Universitat de Girona
Escola Politècnica Superior

Treball final de grau

Estudi: Grau en Enginyeria Informàtica

Títol: Resolució de problemes de cobertura màxima amb múltiples cercles

Document: Memòria

Alumne: Moisès Saus Ten

Tutor: Marta Fort i Masdevall / Narcís Coll Arnau
Departament: Informàtica, Matemàtica Aplicada i Estadística
Àrea: LSI / MA

Convocatòria (mes/any): Juny 2019

Universitat de Girona
Escola Politècnica Superior

Escola Politècnica Superior
Universitat de Girona

Resolució de problemes de cobertura màxima amb múltiples cercles

TREBALL FINAL DE GRAU
GEINF

Document: Memòria
Autor: Moisès Saus Ten
Departament: Informàtica, Matemàtica Aplicada i Estadística
Àrea: LSI / MA
Grup de recerca: Laboratori de Gràfics i Imatge (GiLab)
Convocatòria: Juny 2019

Índex

1. Introducció ... 6
1.1 Motivacions ... 7
1.2 Propòsit i objectius del projecte ... 7
2. Estudi de viabilitat ... 8
3. Metodologia ... 9
4. Planificació ... 10
5. Marc de treball i conceptes previs ... 11
5.1 Marc de Treball ... 11
5.2 Conceptes previs ... 11
5.2.1 Problemes de cobertura ... 11
5.2.1.1 Solució al problema CMCP amb una facilitat ... 12
5.2.2 Metaheurística ... 14
5.2.2.1 Simulated annealing ... 14
5.2.3 CUDA ... 16
5.2.3.1 Arquitectura CUDA ... 17
5.2.3.2 Tipus de memòria CUDA ... 19
5.2.3.3 Sincronització ... 21
5.2.3.4 Funcions atòmiques ... 22
5.2.3.5 Texture fetching ... 23
5.2.3.6 Paral·lelisme dinàmic ... 26
6. Requisits del sistema ... 27
7. Estudis i decisions ... 28
7.1 Tipus de llicències ... 28
7.2 Llibreries utilitzades ... 29
7.2.1 CUDA ... 29
7.2.2 OpenGL ... 29
7.2.3 QT ... 30
7.2.4 Altres llibreries ... 30
7.3 Programari utilitzat ... 31
7.3.1 Visual Studio + Nsight + Qt VS Tools ... 31
7.3.2 Enterprise Architech ... 31
7.3.3 Draw.io ... 31
7.3.4 Gannt project ... 32
7.3.5 Stack ELK ... 32

7.3.6 GitLab 33
7.3.7 Docker 33
7.3.8 VirtualBox 33
8. Anàlisi i disseny del sistema 34
8.1 Anàlisi del problema i disseny de la solució 34
8.1.1 Càlcul dels nous cercles 36
8.1.2 Ús de la memòria de textura per al càlcul de les àrees. 37
8.1.3 Estratègies de paral·lelització 39
8.1.3.1 Mètode 1 39
8.1.3.2 Mètode 2 41
8.1.3.3 Mètode 3 43
8.1.3.4 Estratègia d'execució dels kernels 45
8.2 Fitxer d'entrada 47
8.3 Anàlisi i disseny de la interfície d'usuari 48
8.4 Anàlisi i disseny del codi 50
8.4.1 Immersió al codi de (Coll, Fort, & Sellarès, 2019) 50
8.4.2 Diagrama de classes 51
8.4.3 Diagrama de seqüència 58
8.5 Anàlisi de les dades 60
9. Implementació i proves 61
9.1 Implementacions 61
9.1.1 Generació de nombres aleatoris 61
9.1.2 Simulated Annealing 62
9.1.2.1 Lògica Simulated Annealing 63
9.2 Proves 64
9.2.1 Generar grid 64
9.2.2 Càlcul d'àrees del *grid* 64
9.2.3 Marge d'error *texture fetching* 65
9.2.4 Aleatorietat 66
10. Implantació i resultats 67
10.1 Implantació stack ELK 67
10.1.1 Implantació Elascticsearch 67
10.1.2 Implantació Logstash 69
10.1.3 Implantació Kibana 70
10.2 Resultats 71
11. Conclusions 74

12. Treball futur 75
13. Manual d'usuari i/o instal·lació 76
13.1 Execució del programa 76
13.1.1 Representació gràfica de la solució 78
13.1.2 Interpretació dels resultats 78
13.1.3 Pestanya "Solution" 79
13.2 Anàlisi del resultats 80
14. Bibliografia 82
15. Annexos 83
Annex A. Parallel Simulated Annealing for coverage area maximization 83

1. Introducció

El present projecte és una continuació de la recerca en el camp de la Geometria Computacional dels doctors Narcís Coll, Marta Fort i J. Antoni Sellarès (Coll, Fort, & Sellarès, 2019) del grup de recerca Graphics & Imaging laboratory (GILab) de la UdG que tracta de resoldre una de les múltiples variants dels problemes de cobertura.

Els problemes de cobertura s'ocupen de la col·locació d'un conjunt limitat de recursos per tal de cobrir una demanda optimitzant una funció objectiu. Com per exemple maximitzar cobertura, minimitzar costos o distància de viatge, etc. Els elements d'aquest conjunt limitat d'instal·lacions podrien ser torres de telefonia, sirenes d'advertència, etc. Aquest tipus d'instal·lacions sovint poden ubicar-se a quasi qualsevol lloc, ja que poden muntar-se en pals d'electricitat, antenes o estructures ja existents. Els problemes de cobertura són de gran aplicabilitat al planificar la ubicació de les instal·lacions tant en el sector públic com en el sector privat.

El problema és difícil de resoldre i molt costós computacionalment perquè poden considerar-se un nombre infinit de localitzacions, tant per la demanda del servei com per a les ubicacions de les instal·lacions. Per tant, les tècniques d'optimització estàndard per als models de localització discrets no són aplicables a aquest problema i requereixen noves tècniques eficients per a la solució del problema.

La programabilitat i les altes taxes computacionals de les Unitats de Processament Gràfic (GPU) les converteixen en una plataforma potent per a tasques computacionalment exigents on es necessita processar una gran quantitat de dades o realitzar una gran quantitat d'operacions. La capacitat de processament paral·lel de la GPU permet dividir tasques complexes de computació en milers de tasques més petites que es poden executar simultàniament.

Les GPUs s'han convertit ràpidament en un estàndard de la indústria que potencia milers d'ordinadors i estacions de treball de tot el món. En particular, la computació de propòsit general en les GPUs, conegut com a GPGPU per les seves sigles en anglès General-Purpose Computing on Graphics Processing Units, està atraient l'atenció dels investigadors. I s'utilitzen en molts camps computacionals que van des de les operacions numèriques de computació i simulacions físiques, fins a la bioinformàtica, mineria de dades i processament de geometria, ja que redueixen dràsticament els temps d'execució.

En (Coll, Fort, & Sellarès, 2019) es presenta una manera de resoldre el problema d'ubicar un únic recurs per cobrir una zona delimitada per segments de recta i arcs de circumferència utilitzant les GPU en el procés. El present projecte pretén prendre aquest punt de partida i ubicar no un únic recurs, sinó *k* recursos i distribuir-los de tal manera que obtinguin una cobertura màxima d'una regió de demanda, delimitada només per segments de recta utilitzant la GPU en el procés.

1.1 Motivacions

Sempre he tingut curiositat per la informàtica, he crescut envoltat d'ella des de petit. Des de la commodore 64 passant pels 386, els primers intel pentium, internet... i fins al dia d'avui. Han passat per les meves mans moltes màquines diferents i he teclejat potser massa vegades "format c:". Les targetes gràfiques no han estat excloses d'aquesta curiositat i més quan llegia o escoltava notícies de que s'utilitzaven per finalitats per les quals no havien estat dissenyades; em resultava realment curiós. Però era un món que em semblava realment llunyà, em limitava a ser usuari. Fins que vaig decidir saciar la meva curiositat estudiant la carrera d'enginyeria informàtica, la qual m'ha donat les bases per entendre aquestes tecnologies. Durant el transcurs de la carrera a la universitat em van arribar els rumors que el departament d'IMAE havia adquirit una targeta gràfica i que no era per jugar a videojocs. Així que la meva motivació principal per picar a la porta del departament i desenvolupar un projecte amb targetes gràfiques ha estat la curiositat.

També volia un problema que exigís un mínim a les capacitats de la targeta gràfica i que exigís també un mínim a l'hora de trobar una estratègia per a la paral·lelització. Com si d'un puzle es tractés.

Per altra banda, i ja centrats en el problema que ens ocupa, els recursos en el món actual solen ser escassos, o si més no, sempre s'ha d'intentar d'optimitzar-los. Per tant, una gestió eficient d'aquests recursos limitats sol ser un aspecte clau. En aquest cas existeix una necessitat d'ubicar estratègicament recursos per a maximitzar el seu rendiment i minimitzar el cost de cobrir una zona de demanda. Així que, aportar el meu granet de sorra i trobar una solució eficient i òptima a aquest problema amb GPU em semblava prou atractiu.

1.2 Propòsit i objectius del projecte

El principal propòsit del projecte és desenvolupar una eina que sigui capaç de trobar una solució òptima de forma eficient a un problema de cobertura on es busca trobar la màxima cobertura en una àrea donat un nombre k de recursos. Un segon propòsit és el de la presentació de les dades i els resultats, de tal manera que sigui fàcil per a un usuari utilitzar i interpretar els resultats.

Partint d'aquests propòsits sorgeixen tres objectius. Un d'ells és trobar un algorisme eficient que s'adeqüi a la morfologia del problema i que al mateix temps es pugui paral·lelitzar. La cerca de la millor estratègia per a la paral·lelització d'aquest algorisme amb les targetes gràfiques és un segon objectiu. I per últim la presentació de les dades i els resultats. Es pretén desenvolupar una interfície per a visualitzar la informació obtinguda, amb la finalitat que aquesta sigui fàcilment utilitzable.

2. Estudi de viabilitat

Des d'un punt de vista tecnològic els elements que fan falta per al desenvolupament d'aquest projecte són un ordinador amb targeta gràfica compatible amb CUDA. Les targetes gràfiques són cares a causa del seu potencial, però tant jo, personalment, com el departament d'IMAE ja comptàvem amb aquest recurs. Un altre aspecte a tenir en compte són les llibreries i software de tercers, que com es veurà més endavant en l'apartat 7, la utilització d'aquests no té cap cost. Per tant, ja es comença amb els recursos tecnològics necessaris per al desenvolupament del projecte i no és necessari la inversió econòmica en cap recurs.

Per altra banda, des d'un punt de vista d'investigació caldria analitzar si el problema que ens ocupa és un problema paral·lelitzable i per tant té sentit utilitzar la potència que ofereixen les targetes gràfiques. Sense entrar en grans detalls, es tracta d'un problema de cobertura on es busca la màxima cobertura en una àrea donat un nombre *k* de recursos. L'exploració d'aquest espai de solucions és paral·lelitzable, ja que la creació i avaluació de cada solució és independent una de l'altre(Figura 1). Aleshores té sentit intentar utilitzar les GPU's per a aquesta cerca.

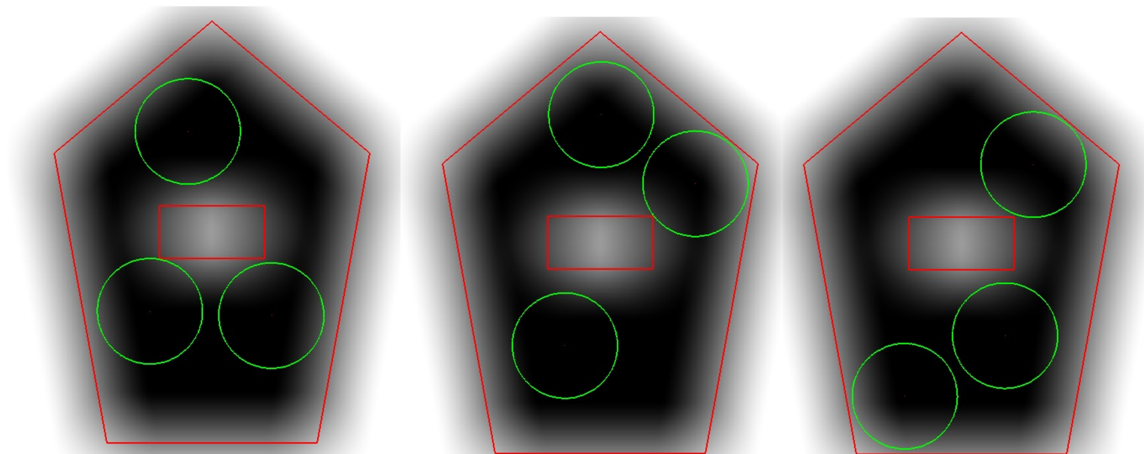


Figura 1. Possibles estats d'un espai de cerca

3. Metodologia

El desenvolupament del projecte presentava certes incerteses al moment de planificar les implementacions, ja que s'havien d'utilitzar tecnologies com OpenGL, CUDA i QT, que són tecnologies que jo mai havia utilitzat abans, i que requerien un aprenentatge previ. El fet de la desconeixença d'aquestes tecnologies feia difícil calcular la magnitud de feina que podria comportar en general, però una de les metodologies que més s'adequa aquest escenari és el Scrum.

Bàsicament es fa un llistat de les funcionalitats que es vol que tingui el nostre programari. Per a cada funcionalitat es fa un llistat de tasques. A partir d'aquí es prepara un sprint i s'implementa. Un Sprint és una llista d'una o més tasques. Les tasques són més fàcils de planificar en el temps, ja que són coses molt concretes.

Un cop implementada es produïa la reunió amb l'equip, normalment setmanals i a vegades quinzenals. De les reunions sorgien dues coses, per una banda sorgien noves llistes de tasques i per l'altra banda es feia un feedback de les tasques implementades. Si tasques implementades eren millorables o inacabades aleshores es torna a afegir a la llista inicial junt amb les noves llistes de tasques. En la Figura 2 es mostra el diagrama de la metodologia que s'ha seguit per al desenvolupament del projecte.

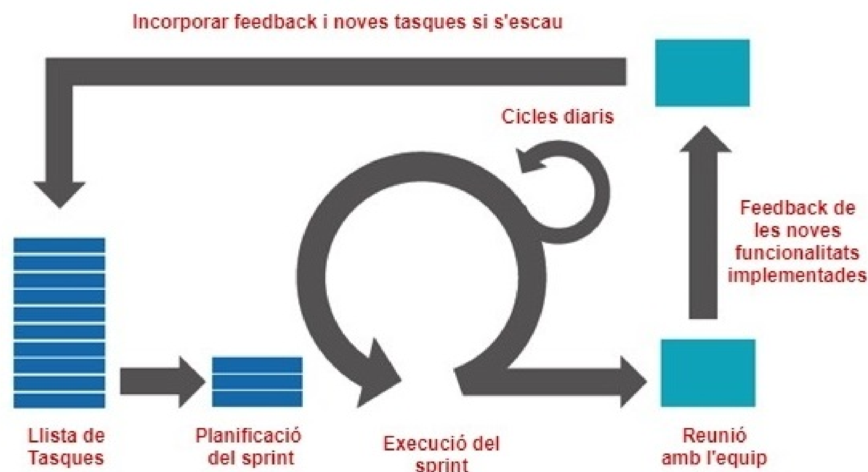


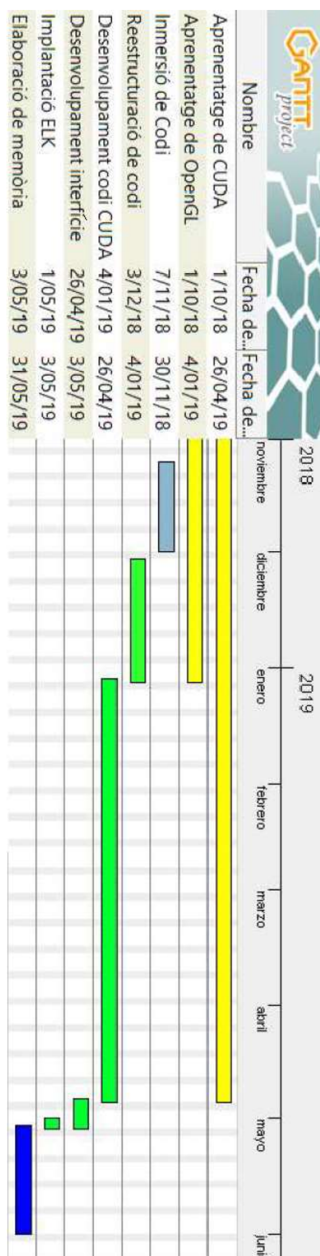
Figura 2. Diagrama de flux de la metodologia Scrum

4. Planificació

La planificació del projecte és com la que es mostra en la Figura 3. Primer calia un aprenentatge previ de les tecnologies OpenGL i CUDA. En el cas de CUDA ha estat un aprenentatge constant durant tot el desenvolupament del projecte. El present projecte és la continuació del treball d'investigació (Coll, Fort, & Sellarès, 2019), per tant també calia fer una immersió en el codi per tal de saber com funcionava i un cop analitzat es va fer una reestructuració d'aquest codi per tal detenir un codi més net, modulable i més fàcil de mantenir en el temps.

En verd és mostra tota la fase del scrum. Ha estat tota la fase de desenvolupament i reunions amb l'equip, on parlàvem de noves funcionalitats, estratègies de paral·lelització, anàlisis de resultats, coses a millorar, etc..

I per últim, les ultimes setmanes s'han dedicat a la redacció del present document.



5. Marc de treball i conceptes previs

Per a poder seguir bé el projecte, abans d'entrar en profunditat amb la temàtica cal descriure el context en què s'ha desenvolupat (capítol 5.1) i donar a conèixer un seguit de conceptes previs necessaris (capítol 5.2).

5.1 Marc de Treball

Com ja s’ha mencionat anteriorment en la introducció, el present projecte és un projecte d’investigació del departament d’IMAE de la Universitat de Girona en el marc de la recerca de GiLab. El treball previ a aquest projecte (Coll, Fort, & Sellarès, 2019) fet per membres del departament, resol el problema de cobertura màxima contínua (CMCP) amb mètrica euclidiana, amb una única facilitat i utilitzant les capacitats de les Unitats de Processament Gràfic (GPU) per accelerar, mitjançant la paral·lelització, l’obtenció de la solució. Tenint en compte aquest punt de partida, el que es pretén en aquest projecte és seguir avançant i considerar *k* facilitats al problema, en lloc d’una sola facilitat.

La Doctora Marta Fort Masdevall i el Doctor Narcís Coll Arnau són professors i investigadors del departament d’IMAE de la UdG, ells han col·laborat i guiat en l’elecció dels algorismes i en les diferents estratègies de paral·lelització que es presentaran en els següents capítols.

5.2 Conceptes previs

En els següents apartats es pretén fer un repàs i/o introducció a tots aquells conceptes que aniran apareixent al llarg d’aquest document. En particular presentem els problemes de cobertura (capítol 5.2.1), metaheurístiques per a resoldre problemes complexos (capítol 5.2.2) i aspectes relacionats amb CUDA (capítol 5.2.3).

5.2.1 Problemes de cobertura

L’objectiu principal d’aquest grup de problemes és el de cobrir una regió de demanda de forma total o parcial. O dit d’una altra manera, trobar la localització òptima d’un conjunt limitat de recursos, que en aquest àmbit es denominen ***facilitats***, de tal manera que es cobreixi la demanda total o parcial. Els problemes de localització d’instal·lacions utilitzen el concepte de **cobertura**. Una demanda és coberta per un recurs si la distància o el temps de viatge són menors a un cert valor predeterminat anomenat radi de cobertura.

Els problemes de cobertura es poden diferenciar de dos tipus:

- *Problema de Localització de Cobertura d’un Conjunt* (**SCLP** per les seves sigles en anglès) que consisteix a minimitzar el nombre de facilitats per aconseguir la cobertura de tota la demanda. El SCLP va ser formulat per primera vegada com a un problema de programació lineal entera (Toregas, Swain, Charles, & Bergman, 1971) (Toregas & ReVelle, Binary Logic Solutions to a Class of Location Problems, 1973). La primera aplicació d’aquest model va ser en el

camp dels serveis d'emergència. En (Daskin, 1983), Daskin va incorporar a aquest model els problemes de disponibilitat horària de congestió de tràfic.

- *Problema de Localització de Màxima Cobertura* (**MCLP** per les seves sigles en anglès). En aquest cas es restringeix el nombre de facilitats de tal manera que s'ha de maximitzar la cobertura a partir d'aquest nombre fix de facilitats. Les primeres investigacions sobre aquest model les van realitzar Church i ReVelle (Church & ReVelle, 1974) i White i Case (White & Case, 1974) on van formular el MCLP, també com a un problema de programació lineal entera. En treballs posteriors es proposen metodologies per resoldre aquest problema (Megiddo, Zemel, & Hakimi, 1983) (Resende, 1998).

Tant el model SCLP com el MCLP són del tipus NP-hard (Garey & Johnson, 1979) (Megiddo, Zemel, & Hakimi, 1983). Originalment en el model MCLP, tant la demanda com les facilitats van ser concebudes per a ser un conjunt finit de punts, però en la majoria de casos la ubicació de la demanda i de les localitzacions es distribueixen contínuament sobre una regió en el pla. En (Murray, Matisziw, Hu, & Tong, 2008) s'introdueix el Problema de Cobertura Màxima Continua (**CMCP**), en el que la demanda es distribueix per tota la regió i les facilitats poden estar ubicades en qualsevol part de la regió. Per exemple, en buscar localitzacions per situar sirenes d'emergència, torres de comunicació, etc. I és justament aquesta variant del model MCLP en la que es treballa en (Coll, Fort, & Sellarès, 2019) i en la que aprofundirem en aquest projecte: El CMCP.

En (Matisziw & Murray, 2009) es proposa un plantejament geomètric per a abordar el problema amb una sola facilitat amb radi de cobertura circular en una regió no-convexa on la demanda està uniformement distribuïda.

5.2.1.1 Solució al problema CMCP amb una facilitat

En (Coll, Fort, & Sellarès, 2019), el departament d'IMAE de la UdG resol el CMCP per a una sola facilitat, assumint una demanda distribuïda uniformement i tenint en compte les cobertures parcials. El problema és difícil de resoldre i molt costós computacionalment, perquè han de considerar-se un nombre infinit de localitzacions, tant per la demanda com per les facilitats. Per tant les tècniques d'optimització per als models de localització discreta no són aplicables al problema CMCP.

Per tal de compensar el cost computacional dels càlculs, en (Coll, Fort, & Sellarès, 2019) s'utilitzen les capacitats de les GPUs. La capacitat de processament paral·lel de la GPU permet dividir tasques complexes de computació en milers de tasques més petites que es poden executar simultàniament. Aquesta capacitat permet obtenir la solució de molts problemes amb la GPU en una fracció de temps més petita de la requerida per la CPU.

El que es planteja en (Coll, Fort, & Sellarès, 2019) és que l'usuari indiqui el polígon que defineix la regió de demanda, el radi de la facilitat i l'error que s'accepta que hi hagi a la solució. Aleshores el mètode calcula una graella adequada a l'error i radi indicats per l'usuari. Obtenint d'aquesta manera un conjunt discret d'ubicacions. I per cada punt es calcula l'àrea d'intersecció entre el cercle i el polígon que defineix la regió de demanda, formant un mapa de punts com el que es mostra en la Figura 4. On es reflecteix tota la informació extreta pel mètode mitjançant un codi de colors. Els punts vermells són les solucions ideals, aquelles en què si s'hi col·loca el centre de la facilitat, el cercle que la defineix està completament contingut a la regió de demanda. Les verdes són les solucions òptimes, l'àrea que es cobrirà és suficientment gran d'acord amb un llindar d'acceptació que estableix l'usuari al començar el procés. La resta de possibles ubicacions analitzades estan pintades en una gradació de blaus de manera que com més clar és el blau més àrea cobreix el cercle centrat en aquell punt.

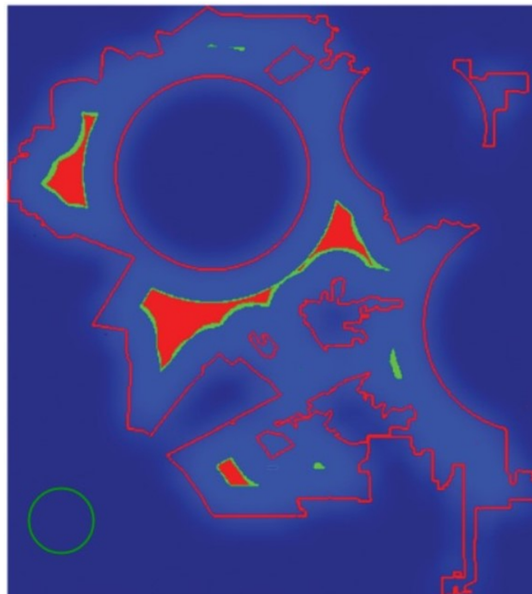


Figura 4. Imatge del projecte de investigació (Coll, Fort, & Sellarès, 2019)

5.2.2 Metaheurística

Quan parlem de problemes d'optimització *NP-hard* estem fent referència a problemes de difícil solució els quals presenten una gran complexitat computacional per a ser resolts. Són aquells pels quals no podem garantir trobar la millor solució possible en un temps raonable. La resolució d'aquests es realitza a través d'algoritmes aproximats. Aquests proporcionen solucions considerades bones per a un determinat problema, en un temps moderat, però no necessàriament la solució òptima. Aquests mètodes, en els que la rapidesa del procés és tan important com la qualitat de la solució es denominen heurístics o aproximats.

Amb el propòsit d'obtenir millors resultats que els assolits per els heurístics tradicionals sorgeixen els denominats procediments metaheurístics. Aquests procediments són una classe de mètodes aproximats que estan dissenyats per a resoldre problemes d'optimització complexos, en els que els heurístics clàssics no són efectius. Segons afirmen (H Osman & P. Kelly, 1997) les metahurístiques proporcionen un marc general per a crear nous algoritmes híbrids combinant diferents conceptes derivats de la intel·ligència artificial, evolució biològica i els mecanismes estadístics. L'aplicació de tècniques metaheurístiques és especialment interessant en casos de problemes d'optimització combinatòria.

Les tècniques metaheurístiques més esteses són els algoritmes genètics, Tabu Search, Simulated Annealing, Scatter Search, entre d'altres.

5.2.2.1 Simulated annealing

Donada la complexitat del problema CMCP, que té un espai molt ampli de solucions, i més quan es pretén col·locar no una única facilitat sinó *k*, vam considerar necessari utilitzar un mètode metaheurístic per trobar la solució i ens vam inclinar pel Simulated Annealing.

El nom de l'algorisme Simulated annealing ve donat pel procés de recuit dels acers i les ceràmiques, que consisteix a escalfar i després controlar la velocitat de refredament del material per tal de variar les seves propietats físiques. Si es fa adequadament, l'estat final del metall és un estat de mínima energia. La calor causa que els àtoms augmentin la seva energia i puguin així desplaçar-se de les seves posicions inicials (un mínim local d'energia). El refredament lent els hi dóna majors probabilitats de recristal·litzar en configuracions de menys energia que la inicial (mínim global).

Aquest algorisme és una mescla entre l'algorisme Hill-Climbing i un algorisme totalment aleatori. L'algorisme Hill-climbing és un algorisme de cerques locals. És a dir, els seus moviments estan determinats per ser millors que els previs. Comença en un punt aleatori en l'espai de cerca, a partir d'aquí si el nou punt de cerca és millor, aleshores, es transforma en el punt actual; en cas contrari, es selecciona i avalua un altre punt veí. El mètode finalitza quan no es troben punts veïns millors.

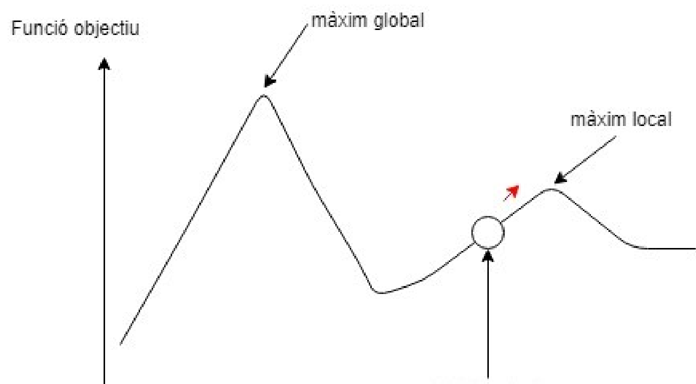


Figura 5. Cerca local, algorisme Hill Climbing

L'algorisme Hill-Climbing és un algorisme incomplet, ja que pot estancar-se en màxims locals i no assolir un màxim global. Per altra banda, un algorisme que prengués solucions completament aleatòries seria complet però totalment ineficient.

L'algorisme Simulated annealing està a la meitat entre aquests dos algorismes, combina el Hill-climbing amb l'aleatorietat. En la Figura 6 es mostra el pseudocodi de l'algorisme, on:

- s és l'estat actual
- s' és el nou estat
- m és el millor estat
- E és la funció objectiu
- Temp disminueix a mesura que l'algorisme avança

Cada vegada que es genera un nou estat aquest és avaluat i si el nou estat és millor, aleshores l'escollim, altrament l'escollim amb una certa probabilitat. La probabilitat depèn de Temp i de la diferència entre les funcions objectiu de l'estat actual i el nou estat.

```
1   Crear solució inicial s
2   m = s
2   Inicialitzar temperatura Temp
3   For i = 1 to numIteracions
5      Generar un veï s' aleatori de s
6        If (E(s') ≥ E(m)
7            m = s'
8        If (E(s) ≥ E(s')
9            s = s'
10       else
11           if (e^((E(s)-E(s'))/Temp) > random[0,1))
12               s = s'
13      Reduir Temperatura Temp
14  Fi For
15  Resultat m
```

Figura 6. Pseudo-codi Simulated Annealing

Aquesta probabilitat d'escollir un estat encara que empitjori l'estat actual evita quedar estancats en mínims locals. Per altra banda, a més de l'estat actual es guarda també l'estat òptim d'entre els analitzats.

5.2.3 CUDA

Compute **U**nified **D**evice **A**rchitecture o Arquitectura Unificada de Dispositius de Còmput és una arquitectura de càlcul paral·lel de NVIDIA que permet als programadors aprofitar la potència de les GPU i el seu paral·lelisme per a desenvolupar aplicacions, algorismes, etc. i tot allò que requereixi una potència de càlcul elevada i que compleixi amb les condicions per a poder ser paral·lelitzat.

