20.        FLOAT lik2 = lik * lik;
21.        FLOAT uik2 = uik * uik;

22.        FLOAT term = lik - uik + 0.25 * distance * (uik2 - lik2)
23.                    + (0.5 / distance) * log(uik / lik)
24.                     + (0.25 * sk2/distance) * (lik2 - uik2);

25.        if (bornRadiiOffset < sk - distance) {
26.          term += 2.0 * (1.0 / bornRadiiOffset - lik);
27.        }

28.        sum += 0.5 * term;
29.      }
```

```
30.    }
31.  }

32.  // Compute reduction of sum.
33.  // Based on Two-Stage reduction approach:
34.  // http://developer.amd.com/resources/documentation-articles/
35.  // articles-whitepapers/opencl-optimization-case-study-simple-reductions/
36.  partialSums[local_id] = sum;
37.  barrier(CLK_LOCAL_MEM_FENCE);
38.  for(int offset = group_size / 2;
39.      offset > 0;
40.      offset = offset / 2)
41.  {
42.    if (local_id < offset) {
43.      FLOAT other = partialSums[local_id + offset];
44.      FLOAT mine = partialSums[local_id];
45.      partialSums[local_id] = other + mine;
46.    }
47.    barrier(CLK_LOCAL_MEM_FENCE);
48.  }

49.  // Reduce
50.  if (local_id == 0)
51.    contributions[i] = bornRadiiOffset * partialSums[0];
52.}
```

A.13 Kernel *updateAlphas_kernel*

```
// Update Alphas Kernel -- ENTRY POINT
__kernel
void updateAlphas_kernel( const __global FLOAT *gbr,
                          __global FLOAT *alpha,
                          const __global FLOAT *contributions,
                          const __global FLOAT *BRO,
                          const __global int *config )
{
1.  const FLOAT ALPHA = 1.0;
2.  const FLOAT BETA = 0.8;
3.  const FLOAT GAMMA = 4.85;

4.  unsigned i = (int) get_global_id(0);

5.  if (i < config[0]) {
6.     FLOAT sum = contributions[i];

7.     FLOAT squaredSum = sum*sum;
8.     FLOAT cubicSum = sum*squaredSum;

9.     FLOAT tangSum = tanh(ALPHA * sum - BETA * squaredSum + GAMMA * cubicSum);
10.    FLOAT bornRadii = 1.0 / BRO[i] - tangSum / gbr[i];

11.    alpha[i] = 1.0 / bornRadii;
```

```
12.  }
}
```

Apèndix B

Energia nonbonding

B.1 Funció d'energia *nonbonding*

```
double NonbondingTermSerialSgb::calcEnergyGradient(
                  const double * const coords, double *grad,
                  const std::vector<NonBondPair > & nonBondingInteractions,
                  double & nonBondingVacuum, double & nonBondingSgb,
                  const SolventEnergyParams & params, bool calcGradient)
{
1.  double electrostatic_tot = 0.0;
2.  double lennard_jones_tot = 0.0;
3.  double fnbon_total_sgb = 0.0;

4.  unsigned int numberOfNonBondingInteractions = nonBondingInteractions.size();
    // For each non-bonded interaction
5.  for(unsigned int i=0; i<numberOfNonBondingInteractions; ++i)
6.  {
7.      const NonBondPair & onePair = nonBondingInteractions[i];

        // Get atoms
8.      Atom* a = onePair.first;
9.      Atom* b = onePair.second;

        // Components of the vector which goes from one point to the other
10.     double dx = coords[a->ix] - coords[b->ix];
11.     double dy = coords[a->iy] - coords[b->iy];
12.     double dz = coords[a->iz] - coords[b->iz];

        // Squared distance between points
13.     double dist2=dx*dx+dy*dy+dz*dz;

        // Inverse of the distance and squared distance
14.     double disti = 1.0/sqrt(dist2);
15.     double dist2i = disti*disti;

        // Parameters for the Lennard-jones
16.     double sig2 = onePair.getSigma2();
17.     double epsij = a->getEpsilon() * b->getEpsilon();
```

```
    // terms for variable dielectric
18.     double dielectricScreen =
19.           solventEnergyCalculator_computeDielectricScreen(a->dielectricType,
20.           b->dielectricType, params);

        // Charge product
21.     double cgij = onePair.getCharge();

        // Coulomb
22.     electrostatic_tot += cgij * disti * dielectricScreen;

        // Lennard-Jones
23.     double rr2 = sig2*dist2i;
24.     double rr6 = rr2*rr2*rr2;
25.     double rr12 = rr6*rr6; //screen param is missing!!!
26.     lennard_jones_tot += epsij * (rr12 - rr6);

27.     if (calcGradient){
28.         double dfeldr = - cgij * dist2i * dielectricScreen;
29.         double dfljdr = -6.0 * epsij * disti * (rr12 + rr12 - rr6);
30.         double dfdr = dfeldr + dfljdr;
31.         double drdx = dx*disti;
32.         double drdy = dy*disti;
33.         double drdz = dz*disti;

