Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Kvantummechanikai számítások optimalizálása

Diplomaterv

Készítette
Király Zoltán

Konzulens
dr. Szegletes Luca

2022. május 29.

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott *Király Zoltán*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2022. május 29.

Király Zoltán
hallgató

Kivonat

A kvantummechanika területén a különböző atomok és molekulák elektronszerkezetének számítása során gyakran felmerülő probléma az úgynevezett kételektron-integrálok megállapítása, mely egy költséges, számításintenzív művelet. Dr. Szegletes Luca és Dr. Pipek János 2014-ben megmutatták, hogy a kételektron-integrálok számítása megoldható egy, a waveletek skálafüggvényein alapuló módszer segítségével. Ennek a módszernek az előnye az, hogy az eredményei előre kiszámíthatók és újrafelhasználhatók. Ennek köszönhetően a megoldás nagymértékben javíthatná kvantummechanikai számítások sebességét. Sajnos a wavelet alpú eredmények meghatározása is számításigényes feladat, ezért ahhoz, hogy a gyakorlatban is használni lehessen, a módszer jelentős optimalizálása szükséges. Dolgozatom célja a wavelet alapú megoldás bemutatása és optimalizálása, valamint az ehhez kapcsolódó témakörök, a wavelet transzformáció, a párhuzamos számítási platformok és számítógépes integrálszámítási módszerek ismertetése. A kételektron-integrálok számításának optimalizálására különböző, az NVIDIA CUDA platform használatával készített implementációkat mutatok be és elemzek. Ezen felül megvizsgálom, hogy lehetséges-e a módszer javítása azáltal, hogy az integrálandó függvény pontjainak ritkább mintavételezése mellett különböző numerikus approximációkat végzek el. A dolgozatban bemutatott technikákkal a precízió megtartása mellett a számítás futásidejét három nagyságrenddel sikerült javítani a CPU alapú megoldáshoz képest. Ezen felül megállapítottam, hogy a mintavételezés ritkítása és az integrál Newton-Cotes kvadratúrákkal való közelítése az eredmények pontatlanságából adódóan nem alkalmazható a módszer gyorsítására.

Abstract

In the field of quantum mechanics, a problem that often arises during the calculation of the electronic structure of different atoms and molecules is the determination of the so-called two-electron integrals, which is a computationally intensive operation. In 2014, Dr. Luca Szegletes and Dr. János Pipek showed that the calculation of these integrals can be solved by using a method based on wavelet scaling functions. The advantage of this method is that its results can be determined in advance for future reusability. Consequently, this solution could greatly improve the speed of certain quantum mechanical calculations. Unfortunately, the calculation of the wavelet-based results is also a computationally intensive task, so in order to be able to use it in practice, a significant optimization of the method is required. The goal of my thesis is to introduce and optimize the wavelet-based solution and present the related topics, such as the wavelet transform, parallel computing platforms and different integration techniques. To optimize the computation of two-electron integrals, I present and analyze different implementations using the NVIDIA CUDA platform. In addition, I examine whether it is possible to improve the method by performing different numerical approximations on a less frequent sampling of the points of the function to be integrated. With the techniques presented in this work, while maintaining the original precision, I was able to improve the runtime of the calculation by three orders of magnitude, compared to the CPU based soultion. In the terms of the approximation, I found that the technique of decreasing the number of samples while applying different Newton-Cotes quadratures cannot be applied to accelerate the method due to the inaccuracy of the results.

1. fejezet

Bevezetés

A különböző számítások optimalizációjára magától értetődő igény adódik. Különösen igaz ez az olyan algoritmusok esetén, melyek futásideje napokban, vagy akár években mérhető. A fizika, kémia, csillagászat és egyéb, a világunk működésével foglalkozó tudományágak bővelkednek az olyan problémákban, melyek még a létező precíz modellek ellenére is csak bizonyos közelítések használatával oldhatók meg. Egy ilyen közelítő módszer a kvantummechanikai elektronszerkezet számítások során alkalmazott Hartree-Fock módszer is. Annak ellenére, hogy ez egy approximációs technika, az alkalmazása jelentős idő- és számítási igénnyel jár, melynek forrása az iteratív eljárás során felmerülő, az elektronok kölcsönhatását leíró kételektron-integrálok meghatározása. A kételektron-integrálok számítására létezik egy, a waveletek elméletére építő technika, mellyel a Hartree-Fock módszert jelentősen fel lehetne gyorsítani. Ehhez arra van szükség, hogy az integrál értékeit előre meghatározzuk, azonban ez egyelőre nem lehetséges, mivel a wavelet alapú módszer is egy rendkívül számításintezív megoldás. A dolgozatom célja a módszer optimalizációs lehetőségeinek vizsgálata.