La funció principal per la qual sempre han estat dissenyades les targetes gràfiques ha estat poder delegar la tasca de visualització per pantalla a aquests dispositius. El gran creixement de la indústria dels videojocs sempre ha anat lligat a la indústria d'aquests dispositius, ja que a mesura que els dispositius podien oferir més capacitat de càlcul, els desenvolupadors de jocs podien oferir més qualitat d'imatge i escenaris de joc més grans.

El 1999 NVIDIA inventa la GPU (**G**raphics **P**rocessing **U**nit). Es tracta d'un microprocessador dedicat exclusivament al processament de gràfics o operacions de coma flotant. Aquest mateix any comercialitza la primera targeta gràfica que incorpora GPU's, la targeta GeForce 256. La qual era capaç de processar més de 10 milions de polígons per segon. Les GPUs actuals processen més de 2000 milions de polígons per segon. I en el 2001 NVIDIA introdueix la primera GPU programable, la Nvidia GeForce3, això va suposar un fet important dins de la indústria, ja que va permetre als desenvolupadors crear efectes visuals personalitzats.

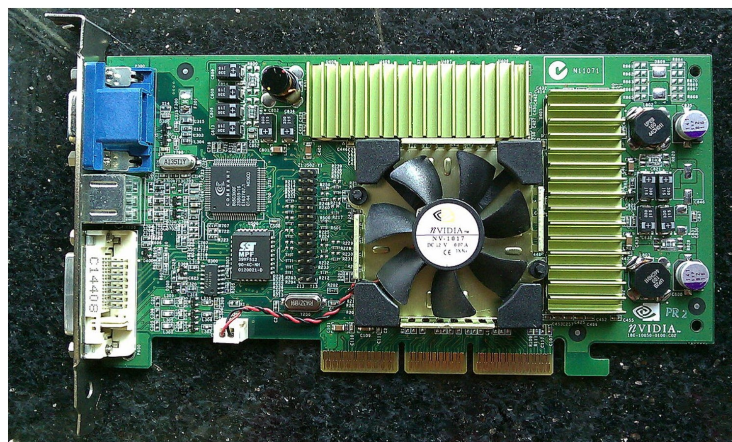


Figura 7. Model GeForce 3 Ti 500 de NVIDIA

Científics i investigadors van començar a utilitzar les targetes gràfiques per al càlcul científic. Va ser el naixement del **GPGPU** (General-Purpose Computing on Graphics Processing Units). El problema principal és que requeria l'ús de llenguatges de

programació específics per a gràfics com OpenGL, fet que limitava l'accés del món científic a les targetes gràfiques.

Nvidia, es va adonar del potencial que suposava obrir aquest rendiment a la comunitat científica en general i va decidir investigar la manera de modificar l'arquitectura de les seves GPUs per a que fossin completament programables per a aplicacions científiques a més d'afegir suport per a llenguatges d'alt nivell com C, C++, Fortran... Així que el 2006 NVIDIA presenta CUDA, una arquitectura de càlcul en la GPU que permet als científics i investigadors aprofitar aquesta capacitat de processament paral·lel que ofereixen les GPU per a resoldre problemes computacionals més complexos. És el naixement del GPU Computing, l'ús de targetes gràfiques per a realitzar càlculs científics de propòsit general.

Des de l'aparició de CUDA fins el dia d'avui, CUDA s'ha utilitzat i s'utilitza en molts camps: algorismes, mètodes numèrics, fotografia computacional, deep learning i IA, networking, visió per computador, robòtica, etc...

El fet de tenir unitats de processament (GPU) programables fan de CUDA una molt bona plataforma per a tasques de computació exigents que necessiten processar gran quantitat de dades o realitzar moltes operacions. La capacitat de processament paral·lel de les GPU permet dividir tasques de computació complexes en milers de tasques més petites que es poden executar simultàniament.

En les següents seccions d'aquest capítol s'explicaran tots aquells conceptes, característiques i funcionalitats de CUDA que són necessaris saber per poder extreure totes les capacitats de les targetes gràfiques. Són els coneixements que he anat adquirint durant el desenvolupament d'aquest projecte.

5.2.3.1 Arquitectura CUDA

Tal com es mostra en la Figura 8, les arquitectures CUDA estan formades per unitats d'execució denominades *Streaming Multiprocessors* (**SM**), 16 unitats en l'exemple de la Figura 8, que estan interconnectades entre si per una zona de memòria comú. Cada SM està compost per uns nuclis de còmput anomenats *CUDA core* o *Streaming Processors* (**SP**), que són els encarregats d'executar les instruccions. En el nostre exemple de la Figura 8 veiem que hi ha 32 SP per cada SM, per tant tenim 512 nuclis de processament.

Figura 8. Arquitectura d’una targeta gràfica CUDA

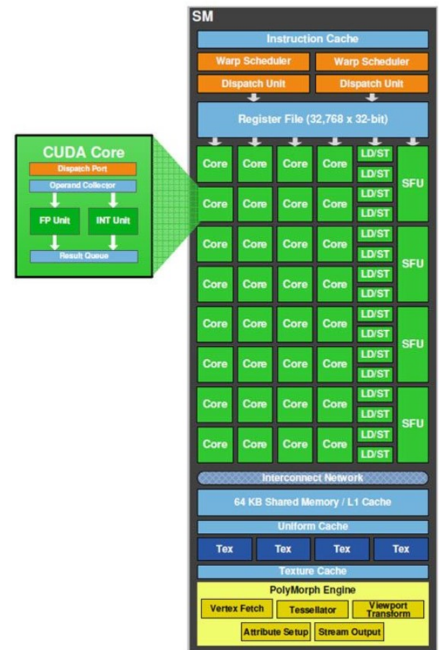


Figura 9. Streaming Processor

El que permet aquesta arquitectura al programador és una programació senzilla utilitzant un llenguatge d’alt nivell i dins del qual es fan crides al que s’anomena **Kernel**. Un *Kernel* és una funció o programa sencer que s’executa de forma paral·lela com un conjunt de fils (***threads***) i que el programador organitza en blocs (***blocks***), i al mateix temps aquests es poden distribuir formant una malla (***grid***) tal com es mostra en la Figura 10.

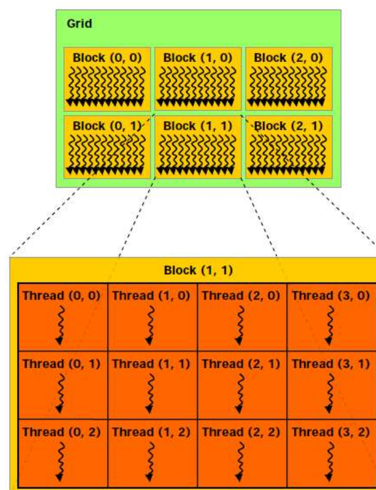


Figura 10. Jerarquia de threads en una aplicació CUDA

La capacitat d'una GPU per abordar un problema depèn dels seus recursos de hardware: Nombre de nuclis (SP), nombre de threads que pot generar, nombre de registres disponibles, capacitat de les memòries, etc. A aquesta capacitat, CUDA l'anomena ***Compute Capability*** i s'indica mitjançant dos números de la forma C.c, que representen la revisió major i la revisió menor respectivament de l'arquitectura del dispositiu.

5.2.3.2 Tipus de memòria CUDA

Els dispositius CUDA tenen el seu propi espai de memòria. Un *kernel* només pot operar sobre la memòria del dispositiu, per això les llibreries de CUDA ens ofereixen funcions específiques per a reservar, alliberar i transferir de dades entre el *Host* i el *Device*. En la Figura 11 es mostra la jerarquia de memòria en un dispositiu de CUDA. La **memòria global**, la **memòria constant** i la **memòria de textures** són accessibles des de el *Host* i el *Device* i des d'aquestes zones el *kernel* pot transferir dades a la resta de nivells. Es pot observar que tots el *threads* poden accedir a la memòria global, memòria constant i memòria de textures mentre que únicament els *threads* pertanyents a un mateix *block* poden accedir a un espai de memòria anomenat **memòria compartida**. Cada tipus de memòria té les seves peculiaritats. Detallarem les més importants:

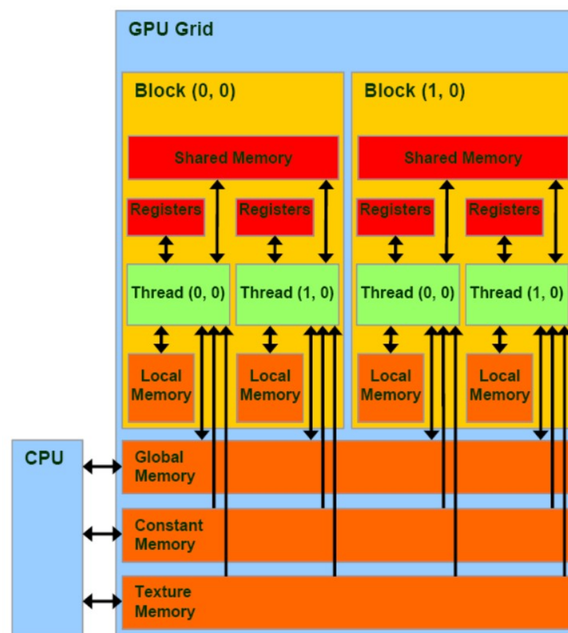


Figura 11. Jerarquia de memòria en CUDA

- ***Shared Memory***

El fet més rellevant de la *shared memory* és que es troba en el mateix xip que el processador i que té molt baixa latència, és molt semblant a una memòria *caché* de CPU. Això fa que sigui una memòria molt més ràpida que els espais de memòria constant i global, i per contra és una memòria de molt poca capacitat. El més habitual és utilitzar aquesta memòria per a realitzar càlculs, per a col·locar dades que són consultades amb freqüència, o bé per llegir o escriure resultats que altres *threads* del mateix *block* necessiten.

La shared memory es divideix en bancs de memòria, que a partir d'ara anomenarem ***banks***, els quals s'hi pot accedir de forma simultània. Això significa que qualsevol accés a *n* adreces de memòria que corresponguin a *n banks* diferents poden ser atesos simultàniament. El problema es troba quan dos accessos coincideixen en un mateix *bank*, tal com es mostra en la Figura 12. En aquest cas els accessos es realitzen de forma seqüencial, fet que fa que disminueixi l'amplada de banda.

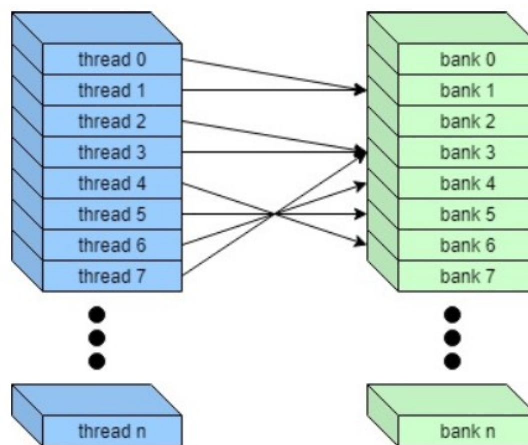


Figura 12. Exemple bank conflict

- ***Global memory***

És una memòria de gran capacitat i que ens permet tant escriptura com lectura. És visible des de tots els *threads* de tots els *blocks* i accessible tant des del *host* com el *device* però és la memòria amb menys amplada de banda.

- ***Constant memory***

S'utilitza per emmagatzemar dades que són constants, tal com el seu nom indica. La memòria constant és semblant a global memory amb la peculiaritat que és només de lectura i per tant és tractada de forma diferent. La memòria constant té una amplada de banda molt més gran que la memòria global i una de les raons és perquè té associada una memòria *caché* que es troba a dins de cada *streaming multiprocessor*, per tant afavoreix les lectures consecutives sobre una mateixa adreça de memòria.

- ***Texture memory***

Com la memòria constant la memòria de textura també té associada una memòria *caché*, per el que també proporciona un gran amplada de banda a l'hora de fer lectures. Aquestes *caches* de textura estan dissenyades per aplicacions gràfiques on els patrons d'accés a la memòria manifesten una gran quantitat de localitats espaials. Per a aquest tipus de memòria CUDA proporciona certes funcionalitats de lectura, típiques de les textures de gràfics, i que més endavant, en la capítol 8.1.2, explicarem com les apliquem en el nostre projecte.

En la Taula 1 es mostra un quadre resum de les característiques més rellevants de les memòries de les que s'han fet menció.

Memory	Location	Cached	Access	Scope	Lifetime
Register	On-Chip	N/A	R/W	One thread	Thread
Local	Off-Chip	No	R/W	One thread	Thread
Shared	On-Chip	N/A	R/W	All threads in a block	Block
Global	Off-Chip	No	R/W	All threads + host	Application
Constant	Off-Chip	Yes	R	All threads + host	Application
Texture	Off-Chip	Yes	R	All threads + host	Application

Taula 1. Taula de comparació de les diferents memòries

5.2.3.3 Sincronització

CUDA ens ofereix la possibilitat de dividir una tasca en milers de *threads* que s'executen de forma independent i que inclús tenen el seu propi espai de memòria. Però en molts casos existeix la necessitat de cooperar entre *threads* compartint dades a través d'alguna zona de memòria, i per tant és necessari que un *thread* A esperi al fet que el *thread* B acabi la feina. CUDA ofereix la possibilitat d'especificar punts de sincronització dins del codi del *kernel* fent crides a la funció:

```
__syncthreads();
```

Aquesta funció actua com una barrera fent que tots els *threads* d'un mateix *block* esperin abans de continuar amb la seva execució. Garanteix que tots els *threads* hagin executat totes les instruccions anteriors a la crida abans de continuar.

La possibilitat de sincronitzar *threads* de diferents *blocks* no va aparèixer fins a la versió CUDA 9.0, la qual incorpora una API anomenada *Cooperative Groups*. No totes les arquitectures de CUDA ho suporten. Però existeixen tècniques per provocar la sincronització entre *blocks*. La més habitual és dividir un *kernel* en dos, ja que no s'iniciarà el segon *kernel* fins que finalitzi el primer.

5.2.3.4 Funcions atòmiques

Quan es treballa en aplicacions *multi-thread* apareix el problema d'escriptura i lectura de diferents *threads* en un mateix espai de memòria. Imaginem que dos *threads* han d'executar la següent expressió:

```
X++;
```

Aquesta instrucció implica: llegir la variable, modificar i tornar a escriure el resultat a la variable. Al no haver-hi cap control sobre l'ordre de les instruccions, ja que l'ordre el genera el compilador, no tenim cap garantia que no succeeixi la seqüència que es mostra en la Taula 2, en la qual dos *threads* llegeixen el valor de la variable i li sumen 1, però en canvi el resultat final no és el desitjat.

Seqüència	Exemple
1. Thread A llegeix el valor de X	A llegeix 7 de X
2. Thread B llegeix el valor de X	B llegeix 7 de X
3. Thread A suma 1 al valor llegit	A suma 7 +1
4. Thread B suma 1 al valor llegit	B suma 7 + 1
5. Thread A escriu el valor a X	X <-8
6. Thread B escriu el valor a X	X <-8

Taula 2. Exemple de seqüència d'execució read-modify-write amb multithread

És a dir, el resultat final depèn de l'ordre de les instruccions dels diferents *threads* que treballen sobre un mateix recurs. Aquest ordre és arbitrari, ja que el determina el compilador i això pot generar resultats no desitjats com el de la Taula 2. Aquest problema és conegut amb el nom de *condition race*. Per a solucionar aquest problema CUDA ofereix un conjunt d'operacions anomenades operacions atòmiques. Aquestes operacions atòmiques són operacions que realitzen les tasques de llegir, modificar i escriure un valor de la memòria sense la interferència de cap altre *thread*.

És important minimitzar l'ús d'aquestes funcions atòmiques, ja que si molts *threads* diferents realitzen una operació atòmica a la mateixa adreça de memòria, aquestes operacions seran serialitzades. La penalització en temps serà proporcional al nombre de *threads* que hagin d'accedir simultàniament a la mateixa posició de memòria.

A continuació detallarem les funcions atòmiques utilitzades en aquest projecte, per obtenir una llista completa de les funcions atòmiques disponibles, llegir CUDA Toolkit Documentation (NVIDIA Corporation, 2019, pág. B.12).

En el projecte s'ha utilitzat dues funcions atòmiques:

- `int atomicMax(int* address, int val)`: Llegeix el valor de *address* i el compara amb *val*. El valor més alt serà emmagatzemat a *address* i el valor més petit serà retornat.

- `int atomicAdd(int* address, int val)`: Llegeix el valor de *address*, li suma *val* i emmegazema el nou valor a *address*. Retorna l'antic valor de address.

Es podria donar el cas que necessitéssim alguna cosa més que una funció atòmica, un tros de codi dins d'un *kernel* que necessitem serialitzar. Tot i que no és una pràctica gens recomanable, existeix la possibilitat de crear un *lock* mitjançant funcions atòmiques. El que es coneix en el món de la programació concurrent com a *mutex* (mutual exclusion). Aquesta pràctica es pot consultar a (Sanders & Kandrot, 2010, págs. 251-254)

5.2.3.5 Texture fetching

Ja s'ha esmentat anteriorment que les memòries de textura de les arquitectures CUDA tenen una amplada de banda superior al de l'espai de memòria global de les targetes i que aquesta memòria de textura incorpora algunes funcionalitats de lectura. És el que es coneix com a *texture fetching*.

Les textures es guarden en la memòria de textura en forma de matriu, i aquestes tenen diversos atributs. Un d'aquests atributs és la seva dimensionalitat, que especifica si la textura es tracta com una matriu unidimensional que utilitza una coordenada de textura, una matriu bidimensional que utilitza dues coordenades de textura o una matriu tridimensional que utilitza tres coordenades de textura. Els elements d'aquestes matrius s'anomenen **texels** i el seu tipus està restringit a enters o reals.

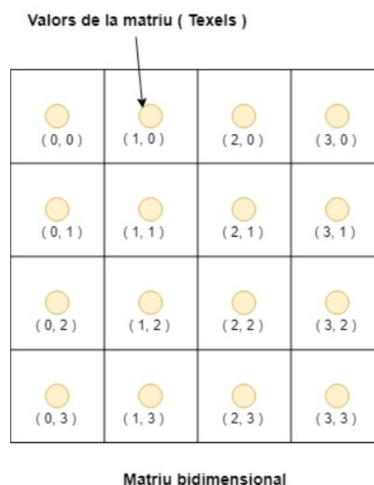


Figura 13. Representació d'una textura bidimensional

Les coordenades amb les quals s'accedeix a la textura, els índexs, són per defecte nombres reals, nombres de coma flotant entre [0.0, N.0] (*Read mode element type*), sent N la dimensió de la matriu, encara que també es poden normalitzar en el rang [0.0, 1.0] o [-1.0, 1.0] (*Read mode normalized float*). Aquest atribut CUDA l'anomena *Texture read mode*. La Figura 14 il·lustra aquestes diferents maneres d'adreçament en una textura de 4x4.

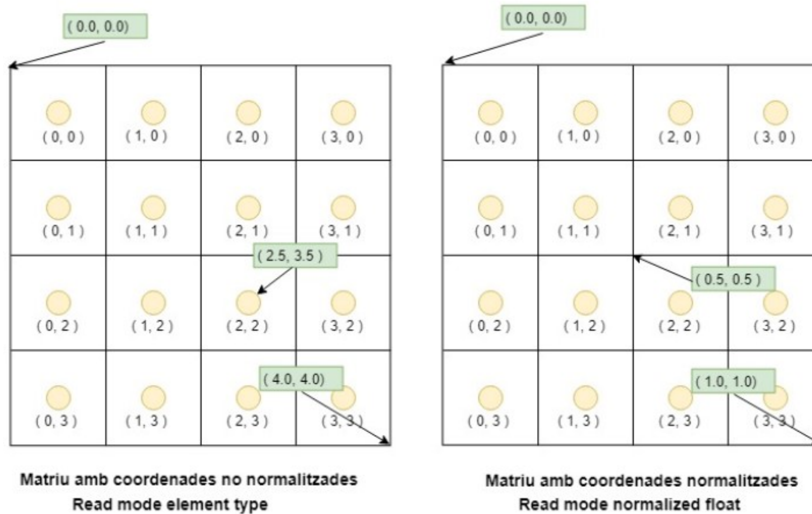


Figura 14. Tipus d'indexació a les textures

Ara bé, quin és el *texel* que retorna? CUDA ho anomena *Texture filtering mode* i té dues modalitats: *Point filter mode* i *Linear filter mode*. Per defecte utilitza *Point filter mode* que retorna el *texel* més proper a les coordenades. L'altra modalitat és Linear *filter mode*, la qual fa una interpolació amb els 2, 4, o 8 veïns més propers a les coordenades segons tinguem una matriu de 1D, 2D o 3D. La Figura 15 mostra gràficament aquestes dues modalitats.

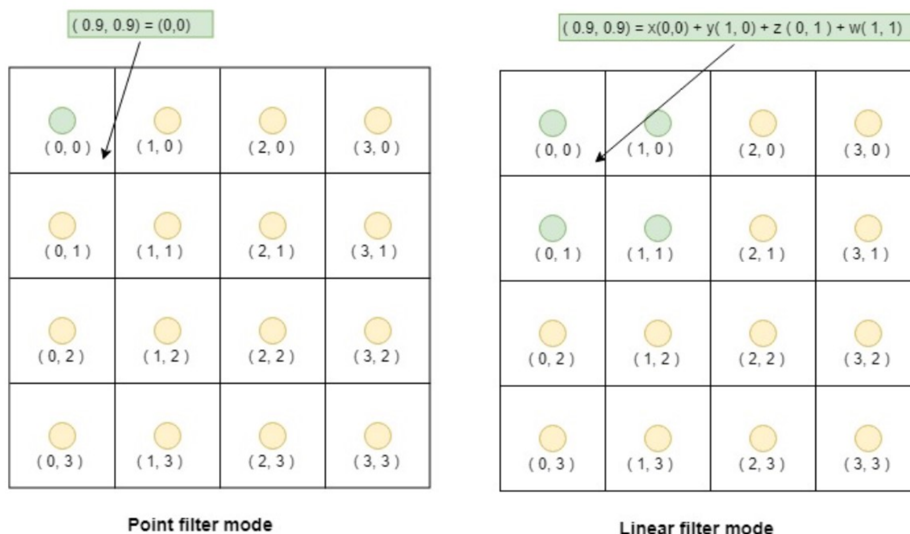


Figura 15. Tipus de texture filtering

Un dels altres atributs a tenir en compte és el comportament de l'accés fora de rang. CUDA el defineix com a *Texture addressing mode*. El qual pot ser diferent per a cada dimensió i tenim quatre modalitats:

- *Wrap*: Copia la textura a les bandes.

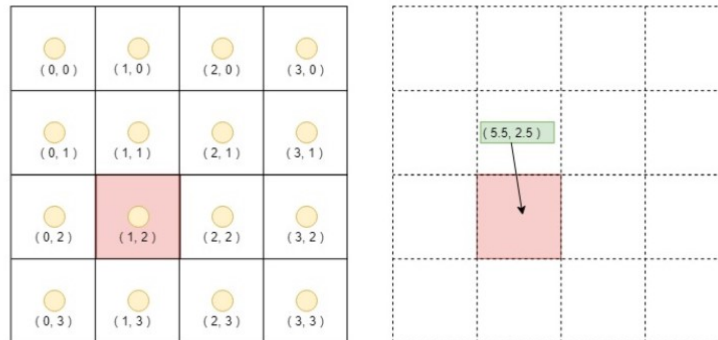


Figura 16. Texture addressing wrap mode

- *Clamp*: Les coordenades fora dels límits es substitueix per la frontera més propera.

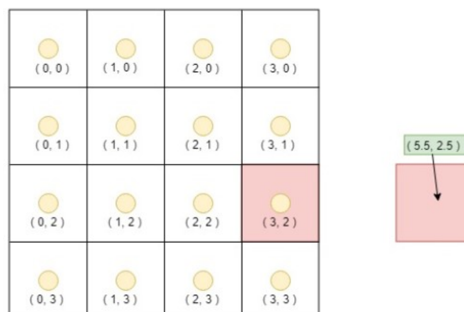


Figura 17. Texture addressing clamp mode

- *Mirror*: Efecte mirall.

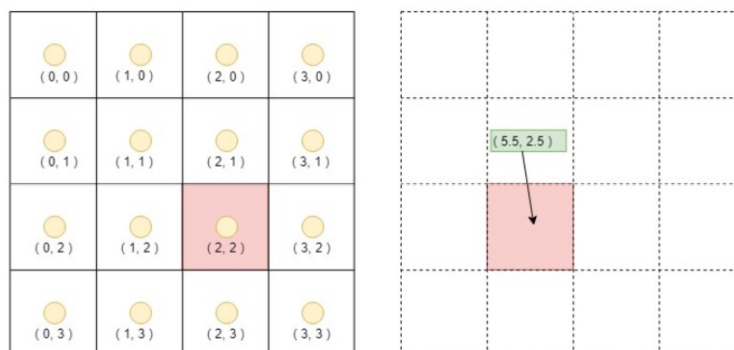


Figura 18. Texture addressing mirror mode

- *Border*: Retornarà 0 si accedim fora de rang.

5.2.3.6 Paral·lelisme dinàmic

El paral·lelisme dinàmic de CUDA permet a un kernel llançar un altre kernel niat dintre seu. Aquesta funcionalitat està disponible en CUDA 5.0 i versions posteriors. CUDA 5.0 està suportat en els dispositius amb Compute Capability 3.5 o superior.

Quan un *grid* de *threads* llança un altre *grid* de *threads*, aquest són anomenats *grid* secundaris. Aquest *grid* secundari hereta certs atributs del *grid* principal, com la configuració de la memòria *caché*, la de la *shared memory* i la mida del *stack*. Per tant s'ha de tenir en compte que cada *thread* que es trobi amb un *kernel*, aquest serà executat. Per tant, si el *grid* principal té 128 *blocks* amb 64 *threads* cadascun, i no hi ha cap control de flux al voltant d'aquests llançaments secundaris, aleshores es llançaran un total de 8192 threads.

Els *grids* secundaris estant totalment niats dintre dels principals. Això significa que un *grid* secundari mai acabarà abans que el seu *grid* principal.

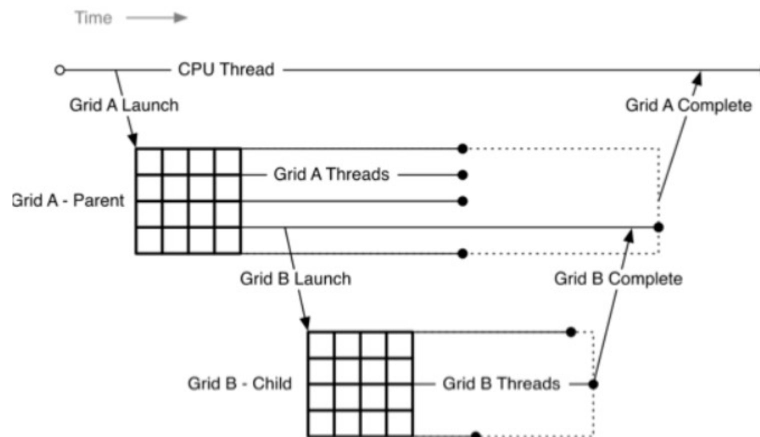


Figura 19. Anidament de grids sense sincronització

Ara bé, si el *kernel* pare, necessita els resultats del *kernel* fill, per a seguir realitzant el seu propi treball, aleshores necessita assegurar-se de què el *kernel* fill ha acabat. Per assegurar-se s'ha d'utilitzar una sincronització explicita. En aquest cas s'utilitza la funció *cudaDeviceSyncronize()*. Aquesta funció espera la finalització de tots els *grids* llançats prèviament des del *thread* des del qual s'ha cridat. La nidificació assegura que tots els *grids* fills d'aquest *thread*, i els *grids* dels *grids* fills, també hagin finalitzat.

Hem de tenir en compte que aquesta sincronització és només a nivell de *thread*, és a dir, el *thread* no esperarà al fet que els *grids* secundaris d'altres *threads* finalitzin. Si realment es necessita una sincronització de la finalització de tots *grids* secundaris de tots els *threads* d'un *block*, aleshores és necessari cridar a la funció *__syncthreads()* després de la crida a *cudaDeviceSyncronize()*.

Un altre aspecte clau al moment d'utilitzar el paral·lelisme dinàmic és el pas de punters com a paràmetres als *threads* fills. Aquí hi ha algunes restriccions. No es poden passar

punters que apuntin a la *shared memory* ni memòria local ni els registres, només es poden passar punters que estiguin apuntant a la memòria global o a la memòria constant.

6. Requisits del sistema

Pel que fa als requisits que es vol que tingui el nostre software final es poden dividir en funcionals i no funcionals.

Requisits funcionals:

- El sistema genera una solució al problema CMCP amb *k* cercles.
- L'usuari pot configurar una nova cerca de solució.
- L'usuari pot generar una nova cerca.
- L'usuari pot guardar resultats.
- L'usuari pot rotar la imatge.
- L'usuari pot fer zoom a la imatge.
- L'usuari pot moure la imatge.
- L'usuari pot guardar la imatge del resultat.

Requisits no funcionals:

- El sistema utilitza l'algorisme Simulated Annealing per cercar una solució.
- El sistema utilitza GPU per cercar una solució.
- L'usuari utilitza una interfície gràfica d'usuari per a l'execució del programa.

Pel que fa als requisits tècnics, només hi ha un i és que es necessita un ordinador amb una targeta gràfica compatible amb CUDA amb una *Compute Capability* de 3.5 o superior.

7. Estudis i decisions

En aquest capítol es detallaran totes aquelles llibreries i programari utilitzat per al desenvolupament del projecte. Per tal d'entendre algunes decisions primer es farà una petita descripció de les llicències que apareixeran durant la lectura d'aquest capítol.

7.1 Tipus de llicències

- **MIT:** També coneguda amb el nom de X11 és una llicència que implica l'obligatorietat d'adjuntar una còpia de la llicència amb el programari que es distribueixi. Per altra banda, concedeix a l'usuari final els drets d'utilitzar, copiar, modificar, publicar, distribuir o subllicenciar el software.

- **Zlib/libpng:** És una llicència de software lliure, simplement s'ha de tenir en compte que:
  - No s'ha d'indicar que s'és l'autor del programari original.
  - Versions del codi font alterades no han de ser representades com la versió original del programari.
  - L'avís de la llicència no ha de ser eliminat de les distribucions derivades.

- **Freeware:** Defineix un tipus de programari que es distribueix sense cap cost per al seu ús i per temps il·limitat. No es permet modificar i/o vendre.

- **GNU General Public License**: Permet a l'usuari final la llibertat d'ús, estudi, compartir i modificar.