34.         double dfdr_sgb;
            // SGB contribution to the energy and gradient
35.         fnbon_total_sgb += SgbPairCalculator_energyAndGradientContribution(a,
36.                             b, dist2, sqrt(dist2), dfdr_sgb, cgij, params);

            // The gradient of the two atoms is updated
37.         GradientHessianPairUpdater_updateGradient(grad, a, b, dfdr+dfdr_sgb,
                                                                  drdx, drdy, drdz);
38.    } else {
39.         fnbon_total_sgb += SgbPairCalculator_energy(onePair.first,
40.                                              onePair.second, dist2, cgij, params);
41.    }
42.  }

 // Store total contribution of non bonding vacuum energy term
43.  nonBondingVacuum = lennard_jones_tot + electrostatic_tot;

 // Store total contribution of SGB non bonding energy term including penalties
44.  nonBondingSgb = fnbon_total_sgb;

45.  return (lennard_jones_tot + electrostatic_tot + fnbon_total_sgb);
}
```

B.2 Funció *GradientHessianPairUpdater_updateGradient*

```
void GradientHessianPairUpdater_updateGradient(
                          double *grad, const Atom * const a,
                          const Atom * const b, double dfdr,
                          double drdx, double drdy, double drdz)
{
   // Gradient components are computed
1. double grad_x = dfdr*drdx;
2. double grad_y = dfdr*drdy;
3. double grad_z = dfdr*drdz;

   // The gradient of the two interacting atoms is updated
4. grad[ a->ix ] += grad_x;
5. grad[ a->ix + 1 ] += grad_y;
6. grad[ a->ix + 2 ] += grad_z;

7. grad[ b->ix ] -= grad_x;
8. grad[ b->ix + 1 ] -= grad_y;
9. grad[ b->ix + 2 ] -= grad_z;
}
```

B.3 Kernel *OpenCLEnergyGradient*

```
__kernel
void OpenCLEnergyGradient(
       const __global FLOAT  *coords,
       const __global FLOAT *dielectricConstantsMatrix,
       const __global int *dielectricTypes, const __global int *indexAX,
       const __global int *indexAY,         const __global int *indexAZ,
       const __global int *indexBX,         const __global int *indexBY,
       const __global int *indexBZ,          const __global FLOAT  *epsilonA,
       const __global FLOAT  *epsilonB,     const __global int    *dielectricA,
       const __global int  *dielectricB,    const __global FLOAT  *alphaA,
       const __global FLOAT  *alphaB,       const __global FLOAT  *charge,
       const __global FLOAT  *sigma,        const __global FLOAT *params,
       volatile __global FLOAT *grad,       __global FLOAT *electrostatic,
       __global FLOAT *lennard_jones,       __global FLOAT *fnbon_sgb )
{
    // Agafo els ids del work-item
1.  unsigned local_id = (int) get_local_id(0);
2.  unsigned group_size = (int) get_local_size(0);
3.  unsigned group_id = (int) get_group_id(0);
4.  unsigned global_id = (int) get_global_id(0);

    // Array local que contindra els sumatoris de les energies
5.  __local my_struct energySums[256];
6.  energySums[local_id].electrostatic = 0.0;
7.  energySums[local_id].lennard_jones = 0.0;
8.  energySums[local_id].fnbon_sgb = 0.0;

9. if ( global_id < (int) params[4] )
```

```
10.{
      // Es recuperen els index de les coordenades del parell de atoms A i B
11.    unsigned ix_a = indexAX[global_id];
12.    unsigned iy_a = indexAY[global_id];
13.    unsigned iz_a = indexAZ[global_id];

14.    unsigned ix_b = indexBX[global_id];
15.    unsigned iy_b = indexBY[global_id];
16.    unsigned iz_b = indexBZ[global_id];

      // Components of the vector which goes from one point to the other
17.    FLOAT dx = coords[ix_a] - coords[ix_b];
18.    FLOAT dy = coords[iy_a] - coords[iy_b];
19.    FLOAT dz = coords[iz_a] - coords[iz_b];

      // Squared distance between points
20.    FLOAT dist2=dx*dx+dy*dy+dz*dz;

      // Inverse of the distance and squared distance
21.    FLOAT disti = 1.0/sqrt(dist2);
22.    FLOAT dist2i = disti*disti;

      // Parameters for the Lennard-jones
23.    FLOAT epsij = epsilonA[global_id] * epsilonB[global_id];

      // terms for variable dielectric
24.    FLOAT dielectricScreen =
25.                solventEnergyCalculator_computeDielectricScreen(
26.                    dielectricA[global_id], dielectricB[global_id],
27.                    dielectricConstantsMatrix, dielectricTypes[0]
28.                );

29.    energySums[local_id].electrostatic =
30.                      charge[global_id] * disti * dielectricScreen;

      // Lennard-Jones
31.    FLOAT rr2 = sigma[global_id] * dist2i;
32.    FLOAT rr6 = rr2 * rr2 * rr2;
33.    FLOAT rr12 = rr6 * rr6;

34.    energySums[local_id].lennard_jones = epsij * (rr12 - rr6);

