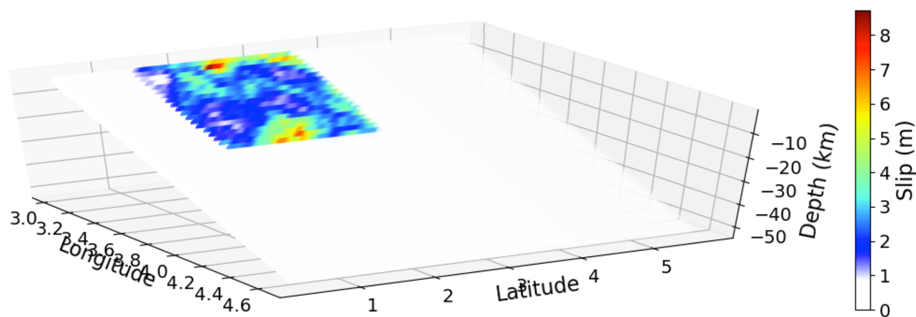
Treball de Final de Màster

Paral·lelització del codi del simulador de terratrèmols MudPy amb CUDA

Àrea: Computació d'Altes Prestacions

Autor: Marc Coll Carrillo
Director: Ivan Rodero Castro

June 29, 2021



Universitat Oberta de Catalunya

Full del projecte

Títol	Paral·lelització del codi del simulador de terratrèmols MudPy amb CUDA
Autor	Marc Coll Carrillo
Tutor	Ivan Rodero Castro
Data	June 29, 2021
Programa	Màster en Enginyeria Informàtica
Àrea	Computació d'Altes Prestacions
Idioma	Català
Paraules clau	Algorismes paral·lels Computació d'Altres Prestacions CUDA GPU GPGPU Python Numba MudPy Simulació de terratrèmols

Abstract

MudPy is an earthquake simulator developed by a team of geologists and geophysicists. It works well an serves its purpose, but it has some weaknesses, the main one being its relative poor performance. This problem limits its usefulness, since a normal simulation can take hours or even days. The main goal of this project is to parallelize part of MudPy's code with CUDA, so that simulations can work faster in those machines where a GPU is available. Since MudPy is mainly written in Python, Numba will be used, which is a platform that allows using all the power of CUDA directly in Python, without having to rewrite the code to C or C++. Other possible strategies to increase the performance are also explored.

Resum

MudPy és una aplicació per simular terratrèmols desenvolupada per un equip de geòlegs i geofísics. Funciona bé i compleix amb la seva tasca, però té algunes carències, la més important de les quals és la seva relativa lentitud. Aquests problemes en limiten la seva utilitat, ja que una simulació normal pot trigar hores o, fins i tot, dies. El principal propòsit d'aquest projecte es paral·lelitzar una part del codi del MudPy per a ser executada mitjançant CUDA, de manera que les simulacions puguin funcionar més ràpidament en aquelles màquines que disposin d'una GPU. Com que el MudPy està escrit principalment en Python, es farà servir Numba, una plataforma que permet aprofitar tota la potència de CUDA directament en Python, sense haver de migrar el codi a C o C++. També s'exploren altres possibles estratègies alternatives que permetin incrementar-ne el rendiment.

El Dr. Ivan Rodero Castro certifica que l'estudiant Marc Coll Carrillo ha elaborat el treball sota la seva direcció, i autoritza la presentació d'aquesta memòria per a la seva avaluació.

Firma del director:

Índex

1 Introducció **7**
1.1 El simulador de terratrèmols MudPy 7
1.2 La plataforma CUDA 8

2 Anàlisi del MudPy **10**
2.1 Funcionament 10
2.2 Anàlisi del rendiment 11
2.3 Anàlisi de la usabilitat 12

3 Abast i objectius del projecte **15**
3.1 Motivacions 15
3.2 Objectius 16
3.3 Abast 17

4 Planificació del projecte **18**
4.1 Anàlisi de riscos 18
4.2 Lliurables 19
4.3 Planificació 20

5 Pla de proves **22**
5.1 Proves de validació del codi 22
5.2 Proves de rendiment 23

6 Implementació **25**
6.1 L'entorn de desenvolupament 25
6.2 Modificant el codi 26
6.3 Modificacions per millorar el rendiment 28
6.4 Modificacions per millorar la usabilitat 39
6.5 Problemes trobats 43

7 Proves **46**

7.1 L'entorn de proves . . . 46
7.2 Script de proves . . . 46
7.3 Proves realitzades . . . 50
7.4 Interpretació dels resultats . . . 52

8 Possibles millores futures **54**
8.1 Optimitzar el codi . . . 54
8.2 Migrar el codi a C o C++ . . . 55
8.3 Fer servir alternatives a Numba . . . 55
8.4 HTC i grid computing . . . 55

9 Conclusió **58**
9.1 Objectius assolits . . . 58
9.2 Seguiment de la planificació . . . 58
9.3 Coneixements aplicats . . . 59
9.4 Coneixements adquirits . . . 59
9.5 Valoració personal . . . 60

Bibliografia **61**

Capítol 1

Introducció

El principal propòsit d'aquest projecte es analitzar i paral·lelitzar una part del codi del simulador de terratrèmols MudPy per a ser executada mitjançant CUDA, de manera que les simulacions puguin funcionar més ràpidament en aquelles màquines que disposin d'una GPU.

En aquest capítol introduiré alguns conceptes bàsics necessaris per posar en context el projecte i els seus objectius.

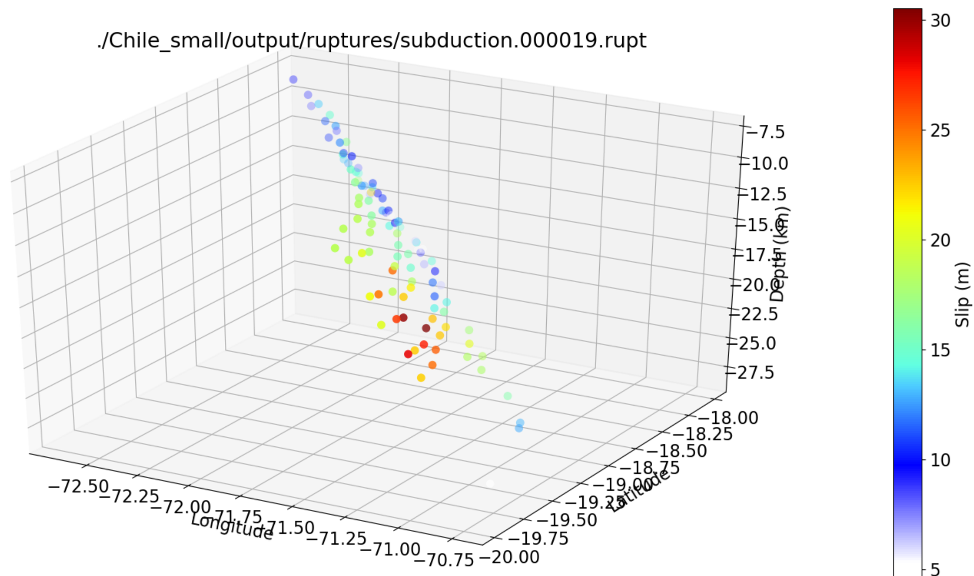
1.1 El simulador de terratrèmols MudPy

MudPy[1][2] és una aplicació per simular terratrèmols desenvolupada per un equip de geòlegs i geofísics:

- Diego Melgar
- Jiun-Ting "Tim" Lin
- Qingkai Kong
- Christine J. Ruhl

Funciona bé i compleix amb la seva tasca, però té algunes carències, la més important de les quals és la seva relativa lentitud. Aquests problemes en limiten la seva utilitat, ja que una simulació normal pot trigar hores o, fins i tot, dies. Idealment, els seus autors voldrien poder executar centenars de simulacions, però actualment és inviable.

S'han fet intents d'incrementar-ne la velocitat fent servir MPI[3], un estàndard en la programació d'aplicacions paral·leles per a ser executades en supercomputadors, però no ha millorat gaire.



Exemple de simulació generada pel MudPy.

En aquest projecte exploraré algunes possibles millores a l'aplicació, centrant-me en la velocitat però sense descartar altres aspectes, com ara la usabilitat.

1.2 La plataforma CUDA

En els darrers anys s'ha anat consolidant un concepte anomenat GPGPU[4] (*General-Purpose computing on Graphics Processing Units*), que consisteix en fer servir els processadors continguts dins de les GPUs de les targetes gràfiques modernes per realitzar computacions de propòsit general. És a dir, agafar processadors que normalment s'utilitzen per a aplicacions gràfiques en 2D i 3D i emprar-los per fer la mena de càlculs que normalment es realitzen sobre un processador normal i corrent.

Però els nuclis d'una GPU treballen a velocitats inferiors a les d'una CPU tradicional i són menys potents. Per què fer-los servir, aleshores? El principal avantatge de les GPUs és que un sol xip pot contenir milers de nuclis. Per exemple: la targeta GeForce RTX 2080 Ti de Nvidia[5], una de les més recents que han sortit al mercat, funciona a una velocitat de rellotge de 1,35 GHz. És una velocitat relativament lenta, doncs la majoria de CPUs actuals funcionen a entre 3 i 5 GHz[6]. Però

mentre que una CPU normal pot tenir de l'ordre de 10 o 20 nuclis, aquesta targeta en té 4352.

Segons la llei de Gustafson[7], la part no paral·lela d'un programa decreix a mesura que augmenta la mida del problema. Per tant, qualsevol problema suficientment gran es pot paral·lelitzar de manera eficient. I això és precisament el que fa tant atractives a les GPUs: si tenim un problema molt gran que es pot parl·lelitzar de forma massiva, una GPU ens permetrà multiplicar el rendiment. Per fer-nos una idea de la diferència que això suposa, el rendiment d'una CPU actual s'acostuma a mesurar en GFLOPS/s, mentre que el de una GPU es mesura en TFLOPS, és a dir, 3 ordres de magnitud més gran.

Programar una GPU, tanmateix, no és trivial. Afortunadament, els programadors disposen d'eines com CUDA[8], una plataforma de programació paral·lela desenvolupada per Nvidia[9] que permet explotar de manera relativament senzilla la potència de les GPUs per a realitzar càlculs.

Actualment, una de les aplicacions més populars de la GPGPU és la de minar criptomonedes, especialment Bitcoins[10]. La demanda deguda a això és tan gran que ha provocat una escassetat mundial de GPUs[11][12]. S'ha arribat al punt en el que Nvidia, en un intent de frenar aquesta falta de targetes que té frustrats a molts *gamers*, ha decidit treure al mercat targetes específiques per a aquest propòsit[13].

CUDA és una tecnologia propietària, pensada per executar-se en GPUs de Nvidia, però hi ha altres alternatives de codi obert i multiplataforma, com ara OpenCL[14]. Aquesta plataforma és mantinguda per un consorci tecnològic sense ànim de lucre anomenat Khronos Group[15], que també és el responsable d'estàndards tan coneguts com OpenGL[16].

CUDA està pensat per a llenguatges com C, C++ o Fortran. Però hi ha una altra plataforma, anomenada Numba[17], que forma part del projecte CUDA Python[18] i que permet aprofitar tota la potència de CUDA directament amb Python. Aquesta plataforma és la que faré servir per a fer el projecte, ja que la major part del codi del MudPy està escrit en Python i migrar-lo a un altre llenguatge seria un procés massa costós i propens a errors.

Capítol 2

Anàlisi del MudPy

El primer que hom descobreix quan analitza el simulador MudPy és que en realitat no es tracta d'un programa monolític, sinó d'un conglomerat heterogeni de diferents subprogrames, cadascun d'ells amb una tasca concreta i escrits en diferents llenguatges de programació (Perl, C, Fortran, MPI...). El conjunt es manté unit fent servir Python com a llenguatge principal.

2.1 Funcionament

Per poder generar una simulació, el programa s'executa en diferents passos:

1. **Inicialitzar el projecte**
   Això genera una estructura de fitxers i directoris on es guarden la configuració i les dades d'entrada i sortida.
2. **Calcular les ruptures**
   Aquest pas genera una sèrie de ruptures aleatòries.
3. **Calcular les funcions de Green**
   Aquest pas calcula les funcions de Green sobre una sèrie de respostes d'impulsos. Això genera unes *base waveforms* sobre les que treballar.
4. **Calcular les inversions**
   Aquest pas calcula ones del terratrèmol (*earthquake waveforms*) mitjançant la combinació lineal de les ruptures i les *base waveforms*.

El primer pas és trivial. Els dos següents poden trigar força temps, però només cal executar-los una vegada. I el darrer pas és el que ens interessa paral·lelitzar, doncs

triga força i ens interessa poder executar-lo moltes vegades (centenars o, fins i tot, milers).

Per poder executar cadascun d'aquests passos cal modificar una sèrie de variables de configuració dins d'un script en Python, que és l'encarregat d'anar cridant les diferents funcions.

2.2 Anàlisi del rendiment

Mirant el codi en Python del MudPy, es pot veure com el darrer pas acaba cridant al mètode `forward.waveforms_fakequakes_dynGF`, que es troba dins del fitxer forward.py. Per tal d'identificar els principals colls d'ampolla, vaig modificar el codi afegint un cronòmetre molt senzill que de tant en tant anés mostrant el temps transcorregut:

```
[...]
import time
[...]

    start_time = time.time()
    [...]
    print("1 elapsed time: {0} seconds".format(time.time() - start_time))
    [...]
    print("2 elapsed time: {0} seconds".format(time.time() - start_time))
```

D'aquesta manera vaig descobrir que el principal coll d'ampolla és, amb diferència, la crida a la funció `get_fakequakes_G_and_m_dynGF`, dins de `loop_sources`. La resta de passos, exceptuant la càrrega inicial de paràmetres, triguen una fracció de segon, però aquest triga entre 3 i 4 segons, i s'executa moltes vegades. A més, els paràmetres de la funció no depenen de cap altre execució. Per tant, la funció `loop_sources` és la candidata ideal per a ser paral·lelitzada. De fet, es pot veure en el codi que ja s'han fet intents de paral·lelitzar-la abans, però sense gaire èxit:

```
#Parallel(n_jobs=ncpus,backend='loky')(delayed(loop_sources)(ksource) for ksource in
    range(hot_start,len(all_sources)))
#Oh no, this is REALLY slow why?
ps=[]
for ksource in range(hot_start,len(all_sources)):
    p = mp.Process(target=loop_sources, args=([ksource]) )
    ps.append(p)
```

Si filem una mica més prim, veurem que `loop_sources` crida a dues funcions: `sta_close_to_rupt` i `get_fakequakes_G_and_m_dynGF`. Totes dues que executen una sèrie de càlculs fent servir bucles fàcilment paral·lelitzables. Teòricament, si aconsegueixo paral·lelitzar aquestes funcions amb Numba[17] aconseguiré que les simulacions siguin molt més ràpides.