7.2 Llibreries utilitzades

El present projecte tenia 3 grans necessitats: La programació en GPU, la visualització del resultat i una interfície gràfica d'usuari. Per les quals s'han utilitzat les llibreries de CUDA, OpenGL i QT respectivament, que es detallen a continuació:

7.2.1 CUDA

Figura 20. Logo de CUDA

CUDA és una arquitectura de càlcul paral·lel de Nvidia que aprofita la potència de càlcul de les targetes gràfiques per a realitzar càlculs científics de propòsit general. CUDA ofereix un conjunt d'eines (llibreries, compilador...) anomenat CUDA Toolkit, les quals ens permeten dissenyar codi que s'executarà en les GPU. Estan sota una llicència Freeware. Tot i que existeix una alternativa open-source anomenada OpenCL, aquesta és més jove i no té una comunitat tan gran com CUDA. A més que els directors del projecte ja estaven treballant amb aquestes eines, CUDA va ser pionera en aquest camp de la programació amb GPU i té bona documentació amb molts exemples en la seva web.

7.2.2 OpenGL

Figura 21. Logo de OpenGL

Per a la visualització de resultats s'ha utilitzat OpenGL. OpenGL és una especificació estàndard que defineix una API multillenguatge i multiplataforma per escriure aplicacions que produeixin gràfics 2D i 3D. L'alternativa més directa a OpenGL és DirectX, aquesta també és una API però és de Microsoft i té una gran dependència amb Windows. Els tres motius principals d'haver escollit OpenGL és el fet que OpenGL no té aquesta dependència amb el sistema operatiu Windows, també té una gran comunitat ja consolidada i en el grup de recerca ja havien treballat amb aquesta API.

Existeixen vàries llibreries que implementen l'API OpenGL. Glew (OpenGL Extension Wrangler Library) i Glfw (Graphics Library Framework) són les llibreries escollides que implementen OpenGL.

Glew és la llibreria Open Source multiplataforma que implementa les funcions especificades en l'API d'OpenGL. Per altra banda OpenGL no gestiona ni les finestres ni l'entrada de teclat, mouse, joysticks, etc. per tant existeixen algunes llibreries per a aquesta finalitat. Inicialment la llibreria GLUT era la llibreria que s'utilitzava per a aquesta finalitat, però aquesta es troba desfasada des de la seva versió 3.7, que data de l'agost de 1998. A més a més la llicència de GLUT és incompatible amb algunes distribucions de software, ja que GLUT té una llicència no lliure. La seva evolució natural ha estat la llibreria FreeGLUT, la qual utilitza la mateixa API que GLUT i per tant tots els exemples que es poden trobar per Internet amb GLUT també haurien de

funcionar amb FreeGlut. FreeGlut té una llicència MIT i a més també soluciona alguns inconvenients que hi havia a GLUT.

Glfw és una alternativa a FreeGlut. Aquesta compte amb una llicència Zlib, i és una llibreria que té la mateixa funcionalitat que freeGlut, és a dir per controlar inputs i gestionar finestres OpenGL. Però en Aquest cas Glfw ha estat dissenyada des de zero i tenint en compte algunes mancances que sembla que té FreeGlut. En el meu cas he escollit Glfw, perquè la documentació hem va semblar més clara i amb més exemples, ja que totes dues tenen una llicència lliure i la nostra aplicació no requereix de cap funcionalitat que no tinguin ni FreeGlut ni GLFW.

7.2.3 QT

Figura 22. Logo de QT

Per al desenvolupament de la interfície gràfica s'ha utilitzat QT. QT és un framework multiplataforma per a desenvolupar programes que utilitzin una interfície gràfica d'usuari. Està desenvolupada com a software lliure i de codi obert. Les seves llicències, el fet que sigui multiplataforma i la fàcil integració amb el llenguatge C++ i CUDA, el fan un candidat perfecte per a desenvolupar la interfície gràfica en aquest projecte.

7.2.4 Altres llibreries

De forma no tan rellevant s'han utilitzat un parell de llibreries amb la finalitat de tenir un sistema de logs en el nostre projecte i la llibreria FreeImage per a la conversió de finestra d'OpenGL a imatge:

- **SpeedLogger:** En el nostre projecte existia la necessitat de tenir un sistema de logs. SpeedLogger proporciona aquesta funcionalitat de manera fàcil i ràpida. Està desenvolupada sota una llicència MIT, es troba molt ben documentada en la seva pàgina i té un ús molt fàcil.

- **RapidJson**: S'han utilitzat per poder serialitzar i desserialitzar objectes a *string* en format JSON. És de les poques llibreries que existeix per serialitzar desserialitzar en C++ i té una llicència MIT també.

- **FreeImage:** Incorpora funcionalitats per treballar amb fitxers d'imatges i OpenGL. Està desenvolupada sota una llicència GNU General Public License.

7.3 Programari utilitzat

7.3.1 Visual Studio + Nsight + Qt VS Tools

Figura 23. Logo de Visual Studio, plugin Nsight, plugin QT VS Tools

Per al desenvolupament del projecte, l'IDE triat ha estat Visual Studio juntament amb el plugin NSIGHT per a Visual Studio que ofereix CUDA toolkit més el plugin QT VS Tools que ofereix QT.

CUDA es pot programar en diferents llenguatges d'alt nivell com Fortran, Python i C/C++. S'ha triat C/C++ perquè en el departament d'IMAE ja utilitzaven aquest llenguatge i és el més estès en la comunitat.

Pel que fa a l'IDE, des de la web oficial de CUDA la gran part de documentació que fa referència a la instal·lació de les eines necessàries per programar en CUDA està orientada a Visual Studio, encara que també ofereixen suport per al IDE Eclipse. S'ha triat Visual Studio, ja que amb els plugins de CUDA i de QT permet integrar molt bé les dues tecnologies.

7.3.2 Enterprise Architech

Figura 24. Logo de Enterprise Architect

Per als diagrames de classes i diagrames de seqüència s'ha utilitzat el programa Enterprise Architect. És una eina de disseny i modelatge visual basada en OMG UML. He utilitzat aquesta eina perquè personalment ja l'havia utilitzat anteriorment d'aquesta i és un programa molt complet, amb funcionalitats que no incorporen altres softwares lliures.

7.3.3 Draw.io

Figura 25. Logo de Draw.io

Per a la creació dels esquemes conceptuals presents en l'actual document, s'ha utilitzat draw.io. És un software online gratuït que permet dissenyar tota classe de diagrames. Segurament existeixin altres alternatives, però durant el meu recorregut en la universitat i en el món laboral he utilitzat draw.io. És molt fàcil d'utilitzar i mai he trobat a faltar cap funcionalitat.

7.3.4 Gannt project

Figura 26. Logo de Gantt project

Gannt Project és una aplicació gratuïta que permet fer diagrames de Gantt com el que es mostra en l'apartat 4 d'aquest document.

7.3.5 Stack ELK

Figura 27. Logo stack ELK

Per al monitoratge, anàlisi i visualització de resultats s'ha utilitzat el stack ELK. ELK és un conjunt d'eines de codi obert que permeten recollir, emmagatzemar i visualitzar dades en temps real.

És tracta d'un stack format per tres eines:

- *Elasticsearch*: És una base de dades distribuïda, i que permet fer lectures d'una forma molt ràpida.
- *Logstash*: Llegeix la informació dels logs generats i els envia a Elasticsearch.
- *Kibana*: És un servidor web que ens permet generar diferents visualitzacions de les dades que més interessen.

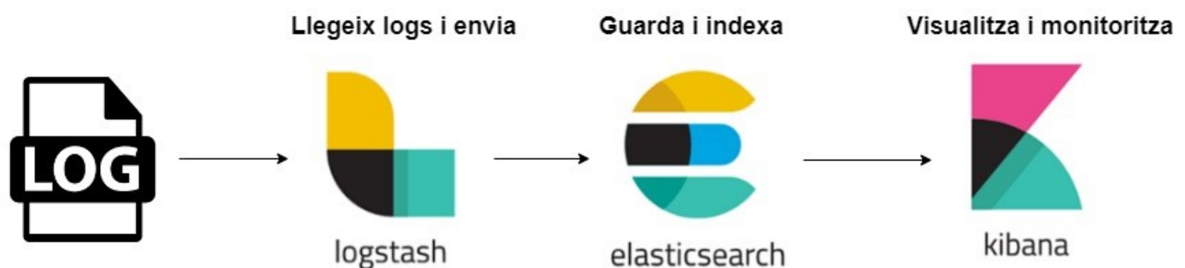


Figura 28. Diagrama de funcionament del stack ELK

Aquest conjunt d'eines l'utilitzem per analitzar ràpidament i de forma fàcil i visual tota aquella informació susceptible de ser important i tots aquells aspectes que ens puguin interessar de les dades generades pel nostre programa. Per exemple ho poden ser els temps d'execució dels diferents algorismes, temps d'escriptura a la memòria dels dispositius CUDA, radi de pertorbació, nombre de *threads*, nombre de cercles, etc.

Existeixen alternatives a ELK, com són GrayLog i Splunk. S'ha escollit ELK perquè és un software que ja coneixia, he treballat amb ell anteriorment i és gratuït. Les versions de Elasticsearch, kibana i logstash són la ultima versió estable a data d'avui, la versió 7.0.

7.3.6 GitLab

Figura 29. Logo GitLab

Per al manteniment i desenvolupament del codi s'ha utilitzat el repositori GitLab. He utilitzat Gitlab perquè és el que utilitzo normalment. La gran alternativa es GitHub. Les diferències principals de Gitlab enfront de GitHub és que GitLab ofereix repositoris privats gratuïts i una solució completa per a tot el cicle de desenvolupament. Tot i que GitLab té una comunitat més petita no té importància, ja que al cap i a la fi és un repositori.

7.3.7 Docker

Figura 30. Logo docker

Docker és una tecnologia Open Source la qual permet crear màquines virtuals molt lleugeres les quals estan dissenyades per a contenir serveis i/o aplicacions dintre seu. Els avantatges són que ofereix una capa d'abstracció entre el servei i el sistema operatiu. És a dir, un cop muntat es pot duplicar aquest servei a diferents màquines amb diferents sistemes operatius. També aïlla el servei dels recursos del sistema així que també afegeix una capa de seguretat. S'utilitza molt en el camp de IT. En el nostre cas ho hem utilitzat per a muntar els serveis de elasticsearch i kibana.

7.3.8 VirtualBox

Figura 31. Logo VirtualBox

VirtualBox és un software de virtualització el qual ens permet crear màquines virtuals i instal·lar-hi sistemes operatius. El seu gran competidor és VMWare. Utilizo VirtualBox perquè és el que utilitzo normalment i a nivell d'usuari copsa les meves necessitats. En aquest projecte s'ha utilitzat per a instal·lar una màquina Ubuntu la qual conté els serveis de elasticsearch i kibana muntats en docker.

8. Anàlisi i disseny del sistema

Els requeriments del nostre sistema han estat declarats en el capítol 6. En tractar-se d'un projecte de recerca els requeriments que necessiten especial atenció és el fet que s'ha de generar una solució al problema CMCP utilitzant l'algorisme Simulated Annealing i utilitzant la GPU al mateix temps. Així que es dedicarà un apartat a l'anàlisi del problema i al disseny conceptual de la solució.

Un altra part dels requeriments són els que tenen a veure amb l'usuari, el qual necessita fer un input i interpretar un resultat. Per tant és necessari establir una estandardització per el input i l'anàlisi i el disseny d'una interfície.

Un cop definit tot això es definirà el disseny del codi respecte a les necessitats del sistema. I durant la lectura d'aquest capítol veurem com apareix la necessitat d'analitzar els resultats i quina és la solució que es presenta.

8.1 Anàlisi del problema i disseny de la solució

El problema a tractar és el problema de cobertura màxima continua amb múltiples facilitats, en una zona de demanda distribuïda uniformement en un domini poligonal ***P*** que pot tenir forats. Considerem ***k*** recursos en forma de disc D_i amb centre ***c*** i radi r_i, un disc per a cada recurs. Els recursos poden ubicar-se en qualsevol lloc del pla i els recursos no estan necessàriament completament continguts dintre de ***P***. Anomenarem ***B*** al rectangle que conté ***P*** (Figura 32).

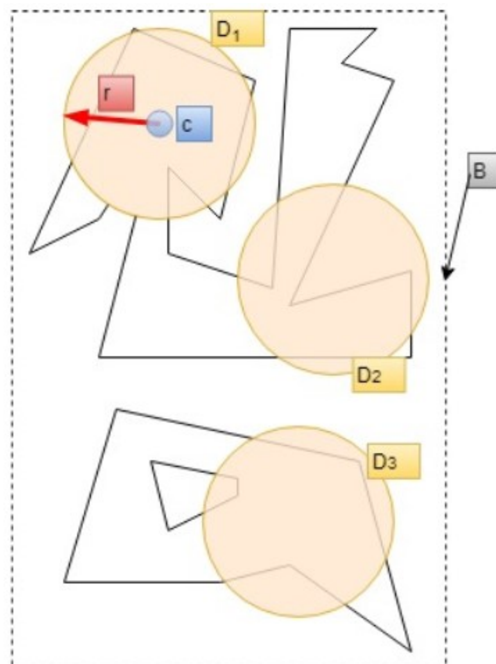


Figura 32. Exemple domini poligonal parcialment cobert per tres cercles.

Es tracta de trobar la millor ubicació dels k cercles per a obtenir la major cobertura possible. Per tant, per una banda és necessari calcular el sumatori de les àrees superposades entre cercle i polígon, i per altra banda cal tenir en compte la intersecció dels cercles entre ells.

La intersecció dels cercles presenta un problema, la intersecció de 3 o més cercles cobrint una mateixa area, com es mostra en la Figura 33.

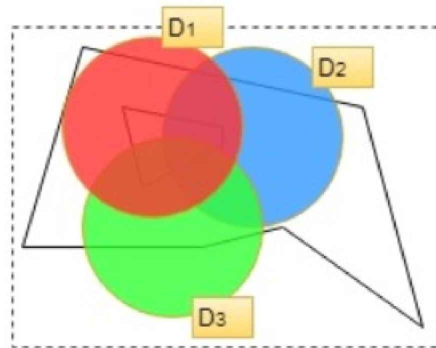


Figura 33. Intersecció de 3 cercles entre ells

Per al càlcul exacte de l'àrea del polígon coberta pels cercles que estan superposats, primer s'hauria de fer el càlcul de la subdivisió planar generada pels k cercles entre ells i després el càlcul de l'àrea d'intersecció entre polígon i cada regió de la subdivisió planar.

Aquest fet comporta un esforç computacional considerable i donat que per a trobar una solució al problema no ens interessa el fet que els cercles estiguin superposats entre ells, el que es planteja per tal de reduir el càlcul computacional i al mateix temps penalitzar la superposició entre els cercles, és calcular el sumatori de l'àrea d'intersecció entre les parelles de cercles:

$$\sum_{i,j=1;\, i<j}^{k} Area(D_i \cap D_j)$$

Per tant, considerem la següent funció objectiu que potencia la superposició entres el cercles i el polígon i penalitza la superposició entre els cercles.

$$F_o = \sum_{i=1}^{k} Area(D_i \cap P) - \sum_{i,j=1;\, i<j}^{k} Area(D_i \cap D_j)$$

Les tasques bàsiques que s'han de dur a terme en el procés de cercar una solució són les següents:

- Calcular els centres dels nous cercles.
- Calcular per cada cercle l'àrea d'intersecció amb el polígon.
- Calcular les interseccions dels *k* cercles entre ells.
- Calcular el valor de la funció objectiu.

Aquí hi ha diversos punts clau. Per una banda hi ha la generació dels nous cercles, per altra banda el càlcul de l'àrea d'intersecció entre cercle i polígon i per una altra banda dissenyar les estratègies de paral·lelització. Perquè ja sigui per allotjar els nous cercles, les seves funció objectiu, les millors solucions, etc. es necessita espai de memòria en el *device, i* en aquest punt es poden adoptar diferents estratègies per a la paral·lelització. En el present projecte s'han implementat tres estratègies diferents, on cada una utilitza una configuració de l'espai de memòria diferents segons les necessitats i condicionada a les exigències de l'arquitectura de les targetes.

Per tant, en els següents apartats d'aquest capítol s'explicarà en detall el càlcul dels nous cercles (8.1.1), com s'ha utilitzat la memòria de textura per dur a terme el càlcul de l'àrea d'intersecció entre cercle i polígon (8.1.2), i les diferents estratègies implementades (8.1.3).

8.1.1 Càlcul dels nous cercles

El càlcul dels nous centres dels cercles bé determinat per tres factors: El centre del cercle actual, el radi de pertorbació i l'aleatorietat.

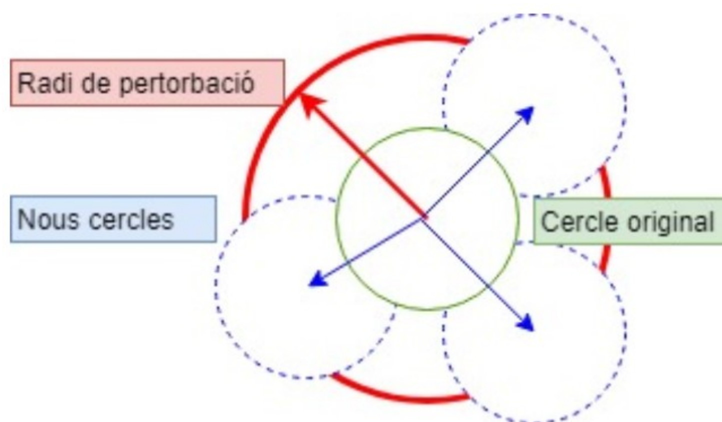


Figura 34. Creació de nous cercles

El radi de pertorbació determina els límits geomètrics per a la generació de nous punts aleatoris. És a dir, el que es fa és calcular un nou punt aleatori, el qual es trobarà dins de la circumferència delimitada pel centre del cercle original i el radi de pertorbació. Aquest radi de pertorbació és un paràmetre molt important dins de l'algorisme, ja que

determina com de llunyans poden arribar a ser els nous centres dels cercles. Pot haver-hi casos en el que aquest radi ens interessi que sigui petit, com per exemple quan l'algorisme està a punt d'apropar-se a un màxim o por el contrari pot ser que ens interessi que el radi sigui gran quan els cercles estan molts junts o queda molta àrea per cobrir. No és trivial decidir un valor per a aquest paràmetre, ja que també es poden donar casos mixtos, és a dir, que els cercles estiguin molt junts però que ja estiguin cobrint tota l'àrea del polígon. Per tant és un paràmetre que requereix estudi i fer diferents proves amb diferents polígons i diferents nombres de cercles i intentar trobar una relació entre l'àrea total a cobrir, el nombre de cercles i l'aproximació del valor objectiu a un màxim. L'estratègia que s'ha adoptat en aquest projecte és començar amb un radi de pertorbació gran i a mesura que l'algorisme va avançant i va aconseguint millors funcions objectiu, aquest radi es va reduint.

8.1.2 Ús de la memòria de textura per al càlcul de les àrees.

Per tal de saber quina és l'àrea d'intersecció d'un nou punt generat per l'algorisme Simmulated Annealing, el que s'ha fet és aprofitar el codi de (Coll, Fort, & Sellarès, 2019), on el resultat del qual és un *grid* on cada punt conté el valor de l'àrea d'intersecció entre un cercle i el polígon. La Figura 35 mostra un exemple d'aquest *grid*. Com més fosc són els punts més gran és el valor de l'àrea d'intersecció calculada.

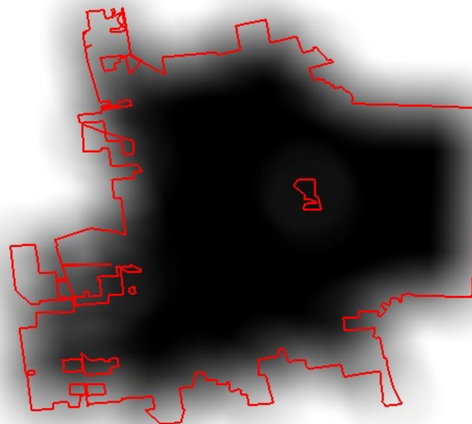


Figura 35. Grid amb àrees precalculades

En aquest punt del projecte es van plantejar dues opcions. Una opció era que dins d'un únic kernel, cada vegada que l'algorisme generés un nou cercle es calculés la seva àrea d'intersecció amb el polígon fent crides al codi de (Coll, Fort, & Sellarès, 2019). L'altra opció era aprofitar d'alguna manera aquest grid ja calculat i poder fer-li consultes.

La primera opció presentava diversos inconvenients. Un dels més importants és la quantitat de càlculs doncs s'hauria de calcular l'àrea d'intersecció per cada nou cercle

generat. Per tant, el nombre de càlculs seria (Nombre de cercles) * (Nombre de *threads*) * (Nombre d'iteracions de l'algorisme). Un altre inconvenient és que es podria donar el cas què és calculés el mateix cercle diverses vegades, o que es calculessin cercles molt pròxims. Això significaria fer la mateixa feina moltes vegades. En canvi la segona opció és fer els càlculs de les àrees per cada cercle centrar en un punt del *grid*. Així el nombre de càlculs està limitat al nombre de punts del *grid*.

En la segona opció es plantejava aprofitar les característiques que ofereix l'espai de memòria de les textura per a guardar el *grid* amb el valor de les àrees ja calculades i fer consultes a aquesta memòria. Aquesta era una opció millor, ja que representava molts menys càlculs i el marge d'error al moment de fer interpolació és molt petit. Així que aquest *grid* d'àrees precalculades s'ha tractat com a una textura, creant una matriu bidimensional, on cada element d'aquesta matriu conté una àrea precalculada del nostre *grid*. En aquesta textura s'han configurat els atributs *Linear filter mode*, i *texture addressing border mode*. D'aquesta manera cada vegada que el nostre algorisme generi un nou punt pseudoaleatori on col·locar un nou cercle, només ha de preguntar a la textura quin és el valor d'àrea que té el cercle en aquest punt.

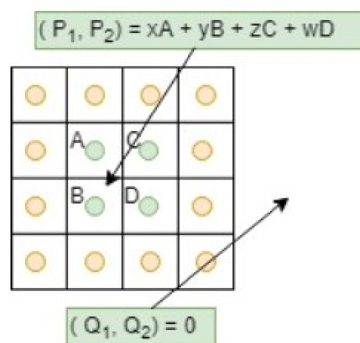


Figura 36. Exemple de fetch's a la nostra textura

Com mostra la Figura 36, si aquest nou punt es troba fora de la textura aleshores retornarà zero, és a dir l'àrea serà zero, altrament farà la interpolació corresponent de les àrees, retornant d'aquesta manera un valor molt aproximat de l'àrea real.

8.1.3 Estratègies de paral·lelització

8.1.3.1 *Mètode 1*

En aquest primer mètode la distribució de la memòria és com la que es mostra en la Figura 37. On en la memòria global es reserven 5 arrays:

- *millorsConfiguracions*: Array per a que cada thread pugui guardar les coordenades d'aquells cercles que han aconseguit la millor funció objectiu. Per cada cercle es guarden la seva coordenada x i y respectivament, que són del tipus float. Per tant la mida d'aquest array és (n threads) * (*k* cercles * 2) * (sizeof(float)).
- *millorFuncioObjectiu*: Array per a que cada thread pugui guardar el valor de la millor funció objectiu, és del tipus float i la seva mida és (n threads) * (sizeof(float)).
- *configuracionsActuals*: Array igual que *millorsConfiguracions* però amb el propòsit de guardar les coordenades dels cercles actuals.
- *funcioObjectiu*: Igual que *millorFuncioObjectiu* però amb la finalitat de guardar la funció objectiu actual.
- *posicioInicial*: Array amb la posició inicial dels cercles.

I per últim, en els registres individuals de cada thread es declara un array amb la finalitat de guardar els *k* nous cercles generats i que seran avaluats.

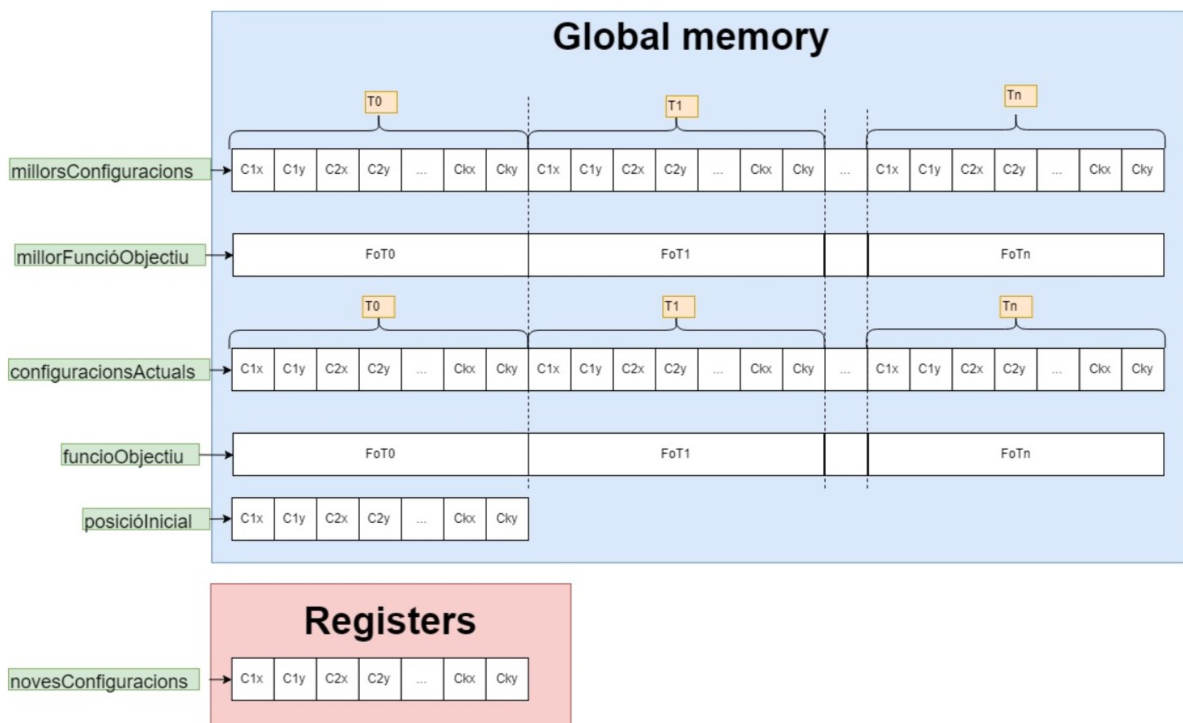


Figura 37. Distribució de la memòria mètode 1

El fil d'execució és com el que es mostra a la Figura 38. Cada *thread* és independent de la resta. Cada vegada que es genera una nova configuració de cercles, aquesta s'avalua i si és la millor solució fins al moment, aleshores es guardarà en els arrays *millorsConfiguracions* i *configuracionsActuals*, en cas contrari pot ser acceptada amb una certa probabilitat i només serà guardada a *configuracionsActuals*. Aquest procés serà repetit X vegades.

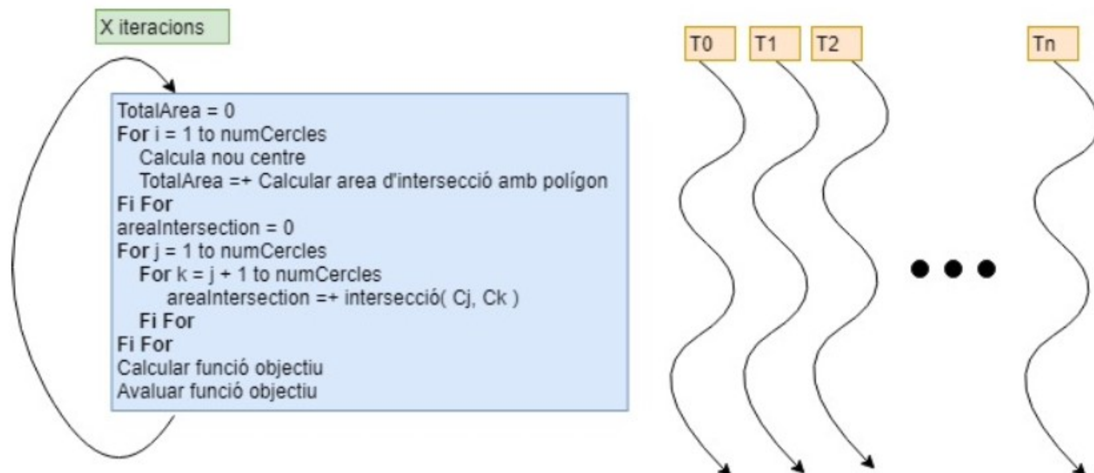


Figura 38. Fil d'execució mètode 1

Al final de l'execució d'aquest *kernel* obtenim un array amb totes les millors solucions de cada thread, per tant des de la CPU s'ha de fer una cerca de quina ha estat la millor funció objectiu. La longitud d'aquest array serà de n threads.

En aquesta primera estratègia, el que més penalitza és que s'utilitza bàsicament memòria global on és fan moltes escriptures i lectures, sent la memòria global la més lenta de totes. Un altre fet que pot comportar penalització, en el cas que hi hagués molts cercles, són el FOR i el doble FOR que s'utilitzen per pertorbar els cercles i calcular les interseccions entre ells. Per contra cap *thread* no ha d'accedir a la mateixa posició de memòria ni existeix cap sincronització entre els *threads*. Un altre aspecte atenir en compte és la quantitat de memòria que es reserva per cada *thread*, això limita molt el nombre de *threads* que podem llançar, ja que cada *thread,* només per les matrius *millorsConfiguracions* i *configuracionsActuals* reserva espai per (*k* circles · 2) ·2.

8.1.3.2 *Mètode 2*

En el segon mètode la distribució de la memòria és com la que es mostra en la Figura 39. Aquesta vegada s'utilitzaran els registres individuals de cada thread per a declarar tres arrays, on es guardarà la millor configuració obtinguda per cada thread, un altre array amb la configuració de cercles actuals i un altre per emmagatzemar els cercles nous que es generin a cada iteració. En la memòria global només es guarda l'array de *posicioInicial*, en la qual, al finalitzar l'execució del *kernel* s'utilitzarà per allotjar la millor configuració de cercles global. També es reserva espai per guardar el valor de la millor funció objectiu en i quin ha estat el thread que ho ha aconseguit.