35.    if ( params[3] == 1.0 ) {
36.       FLOAT dfeldr = - charge[global_id] * dist2i * dielectricScreen;
37.       FLOAT dfljdr = -6.0 * epsij * disti * (rr12 + rr12 - rr6);
38.       FLOAT dfdr = dfeldr + dfljdr;
39.       FLOAT drdx = dx*disti;
40.       FLOAT drdy = dy*disti;
41.       FLOAT drdz = dz*disti;

42.       FLOAT dfdr_sgb[1];
43.       energySums[local_id].fnbon_sgb =
44.                     SgbPairCalculator_energyAndGradientContribution(
```

```
45.                             global_id, alphaA, alphaB, dielectricA,
46.                             dielectricB, dist2, sqrt(dist2), dfdr_sgb,
47.                             charge[global_id], params,
48.                             dielectricConstantsMatrix, dielectricTypes
49.                     );

50.       GradientHessianPairUpdater_updateGradient(grad, global_id, indexAX,
51.                                indexBX, dfdr+dfdr_sgb[0], drdx, drdy, drdz
52.                          );
53.   }
54.   else if ( params[3] == 0.0) {
55.       energySums[local_id].fnbon_sgb = SgbPairCalculator_energy( global_id,
56.                           alphaA, alphaB, dielectricA, dielectricB, dist2,
57.                           charge[global_id], params, dielectricConstantsMatrix,
58.                           dielectricTypes
59.                 );
60.   }
61. }

    // Compute reduction of sum.
    // Based on Two-Stage reduction approach:
    // http://developer.amd.com/resources/documentation-articles/
    // articles-whitepapers/opencl-optimization-case-study-simple-reductions/
62. barrier(CLK_LOCAL_MEM_FENCE);
63. for(int offset = group_size / 2;
64.    offset > 0;
65.    offset = offset / 2)
66. {
67.    if (local_id < offset)
68.    {
69.       my_struct other = energySums[local_id + offset];
70.       my_struct mine = energySums[local_id];
71.       energySums[local_id].electrostatic =
72.                                   other.electrostatic + mine.electrostatic;
73.       energySums[local_id].lennard_jones =
74.                                   other.lennard_jones + mine.lennard_jones;
75.       energySums[local_id].fnbon_sgb = other.fnbon_sgb + mine.fnbon_sgb;
76.    }
77.     barrier(CLK_LOCAL_MEM_FENCE);
78. }

79. if ( local_id == 0 )
80. {
81.     electrostatic[group_id] = energySums[local_id].electrostatic;
82.     lennard_jones[group_id] = energySums[local_id].lennard_jones;
83.     fnbon_sgb[group_id]     = energySums[local_id].fnbon_sgb;
84. }
}
```

B.4 Fnció *GradientHessianPairUpdater_updateGradient* del kernel

```
void
GradientHessianPairUpdater_updateGradient(
                    volatile __global FLOAT *grad,
                    int global_id, const __global int *indexAX,
                    const __global int *indexBX, FLOAT dfdr, FLOAT drdx,
                    FLOAT drdy, FLOAT drdz )
{
    // Gradient components are computed
1.  FLOAT grad_x = dfdr*drdx;
2.  FLOAT grad_y = dfdr*drdy;
3.  FLOAT grad_z = dfdr*drdz;

    // The gradient of the two interacting iatoms is updated
4.  unsigned ix_a = indexAX[global_id];
5.  unsigned ix_b = indexBX[global_id];

6.  grad[ ix_a ]     += grad_x;
7.  grad[ ix_a + 1 ] += grad_y;
8.  grad[ ix_a + 2 ] += grad_z;

9.  grad[ ix_b ]     -= grad_x;
10. grad[ ix_b + 1 ] -= grad_y;
11. grad[ ix_b + 2 ] -= grad_z;
}
```

B.5 Funcions atòmiques

```
inline void
float_atomic_add(volatile __global FLOAT *source, const FLOAT operand){
    union {
        ulong uVal;
        FLOAT floatVal;
    } newVal;
    union {
        ulong uVal;
        FLOAT floatVal;
    } prevVal;
    do {
        prevVal.floatVal = *source;
        newVal.floatVal = prevVal.floatVal + operand;
    } while( atom_cmpxchg((volatile __global ulong *) source,
                          prevVal.uVal, newVal.uVal) != prevVal.uVal);

}

inline void
float_atomic_sub(volatile __global FLOAT *source, const FLOAT operand){
    union{
        ulong uVal;
```

```
        FLOAT floatVal;
    } newVal;
    union {
        ulong uVal;
        FLOAT floatVal;
    } prevVal;
    do {
        prevVal.floatVal = *source;
        newVal.floatVal = prevVal.floatVal - operand;
    } while( atom_cmpxchg((volatile __global ulong *) source,
                          prevVal.uVal, newVal.uVal) != prevVal.uVal);
}
```

B.6 Funció *GradientHessianPairUpdater_updateGradient* del kernel amb atòmiques