Az integrál meghatározása nagy mennyiségű művelet elvégzését követeli meg, melyek egymástól függetlenül végrehajthatók. Ennek köszönhetően a megoldás könnyen párhuzamosítható. Az ilyen jellegű problémák elvégzésére a grafikus feldolgozóegységek hatékony eszköznek bizonyulnak, ezért a módszer futásidejének javítását a grafikus processzor általános célú programozásával végzem az NVIDIA CUDA platform eszközeit használva.

A dolgozatom második fejezetében a megoldandó probléma és a megoldás alapjául szolgáló elméleti témaköröket foglalom össze. Először is a Hartree-Fock módszert, és ezen belül a kételektron-integrálokat ismertetem. Ezt követően a waveletek bemutatásátra térek át, azonban ezek létezése szorosan összefügg az alkalmazási területükkel, a wavelet transzformációval. Ennek megfelelően erre a témakörre a wavelet transzformáció jelfeldolgozás területén való alkalmazásán keresztül nyújtok rálátást. A waveletek megismerése után bemutatom a skálafüggvényeken alapuló megoldást a kételektron-integrálok számítására. Az elméleti rész utolsó fejezetében a számítógépen való integrálszámítás különböző technikáira térek ki.

A harmadik fejezet a megoldásom technológiai hátterét, az általános célú grafikus programozást mutatja be. Egy rövid történeti áttekintés után az ezt megvalósító jelentősebb keretrendszereket ismertetem, különös tekintettel az NVIDIA CUDA keretrendszerre, melyet az optimalizáció során használtam.

A negyedik fejezetben felvezetem a megoldandó feladatot, majd a módszer gyorsítására készített CUDA alapú implementációimat ismertetem. A fejezet végén bemutatom egy, a numerikus integrálszámítási módszereken végzett kísérlet alapjait.

Az ötödik fejezetben a különböző megoldások eredményét mutatom be: az egyes implementációk teljesítményét és pontosságát, valamint integrálszámítási módszerekkel végzett kísérletből levont következtetést.

A hatodik fejezetben összefoglalom a dolgozat tapasztalatait, a hetedik fejezetben pedig iránymutatást adok a különböző továbbfejlesztési lehetőségekkel kapcsolatban.

2. fejezet

Elmélet

2.1. Kételektron-integrálok

A kételektron-integrálok olyan matematikai kifejezések, melyek leírják, hogy két elektron hogyan lép kölcsönhatásba egymással. Ez az építőelem fontos része a hullámfüggvényeket leíró matematikai kifejezéseknek és az ezeket kiszámító számítógépes programoknak. Ebben a szekcióban a kételektron-integrálok levezetését szeretném ismertetni.

2.1.1. Motiváció

A kémia és szilárdtestfizika nagy pontosságú elektronszerkezet számításai során elengedhetetlen a hullámfüggvények előállítása: $\Psi(r_1, r_2, \cdots, r_N)$, ahol az r_i értékek háromdimenziós vektorok. Az időfüggetlen Schrödinger-egyenlet megoldása általában a Hartree-Fock (HF) módszerrel[17] vagy valamilyen, a HF eljáráson alapuló elektronkorrelációs módszerrel történik.

2.1.2. Kételektron-integrálok levezetése[18]

A HF módszer egy pszeudo-sajátérték problémához vezet

$$F\varphi_i = \varepsilon_i \varphi_i \quad i = 1, \cdots, n. \tag{2.1}$$

Itt F az úgynevezett Fock operátor, φ_i az i-edik térbeli pálya, ε_i a pályaenergia, n pedig a pályák száma. A Fock operátor az egyelektron mozgási energia kifejezésből, az atommagokkal való V Coulomb kölcsönhatásból, valamint az elektronok V_{eff} effektív kölcsönhatásából tevődik össze, mely a φ_i pályáktól függ.

$$F = -\frac{1}{2}\Delta + V + V_{eff}[\varphi_i] \tag{2.2}$$

A V_{eff} effektív kölcsönhatás a kicserélődési és Coulomb kifejezésekből áll. Mivel a Fock operátor a 2.1 egyenlet sajátvektoraira épül, ez a probléma önkonzisztens iterációs módon (SCF módszer) oldható meg.