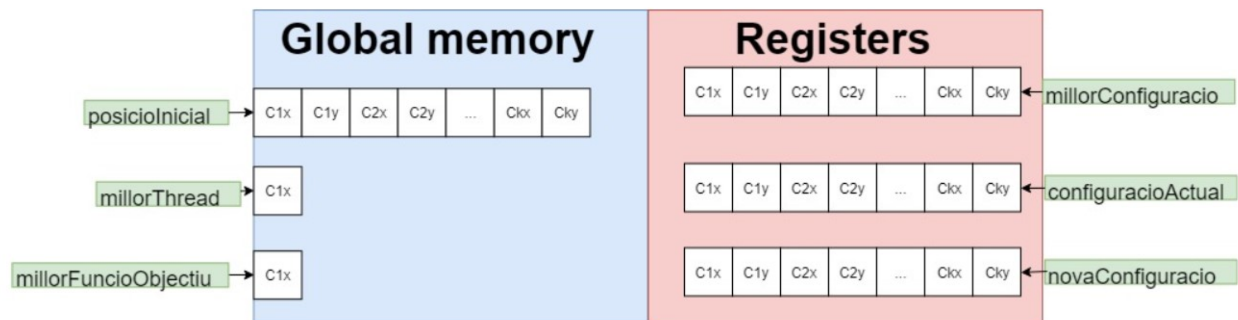


Figura 39. Distribució de la memòria mètode 2

Aquesta configuració de memòria implica que al final de cada *thread*, s'han de comparar tots els *threads* i saber quin ha estat el millor, i només aquest escriurà la solució a la memòria global per poder accedir des del host. Això s'ha implementat tal com es mostra en la Figura 40. Un cop acabades totes les iteracions, cada *thread* contindrà dintre seu la seva millor solució aconseguida, aleshores s'utilitza un AtomicMax per guardar el millor valor de funció objetiu a *millorFuncioObjectiu*. Es fa una sincronització i seguidament cada *thread* llegeix d'aquesta posició de memòria per saber si ha sigut ell el que l'ha guardat. En cas afirmatiu guardarà el seu id en *millorThread* i es torna a sincronitzar. De nou, cada *thread* mira si ell mateix és el millor *thread* llegint a *millorThread*, en cas afirmatiu guardarà a *posicioInicial* la millor configuració.

Es podria donar el cas que hi hagi *threads* que tenen el mateix valor de funció objectiu màxim, però en aquest cas només ens interessa una solució, que serà la que hi haurà a *posicilnicial*.

En aquesta estratègia es redueix molt la reserva d'espai a memòria global i les lectures i escriptures a aquesta. El preu a pagar és un coll d'ampolla com és l' AtomicMax, en el qual tots els *threads* han de llegir, i potser escriure, de manera atòmica. A part també de les dues sincronitzacions i el FOR i doble FOR que penalitzaran en el cas de tenir molts de cercles.

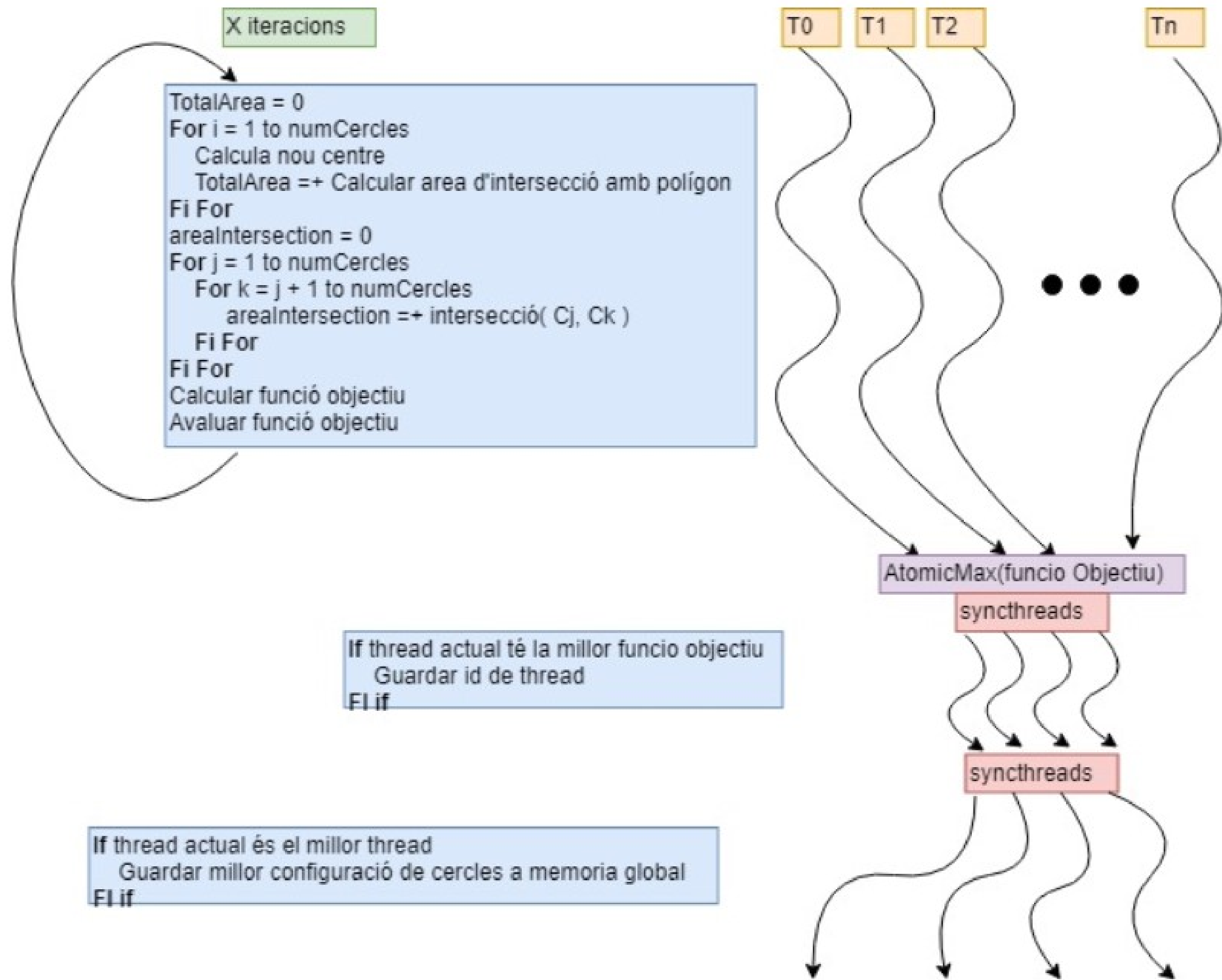


Figura 40. Fil d'execució mètode 2

8.1.3.3 *Mètode 3*

En aquest mètode s'ha fet ús del paral·lelisme dinàmic per tal de calcular la pertorbació dels cercles i les interseccions entre ells, amb l'objectiu d'eliminar el FOR i el doble FOR. Utilitzar paral·lelisme dinàmic té les seves conseqüències. La primera és que haurem de fer més ús de la memòria global que en el mètode 2, ja que necessitarem passar punters a paràmetres als *kernel* fills per tal de recollir resultats des del *kernel* pare, i com ja s'ha mencionat anteriorment en el capítol 5.2.3.6 aquests paràmetres només poden estar allotjats a la memòria constant o global. Per tant, amb diferència del mètode 2, a la memòria global s'ha afegit un array per a guardar la suma d'interseccions entre els cercles, i l'array novesConfiguracions ha passat dels registres a memòria global.

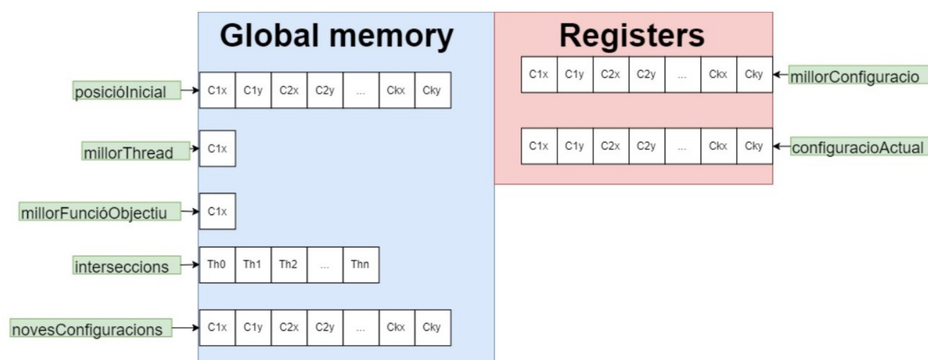


Figura 41. Distribució de la memòria mètode 3

Com ja s'ha esmentat anteriorment en el capítol 5.2.3.6, els *kernels* fills hereten dels seus pares certs atributs com la configuració de la memòria i la mida de la pila. També cal tenir en compte la limitació de *threads* per *block*. Per exemple, si es llancen 256 *threads* per *block* i tenim 5 cercles, cada *thread* pare llançaria 5 threads fills, i tindríem un total de 256 x 5 = 1280 *threads*. En tal cas ja s'estaria sobrepassant els 1024 *threads* per *block* que ofereixen la gran majoria de targetes gràfiques. Per tal d'evitar això el que es fa en aquest mètode, tenint en compte que la majoria de targetes gràfiques suporten 65535 *blocks*, és executar un sol *thread* per *block*.

Però el fet d'executar un sol *thread* per *block* comporta un altre problema, i és la sincronització entre *threads* de diferents *blocks*, ja que no totes les targetes gràfiques de CUDA incorporen aquesta funcionalitat. Per tal de resoldre aquest problema s'ha dividit el mètode en dos *kernels*, ja que un segon *kernel* no començarà mai fins que finalitzi el primer.

Els fils d'execució del primer i segon *kernel* estan representats en la Figura 42 i Figura 43 respectivament. En el *kernel* principal es crea un *thread* per cada cercle, on es calculen la seves noves coordenades, es fa una sincronització per esperar al fet que tots els cercles nous s'hagin generat, i aleshores cada cercle calcula les seves

interseccions amb la resta de cercles i es guarda el resultat a l'array *interseccions*. Un cop s'hagin fet les X iteracions cada *thread* principal guarda la seva millor configuració en el array *novesConfiguracions*, s'aprofita l'array *interseccions* per guardar la millor funció objectiu de cada *thread* i es fa un atomicMax de la millor funció objectiu de cada thread. En el segon *kernel* cada *thread* compara la seu valor de funció objectiu màxim allotjat a la posició corresponent de l'array interseccions amb el valor de *millorFuncioObjectiu*. En cas que coincideixin guardarà el seu id en millorThread i es torna a sincronitzar. De nou, cada *thread* mira si ell mateix és el millor *thread* llegint a *millorThread*, i en cas afirmatiu es guarda a *posicioInicial* la millor configuració.

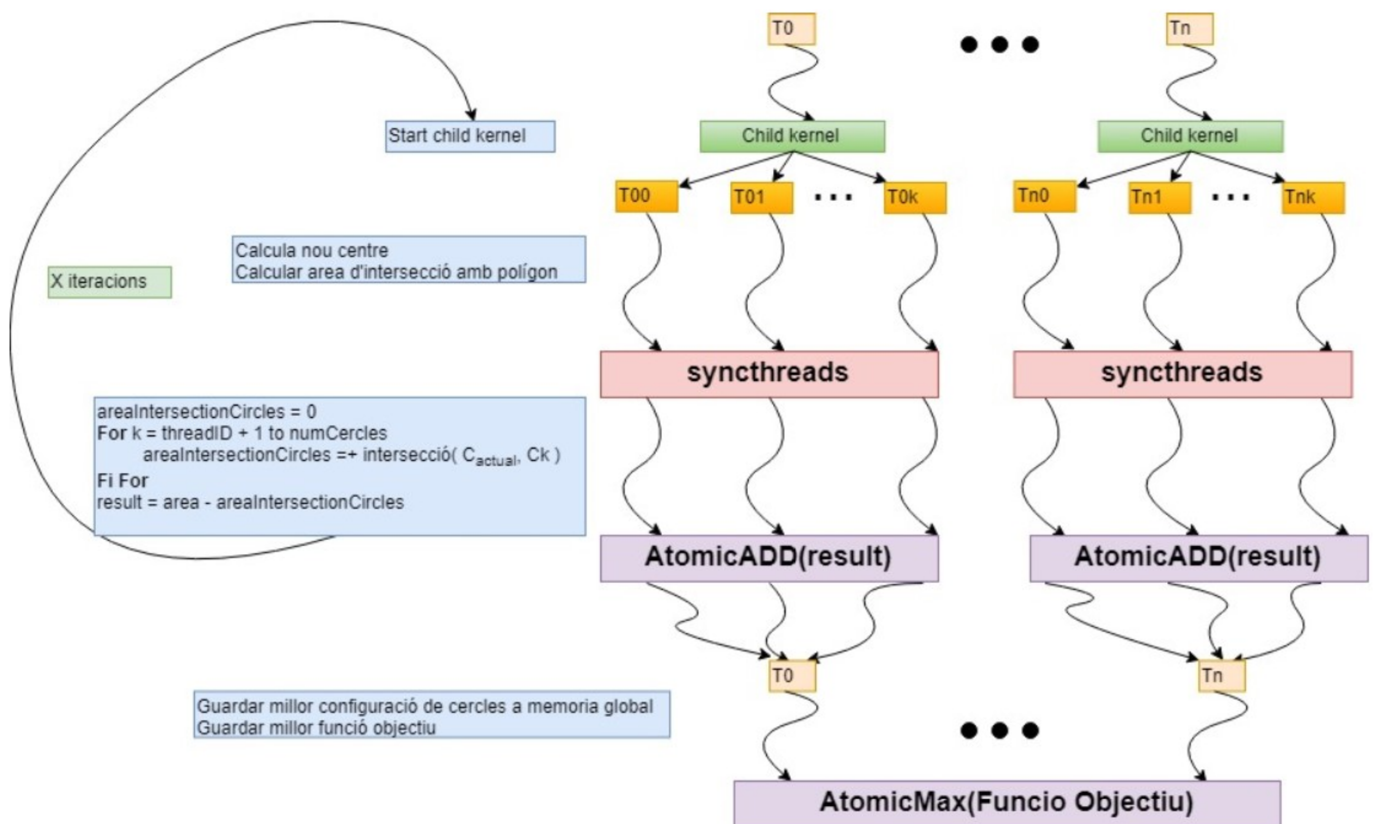


Figura 42. Fil d'execució mètode 3, kernel 1

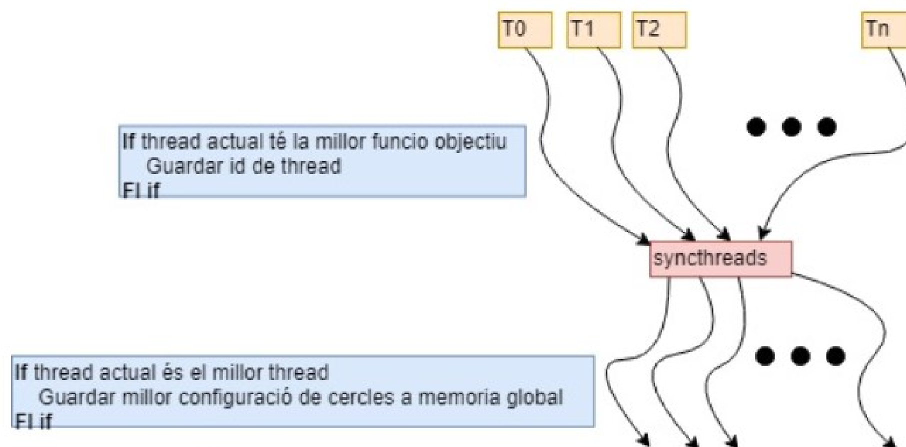


Figura 43. Fil d'execució mètode 3, kernel 2

8.1.3.4 *Estratègia d'execució dels kernels*

En aquest apartat es parlarà de l'estratègia d'execució dels *kernels*, ja que en aquest aspecte també s'ha plantejat una estratègia que només implementen el segon i tercer mètode.

Un primer punt a tenir en compte és la posició inicial dels cercles. En aquest cas la posició inicial de tots els cercles és la mateixa, i es tracta d'un punt qualsevol del *grid* el qual la seva àrea és màxima.

A partir d'aquí es fa una primera crida al *kernel* amb un radi de pertorbació alt i poques iteracions de l'algorisme Simulated Annealing, per tal de repartir els cercles pel mapa i quedar-nos amb la millor solució. La segona crida al *kernel* ja parteix d'aquesta millor solució i tindrà un radi de pertorbació més petit i amb més iteracions de l'algorisme.

Quan parlem de radi de pertorbació alt fem referència a un rang aproximat entre [0.3, 0.4] i radi de pertorbació baix a un rang [0.08, 0.15] aproximadament. El nombre d'iteracions que ofereix millors resultats són 45 pel primer *kernel* i uns [400, 450] pel segon. Ja que aquests valors depenen de la morfologia del mapa, el nombre de cercles i el seu radi, s'han deixat com a paràmetres d'entrada.

Utilitzant aquesta estratègia de fer una primera crida que reparteixi els cercles amb un cert criteri aconseguim que en l'inici del segon *kernel* tots els *threads* ja parteixin d'una posició relativament bona i sigui més fàcil aconseguir solucions òptimes. Tots aquells *threads* que no tenien bones solucions queden descartats i en el segon *kernel* tots parteixen d'una solució millor.

En la Figura 44 és mostra un diagrama d'aquesta estratègia juntament amb un exemple gràfic.

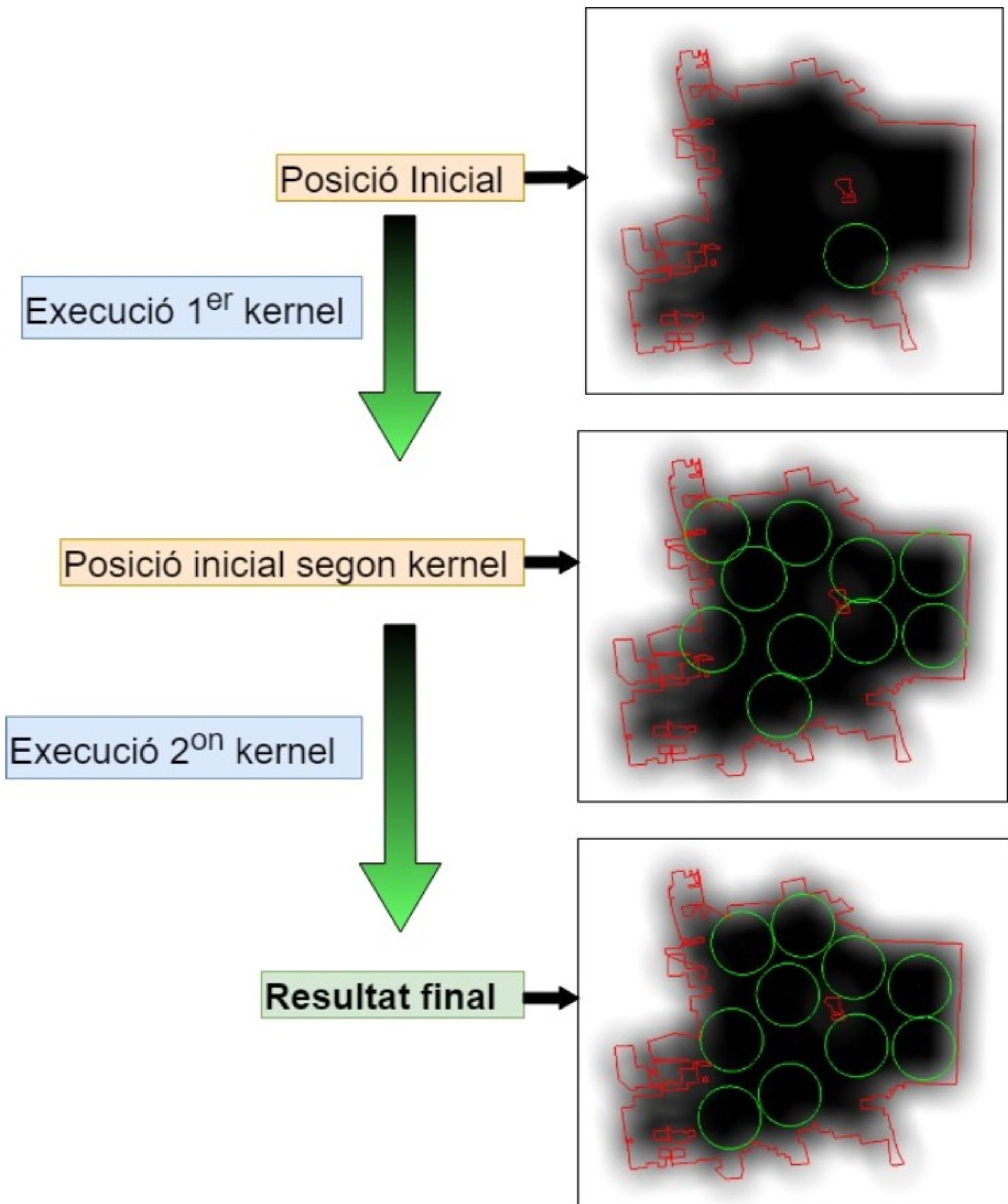


Figura 44. Estratègia d'execució dels kernels

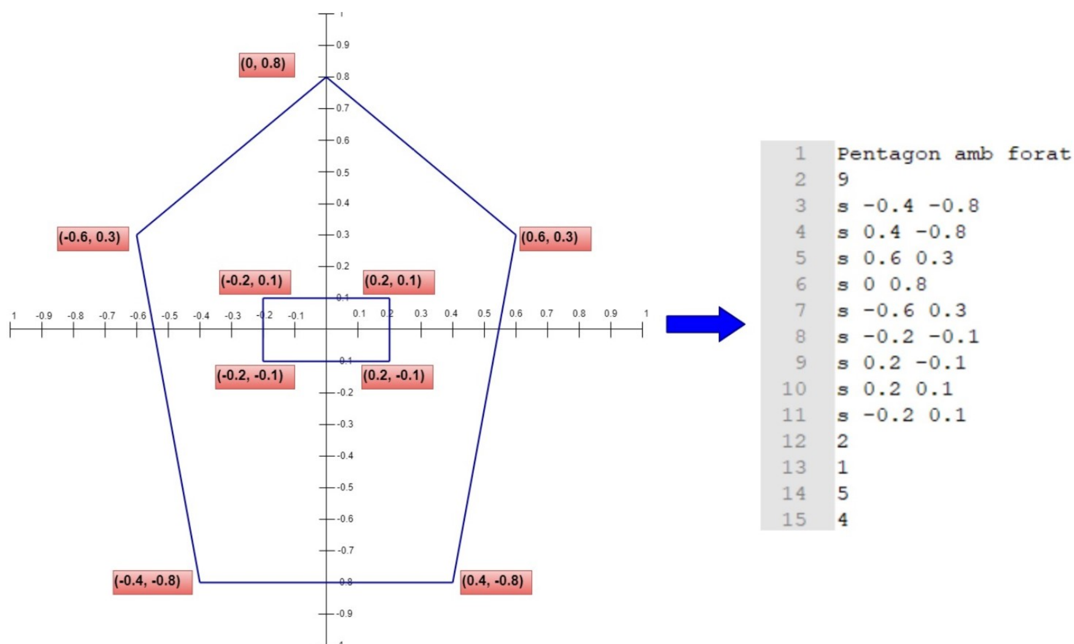
8.2 Fitxer d'entrada

El fitxer d'entrada serà el que definirà el nostre polígon. El polígon ha d'estar delimitat per segments de línia recta. Els punts del polígon hauran d'estar declarats en ordre antihorari i amb una 's' de "segment" davant de cada un. El sistema de coordenades dins del programa té un rang de [-1.0, 1.0] tant en l'eix de les x com en l'eix de les y. Per tant els punts han d'estar dins d'aquest rang.

Els polígons poden ser polígons no connexos, i aquells polígons que es trobin dins d'altres polígons seran considerats com a *holes* (forats), és a dir seran àrees les quals no ens interessa cobrir. El format del fitxer haurà de ser el següent:

- Descripció del fitxer
- Nombre total de vèrtexs
- Llistat de tots els vèrtexs
- Nombre total de polígons
- Nombre de polígons forats
- Nombre de vèrtexs que defineix cada polígon

En la Figura 45 es mostra un exemple d'un pentàgon amb un quadrat a dins i com hauria de ser el contingut del seu respectiu fitxer d'entrada.




```
1  Pentagon amb forat
2  9
3  s -0.4 -0.8
4  s 0.4 -0.8
5  s 0.6 0.3
6  s 0 0.8
7  s -0.6 0.3
8  s -0.2 -0.1
9  s 0.2 -0.1
10 s 0.2 0.1
11 s -0.2 0.1
12 2
13 1
14 5
15 4
```

Figura 45. Exemple de polígon i el seu respectiu fitxer d'entrada

8.3 Anàlisi i disseny de la interfície d'usuari

La interfície d'usuari pretén ser una eina que faciliti l'execució del nostre programa i la interpretació dels resultats, i per altre banda que ens permeti guardar els resultats obtinguts d'una manera fàcil. Els requeriments de la interfície són els que es mostren en la Figura 46, que bàsicament consta de dues accions principals:

- Configurar un *input*
- Veure i guardar resultats

On el input són un conjunt de paràmetres que defineixen el procés. Aquests paràmetres són:

- Un fitxer d'entrada
- Nombre de cercles
- Radi dels cercles
- Nombre de *threads*
- Escollir mètode del procés
- Escollir dispositiu
- Radi de pertorbació dels *kernels*
- Nombre de iteracions dels *kernels*

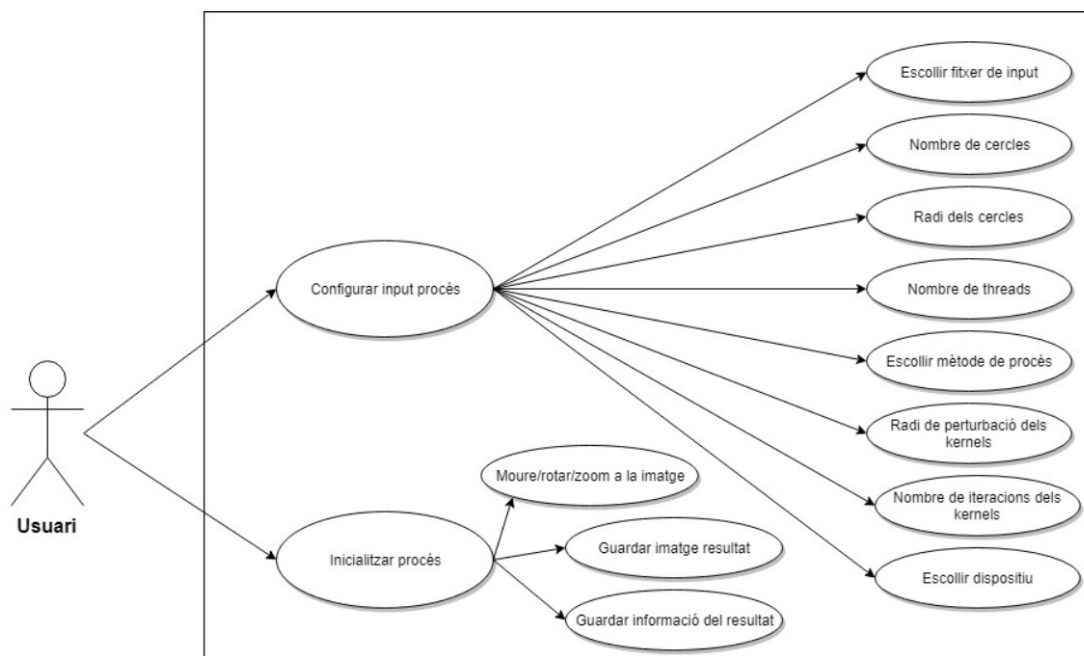


Figura 46. Diagrama cas d'ús d'usuari

Fitxa de cas d'ús "Configurar input":

Nom	Configurar input
Autor	Moisès Saus
Data	12/05/2019
Descripció	L'usuari configura els paràmetres del procés
Actors	Usuari
Precondicions	-
Postcondicions	-
Escenari principal	1. L'usuari interactua amb elements de configuració de la interfície.
Escenari alternatiu	-

Fitxa de cas d'ús "Inicialitzar procés"

Nom	Inicialitzar procés
Autor	Moisès Saus
Data	12/05/2019
Descripció	L'usuari inicialitza el procés i guarda resultats
Actors	Usuari
Precondicions	S'ha configurat un input
Postcondicions	-
Escenari principal	1. L'usuari inicialitza el procés. 2. L'usuari pot guardar imatge i dades del resultat.
Escenari alternatiu	-

I el disseny inicial de la interfície és com el que es mostra en la Figura 47.

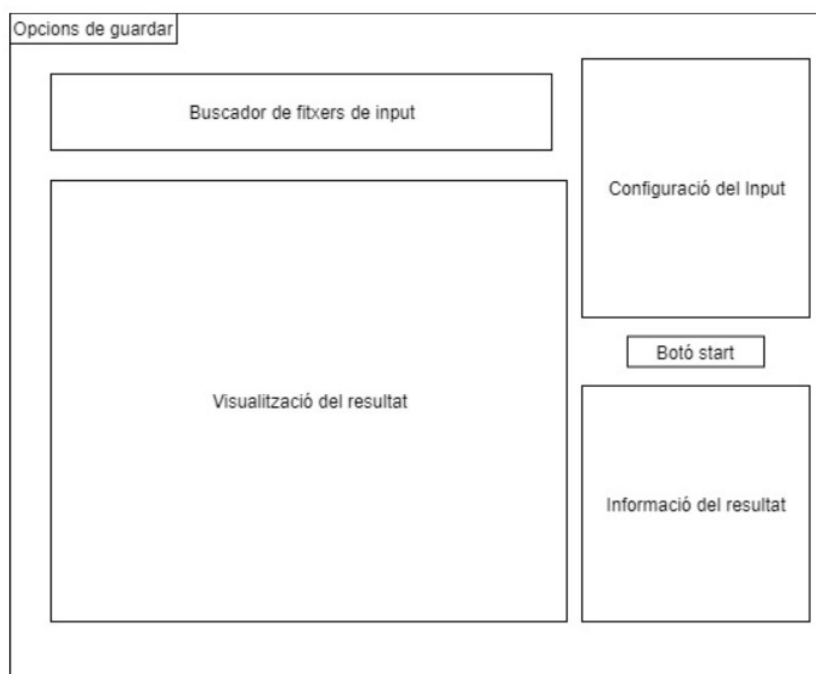


Figura 47. Disseny interfície d'usuari

8.4 Anàlisi i disseny del codi

8.4.1 Immersió al codi de (Coll, Fort, & Sellarès, 2019)

Abans de començar a dissenyar el codi, hi ha hagut un treball previ que ha consistit a fer una immersió en el codi de (Coll, Fort, & Sellarès, 2019) per tal de saber com funcionava aquest i com estava organitzat. El codi de (Coll, Fort, & Sellarès, 2019) resol un problema més general del que es tracta en aquest projecte, per tant s'havien d'identificar aquelles funcionalitats que eren necessàries per a aquest projecte, plantejar un disseny en concordança a les necessitats d'aquest projecte i fer-lo funcionar. En aquest apartat només es farà una breu descripció de com s'ha organitzat el codi i en el capítol 8.4.2 es farà una explicació més detallada i gradual de tot el codi del projecte per a una millor comprensió.

