Kételektron-integrálok wavelet alapú számítása párhuzamos programozás segítségével

TDK DOLGOZAT

Készítette
Horváth Kristóf Attila

Konzulens
Szegletes Luca

2013. október 25.

Tartalomjegyzék

Kivonat

Waveletek használatával a Coulomb kölcsönhatás kételektron-integráljainak számítása sokkal gazdaságosabban lehetséges az atomi bázisfüggvények (LCAO) alapú számítással szemben. Az eljárás egy általános, atompálya független, ebből kifolyólag hatékonyabb megoldást eredményez.

Mindazonáltal ezen integrálok kiszámítása rendkívül idő- és CPU igényes feladat. A futásidő csökkentése érdekében párhuzamos számítás alkalmazható. A párhuzamos feldolgozás két formájára tér ki a dolgozat: az egyik lehetőség a GPU (Graphics Processor Unit) alapú számítás, míg a másik önkéntes számítási projekt használata. A két módszer rendkívül eltérő, így különböző követelményeket támasztanak az algoritmussal szemben.

A dolgozat áttekinti a CUDA párhuzamos számítás platformot, illetve a BOINC (Berkeley Open Infrastructure for Network Computing) rendszert. Bemutatásra kerülnek a kidolgozott algoritmusok csakúgy, mint platform-függő sajátosságaik illetve korlátjaik.

Abstract

Using a wavelet approach, the evaluation of two-electron integrals of the Coulomb interaction potential can lead to a much more economical scheme compared to the atomic basis function (LCAO) based calculations. The method gives a general, system independent, therefore more efficient solution.

However, the calculation of such integrals is a highly time-consuming and CPU-intensive task. To reduce the run time, parallel processing techniques are applicable. Two ways of parallelism are proposed in the paper. One option is the calculation on graphics processor unit (GPU), another opportunity is using volunteer computing. The two methods are very distinct; there are different requirements for the algorithm.

The paper presents a comparative study, where two different methods are introduced: the parallel computing platform called CUDA and the Berkeley Open Infrastructure for Network Computing (BOINC). By designing the final algorithms I considered the specific features and limitations of these platforms.

Első fejezet

Bevezető

1.1. Az alapprobléma

A molekulapályák klasszikus számításához a Hartree–Fock módszert alkalmazzák. A hullámfüggvényeket az LCAO (Linear Combination of Atomic Orbitals) módszerrel írják le, azaz a molekulapálya hullámfüggvénye atompályák hullámfüggvényeinek lineáris kombinációjaként állítható elő.

A Hartree–Fock módszer egy iterációs eljárás, melynek során minden lépésben kételektron-integrálok számítása szükséges, melyek az LCAO bázisfüggvényei miatt egy adott atomra jellemző atompályáktól függnek, így a kételektron-integrálokat minden iterációs lépésben újra és újra ki kell számolni, ami számításigényes művelet.

1.2. Waveletek

A fenti technika számításigényét az okozza, hogy a kételektron-integrálok atomi bázisfüggvényektől függnek. Waveletek alkalmazásával bázisfüggvényeknek a skálafüggvények választhatók, ennek következtében egy univerzális, atompályafüggetlen módszert kapunk, a kételektron-integrálok egyszeri kiszámítása után azok minden molekulánál alkalmazhatók. [1] [2]

Ezen okból kifolyólag az eljárás rendkívül hatékony, hiszen a molekulapályák számításakor a kételektron-integrál eredmények táblázatból olvashatók ki. Ehhez azonban szükséges a integrál-eredmények előzetes egyszeri előállítása, mely rendkívül erőforrásigényes művelet, ennek implementálásával, optimalizálásával foglalkozik a dolgozat.

A módszer használatához az alábbi kételektron-integrálokat kell kiszámítani:

$$I_{abc}^{(m)} = 2^m \int \int dr_1'' dr_2'' \, \frac{s(r_1'')s(r_1''+a)s(r_2'')s(r_2''+b)}{|r_1''-r_2''+c|} \tag{1.1}$$

Tekintettel arra, hogy a skálafüggvények kompakt tartójúak, az integrálok eredménye legtöbbször nulla. Továbbá a nem nulla eredményű integrálok legtöbbjére jó közelítés adható, így csak egy kis részüket szükséges numerikus módszerekkel kiszámítani.

A vektorokat kifejtve ($r''_1 = (n''_1, m''_1, o''_1)$, $r''_2 = (n''_2, m''_2, o''_2)$, $a = (a_1, a_2, a_3)$, $b = (b_1, b_2, b_3)$ és $c = (c_x, c_y, c_z)$) az alábbi alakot kapjuk:

$$I_{abc}^{(m)} = 2^m \int\int\int\int\int\int dn''_1 dm''_1 do''_1 dn''_2 dm''_2 do''_2 \\ \frac{s(n''_1)s(n''_1+a_1)s(m''_1)s(m''_1+a_2)s(o''_1)s(o''_1+a_3)s(n''_2)s(n''_2+b_1)s(m''_2)s(m''_2+b_2)s(o''_2)s(o''_2+b_3)}{\sqrt{(n''_1-n''_2+c_x)^2+(m''_1-m''_2+c_y)^2+(o''_1-o''_2+c_z)^2}} \tag{1.2}$$

1.3. Integrálok számítása

A dolgozat további része az előbb ismertetett integrálok számításával foglalkozik. A második fejezetben a hatalmas számításigény miatt felmerülő párhuzamos programozás, illetve a megvalósítás során alkalmazott platformok kerülnek áttekintésre. A harmadik fejezetben az integrálokat a különböző platformokon kiszámító algoritmusokat mutatom be, majd a negyedik fejezetben az implementáció során alkalmazott megfontolások, illetve a futásidők és az eredmények bemutatása történik. A dolgozatot az értékelés és ennek alapján a folytatás lehetőségének ismertetése zárja.

Második fejezet

Párhuzamos számítás

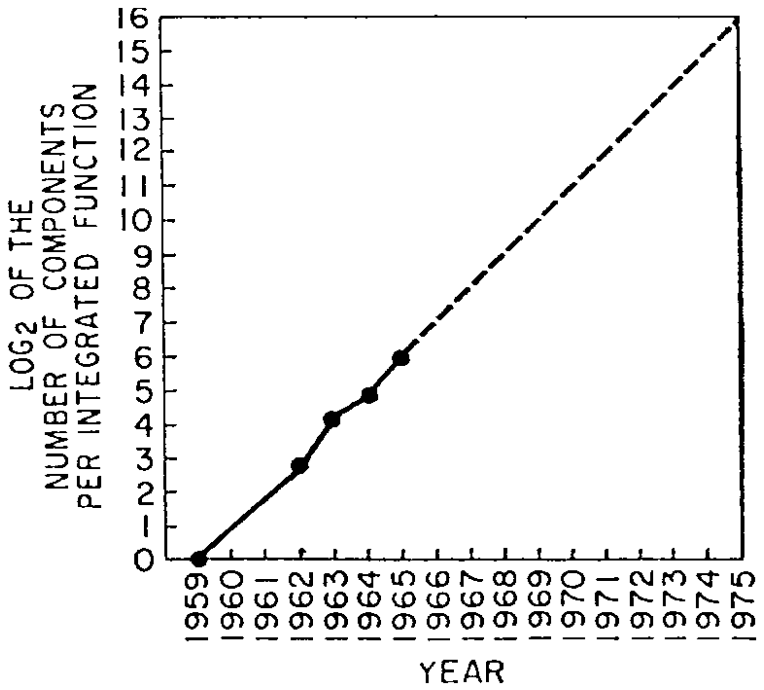
Az egyszálú műveletvégzés sebességnövelésének akadályai miatt egyre inkább előtérbe kerülnek a párhuzamos számítást alkalmazó különféle megoldások. Jelen projekt keretein belül a CUDA grafikus kártya programozási platformot, illetve a BOINC önkéntes számítási projektet alkalmaztam a korábban bemutatott probléma megoldásához.

A fejezet bemutatja a párhuzamos számítás elterjedésének motivációit, a párhuzamos programozás néhány alapvető különbségét az egyszálú programozáshoz képest, majd részletesen áttekinti a CUDA platformot és a BOINC rendszert.