2.3 Anàlisi de la usabilitat

Abans d'analitzar la usabilitat del programa, cal tenir en compte que no ha estat desenvolupat per informàtics, sinó per geòlegs i geofísics, i que ha estat creat des d'un punt de vista purament utilitari. Ara bé, tot i que sigui útil i compleixi amb la seva funció, això no vol dir que no es pugui beneficiar d'algunes millores.

Actualment, la interfície del programa (per dir-ho d'alguna manera) té aquest aspecte:

```
[...]

########                          GLOBALS                           ########
home='/Users/dmelgar/Slip_inv/'
project_name='2015_Nepal'
run_name='static'
########################################################################

#####              What-do-you-want-to-do flags, 1=do, 0=leave be          #####
init=1 #Initalize project
make_green=0 #Compute GFs
make_synthetics=0 #Compute synthetics for a given model at given stations
solve=0  # =1 solves forward problem or runs inverse calculation, =0 does nothing
#######################################################################

###############            Green function parameters               #############
hot_start=0  #Start at a certain subfault number
static=1  #=1 computes static GFs only, =0 computes the complete waveform
tsunami=False
model_name='nepal.mod'   #Velocity model
rupture_name='usgs.rupt'   #Rupture model, not needed for inversion
fault_name='nepal.fault'    #Fault geometry
station_file='nepal.sta'   #Station distribution
GF_list='nepal.gflist'#What GFs are to be computed for each station
NFFT=256 ; dt=1.0  #Time parameters
dk=0.2 ; pmin=0 ; pmax=1 ; kmax=10   #fk integration parameters
########################################################################

############                 Synthetics parameters               ###############
time_epi=UTCDateTime('2015-04-25T06:11:26')
epicenter=array([84.708,28.147,15])
resample=1 #Resample synthetics to this rate (in Hz)
integrate=1 #=0 produces velocities, =1 makes displacements
beta=0 #Rake offset, usually a good idea to keep at zero
rupture_speed=3.0 #Fastest rupture allowed in km/s
num_windows=1
########################################################################

[...]
```

Bàsicament, es tracta d'una plantilla de script que s'ha d'emplenar amb les dades de la simulació que volem fer. Després, cal modificar les variables de la secció *What-do-you-want-to-do flags* per dir-li al programa exactament què volem fer, i a continuació es modifiquen les variables corresponents dins de *Green function parameters* i *Synthetics parameters*. Finalment, s'executa el script. El que es veu

aquí, a més, no és el script complet, només la part que conté les variables de configuració. Més avall hi ha codi que depèn d'aquestes variables.

Certament, no és la manera de treballar a la que la majoria de la gent està acostumada en una utilitat de línia de comandes. Això violaria alguns dels principis heurístics de la usabilitat[19]. Concretament:

- **Simplicitat (Larry Constantine)**: Fes fàcils les tasques comunes que l'usuari faci habitualment.

  En aquest cas, haver de modificar un script cada vegada no facilita les tasques de l'usuari. No tots els usuaris tenen per què saber el que és un script.

- **Diàleg simple i natural (Keith Instone)**: Porta les converses al nivell de l'usuari.

  Quan un usuari fa servir una utilitat de línia de comandes, està acostumat a passar-li els paràmetres al programa directament des de la mateixa línia de comandes, o en un fitxer de configuració. No és habitual haver de tocar un script.

- **Familiaritat (Deborah Mayhew)**: Un sistema que resulti familiar a l'usuari per presentar similituds amb un sistema anterior, serà més fàcil d'utilitzar.

  En aquest cas, les similituds serien les que he mencionat en el punt anterior: passar paràmetres per línia de comandes o fitxer de configuració.

- **Tecnologia invisible (Deborah Mayhew)**: La tecnologia utilitzada en el sistema ha de ser invisible per a l'usuari.

  En el MudPy l'usuari pot veure part de la tecnologia directament dins del script que està modificant.

Segurament n'incompleixi algun més, però al final la majoria són variacions sobre el mateix: l'usuari no hauria de modificar scripts per dir-li al programa el que vol fer.

Hi ha altres maneres més fàcils i intuïtives per a fer el mateix, com ara paràmetres per línia de comandes o fitxers de configuració. Aquests mètodes tenen l'avantatge de ser estàndard i coneguts per a la majoria d'usuaris. A més, eviten els problemes derivats de que l'usuari s'equivoqui i modifiqui una part del script que no hagi de modificar. En el millor dels casos, el programa donarà un error estrany durant l'execució que l'usuari molt probablement no sabrà resoldre; en el pitjor, el programa

seguirà funcionant però ho farà duna manera diferent a l'esperada i produirà un resultat erroni. Cap d'aquestes dues situacions és desitjable.

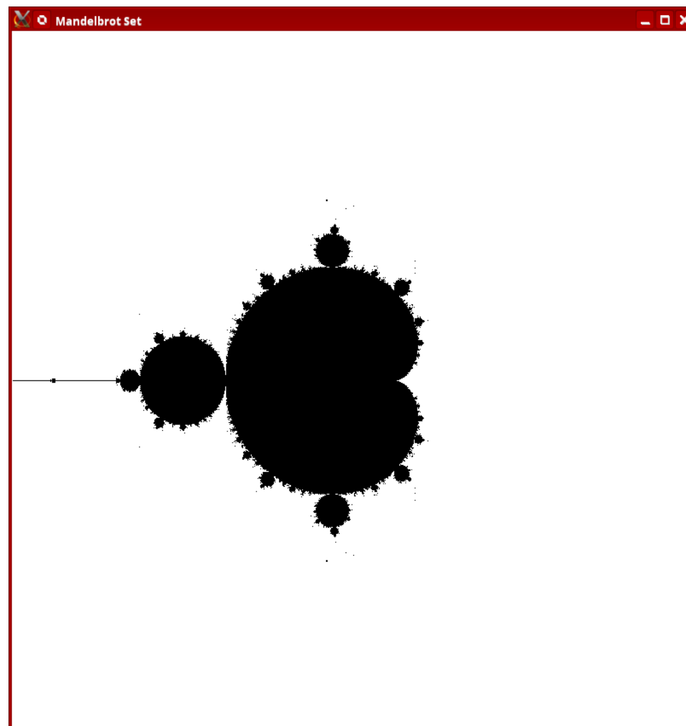
Capítol 3

Abast i objectius del projecte

En aquest capítol explico les meves motivacions per a fer el projecte, n'estableixo clarament el seu abast, i defineixo de manera concreta tots els seus objectius, tant primaris com secundaris.

3.1 Motivacions

Les meves motivacions per a fer aquest projecte són diverses. Per una banda, m'encanta programar, així que tenia clar que volia fer un projecte que impliqués haver d'escriure codi. Per l'altra, mentre cursava el màster vaig descobrir el paradigma de programació de GPUs a l'assignatura de Computació d'Altes Prestacions. En una de les PACs fins i tot vam haver d'implementar una versió paral·lela d'un algorisme per generar el famós conjunt fractal de Mandelbrot[20] (veure la imatge inferior, extreta de la meva pròpia PAC) per a ser executada en una GPU. Però amb prou feines vaig trigar un cap de setmana en acabar-la, i em vaig quedar amb ganes de més. Volia aprofundir una mica en la programació de GPUs, doncs em va semblar un tema molt interessant, així que li vaig demanar al professor de l'assignatura si tenia algun projecte relacionat amb aquest tema.



El conjunt de Mandelbrot, generat amb CUDA.

També em va semblar una bona oportunitat d'ampliar els meus coneixements sobre Python, que és un llenguatge molt popular i del qual no en sé gran cosa.

Finalment, em considero un apassionat de la ciència, i el fet de poder ajudar a un equip de científics en la seva recerca em proporciona una gran satisfacció a nivell personal. Saber que el resultat d'aquest projecte serà útil per a algú, ja que els permetrà executar moltíssimes més simulacions en un temps raonable, és molt important per a mi.

3.2 Objectius

Aquest projecte conté un component important de recerca, doncs es vol esbrinar si és possible o no paral·lelitzar el MudPy de manera que es pugui executar amb l'ajuda d'una GPU. Per tant, els objectius principals són:

- Investigar el potencial de paral·lelització del MudPy, especialment a l'hora d'executar-lo amb l'ajuda d'una GPU.
- Millorar el rendiment del MudPy a l'hora de calcular les ones del terratrèmol.

Per tal d'aconseguir això, adaptaré la funció `loop_sources` del fitxer `forward.py` per tal de que es pugui executar de forma paral·lela en una GPU fent servir Numba, una plataforma que permet paral·lelitzar codi Python fent servir CUDA. Idealment, simulacions que actualment triguen hores en executar-se s'haurien de poder generar en qüestió de minuts.

En quant als objectius secundaris, podem destacar els següents:

- Millorar la usabilitat de l'aplicació de manera que no calgui modificar cap script per executar-la. La idea seria passar-li els paràmetres mitjançant la línia de comandes o amb un fitxer de configuració.
- Identificar altres parts del programa que siguin paral·lelitzables i, en cas de ser possible, fer-ho.

Per últim, aquests serien els meus objectius a nivell personal:

- Aprofundir en el meu coneixement sobre CUDA i la programació de GPUs en general, que considero una camp molt interessant.
- Incrementar els meus coneixements sobre Python, que és un llenguatge molt popular i que segurament em resultarà molt útil en el meu futur desenvolupament com a professional.

3.3 Abast

S'agafarà el codi del MudPy i es modificarà per tal de permetre la seva execució en una GPU mitjançant la plataforma Numba. Es reaprofitarà el codi existent, que podrà ser reorganitzat o modificat, però no es reescriurà cap part del programa. Tampoc s'implementaran funcionalitats noves ni s'hi afegirà cap característica.

No es paral·lelitzaran les parts del programa encarregades d'inicialitzar l'estructura de directoris o de calcular les funcions de Green, ja que aquestes parts triguen relativament poc temps i només cal executar-les una vegada.

Els canvis en el codi es faran en Python per evitar els problemes derivats de migrar un codi a un nou llenguatge de programació, i també per facilitar-ne el manteniment per part dels autors originals.

Capítol 4

Planificació del projecte

En aquest capítol es detallen tots els aspectes relacionats amb la gestió del projecte: l'anàlisi dels possibles riscos, la definició dels lliurables, i finalment la planificació de cadascuna de les tasques que s'han de dur a terme per completar el projecte.

4.1 Anàlisi de riscos

A continuació s'enumeren els principals riscos que he identificat en el projecte, així com les possibles maneres de mitigar-los.

4.1.1 El rendiment no s'incrementa

Aquest és, sens dubte, el principal risc. La probabilitat de que passi és mitja-baixa, però l'impacte seria molt gran, ja que si s'arribés a materialitzar suposaria haver de replantejar el projecte.

Si es donés el cas, es podria fer és repetir l'anàlisi inicial per identificar noves parts del codi que es puguin paral·lelitzar. I si això no fos possible, aleshores es podrien buscar alternatives per accelerar l'execució del MudPy. Però això provocaria retards de ben segur.

El millor és evitar que passi. Primer, assegurant-nos de que l'anàlisi inicial és correcte i de que hem identificat realment els principals colls d'ampolla del codi. I segon, assegurant-nos de que les proves de rendiment són adequades. És perfectament possible que el codi resultant sigui menys eficient si fem servir simulacions petites, però que sigui molt més ràpid en simulacions molt més grans, que al final

és el que realment ens interessa. Per tant, les proves haurien de tenir en compte aquest aspecte.

4.1.2 El codi implementat no fa el mateix que l'anterior

Aquest és un risc important que sempre cal tenir en compte. La idea és modificar el codi per fer-lo més ràpid, però és imprescindible que segueixi funcionant exactament igual que abans. Si no, les simulacions no seran correctes i tota la feina no haurà servit de res. La probabilitat de que passi és baixa, però l'impacte seria molt alt.

Per tal d'evitar-lo, convé definir una bona bateria de proves que sigui capaç de detectar canvis en el comportament del codi. Una bona manera seria comparar els resultats d'una o diverses simulacions executades amb el codi actual i amb el nou, per veure si coincideixen.

4.1.3 Als usuaris no els agraden els canvis proposats per millorar la usabilitat

Aquest és un risc força probable però amb un impacte molt baix. És perfectament possible que els usuaris ja estiguin acostumats a la manera de funcionar actual del programa i que els canvis proposats no els agradin. Afortunadament, mitigar aquest problema és tan fàcil com assegurar-me de que tots els canvis relacionats amb la usabilitat del programa es facin en un o varis *commits* separats de la resta, de manera que es puguin desfer fàcilment en cas necessari. Una altra forma de minimitzar el risc seria fer els canvis de manera que ambdues possibilitats funcionin, és a dir, que els usuaris del programa puguin seguir fent-lo servir tal i com estan acostumats si així ho desitgen.

4.2 Lliurables

Durant les diferents fases del projecte es generaran els següents lliurables:

- **Memòria del projecte**
  El document complet amb tota la informació sobre la realització del projecte, incloses totes les PACs. Es lliurarà al final de tot.
- **Codi**
  El codi implementat es lliurarà en totes les PACS excepte la primera. Totes

les entregues seran parcials, excepte la darrera.

- **PACs**
  En cada PAC s'entregarà una versió parcial de la memòria del projecte. Aquesta versió contindrà la informació completa requerida en l'entrega actual, així com la informació ja existent de les entregues anteriors. També pot contenir informació parcial de les posteriors:
  - PAC1: anàlisi del codi del MudPy, abast del projecte, definició d'objectius i planificació.
  - PAC2: codi implementat fins al moment, pla de proves.
  - PAC3: codi implementat fins al moment, resultats parcials de les primeres proves.
  - PAC4: codi final, memòria del projecte, presentació.