D'aquesta immersió i reestructuració de codi han sorgit les classes que es mostren en el Diagrama 1. Totes aquelles parts de codi que tenien a veure amb la graella de punts a ser calculats i la seva generació s'han encapsulat a la classe GridGenerator. Tota la part de càlculs d'àrees per cada punt de *grid* s'han encapsulat a AreaCalculator i les seves classes filles AreaCalculatorCPU i AreaCalculatorGPU. I totes aquelles funcions de transferència de dades entre host i *device* s'han encapsulat a la classe *Device*.

La classe ManagePolygon conté tota la informació referent als polígons: nombre de vèrtexs, nombre de polígons,etc. I la classe PolygonReader es la que s'encarrega de llegir els polígons des de un fitxer.

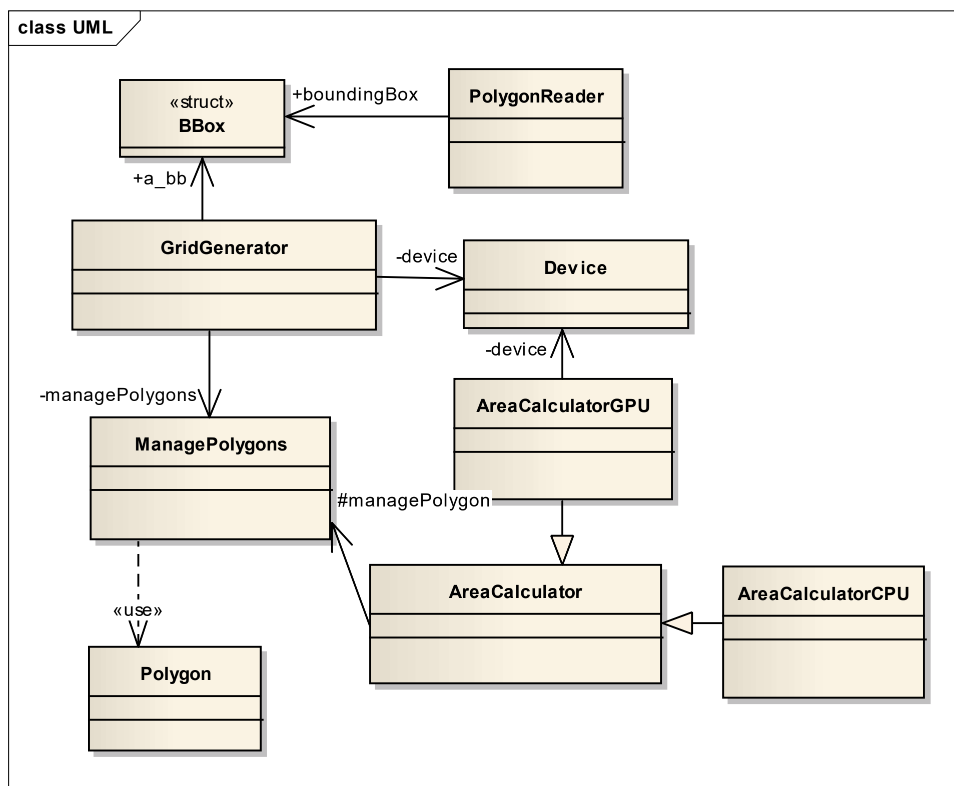


Diagrama 1. Classes generades a partir de l'estudi del codi de (Coll, Fort, & Sellarès, 2019)

8.4.2 Diagrama de classes

Donada la magnitud del diagrama de classes s'anirà comentant les diferents classes i les diferents relacions entre elles separadament. Començarem per les estructures més bàsiques i anirem ampliant, destacant aquells aspectes més rellevants per tal d'entendre el disseny del codi.

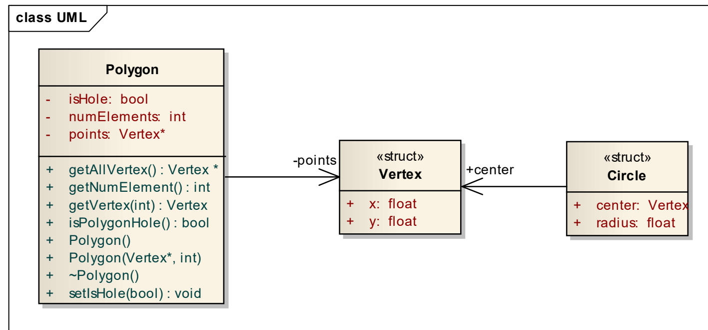


Diagrama 2. Estructures bàsiques

En el Diagrama 2 estan representades les classes i structs més bàsiques, que són Vertex, Circle i Polygon. Un Polygon té un conjunt de Vertexs i un Circle té un Vertex que representa el seu centre i també té un radi del tipus float.

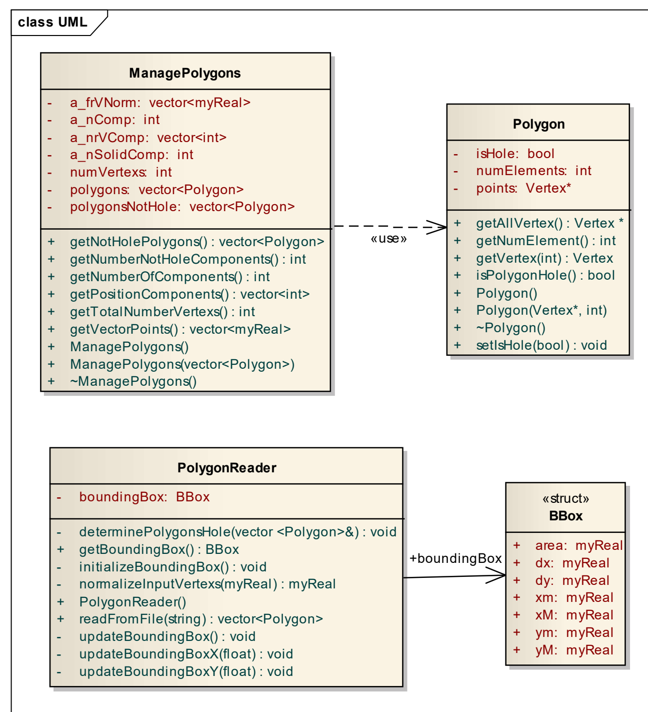


Diagrama 3. Classe ManagePolygons, PolygonReader i struct BBox

La classe PolygonReader és la que s'encarrega de llegir els polígons d'un fitxer de text, determinar aquells polígons que estan dins d'un altre i de generar el BBox. BBox és un

struct que representa un bounding box, on es guarda la informació que delimita les coordenades de la regió on estan continguts els polígons. És a dir, fora del bounding box no hi ha cap polígon. El tipus MyReal és un tipus definit com a float. Es va definir perquè existia la necessitat de fer diferents proves de marge d'error entre els tipus double i float.

ManagePolygons conté tota la informació relacionada amb els polígons. Nombre de polígons que tenim, nombre total de vèrtexs, vector de polígons, vector de vèrtexs...

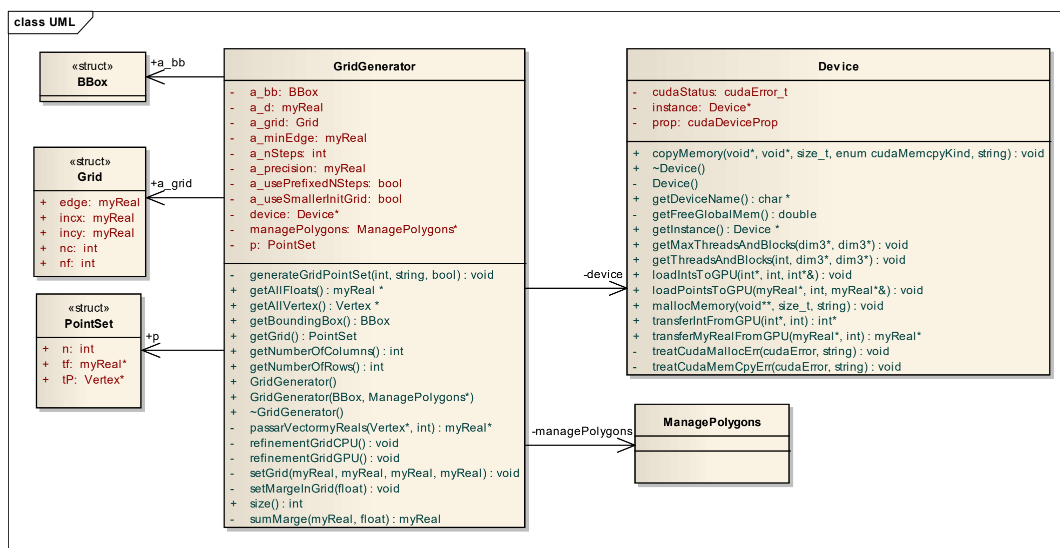


Diagrama 4. Generador de grid.

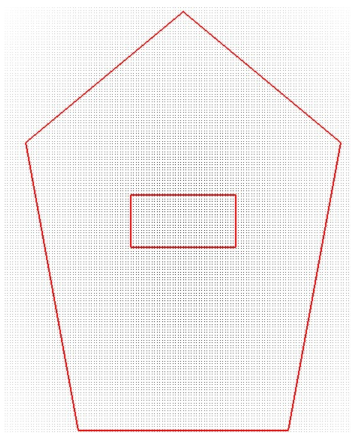


Figura 48. Polígon amb graella de punts

GridGenerator és la classe encarregada de generar i emmagatzemar la informació relacionada amb el que anomenem *grid*. Conceptualment, el *grid* representa una graella de punts, com la que es mostra en la *Figura 48*. On es veu un polígon i molts punts. El *grid* està contingut dins del *bounding box* i per cada punt d'aquesta graella és col·locarà el centre d'un cercle i es calcularà l'àrea d'intersecció entre el cercle i el polígon. El struct PointSet ens defineix un conjunt de vèrtexs, on aquests són guardats com vector de Vertex i vector de myReal.

La classe Device és una classe singleton, aquesta és instanciada a l'inici del programa i és l'encarregada de gestionar la comunicació amb el *device*: reservar memòria en el *device*, copia dades de *device* a *host* o a la inversa, consultar *threads* i *blocks* disponibles, etc. La classe GridGenerator utilitza la classe Device perquè té dos mètodes per refinar el *grid*: refinementGridCPU() i refinementGridGPU(). Refinar el *grid* significa eliminar tots aquells punts del *grid* que no es troben dins del polígon, un d'aquests dos mètodes, refinementGridGPU(), utilitza

un *kernel* per fer aquest refinament, l'altre utilitza la CPU. A l'última versió del programa no es fa un refinament del *grid*, perquè hi ha casos en els quals interessa situar cercles fora del polígon.

En el Diagrama 5 està representat el diagrama de classes d'AreaCalculator. AreaCalculator és una classe abstracta i les dues classes que hereten d'ella són AreaCalculatorCPU i AreaCalculatorGPU, que són les encarregades de calcular l'àrea d'intersecció entre un cercle i el polígon per cada punt del *grid*. ÀreaCalculatorCPU utilitza la CPU per fer els càlculs i ÀreaCalculatorGPU utilitza la GPU. La classe Candidates conté tots aquells punts que volem que siguin calculats, en el seu constructor pot rebre un GridGenerator o un vector de Circle.

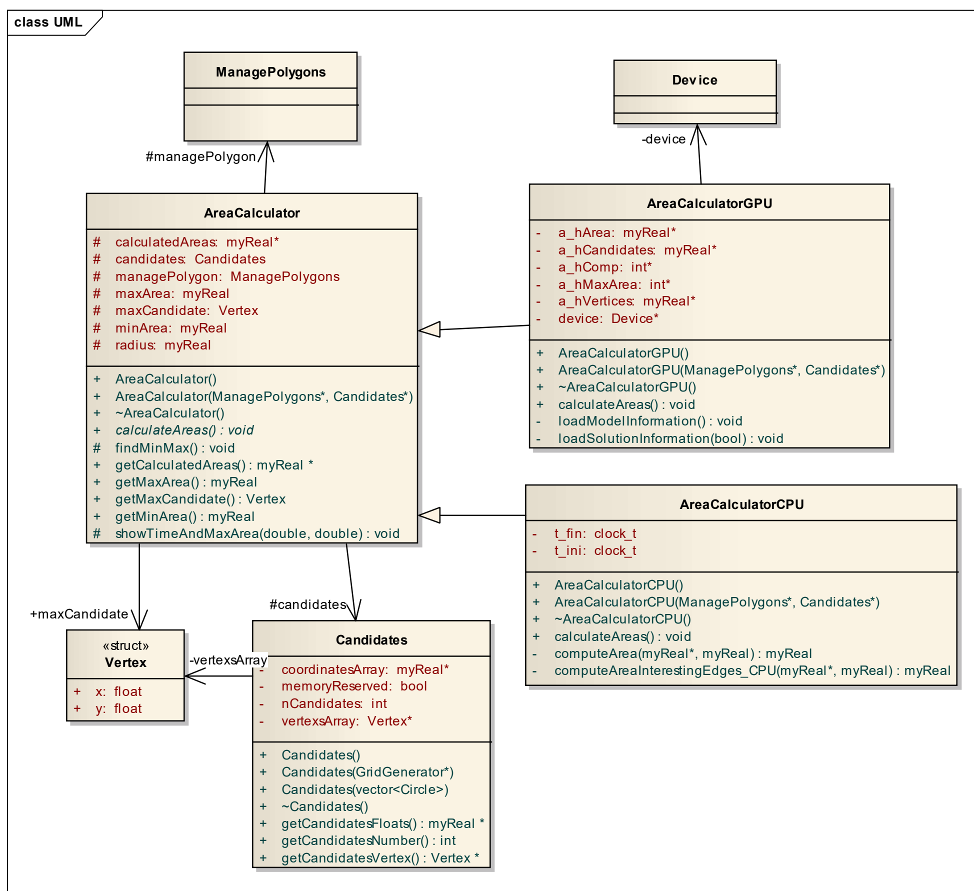


Diagrama 5. Diagrama de classes d'ÀreaCalculator

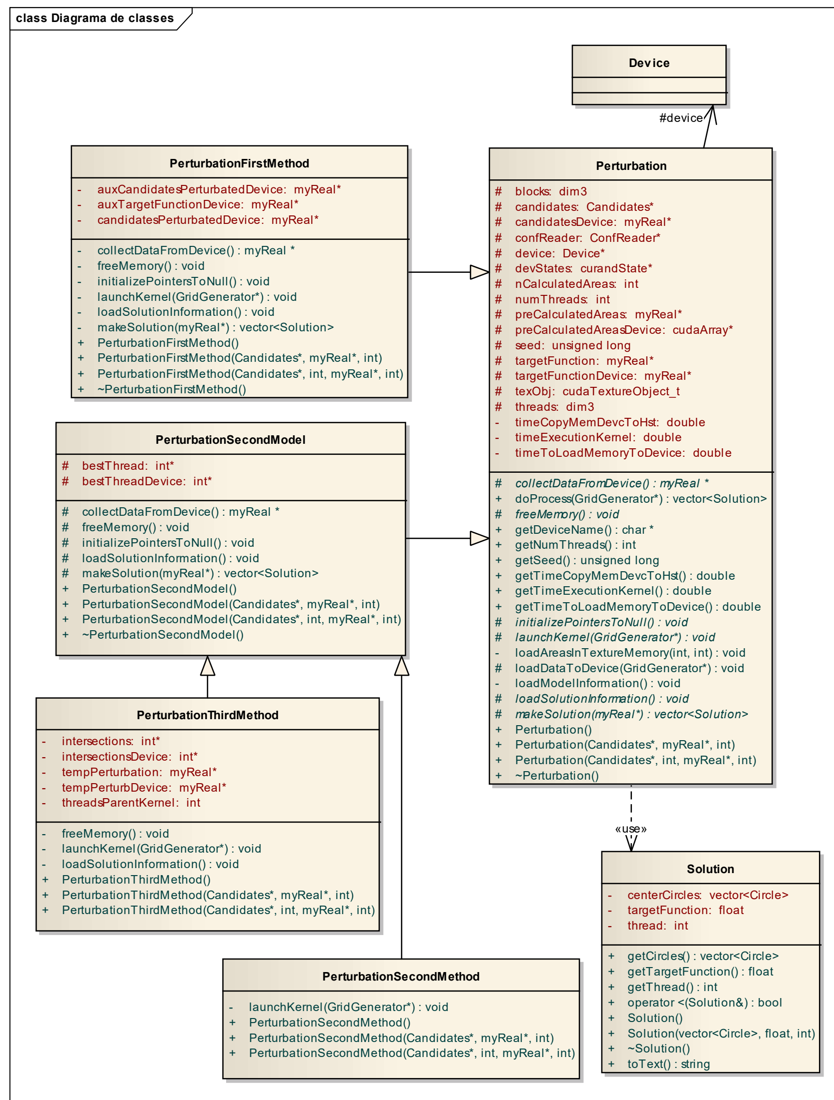


Diagrama 6. Diagrama de la classe Perturbation

El Diagrama 6 mostra el diagrama de classes de la classe Perturbation. La classe Perturbation és una classe abstracta i cada classe filla té una implementació diferent de l'algorisme *Simulated Annealing*. La seva diferència bàsicament radica en les diferents estratègies per paral·lelitzar l'algorisme, explicades anteriorment en el capítol 8.1.3. A part d'encarregar-se de l'execució de l'algorisme també mesura el temps que ha trigat i encapsula la solució dins de la classe Solution. On es guarda quin ha estat el valor de la funció objectiu, quins són els cercles d'aquesta funció objectiu i quin ha estat el *thread* que l'ha aconseguit.

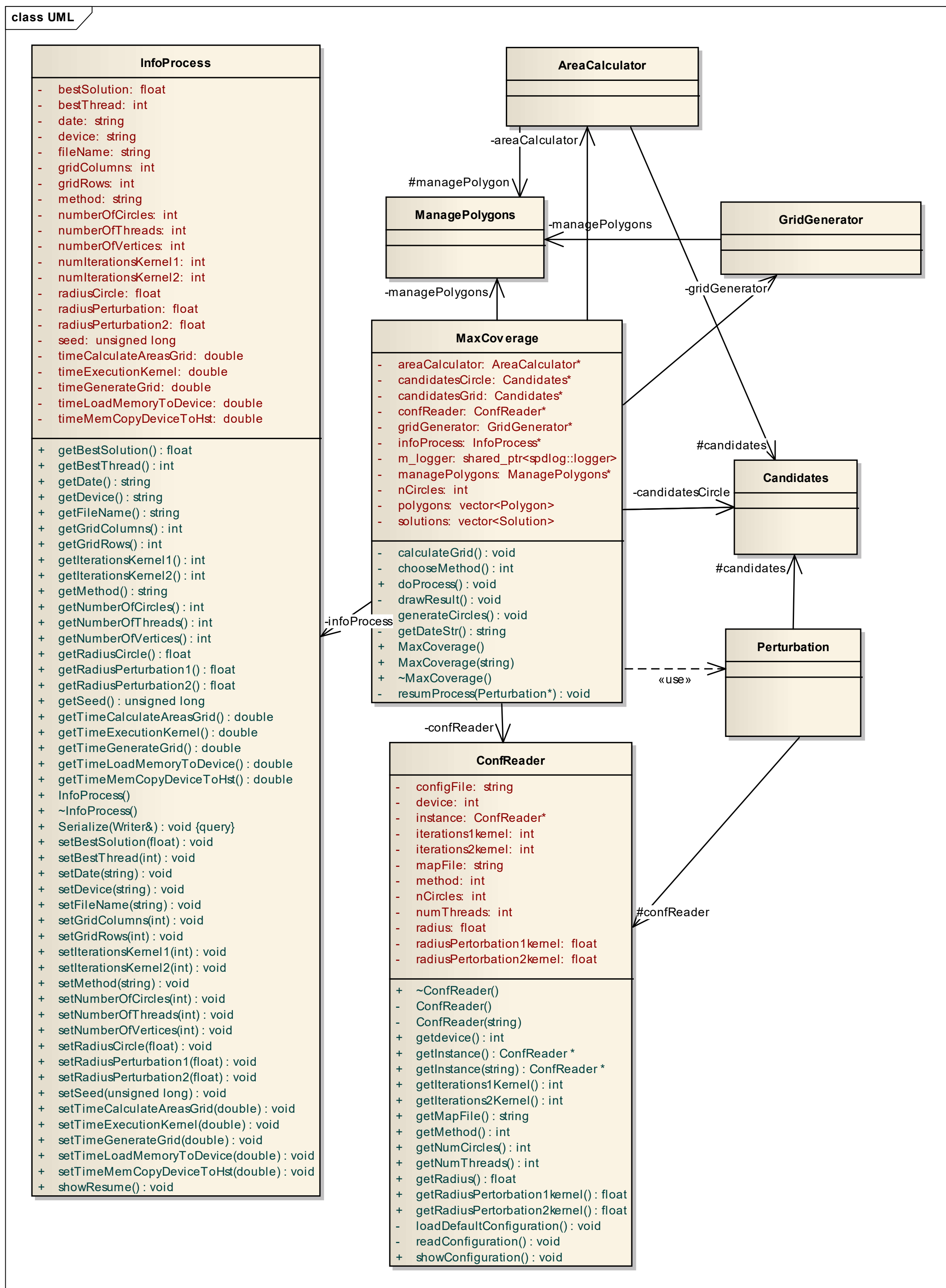


Diagrama 7. Diagrama de classes de MaxCoverage

La classe MaxCoverage, que es mostra en el Diagrama 7, és la classe que s'encarrega de fet tot el procés, és a dir de fer les crides pertinents a AreaCalculator i Perturbation per a que facin tots els càlculs. També l'encarregada de recollir tota la informació que ha generat el procés i guardar-la a l'objecte InfoProcess. La classe ConfReader, que també apareix en el Diagrama 7, també és una classe singleton i és la classe que conté tota la configuració del procés: fitxer d'entrada, nombre de cercles, nombre de *threads*, radi dels cercles, radis de pertorbació, nombre d'iteracions... La classe ConfReader pot llegir de fitxer o se li poden assignar els valors.

En el Diagrama 8 es mostra el disseny de les classes que tenen a veure amb la interfície d'usuari. Hi ha una classe principal que és MainForm0. Aquesta és l'encarregada de recollir la configuració de la interfície, encapsular-la dins de la classe ConfReader, i fer la crida a MaxCoverage per a què faci tot el procés de càlculs. MainForm0 conté un GLWidget, que és la classe encarregada de mostrar visualment els resultats amb OpenGL. La finestra OpenGL dissenyada a més, té la funcionalitat de poder navegar a través d'ella (amunt, avall, esquerra, dreta) fer zoom i rotar. Les classes Camera2d, Matrix , Transformation i Vector2f són les encarregades de fer les operacions de translació, rotació i escalat que atorga aquesta funcionalitat a la finestra GLWidget.

EL codi es pot trobar a: https://gitlab.com/MSaten/max-coverage-cuda

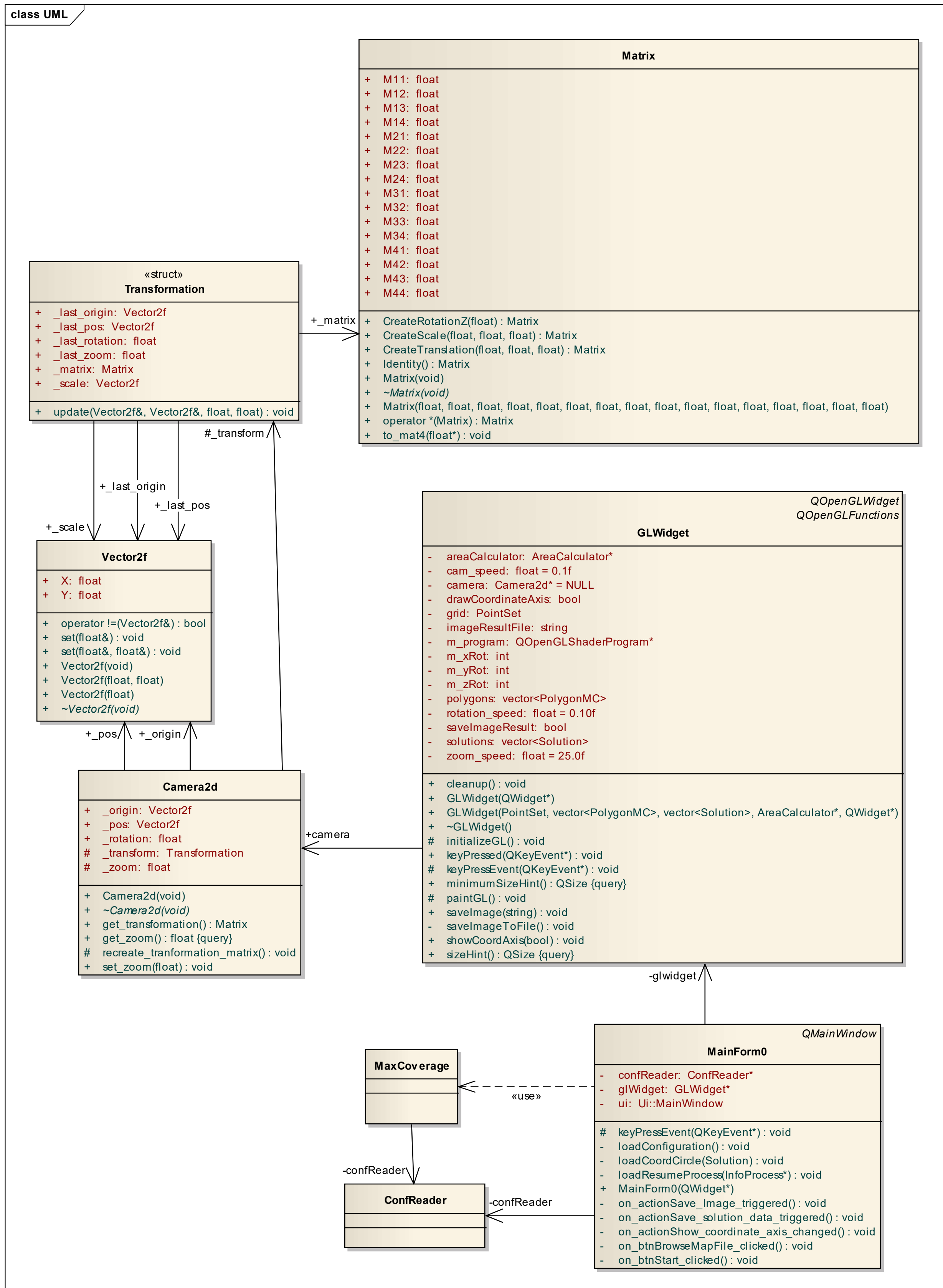


Diagrama 8. Diagrama de classes de la interfície d'usuari

8.4.3 Diagrama de seqüència

Els diagrames Diagrama 9, Diagrama 10 i Diagrama 11 són els diagrames de seqüència que mostren el flux del programa.