2.1. A párhuzamos számítás motivációja

Gordon Moore, az Intel egyik alapítója, 1965-ben felfedezte, hogy az integrált áramkörökön található tranzisztorok száma minden évben megduplázódott az integrált áramkörök 1958-as felfedezése óta. Megfigyelését az Electronics Magazine hasábjain publikálta, illetve megjósolta a trend folytatódását is, ahogyan az a 2.1 ábrán látható. A jóslat megalapozottnak bizonyult – az exponenciális növekedés mind a mai napig megfigyelhető. [3]

Moore törvénye nem csak a tranzisztorok számára, hanem a processzorok sebességére illetve a memóriakapacitásra is alkalmazható (a duplázódás időtávjának más-más egységeket választva). Azonban a processzorok sebességnövekedésének tartása manapság különleges kihívások elé állítja a gyártókat. Az órajel sebességének növekedése néhány évvel ezelőtt megállt. Habár az integrált áramköri chipek mérete rendkívül apró, a jelterjedési idők a ma alkalmazott órajel-frekvenciák mellett már nem elhanyagolhatók. Továbbá a chip méreteinek csökkentése is egyre nehezebb: a csíkszélesség egyre inkább közelít az atomi korlátokhoz, illetve az alkatrészsűrűség növekedésével a disszipáció-sűrűség is nő, ennek következtében a megfelelő hűtés megvalósítása is egyre nehezebb, így a további sebességnövekedést gátló tényező. Érdekes módon Moore is felveti a hőtermelés problémáját, azonban a disszipáció-sűrűséget figyelmen kívül hagyta, így mai szemmel vizsgálva ebben a kérdésben hibás konklúzióra jutott. [4]



2.1. ábra. *Gordon Moore megfigyelése és jóslata az integrált áramkörökön található tranzisztorok számára vonatkozólag [3]*

Ezek a problémák más megoldások alkalmazására sarkallták a kutatókat és mérnököket a számítási sebesség növelésének fenntartása érdekében. Egy ilyen megoldás a párhuzamos számítás – ennek során több számítóegységet alkalmazunk, melyek egymástól függetlenül képesek működni (de természetesen együttműködve, ha erre szükség van). Ilyen számítóegység egy többmagos processzor, egy videokártya, vagy egy önkéntes számítási projektben résztvevő számítógép-hálózat – ez a dolgozat utóbbi kettőt tárgyalja. Az ekképpen megvalósított rendszerre történő programozás módszertanával az úgynevezett párhuzamos programozás foglalkozik.

2.2. Párhuzamos programozás [5]

A párhuzamos programozás régóta kutatott terület, ugyanis egyprocesszoros rendszerekben is lehet létjogosultsága. Multitasking alkalmazásával a különböző feladatok logikailag párhuzamosan futnak, azonban a párhuzamosság ez esetben csak virtuális – az ütemező rövid időszeleteket oszt a feladatoknak, melyek lejártakor taszkváltás segítségével történik a vezérlés átadása az addig várakozó állapotban lévő feladatnak. A taszkváltás során biztosítani kell, hogy az épp megszakított feladat a későbbiekben folytatható legyen, így állapotváltozóit menteni kell, illetve az új feladat számára vissza kell tölteni korábbi állapotát. Ez viszonylag erőforrásigényes művelet, az ütemező feladata, hogy optimumot találjon a taszkváltás gyakoriságát illetően.

2.2.1. Kommunikáció

Bizonyos problémák úgy dekomponálhatók párhuzamos részfeladatokra, hogy azok teljesen különállóan képesek futni, csak a bemeneti és kimeneti adatok cseréjére van szükség a folyamatokkal. Számtalan párhuzamosítható probléma azonban nem dekomponálható ilyen mélységben, és szükséges a részfeladatok közötti kommunikáció megvalósítása.

Több lehetőség is a programozó rendelkezésére áll a folyamatok közötti kommunikáció implementálására. Bizonyos modellekben a kommunikáció explicit, más modellekben implicit formában van jelen. A tervezés során figyelembe kell venni a kommunikáció erőforrásigényét, optimumot kell találni az üzenetek adatsebessége és késleltetése között, stb.

A kommunikáció megvalósulhat szinkron vagy aszinkron módon is. Szinkronizált kommunikáció a folyamatokat blokkolja az információcsere lezajlásáig, így kevésbé hatékony megoldás. Aszinkron esetben nem történik blokkolás, így a folyamat tovább futhat, azonban ez esetben némileg körülményesebb megállapítani az információ beérkezésének tényét, majd feldolgozni azt.

2.2.2. Szinkronizáció

A kommunikációhoz hasonlóan a folyamatok szinkronizálása is többféleképpen valósulhat meg.

- Várakozópont
- Zárolás, szemafor
- Szinkronizált kommunikáció

Várakozópont beiktatása esetén az adott folyamat a várakozóponthoz érve felfüggeszti működését, és várakozik mindaddig, míg a többi folyamat is elér a várakozóponthoz. A metódus olyan esetekben alkalmazható a legjobban, amikor a párhuzamos kódrészlet befejeződése után egy soros kódrészletnek kell lefutnia.

Szemafor alkalmazásával leggyakrabban a közös adatokhoz történő hozzáférést szabályozzák, segítségével megvalósítható a kölcsönös kizárás. Az közös adatokkal történő manipulációt tartalmazó kódrészletet „kritikus szakaszként” kell kezelni, melyet adott időben mindig csak egy folyamat hajthat végre. A vasúti irányítóberendezés emlékét őrző szemafor a kritikus szakaszba történő belépést szabályozza, amennyiben az egyik folyamat belépett a kritikus szakaszba, más folyamatnak nem engedi a szakasz futtatását.

Korábban láttuk, hogy a szinkronizált kommunikáció blokkolja a folyamatok működését, így ez a metódus is alkalmazható szinkronizációra.

2.2.3. Modellek

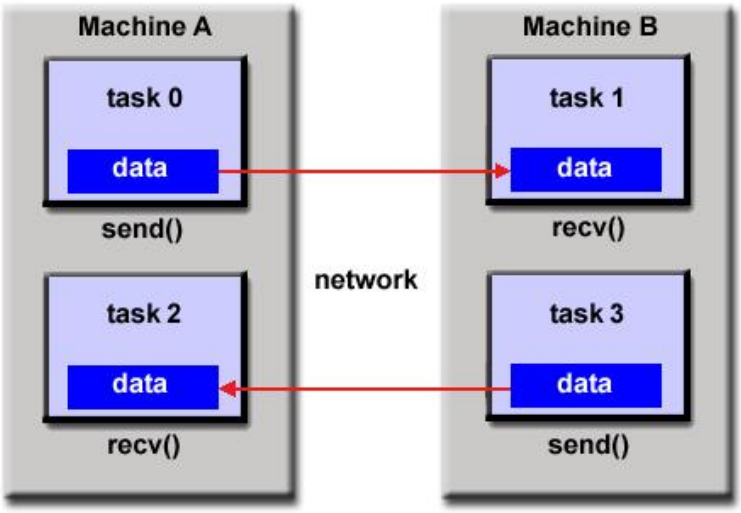
A párhuzamos programozás különböző modellek szerint valósítható meg:

- Közös memória modell
- Szál modell
- Üzenetváltás alapú modell

Közös memória modellben a feladatok egy közös memóriaterületen osztoznak, melyet aszinkron módon olvasni és írni is képesek. A közös memória írása inkonzisztens állapotokat idézhet elő, így az írás szabályozására szemaforok, zárolások alkalmazandók. A modell előnye, hogy a feladatok közötti adatcsere egyszerű, azonban hátránya, hogy a zárolások megfelelő alkalmazása bonyolulttá teszi a programozást, az inkonzisztens állapotba kerülés veszélyén túl a rosszul megválasztott zárolási stratégia jelentősen csökkentheti a teljesítményt, illetve holtpont is kialakulhat.

A szál modell a közös memória modell egy speciális típusa. Ebben az esetben elkülöníthető egy főszál, illetve több mellékszál, melyek párhuzamosan futnak. A főszál kommunikál a külvilággal, kezeli az erőforrásokat. Amikor egy párhuzamosítható rész következik, mellékszálakat hoz létre, melyek párhuzamosan futnak. A szálak saját lokális memóriával rendelkeznek, azonban elérik a főszál globális memóriáját is, mely lehetőséget nyújt az előbbi modellben bemutatott közös memóriakezelésre is. Szál modellt alkalmaz a párhuzamosítás során a POSIX Threads vagy az OpenMP.

Üzenetváltás alapú modell alkalmazása esetén a feladatok saját, lokális memóriájukat használják a számításokhoz. A feladatok közötti kommunikáció üzenetek küldésével és fogadásával valósul meg, mint azt a 2.2 ábra szemlélteti. Üzenetváltás alapú modellt valósít meg az MPI (Message Passing Interface) is.