4.3 Planificació

Aquesta és la planificació de les diferents tasques del projecte. En negreta es mostren les diferents fites, que coincideixen amb les 4 PACs definides, sent la PAC4 l'entrega final:

- Anàlisi del codi (17/2/2021 - 8/3/2021)
- Anàlisi de riscos (1/3/2021 - 8/3/2021)
- Definir l'abast (1/3/2021 - 8/3/2021)
- Planificar (9/3/2021 - 15/3/2021)
- **PAC1**
- Definir les proves (16/3/2021 - 31/3/2021)
- Començar la implementació (31/3/2021 - 19/4/2021)
- **PAC2**
- Acabar la implementació (19/4/2021 - 1/5/2021)
- Realitzar les proves de validació (1/5/2021 - 17/5/2021)
- **PAC3**
- Mesurar el rendiment i comparar (10/5/2021 - 17/5/2021)

- Acabar la memòria (18/5/2021 - 3/6/2021)
- Preparar la presentació (3/6/2021 - 18/6/2021)
- Realitzar la presentació (18/6/2021 - 30/6/2021)
- **PAC4 (entrega final)**

Capítol 5

Pla de proves

A continuació es detallen les proves que es faran sobre el codi un cop feta la implementació per tal de determinar si és correcte (és a dir, que fa el mateix que feia abans) i si realment és més ràpid.

L'entorn on es realitzaran les proves serà una màquina virtual al núvol (concretament a AWS, Amazon Web Services[21]), equipada amb una GPU de Nvidia[22]. A l'estar al núvol, hi accediré mitjançant SSH, i el meu PC actuarà de servidor X per quan sigui necessari visualitzar alguna cosa gràficament (per exemple, per veure el resultat d'una simulació).

5.1 Proves de validació del codi

Aquestes proves permetran garantir que el codi implementat segueix funcionant de la mateixa manera i genera els mateixos resultats per a una simulació concreta.

Hi ha dues proves bàsiques que es poden fer:

1. **Comparar els fitxers generats**
   He comprovat de manera empírica que els resultats de les simulacions són repetibles. Això vol dir que els fitxers de sortida (ubicats als subdirectoris output/waveforms/subduction.XXXXXX/) es poden comparar entre execucions i han de ser exactament iguals. Si no ho són, aleshores és que alguna cosa no funciona bé.

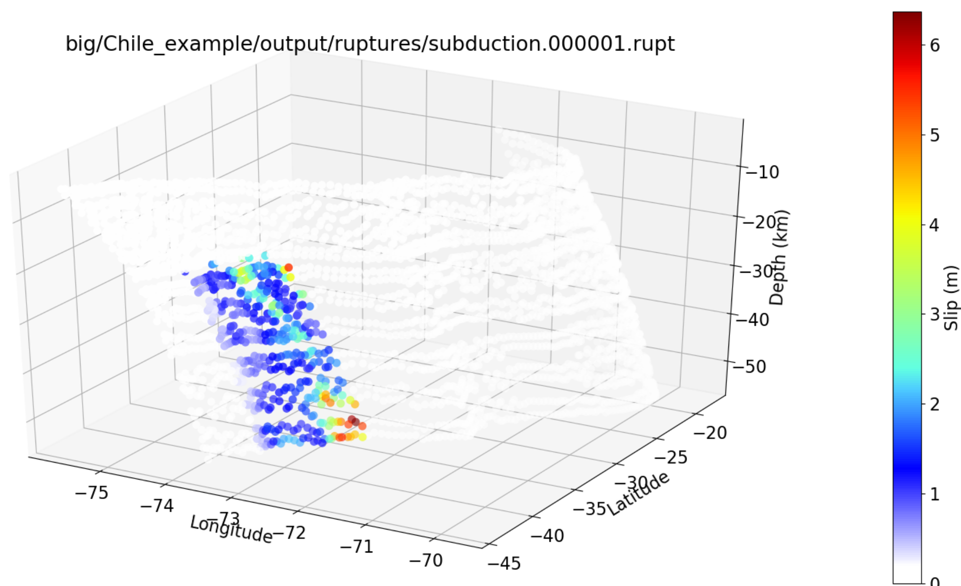
   Per tant, la primera prova de validació consistirà en executar una sèrie de simulacions amb paràmetres diferents fent servir el codi antic, i després repetir les mateixes proves fent servir el codi nou. Finalment, faré servir un simple

diff per comparar els resultats d'ambdues execucions i assegurar-me de que coincideixen.

2. **Inspecció visual del resultat**
Ja hem vist en el capítol d'introducció que el resultat de la simulació es pot mostrar de manera gràfica. Només cal fer servir un script com aquest:

```
from mudpy import view
rupture='big/Chile_example/output/ruptures/subduction.000001.rupt'
view.slip3D(rupture)
```

I es genera una imatge com la següent:



Així doncs, una altra prova que es pot fer és generar la representació gràfica dels resultats per a les dues proves realitzades (codi antic i codi nou) i veure si visualment coincideixen.

5.2 Proves de rendiment

Aquestes proves serviran per comprovar si s'ha complert l'objectiu d'accelerar els càlculs de les simulacions o no. Cal tenir en compte que, al ser un problema paral·lel, cal que la mida de les proves sigui gran, doncs és possible que amb proves petites no es vegi cap millora apreciable o, encara pitjor, que el codi s'executi més lentament. Per tal d'evitar aquest problema i que les proves siguin vàlides, miraré d'aconseguir diferents simulacions d'exemple amb diferents mides.

Aquestes simulacions s'executaran fent servir la comanda time[23] per poder veure quant triguen. Cadascuna d'elles s'executarà dues vegades: una amb el codi antic (sense GPU) i una altra amb el nou (amb GPU). Com que algunes poden trigar hores, les simulacions s'executaran en *batch mode*, és a dir, faré un script que les executarà automàticament de manera desatesa i guardarà els resultats en un fitxer.

Les proves es realitzaran sobre una màquina virtual amb GPU ubicada al núvol, concretament a Amazon Web Services. És important que totes les execucions es facin en la mateixa màquina. Si no, els resultats no serien comparables. Un cop fetes les simulacions, s'analitzaran els resultats per veure quin ha estat l'increment de rendiment aconseguit en cada cas.

Capítol 6

Implementació

En aquest capítol s'expliquen els detalls relatius a la implementació del projecte.

6.1 L'entorn de desenvolupament

Abans de res, cal preparar l'entorn de desenvolupament. Ha de ser un entorn que em permeti fer proves bàsiques d'execució, és a dir, proves que duren pocs segons, però que sigui el més semblants possible a l'entorn de proves finals (les de rendiment).

S'ha fet servir una màquina virtual al núvol. Concretament, una EC2 a Amazon Web Services de tipus G3[24], que inclou una GPU. Però abans de poder començar a provar, cal fer moltes coses: instal·lar els paquets del sistema necessaris (gcc, git, l'entorn de X Window...), definir un entorn de Python amb tots els paquets necessaris (NumPy, Numba...), instal·lar els controladors de CUDA[25], descarregar i compilar el MudPy....

Per tal de simplificar la tasca, he fet un script que executa tots els passos automàticament de manera desatesa:

```
#!/bin/bash

# Paquets base
sudo yum --assumeyes install git mc bash-completion gcc gcc-c++ gcc-gfortran xorg-x11-
    xauth xeyes gdb
sudo yum install -y gcc kernel-devel-$(uname -r)
sudo yum clean

# Drivers nVidia
BASE_URL=https://us.download.nvidia.com/tesla
DRIVER_VERSION=460.32.03
```

```
curl -fSsl -O $BASE_URL/$DRIVER_VERSION/NVIDIA-Linux-x86_64-$DRIVER_VERSION.run
sudo sh NVIDIA-Linux-x86_64-$DRIVER_VERSION.run --silent
rm -f NVIDIA-Linux-x86_64-$DRIVER_VERSION.run

# Anaconda
cd
wget https://repo.anaconda.com/archive/Anaconda3-2020.11-Linux-x86_64.sh
bash Anaconda3-2020.11-Linux-x86_64.sh -b -p /home/marc/anaconda3
./anaconda3/bin/conda init
rm -f Anaconda3-2020.11-Linux-x86_64.sh

# .bashrc
echo "alias l='ls -Llisha --color --group-directories-first'" >> ~/.bashrc
echo "export PYTHONPATH=\$PYTHONPATH:/home/marc/MudPy/src/python" >> ~/.bashrc
echo "export MUD=/home/marc/MudPy" >> ~/.bashrc
echo "export PATH=\$PATH:/home/marc/MudPy/src/fk" >> ~/.bashrc
source ~/.bashrc

# Entorn de Python
conda config --set auto_activate_base False
conda create --yes -n mudpy python=2.7
conda update --yes conda
conda install --yes -n mudpy -c conda-forge obspy
conda install --yes -n mudpy -c conda-forge pyproj
conda install --yes -n mudpy -c conda-forge utm
conda install --yes -n mudpy -c anaconda mpi4py
conda install --yes -n mudpy -c anaconda pandas
conda install --yes -n mudpy -c anaconda cudatoolkit
conda install --yes -n mudpy -c anaconda numba
conda clean --yes --tarballs

# Repositori del MudPy
#git clone https://github.com/dmelgarm/MudPy.git
git clone https://github.com/marcoll/MudPy.git
cd MudPy/src/fk
make clean
make -j4 all
```

Aquest script també em permetrà configurar ràpidament l'entorn de proves final.

6.2 Modificant el codi

Abans de començar a modificar el codi, cal tenir en compte una sèrie de consideracions respecte a CUDA/Numba:

- La GPU s'anomena *device* i la CPU normal *host*. Les funcions que s'executen a la GPU s'anomenen *kernels*.
- En general, el mètode `print` no funciona dins dels kernels, doncs s'executen al device i no tenen accés a la consola del host. Això pot dificultar el desenvolupament i els diagnòstics d'error posteriors. Però per sort existeix la possibilitat de fer servir un simulador, on sí que funciona aquest mètode.

- Els kernels s'executen a la memòria de la GPU. Per tant, abans de llançar-los cal copiar totes les dades necessàries des del host cap al device, i un cop acabada l'execució cal recuperar els resultats copiant-los des del device cap al host.
- Al copiar les dades des del host cap al device no es poden fer servir tipus de dades complexes. Tot ha de ser tipus bàsics o vectors numèrics.
- MudPy fa un ús molt intensiu de NumPy, que és una biblioteca de Python molt popular que permet fer tota mena de càlculs numèrics[26].
- Numba no permet fer servir totes les característiques de Python. Els kwargs[1], per exemple, no funcionen.
- Numba tampoc permet fer servir totes les característiques de NumPy. De fet, el que fa Numba és convertir totes les crides a mètodes de NumPy en crides a mètodes propis[27] compilats a codi màquina i optimitzats. Per tant, si Numba no ho implementa no es pot fer servir. A més, alguns dels mètodes que teòricament estan implementats i funcionen poden tenir algunes limitacions quan s'executen en una GPU[28]. És el cas de mètodes com `numpy.zeros`, `numpy.ones` o, en general, qualsevol mètode que treballi amb vectors dinàmics, que no estan disponibles dins d'un kernel. El MudPy fa servir molts d'aquests mètodes en la part del codi que s'ha de modificar. Per a més detalls, veure la secció 6.5.3.

Per tant, a l'hora de fer les modificacions segons indica la documentació de Numba[29] caldrà seguir els següents passos:

1. Identificar tots els tipus de dades complexes i convertir-los en vectors o tipus numèrics més simples, per tal de poder copiar-los al device.
2. Copiar totes les dades necessàries des del host cap al device. És important fer-ho de manera explícita per maximitzar el rendiment, ja que Numba pot copiar automàticament les dades per nosaltres, però les copia sempre en els dos sentits (host $\rightarrow$ device i device $\rightarrow$ host) i moltes d'aquestes còpies són redundants.
3. Un cop executat el kernel, recuperar els resultats copiant les dades des del device cap al host.
4. Comprovar que els resultats de l'execució a la GPU coincideixen amb els resultats sense GPU.

[1]Els kwargs permeten que una funció tingui un nombre variable de paràmetres que es passen per nom.

5. Optimitzar el codi: definir la mida adequada dels thread blocks, fer-lo el més escalable possible, minimitzar les transferències de memòria...

Per últim, també cal tenir en compte que l'ús de la GPU haurà de ser opcional, doncs no sempre el programa s'executarà en un entorn on aquesta opció estigui disponible. Per tant, el codi actual sense GPU ha de seguir funcionant. Això també resultarà molt útil en el moment de realitzar les proves de validació, doncs permetrà comparar les dues implementacions fàcilment.

Tots els canvis realitzats es poden veure en el fork del repositori original que tinc en el meu compte de GitHub: https://github.com/marcoll/MudPy

6.3 Modificacions per millorar el rendiment

Inicialment, la meva idea era implementar gairebé tota la funció `get_fakequakes_G_and_m_dynGF` del fitxer `forward.py` en un kernel. Aquesta funció calcula una matriu relativament gran, i jo volia que cada element de la matriu fos calculat per un fil de la GPU. Però, malauradament, les limitacions de Numba ho han fet impossible. Algunes funcions no suportades, com ara els mètodes per calcular diferències de temps o filtrar vectors, es podien reimplementar de manera relativament senzilla. Altres, en canvi, eren molt més complicades, com ara la funció `scipy.integrate.trapz` o `numpy.convolve`. A més, molts d'aquests mètodes requerien l'ús de vectors dinàmics temporals, que en un kernel no es poden fer servir, de manera que s'havien de crear tots des del host i després transferir-los al device. Tot això complicava molt el codi i afegia transferències de dades entre host i device que al final acaben penalitzant el rendiment.

Així que vaig decidir abandonar la idea inicial i centrar-me en millorar parts concretes de la funció `get_fakequakes_G_and_m_dynGF`. Al final vaig identificar quatre grans àrees de millora, que detallo a continuació, juntament amb els canvis realitzats per tal d'incrementar-ne el rendiment i l'escalabilitat allà on ha estat possible.