A fenti probléma hatékony kezelésére egy algebrai megközelítést használnak egy $\chi_\mu(r)$ bázis halmaz bevezetésével, ahol $\mu = 1, \cdots, N_B$, N_B pedig a bázisfüggvények halmazának számossága. Minden φ_i pálya leírható a

$$\varphi_i(r) = \sum_\mu c_\mu^i \chi_\mu(r) \tag{2.3}$$

kifejtéssel, ahol c_μ^i értékek a lineáris kifejtés együtthatói. Ezáltal a számítás az alábbi pszeudo mátrix sajátérték problémához vezet, ami megoldható különböző numerikus

algoritmusok segítségével.

$$\sum_v F_{\mu\nu} c_\nu^i = \varepsilon_i c_\mu^i. \tag{2.4}$$

Az $F_{\mu\nu}$ Fock mátrix a következő módon áll elő:

$$F_{\mu\nu} = h_{\mu\nu} + \sum_{\rho\sigma} \left(\sum_i c_\rho^i c_\sigma^i \right) [2(\mu\nu \mid \rho\sigma) - (\mu\rho \mid \sigma\nu)] \tag{2.5}$$

Itt $h_{\mu\nu}$ az alábbi módon definiálható:

$$h_{\mu\nu} = \int dr \chi_\mu^*(r) \left[-\frac{1}{2}\Delta + V \right] \chi_\nu(r) \tag{2.6}$$

illetve a

$$(\mu\nu \mid \rho\sigma) = \int\int dr_1 dr_2 \frac{\chi_\mu^*(r_1)\chi_\nu(r_1)\chi_\rho^*(r_2)\chi_\sigma(r_2)}{|r_1 - r_2|} \tag{2.7}$$

szimbólumok a bázisfüggvények úgynevezett kételektron integráljait jelölik.

Az általános molekulapálya elméletben a χ_μ bázisfüggvényeknek az atompályákat választják és a fent ismertetett módszer a *"linear combination of atomic orbitals"* (LCAO) nevet viseli.

2.2. Wavelet transzformáció

Megoldásunk során az úgynevezett waveletek elméletére építünk. A waveletek speciális tulajdonságokkal rendelkező függvények, melyeknek köszönhetően praktikus eszközöknek bizonyulnak a jelfeldolgozás területén. A waveletek ismertetését a legnépszerűbb alkalmazási területük, a wavelet transzformációk bemutatásával teszem. A wavelet-analízis motivációjának és működésének megértése érdekében - mint ahogyan ezt a szakirodalom is sok esetben teszi - először a *Fourier-transzformációt* (FT) és a belőle származtatott *rövid idejű Fourier-transzformációt* (STFT) mutatom be. Ezek a módszerek adják az alapot a *multirezolúciós analízis* (MRA) és a *folytonos wavelet transzformáció* (CWT) bevezetéséhez, így jutunk el végül a *diszkrét wavelet transzformáció* (DWT) ismertetéséhez.

Hangsúlyozandó, hogy a fejezet célja az, hogy a felsorolt módszerekre és azok problémáira rálátást adjon. A koncepciók teljeskörű elméleti hátterének bemutatása jóval túlmutat a dolgozat keretein.

2.2.1. Fogalmak, jelölések

Általánosságban az időben és/vagy térben változó fizikai mennyiségeket nevezzük **jelnek**. A **jelfeldolgozás** a villamosmérnöki, szoftvermérnöki, informatikai tudományok, valamint a fizika és az alkalmazott matematika határterülete, mely a jelek manuális és automatizált átalakításával, elemzésével, feldolgozásával foglalkozik.

A továbbiakban **nyers jel** alatt egy, az idő tartományon értelmezett folytonos függvényt értünk. A nyers jelet az $x(t)$ kifejezéssel jelöljük.

Az egyes **transzformációk** célja a fentiek értelmében a nyers jel átalakítása (transzformálása) abból a célból, hogy ezáltal olyan információkat nyerjünk ki belőle, melyek közvetlenül nem álltak rendelkezésünkre.

A **feldolgozott** vagy **transzformált jel** alatt a transzformáció kimeneteként kapott jelet értjük.

Egy f függvény **tartóján** az értelmezési tartományának azt a részhalmazát értjük, melyekre $f(x) \neq 0$. A tartót **kompaktnak** nevezzük, ha folytonos és véges. Ennek

megfelelően **kompakt tartójú függvény** alatt olyan függvényt értünk, mely egy véges intervallumon elenyészik ($\exists x_0$, hogy ha $|x| > x_0$, akkor $f(x) = 0$).

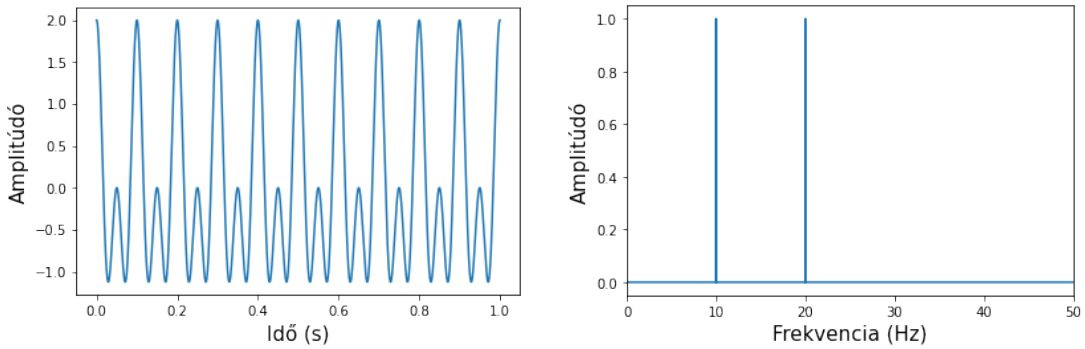
2.2.2. Fourier-transzformáció (FT)