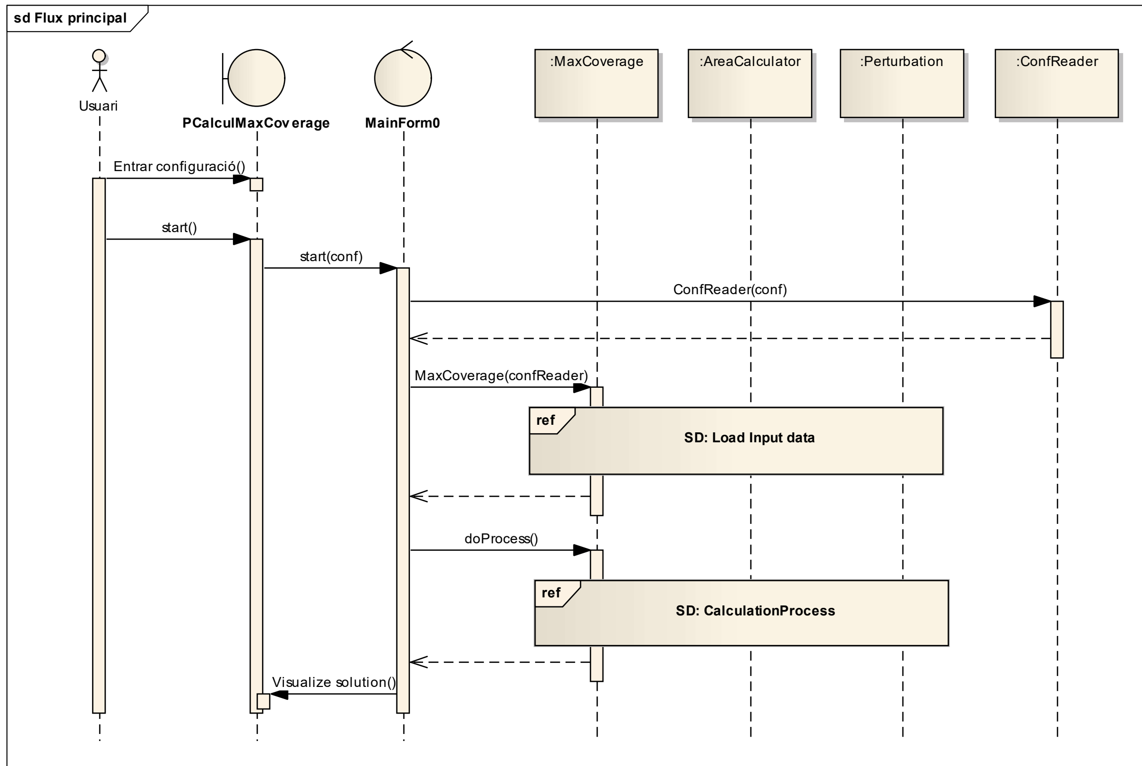


Diagrama 9. Diagrama de seqüència del flux principal

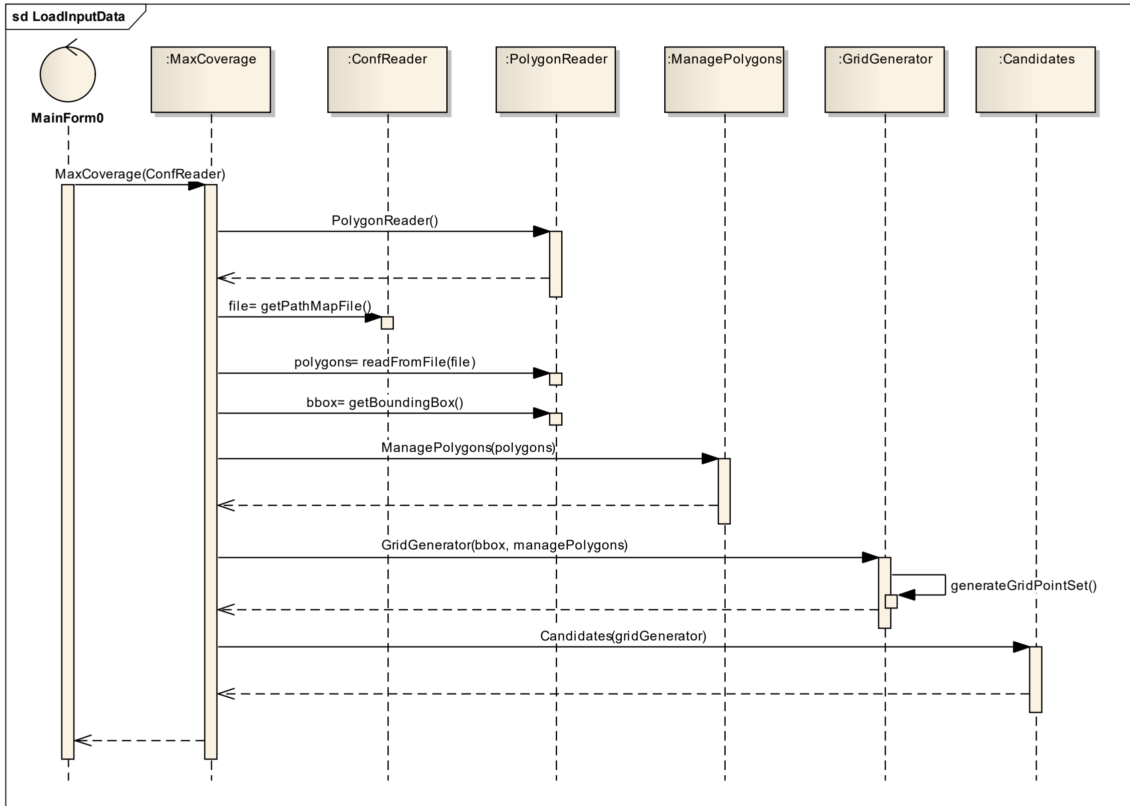


Diagrama 10. Diagrama de seqüència SD Load Input Data

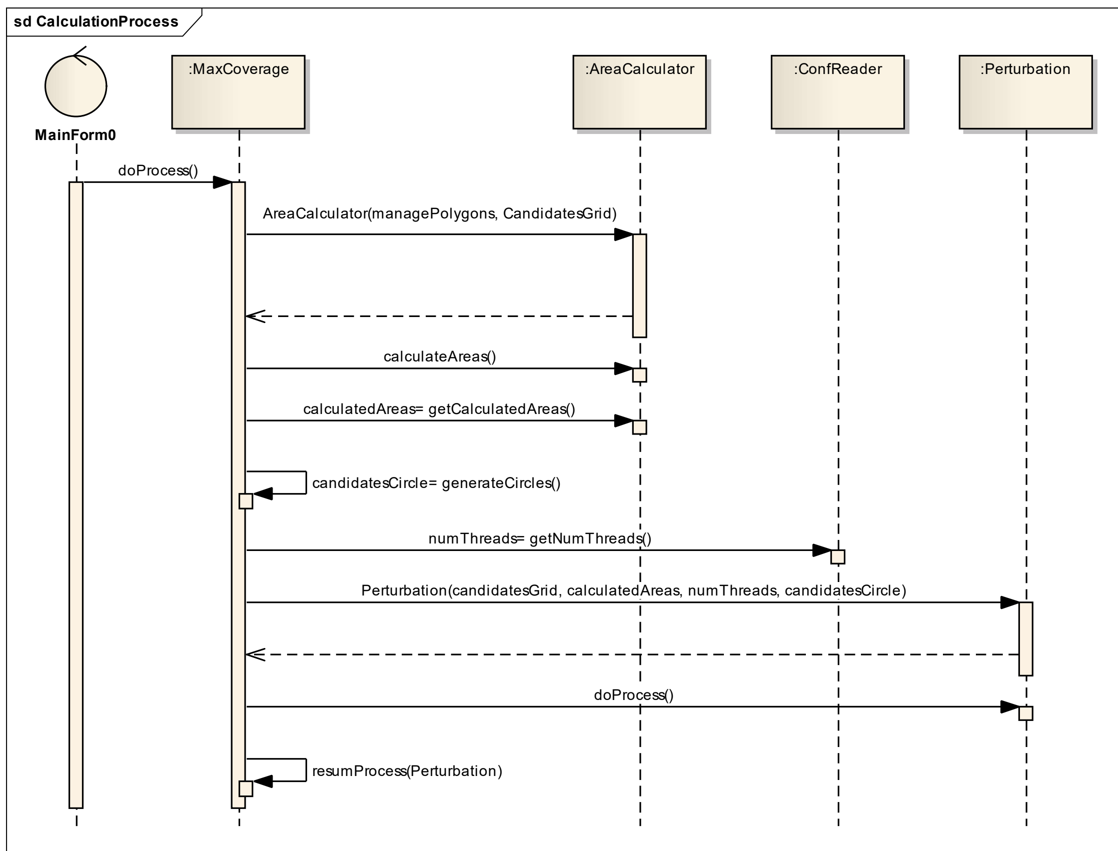


Diagrama 11. Diagrama de seqüència SD Calculation Process

8.5 Anàlisi de les dades

Durant l'evolució del projecte ha sorgit la necessitat d'analitzar el comportament dels diferents mètodes i dels seus paràmetres de configuració com per exemple el radi de pertorbació i nombre d'iteracions dels *kernels*. Per tal de fer aquesta anàlisi d'una forma més rapida i visual s'ha fet ús del *stack* ELK.

El primer que cal definir, és tota aquella informació que ens interessa monitoritzar. En el nostre cas s'ha fet que el nostre programa generi un fitxer de log on cada registre, en format JSON, guarda la informació de cada execució. En la Figura 49 es mostra un exemple de la informació que guardem per cada execució.


```
{
    "device": "Tesla K40c",
    "filename": "Polygons/losAngeles.txt",
    "date": "2019-04-14T17:40:40Z",
    "method": "2",
    "numVertices": 4951,
    "gridRows": 961,
    "gridColumns": 808,
    "numberOfCircles": 20,
    "radiusCircle": 0.2,
    "radiusPerturbation1": 0.4,
    "radiusPerturbation2": 0.87,
    "iterationsKernel1": 50,
    "iterationsKernel2": 450,
    "timeGenerateGrid": 0.01381182,
    "timeCalculateAreasGrid": 17.09052864,
    "timeLoadMemoryToDevice": 0.00522315,
    "timeExecutionKernel": 3.46709086,
    "timeMemCopyDeviceToHst": 0.00008424,
    "numberOfThreads": 512,
    "bestThread": 109,
    "seed": 1555256437,
    "bestSolution": 1.11656892
}
```


Figura 49. Exemple d'un registre de log

Cada vegada que s'insereix un registre en el nostre log, Logstash ho detecta, el llegeix i l'envia a Elasticsearch. Per altra banda Kibana mostra la informació de manera visual. Només s'ha d'obrir el navegador i anar a la direcció on tinguem el servei kibana. La Figura 50 és un exemple on és mostra la mitjana de temps d'execució del *kernel* depenent del nombre de cercles entre els diferents mètodes.

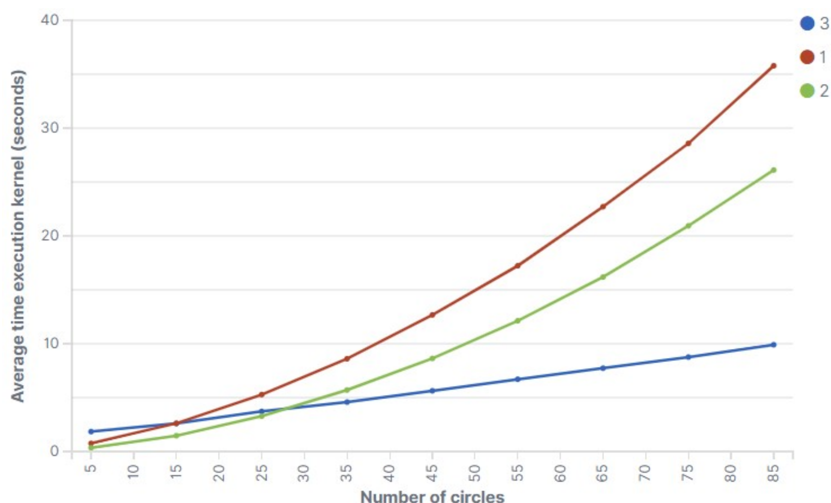


Figura 50. Exemple d'una gràfica generada per Kibana

9. Implementació i proves

En aquest capítol es comentarà en la capítol 9.1.1 alguns aspectes a tenir en compte a l'hora de generar nombres aleatoris amb CUDA i la implementació de l'algorisme Simulated Annealing en la capítol 9.1.2. En l'apartat de proves es detallaran algunes de les proves que testegen algunes de les funcionalitats del projecte.

9.1 Implementacions

9.1.1 Generació de nombres aleatoris

L'aleatorietat per a la generació de nous cercles dintre dels *kernels* presentava un problema. Ja que, fer que tots els *threads* parteixin d'un mateix *seed* és una mala idea, ja que dues successions a partir del mateix *seed* seran idèntiques. Una altra opció seria crear nombres aleatoris i pujar-los a la memòria global del *device*, però això també és mala idea perquè es malgastaria molta amplada de banda. Per a aquesta problemàtica CUDA ofereix la llibreria CURAND, i la seva documentació es troba a (NVIDIA Corporation, 2019, pág. cuRAND)

A la Figura 51 hi ha el codi que s'ha utilitzat per testejar aquesta aleatorietat dintre del *kernels*. I que es pot veure un exemple de la sortida en la capítol 9.2. El que es fa és reservar memòria en el *device* per allotjar una matriu de *curandState*, tants *curandState* com *threads* tinguem. Cada *thread* inicialitza amb el mètode `curand_init()` amb el mateix *seed* però amb un valor de seqüència diferent, en aquest cas el seu Id de *thread*. El mètode `curan_init()` deixa guardat a l'adreça de memòria corresponen un *state*. Aquest *state* és el que s'ha de passar a la funció `curand_uniform()` per a que ens doni el nombre aleatori.

```
__global__ void setup_kernel(curandState * state, unsigned long seed)
{
    int id = threadIdx.x;
    if (id < THREADS_TEST) {
        curand_init(seed, id, 0, &state[id]);
    }

}
__global__ void generate(curandState* globalState)
{
    int ind = threadIdx.x;

    if (ind < THREADS_TEST) {
        curandState localState = globalState[ind];
        float RANDOM = curand_uniform(&localState);
        globalState[ind] = localState;
        printf("Thread: %d Random: %.6f\n", ind, RANDOM);
    }
}
extern "C" void launchTestKernelRandomNumbers() {
    dim3 tpb(THREADS_TEST, 1, 1);
    curandState* devStates;
    cudaMalloc(&devStates, THREADS_TEST * sizeof(curandState));

    // setup seeds
    setup_kernel <<< 1, tpb >>> (devStates, time(NULL));

    // generate random numbers
    generate <<< 1, tpb >>> (devStates);
}
```

Figura 51. Codi de test de generació de números aleatoris amb CUDA

9.1.2 Simulated Annealing

En la Figura 52 és mostra la implementació de l'algorisme Simulated Annelaing implementada en el mètode 2. Les diferències amb la resta d'implementacions bàsicament són el pas de paràmetres a les funcions, i l'estructura del codi, ja que tenen distribucions de memòria diferents, i en el mètode 3 s'utilitza paral·lelisme dinàmic.

En el codi de la Figura 52 es pertorba i es calcula l'àrea per cada cercle, es calcula la intersecció entre els cercles i es calcula la funció objectiu. Un cop calculada s'avalua si s'accepta o no. I es redueix la temperatura (T) i el radi de pertorbació per a la pròxima iteració.

```
for (int i = 0; i < numIteracions; i++) {
    newTargetFunction = 0.0f;
    float totalArea = 0;
    for (int j = 0; j < nCandidates; j++) {
        //Perturb cercle
        perturbCercle(&algorithmPerturbation[j * 2], &tempPerturbation[j * 2],
            auxRadiPert, &globalState[gbId], tex, nRows, nColumns, xMin, xMax, yMin, yMax)

        //Get precalculated Area
        float preCalculatedArea = getPrecalculatedArea(&tempPerturbation[j * 2],
            tex, nRows, nColumns, xMin, xMax, yMin, yMax);

        //Total Area of all new circles
        totalArea = totalArea + preCalculatedArea;
    }
    //Calculate intersections
    float areaIntersection = calculateIntersections(&tempPerturbation[0], nCandidates,
        radiusCircle);

    //Calculate target function
    newTargetFunction = totalArea - areaIntersection;

    float probability = getProbability(newTargetFunction, algorithmTargetFuncion, T);
    if (acceptNewPerturbation(newTargetFunction, algorithmTargetFuncion, probability,
        &globalState[gbId])) {
        updateArrays(&bestTargetFuncion, newTargetFunction, &algorithmTargetFuncion,
            &bestPerturbation[0], &tempPerturbation[0],
            &algorithmPerturbation[0], nCandidates);
    }
    T = T * 0.95;
    if((i+1)%10==0)auxRadiPert = auxRadiPert * 0.95;
```

Figura 52. Implementació d'una iteració de Simulated Annealing del mètode 2

9.1.2.1 *Lògica Simulated Annealing*

En la Figura 53 es mostra el codi que decideix si una nova solució és acceptada o no, tal com està descrit en les linies 5-11 del pseudocodi que es mostra en el capítol 5.2.2.1.

```
__device__ bool acceptNewPerturbation(float newValue, float oldValue,
    float probability, curandState* _localState) {
    if (newValue > oldValue) {
        return true;
    }
    else {
        float r = getRandom(_localState);

        if (r < exp(probability)) {
            return true;
        }
        else {
            return false;
        }
    }
}
```

Figura 53. Codi lògica algorisme Simulated Annealing

9.2 Proves

Les proves més generals són les que es mostren a continuació. Bàsicament són funcionalitats essencials per saber si el codi funciona correctament.

9.2.1 Generar grid

La prova consisteix en generar un *grid* i que aquest es mostri per pantalla amb una finestra amb openGL com el de la *Figura 54*.

Es verifica que la classe GridGenerator genera el *grid* correctament.

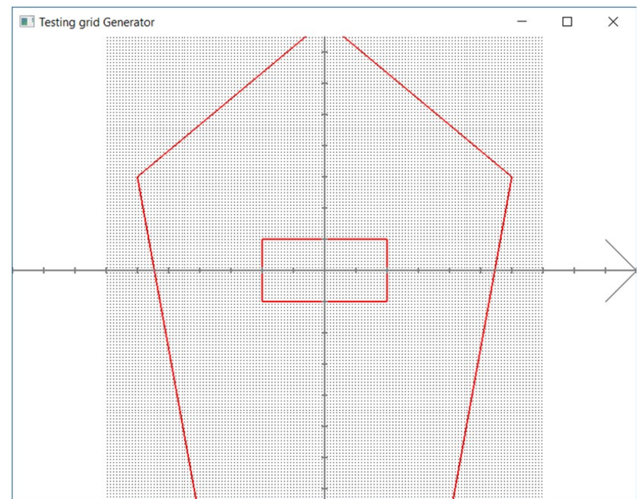


Figura 54. Finestra de comporvació del grid

9.2.2 Càlcul d'àrees del *grid*

- Càlcul d'àrees del *grid* amb GPU

Aquesta prova genera un *grid*, calcula quina és l'àrea d'intersecció entre cercle i polígon per cada punt del *grid* i el dibuixa (Figura 55). Essent les parts més fosques on l'àrea d'intersecció és màxima i blanc on l'àrea d'intersecció és zero. Per consola també es mostra el temps que s'ha trigat a calcular el *grid* i quina és l'àrea màxima que ha arribat a trobar (Figura 56).

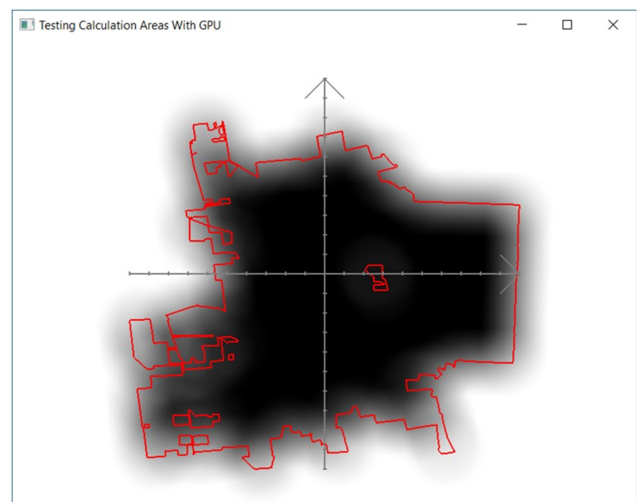


Figura 55. Finestra de comprovació del càlcul d'àrees amb GPU

```
Z:\workspace\Max_coverage_cuda\x64\Debu
*** Test CalculationAreaGPU ***
Time: 4.64578 (s)
Area max = 0.101794 u2
```

Figura 56. Sortida per consola de la prova de càlcul d'areas amb GPU

- Càlcul d'àrees del *grid* amb CPU

Aquesta prova és quasi igual que l'anterior, també mostra per pantalla el polígon amb el grid (Figura 54). L'única diferència és, que en lloc d'utilitzar la GPU per a fer els càlculs utilitza la CPU. Tant aquesta prova com l'anterior també tenen una variable del tipus bool per si volem que mostri per pantalla els valors de les àrees Figura 58. Només recomanable si s'estan executant proves amb grids petits.

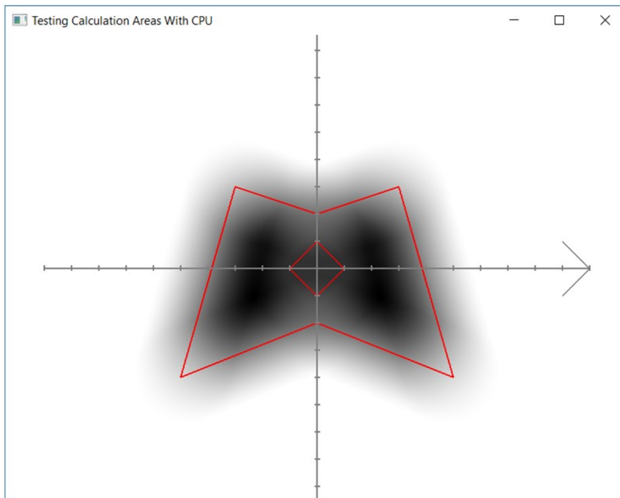


Figura 57. Finestra de comprovació del càlcul d'àrees amb CPU

```
Z:\workspace\Max_coverage_cuda\x64\De
*** Test CalculationAreaCPU ***
Time: 33.5563 (s)
Area max = 0.101071 u2
```

```
Z:\workspace\Max_co
0.00238318
0.0024455
0.00250925
0.00257063
0.00263021
0.00268804
0.00274418
0.00279
0.00156071
0.00161571
0.00167134
0.00172752
0.00178433
0.00184175
```

Figura 58. Sortida per consola de la prova de càlcul d'areas amb CPU

```
Z:\workspace\Max_coverage_cuda\x64\Debug\Max_covera
Area Fetched: 0.1010268
Area Calculated: 0.1010277
Dif: 0.0000009
Time: 0.00177214 (s)
Area max = 0.10104 u2

 **** Perturbation 97 ****
Area Fetched: 0.1010378
Area Calculated: 0.1010406
Dif: 0.0000028
Time: 0.00191798 (s)
Area max = 0.10105 u2

 **** Perturbation 98 ****
Area Fetched: 0.1010492
Area Calculated: 0.1010505
Dif: 0.0000013
Time: 0.00194777 (s)
Area max = 0.101069 u2

 **** Perturbation 99 ****
Area Fetched: 0.1010661
Area Calculated: 0.1010691
Dif: 0.0000030
Error average: 0.0000224
Press 0 to exit
```

Figura 59. Prova de marge d'error de Texture Fetching

9.2.3 Marge d'error *texture fetching*

Aquest prova fa N iteracions i per cada una d'elles pertorba un cercle, calcula l'àrea d'intersecció entre cercle i polígon i seguidament consulta a la memòria de textura quina és l'àrea. Es comparen aquestes dues àrees. Per cada iteració mostra l'àrea calculada, l'àrea consultada, i la diferència entre elles. Al final es mostra la mitja d'error (Figura 59).

9.2.4 Aleatorietat

Aquesta prova executa el llançament d'un kernel on cada thread mostra per pantalla el seu nombre aleatori generat en el rang [0.0,1.0](Figura 60).

```
Z:\workspace\Max_coverage_cuda\x64\Debug\Max_cove
Thread: 5 Random: 0.617117
Thread: 6 Random: 0.433868
Thread: 7 Random: 0.416028
Thread: 8 Random: 0.193397
Thread: 9 Random: 0.924033
Thread: 10 Random: 0.019400
Thread: 11 Random: 0.213180
Thread: 12 Random: 0.787544
Thread: 13 Random: 0.282200
Thread: 14 Random: 0.205503
Thread: 15 Random: 0.054662
Thread: 16 Random: 0.969279
Thread: 17 Random: 0.623430
Thread: 18 Random: 0.335302
Thread: 19 Random: 0.686278
Thread: 20 Random: 0.310352
Thread: 21 Random: 0.547326
Thread: 22 Random: 0.039274
Thread: 23 Random: 0.996867
Thread: 24 Random: 0.748574
Thread: 25 Random: 0.841495
Thread: 26 Random: 0.001161
Thread: 27 Random: 0.381696
Thread: 28 Random: 0.301868
Thread: 29 Random: 0.967178
Thread: 30 Random: 0.907571
Thread: 31 Random: 0.195311
Press 0 to exit
```

Figura 60.Sortida de Randomtest.cu

10. Implantació i resultats

EL nostre programa no requereix implantació ja que es tracta d'un projecte d'investigació i el resultat final és un executable, però sí que en el capítol 10.1 es comentarà com s'ha dut a terme la implantació del stack ELK, i en el capítol 10.2 es comentaran els resultats del nostre programa.

10.1 Implantació stack ELK

Per a la implantació dels serveis Elasticsearch i Kibana s'ha utilitzat una màquina virtual Ubuntu 16.04 LTS .

L'escenari on s'ha muntat el stack ELK és el que es mostra a la Figura 61, el qual ens ajudarà a entendre alguns paràmetres de les configuracions d'aquests serveis. Bàsicament els serveis Elasticsearch i Kibana es troben en una màquina Ubuntu dins de la xarxa local, i al mateix temps dintre d'una xarxa virtual 172.17.0.0/16 creada automàticament per docker. En la màquina Windows 10 es troben els logs que volem visualitzar i Logstash que els llegirà i els enviarà a Elasticsearch.

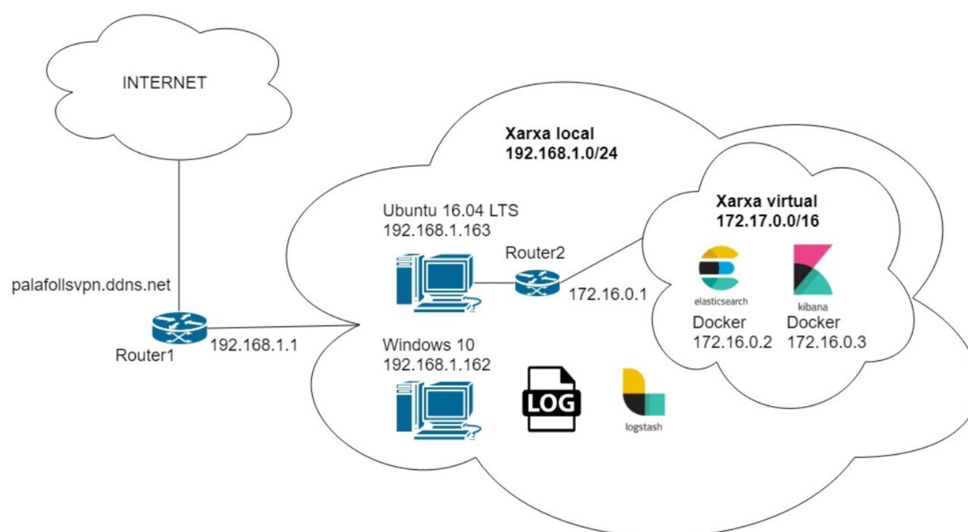


Figura 61. Escenari del stack ELK

10.1.1 Implantació Elascticsearch

Des de la consola de la màquina Ubuntu només cal descarregar la imatge d'Elasticsearch disponible en el repositori de docker amb la comanda **`docker pull`**, i la posem en marxa amb la comanda **`docker run`** i amb els ports 9200 i 9300 oberts:

```
$ docker pull docker.elastic.co/elasticsearch/elasticsearch:7.0.0
```

```
$ docker run -p 9200:9200 -p 9300:9300 \
-e "discovery.type=single-node" -e "TZ=Europe/Madrid" \
docker.elastic.co/elasticsearch/elasticsearch:7.0.0
```

Un cop està en marxa hem de configurar el nostre model de dades, aquest és el que es mostra en la ***Figura 49***. En aquest cas crearem un índex que li direm “cuda” i haurem d’especificar quin format tindrà aquest índex. Fent una analogia amb una base de dades convencional seria com crear una taula i especificar quins camps hi haurà i de quin tipus seran:

```
$ curl -X PUT "localhost:9200/cuda"
```

```
$ curl -X PUT "localhost:9200/cuda/_mapping" -H 'Content-Type: application/json' -d'{
 "properties": {
              "device" : { "type" : "text",
                             "fields" : {
                                    "keyword" : {
             "type" : "keyword",
             "ignore_above" : 256
            }
          }
      },
              "filename" : { "type" : "text",
           "fields" : {
             "keyword" : {
             "type" : "keyword",
             "ignore_above" : 256
              }
               }
      },
              "method" : { "type" : "text",
           "fields" : {
             "keyword" : {
             "type" : "keyword",
             "ignore_above" : 256
            }
          }
      },
        "date" : { "type": "date" },
    "numVertices" : { "type": "long" },
    "gridRows" : { "type": "long" },
    "gridColumns" : { "type": "long" },
    "numberOfCircles" : { "type": "long" },
        "radiusCircle" : { "type": "double" },
        "radiusPerturbation1" : { "type": "double" },
        "radiusPerturbation2" : { "type": "double" },
        "iterationsKernel1" : { "type": "integer" },
        "iterationsKernel2" : { "type": "integer" },
        "timeGenerateGrid" : { "type": "double" },
        "timeCalculateÀreasGrid" : { "type": "double" },
        "timeLoadMemoryToDevice" : { "type": "double" },
        "timeExecutionKernel" : { "type": "double" },
        "timeMemCopyDeviceToHst" : { "type": "double" },
        "numberOfThreads" : { "type": "long" },
        "bestThread" : { "type": "long" },
        "seed" : { "type": "long" },
    "bestSolution" : { "type": "long" }
  }
}'
```

Amb aquestes dues últimes comandes ja tindríem Elasticsearch amb un índex definit i preparat per començar a rebre dades.

10.1.2 Implantació Logstash

En aquest cas Logstash s'executarà des de windows 10, així que descomprimim l'arxiu descarregat de la pàgina oficial(B.V., 2019) i dintre de la carpeta *"<logstashFolder>/config"* creem un arxiu de configuració que anomenarem *"logstash.conf".*

El contingut del fitxer de configuració és senzill, és com el que es mostra en la Figura 62 on hi ha 3 parts:

- Input: Especifiquem que l'entrada és un fitxer i a on es troba aquest.
- Filter: El mateix Logstash envia un JSON a Elasticsearch on la nostra informació va continguda dins del camp *"message"*. En aquest cas indiquem que enviem un JSON dins d'aquest camp.
- Output: A on volem que enviï les dades, en aquest cas, a la nostra màquina Ubuntu pel port 9200 i també indiquem l'índex on volem que l'enviï, ja que podríem tenir varis.

```
input {
  file {
    path => "Z:/workspace/test-log.txt"
  }
}

filter {
    json {
        source => "message"
    }
}

output {
    elasticsearch {
        hosts => ["192.168.1.163:9200"]
        index => "cuda"
  }
}
```

Figura 62. Fitxer de configuració de Logstash

Un cop tenim el fitxer configurat, executem l'arxiu *"logstash.bat"* des de la consola de Windows indicant-li amb el paràmetre *"–f"* la ubicació del fitxer de configuració:

```
C:\<Path_to_logstashFolder>\bin\logstash.bat –f config\logstash.conf
```

A partir d'aquest moment qualsevol nova línia en el log serà enviada a Elasticsearch.

10.1.3 Implantació Kibana

La implantació de Kibana és molt semblant a la d'Elasticsearch. Des de la consola d'Ubuntu descarreguem el docker de Kibana del repositori amb docker pull, i aixequem el servei amb docker run. El servei Kibana estarà escoltant pel port 5601 i en aquest cas afegim el paràmetre -e ELASTICSEARCH_HOSTS=http://172.17.0.2:9200 on indiquem l'adreça IP i el port on està escoltant Elasticsearch.

```
$ docker pull docker.elastic.co/kibana/kibana:7.0.0

$ docker run -d -e ELASTICSEARCH_HOSTS=http://172.17.0.2:9200 \
-e "TZ=Europe/Madrid" \
--name kibana7-docker -p 5601:5601 \
docker.elastic.co/kibana/kibana
```

Un cop aixecat el servei Kibana, ja podríem accedir a aquest des de qualsevol navegador amb la direcció http://192.168.1.163:5601 des de la xarxa local, o si accedim des de fora de la xarxa local, havent obert prèviament una entrada NAT en el Router1, a la direcció http://palafollsvpn.ddns.net:5601.

Ara cal dir a kibana quin model de dades (índex) volem que ens ensenyi. En el nostre cas el model de dades que hem declarat en l'apartat 10.1.1 és "cuda". Per a crear aquest índex a kibana hem d'anar al menú esquerre i fer clic a "managment" i seguidament a "create index patern". Ens apareixerà un camp per escriure on escrivim "cuda". Ell sol ja detectarà que elasticsearch té aquest índex i ens el mostrarà, aleshores cliquem i ja tindrem el nostre patró d'índex creat.