6.3.1 Còpia inicial de les dades

Al principi de cada iteració del bucle que calcula els diferents elements de la matriu podem trobar aquest codi:

```
#Get synthetics
nss=Nss[read_start+i_non_zero[ksource]].copy()
ess=Ess[read_start+i_non_zero[ksource]].copy()
zss=Zss[read_start+i_non_zero[ksource]].copy()
nds=Nds[read_start+i_non_zero[ksource]].copy()
```

```
eds=Eds[read_start+i_non_zero[ksource]].copy()
zds=Zds[read_start+i_non_zero[ksource]].copy()
```

Això fa una còpia de les dades amb les que treballarà aquesta iteració. El problema és que copiar aquestes dades és una operació relativament costosa, i en realitat no cal, doncs les dades originals no es modifiquen en cap moment, ja que els càlculs es realitzen sobre els vectors retornats per la funció `tshift_trace`, que els construeix a partir de les dades originals. Per tant, copiar les dades d'aquesta manera és innecessari.

Tanmateix, com que aquestes dades s'han de fer servir en el kernel que implementarà la versió paral·lela de `tshift_trace`, sí que cal copiar-les a la memòria del device. Però aquesta còpia només cal fer-la una sola vegada. He creat el mètode `copy_data_to_device` que es crida al principi:

```
def copy_data_to_device():
    import numba
    from numba import cuda

    #Convert all necessary data into arrays
    print('... converting data for the device')
    Nss_data_array=to_data_array(Nss)
    Ess_data_array=to_data_array(Ess)
    Zss_data_array=to_data_array(Zss)
    Nds_data_array=to_data_array(Nds)
    Eds_data_array=to_data_array(Eds)
    Zds_data_array=to_data_array(Zds)

    # Copy the arrays to the device. This data will not be modified, so we only need to
     do it once
    print('... copying data to the device')
    d_Nss_data_array=cuda.to_device(Nss_data_array)
    d_Ess_data_array=cuda.to_device(Ess_data_array)
    d_Zss_data_array=cuda.to_device(Zss_data_array)
    d_Nds_data_array=cuda.to_device(Nds_data_array)
    d_Eds_data_array=cuda.to_device(Eds_data_array)
    d_Zds_data_array=cuda.to_device(Zds_data_array)

    return d_Nss_data_array,d_Ess_data_array,d_Zss_data_array,d_Nds_data_array,
    d_Eds_data_array,d_Zds_data_array
```

El mètode `to_data_array` és l'encarregat de convertir les dades inicials (streams d'ObsPy) en vectors de dades simples, ja que Numba no suporta tipus de dades complexes:

```
def to_data_array(stream):
    '''
    Convert a stream's data into a 2 dimensional matrix (first dimension
    is the stream number, second dimension is the actual data), so that it
    can be passed to the CUDA kernel
    '''
```

```
    from obspy import Stream

    #Create the array in pinned memory to speed up transfer to the device
    data_array=cuda.pinned_array((stream.count(),stream[0].data.size),dtype=np.float64)
    for i in range(stream.count()):
        data_array[i]=stream[i].data

    return data_array
```

D'aquesta manera, la còpia de les dades que abans es feia en cada iteració del bucle ara només es fa una vegada. És cert que és més costosa, doncs primer cal convertir les dades i no és el mateix copiar-les a la memòria de la CPU que transferir-les entre host i device. Però al fer-se només una vegada, redueix el temps de cada iteració del bucle i, en conjunts de dades grans, farà que s'executi més ràpid.

Convé notar que les dades es creen en el que s'anomena *pinned memory*[30], que és una zona de memòria on les pàgines estan bloquejades per evitar que el sistema operatiu les transfereixi a disc en cas de que falti memòria. La transferència de memòria entre host i device només es pot fer amb aquest tipus de memòria, o sigui que el més ràpid és directament reservar espai en *pinned memory* per evitar haver de copiar les dades abans de transferir-les.

6.3.2 La funció `tshift_trace`

Aquesta funció genera una sèrie de vectors a partir de les dades inicials i d'una sèrie de paràmetres. En alguns casos, part d'aquests vectors s'inicialitza a 0 o al valor del darrer element. La funció original generava aquests valors fent servir els mètodes `numpy.zeros` i `numpy.ones`, que creen i inicialitzen vectors nous i, per tant, són costosos:

```
if nshift>0:
    #Place in output variable
    nss.data=r_[zeros(nshift),nss.data[0:npts-nshift]]
    ess.data=r_[zeros(nshift),ess.data[0:npts-nshift]]
    zss.data=r_[zeros(nshift),zss.data[0:npts-nshift]]
    nds.data=r_[zeros(nshift),nds.data[0:npts-nshift]]
    eds.data=r_[zeros(nshift),eds.data[0:npts-nshift]]
    zds.data=r_[zeros(nshift),zds.data[0:npts-nshift]]
else:
    nss.data=r_[nss.data[-nshift:],nss.data[-1]*ones(-nshift)]
    ess.data=r_[ess.data[-nshift:],ess.data[-1]*ones(-nshift)]
    zss.data=r_[zss.data[-nshift:],zss.data[-1]*ones(-nshift)]
    nds.data=r_[nds.data[-nshift:],nds.data[-1]*ones(-nshift)]
    eds.data=r_[eds.data[-nshift:],eds.data[-1]*ones(-nshift)]
    zds.data=r_[zds.data[-nshift:],zds.data[-1]*ones(-nshift)]
```

Però com que aquesta funció treballa amb vectors on tots els seus elements són independents els uns dels altres, és molt fàcil de paral·lelitzar mitjançant CUDA. He escrit un kernel que fa el mateix, però sense haver de crear vectors nous i de manera que cada element el genera un fil diferent de la GPU. Vaig fer una primera versió on els fils es dividien en sis grups diferents, de manera que tots sis vectors es generaven en paral·lel. Però calia afegir moltes condicions, cosa que complicava el codi i, a la pràctica, no oferia un rendiment superior a la versió definitiva, que és més simple:

```
@cuda.jit
def tshift_trace_kernel(nss,ess,zss,nds,eds,zds,
                        Nss_data_array,Ess_data_array,Zss_data_array,
                        Nds_data_array,Eds_data_array,Zds_data_array,
                        index,npts,nshift):
    '''
    Does pretty much the same as tshift_trace, but more efficiently and adapted
    to run on the GPU.
    Warning: nshift has to be calculated outside of the kernel. Otherwise, it crashes.
    Probably due to some weird compilation bug
    '''

    # Identify the thread in order to know which one of the array elements is ours
    pos = cuda.grid(1)

    if (pos < npts):
        if (nshift > 0):
            #This code does the equivalent to:
            #   arr.data=r_[zeros(nshift),arr.data[0:npts-nshift]]
            if (pos < nshift):
                nss[pos]=0.0
                ess[pos]=0.0
                zss[pos]=0.0
                nds[pos]=0.0
                eds[pos]=0.0
                zds[pos]=0.0
            else:
                nss[pos]=Nss_data_array[index][pos-nshift]
                ess[pos]=Ess_data_array[index][pos-nshift]
                zss[pos]=Zss_data_array[index][pos-nshift]
                nds[pos]=Nds_data_array[index][pos-nshift]
                eds[pos]=Eds_data_array[index][pos-nshift]
                zds[pos]=Zds_data_array[index][pos-nshift]
        else:
            #This code does the equivalent to:
            #   arr.data=r_[arr.data[-nshift:],arr.data[-1]*ones(-nshift)]
            if (pos < (npts + nshift)):
                nss[pos]=Nss_data_array[index][pos-nshift]
                ess[pos]=Ess_data_array[index][pos-nshift]
                zss[pos]=Zss_data_array[index][pos-nshift]
                nds[pos]=Nds_data_array[index][pos-nshift]
                eds[pos]=Eds_data_array[index][pos-nshift]
                zds[pos]=Zds_data_array[index][pos-nshift]
            else:
                nss[pos]=Nss_data_array[index][-1]
                ess[pos]=Ess_data_array[index][-1]
                zss[pos]=Zss_data_array[index][-1]
                nds[pos]=Nds_data_array[index][-1]
                eds[pos]=Eds_data_array[index][-1]
```

```
            zds[pos]=Zds_data_array[index][-1]
```

Recordem que les dades inicials (Nss_data_array, Ess_data_array, etc) ja estaven copiades a la memòria del device. La variable `pos` indica quin element dels vectors ha de modificar el fil en qüestió. Convé destacar el fet de que es comprova de manera explícita que aquest element estigui dins del rang del vector. Aquesta comprovació és important, doncs el número de fils que es creen al device no sempre coincideix amb el que es demana. CUDA sempre els crea de 32 en 32[31], de manera que si, per exemple, en demanem 48, en crearà 64.

La resta del codi simplement assigna els valors de cada element depenent del valor de `nshift` i de `pos`.

Per cert, l'advertència que apareix en el comentari inicial és deguda a un problema amb el que em vaig trobar, i que sospito que es tracta d'un error de compilació de Numba. A la secció 6.5.6 en dono més detalls.

6.3.3 La funció `build_source_time_function`

Aquesta funció conté una sèrie d'operacions de filtratge i generació de vectors. Per començar, es genera un vector t fent servir el mètode `numpy.arange`:

```
t=arange(0,total_time+dt,dt)
```

La funció `numpy.arange` és una de les que no funcionen dins d'un kernel degut a que necessita crear un vector nou de manera dinàmica. Però es pot crear directament al device i omplir-lo de manera totalment paral·lela, ja que calcular cadascun dels seus elements és una operació trivial.

Després, tenim un codi que, a partir dels valors del vector t i una sèrie de paràmetres calculats genera un vector anomenat $Mdot$ fent servir tres mètodes diferents, depenent del tipus de càlcul necessari:

- Triangle:

```
#Assign moment rate
i=where(t<=rise_time/2)[0]
Mdot[i]=m*t[i]+b1
i=where(t>rise_time/2)[0]
Mdot[i]=-m*t[i]+b2
i=where(t>rise_time)[0]
Mdot[i]=0
```

- Cosine:

```
#Build in pieces
i1=where(t<tau1)[0]
i2=where((t>=tau1) & (t<2*tau1))[0]
i3=where((t>=2*tau1) & (t<rise_time))[0]
Mdot[i1]=Cn*(0.7-0.7*cos(t[i1]*pi/tau1)+0.6*sin(0.5*pi*t[i1]/tau1))
Mdot[i2]=Cn*(1.0-0.7*cos(t[i2]*pi/tau1)+0.3*cos(pi*(t[i2]-tau1)/tau2))
Mdot[i3]=Cn*(0.3+0.3*cos(pi*(t[i3]-tau1)/tau2))
```

- Dreger:

```
Mdot=(t**zeta)*exp(-t/tau)
```

Tal i com es pot veure, el vector *Mdot* es genera de tres maneres diferents. La funció where és una altra de les que no funcionen en un kernel, i es fa servir per filtrar quins elements de *Mdot* es calculen de quina manera. Totes aquestes operacions es poden paral·lelitzar fàcilment.

Com que els tres mètodes són molt diferents entre ells, he creat tres kernels diferents:

```
@cuda.jit
def triangle_stf_kernel(Mdot,t,rise_time,dt,time_offset,m,b1,b2):
    # Identify the thread in order to know which one of the array elements is ours
    i = cuda.grid(1)

    if (i < len(Mdot)):
        t[i] = dt * i

        if (t[i] <= rise_time / 2):
            Mdot[i] = m * t[i] + b1
        elif (t[i] > rise_time):
            Mdot[i] = 0.0
        elif (t[i] > rise_time / 2):
            Mdot[i] = -m * t[i] + b2
        else:
            Mdot[i] = 0.0

        #offset origin time
        t[i] = t[i] + time_offset


@cuda.jit
def cosine_stf_kernel(Mdot,t,rise_time,dt,time_offset,tau1,tau2,Cn):
    # Identify the thread in order to know which one of the array elements is ours
    i = cuda.grid(1)

    if (i < len(Mdot)):
        t[i] = dt * i

        if (t[i] < tau1):
            Mdot[i] = Cn * (0.7 - 0.7 * math.cos(t[i] * math.pi / tau1) +0.6 * math.sin
    (0.5 * math.pi * t[i] / tau1))
        elif ((t[i] >= tau1) & (t[i] < 2 * tau1)):
            Mdot[i] = Cn * (1.0 - 0.7 * math.cos(t[i] * math.pi / tau1) + 0.3 * math.
    cos(math.pi * (t[i] - tau1) / tau2))
```

```
        elif ((t[i] >= 2 * tau1) & (t[i] < rise_time)):
            Mdot[i] = Cn * (0.3 + 0.3 * math.cos(math.pi * (t[i] - tau1) / tau2))
        else:
            Mdot[i] = 0.0

        #offset origin time
        t[i] = t[i] + time_offset


@cuda.jit
def dreger_stf_kernel(Mdot,t,dt,time_offset,zeta,tau):
    # Identify the thread in order to know which one of the array elements is ours
    i = cuda.grid(1)

    if (i < len(Mdot)):
        t[i] = dt * i

        Mdot[i] = (t[i] ** zeta) * math.exp(-t[i] / tau)

        #offset origin time
        t[i] = t[i] + time_offset
```

Les primeres línies són comunes per als tres: identificar el fil per saber quins són els elements dels vectors que hem de generar, comprovar que no estem fora de rang (igual que a l'apartat anterior), i generar l'element corresponent del vector t. A continuació es generen els elements del vector fent servir un mètode diferent a cada kernel, a partir dels valors indicats, el valor de t i la posició del vector. Per últim, també de manera comuna s'incrementa el valor de t, que és una operació que es feia cap al final de la funció `build_source_time_function`.

6.3.4 Convolucions

Gairebé al final de la iteració es fan una sèrie de convolucions[32] entre vectors de dades:

```
nss.data=convolve(nss.data,stf)[0:NFFT]
ess.data=convolve(ess.data,stf)[0:NFFT]
zss.data=convolve(zss.data,stf)[0:NFFT]
nds.data=convolve(nds.data,stf)[0:NFFT]
eds.data=convolve(eds.data,stf)[0:NFFT]
zds.data=convolve(zds.data,stf)[0:NFFT]
```

Aquestes convolucions també són operacions relativament costoses, i per desgràcia no sembla que es puguin paral·lelitzar fàcilment.