2.2. ábra. *Feladatok közötti kommunikáció üzenetekkel [5]*

2.3. CUDA [6] [7]

Egy speciális formája a párhuzamos programozásnak GPU (Graphics Programming Unit), azaz videokártya használata. A GPU-kat eredetileg valósidejű video-rendereléshez fejlesztették ki, azonban képességeik alkalmassá teszik azokat számításigényes, általános problémák megoldására is. A videokártyák a játékipar nyomásának hatására elképesztő ütemben fejlődnek, ami az általános célú grafikuskártya programozás (GPGPU: General-purpose computing on graphics processing units) révén a tudományos számításoknál is kihasználható. A grafikus kártyák több száz, vagy akár több ezer magot is tartalmazhatnak, ez rendkívül gyors számítást tesz lehetővé, ha a probléma jól párhuzamosítható. [6]

A CUDA (Compute Unified Device Architecture) egy programozási platform és modell, melyet 2006-ban mutatott be az NVIDIA. A CUDA architektúra három egységet tartalmaz:

- GPU segítségével gyorsított programkönyvtárak
- OpenACC direktívák
- Programnyelv kiterjesztések

A felsorolásban egymást követő elemek segítségével egyre komplexebb feladatok oldhatók meg GPU használatával, azonban alkalmazásuk is egyre bonyolultabb. A GPU-ra írt programkönyvtárak függvényeinek alkalmazása teljesen elrejti a programozó elől a párhuzamos programozás feladatait, csupán a megfelelő függvényeket kell kiválasztani és meghívni. Ilyen függvénykönyvtárak a lineáris algebra problémákat megoldó cuBLAS és cuSPARSE, vagy a jel- és képfeldolgozásban alkalmazható NPP és cuFFT, de számtalan egyéb függvénykönyvtár is rendelkezésre áll.

Az OpenACC direktívák segítségével egy – a programozó által megírt – program bizonyos részeinek automatikus párhuzamosítása, és GPU-n történő futtatása végezhető el. Például egy *for* ciklus egyszerűen szervezhető a GPU-ra a direktívák segítségével.

Legkomplexebb megoldásként a programozó saját maga kezelheti a grafikus kártyát, menedzselheti a videokártya memóriáját, illetve saját függvényeket hívhat azon. C/C++, Fortran, Python és .NET nyelvekhez biztosít támogatást a CUDA.

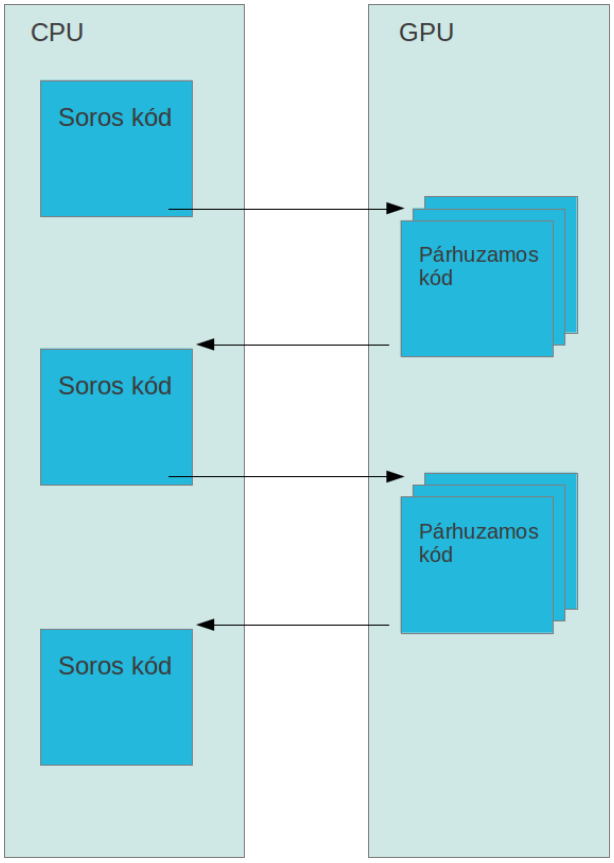
2.3.1. Programozási modell

Élesen elkülönül egymástól a CPU és a hozzá tartozó memória – melyet „host” néven illet a terminológia –, illetve a GPU és memóriája („device”). Ennek fényében a CUDA heterogén programozási modellt alkalmaz. Két kódtípus különíthető el: a soros kód, mely a host-on, illetve a párhuzamos kód, mely a device-on fut. Attól függően, hogy egy függvény milyen direktívával lett deklarálva, az egyik vagy a másik kódtípushoz fog tartozni.

Alapvetően az alábbi metódus alapján történik egy párhuzamos kódrészlet futtatása a CUDA platformon:

1. Bemeneti adatok másolása a host memóriájából a device memóriájába
2. Futtatható függvény betöltése a device-ra, majd a függvény futtatása párhuzamosan
3. Az eredmények másolása a device memóriájából a host memóriájába

A heterogén programozási modellt szemlélteti a 2.3 ábra.



2.3. ábra. *Soros és párhuzamos kódblokkok futása*

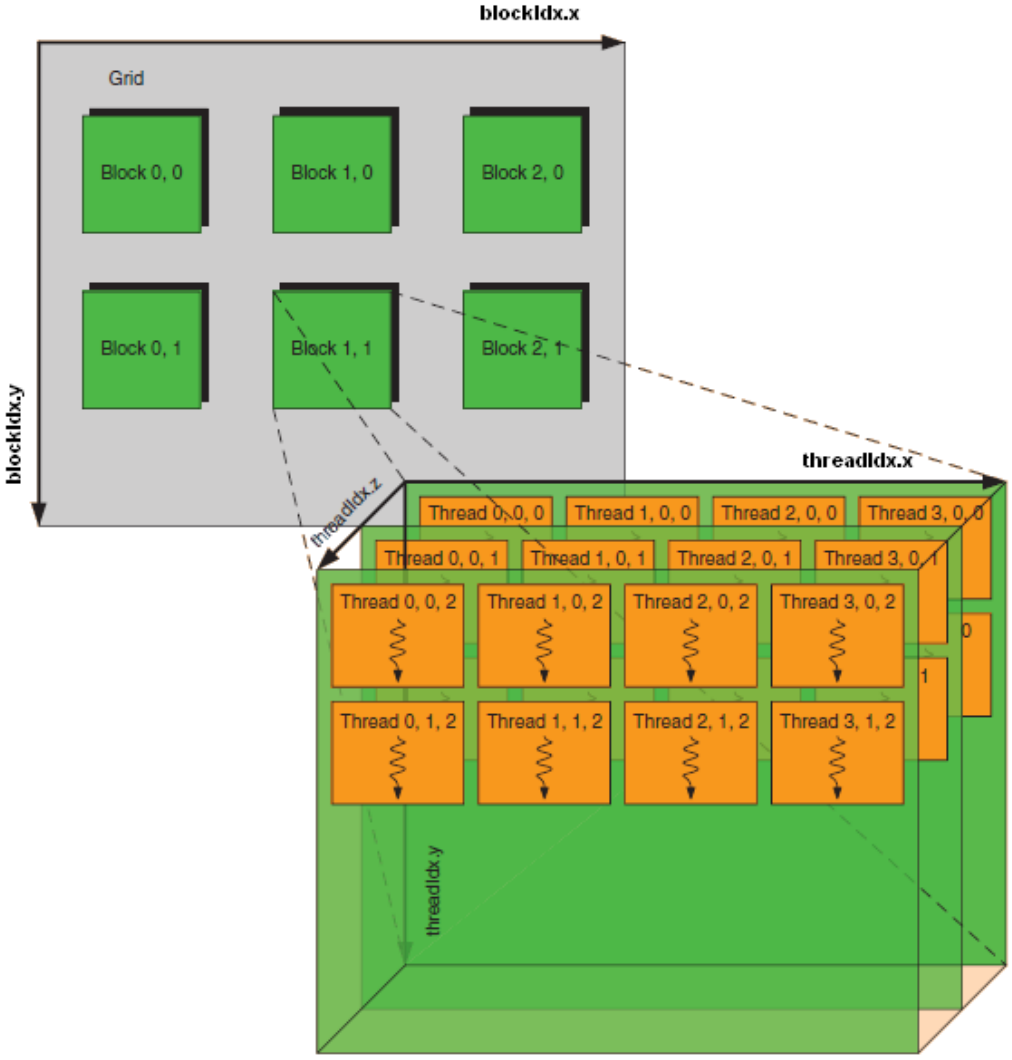
A device memóriakezelő függvényei hasonló nevűek és hasonlóképpen implementáltak, mint a host-é (*cudaMalloc*, *cudaFree*, *cudaMemcpy*). A *cudaMemcpy* függvény segítségével lehetséges a két memória között is adatokat mozgatni, ez teszi lehetővé az első és harmadik pont végrehajtását.