```
void
GradientHessianPairUpdater_updateGradient(
                volatile __global FLOAT *grad, int global_id,
                const __global int *indexAX,  const __global int *indexBX,
                FLOAT dfdr, FLOAT drdx, FLOAT drdy, FLOAT drdz
          )
{
    // Gradient components are computed
    FLOAT grad_x = dfdr*drdx;
    FLOAT grad_y = dfdr*drdy;
    FLOAT grad_z = dfdr*drdz;

    // The gradient of the two interacting iatoms is updated
    unsigned ix_a = indexAX[global_id];
    unsigned ix_b = indexBX[global_id];

    float_atomic_add(&grad[ ix_a ], grad_x);
    float_atomic_add(&grad[ ix_a + 1 ], grad_y);
    float_atomic_add(&grad[ ix_a + 2 ], grad_z);

    float_atomic_sub(&grad[ ix_b ], grad_x);
    float_atomic_sub(&grad[ ix_b + 1 ], grad_y);
    float_atomic_sub(&grad[ ix_b + 2 ], grad_z);
}
```

B.7 Kernel *OpenCLEnergyGradient* de la quarta versió.

```
__kernel
void OpenCLEnergyGradient(
       const __global FLOAT  *coords,
       const __global FLOAT *dielectricConstantsMatrix,
       const __global int *dielectricTypes, const __global int *indexAX,
       const __global int *indexAY,         const __global int *indexAZ,
       const __global int *indexBX,         const __global int *indexBY,
```

```
        const __global int *indexBZ,          const __global FLOAT  *epsilonA,
        const __global FLOAT  *epsilonB,     const __global int    *dielectricA,
        const __global int  *dielectricB,    const __global FLOAT  *alphaA,
        const __global FLOAT  *alphaB,       const __global FLOAT  *charge,
        const __global FLOAT  *sigma,        const __global FLOAT *params,
        __global FLOAT *electrostatic,       __global FLOAT *lennard_jones,
        __global FLOAT *fnbon_sgb,           __global FLOAT *gradientX,
        __global FLOAT *gradientY,           __global FLOAT *gradientZ )
{
    // Agafo els ids del work-item
1.  unsigned local_id = (int) get_local_id(0);
2.  unsigned group_size = (int) get_local_size(0);
3.  unsigned group_id = (int) get_group_id(0);
4.  unsigned global_id = (int) get_global_id(0);

    // Array local que contindra els sumatoris de les energies
5.  __local my_struct energySums[256];
6.  energySums[local_id].electrostatic = 0.0;
7.  energySums[local_id].lennard_jones = 0.0;
8.  energySums[local_id].fnbon_sgb = 0.0;

9. if ( global_id < (int) params[4] )
10.{
       // Es recuperen els index de les coordenades del parell de atoms A i B
11.    unsigned ix_a = indexAX[global_id];
12.    unsigned iy_a = indexAY[global_id];
13.    unsigned iz_a = indexAZ[global_id];

14.    unsigned ix_b = indexBX[global_id];
15.    unsigned iy_b = indexBY[global_id];
16.    unsigned iz_b = indexBZ[global_id];

       // Components of the vector which goes from one point to the other
17.    FLOAT dx = coords[ix_a] - coords[ix_b];
18.    FLOAT dy = coords[iy_a] - coords[iy_b];
19.    FLOAT dz = coords[iz_a] - coords[iz_b];

       // Squared distance between points
20.    FLOAT dist2=dx*dx+dy*dy+dz*dz;

       // Inverse of the distance and squared distance
21.    FLOAT disti = 1.0/sqrt(dist2);
22.    FLOAT dist2i = disti*disti;

       // Parameters for the Lennard-jones
23.    FLOAT epsij = epsilonA[global_id] * epsilonB[global_id];

       // terms for variable dielectric
24.    FLOAT dielectricScreen =
25.                solventEnergyCalculator_computeDielectricScreen(
26.                    dielectricA[global_id], dielectricB[global_id],
27.                    dielectricConstantsMatrix, dielectricTypes[0]
28.                );
```

```
29.    energySums[local_id].electrostatic =
30.                         charge[global_id] * disti * dielectricScreen;

       // Lennard-Jones
31.    FLOAT rr2 = sigma[global_id] * dist2i;
32.    FLOAT rr6 = rr2 * rr2 * rr2;
33.    FLOAT rr12 = rr6 * rr6;

34.    energySums[local_id].lennard_jones = epsij * (rr12 - rr6);

35.    if ( params[3] == 1.0 ) {
36.       FLOAT dfeldr = - charge[global_id] * dist2i * dielectricScreen;
37.       FLOAT dfljdr = -6.0 * epsij * disti * (rr12 + rr12 - rr6);
38.       FLOAT dfdr = dfeldr + dfljdr;
39.       FLOAT drdx = dx*disti;
40.       FLOAT drdy = dy*disti;
41.       FLOAT drdz = dz*disti;