Buscant per Internet possibles mètodes alternatius d'implementar aquesta operació[33][34], vaig trobar-ne un de molt prometedor que semblava que em podria

permetre optimitzar una mica la manera de realitzar els càlculs: el mètode de la transformada ràpida de Fourier[2].

Fent servir aquest mètode, l'algorisme per calcular la convolució entre dos vectors a i b es converteix en:

1. Calcular la transformada ràpida de Fourier del vector a.
2. Calcular la transformada ràpida de Fourier del vector b.
3. Multiplicar els vectors a i b.
4. Calcular la transformada ràpida de Fourier inversa sobre el resultat de la multiplicació.

Aquest mètode té dos avantatges:

- Si ens fixem en el codi original, el vector b sempre és el mateix: stf. Això vol dir que no cal repetir el càlcul sobre aquest vector sis vegades diferents, podem fer-ho només una vegada i després fer servir el mateix resultat en totes les convolucions.
- La multiplicació dels dos vectors és fàcilment paral·lelitzable fent servir la GPU.

Així que vaig modificar el codi per fer una primera prova:

```
L=(NFFT+len(stf)-1)
fft_stf=fft(stf,L)
nss=ifft(fft(nss,L)*fft_stf).real[0:NFFT]
ess=ifft(fft(ess,L)*fft_stf).real[0:NFFT]
zss=ifft(fft(zss,L)*fft_stf).real[0:NFFT]
nds=ifft(fft(nds,L)*fft_stf).real[0:NFFT]
eds=ifft(fft(eds,L)*fft_stf).real[0:NFFT]
zds=ifft(fft(zds,L)*fft_stf).real[0:NFFT]
```

Fent les convolucions d'aquesta manera es triga aproximadament un 25 % del temps que es triga fent servir el codi original. Una millora considerable, tot i no fer servir la GPU. Malauradament, quan es fa servir aquest mètode sembla que el resultat és diferent de l'esperat. Tal i com vaig definir al pla de proves, vaig comprovar la correcció del programa comparant els fitxers binaris generats, i ja no eren exactament iguals que els generats pel codi original. Els fitxers de text i la representació gràfica del resultat, curiosament, sí que coincidien, lo qual pot indicar que les diferències són petites i que podria tractar-se d'un mètode vàlid.

[2]La transformada ràpida de Fourier (sovint abreujada com a FFT) és un algorisme molt eficient per calcular la transformada discreta de Fourier, que serveix per descompondre un senyal en les diferents freqüències que el constitueixen[35].

Però he preferit deixar el codi tal i com estava al principi fins que ho pugui parlar amb l'equip del MudPy, per evitar problemes.

6.3.5 Multiprocés

El MudPy és capaç d'executar-se en diferents processos fent servir el mòdul `multiprocessing`[36] de Python. Òbviament, si aquesta característica està disponible el millor és fer-la servir, ja que incrementa bastant el rendiment. Però quan es fa servir la GPU, la cosa es complica força.

A continuació explico els passos que vaig seguir per tal d'implementar-ho de la manera que vaig pensar que era la millor, i després explicaré per què això no funciona i com ha quedat al final.

La idea inicial

És molt important que tots els processos comparteixin les mateixes dades a la GPU, per evitar duplicar esforços i, encara més important, esgotar la memòria. Les dades generades per la funció `tshift_trace` ocupen espai, i si cada procés n'ha de tenir la seva còpia ens quedarem sense memòria ràpidament. Per tant, interessa que les dades es copiïn una sola vegada i que tots els processos hi puguin accedir.

Però hi ha un problema: les dades generades en un procés no es poden fer servir directament en un altre, doncs cada procés té el seu propi espai de memòria. Per solucionar això, Numba proporciona un mecanisme que permet compartir dades entre diferents processos, anomenat CUDA IPC API[37]. Això permet obtenir uns identificadors especials (identificadors IPC) per a les zones de memòria reservades en el device i després compartir-los amb la resta de processos.

El mòdul `multiprocessing` de Python proporciona eines per crear una memòria compartida entre diferents processos. D'aquesta manera, els identificadors IPC es poden desar en aquesta memòria de manera que tots els processos hi tinguin accés.

Però com gairebé tot a Numba, hi ha algunes limitacions: el procés pare que crea els altres processos no pot inicialitzar CUDA abans de crear-los (abans de fer el *fork*), ni tampoc pot accedir a les zones de memòria compartides del device un cop creades. Això vol dir que cal seguir els següents passos, en aquest ordre:

1. El procés pare crea una zona de memòria compartida.
2. El procés pare crea tots els fills.

3. Els processos fills queden bloquejats a l'espera de tenir les dades disponibles.
4. El procés pare copia les dades al device.
5. El procés pare copia els identificadors IPC a la memòria compartida.
6. El procés pare notifica als fills que les dades ja es troben disponibles.
7. Els processos fills fan servir els identificadors IPC per obtenir les adreces de les dades a la memòria del device.
8. Els processos fills comencen a treballar.
9. El procés pare espera a que els fills acabin.

Per tant, serà necessari crear algun tipus de mecanisme de sincronització entre processos. El mòdul `multiprocessing` també permet crear semàfors, que és el mecanisme ideal per a aquest cas concret per un motiu important: hi ha una limitació en el nombre de processos que poden accedir al mateix temps a la memòria de la GPU. Cada procés que accedeix a la GPU consumeix una mica de memòria extra, fins que arriba un punt en el que comença a donar errors de CUDA_ERROR_OUT_OF_MEMORY. I els semàfors permeten no només sincronitzar processos, sinó també limitar el nombre d'accessos simultanis a un recurs, que és exactament el que volem en aquest cas.

Aquest és el codi del procés pare, que crea la memòria compartida i sincronitza els fills. Les parts del codi que no tenen res a veure amb la sincronització han estat omeses:

```
if use_gpu:
    #Create process shared memory to exchange the device memory handlers
    manager=mp.Manager();
    ns=manager.Namespace()

    '''
    Create a semaphore to synchronize the different processes. This is also used to
    control how many of them are running at the same time, since too many of them
    connected to the GPU can lead to memory problems (CUDA_ERROR_OUT_OF_MEMORY).
    The idea is to use ncpus as the maximum number of processes allowed to run at
    the same time.
    '''
    semaphore=mp.Semaphore(0)
else:
    ns=None

if use_parallel:

    [...]

    for ksource in range(hot_start,len(all_sources)):
        p = mp.Process(target=loop_sources, args=([ksource,use_parallel,use_gpu,ns]) )
        ps.append(p)
        p.start()
```

```
    #Now that all processes have been forked, we can copy the data to the device
    if use_gpu:
        d_Nss_data_array,d_Ess_data_array,d_Zss_data_array,d_Nds_data_array,
    d_Eds_data_array,d_Zds_data_array=copy_data_to_device()

        #Send the handles to the other processes
        ns.Nss_h=d_Nss_data_array.get_ipc_handle()
        ns.Ess_h=d_Ess_data_array.get_ipc_handle()
        ns.Zss_h=d_Zss_data_array.get_ipc_handle()
        ns.Nds_h=d_Nds_data_array.get_ipc_handle()
        ns.Eds_h=d_Eds_data_array.get_ipc_handle()
        ns.Zds_h=d_Zds_data_array.get_ipc_handle()

        #Control the amount of processes accessing the GPU at the same time (ncpus at
    most)
        for i in range(ncpus):
            semaphore.release()
```

Tal i com es pot veure al final, es fa servir el paràmetre ncpus per limitar la quantitat de processos que poden executar-se al mateix temps.

Aquest és el codi que executen els processos fills. Només s'agafen les dades compartides en cas d'estar fent servir la GPU i executant processos en paral·lel. Novament, les parts del codi que no tenen res a veure amb la sincronització han estat omeses:

```
def loop_sources(ksource,use_parallel,use_gpu,ns):
    if use_gpu:
        if use_parallel:
            #Wait until the data has been copied to the device
            semaphore.acquire(True)

            d_Nss_data_array=ns.Nss_h.open()
            d_Ess_data_array=ns.Ess_h.open()
            d_Zss_data_array=ns.Zss_h.open()
            d_Nds_data_array=ns.Nds_h.open()
            d_Eds_data_array=ns.Eds_h.open()
            d_Zds_data_array=ns.Zds_h.open()
        else:
            d_Nss_data_array,d_Ess_data_array,d_Zss_data_array,
            d_Nds_data_array,d_Eds_data_array,d_Zds_data_array=copy_data_to_device()

    [...]

    #Release the semaphore, so that the next process can start working
    if use_gpu and use_parallel:
        semaphore.release()
```

Les últimes línies són importants. Quan un procés acaba la seva execució cal que alliberi el semàfor per permetre que comenci a treballar el següent.

El problema

Un cop implementada la solució i fetes les primeres proves, tot semblava anar bé: les dades es copiaven a la GPU, tots els processos hi tenien accés, la sincronització funcionava, els resultats de les proves eren correctes, el rendiment era bo... Fins que vaig començar a fer proves amb simulacions grans, i vaig veure que les dades no hi cabien a la memòria de la GPU. És a dir, que la meva solució no era prou escalable en quant a memòria.

El resultat final

Al final, vaig haver d'eliminar la part on es copiaven de cop totes les dades a la memòria, inclosa la còpia que es feia inicialment a la secció 6.3.1. Simplement, cada procés copia les dades que necessita en cada moment, és a dir, al principi de cada iteració del bucle. Això permet que la memòria escali molt bé, ja que la quantitat de dades que es requereix en un moment determinat és petita, però augmenta molt la quantitat d'operacions de memòria, que redueixen moltíssim el rendiment.

Per a fer-nos una idea de la diferència que suposa aquest canvi, podem comparar una de les proves que vaig fer amb i sense aquest canvi. Es tracta de la prova Chile_example 1 (veure la subsecció 7.3.3), però amb un NFFT de 128 per evitar problemes de memòria. En la primera execució, copiant totes les dades al principi, amb GPU va trigar 02:12:31 i sense GPU 02:28:52. Descomptant el temps inicial emprat en carregar fitxers (unes dues hores en cada cas), realment són 17:06 i 30:58 de temps de càlcul real, respectivament. És a dir, 1026 segons contra 1858, un 45 % menys si fem servir la GPU. En el segon cas, sense copiar les dades al principi, amb GPU va trigar 02:27:14 i sense GPU 02:26:04. És a dir, que amb la GPU fins i tot va trigar una mica més.

Per tant, aquest guany inicial d'un 45 % ha desaparegut pel fet de no poder copiar totes les dades a memòria al principi de l'execució. També cal tenir en compte que aquesta prova no és gaire gran (NFFT de 128), i que amb valors de NFFT més alts segurament la GPU serà més ràpida, però ja es veu que hi ha hagut una penalització de rendiment molt important.

6.4 Modificacions per millorar la usabilitat

L'objectiu secundari d'aquest projecte consistia en millorar una mica la usabilitat del MudPy, especialment a l'hora d'invocar-lo. Recordem que, per tal de generar

una simulació des del principi, calia seguir tres passos, i cadascun d'ells implicava modificar un script que servia com a fitxer de configuració.

6.4.1 Millores a l'hora d'invocar el MudPy

Una de les característiques més molestes del MudPy, especialment a l'hora de fer proves, és la seva peculiar manera de ser invocat, mitjançant un script que al mateix temps serveix com a fitxer de configuració. Això vol dir que, en cas de voler executar el MudPy amb un valor diferent per a qualsevol d'aquests paràmetres, abans cal modificar el script.

Per tal de millorar aquest funcionament, he afegit la possibilitat de passar-li els paràmetres directament a través de la línia de comandes. He fet servir el mòdul argparse[38] de Python. He procurat que el resultat fos el menys intrusiu possible, de manera que els scripts existents no s'haguessin de modificar gaire. També m'he assegurat de que els seu ús fos opcional: si no es fa servir, funciona com fins ara; i si es fa servir, tots els paràmetres que no s'indiquin per la línia de comandes agafaran per defecte els valors ja especificats al script.