2.3.2. Blokkok és szálak

Két mechanizmuson keresztül is megvalósul párhuzamosítás a CUDA platformon: a device függvények blokkokban vagy szálakban futtathatók (illetve ezek kombinálása is lehetséges). A blokkok különálló GPU magokon futnak. Amennyiben nem áll rendelkezésre annyi mag, mint ahány blokkot adott pillanatban futtatni szükséges, a blokkok a magokon szekvenciálisan, egymás után futnak le. A szálak egy magon futnak, így képesek egymással kommunikálni közös memóriát használva, illetve szinkronizálható is a működésük. A CPU

taszkváltással ellentétben a GPU szálak közötti váltás rendkívül kis erőforrásigényű művelet, mivel a szálak állapotváltozói erre a célra elkülönített regiszterekben tárolódnak, így nincs szükség az állapot mentésére szálváltáskor.

Egy device függvényben az aktuális blokk, illetve szál sorszáma a *blockIdx* és *threadIdx* objektumok segítségével kérdezhető le, míg a létrehozott összes blokk és a blokkonkénti összes szál a *gridDim* és *blockDim* objektumok által. A blokkok legfeljebb kétdimenziós tömbbe szervezhetők, minden blokkban létrehozhatók szálak, melyek a blokkokkal ellentétben akár háromdimenziós tömbbe is szervezhetők. A blokkokkal szemben azonban az egy blokkon belül indítható szálak száma (a tárolótömb dimenziójától függetlenül) limitált, amit a korábban említett dedikált regiszterek végessége indokol. A blokkok és szálak elrendeződését a 2.4 ábra szemlélteti.



2.4. ábra. *Blokkok és szálak [8]*

A fentiek értelmében egy device függvény hívásakor meg kell adnunk, hogy hány darab blokkban és ezeken belül hány darab szálon fusson le a függvény. Erre a CUDA egy speciális szintaktikát alkotott, a függvénynév után, de még az argumentumlista előtt tripla kacsacső-

rök között meg kell adni a blokkok és szálak számát, például: *function<<<i,j>>>()*, ahol i a blokkok, és j az azokon belüli szálak száma. Többdimenziós blokk- és szálstruktúrák létrehozásához a CUDA beépített objektumokat biztosít.

A szálak közötti kommunikációt lehetővé tévő közös memóriát a *__shared__* direktívával lehet deklarálni, míg a szinkronizáció legegyszerűbben a *__syncthreads()* függvény meghívásával történhet, mely a 2.2.2 fejezetben bemutatott várakozópontot implementálja, így a blokkban található összes szálat összeszinkronizálja.

2.4. BOINC [9] [10]

A BOINC (Berkeley Open Infrastucture for Network Computing) egy nyílt forráskódú keretrendszer önkéntes számítási projektben, vagy griden történő számításhoz. A két metódus közötti különbség, hogy míg az első esetben a világ bármely pontján lévő önkéntes számítógépe csatlakozik a számítási hálózathoz, utóbbi esetben az elosztott számítást végző számítógépek egy vállalat, egyetem, stb. belső hálózatának részei. Griden végzett számítás előnyei, hogy a beérkező eredmények megbízhatónak tekinthetők, illetve nem szükséges az önkéntesek érdeklődését felkeltő hozzáadott értéket nyújtani (például a számítás alapján valamilyen ábrát renderelő képernyővédőt). Hátránya azonban, hogy a rendelkezésre álló számítóegységek száma jelentősen limitált.

A BOINC rendszert David Anderson vezetésével a kaliforniai Berkeley egyetemen fejlesztették, első verziója 2002-ben jelent meg. A platformot eredetileg a SETI@home projekt támogatásához alakították ki, mellyel rádióteleszkópok által vett jeleket elemeztek önkéntesek számítógépein, földönkívüli intelligencia nyomát kutatva. A BOINC később számtalan más projekt platformjává vált, többek között a LIGO (Laser Interferometer Gravitational-Wave Observatory) adatsorait elemző Einstein@home, a CERN LHC detektorának adatait elemző LHC@home, a fehérje-szerkezeteket számító Rosetta@home, vagy a Tejútrendszer dinamikai modelljét pontosító Milkyway@home.

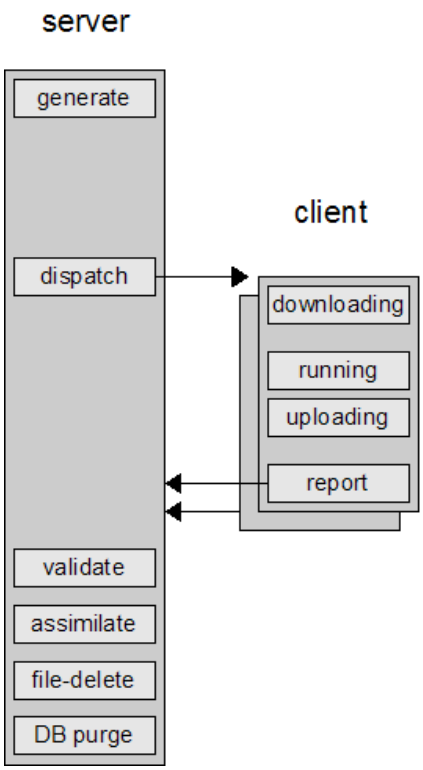
A BOINC a legnagyobb önkéntes számítási projektekhez használható keretrendszer, projektjei körülbelül 600 000 aktív számítógépen közel 10 petaFLOPS számítási teljesítményt vesznek igénybe. A keretrendszer széleskörű elterjedésének egyik kulcsa, hogy Microsoft Windows, Mac OS X, Android és GNU/Linux operációs rendszerekhez is támogatást nyújt.

2.4.1. Feladatok életciklusa

A platform architektúrája szerver-kliens modellt alkalmaz. A számítási feladatot a programozó részfeladatokká dekomponálja, mely részfeladatokat a szerver algoritmikusan képes generálni. A feladatokat letöltik a kliens számítógépek, kiszámolják az eredményt, majd a feltöltött adatokat a szerver dolgozza fel. Részletesebben az alábbi folyamat zajlik le, melyet a 2.5 ábra szemléltet.

1. A szerver generátora egy részfeladatot generál.

2. A részfeladat egy- vagy több példányban kiküldésre kerül a kliensekhez.
3. A kliens letölti a bemeneti adatokat.
4. A kliens futtatja a feladatot, majd feltölti az eredményt.
5. A kliens jelenti a feladat befejeződését.
6. A szerver validálja az eredményt.
7. Az asszimilátor feldolgozza a részfeladat eredményét (például eltárolja egy adatbázisban további felhasználás céljára).
8. Törlődnek a részfeladathoz tartozó be- és kimeneti fájlok, illetve adatbázisbejegyzések.



2.5. ábra. *Egy feladat életciklusa BOINC rendszerben [10]*

2.4.2. Eredmények validálása

Az önkéntes számítási projektben résztvevő számítógépektől beérkező eredmények nem megbízhatóak, hibás számítási értékek adódhatnak. Ennek két oka lehetséges:

- Az önkéntes számítógépe hardverhibás, különösen lebegőpontos műveletvégzésnél adódhatnak ebből problémák.
- Az önkéntes rosszindulatúan hibás eredményt küld vissza.

Akár az egyik, akár a másik okból kifolyólag érkezik hibás eredmény, ez ellen védekezni kell, az eredmények ellenőrzését a validátor végzi. A validálásnak két formája lehetséges: az egyik esetben minden részfeladat csak egyszer kerül végrehajtásra, a beérkező eredményt valamilyen alkalmazás-specifikus kritérium alapján fogadja el a rendszer. Gyakran azonban nincs mód a hibás eredmények nagy pontosságú szűrésére algoritmikusan, ez esetben a részfeladatok többszöri kiküldése, és az eredmények összehasonlítása segíthet. Ennek a módszernek a nehézsége lebegőpontos műveletek alkalmazása esetén mutatkozik meg, ahol az eredmények függnek az adott architektúrától. Ilyenkor az eredmények különbsége vizsgálható, elfogadható az eredmény, ha a különbség küszöbérték alatti.