42.       FLOAT dfdr_sgb[1];
43.       energySums[local_id].fnbon_sgb =
44.                      SgbPairCalculator_energyAndGradientContribution(
45.                             global_id, alphaA, alphaB, dielectricA,
46.                             dielectricB, dist2, sqrt(dist2), dfdr_sgb,
47.                             charge[global_id], params,
48.                             dielectricConstantsMatrix, dielectricTypes
49.                      );
50.       FLOAT auxDfdr = dfdr+dfdr_sgb[0];
51.       FLOAT grad_x  = auxDfdr*drdx;
52.       FLOAT grad_y  = auxDfdr*drdy;
53.       FLOAT grad_z  = auxDfdr*drdz;

54.       gradientX[global_id] = grad_x;
55.       gradientY[global_id] = grad_y;
56.       gradientZ[global_id] = grad_z;

57.   }
58.   else if ( params[3] == 0.0) {
59.       energySums[local_id].fnbon_sgb = SgbPairCalculator_energy( global_id,
60.                         alphaA, alphaB, dielectricA, dielectricB, dist2,
61.                         charge[global_id], params, dielectricConstantsMatrix,
62.                         dielectricTypes
63.                  );
64.   }
65. }

    // Compute reduction of sum.
    // Based on Two-Stage reduction approach:
    // http://developer.amd.com/resources/documentation-articles/
    // articles-whitepapers/opencl-optimization-case-study-simple-reductions/
66. barrier(CLK_LOCAL_MEM_FENCE);
67. for(int offset = group_size / 2;
68.    offset > 0;
```

```
69.    offset = offset / 2)
70. {
71.    if (local_id < offset)
72.    {
73.       my_struct other = energySums[local_id + offset];
74.       my_struct mine = energySums[local_id];
75.       energySums[local_id].electrostatic =
76.                                  other.electrostatic + mine.electrostatic;
77.       energySums[local_id].lennard_jones =
78.                                  other.lennard_jones + mine.lennard_jones;
79.       energySums[local_id].fnbon_sgb = other.fnbon_sgb + mine.fnbon_sgb;
80.    }
81.     barrier(CLK_LOCAL_MEM_FENCE);
82. }

83. if ( local_id == 0 )
84. {
85.     electrostatic[group_id] = energySums[local_id].electrostatic;
86.     lennard_jones[group_id] = energySums[local_id].lennard_jones;
87.     fnbon_sgb[group_id]     = energySums[local_id].fnbon_sgb;
88. }
}
```

B.8 Kernel *ComputeGradientAdd.*

```
__kernel
void ComputeGradientAdd(
          const __global int *indexos,     const __global int *numberOfIndexos,
          const __global int *prefixSums,  __global FLOAT *grad,
          __global FLOAT *gradientX,       __global FLOAT *gradientY,
          __global FLOAT *gradientZ,       const __global FLOAT *params)
{
    // Declarar les arrays locals
1.  __local FLOAT valorsX[500];
2.  __local FLOAT valorsY[500];
3.  __local FLOAT valorsZ[500];

    // Agafo el group_id
4.  unsigned group_id = (int) get_group_id(0);

    // Agafar local_id
5.  unsigned local_id = (int) get_local_id(0);

    // Agafar local_size
6.  unsigned local_size = (int) get_local_size(0);

    // Agafar el numero de indexos
7.  unsigned N = numberOfIndexos[group_id];

    // Agafar els offsets
8.  unsigned offset = prefixSums[group_id];
```

```
    // Fer la lectura dels gradients en paralel
9.  for ( unsigned i = local_id; i < N; i += local_size )
10. {
11.     unsigned pos = indexos[offset+i];
12.     valorsX[i]  = gradientX[pos];
13.     valorsY[i]  = gradientY[pos];
14.     valorsZ[i]  = gradientZ[pos];
15. }

    // Sincronitzar threads
16. barrier(CLK_LOCAL_MEM_FENCE);

    // Sols el thread 0 fara les sumes
17. if ( local_id == 0 )
18. {
19.     FLOAT sumOfAtomX = 0;
20.     FLOAT sumOfAtomY = 0;
21.     FLOAT sumOfAtomZ = 0;

22.     for ( unsigned i = 0; i < N; i++)
23.     {
24.         sumOfAtomX += valorsX[i];
25.         sumOfAtomY += valorsY[i];
26.         sumOfAtomZ += valorsZ[i];
27.    }

        // Guardar el resultat de les sumes en el vector de gradient
28.     unsigned indexOfGradient   = group_id*3;
29.     grad[indexOfGradient]      = sumOfAtomX;
30.     grad[indexOfGradient + 1 ] = sumOfAtomY;
31.     grad[indexOfGradient + 2 ] = sumOfAtomZ;
32. }
}
```

B.9 Kernel *ComputeGradientSub.*

```
__kernel
void ComputeGradientSUb(
          const __global int *indexos,     const __global int *numberOfIndexos,
          const __global int *prefixSums,  __global FLOAT *grad,
          __global FLOAT *gradientX,       __global FLOAT *gradientY,
          __global FLOAT *gradientZ,       const __global FLOAT *params)
{
    // Declarar les arrays locals
1.  __local FLOAT valorsX[500];
2.  __local FLOAT valorsY[500];
3.  __local FLOAT valorsZ[500];