Simplement cal afegir aquestes dues línies a qualsevol dels scripts, just després d'haver declarat totes les variables globals, i ja es pot fer servir la línia de comandes:

```
from mudpy import parsecmdargs
parsecmdargs.parse_cmdline_arguments(globals())
```

El mètode `parsecmdargs.parse_cmdline_arguments` està implementat al fitxer parsecmdargs.py, que és nou i conté tota la lògica relacionada amb la nova interfície per línia de comandes. Així és com es veu:

```
(mudpy) [marc@ip-172-30-0-65 small]$ python Chile.py -h
usage: Chile.py [-h] [-load_distances [TF]] [-g_from_file [TF]] [-ncpus N]
                [-use_gpu [TF]] [-model_name NAME] [-fault_name NAME]
                [-slab_name NAME] [-mesh_name NAME] [-distances_name NAME]
                [-utm_zone ZONE] [-scaling_law [TSN]] [-dynamic_gflist [TF]]
                [-dist_threshold N] [-nrealizations N] [-max_slip N]
                [-hurst N] [-ldip {auto,MB2002,MH2019}]
                [-lstrike {auto,MB2002,MH2019}] [-lognormal [TF]]
                [-slip_standard_deviation N] [-num_modes N] [-rake N]
                [-force_magnitude [TF]] [-force_area [TF]] [-no_random [TF]]
                [-time_epi TIME] [-hypocenter N,N,N] [-force_hypocenter [TF]]
                [-mean_slip NAME] [-center_subfault N]
                [-use_hypo_fraction [TF]]
                [-source_time_function {triangle,cosine,dreger}]
                [-rise_time_depths N,N] [-shear_wave_fraction N]
                [-gf_list NAME] [-g_name NAME] [-nfft N] [-dt N] [-dk N]
                [-pmin N] [-pmax N] [-kmax N] [-custom_stf STF]
                [[{init,make_ruptures,make_g_files,make_waveforms}]
                [[{init,make_ruptures,make_g_files,make_waveforms}] ...]]
```

```
MudPy's command line arguments

positional arguments:
  [{init,make_ruptures,make_g_files,make_waveforms}]
                        What do you want to do?
                         * init: initialize project folders
                         * make_ruptures: generate rupture models
                         * make_g_files: prepare waveforms and synthetics
                         * make_waveforms: synthesize the waveforms

optional arguments:
  -h, --help            show this help message and exit
  -load_distances [TF]  load the distances instead of calculating them
  -g_from_file [TF]     read the LARGE matrix (must run make_waveforms first)
  -ncpus N              number of CPUs to use
  -use_gpu [TF]         use the GPU, if available
  -model_name NAME      velocity model
  -fault_name NAME      fault geometry
  -slab_name NAME       slab 1.0 Ascii file (only used for 3D fault)
  -mesh_name NAME       GMSH output file (only used for 3D fault)
  -distances_name NAME  name of distance matrix
  -utm_zone ZONE        as defined in the Universal Transverse Mercator coordinate system
  -scaling_law [TSN]    T for thrust, S for strike-slip, N for normal
  -dynamic_gflist [TF]  use dynamic GFlist
  -dist_threshold N     (degree) station to the closest subfault must be closer to this distance
  -nrealizations N      fake ruptures to generate per magnitude bin. Let Nrealizations % ncpus=0
  -max_slip N           maximum slip (m) allowed in the model
  -hurst N              0.4~0.7 is reasonable
  -ldip {auto,MB2002,MH2019}
                        Correlation length scaling
  -lstrike {auto,MB2002,MH2019}
                        Lstrike value (correlation function parameter)
  -lognormal [TF]
  -slip_standard_deviation N
  -num_modes N          modes in K-L expantion (max#= number of subfaults)
  -rake N
  -force_magnitude [TF]
                        make the magnitudes EXACTLY the value in target_Mw
  -force_area [TF]      forces using the entire fault area defined by the .fault file
  -no_random [TF]       if true uses median length/width if false draws from prob. distribution
  -time_epi TIME        defines the hypocentral time
  -hypocenter N,N,N     defines the specific hypocenter location if force_hypocenter=True
  -force_hypocenter [TF]
                        forces hypocenter to occur at specified location as opposed to random
  -mean_slip NAME       provide path to file name of .rupt to be used as mean slip pattern
  -center_subfault N    use this subfault as center for defining rupt area
  -use_hypo_fraction [TF]
                        if true use hypocenter PDF positions from Melgar & Hayes 2019
  -source_time_function {triangle,cosine,dreger}
                        calculation method used in source time function
  -rise_time_depths N,N
                        transition depths for rise time scaling
  -shear_wave_fraction N
                        fraction of shear wave speed to use as mean rupture velocity
  -gf_list NAME         name of the GF list file
  -g_name NAME          name of the GF folder
  -nfft N               NFFT (displacement and velocity waveform parameter)
  -dt N                 dt (displacement and velocity waveform parameter)
  -dk N                 dk (fk parameter)
  -pmin N               pmin (fk parameter)
  -pmax N               pmax (fk parameter)
```

```
  -kmax N                 kmax (fk parameter)
  -custom_stf STF         custom_stf (fk parameter)

Examples:
    Chile.py init
    Chile.py make_ruptures -load_distances=T
    Chile.py make_waveforms -ncpus=16 -use_gpu=F -nfft=128 -source_time_function=dreger
    Chile.py make_ruptures make_g_files make_waveforms -load_distances=F -ncpus=16 -use_gpu=T

(mudpy) [marc@ip-172-30-0-65 small]$ _
```

D'aquesta manera, si volem executar el MudPy per generar ruptures podem simplement executar `python Chile.py generate_ruptures`. Si volem fer una simulació completa (ruptures, fitxers G i waveforms), podem executar `python Chile.py generate_ruptures make_g_files make_waveforms`. Si volem generar les waveforms fent servir la GPU, aleshores podem fer-ho amb `python Chile.py make_waveforms -use_gpu=T`. I si volem fer servir més d'una CPU, podem invocar-lo amb `python Chile.py make_waveforms -use_gpu =T -ncpus=32`.

Aquests són només alguns exemples, però gairebé totes les variables definides en els scripts es poden modificar d'aquesta manera sense haver d'editar el fitxer cada vegada.

6.4.2 Altres millores

A més dels canvis en la manera d'invocar el MudPy, també s'han fet una sèrie de petites millores:

- **Corregir un error en els noms de fitxer**
  Al generar les ruptures (un dels passos proves abans de generar les waveforms) hi havia un error que feia que el nom d'un fitxer no es generés correctament. Concretament, l'extensió del nom del fitxer no s'afegia. Aquest error ha estat corregit modificant el mètode `run_generate_ruptures` del fitxer fakequakes.py .

- **Fer que els errors del fk.pl apareguin a la consola**
  Per generar els fitxers G (un dels passos previs per a generar la simulació) el MudPy invoca un script de Perl. Malauradament, si en cridar a aquest script es produeix algun error no ens n'assabentarem, doncs no apareixerà enlloc. Per tal de millorar aquest aspecte, vaig fer un petit canvi al mètode `run_green` del fitxer green.py per assegurar-me de que, en cas d'haver-hi un error, el missatge corresponent aparegui en pantalla.

6.5 Problemes trobats

Cap projecte està exempt de reptes i desafiaments, i aquest no n'és cap excepció. A continuació es detallen els principals obstacles amb els que m'he trobat durant la fase d'implementació.

6.5.1 Numba encara no està madur

El primer problema és que Numba no és un projecte totalment madur. La seva darrera versió estable és la 0.53, lluny de la 1.0, lo qual ja indica que li queda força camí per recórrer. Això vol dir que hi ha moltes característiques que encara no estan suportades, que té limitacions, i que també conté alguns errors.

Per acabar-ho d'empitjorar, la documentació no és gaire extensa, i conté pocs exemples. Per sort, tenen un fòrum on la gent pot preguntar els seus dubtes i acostumen a respondre ràpidament. Jo mateix els he fet alguna pregunta, doncs al principi no entenia per què Numba no em deixava fer servir certes funcions dins d'un kernel, i em van respondre al dia següent.

6.5.2 La versió de Numba

Al fet de que Numba no estigui del tot madur se li suma un segon problema: la seva darrera versió només és compatible amb Python 3. Però el codi del MudPy està fet amb Python 2, lo qual significa que no puc fer servir la darrera versió de Numba. Al final he hagut de fer servir l'última versió compatible amb Python 2, la 0.47.

6.5.3 Limitacions en els kernels

Tal i com ja he mencionat anteriorment, hi ha unes quantes limitacions a l'hora d'implementar kernels amb Numba:

- No es poden crear vectors dinàmics, tota la memòria ha de ser estàtica o s'ha de reservar des del host (lo qual és més lent).
- No es poden fer servir funcions amb kwargs.
- No es poden fer servir cadenes de caràcters (strings), doncs Numba només és compatible amb les de Python 3 i MudPy està fet en Python 2.

- No tots els mètodes de NumPy estan suportats.
- Altres llibreries de Python que fa servir el MudPy (com ara SciPy o ObsPy) tampoc estan suportades.

Degut a aquestes limitacions em vaig haver de replantejar la meva estratègia a l'hora d'optimitzar el codi mitjançant CUDA. Inicialment volia implementar un kernel molt gran que calculés en paral·lel tots els elements d'una matriu. Però veient que no era possible utilitzar els mètodes `numpy.convolve` i `scipy.integrate.trapz`, al final vaig abandonar la idea i vaig decidir implementar kernels mes petits.

6.5.4 Limitacions en el multiprocés

Numba també té limitacions a l'hora de compartir una GPU entre varis processos. Si es vol que un procés pare copiï dades compartides al device i que després tots els altres processos accedeixin a les mateixes dades, hi ha una sèrie de restriccions que cal seguir. Per començar, el procés pare no pot inicialitzar CUDA abans de crear els processos fills. Si no, CUDA donarà errors. I a més, el procés pare pot crear la memòria del device, inicialitzar-la i crear els identificadors IPC que faran servir els altres processos, però no pot fer servir aquests mateixos identificadors.

A la pràctica, tot això implica que que el procés pare només servirà per copiar les dades i sincronitzar els fills (no podrà fer feina útil), i que caldrà parar molta atenció a l'ordre en el que es duen a terme les diferents accions (crear fills, inicialitzar CUDA, copiar les dades…).

6.5.5 Limitacions de memòria

Una altra limitació (aquesta més previsible) és la mida de la memòria del device, que no és infinita. Quan es treballa amb la GPU sempre s'ha de mirar de minimitzar la quantitat d'operacions de memòria (és a dir, copiar dades), doncs són lentes i penalitzen molt el rendiment. La meva idea inicial era copiar gairebé totes les dades que la GPU necessitava al principi de tot, en una zona de la memòria del device compartida per tots els processos, de manera que després només caldria anar copiant els resultats obtinguts. Però malauradament això només era possible amb simulacions petites. Amb simulacions grans em quedava sense memòria ràpidament, així que vaig haver de canviar d'estratègia i simplement anar copiant les dades que es feien servir en cada moment.

Això va fer desaparèixer el problema de la memòria, però va fer caure tant el rendiment que amb prou feines hi havia diferència entre executar les simulacions

fent servir la GPU i executar-les només amb la CPU.

6.5.6 Error misteriós

Hi va haver un error que em va fer ballar el cap ben bé durant un parell de dies. Al principi, al implementar el kernel `tshift_trace_kernel` vaig deixar allà dins el càlcul de la variable `nshift`, que és el següent:

```
# Shifted onset time
tstart=Nss[index].stats.starttime+timedelta(seconds=tdelay)
#How many positions from start of tr to start of trace?
nshift=int(round((tstart-time_epi)/dt))
```

Les variables `tstart`, `time_epi` i `dt` són nombres en coma flotant, mentre que `nshift` és un enter. Bàsicament, es fa un càlcul senzill en coma flotant i el resultat s'arrodoneix i es converteix en un enter[3]. Sembla senzill, però per algun motiu provocava un error molt estrany quan Numba compilava el kernel. Aquest error feia que qualsevol assignació que es fes d'un valor de qualsevol dels vectors de sortida (encara que fos simplement assignar-li un zero) provoqués un error fatal que finalitzava l'execució del programa. Però només passava quan aquesta assignació es feia dins d'un dels ifs. Si es feia fora, no hi havia problema. I si `nshift` tenia un valor enter fix en comptes de calcular-lo, tampoc passava.

Al final vaig arribar a la conclusió de que Numba no interpretava bé la conversió de coma flotant a enter a l'hora de compilar el codi del kernel. La solució va ser realitzar el càlcul de la variable `nshift` fora del kernel i passar-la com a paràmetre.

[3]El mètode `round` de Python no funciona de la manera que un podria esperar, ja que en realitat no retorna un valor enter, sinó un altre nombre en coma flotant[39]. Per això després es fa la conversió a enter de manera explícita.

Capítol 7

Proves

A continuació es detallen les proves realitzades, els seus resultats, i la interpretació que en faig.

7.1 L'entorn de proves

Per realitzar les proves s'ha fet servir una altra màquina virtual al núvol. En aquest cas, una EC2 a Amazon Web Services de tipus g3.8xlarge[24], que inclou 32 CPUs, 240 GB de RAM i 2 GPUs amb una memòria de 7 GB cadascuna.