A többszöri küldés legegyszerűbb implementálása, ha minden részfeladatot n példányban küld ki a rendszer. Ez azonban rendkívül erőforráspazarló megoldás. Már $n = 2$ esetben is 50%-ra esik vissza a CPU idő effektív kihasználása. A BOINC támogatja az adaptív többszörözést, ennek segítségével a többszörözésből származó erőforrásigény akár a tizedére csökkenhet.

Az eljárás alapja, hogy a kliensektől visszaérkező eredmények alapján megbecsüljük megbízhatóságukat. Jelölje E_i az i-edik kliens hibázási mérőszámát. A mérőszám a visszaérkező eredmények függvényében a következőképpen alakul:

- E_i kezdeti értéke $0,1$
- Minden helyes (többszörözéssel helyesnek validált) eredmény hatására $E'_i = E_i \times 0,95$
- Minden helytelen (többszörözéssel helytelennek minősített) eredmény hatására $E'_i = E_i + 0,1$

A mérőszám helytelen eredmények hatására gyorsan növekszik, a helyes eredmények azonban csak lassan csökkentik. A hibázási mérőszám segítségével a feladat kiküldése a következőképpen alakul:

- A klienstől beérkező kérelemnél az ütemező megvizsgálja a kliens hibázási mérőszámát. Ha $E_i > \varepsilon$, a kliens nem megbízható, és a feladat egy másik kliensnek is kiküldésre kerül. (ε alapértelmezett értéke $0,05$)
- Ha a kliens megbízható, ellenőrzése céljából $\sqrt{E_i/\varepsilon}$ valószínűséggel ez esetben is duplikálódik a feladat.

2.4.3. Kód hitelesítés

Ha egy támadónak sikerül a szerver és kliens közé ékelődnie, saját futtatható állományt küldhet a kliens számára, mely adott esetben kártékony, rosszindulatú kódot is tartalmazhat. Ez ellen a BOINC nyilvános kulcsú titkosítással védekezik. A szerver a futtatandó állományt privát kulcsának segítségével aláírja, az aláírás a publikus kulcs segítségével ellenőrizhető, így a kód forrásának hitelessége biztosítható.

A privát kulcs nyilvánosságra kerülése hatalmas biztonsági kockázatot rejt magában, hiszen a kulcs segítségével a támadó hitelesített kódot juttathat a klienshez. Egy ilyen jellegű támadás az adott projekt végét jelentheti, illetve dehonesztálóan hathat magára a BOINC rendszerre is. A rendszer fejlesztői egy dedikált „aláíró" számítógép használatát ajánlják, mely nem csatlakozik a hálózatra, és azon csak a futtatható állományok hitelesítése történik, így a privát kulcs kikerülésének valószínűsége minimálisra csökken.

2.4.4. BOINC API

Alapfunkciók

A BOINC platform használatához a rendszert a *boincinit()* függvénnyel kell inicializálni. A számítási feladat befejeződésekor a *boinc_finish(int status)* függvény hívása indítja a feltöltési és jelentési procedúrát, az *exit()* hatására az alkalmazás újraindul, ami általában nem a megkívánt működés.

A be- és kimeneti fájlok logikai neve, és a fájlstruktúrára ténylegesen leképződő fizikai nevek közötti névfeloldást a *boinc_resolve_filename* függvény végzi el.

A felhasználó a számítási feladatból elvégzett részről a *boinc_fraction_done(double fraction_done)* függvény segítségével tájékoztatható, mely beállítja a kliens programban a folyamatjelző sávot.

Állapotmentés

A kliensek számítógépén futtatott feladatok általában hosszú futásidejűek, azonban a számítógép kikapcsolása, vagy más programok CPU igénye megszakíthatja a futó számítást, ezért időről-időre a program állapotát menteni kell, hogy újraindulás után ne kelljen a számítást elölről kezdeni.

A programozó feladata a program állapotának mentésére, illetve annak újraindulás utáni visszaállítására szolgáló mechanizmus implementálása, azonban a BOINC API bizonyos függvényei támogatják ezt a feladatot. A *boinc_time_to_checkpoint()* ellenőrzi a legutóbbi állapotmentés óta eltelt időtartamot, így használatával elkerülhető a túl sűrű állapotmentés okozta teljesítménycsökkenés. A metódus alapjául szolgáló időzítő újraindítása a *boinc_checkpoint_completed()* függvény segítségével lehetséges.

Az állapotmentéshez az állapotot tároló leírófájl atomi írása szükséges, ellenkező esetben az állapotmentés közben történő programmegszakítás inkonzisztens állapotot idézhet elő, ami miatt az állapot visszaállítása esetleg nem lehetséges, vagy hibás állapotba kerül a program. A problémát az *MFILE* osztály oldja meg, mely a fájlba írt adatokat a memóriában puffereli, és csak a *close()* vagy *flush()* függvényeinek hatására írja lemezre.

Harmadik fejezet

Algoritmusok

Ebben a fejezetben a probléma megoldására létrehozott algoritmusaimat mutatom be. Az 1.2 képlet numerikus integrálása az alábbi formulához vezet:

$$I_{abc}^{(m)} = \sum_{n_1=1}^{n_{1max}} \sum_{m_1=1}^{m_{1max}} \sum_{o_1=1}^{o_{1max}} \sum_{n_2=1}^{n_{2max}} \sum_{m_2=1}^{m_{2max}} \sum_{o_2=1}^{o_{2max}} \frac{s(n_1)s(n_1+a'_1)s(m_1)s(m_1+a'_2)s(o_1)s(o_1+a'_3)s(n_2)s(n_2+b'_1)s(m_2)s(m_2+b'_2)s(o_2)s(o_2+b'_3)}{\sqrt{(n_1dr-n_2dr+c_x)^2+(m_1dr-m_2dr+c_y)^2+(o_1dr-o_2dr+c_z)^2}} \tag{3.1}$$

Párhuzamosítás nélkül 6 egymásba ágyazott *for* ciklussal számítható ki az eredmény, így az algoritmus pszeudokódja az alábbi:

$sum_{n_1} \leftarrow 0$
for $n_1 \leftarrow 0$ to n_{1max} **do**
  $sum_{m_1} \leftarrow 0$
  for $m_1 \leftarrow 0$ to m_{1max} **do**
    $sum_{o_1} \leftarrow 0$
    for $o_1 \leftarrow 0$ to o_{1max} **do**
      $sum_{n_2} \leftarrow 0$
      for $n_2 \leftarrow 0$ to n_{2max} **do**
        $sum_{m_2} \leftarrow 0$
        for $m_2 \leftarrow 0$ to m_{2max} **do**
          $sum_{o_2} \leftarrow 0$
          for $o_2 \leftarrow 0$ to o_{2max} **do**
            $den \leftarrow \sqrt{(n_1dr-n_2dr+c_x)^2+ \\ +(m_1dr-m_2dr+c_y)^2+(o_1dr-o_2dr+c_z)^2}$
            $sum_{o_2} \leftarrow sum_{o_2} + \frac{s(o_2)s(o_2+b'_3)}{den}$

end for

$sum_{m_2} \leftarrow sum_{m_2} + sum_{o_2} s(m_2) s(m_2 + b'_2)$

end for

$sum_{n_2} \leftarrow sum_{n_2} + sum_{m_2} s(n_2) s(n_2 + b'_1)$

end for

$sum_{o_1} \leftarrow sum_{o_1} + sum_{n_2} s(o_1) s(o_1 + a'_3)$

end for

$sum_{m_1} \leftarrow sum_{m_1} + sum_{o_1} s(m_1) s(m_1 + a'_2)$

end for

$sum_{n_1} \leftarrow sum_{n_1} + sum_{m_1} s(n_1) s(n_1 + a'_1)$

end for

Az integrál eredménye a sum_{n_1} változó értéke az algoritmus futásának befejeződésekor.