    // Agafo el group_id
4.  unsigned group_id = (int) get_group_id(0);

    // Agafar local_id
```

```
5.  unsigned local_id = (int) get_local_id(0);

    // Agafar local_size
6.  unsigned local_size = (int) get_local_size(0);

    // Agafar el numero de indexos
7.  unsigned N = numberOfIndexos[group_id];

    // Agafar els offsets
8.  unsigned offset = prefixSums[group_id];

    // Fer la lectura dels gradients en paralel
9.  for ( unsigned i = local_id; i < N; i += local_size )
10. {
11.     unsigned pos = indexos[offset+i];
12.     valorsX[i]  = gradientX[pos];
13.     valorsY[i]  = gradientY[pos];
14.     valorsZ[i]  = gradientZ[pos];
15. }

    // Sincronitzar threads
16. barrier(CLK_LOCAL_MEM_FENCE);

    // Sols el thread 0 fara les sumes
17. if ( local_id == 0 )
18. {
19.     FLOAT sumOfAtomX = 0;
20.     FLOAT sumOfAtomY = 0;
21.     FLOAT sumOfAtomZ = 0;

22.     for ( unsigned i = 0; i < N; i++)
23.     {
24.         sumOfAtomX += valorsX[i];
25.         sumOfAtomY += valorsY[i];
26.         sumOfAtomZ += valorsZ[i];
27.    }