7.2 Script de proves

Per tal d'automatitzar l'execució de les proves he creat un script en Perl. Cada prova està ubicada en un subdirectori dins del directori `tests`, i aquest script simplement va recorrent totes les subcarpetes i executant les diferents simulacions. Primer genera les ruptures i els fitxers GF (passos previs comuns per a totes les simulacions), i després executa les simulacions dues vegades: primer sense fer servir la GPU, i després fent-la servir. Entre les dues simulacions fa una còpia del resultat, per després poder comparar-los:

```
#!/usr/bin/perl


use Fcntl ':mode';
use File::Basename;
```

```
$PYTHON_CMD=$ENV{'HOME'} . '/anaconda3/envs/mudpy/bin/python';
$DATE_CMD='date --rfc-3339=seconds';
$DIFF_CMD='diff --brief --recursive';
$NCPUS=32;


if (scalar(@ARGV) == 0)
{
    $TESTS_DIR = $ENV{'HOME'} . '/tests';
}
elsif (-e $ARGV[0])
{
    $TESTS_DIR = $ARGV[0];
}
else
{
    die "No s'ha trobat el directori de tests\n";
}
print("Directori de tests: $TESTS_DIR\n");

# Recòrrer tots els subdirectoris dins del directori de tests
my $file, $filename, $mode;
opendir(DIR, $TESTS_DIR) or
    die("No s'ha pogut obrir el directori '$TESTS_DIR'\n");
my @files = readdir(DIR);
closedir(DIR);
foreach $file (@files)
{
    if ($file !~ m/^\./)
    {
        $filename = "$TESTS_DIR/$file";
        $mode = (stat($filename))[2];
        if (S_ISDIR($mode))
        {
            &process_test_subdir($filename);
        }
    }
}


# Processa un directori de test
sub process_test_subdir
{
    my $tdir = @_[0];
    my $testname = basename($tdir);
    my $log_file = "$TESTS_DIR/$testname.log";
    my $file, $script;

    print("Processant $testname...\n");

    # Localitzar el script per llançar el test
    opendir(TDIR, $tdir) or
        die("No s'ha pogut obrir el directori '$tdir'\n");
    my @files = readdir(TDIR);
    closedir(TDIR);
    foreach $file (@files)
    {
        if ($file =~ m/\.py$/)
        {
            my $script = "$tdir/$file";
            print("Script detectat: $script\n");
            &do_test($script, $log_file);
```

```
            return;
        }
    }
}


# Executa un test. Amb i sense GPU, i després compara els resultats
sub do_test
{
    my ($script, $log) = @_;
    my $data_dir;

    # Localitzar el directori de les dades
    open(my $f, "< $script") or
        die "No s'ha pogut obrir el fitxer '$script'\n";
    while (<$f>)
    {
        if ($_ =~ m/^\s*home\s*=\s*'(.*)'\s*$/)
        {
            $home = $1;
        }
        elsif ($_ =~ m/^\s*project_name\s*=\s*'(.*)'\s*$/)
        {
            $name = $1;
        }
    }
    close(FILE);

    $data_dir = "$home/$name";
    if (length($data_dir) <= 1 || !-e $data_dir)
    {
        die "No s'ha pogut trobar el directori de dades '$data_dir'\n";
    }
    print("Directori de dades detectat: $data_dir\n");

    $pwd = $ENV{'PWD'};
    chdir($home);

    system("rm -f $log");

    # Primer, generar les ruptures
    system("echo '' >> $log");
    system("$DATE_CMD >> $log");
    system("echo 'Generant ruptures...' >> $log");
    system("$PYTHON_CMD $script make_ruptures -ncpus=$NCPUS >> $log 2>&1");
    system("echo 'Ruptures generades' >> $log");
    system("$DATE_CMD >> $log");

    # Després, generar les GFs
    system("echo '' >> $log");
    system("$DATE_CMD >> $log");
    system("echo 'Generant GFs...' >> $log");
    system("$PYTHON_CMD $script make_g_files -ncpus=$NCPUS >> $log 2>&1");
    system("echo 'GFs generades' >> $log");
    system("$DATE_CMD >> $log");

    # Executar el test sense la GPU
    system("echo '' >> $log");
    system("$DATE_CMD >> $log");
    system("echo 'Generant waveforms sense GPU...' >> $log");
    $t0 = time();
    system("$PYTHON_CMD $script make_waveforms -ncpus=$NCPUS >> $log 2>&1");
```

```
    $t_nogpu = (time() - $t0);
    system("echo 'Waveforms sense GPU generades' >> $log");
    system("$DATE_CMD >> $log");

    # Copiar el resultat a un altre directori, per poder comparar posteriorment
    system ("rm -rf $data_dir.nogpu");
    system ("cp -r $data_dir $data_dir.nogpu");

    # Executar el test amb la GPU
    system("echo '' >> $log");
    system("$DATE_CMD >> $log");
    system("echo 'Generant waveforms amb GPU...' >> $log");
    $t0 = time();
    system("$PYTHON_CMD $script make_waveforms -use_gpu=T -ncpus=$NCPUS >> $log 2>&1");
    $t_gpu = (time() - $t0);
    system("echo 'Waveforms amb GPU generades' >> $log");
    system("$DATE_CMD >> $log");

    open(my $LOG, ">> $log") or
        die "No s'ha pogut obrir el fitxer de traces\n";

    #Comparar els dos resultats
    print $LOG ("\n\nComparant resultats...\n");
    my $diff = `$DIFF_CMD $data_dir.nogpu/output $data_dir/output`;
    if (length($diff) == 0)
    {
        print $LOG ("OK, els resultats amb i sense GPU coincideixen\n");
    }
    else
    {
        print $LOG ("ERROR! Els resultats amb i sense GPU NO coincideixen:\n");
        print $LOG ($diff)
    }
    printf $LOG ("Temps d'execució sense GPU: %s\nTemps d'execució amb GPU: %s\n",
                 &format_time($t_nogpu), &format_time($t_gpu));

    print $LOG ("\nFet\n");
    close($LOG);
    chdir($pwd);
}


sub format_time
{
    my $t = @_[0];

    $h = ($t / 3600);
    $m = (($t % 3600) / 60);
    $s = (($t % 3600) % 60);

    return sprintf("%02d:%02d:%02d", $h, $m, $s);
}
```

7.3 Proves realitzades

A continuació es detallen les proves realitzades. Totes s'han fet amb el paràmetre ncpus a 32, que és el nombre de CPUs de la màquina de proves.

7.3.1 Chile_small 1

Aquesta primera prova és una versió reduïda d'una altra prova (Chile_example), i és la mateixa que s'ha fet servir durant el desenvolupament. Fa servir 4 estacions, 20 sources i un NFFT de 128. El resultat de l'execució ha estat el següent:

Sense GPU:	0:14
Amb GPU:	0:20
Resultats coincideixen:	Sí

Tal i com es pot veure, l'execució amb la GPU ha estat més lenta que la que s'ha fet sense GPU. Cosa que, de fet, és esperable, ja que la versió amb GPU triga un temps en copiar les dades a la memòria del device, i si el problema no és prou gran aquest temps gastat de més no surt a compte.

7.3.2 Chile_small 2

Aquesta és una versió menys reduïda de Chile_example. Inclou un conjunt de dades més gran que el que s'ha fet servir per al desenvolupament, de manera que triga una mica més en executar-se. Fa servir 4 estacions, 160 sources i un NFFT de 128. El resultat de ha estat el següent:

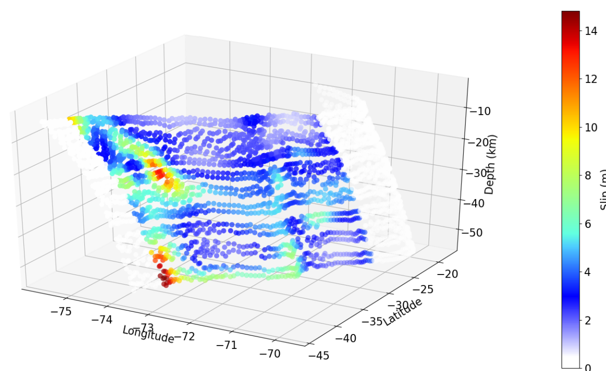
Sense GPU:	0:33
Amb GPU:	0:50
Resultats coincideixen:	Sí

L'execució amb la GPU segueix sent més lenta, lo qual vol dir que el problema encara no és prou gran. De fet, de la manera en que està implementat el codi de la GPU els principals increments de rendiment es veurien en simulacions amb un NFFT més gran, ja que el NFFT és el paràmetre que determina la mida dels vectors de dades que són processats per la GPU. En aquesta prova el NFFT era de 128, igual que el que vaig fer servir per a fer el desenvolupament. Per tant, cal fer una prova amb una simulació que tingui un valor de NFFT més gran.

7.3.3 Chile_example 1

Aquesta prova és la versió completa de Chile_small. Inclou un conjunt de dades més gran que el que s'ha fet servir en les dues proves anteriors: fa servir 121 estacions, 16 sources i un NFFT de 1024. El resultat de l'execució ha estat el següent:

Sense GPU:	02:38:11
Amb GPU:	02:31:50
Resultats coincideixen:	Sí



Cal tenir en compte que, de tot aquest temps, la simulació trigava unes 2 hores en carregar totes les dades de disc. Els temps passats fent càlculs són en realitat 39:46 i 35:37. Això vol dir que la millora real és d'un 10,44 %.

7.3.4 Chile_example 2

Aquesta prova és la mateixa que Chile_example 2, però amb un NFFT encara més gran, de 2048. El resultat de l'execució ha estat el següent:

Sense GPU:	03:02:37
Amb GPU:	02:58:06
Resultats coincideixen:	Sí

Els temps passats fent càlculs són 01:05:39 i 00:59:06. Això vol dir que la millora real és d'un 9,98 %.

7.4 Interpretació dels resultats

Tal i com es pot veure, la millora de rendiment no és gaire significativa. En les proves petites era esperable, doncs és conegut que en aquests casos és molt difícil treure-li un bon rendiment a una GPU. Però en les proves més grans la teoria diu que sí que s'hauria de poder veure una millora apreciable.

Hom podria pensar que potser és que les simulacions encara no són prou grans. Però tenint en compte que doblant el NFFT el rendiment ni tan sols s'ha incrementat, crec que està clar que no és aquest el problema.

El baix rendiment obtingut es pot explicar per dos factors:

- **Impossibilitat de calcular cada element de la matriu a la GPU**

  Ja hem vist abans (secció 6.3) que, degut a les limitacions de Numba, no era possible calcular cada element de la matriu en paral·lel a la GPU. Això va provocar un canvi d'estratègia orientat a optimitzar funcions concretes. Malauradament, la mida de les dades amb les que treballen aquestes funcions és relativament petita, de manera que es fa difícil escalar el problema fins a una magnitud suficient com per que l'ús de la GPU provoqui un increment notable del rendiment.

  En resum: la part del codi que és paral·lelitzable s'ha reduït considerablement a una fracció molt petita del total. I això, segons la llei d'Amdahl[7], és precisament el contrari del que volem, doncs per poder obtenir un bon rendiment paral·lel cal que la major part del programa sigui paral·lelitzable.

- **Impossibilitat de copiar totes les dades a la memòria**

  També hem vist a la subsecció 6.3.5 que no era possible copiar totes les dades a memòria a l'inici de l'execució, de manera que vaig haver de copiar només les dades necessàries en cada iteració del bucle. Copiar dades entre la memòria de la CPU i la GPU és una cosa que sempre cal minimitzar, doncs és un dels principals colls d'ampolla. Això va provocar una notable caiguda del rendiment, tal i com s'ha vist al final de la subsecció 6.3.5, passant d'un increment de rendiment del 45 % en problemes relativament petits a pràcticament 0.

L'origen del problema, doncs, és que hi ha massa operacions de memòria entre el device i el host. Com que la part paral·lelitzable del programa és petita, el temps que es perd copiant dades fa desaparèixer els ja de per sí escassos increments de rendiment obtinguts al fer servir la GPU.

En problemes més grans s'ha aconseguit una millora del 10 %. Però aquest percentatge és massa petit com per justificar l'ús d'una GPU i, a més, no sembla que escali gaire bé, doncs el rendiment no ha millorat tot i doblar la mida del NFFT.

Capítol 8

Possibles millores futures

Tal i com s'ha vist en el capítol anterior, l'increment de rendiment del MudPy mitjançant l'ús d'una GPU no ha estat tan gran com es podia pensar inicialment. Per tant, en aquest capítol exploraré possibles millores futures i alternatives viables per fer que l'execució del MudPy sigui més ràpida.

8.1 Optimitzar el codi

Alguns dels canvis que vaig fer per adaptar el codi del MudPy per executar-lo en una GPU es podrien traslladar al codi actual com a optimitzacions:

- Evitar copiar les dades a l'inici de cada iteració del bucle.
- Sempre que sigui possible, evitar utilitzar funcions de NumPy com ara `numpy.empty`, `numpy.zeros`, `numpy.ones` o `numpy.where`, que fan servir memòria dinàmica per crear vectors nous, un procés relativament costós.
- Revisar el codi de la funció `tshift_trace` i generar els vectors de sortida sense fer servir la funció `numpy.r_`. De la manera en que està fet el codi actualment, es generem molts vectors temporals que requereixen l'ús de memòria dinàmica, però el codi es pot reimplementar de manera que tot això no calgui.

L'increment de rendiment segurament serà modest, però es podrà aplicar a qualsevol execució del MudPy, sense necessitat d'haver de fer servir cap GPU.

8.2 Migrar el codi a C o C++

Una possible solució a la lentitud del MudPy podria ser migrar una part del seu codi a C o C++. Al ser llenguatges compilats i pensats per oferir un bon rendiment, la velocitat d'execució augmentaria de ben segur. A més, permetria adaptar el codi a CUDA més fàcilment.

Però no tot són avantatges. Cal tenir en compte el cost de migrar el codi, els possibles errors que es puguin produir en la migració, el fet de que els autors del MudPy segurament no estiguin familiaritzats amb aquests dos llenguatges, el fet de que són més complexes que Python… A més, caldria buscar biblioteques que implementessin els mètodes de NumPy o SciPy que es fan servir (`scipy.integrate.trapz`, `numpy.convolve`…), o bé implementar-los a mà. I alguns d'ells no són precisament trivials. Implementar un canvi tan profund com aquest no és viable en un projecte d'un semestre.

8.3 Fer servir alternatives a Numba

Numba no és l'única manera de fer servir CUDA amb Python, hi ha altres possibles alternatives que podria ser interessant explorar. Una d'elles és PyCuda[40][41], que permet barrejar codi Python i C++ en un mateix programa. La idea és que els kernels s'implementin en C++ i la resta del codi en Python.

Podria ser un compromís a mig camí entre fer servir Numba i migrar tot el codi a C++, tot i que segueix tenint el problema d'haver de reimplementar manualment alguns mètodes de NumPy o SciPy.

8.4 HTC i grid computing

HTC són les inicials de *High-throughput computing*[42]. El seu objectiu és maximitzar el nombre d'execucions d'una aplicació per unitat de temps. Bàsicament, dividir la feina en diferents treballs independents que després es puguin executar en diferents unitats de procés.

Per la seva banda, el *grid computing* (o computació en graella [43]) consisteix en compartir un conjunt de recursos heterogenis de manera global, entre diferents institucions i grups de recerca, de manera que es vegin com si fossin un únic computador. Aquests recursos s'interconnecten a través d'Internet, i es gestionen fent servir el que s'anomena un *middleware*, que es un programa intermediari

encarregat d'abstraure la complexitat del grid i permetre que s'enviïn treballs per a ser executats.

Encara que el codi del MudPy no sigui fàcilment paral·lelitzable, les seves execucions sí que ho són, doncs són totalment independents entre elles. La idea seria executar els primers passos d'una simulació amb el MudPy, guardar el resultat parcial, i executar en el grid el darrer pas, que és el que s'ha de repetir centenars de vegades. Encara que cada execució duri una setmana, si s'envia a 100 ordinadors diferents obtindrem el resultat de les 100 execucions en qüestió de dies.