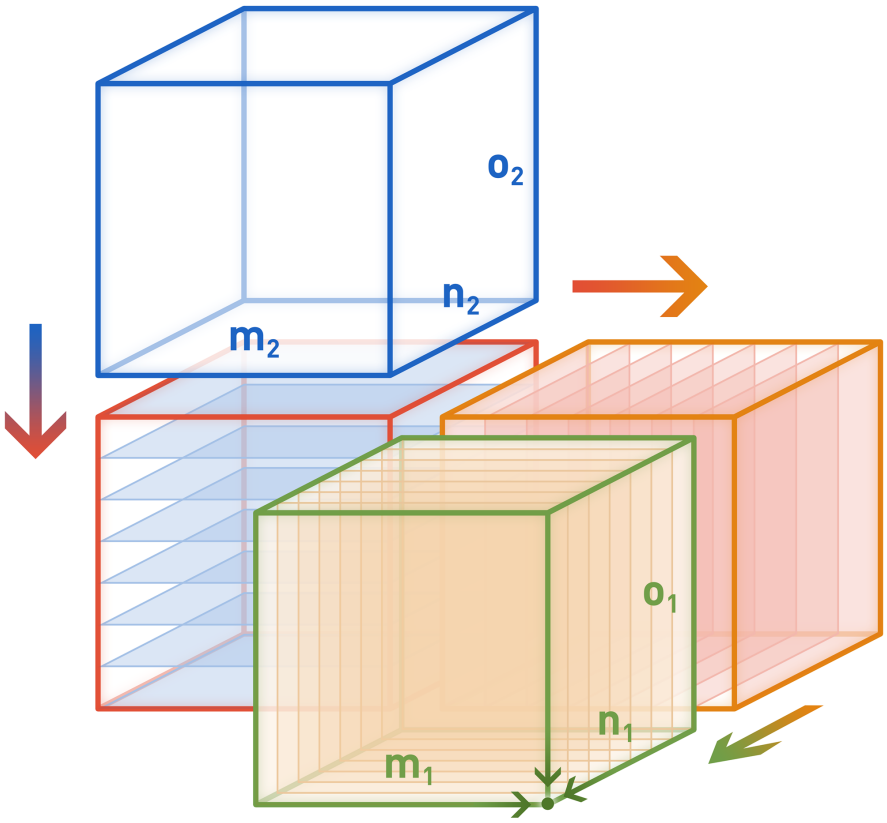
3.1. Algoritmus CUDA környezethez

A probléma párhuzamosításához reprezentáljuk a számítási teret egy hatdimenziós hiperkockával, a hiperkocka oldaléleinek hossza a ciklusváltozók végértékei legyenek. Ilyeténképpen a hiperkocka minden rácspontjához hozzárendelhető egy érték, mely az adott ciklusban kiszámolt nevező értékét (illetve annak reciprokát) tartalmazza. Ezt követően egy-egy ciklus a rácspontokon található aktuális értékeket megszorozza az adott mélységnek megfelelő számláló-résszel, majd egy tengely mentén összegzi az értékeket, azaz a hiperkocka dimenzióját eggyel redukálja – ez geometriailag egy tengely mentén történő vetítésként fogható fel. A hatodik ciklus végén a kocka 0 dimenzióssá redukálódik, azaz egy pontból áll, mely az integrálás eredményét tartalmazza.

Ilyen reprezentációban könnyen láthatóvá válik a párhuzamosítás lehetősége: egyrészt a hatdimenziós hiperkocka pontjainak kezdeti értéke egymástól teljesen függetlenül, párhuzamosan számítható. Továbbá a vetítés is párhuzamosítható – minden mélységben az eredménypontok számától (azaz a vetítés után megmaradó oldalélek szorzatától) függ a párhuzamosan elvégezhető műveletek száma.

A fenti eljárás azonban ilyen formában még nem alkalmazható, a memóriakapacitás korlátja miatt. Mivel a host és device közötti memóriamásolás sebessége limitált, érdemes a köztes számítási eredményeket végig a videokártya memóriájában tartani. Könnyen belátható, hogy a teljes hatdimenziós hiperkocka memóriában tárolása már viszonylag kis felbontás esetén sem megvalósítható. A wavelet felbontása legyen 3, ekkor a wavelet pontjainak száma $k = 2^3 \times 5 + 1 = 41$, így a hiperkocka rácspontjainak száma $n = k^6 = 4\ 750\ 104\ 241$. Csupán 4 bájtos számábrázolást használva a felhasznált memória már $c = n \times 4 \approx 17,7$ GB, double precíziót alkalmazva 35,4 GB. Már ilyen kis felbontás mellett is hatalmas memóriára lenne szükség, mely nem áll rendelkezésre a GPU-kban.

A felhasznált memóriaterület csökkenthető, ha a teljes hiperkocka helyett csak bizonyos altereit tároljuk a memóriában, majd a számítás során ezen alterek mozgatásával fedjük le a teljes hiperkockát. Háromdimenziós altereket kiválasztva már elegendő számú párhuzamos feladatra dekomponálódik a probléma, azonban ezek az alterek még 6-os felbontás mellett is a memóriában tarthatók.



3.1. ábra. *Hatdimenziós hiperkocka alterein végzett számítás*

A számítás módját a 3.1 ábra szemlélteti. A számítás menete a következő:

1. A nevező értéke párhuzamosan kiértékelődik egy háromdimenziós kocka mindegyik pontjában. (Ez az altér az ábrán kék kockaként van jelölve.)

2. A kocka rácspontjaihoz kiértékelésre kerül a számláló megfelelő része is, majd egy összegzés hajtódik végre az egyik él mentén, mely egy vetítésként fogható fel. Az eredmény kétdimenziós.

3. A hiperkocka egy – a kék kocka éleitől különböző – élén végigtolva a kék kockát, és a fenti két lépést végrehajtva a vetületekből ismét egy háromdimenziós kocka épül fel (piros).

4. A piros kocka a számláló megfelelő értékének kiszámítása után a kékhez hasonlóan vetíthető, a vetítéseket dimenzióról-dimenzióra végrehajtva a sárga, végül a zöld kocka adódik.

5. A zöld kocka három éle mentén történő vetítés eredményeképpen adódik az eredménypont.

A fentiek alapján a GPU-ra írt program algoritmusa az alábbi:

---

function DENOMINATOR(n_1, m_1, o_1)
  $n_2, m_2, o_2 \leftarrow$ GPU block indices
  return $\frac{1}{\sqrt{((n_1-n_2)\times dr+c_x)^2+((m_1-m_2)\times dr+c_y)^2+((o_1-o_2)\times dr+c_z)^2}}$
end function

function PROJECT(input, offset, edgeSize)
  for $i \leftarrow 0$ to edgeSize **do**
    RESULT $\leftarrow$ RESULT + input[i] $\times$ s(i) $\times$ s(i + offset) $\times$ dr
  end for
end function

$den, res_{o_2}, res_{m_2}, res_{n_2}$ is three-dimensional
res_{o_1} is two-dimensional
res_{m_1} is one-dimensional
res_{n_1} is zero-dimensional
for $n_1 \leftarrow 0$ to n_{1max} **do**
  for $m_1 \leftarrow 0$ to m_{1max} **do**
    for $o_1 \leftarrow 0$ to o_{1max} **do**
      $den \leftarrow denominator(n_1, m_1, o_1)$ ON GPU
        <<< $n_{2max} \times m_{2max} \times o_{2max}$ >>> COPIES
      $res_{o_2}[o_1] \leftarrow project(den, b'_3, o_{2max})$ ON GPU
        <<< $n_{2max} \times m_{2max}$ >>> COPIES
    end for
    $res_{m_2}[m_1] \leftarrow project(res_{o_2}, b'_2, m_{2max})$ ON GPU
      <<< $o_{1max} \times n_{2max}$ >>> COPIES
  end for
  $res_{n_2}[n_1] \leftarrow project(res_{m_2}, b'_1, n_{2max})$ ON GPU
    <<< $m_{1max} \times o_{1max}$ >>> COPIES
end for
$res_{o_1} \leftarrow project(res_{n_2}, a'_3, o_{1max})$ ON GPU <<< $n_{1max} \times m_{1max}$ >>> COPIES
$res_{m_1} \leftarrow project(res_{o_1}, a'_2, m_{1max})$ ON GPU <<< n_1 >>> COPIES
$res_{n_1} \leftarrow project(res_{m_1}, a'_1, n_{1max})$ ON GPU <<< 1 >>> COPIES

3.2. Algoritmus BOINC környezethez

Bár a BOINC rendszer használatával lehetőség nyílik grafikus kártya programozására is, ha az az önkéntes számítógépén rendelkezésre áll, azonban univerzális megoldásként csupán egy processzormag meglétét feltételezhetjük. A párhuzamosság itt magában a grid rendszerben keletkezik, az önkéntes számítógépek a problémának csak egy részfeladatát hajtják végre. A BOINC menedzselhetőségének nehézségei miatt érdemes a problémát úgy részekre bontani, hogy a részmegoldások összefésülése, a végső megoldás kiszámolása minél egyszerűbb legyen. Természetesen ez csak abban az esetben tehető meg, ha a részfeladatok futásideje ilyen feltételek mellett még nem túlságosan nagy.