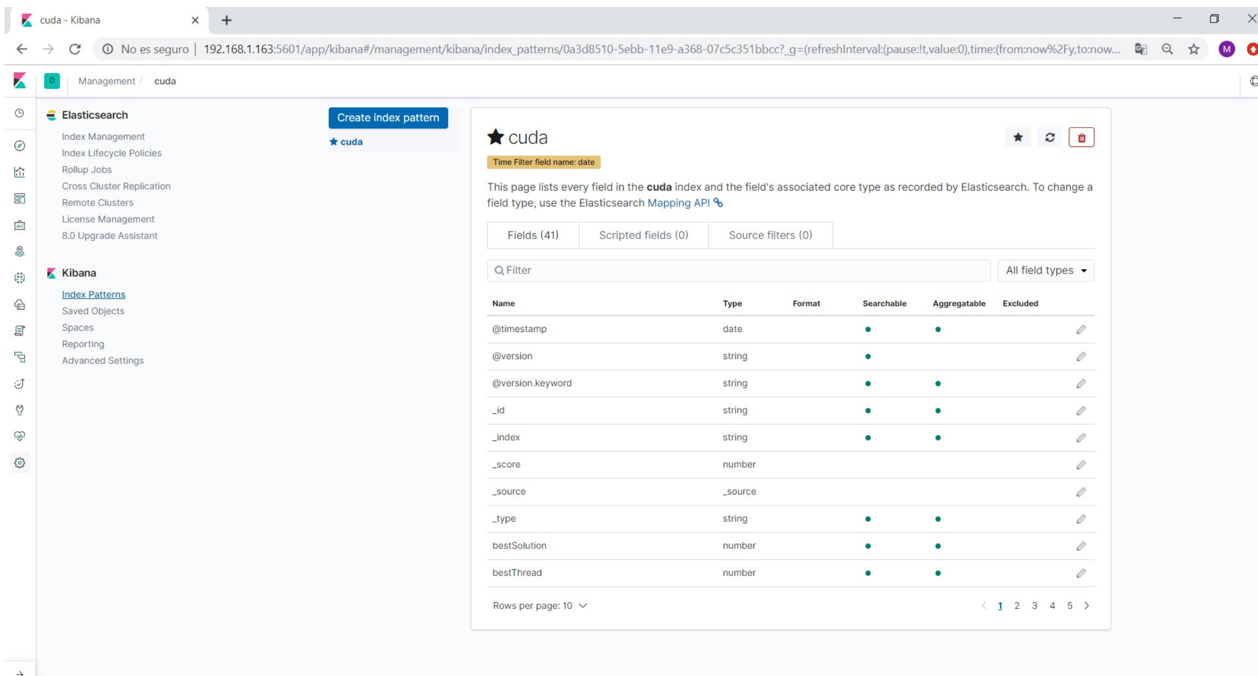


Figura 63. Create index pattern Kibana

10.2 Resultats

A continuació es detallen les proves que s'han fet. El hardware que s'ha utilitzat per a dur a terme les proves són:

- CPU: Inter(R) Core(TM) i7-4790CPU
- RAM: 32 GB
- GPU: Tesla k40

Els resultats finals són satisfactoris. En la Figura 64, Figura 65, Figura 67 i Figura 66 es mostren alguns exemples de proves fetes a la ciutat de los Ángeles, California, EE.UU. i Dublin, Ohio, EE.UU. on a simple vista es pot veure que s'aconsegueixen solucions òptimes.

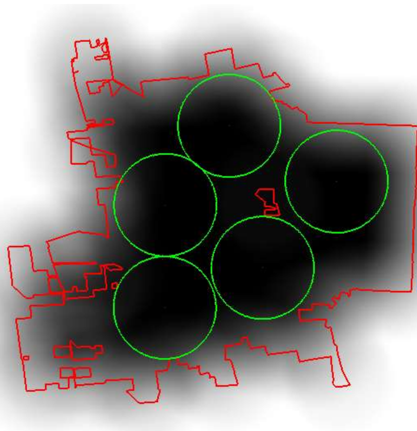


Figura 65. Dublin amb 5 cercles de radi 2,5

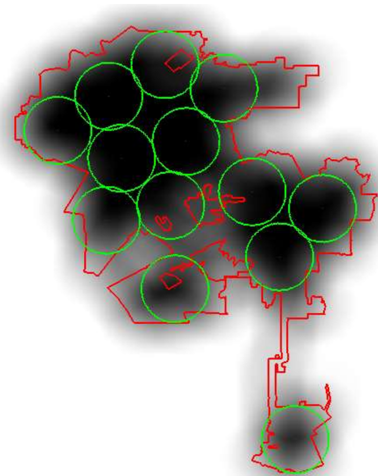


Figura 64. Los Ángeles amb 13 cercles de 0,17de radi

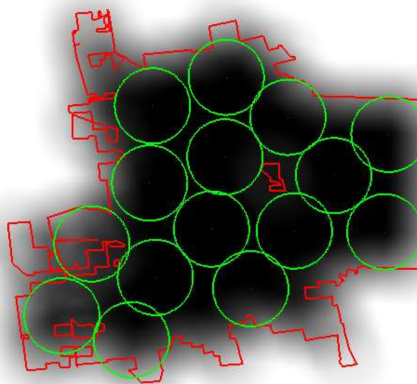


Figura 67. Dublin amb 15 cercles de radi 0,18

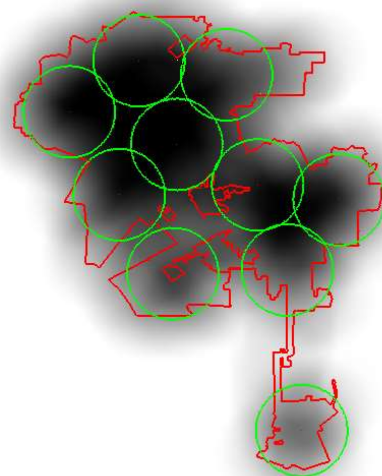


Figura 66. Los Ángeles amb 10 cercles 0,2 de radi

S'han fet algunes proves de rendiment als diferents mètodes. Els paràmetres d'entrada són els que es mostren en la Taula 3.

Nombre de *threads*	512
Nombre de Cercles	10
Radi r	0.18
Radi de pertorbació 1er *kernel*	0.40
Radi de pertorbació 2on *kernel*	0.08
Iteracions 1 er *kernel*	45
Iteracions 2on *kernel*	450
Nom del fitxer	losAngeles.txt

Taula 3. Paràmetres d'entrada dels tests

La Figura 68 mostra els temps d'execució dels diferents mètodes implementats, on es pot veure que el mètode 2 és el mètode més ràpid dels tres amb una mitjana de 0.139s. Una diferència de 1.936s amb el tercer mètode (2.075s) que sobrepassa dels dos segons i una diferència amb el primer de 0.338s. Tot i que el tercer mètode utilitza paral·lelisme dinàmic per al càlcul del nou cercle i la intersecció amb els altres cercles, el cost en temps que té el llançament d'un *grid* secundari no compensa en aquest cas. El primer mètode està penalitzat per la gran quantitat de memòria global que s'utilitza, ja que és la memòria més lenta de totes.

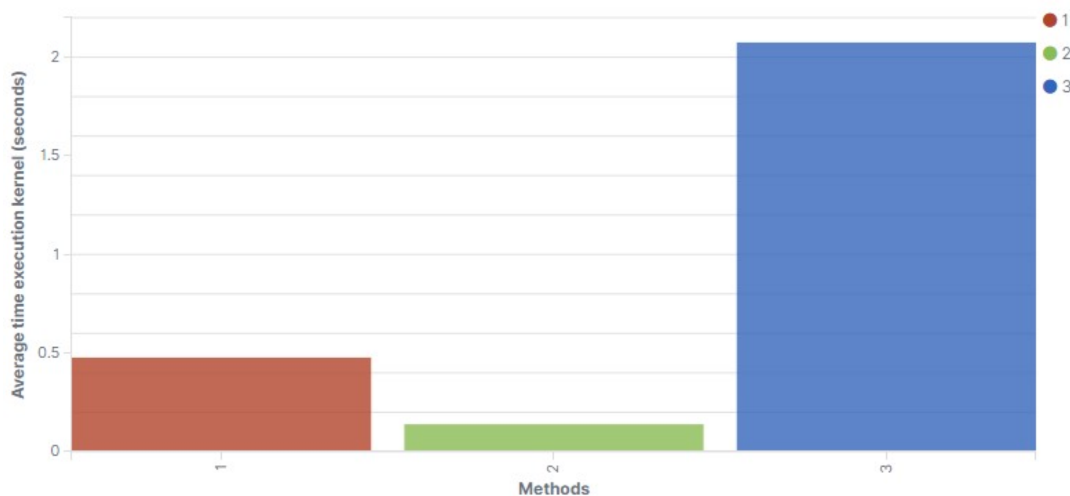


Figura 68. Mitjana dels temps d'execució del kernel dels diferents mètodes

En la Figura 69 mostra el temps dels diferents mètodes respecte al nombre de cercles. Les proves han estat fetes amb els mateixos paràmetres d'entrada que es mostren Taula 3, però s'han fet més proves incrementant el nombre de cercles. Es pot observar que a partir de 50 cercles comença a ser més rentable utilitzar el paral·lelisme dinàmic del mètode 3. Els mètodes 1 i 2 estan penalitzats pel doble *for* comentat en el capítol 8.1.3, que fa que el temps d'execució sigui exponencial, mentre que el paral·lelisme dinàmic del tercer mètode fa que l'augment del temps sigui lineal, tot i que és podria considerar que el temps es constant.

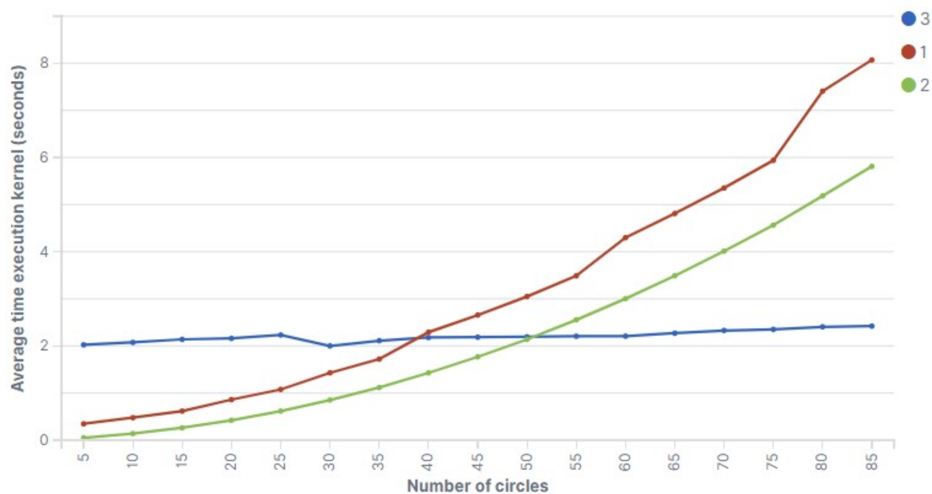


Figura 69. Mitjana del temps d'execució del kernel dels diferents mètodes en funció del nombre de cercles

Pel que fa als resultats de la funció objectiu, la Figura 70 mostra la mitjana dels valors de la funció objectiu dels diferents mètodes. S'observa que tant el segon i tercer mètode han arribat al mateix valor de la funció objectiu, trobant solucions igual de bones. En canvi el primer mètode està lluny de trobar solucions òptimes. Això és degut al fet que el primer mètode només utilitza el llançament d'un kernel, no utilitza l'estratègia descrita en el capítol 8.1.3.4.

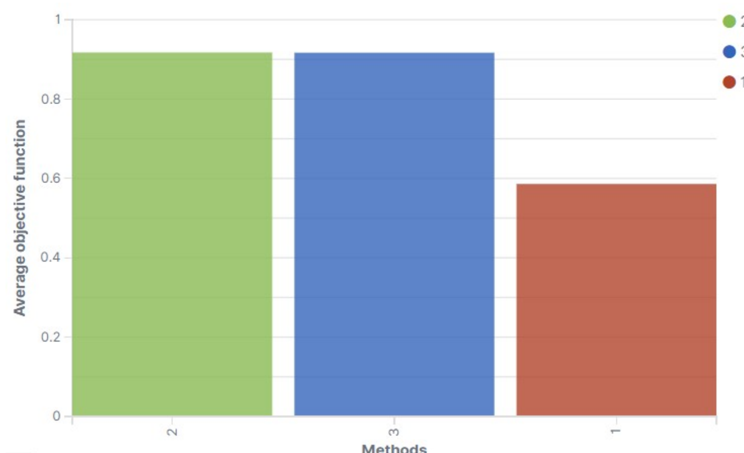


Figura 70. Mitjana de la funció objectiu dels diferents mètodes

Per tant es pot concloure que el segon mètode és el més eficient per trobar una solució optima al problema i que a partir d'uns 25 cercles el tercer mètode trobarà una solució igual d'òptima i de forma més eficient que el mètode 2.

11. Conclusions

Els objectius del projecte consistien a desenvolupar una eina que fos capaç de trobar una solució òptima al problema CMCP amb k facilitats d'una manera eficient, utilitzant les capacitats de les GPU, que fos fàcil d'utilitzar i que a més representés els resultats d'una forma senzilla. Per tant, un cop analitzats els resultats, els temps d'execució i la interfície gràfica es considera l'assoliment dels objectius.

Tot i que no s'ha trobat un valor predeterminat pels paràmetres d'entrada "radi de pertorbació" que funcioni per a totes les combinacions de mapes, nombres de cercles i radi de cercle, sí que s'ha acotat aquest valor i a més s'ha implantat un sistema que ens facilita l'estudi del comportament d'aquests, en aquest cas el stack ELK.

Pel que fa a les conclusions personals estic molt satisfet amb aquest projecte, ja que deixant de banda els coneixements adquirits d'OpenGL i QT, he après des de zero una tecnologia com CUDA, per la que ja sentia curiositat, i aplicar-la a un problema i que funcioni ha estat un repte. També he tingut el plaer de treballar dins d'un equip d'investigació fantàstic que m'ha anat guiant i que ha elaborat una publicació ja acceptada a "Encuentros de Geometría Computacional" aquest mes de Juliol en referència a aquest projecte i que s'adjunta en l'Annex A.

12. Treball futur

Hi ha diversos fronts oberts en els quals es pot seguir desenvolupament aquest projecte.

Un dels més interessants seria fer una llibreria amb tota aquella part del codi que s'encarrega del problema de cobertura utilitzant GPU. És a dir, tota la part que s'encarrega de la generació de la graella, el càlcul de les àrees de la graella, les diferents estratègies que implementen el Simulated Annealing, etc. D'aquesta manera tota aquesta part quedaria totalment aïllada d'una representació gràfica de la solució i/o de la interfície d'usuari i dels llenguatges i/o tecnologies que s'utilitzessin per a aquesta finalitat. És una tasca que no ha donat temps en el desenvolupament d'aquest projecte però el codi ja està força encaminat cap a aquesta finalitat.

Pel que fa a les estratègies implementades actualment caldria fer un estudi més a fons dels resultats i trobar una manera d'automatitzar el càlcul dels valors d'entrada com el radi de pertorbació i el nombre d'iteracions dels *kernels*. I que aquests depenguessin de l'àrea total a cobrir, el nombre de cercles i el radi dels cercles. La implementació del stack ELK va sorgir d'aquesta necessitat d'estudi dels algorismes amb diferents paràmetres i diferents escenaris. De tal manera que és molt fàcil veure la variació de resultats, temps, etc. respecte a paràmetres d'entrada i algorismes. També han quedat en el tinter altres estratègies de paral·lelització que no han donat temps a implementar-se i que podrien oferir millors temps o no.

Donar solució a aquest mateix problema amb facilitats de diferents radis també seria un altre front on poder continuar amb aquest projecte.

Es podrien també afegir funcionalitats a l'usuari, com podrien ser reproduir una solució a partir d'un seed, o guardar i carregar un fitxer de configuració que contingués tots els paràmetres d'entrada: fitxer, nombre de cercles, radi de pertorbació, etc. O inclús automatitzar bateries de proves amb diferents fitxers de configuració.

Una altra funcionalitat per afegir seria una opció per normalitzar les coordenades dels polígons que no estiguin en el rang [-1, 1]. De fet en el codi font ja està implementada aquesta funcionalitat, necessita un fitxer d'entrada i escriu en un fitxer de sortida, però no està afegida a la interfície gràfica.

De cara als investigadors que segueixin amb l'actual projecte també es pot fer una versió que no requereixi interfície gràfica i que les dades de configuració són llegides d'un fitxer. Això agilitza el procés de proves i anàlisi de dades. En el codi font també està implementatda la classe ConfReader que té l'opció de llegir de fitxer, seria fàcil també fer aquest executable.

13. Manual d'usuari i/o instal·lació

13.1 Execució del programa

El resultat final del projecte és un executable per Windows. L'únic requeriment és tenir una targeta gràfica compatible en CUDA amb una Compute Capability de 3.5 o superior i amb els drivers instal·lats perquè el sistema operatiu la reconegui.

Cal descomprimir l'arxiu "MaxCoverage.rar" i dins de la carpeta "MaxCoverage" es troba la carpeta "Polygons" que conté alguns polígons d'exemple per fer proves, "dublinOhio.txt", "losAngeles.txt", etc. La carpeta "Logs" conté el fitxer "executionsLog.txt" on es guardaran els registres de cada execució comentats en el capítol 8.5. L'executable és el fitxer "Max_coverage_cuda.exe", només cal fer doble clic i s'obrirà la interfície com el de la Figura 71. Pressionant el botó "Open file" podem escollir el fitxer de text que contindrà les coordenades del nostre polígon, a la carpeta "Polygons" hi ha alguns. A la dreta hi ha el menú "Configuration" on es poden configurar els diferents paràmetres: nombre de cercles, radi dels cercles, targeta gràfica, etc.

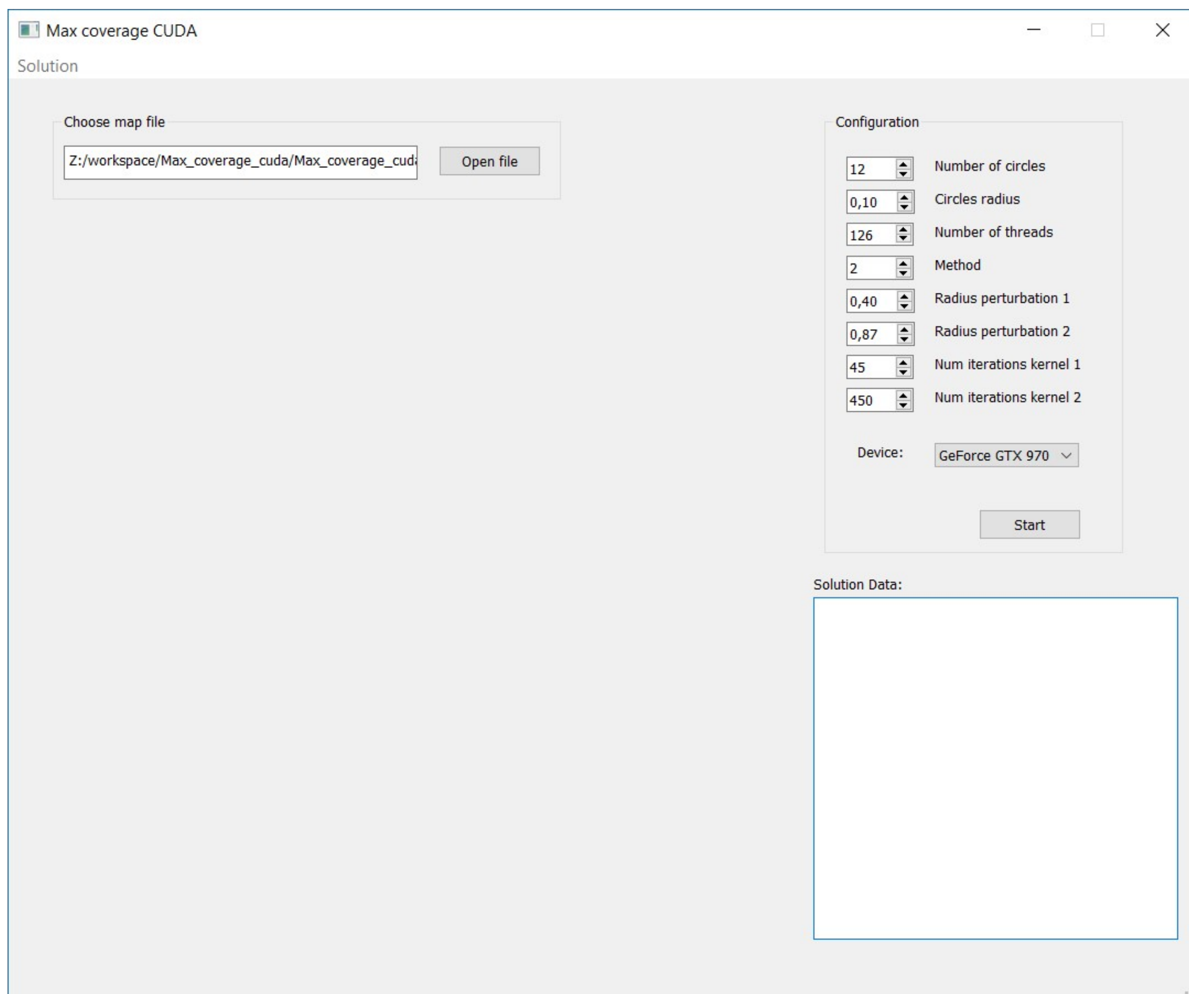


Figura 71. Interfície gràfica de l'aplicació

Un cop escollit el fitxer i configurats els valor d'entrada ja es pot prémer el botó "Start". Un cop acabat el procés apareixerà la solució com la que es pot observar en la Figura 72, on s'activen 3 elements:

- La representació gràfica de la solució.
- Quadre de text "Solution Data" on apareix informació rellevant recollida durant el procés.
- S'activa la pestanya "Solution" que ens permet accedir a opcions de la solució.

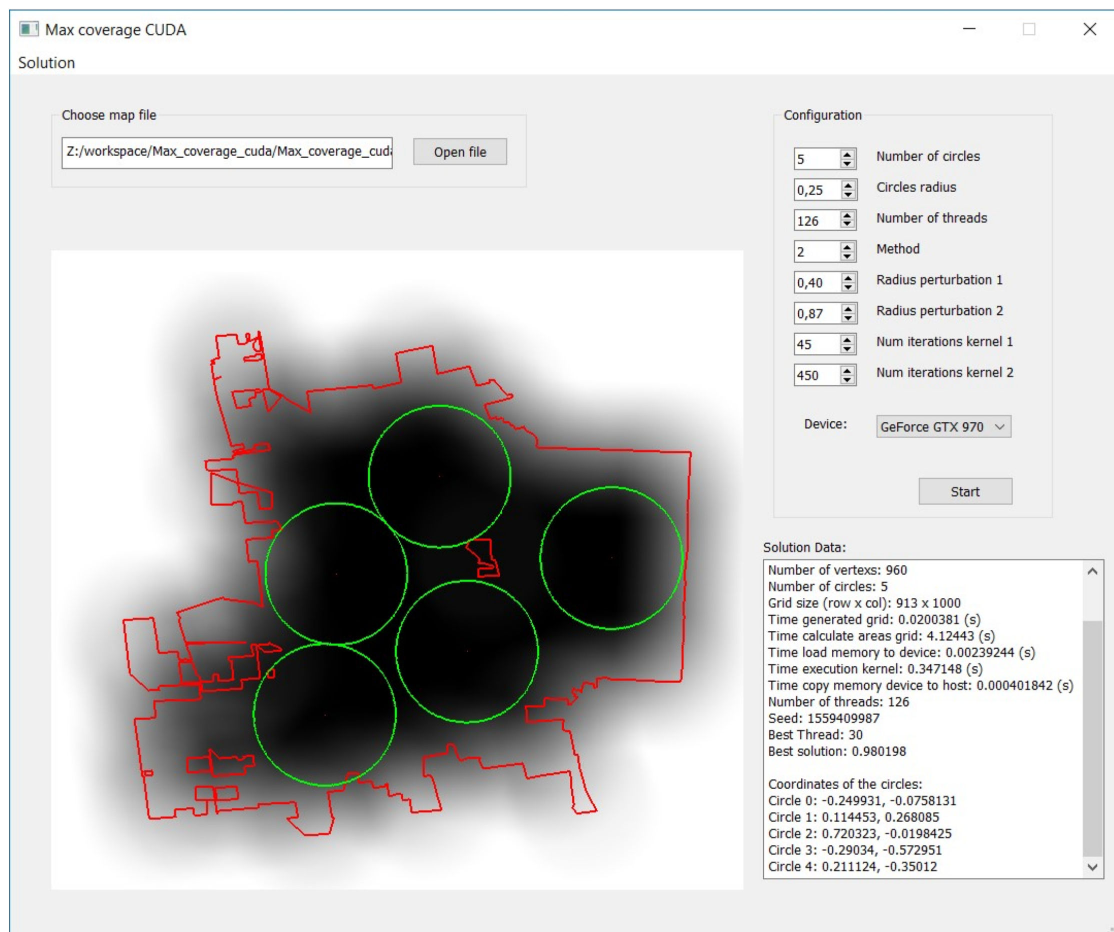


Figura 72. Interfície gràfica amb una solució

13.1.1 Representació gràfica de la solució

Aquesta finestra ens permet moure'ns a través d'ella, rotar la imatge i fer zoom, amb les següents tecles:

- A: Esquerra
- S: Avall
- D: Dreta
- W: Amunt
- Z: Zoom –
- X: Zoom +
- C: Rotar dreta
- V: Rotar esquerra

```
Solution Data:
Device: GeForce GTX 970
File name: dublin_ohio3.txt
Method: 2
Number of vertexs: 960
Number of circles: 5
Grid size (row x col): 913 x 1000
Time generated grid: 0.0198107 (s)
Time calculate areas grid: 5.00778 (s)
Time load memory to device: 0.0041956 (s)
Time execution kernel: 0.308327 (s)
Time copy memory device to host: 0.000533411 (s)
Number of threads: 130
Seed: 1559410675
Best Thread: 127
Best solution: 0.980574

Coordinates of the circles:
Circle 0: 0.0620352, 0.32719
Circle 1: -0.241497, -0.0834049
```

Figura 73. Text box "Solution Data"

13.1.2 Interpretació dels resultats

A la part inferior dreta de la interfície apareix un quadre de text anomenat "Solution Data" que mostra tota la informació referent a la solució, on comentarem els més importants:

- Number of vertexs: Són el nombre total de vèrtexs que té el polígon que defineix la regió de demanda.
- Grid size: Mostra la quantitat de punts del nostre *grid*, files per columnes.
- Time generated grid: És el temps que ha trigat a generar-se el *grid*.
- Time load memory to device: Temps que s'ha trigat a passar la informació necessària des de l'espai de memòria de la CPU a l'espai de memòria de GPU.
- Time execution kernel: Temps que ha trigat l'algorisme Simulated Annealing a trobar la solució.
- Time copy memory device to host: Temps que s'ha trigat un cop finalitzat tot el procés a passar la informació del *device* al *host*.
- Seed: És la llavor que s'ha utilitzat per a l'aleatorietat de les pertorbacions. Potser és d'interès, ja que potser ens interessa reproduir el procés.
- Best Thread: És l'ID del *thread* que ha trobat la solució que és mostra per pantalla. Ha estat la millor solució.
- Best solution: És el valor de la funció objectiu aconseguit.
- Coordinates of the circles: Són les coordenades del centre dels cercles.

13.1.3 Pestanya “Solution”

Aquesta pestanya només s’activa un cop generada la solució i ens permet:

- Guardar la representació gràfica de la solució en un fitxer d’imatge.
- Mostrar o no els eixos de coordenades en la representació gràfica tal com es mostra en la Figura 75.
- Guardar la informació generada pel procés en un fitxer de text.

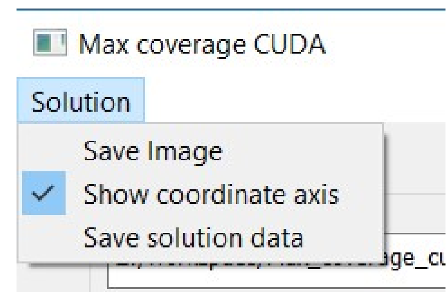


Figura 74. Opcions de la pestanya “Solution”

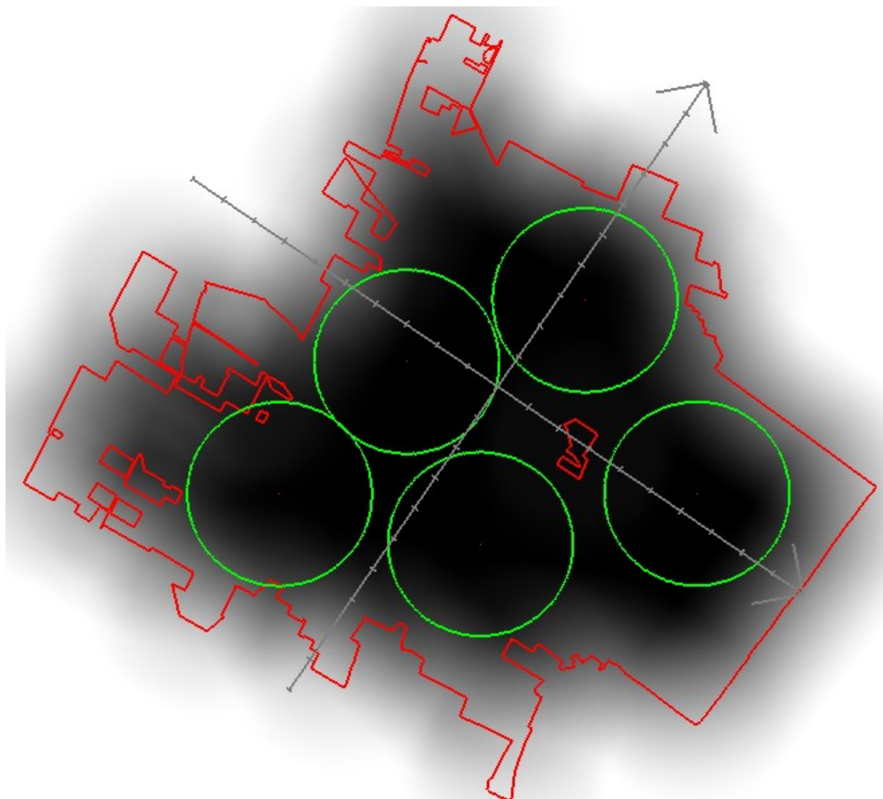


Figura 75. Solució mostrada amb els eixos de coordenades

13.2 Anàlisi del resultats

Per consultar les dades només cal obrir el navegador i anar a l'adreça IP i port on tinguem escoltant el nostre servei, en el nostre cas és: http://palafollsvpn.net.ddns:5601 si accedim des de fora de la xarxa local o http://192.168.1.163:5601 si accedim des de la xarxa local.