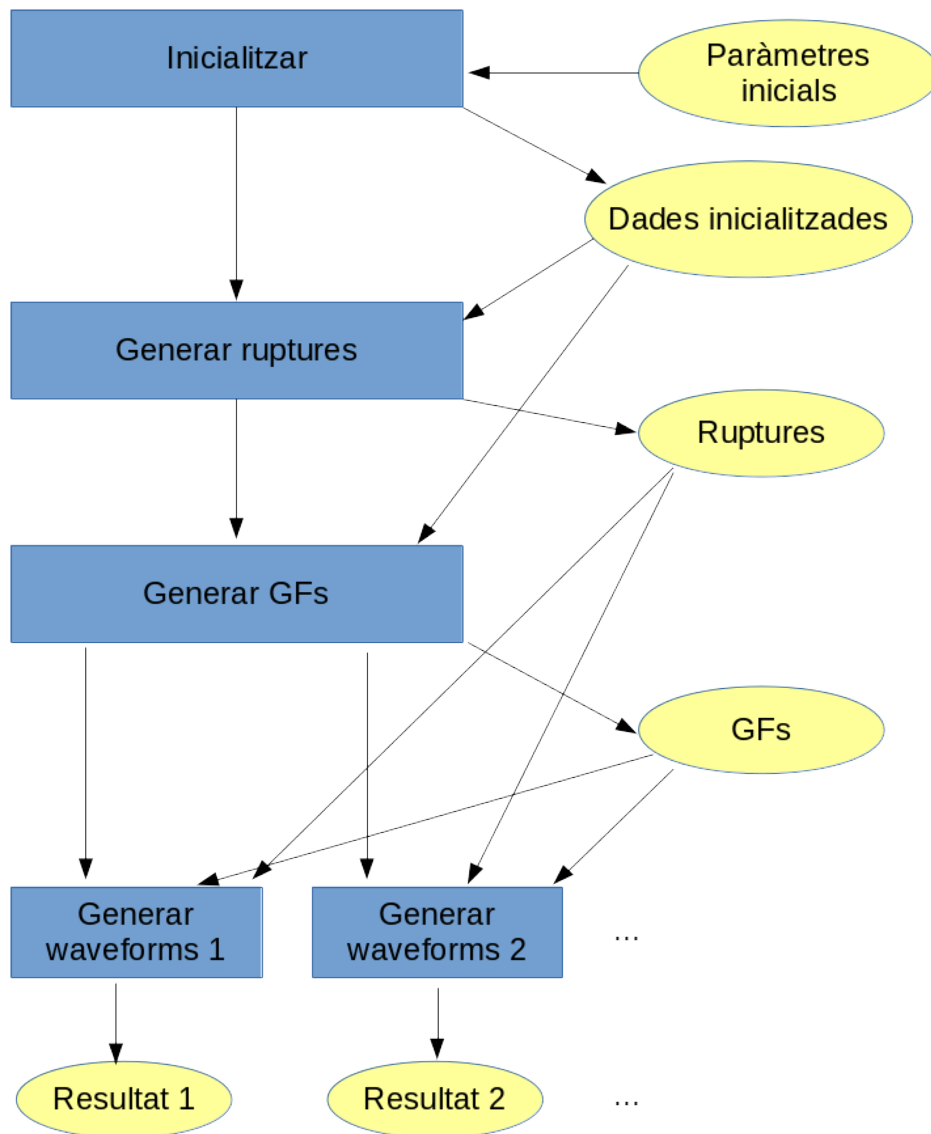
Aquesta idea no és nova. Ja s'han fet treballs en aquest sentit, com ara el TFM de Javier Alexander Mahecha Guerrero[44], que explora la possibilitat d'aprofitar la potència de la computació en graella per tal de processar vídeos en paral·lel fent servir el sistema de gestió de fluxos de treball Pegasus sobre el *middleware* HTCondor.

Per al MudPy es podria fer alguna cosa semblant. A continuació explicaré de manera breu com es podria arribar fer, però sense entrar en detalls, doncs aquest tema donaria per fer un altre TFM sencer.

La idea és fer servir Pegasus per crear fluxos de treball que permetin anar parametritzant les diferents execucions, i després fer servir HTCondor per executar el flux en paral·lel en diferents màquines repartides per tot el món.

Pegasus permet definir els fluxos de treball fent servir grafs dirigits acíclics, de manera que cada vèrtex del graf és una tasca i els arcs representen les dependències entre elles. Les dades que genera cadascuna de les tasques es guarden en un lloc anomenat *staging*, on les altres tasques hi poden accedir. Aquest *staging* pot estar allotjat en un servidor de fitxers al núvol, de manera que tothom hi pugui accedir remotament.

En el cas del MudPy, el flux de treball seria el següent, on les tasques són els rectangles blaus i les dades de sortida generades per cadascuna d'elles són les el·lipses grogues:



Les dependències entre tasques són lineals: abans de passar a la següent cal haver completat l'actual. Només el darrer pas, generar les waveforms, es pot executar en paral·lel. Però com que les primeres tasques només s'han d'executar una vegada, no és un problema.

Fent servir aquest mètode, s'hauria de poder executar una gran quantitat de simulacions en un temps força raonable. Crec que és una solució que val la pena explorar en més profunditat.

Capítol 9

Conclusió

9.1 Objectius assolits

Malauradament, un dels principals objectius del projecte (incrementar el rendiment del MudPy) no s'ha assolit plenament. Una millora del 10 % no és prou significativa com per suposar una diferència important, i certament no justifica l'ús d'una GPU, ja que les màquines amb GPU tenen un cost superior a les que no en tenen.

Una de les contribucions del projecte ha estat l'estudi de les limitacions de plataformes com Numba. La conclusió principal que podem extreure és que, actualment, no és factible paral·lelitzar l'execució del MudPy fent sevir una GPU, doncs les eines disponibles no estan prou madures.

Tanmateix, investigar el potencial de paral·lelització del MudPy era un altre dels objectius, i la informació obtinguda en el procés és molt valuosa; ens permet analitzar millor les possibilitats reals d'optimització del programa i suggerir alternatives més viables (veure el capítol 8).

9.2 Seguiment de la planificació

La planificació inicial del projecte ha resultat ser, gairebé fins a la PAC3, força acurada. A partir d'aquí, les tasques d'acabar la implementació, realitzar les proves i mesurar el rendiment s'han allargant una mica i han acabat encavallant-se entre elles. La part entre la PAC3 i la PAC4 no s'ha complert, doncs al haver-se materialitzat un dels principals riscos del projecte (el rendiment no s'ha incrementat), s'ha hagut

d'invertir temps en buscar alternatives, que és una cosa que no estava prevista inicialment.

Això, per cert, ha estat un error, doncs hauria estat bé que la planificació inicial hagués tingut en compte els escenaris alternatius derivats dels possibles riscos. De poc serveix tenir identificat un risc si no està quantificat, encara que sigui de manera aproximada, quin retard pot provocar en el projecte.

9.3 Coneixements aplicats

Un dels objectiu principals de tot treball de final de màster és que l'alumne apliqui els coneixements adquirits durant els seus estudis. Aquesta és la llista amb els principals coneixements que he fet servir en la realització d'aquest projecte:

- Gran part dels coneixements adquirits en l'assignatura de Computació d'altes prestacions, especialment les lleis d'Amdahl i Gustafson, i la programació de GPUs.
- Els coneixements adquirits en l'assignatura d'Enginyeria de la usabilitat, per tal d'analitzar i millorar la usabilitat del MudPy.
- Els coneixements adquirits en l'assignatura de Gestió avançada de projectes TIC, per tal de planificar el projecte, definir-ne l'abast i els riscos, i fer-ne el seguiment.
- Els coneixements adquirits durant les pràctiques de l'assignatura d'Intel·ligència artificial avançada, doncs és on vaig tenir el meu primer contacte amb Python.

9.4 Coneixements adquirits

Un altre dels objectius principals d'un treball de final de màster és aprendre coses noves o aprofundir en coneixements ja existents. En el meu cas, els principals coneixements que he adquirit o millorat són:

- Programació de GPUs, que és una cosa que vaig aprendre a l'assignatura de Computació d'altes prestacions de manera molt superficial, i sobre la qual he pogut aprendre molt més durant el projecte.

- Programació amb Python, que ja vaig aprendre a l'assignatura d'Intel·ligència artificial avançada i que ara he tingut l'oportunitat de millorar. I és que llegint el codi escrit per altres persones s'aprenen moltes coses.
- Numba, que és un entorn de programació que fins ara desconeixia. De fet, ni tan sols sabia que era possible programar GPUs fent servir Python.
- Edició de documents amb LaTeX, que és el que he fet servir per escriure aquesta memòria. La majoria de PACs del màster ja les havia escrit amb aquest sistema, però el fet d'haver d'escriure un document més llarg m'ha permès ampliar els meus coneixements existents i també aprendre a fer servir BibTeXper gestionar referències.

9.5 Valoració personal

Tot i estar una mica decebut per no haver aconseguit assolir el principal objectiu del projecte, la meva valoració personal és, en general, positiva. He aprés moltes coses noves, he ampliat els meus coneixements en moltes àrees i, tot i la frustració que he sentit en alguns moments, considero que ha estat un projecte força interessant i estimulant.

Bibliografia

[1] Diego Melgar et al. *Codi font del projecte MudPy a GitHub*. URL: https://github.com/dmelgarm/MudPy.

[2] *Wiki del projecte MudPy*. URL: https://github.com/dmelgarm/MudPy/wiki.

[3] MPI Forum. *Message Passing Interface*. URL: https://www.mpi-forum.org.

[4] *GPGPU*. URL: https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units.

[5] nVidia. *Especificaciones nVidia GeForce RTX 2080 Ti*. URL: https://www.nvidia.com/es-es/geforce/graphics-cards/rtx-2080-ti.

[6] AMD. *AMD Processor Specifications*. URL: https://www.amd.com/en/products/specifications/processors.

[7] Ivan Rodero Castro i Frances Guim Bernat. ?Computació d'altes prestacions? A: vol. Mòdul didàctic 1 - Introducció a la computació d'altes prestacions. Cap. 4.2 Llei d'Amdahl.

[8] *CUDA a la Wikipedia*. URL: https://en.wikipedia.org/wiki/CUDA.

[9] *Pàgina oficial de CUDA*. URL: https://developer.nvidia.com/cuda-zone.

[10] *How to pick the best GPU for Bitcoin mining*. URL: https://bitcoinminingsoftware2019.com/bitcoin-mining-with-gpu-top-choices-for-2019/#How_to_pick_the_best_GPU_for_Bitcoin_mining.

[11] Jarred Walton. *GPU Shortages Will Worsen Thanks to Coin Miners*. URL: https://www.tomshardware.com/uk/news/gpu-shortages-worsen-cryptocurrency-coin-miners-ethereum.

[12] Then Wire. *Gamers can once again blame Bitcoin miners for GPU shortage*. URL: https://thenwire.com/gamers-can-once-again-blame-bitcoin-miners-for-gpu-shortage.

[13] *Dedicated GPU for Professional Mining*. URL: https://www.nvidia.com/en-us/cmp.

[14] Khronos Group. *OpenCL*. URL: https://www.khronos.org/opencl.

[15] *Khronos Group*. URL: https://www.khronos.org.
[16] Khronos Grup. *OpenGL*. URL: https://www.khronos.org/opengl.
[17] *GPU Accelerated Computing with Python*. URL: https://developer.nvidia.com/how-to-cuda-python.
[18] *CUDA Python*. URL: https://developer.nvidia.com/cuda-python.
[19] Mònica Zapata Lluch. ?Enginyeria de la usabilitat? A: vol. Mòdul didàctic 2 - Mètodes d'avaluació sense usuaris. Cap. 1.3 Principis heurísitcs.
[20] *Mandelbrot set*. URL: https://en.wikipedia.org/wiki/Mandelbrot_set.
[21] *Amazon Web Services*. URL: https://aws.amazon.com.
[22] *AWS and NVIDIA*. URL: https://aws.amazon.com/nvidia.
[23] *Time command manual page*. URL: https://man7.org/linux/man-pages/man1/time.1.html.
[24] *Amazon EC2 G3 Instances*. URL: https://aws.amazon.com/ec2/instance-types/g3.
[25] *Install NVIDIA drivers on Linux instances*. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/install-nvidia-driver.html.
[26] *NumPy*. URL: https://numpy.org.
[27] *Supported NumPy features in Numba*. URL: https://numba.pydata.org/numba-doc/0.47.0/reference/numpysupported.html.
[28] *Can't use basic NumPy functions with CUDA, like zeros or empty*. URL: https://numba.discourse.group/t/cant-use-basic-numpy-functions-with-cuda-like-zeros-or-empty/628.
[29] *Numba for CUDA GPUs*. URL: https://numba.pydata.org/numba-doc/dev/cuda/index.html.
[30] *How to Optimize Data Transfers in CUDA C/C++*. URL: https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc.
[31] nVidia. *SIMT Architecture*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#simt-architecture.
[32] *Convolution*. URL: https://en.wikipedia.org/wiki/Convolution.
[33] *Methods to compute linear convolution*. URL: https://www.gaussianwaves.com/2014/02/survey-of-methods-to-compute-convolution.
[34] Ron Dror. *Fourier transforms and convolution (without the agonizing pain)*. URL: https://web.stanford.edu/class/cs279/lectures/lecture8.pdf.
[35] *Transformada de Fourier*. URL: https://ca.wikipedia.org/wiki/Transformada_de_Fourier.

[36] *multiprocessing — Process-based "threading" interface.* URL: https://docs.python.org/2.7/library/multiprocessing.html.

[37] *Sharing CUDA Memory.* URL: https://numba.pydata.org/numba-doc/0.47.0/cuda/ipc.html.

[38] *argparse — Parser for command-line options, arguments and sub-commands.* URL: https://docs.python.org/2.7/library/argparse.html.

[39] *Python 2.7 built-in functions.* URL: https://docs.python.org/2/library/functions.html?highlight=round#round.

[40] *Welcome to PyCUDA's documentation!* URL: https://documen.tician.de/pycuda.

[41] Andreas Klöckner et al. ?PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation? A: *Parallel Computing* 38.3 (2012), 157–174. ISSN: 0167-8191. DOI: 10.1016/j.parco.2011.09.001. URL: https://www.sciencedirect.com/science/article/abs/pii/S0167819111001281?via%3Dihub.

[42] Ivan Rodero Castro i Frances Guim Bernat. ?Computació d'altes prestacions? A: vol. Mòdul didàctic 1 - Introducció a la computació d'altes prestacions. Cap. 1.2.1. Aplicacions HTC.

[43] Ivan Rodero Castro i Frances Guim Bernat. ?Computació d'altes prestacions? A: vol. Mòdul didàctic 5 - Introducció a la computació distribuïda. Cap. 2. Computació en graella.

[44] Javier Alexander Mahecha Guerrero. ?Procesamiento en paralelo de vídeos a través de un workflow de grafos acíclicos dirigidos (DAG)? Tesi de màster. Universitat Oberta de Catalunya, 2020.