A következő fejezetben részletesen tárgyalom a különböző módszerek teljesítményét, futásidejét, látható lesz majd, hogy a kialakított algoritmus megfelel az előző követelménynek, és az önkéntes számítógépén nem igényel túlságosan nagy futásidőt. Mindemellett a szerveroldalon az eredmény előállításához csak összegezni kell az egy eredményponthoz rendelt részfeladatok eredményét. A soros algoritmus legfelső szintű *for* ciklusának szerepét a feladat-összeállítás veszi át, a kliens számítógép a kiszámolni kívánt eredménypont koordinátáin kívül (mely az előző algoritmusoknak is paramétere) egy ciklusazonosítót is paraméterül kap, mely az n_1 ciklusváltozó szerepét veszi át.

A leírtak alapján a következő algoritmus alkalmazható a BOINC rendszer kliens gépein:

$n_1 \leftarrow cycleID$
$sum_{m_1} \leftarrow 0$
for $m_1 \leftarrow 0$ to m_{1max} **do**
  $sum_{o_1} \leftarrow 0$
  for $o_1 \leftarrow 0$ to o_{1max} **do**
    $sum_{n_2} \leftarrow 0$
    for $n_2 \leftarrow 0$ to n_{2max} **do**
      $sum_{m_2} \leftarrow 0$
      for $m_2 \leftarrow 0$ to m_{2max} **do**
        $sum_{o_2} \leftarrow 0$
        for $o_2 \leftarrow 0$ to o_{2max} **do**
          $den \leftarrow \sqrt{(n_1 dr - n_2 dr + c_x)^2 + \overline{+(m_1 dr - m_2 dr + c_y)^2 + (o_1 dr - o_2 dr + c_z)^2}}$
          $sum_{o_2} \leftarrow sum_{o_2} + \frac{s(o_2)s(o_2+b'_3)}{den}$
        end for
        $sum_{m_2} \leftarrow sum_{m_2} + sum_{o_2} s(m_2)s(m_2 + b'_2)$
      end for
      $sum_{n_2} \leftarrow sum_{n_2} + sum_{m_2} s(n_2)s(n_2 + b'_1)$
    end for
    $sum_{o_1} \leftarrow sum_{o_1} + sum_{n_2} s(o_1)s(o_1 + a'_3)$
  end for

$sum_{m_1} \leftarrow sum_{m_1} + sum_{o_1} s(m_1) s(m_1 + a'_2)$

end for

$result \leftarrow sum_{m_1} s(n_1) s(n_1 + a'_1)$

Negyedik fejezet

Implementáció

4.1. Platformfüggő megfontolások

Az előző fejezetben a különböző platformokhoz fejlesztett algoritmusok természetesen már igazodnak az alkalmazott rendszer sajátosságaihoz. Azonban az implementáció során olyan megfontolásokat is figyelembe kell venni, melyek az algoritmusok szintjén nem jelennek meg.

4.1.1. CUDA

Az algoritmus megalkotása után további sebességnövekedés elérése érdekében az alábbi megfontolások szerint finomhangoltam a programkódot:

- A GPU-k memóriahozzáférése gyorsabb, ha az olvasás szekvenciálisan történik. A hiperkocka élei mentén történő vetítések során a beolvasandó értékek – a hiperkocka memóriában történő leképzésétől függően – egymástól távol, vagy egymás után is elhelyezkedhetnek. A gyors hozzáférés érdekében a hiperkocka leképezését úgy valósítottam meg, hogy a vetítések szekvenciális hozzáférést eredményezzenek.
- Annak ellenére, hogy a CUDA függvénykönyvtára előre definiált, hatékony hatványfüggvényt biztosít, a tapasztalat azt mutatja, hogy a nevező számítása során szükséges négyzetre emeléseket érdemesebb egy lokális segédváltozó segítségével, szorzással elvégezni.
- A vetítést végző függvényekben sokszor kell elérni ugyanolyan (vagy csak egy ofszetben eltérő) indexű tömbelemeket. Az indexek a *blockIdx*, *threadIdx*, *blockDim*, stb. objektumokból több művelet segítségével állíthatók elő – a hatékonyság érdekében a többször felhasznált indexek állandó részét a device függvény elején kiszámítom, majd egy segédváltozóban tárolom, így az indexelés jelentősen gyorsul.

4.1.2. BOINC

A BOINC platform esetén az implementáció során az állapotmentés megvalósítása érdekes. Bár a *boinc_time_to_checkpoint()* függvény segít elkerülni a túl sűrű állapotmentést, és így a legbelső *for* ciklusban is el lehetne helyezni az állapotmentésre és visszaállításra szolgáló metódust, ez felesleges, elegendően sűrű mentést biztosít a harmadik szintben történő implementálása is.

A program állapotának teljes leírásához az m_1, o_1, n_2, m_2, o_2, valamint a sum_{m_1}, sum_{o_1}, sum_{n_2}, sum_{m_2} és sum_{o_2} változók tárolására van szükség. Az állapotmentő kódrészlet a *boinc_time_to_checkpoint()* függvény segítségével ellenőrzi a legutóbbi mentés óta eltelt időt, és ha szükséges a mentés, fájlba írja a fenti változókat, majd meghívja a *boinc_checkpoint_completed()* függvényt.

Újraindulás esetén létezik már állapotleíró fájl, ezt észlelve a program bebillenti a *restart* segédváltozót, aminek hatására az állapot mentésének helyéhez elérve egy olyan kódrészlet fut le, ami a fájlból visszatölti a változók korábbi értékeit.

4.2. Futásidők

A soros kód (egy AMD Athlon X2 processzoron futtatva) a 4.1 táblázatban feltüntetett idő alatt futott le különböző felbontások esetén. (Kisebb felbontásokkal több futtatást is elvégeztem, így azok átlagolt eredmények, az 5-ös felbontású teszt azonban csak egyszer futott le.)

4.1. táblázat. *Futásidők CPU-n*

Felbontás	Futásidő
2	5,4 s
3	271 s
4	4,5 óra
5	11,3 nap

A felbontást eggyel növelve mindegyik ciklust kétszer annyiszor kell végrehajtani, így elméletileg $2^6 = 64$-szeresére nő a futásidő. A mért értékek megfelelnek ennek a várakozásnak. Ezek alapján 6-os felbontással egy eredmény kiszámításához több mint másfél év lenne szükséges.

A GPU-ra írt programot a BME szuperszámítógépének segítségével teszteltem, melyben 4 darab Tesla M2070 GPU található. A kártyák külön-külön 448 processzor magot és 6 GB memóriát tartalmaznak, a double pontosságú lebegőpontos számítási teljesítményük 515 GFLOPS. Habár az NVIDIA GeForce és Quadro modellek is támogatják a CUDA platformot, a Tesla sorozatot kifejezetten nagy számításigényű problémák megoldására fejlesztették, minden kártya gyárilag tesztelt zérus hiba toleranciával, illetve ECC (Error-correcting Code) memória található bennük az adatok meghibásodását megakadályozandó. A Tesla-n mért futásidőket a 4.2 táblázat foglalja össze.

4.2. táblázat. *Futásidők GPU-n*

Felbontás	Futásidő
2	49 s
3	537 s
4	3,3 óra
5	2,9 nap
6	42 nap

Bár 2-es felbontással a program lassabban fut le GPU-n, azonban a felbontás növelésével a futásidő növekedése kisebb ütemű, mint CPU esetén (a felbontások közötti szorzófaktor körülbelül harmadannyi), így nagyobb felbontások esetén lényegesen jobb teljesítményt nyújt a GPU, 6-os felbontással is belátható időn belül lefut a program. A kis felbontások során tanúsított gyengébb teljesítmény annak tudható be, hogy ilyenkor még nincs kihasználva a kártyában található összes mag.

A BOINC rendszert belső hálózaton, három kliensgép segítségével teszteltem. A klienseken futó kód különböző felbontások esetén a 4.3 táblázatban látható idő alatt futott le.