        // Guardar el resultat de les sumes en el vector de gradient
28.     unsigned indexOfGradient   = group_id*3;
29.     grad[indexOfGradient]      = sumOfAtomX;
30.     grad[indexOfGradient + 1 ] = sumOfAtomY;
31.     grad[indexOfGradient + 2 ] = sumOfAtomZ;
32. }
}
```

Apèndix C

Bitonic Sort

C.1 Algoritme

L'ordenació és un dels problemes més importants en les ciències de la computació i la capacitat d'ordenar grans quantitats de dades de manera eficient és absolutament critica. Els algoritmes d'ordenació tradicionalment han sigut implementats en CPUS i sempre han funcionat molt be, però la possibilitat d'implementar-los en GPUS esdevé un repte a nivell computacional. En el model de programació amb OpenCL, nosaltres tenim tasques i paral·lelisme de dades i aconseguir un algoritme d'ordenació que treballi amb OpenCL és tot un repte, sobretot des del punt de vista algorítmic, és a dir, com es pot crear un algoritme que aprofite el paral·lelisme de les dades.

Els mètodes d'ordenació es poden categoritzar en dos grans grups: *data-driven* i *data-independent*. Els algoritmes *data-driven* executen el següent pas de l'algoritme depenen del valor de la clau sota consideració, per exemple, l'algoritme QuickSort. Els algoritmes *data-independent* són rigids respecte a aquesta perspectiva i no fan cap canvi en l'ordre del processament d'acord amb els valors de les claus. L'algoritme de *Bitonic Sort*, doncs, és un exemple clàssic dels algoritmes d'ordenació *data-independent* i pot ser representat per xarxes d'ordenació (apèndix D.1).

L'algoritme de bitonic sort treballa comparant dos elements en qualsevol punt en el temps. El que significa això és que es consumeixen dos imputs i es decideix si A és igual a B, si A és menor que B, o si A és més gran que B. L'algoritme és un exemple dels algoritmes *non-adamptive*[i].

En l'ordenació amb *bitonic sort* es té un imput que s'anomena *bitonic sequence*. Aquest tipus de seqüències són aquelles que augmenten (disminueixen) monòtonament fins arribar a un màxim (mínim) i, després, disminueixen (augmenten) monòtonament fins arribar a un mínim (màxim).

En general, s'han de considera una sèrie d'escenaris per determinar si l'imput és adequat per a la l'ordenació amb *bitonic sort*. De fet, quan es volen ordenar dades amb un algoritme d'ordenació particular, el que es vol és veure si l'imput ja esta ordenat en base algun criteri. Pel que fa al context de l'algoritme *bitonic sort*, el criteri que s'aplica en la seqüència d'entrada és el que es coneix com *bitonic split sequence*[ii] o *arbitrary sequence* i s'ha d'aplicar fins arribar a l'estat final d'ordenació.

[i] Un algoritme d'ordenació no adaptatiu (non-adaptive) és aquell on la seqüència d'operacions realitzades és independent de l'ordre de les dades, també conegut com data-independent.

[ii] El *bitonic split* és una operació de divisió sobre la seqüència bitonica, tal que si $a_i > a_{i+n/2}$ (on $1 \leq i < n$) els dos elements s'intercanvien i l'operació genera dos seqüències bitoniques A i B, tal que els elements de A són més petits que els elements de B.

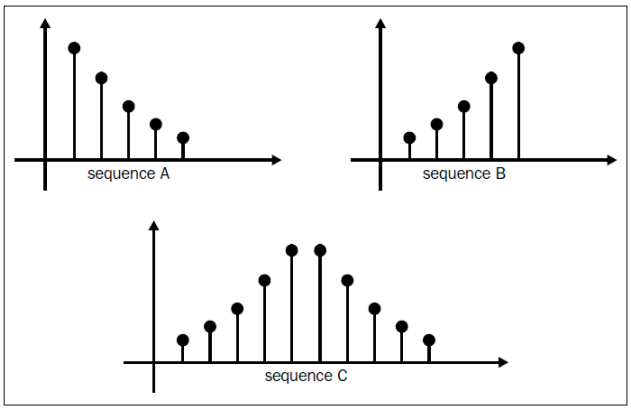


Figura C.1: Ordenació de quatre claus amb Xarxa d'ordenació.

La figura C.1 mostra com dos seqüències bitoniques (seqüències A i B) es poden combinar amb un altre seqüència més llarga (seqüència C) mitjançant l'aplicació d'aquest algoritme d'ordenació.

En la situació on es rep una seqüència arbitraria, és a dir, desordenada i no ordenada bitonicament, bàsicament es té de de produir un seqüencial bitonica a partir de l'entrada i després aplicar la tècnica de les divisions (*bitonic splits*) descrita fins arribar a l'estat d'ordenació final.

La figura C.2 il·lustra com l'algoritme de les divisions (bitonic split) opera en seqüències separades i produeix la seqüència final ordenada.

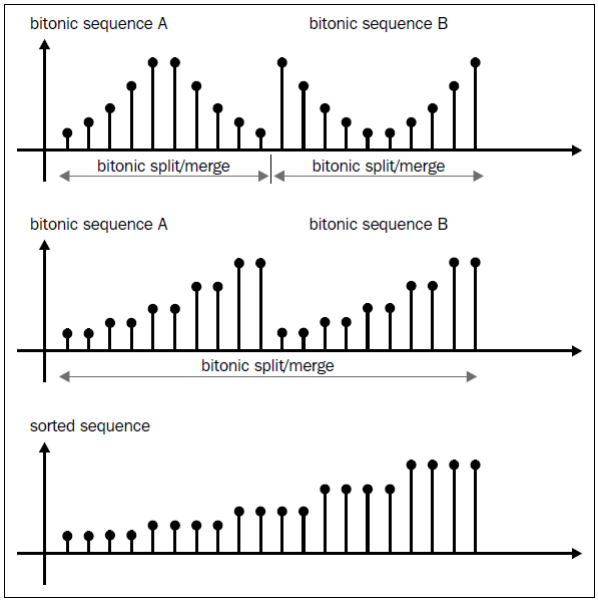


Figura C.2: Ordenació a partir de dues seqüències

En qualsevol cas, es sap quan l'algoritme acaba si al aplica *split bitonic* s'obté una mida dos, perquè en aquest moment es tracta d'una operació de comparació entre A i B, on A és més gran o igual que B o B és més gran o igual que A, i l'únic que farà falta serà posar els elements en la posició que els pertoqui.

Apèndix D

Xarxes d'ordenació

D.1 Xarxes d'ordenació

Un model que es fa servir freqüentment a l'hora d'estudiar els algoritmes d'ordenació no adaptatius (non-adaptive), és el que la literatura tècnica anomena *Xarxa d'Ordenació*. A aquest model també se l'anomena *Xarxa de Comparació*, i és la idea que hi ha darrere de l'algoritme bitonic sort. Les xarxes d'ordenació són el model més simple d'estudi i representen una màquina abstracta que solament accedeix a les dades a través d'operacions de comparació i canvi.