En (B.V., 2019) s'explica tot el necessari per utilitzar kibana, ja que té moltes opcions. Aquí explicarem només les necessàries per veure els nostres resultats.

Quan accedim al servei web ens trobem amb la pàgina principal la qual a la part esquerra es mostra un menú amb opcions (Figura 76). Les que més ens interessen per consultar són "Discover", "Visualize" i "Dashboard".

Si cliquem a "Discover" se'ns mostraran ordenats cronològicament tots els registres (*Figura 77*), aquí es poden fer cerques puntuals sobre algun registre o grup de registres, depèn del que interessi. Per exemple si volem podem veure tots els resultats d'un mapa o d'un mètode en concret, etc.

L'opció "visualize" ens mostra totes les gràfiques que ja tenim dissenyades i també podem generar de noves. La generació de gràfiques és bastant intuïtiva però en cas de dubtes es pot consultar (B.V., 2019).

L'opció dashboards ens mostra tots els dashboards que hem generat. Un dashboard no és res més que una agrupació de gràfiques. En el nostre cas només hem generat un i és el que es mostra en la Figura 78. Si fem clic a qualsevol de les gràfiques que es mostren en el dashboard aquesta se'ns obrirà i mostrarà tots els detalls.

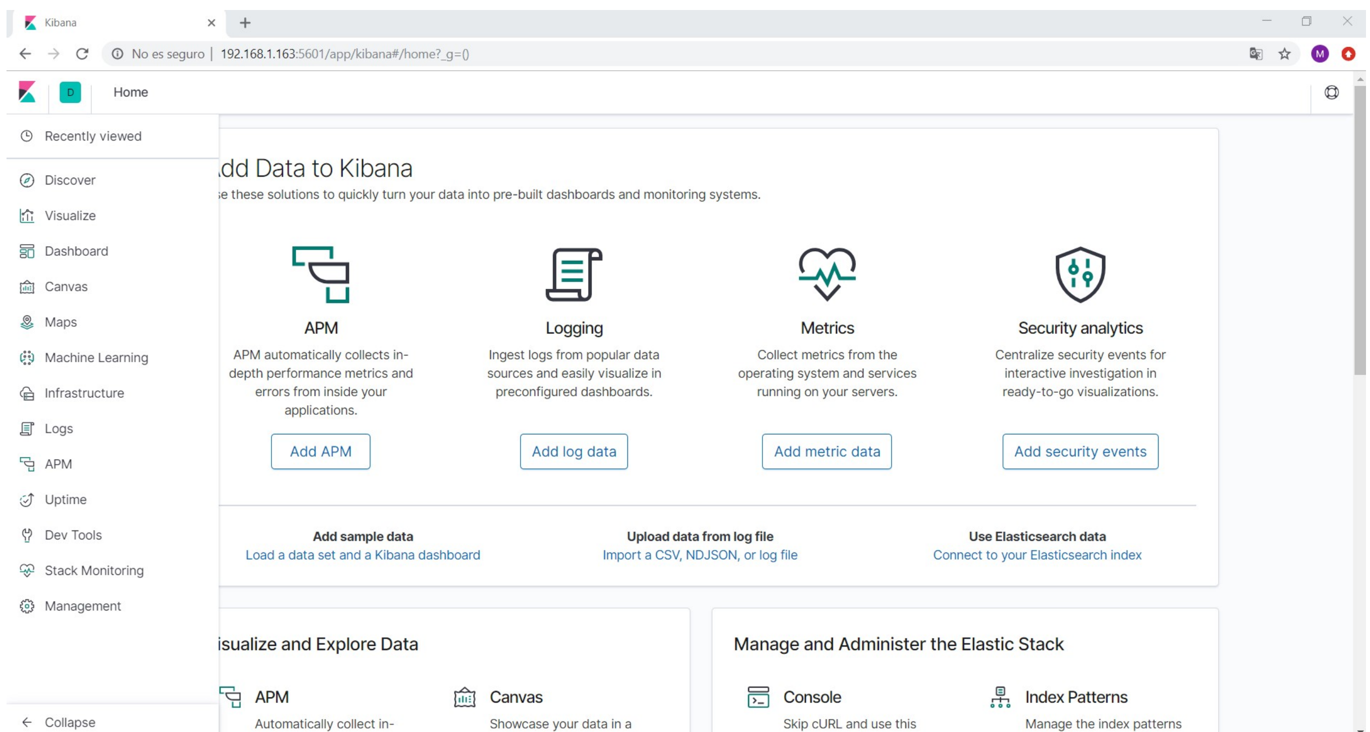


Figura 76. Menú principal kibana

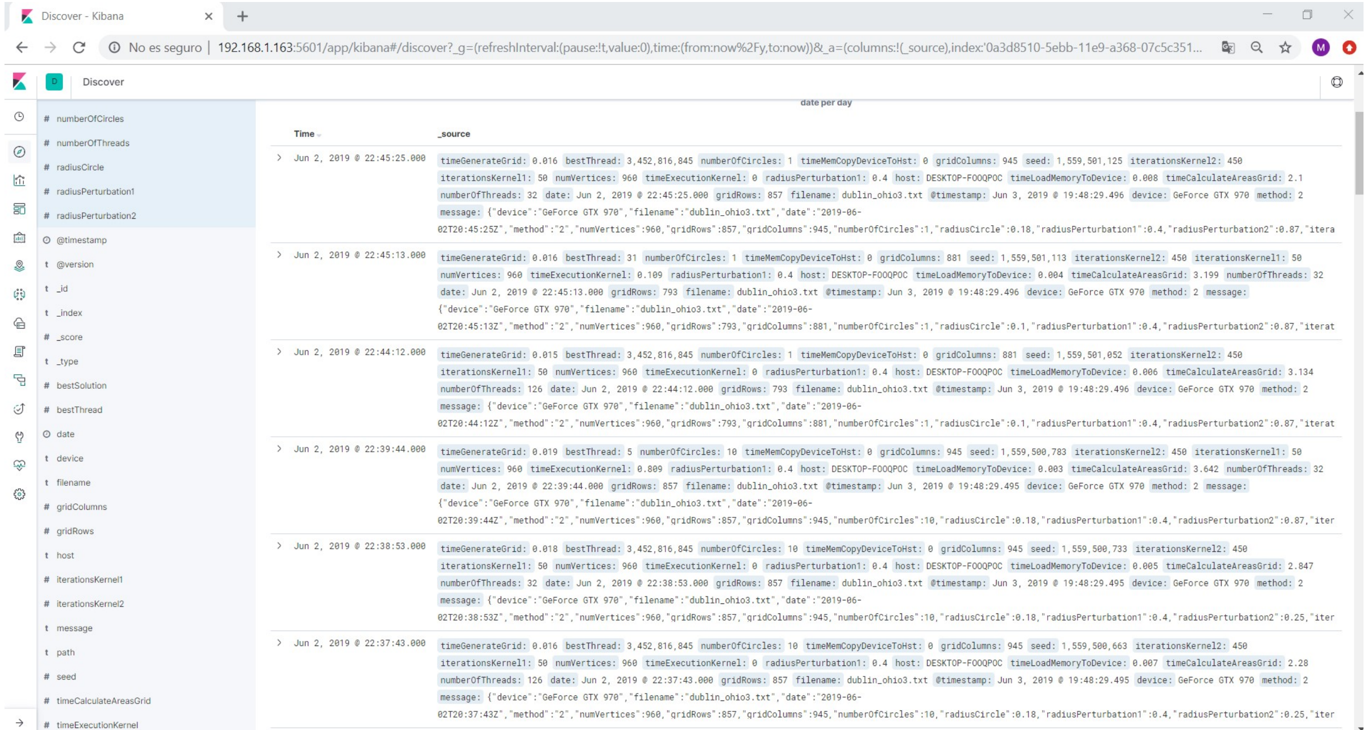


Figura 77. Menú Discover Kibana

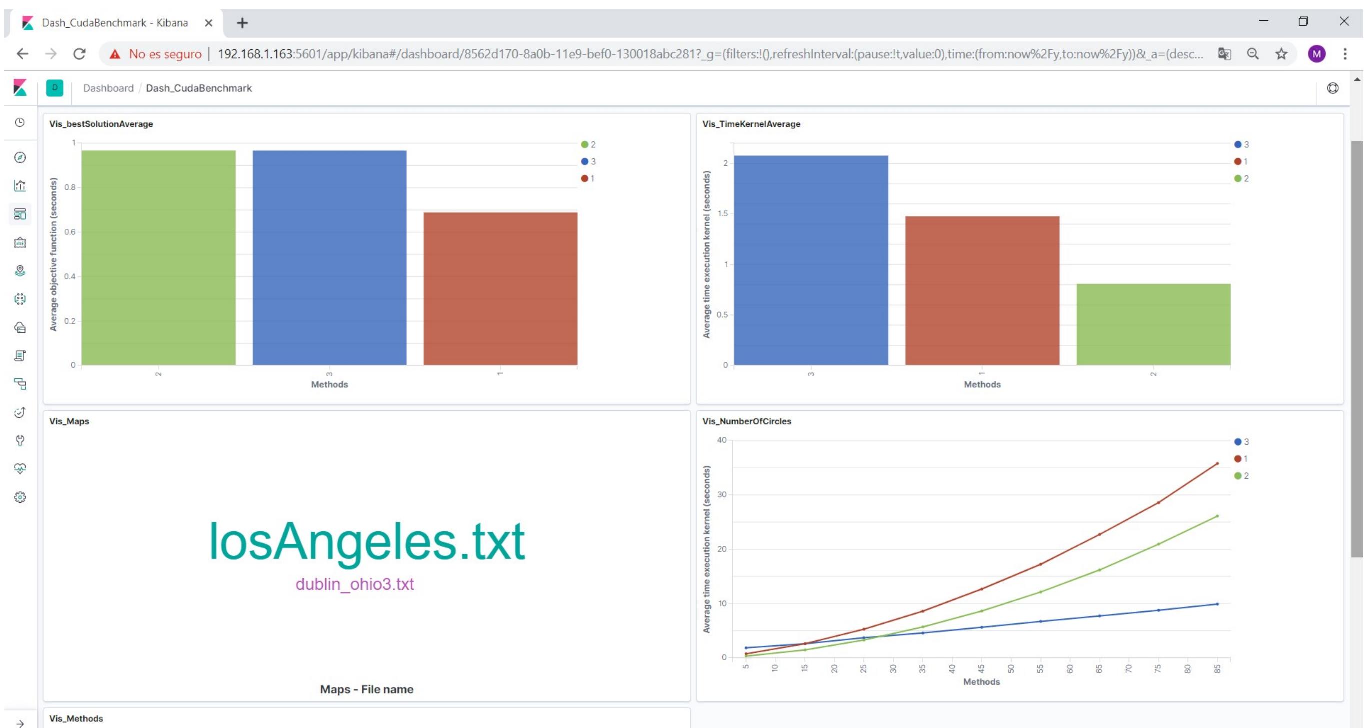


Figura 78. Menú Dashboard Kibana

14. Bibliografia

B.V., E. (2019). *Elastic*. Consultat el 3 / 2019, a Elastic: https://www.elastic.co/es/

Church, R., & ReVelle, C. (1974). The Maximal Covering Location Problem. *Papers of the Regional Science Association 32* , 101-118.

Coll, N., Fort, M., & Sellarès, J. A. (2019). On the overlap area of a disk and a piecewise circular domain. *Computers and Operations Research* , 59-73.

Daskin, M. (1983). A maximum expected covering location model: formulation, properties and heuristic solution. *Transportation Science Vol.17* , 48-70.

Garey, M., & Johnson, D. (1979). *Computers and Intractability; A Guide to the Theory of NP-Completeness.* New York: W. H. Freeman and Company.

H Osman, I., & P. Kelly, J. (1997). Meta-heuristics Theory and Applications. *Journal of the Operational Research Society (48)* , 657.

Matisziw, T., & Murray, A. (2009). Siting a facility in continuous space to maximize coverage of a region. *Socio-Economic Planning Sciences 43(2)* , 131-139.

Megiddo, N., Zemel, E., & Hakimi, L. (1983). The maximum coverage location problem. *SIAM Journal of Algebraic and Discrete Methods 4(2)* , 253-261.

Murray, A., Matisziw, T., Hu, W., & Tong, D. (2008). A Geocomputational Heuristic for Coverage Maximization in Service Facility Siting. *Transactions in GIS 12(6)* , 757-773.

NVIDIA Corporation. (7 / Maig / 2019). *CUDA Toolkit Documentation*. Consultat el Gener / 2019, a CUDA Toolkit Documentation: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Resende, M. G. (1998). Computing Approximate Solutions of the Maximum Covering Problem with GRASP. *J. Heuristics, 4(2)* , 161-177.

Sanders, J., & Kandrot, E. (2010). *CUDA by example, An introduction to general-purpose GPU programming.* Boston: Addison Wesley.

Toregas, C., & ReVelle, C. (1973). Binary Logic Solutions to a Class of Location Problems. *Geographical Analysis 5* , 145-155.

Toregas, C., Swain, R., Charles, R., & Bergman, L. (1971). The location of emergency service facilities. *Operations Research Vol.19* , 1363−1373.

White, J., & Case, K. (1974). On Covering Problems and the Central Facilities Location Problem. *Geographical Analysis Vol. 6* , 281-293.

15. Annexos

Annex A. Parallel Simulated Annealing for coverage area maximization

XVIII Spanish Meeting on Computational Geometry, Girona, July 1-3, 2019

Parallel Simulated Annealing for coverage area maximization

Narcís Coll*[1], Marta Fort†[1,2], and Moisès Saus‡[1]

[1]Graphics and Imaging Laboratory, Universitat de Girona
[2]Universitat Politècnica de Catalunya

Abstract

We present an approach to determine where to locate k disks so that they globally cover as much area of a polygonal domain as possible. The approach takes advantage of the parallel capabilities of the GPU to accelerate the process. We also present some preliminary experimental results.

1 Introduction

Location science is concerned with the placement of a limited set of facilities in order to optimize (minimize or maximize) at least one objective function: coverage, cost, travel distance, etc. A large number of problems locating facilities use the concept of coverage. A demand is covered by a facility if the distance or travel time between the demand and the facility is less than a certain predetermined value called the coverage radius. The problems of coverage are of great applicability when planning the location of facilities for both public and private sectors. Often, complete coverage of all demand in a region is not possible due to budgetary limitations on the number of facilities that can be sited. Thus, limited resources must be efficiently managed and regional demand should be covered to the greatest extent possible.

1.1 Previous work

The maximal covering location problem (MCLP), that was introduced by Church and ReVelle [5], seeks to identify the locations for a specific number of facilities in such a way that the coverage is maximum within a desired service distance. In many cases, a demand point is considered *covered* if it lies inside the disk centered on a facility with radius equal to the specified service distance. Since the MCLP is NP-Hard [10], various heuristic methods which provide approximate solutions have been proposed. There are works that consider: i) both the demand and the location of the facilities to be defined by finite sets of points; ii) the facilities to be placed in continuous space while representing demand as discrete points; iii) both the demand and location of the facilities in the continuous space. However, most of the solutions that consider a continuous domain model start partitioning the demand into several small regions which are considered to be either completely covered or uncovered by the disk. In fact, partial coverage has been studied on rectangular demand and rectangular service zones in [1], and in [7] with a single circular service zone and an unconnected domain with holes bounded by linear or linear segments. Since finding a solution to the problem in any of these cases is not trivial there are several papers that deal with the single source problem. The single and multiple source cases assuming that facilities are located in a discrete set are studied in [9] and [12], respectively. In [13], facilities can be located anywhere in the plane and the demand is present everywhere. The problem of these solutions is that they introduce significant errors because partial coverage is not taken into account in the objective function [14], and hence it does not actually maximize coverage.

In the computational geometry field, related problems have also been studied. For instances, an algorithm to compute the maximum overlap between two simple polygons P and Q with n and m vertices in $O(n^2m^2)$ time is provided in [11]. An algorithm to approximate the maximum overlap using random sampling techniques is presented in [4]. Finally, in [3] an algorithm to approximate the maximum overlap of two polygons P and Q with multiple holes in $O(n^2\varepsilon^{-3}\log^{1.5} n\log(n/\varepsilon))$ time where n denotes the total number of vertices in P and Q. If one of the two polygons is convex the running time is $O(n\log n + \varepsilon^{-3}\log^{2.5} n\log((\log n)/\varepsilon))$ and the additive error of the solution, with high probability, is $\varepsilon \cdot \text{area}(P)$.

Solving exactly the continuous maximal covering problem, even for locating a single facility, is difficult and computationally very expensive because an infinite number of locations must be considered, both for the demand of service and for the installation of the facilities. Therefore, standard optimization techniques for solving discrete location models are not applicable to the CMCP, and new efficient methods are required to solve it. Moreover, the programmabil-

*Email: coll@imae.udg.edu
†Email: mfort@imae.udg.edu
‡Email: msaus@imae.udg.edu

ity and high computational rates of graphics processing units (GPU) make them a powerful platform for computationally demanding tasks where it is needed to process a large amount of data or perform a lot of operations. Parallel processing capability of the GPU allows to split complex computing tasks into thousands of smaller tasks that can be run simultaneously. It allows to solve the problem in a fraction of the time required by the CPU. Hence, the general purpose computing on GPUs (GPGPU) has become a way to reduce execution times. It is being used by many researchers in several computational fields ranging from numerical computing operations and physical simulations to knowledge discovery, data mining and bioinformatics geometry processing [8, 2].

1.2 Problem formalization

In this paper we focus on the multiple-facility case of the continuous maximal covering problem with the assumption of uniformly distributed demand on a polygonal domain that may have holes. We consider k disk-like service areas of radius r, one for each facility. We take into account partial coverage, facilities can be located anywhere on the plane and their service areas are not necessarily completely contained within the polygonal domain. See Figure 1 for a motivational example (the disks configuration in b) is hand made).

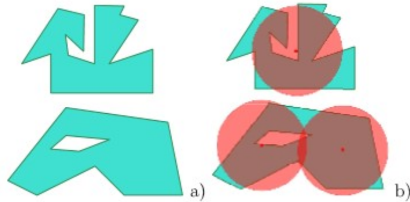


Figure 1: a) Polygonal domain to be partially covered by facilities with circular service coverage; b) Domain partially covered by three facilities

We denote by $\mathcal{P}$ the polygonal domain to be covered, by $\mathcal{B}$ the bounding box of $\mathcal{P}$, r the radius of disk-like services and by $\mathcal{B}'$ the amplified bounding box $\mathcal{B}$ with an offset of size r. Moreover, we denote by $D(c, r)$ the disk of center c and radius r and by $x = (x_1, \cdots, x_k)$ each configuration of k centers x_i with $i = 1 \dots k$.

Finding the optimal location of these k disks is an NP-hard problem. Hence, we provide an heuristic method that allows us to approximate this optimal location by maximizing an objective function whose value can be computed quickly. Thus, we consider the following objective function that potentiates the overlap between the discs and the polygon and penalizes the overlap between the discs.

$$\mathcal{O}bj(x) = \sum_{i=1}^{k} Area(\mathcal{P} \cap D(x_i, r)) - \\ - \sum_{i,j=1}^{k} Area(D(x_i, r) \cap D(x_j, r))$$

Hence, our aim is to heuristically determine the location x_{opt} that maximizes $\mathcal{O}bj$.

1.3 Simulated annealing

Simulated annealing (SA) is a heuristic technique for approximating the global minimum/maximum of a function $\mathcal{O}bj(x)$, called energy, that tries to imitate the annealing process used in metallurgy. At each step, the SA considers some new state x' close to the current state x, and probabilistically decides between staying at x or moving to x'. This step is repeated until the energy of the current state is low/great enough, or until a given the given maximum number of steps is reached. The probability of transition from x to x' depends on $\mathcal{O}bj(x)$, $\mathcal{O}bj(x')$ and a global parameter T called temperature that decreases with respect the number of steps.

1.4 Algorithm overview

The proposed algorithm in a preprocess step we compute the overlapped areas of a set of $H \times W$ basic disks of radius r in parallel. Then we obtain an initial placement x_{ini} for the k disks. Finally the initial configuration x_{ini} is evaluated by the presented objective function $\mathcal{O}bj(x)$ and perturbed in parallel. The best of the analyzed configurations is the one presented as the best location in order to cover as much as possible $\mathcal{P}$ with k disks of radius r.

2 Algorithm description

In this section we describe in detail the proposed algorithm that can be divided into two parts. An initial step where we compute the overlapped area of the $H \times W$ basic disks, and the determination of the best configuration of the k-disks.

2.1 Overlapped area computation

Computing the overlapped area between a disk and the polygonal domain is not trivial when partial coverage is taken into account. We compute it by using the algorithm presented in [6] that exactly computes the area of the intersection between a disk and a polygonal domain with holes and unconnected regions.

In the presented proposal we want to take advantage of placing k disks of the same radius r. We start computing the area of the overlap of $\mathcal{P}$ by a set of

$H \times W$ disks in an initial step that will be used, posteriorly, to infer the overlapped area of the considered disks. We use a $H \times W$ regular grid placed on the bounding box of $\mathcal{P}$ enlarged by r. We consider the centers of its cells as centers of the disks. Their covered areas are computed in parallel in the GPU at the very beginning.

We use a kernel that has as input: i) the bottom-left corner of the bounding box of $\mathcal{P}$; ii) the length and width of the grid cells; and iii) the domain $\mathcal{P}$ stored in two arrays, one with the vertices coordinates and the other describing its components and holes (see [6] for further details). This kernel is executed by a $H \times W$ grid of threads and each thread is identified by a two dimensional integer index $(id_x, id_y) \in [0, H) \times [0, W)$ that directly associates it to a cell of the considered grid. Each thread computes the center c of the cell it represents and then computes the area of $\mathcal{P}$ covered by the disk centered in c. The area is computed by using the algorithm presented in [6] and then stored in a $H \cdot W$ array associated to the grid (linearized in a row first fashion). The array of areas is the kernel output, the grid cell centers are not stored because are never used.

2.2 Best configuration determination

In order to heuristically determine where to place each one of the k we use a technique similar to the simulating annealing but analysing M configurations in parallel and computing the best of the analyzed options.

The parallel SA is done by a CUDA kernel run by a single block of M threads that uses as input a number of steps n, a configuration x_{ini}, a radius R and the temperature $T = k\pi r^2$. The radius R is used to create a new configuration by perturbing a previous one. The kernel returns as output the optimal location found x_{opt} and the maximum value of the objective function found $\mathcal{O}bj_{max}$. Then, each thread starts with $x = x_{ini}$, $x_{best} = x_{ini}$ and, at each step, a new configuration $x' = (x'_1, \cdots, x'_k)$ is computed by perturbing the configuration $x = (x_1, \cdots, x_k)$ by taking each new center x'_i as a random point in the disk $D(x_i, R)$. Then, x_{best} is updated by x' if

$$\mathcal{O}bj(x') > \mathcal{O}bj(x_{best}),$$

and x is updated by x' if

$$\mathcal{O}bj(x') > \mathcal{O}bj(x) \text{ or } p < \exp((\mathcal{O}bj(x') - \mathcal{O}bj(x))/T),$$

where p is random value between 0 and 1. Further, also at each step, the temperature T is updated by αT. Once finished the n steps the thread updates $\mathcal{O}bj_{max}$ according to $\mathcal{O}bj(x_{best})$ by performing an atomic maximum operation. The value of $\mathcal{O}bj(x_{best})$ in set to $\mathcal{O}bj_{max}$ whenever $\mathcal{O}bj(x_{best}) > \mathcal{O}bj_{max}$. Once all the threads have reached this point, i.e. after a synchronization point, the thread checks wether its $\mathcal{O}bj(x_{best})$ coincides with $\mathcal{O}bj_{max}$. In such a case it stores its id in a global integer id_{opt}. Finally, after the threads have been synchronized again, the thread with $id = id_{opt}$ stores its x_{best} in x_{opt} that stores one of the optimal configurations.

2.3 $\mathcal{O}bj$ function value computation

The $H \times W$ areas computed in the preprocess stage are stored in a CUDA-texture to take advantage of the fetching functions they support to extrapolate information from their values. CUDA-textures allow the typical fetching methods of OpenGL textures.

Hence, instead of exactly computing the value $Area(\mathcal{P} \cap D(c, r)))$ of an arbitrary disk $D(c, r)$ of center c, it is approximated extracting it from the texture. CUDA-textures allow to approximate it by $Area(\mathcal{P} \cap D(c', r))$, being c' the closest center to c among the $H \times W$ analyzed ones. But they also allow to approximate it using bilinear interpolation from the neighboring centers.

In our case the threads approximate each term in $\sum_{i=1}^{k} Area(\mathcal{P} \cap D(x_i, r))$ using bilinear interpolation from the initial areas and exactly computes the $k(k+1)/2$ intersection areas $D(x_i, r) \cap D(x_j, r)$ of each pair of the k disks involved in x.

3 Experimental results

The algorithms have been implemented in C++, for the parallel part Cuda C has been used and the visualization is done by using OpenGL. The running times presented in this section have been obtained using a Inter(R) Core(TM) i7-4790CPU with a Tesla k40 active GPU.

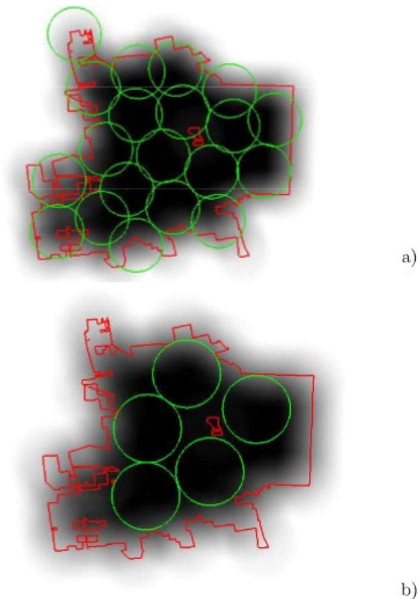
We have run the algorithm with a polygonal domain with holes defined by 488 vertices that corresponds to Dublin from Ohio. This domain has also been used in [14] and has one single component with 13 holes (see the red polygon of Figure 2, the internal polygons define the holes). The regular grid of areas computed at the preprocess stages is of 807×884 and covers the bounding box of the considered polygonal domain with an r-offset reescaled and traslated to $[-1, 1] \times [-1, 1]$.

We tested the algorithm with several parameters α, n and R (R is set taking into account that finally the domain is contained in $[-1, 1] \times [-1, 1]$) . After several tests we found that the parallel SA worked better if:

- the reduction T to αT was done every 10 steps with $\alpha = 0.95$
- it was subdivided in two phases: the first phase with $R = R_0 = 0.7 - r$ and $n = 50$ being $(0, 0)$ the center of all disks of the initial configuration,

while the second fase with $R = R_0/5$ and $n = 450$ and the initial configuration of the second phase the best configuration of the first phase.

In Figure 2a) we present the obtained solutions when trying to locate 20 disks with radius $r = 0.2$. The value of the of the objective function of the obtained solution is 1.94, meanwhile the area of the domain is 2.28. In Figure 2b) we placed only 5 disks of radius $r = 0.25$ and we obtained an optimal solution: the disks are disjoint and contained in the domain (the objective function value is 0.982 which coincides $k\pi r^2$). The time needed to obtain these solutions was 2.1(s) to obtain the solution presented in a) and 0.3(s) the one in b) once the preprocess stage was been done. In both cases computing the grid where the areas are stored took 3.0(s).



a)

b)

Figure 2: Polygonal domain partially covered by 20 and 5 disk-like facilities

4 Conclusions and further work

We presented a framework to determine the location of k disks on a bounding according to an objective function that potentiates the overlap between the disks and the polygon, but prevents from having areas covered by several disks. The presented experimental results are preliminary experimental results, the algorithm works well for sets of up to 30 circles but it has to be improved when the number of disks increases. We also want to consider disks of different radi.

References

[1] M. Bansal and K. Kianfar, Planar maximum coverage location problem with partial coverage and rectangular demand and service zones, INFORMS Journal on Computing, 29 (2017), pp. 152-169.

[2] Y. Cai, S. See (Eds.), *GPU Computing and Applications*, Springer Singapore, (2015).

[3] S.W. Cheng, C.K. Lam, Shape matching under rigid motion, *Comput. Geom. Theory Appl.* **46(6)** (2013) 591–603.

[4] O. Cheong, A. Efrat, S. Har-Peled, Finding a guard that sees most and a shop that sells most, *Discrete & Computational Geometry* **37(4)** (2007) 545—563.

[5] R.L. Church, C. ReVelle, The Maximal Covering Location Problem, *Papers of the Regional Science Association* 32 (1974) 101–118.

[6] N. Coll, M. Fort, J.A. Sellarès, Computing the maximum overlap of a disk and a polygon with holes under translation, *XVI Encuentros de Geometría Computacional* (2015) 57–60.

[7] N. Coll, M. Fort, J.A Sellarès, On the overlap area of a disk and a piecewise circular domain, *Computational and operational research* **104** (2019) 59–73.

[8] W.W. Hwu (Ed.), *GPU Computing Gems Emerald Edition*, Morgan Kaufmann, (2011).

[9] T. C. Matisziw, A. T. Murray, Siting a facility in continuous space to maximize coverage of a region, *Socio-Economic Planning Sciences* **43** (2009) 131–-139.

[10] N. Megiddo, E. Zemel, S.L. Hakimi, The maximum coverage location problem, *SIAM Journal of Algebraic and Discrete Methods* **4(2)** (1983) 253–261.

[11] D.M. Mount, R. Silverman, A.Y. Wu, On the area of overlap of translated polygons, *Computer Vision and Image Understanding* **64(1)** (1996) 53–61.

[12] A.T. Murray, D. Tong, Coverage Optimization in Continuous Space Facility Siting, *Int. J. Geogr. Inf. Sci. Vol.* **21(7)** (2007) 757–776.

[13] A.T. Murray, T.C. Matisziw, H. Wei, D. Tong, A Geocomputational Heuristic for Coverage Maximization in Service Facility Siting, *Transactions in GIS* **12(6)** (2008) 757–773.

[14] R. Wei, A.T. Murray, Continuous space maximal coverage: Insights, advances and challenges, *Computers and Operations Research* **62** (2014) 325—336.