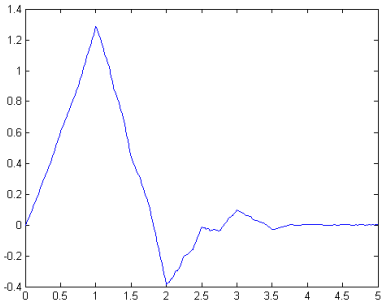
4.3. táblázat. *Futásidők BOINC rendszeren*

Felbontás	Futásidő
2	$<<$ 1 s
3	6,7 s
4	194 s
5	1,7 óra
6	2,3 nap

Fontos megjegyezni, hogy ezek az időértékek csak a részfeladatok kliens gépen történő futásának idejei. Az integrál tényleges kiszámításához több klienst is igénybe kell venni, melyek száma a felbontástól függ. 6-os felbontás esetén 321 darab kliens szükséges, hogy a teljes eredmény ennyi idő alatt előálljon.

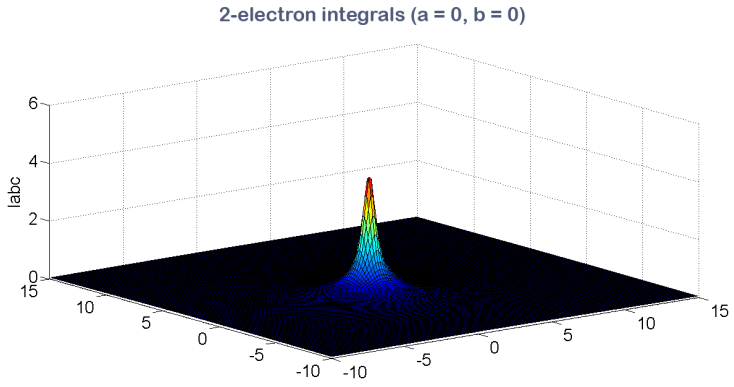
4.3. Eredmények

A számítás skálafüggvényeként a Daubechies-6 függvény szolgált. [1] A függvény a 4.1 ábrán látható.



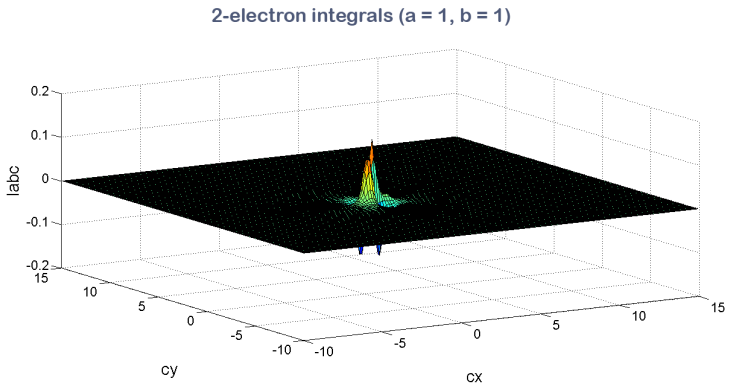
4.1. ábra. *Daubechies-6 skálafüggvény*

Kétdimenzióban $a = (0, 0)$ és $b = (0, 0)$ vektorokat rögzítve a 4.2 ábrán látható eredmények adódtak c vektor függvényében.



4.2. ábra. *Kételektron-integrálok $a = (0, 0)$ és $b = (0, 0)$ esetben*

Egy másik esetben, $a = (1, 1)$ és $b = (1, 1)$ vektorokat rögzítve a 4.2 ábrán látható eredmények születtek.



4.3. ábra. *Kételektron-integrálok $a = (1, 1)$ és $b = (1, 1)$ esetben*

Az ábrákon látható, kétdimenziós eredménysorok kevesebb mint egy perc alatt számolódtak ki GPU-n. Korábban láthattuk, hogy háromdimenziós esetben, 6-os felbontással egyetlen eredménypont kiszámításához is 42 nap szükséges, így nagyfelbontású eredmények még nem állnak rendelkezésre. 2-es felbontással már készültek eredménysorok, azonban ábrázolásuk nehézkes (már a kétdimenziós eredmények megjelenítéséhez is felületek ábrázolására volt szükség), illetve maguknak a kételektron-integrálok eredményeinek az értelmezése túl is mutatna jelen dolgozat keretein, így ettől megkímélem az olvasót.

Ötödik fejezet

A projekt folytatása

A kételektron-integrálok számításához algoritmusokat fejlesztettem és implementáltam CUDA és BONINC platformokra. Nagyobb felbontások esetén jelentős sebességnövekedés mutatkozott a grafikus kártyára írt program esetén az egyszálú programhoz képest, 6-os felbontásban a becsült másfél éves futásidő helyett GPU-n 42 nap alatt lefutott a program.

Sajnos ez még mindig túl hosszú futásidő ahhoz, hogy a szükséges integrálokat háromdimenzióban végig lehessen számolni. Azonban a videokártyák dinamikus fejlődése miatt valószínűsíthető, hogy két-három éven belül rendelkezésre fog állni olyan kártya, mellyel az összes numerikus módszerrel számolandó integrál belátható időn belül kiszámíthatóvá válik.

A BOINC rendszer használata szintén lehetővé teszi az integrálok kiszámítását. A platform és az arra írt program tesztelése lokális hálózaton, három számítógép bevonásával történt. Ez a konstrukció természetesen nem nyújt elegendő számításkapacitást a probléma megoldásához, csupán tesztelés céljára alkalmas. Azonban a projekt folytatásaként, a későbbiekben nyilvános hálózatról elérhető önkéntes számítási projekt indításával – megfelelő számú önkéntest bevonva – ezzel a módszerrel is kiszámítható az összes szükséges integrál.

Köszönetnyilvánítás

A munka szakmai tartalma kapcsolódik az „Új tehetséggondozó programok és kutatások a Műegyetem tudományos műhelyeiben” című projekt szakmai célkitűzéseinek megvalósításához. A BME szuperszámítógépének használatán keresztül a projekt megvalósítását a TÁMOP-4.2.2.B-10/1–2010-0009 program támogatta.

Rövidítések jegyzéke

AMD	Advanced Micro Devices
API	Application Programming Interface
BLAS	Basic Linear Algebra Subprograms
BOINC	Berkeley Open Infrastructure for Network Computing
CERN	European Organization for Nuclear Research
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
ECC	Error-correcting Code
FFT	Fast Fourier Transform
FLOPS	Floating Point Operations Per Second
GPU	Graphics Programming Unit
GPGPU	General-Purpose Computing on Graphics Processing Units
LCAO	Linear Combination of Atomic Orbitals
LHC	Large Hadron Collider
LIGO	Laser Interferometer Gravitational-Wave Observatory
MPI	Message Passing Interface
NPP	NVIDIA Performance Primitives
OpenMP	Open Multi-Processing
POSIX	Portable Operating System Interface for Unix
SETI	Search for Extra-Terrestrial Intelligence

Ábrák jegyzéke

Táblázatok jegyzéke

Irodalomjegyzék

[1] I. Daubechies, *Ten Lectures On Wavelets*, vol. 61 of *CBMS-NSF regional conference series in applied mathematics*. SIAM, Philadelphia, 1992.

[2] S. Nagy and J. Pipek *Int. J. Quantum Chem.*, vol. 84, p. 523, 2001.

[3] G. E. Moore, „Cramming more components onto integrated circuits," *Electronics*, pp. 114–117, Apr. 1965.

[4] Y. . F. A. Suman, S.K. ; Joshi, „Thermal issues in next-generation integrated circuits," *IEEE Trans. on Device and Materials Reliability*, vol. 4, pp. 709–714, Dec. 2004.

[5] B. Barney, „Introduction to parallel computing." `https://computing.llnl.gov/tutorials/parallel_comp/`.

[6] M. Garland, „Parallel computing with CUDA," in *Proc. of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, (Atlanta, GA), p. 1, Apr. 19–23 2010.

[7] „NVIDIA CUDA – a parallel computing platform." `http://www.nvidia.com/object/cuda_home_new.html`.

[8] N. Kinayman, „Parallel programming with GPUs: Parallel programming using graphics processing units with numerical examples for microwave engineering," *Microwave Magazine, IEEE*, vol. 14, no. 4, pp. 102–115, 2013.

[9] D. Anderson, „BOINC: a system for public-resource computing and storage," in *Fifth IEEE/ACM International Workshop on Grid Computing*, pp. 4–10, Nov. 8 2004.

[10] „BOINC project." `http://boinc.berkeley.edu/`.