La figura D.1 il·lustra com s'ordenen quatre claus mitjançant una xarxa d'ordenació. Pe convenció, es pinta la xarxa d'ordenació de n imputs com una seqüència de n línies horitzontals, amb comparadors que connecten un parell de línies. Ens podem imaginar que les claus que es volen ordena viatgen d'esquerra a dreta a traves de la xarxa amb un parell de nombres que s'han d'intercanviar posant a dalt el més petit quan es troba el comparador.

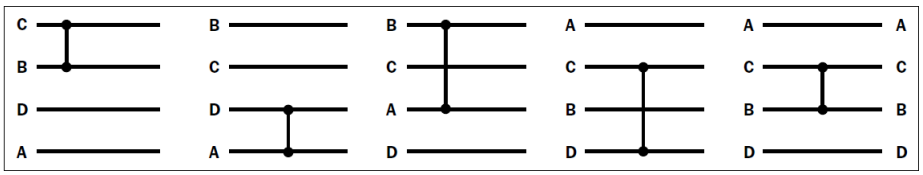


Figura D.1: Ordenació de quatre claus amb Xarxa d'ordenació.

Aquest tipus de xarxes d'ordenació mostren una propietat particular: la xarxa es prou llarga per a que els comparadors no es superposen, fet que permet realitzar les operacions de comparació en paral·lel. Per aplicar paral·lelisme és necessari veure que quines agrupacions de comparacions es poden fer en paral·lel i quines s'han de realitzar en una segona etapa. La figura D.2 mostra una manera optima d'ordenar quatre enters amb una xarxa d'ordenació de manera paral·lela. La figura mostra les agrupacions de comparacions que es poden fer en paral·lel en cada pas.

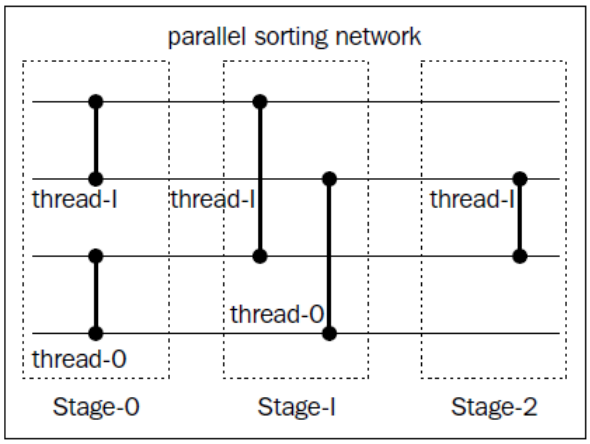


Figura D.2: Ordenació de quatre claus amb Xarxa d'ordenació en paral·lel.

La xarxa de la figura és una de les possibles xarxes d'ordenació per ordena quatre claus en paral·lel. Es pot veure que per realitzar l'ordenació s'han de realitzar tres passos i cinc operacions de comparació.

Apèndix E

Prefix Sum

E.1 algoritme prefix sum

En ciencies de la computació, l'algoritme prefix sum en una seqüència de números x_0, x_1, x_2,... és una segona seqüència y_0, y_1, y_2, de sumes de prefixes de la seqüència d'entrada.

$$y_0 = x_0$$

$$y_1 = x_0 + x_1$$

$$y_2 = x_0 + x_1 + x_2$$

Per exemple, la seqüència prefix sums dels números naturals son els números triangulars:

input numbers	1	2	3	4	5	6	...
prefix sums	1	3	6	10	15	21	...

Figura E.1: Prefix sums de una seqüència de nombres naturals.

Una seqüència de prefix sums es trivial de calcular en models seqüencials de càlcul utilitzant la fórmula $y_i = y_{i-1} + x_i$ per a calcular cada valor de sortida en l'ordre de la seqüència. No obstant, malgrat la seva facilitat de càlcul, les sumes de prefixos són útils en certs algoritmes com *counting sort*, on formen la base de la funció d'scan d'ordre superior en els llenguatges de programació funcionals. L'algoritme de prefix sums també es estudiat en el camp de la computacio paral·lela.

De manera abstracta, l'algoritme de prefix sums solament requereix l'operació associativa $\oplus$, fent-lo útil en moltes aplicacions de calcul.

Bibliografia

[1] *Efficient Nonbonded Interactions for Molecular Dynamics on a Graphics Processing Unit (Peter Eastman, and Vijay S.Pande - October 2009)*

[2] *Accelerating Molecular Dynamic Simulation on Graphics Processing Units (Mark S.Friedrichs, Peter Eastman, Vishal Vaidyanathan, Mike Houston, Scott Legrand, Adam L.Beberg, Daniel L.Ensign, Christopher M.Bruns, and Viajay S.Pande - February 2009)*

[3] *Routine Microsecond Molecular Dynamics Simulations with AMBER on GPUs. 1. Generalized Born (Andreas W.Götz Mark J.Williamson, Dong Xu, Duncan Poole, Scott Le Grand, and Ross C.Walker)*

[4] *PELE: Protein Energy Landscape Exploration. A Novel Monte Carlo Based Technique (Kenneth W. Borrelli, Andreas Vitalis, Raul Alcantara, and Victor Guallar - July 2005)*

[5] *Anisotropy of Fluctuation Dynamics of Proteins with an Elastic Network Model (A.R. Atilgan, S.R. Durell, R.L. Jernigan, M.C. Demirel, O. Keskin, and I. Bahar - January 2009)*

[6] *Exploring Protein Native States and Large-Scale Conformational Changes With a Modified Generalized Born Model (Alexey Onufriev, Donald Bashford, and David A.Case)*

[7] *Introduction to OpenCL^{TM} Programming (Advances Micro Devices Inc)*

[8] *Heterogeneous Computing with OpenCL (Benedict R.Gaster, Lee Howes, David R.Kaeli, Perhaad Mistry, and Dana Schaa)*

[9] *OpenCL Programming by Example - A comprehensive guide on OpenCL programming with examples (Ravishekhar Banger , and Koushik Bhattacharyya)*

[10] *OpenCL Programming Guide (Aaftab Munshi, Benedict R.Gaster, Timothy G.Mattson, James Fung, and Dan Ginsburg)*

[11] *OpenCL Parallel Programming Development Cookbook (Raymond Tay)*

[12] *OpenCL in action (Matthew Scarpino)*

[13] *Barcelona Supercomputing Center* – https://www.bsc.es/

[14] *PELE - Protein Energy Landscape Exploration* – https://pele.bsc.es/pele.wt

[15] *Pharmacelera - Fast and accurate in-silico solutions for drug discovery* – http://www.pharmacelera.com/