Joseph Fourier francia matematikus és fizikus megmutatta, hogy bármely folytonos periodikus függvény (jel) harmonikus összetevőkre bontható, azaz felírható különböző amplitúdójú és frekvenciájú szinusz és koszinusz jelek végtelen összegeként (Fourier sorok). Ez az ötlet később általánosítva lett nem periodikus folytonos, majd periodikus és nem periodikus diszkrét függvényekre. Ez az általánosítás tette lehetővé, hogy napjainkra a módszer az analóg és digitális jelfeldolgozás egyik legfontosabb segédeszközévé vált [8]. Egy adott $x(t)$ jel F Fourier-transzformáltját az alábbi módon definiálhatjuk:

$$F(f) = \int_{-\infty}^{\infty} x(t) \cdot e^{-2i\pi ft} dt \tag{2.8}$$

A fenti egyenletben f változó a frekvenciát jelöli. Az egyenletben található komplex exponenciális tag ekvivalens a $\cos(2\pi ft) + i \cdot \sin(2\pi ft)$ alakkal. Látható tehát, hogy egy adott f érték mellett az $x(t)$ jelet felszorozzuk egy f frekvenciájú szinuszos és koszinuszos jelekből álló kifejezéssel, majd integráljuk a teljes időtartományon. Ha az így kapott integrál értéke nagy, akkor azt mondjuk, hogy az $x(t)$ jelnek az f frekvencián egy domináns spektrális komponense van.

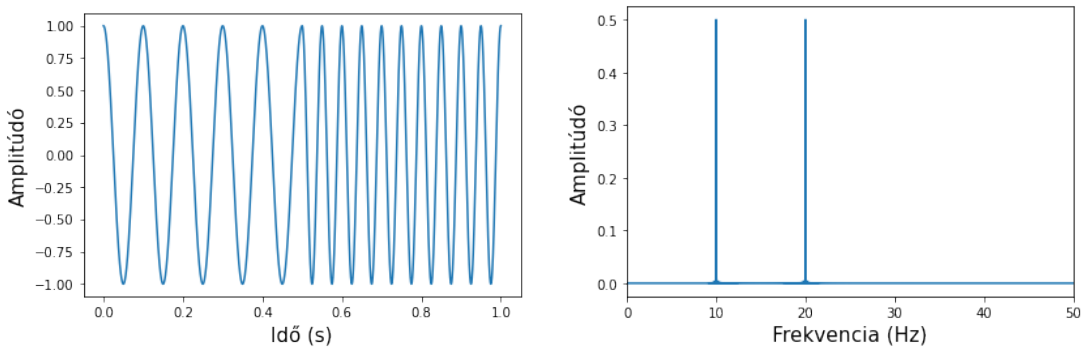
Ha egy jelre a Fourier-transzformációt alkalmazzuk, akkor a jel úgynevezett *spektrális felbontását* kapjuk. Ez a reprezentáció azt mutatja meg, hogy milyen frekvenciák léteznek az adott jelben. A 2.1 ábra az $x(t) = \cos(2\pi 10t) + \cos(2\pi 20t)$ jelet és annak Fourier-transzformáltját ábrázolja. Megfigyelhető, hogy a spektrális felbontásban éppen a 10Hz és 20Hz értékeknél találhatók csúcsok, tehát az x(t) jelnek ezeken a frekvenciákon vannak domináns spektrális komponensei. Ez könnyen belátható a jel értékének konstrukciója alapján.



a, nyers jel b, spektrális felbontás

2.1. ábra. A Fourier-transzformáció bemenete és kimenete

A Fourier-transzformáció segítségével tehát megismerhetjük a bemeneti jel spektrális komponenseit. Amennyiben azt is meg szeretnénk tudni, hogy az egyes komponensek időben mikor fordultak elő, a Fourier-transzformáció nem tud választ adni. Ezt szemlélteti a 2.2 ábra. A nyers jelünk ismét egy 10Hz-es és egy 20Hz-es komponensből áll, de ezek időben elkülönülve jelentkeznek. Látható, hogy a spektrális felbontásban szintén az ezeknek megfelelő helyeken jelentkeznek a csúcsok. Könnyen belátható, hogy a 2.8 egyenletben szereplő integrál (végtelen összeg) értéke független attól, hogy az x függvény időben mikor tartalmazza az adott frekvenciát.



a, nyers jel b, spektrális felbontás

2.2. ábra. Nem stacionárius jel transzformáltja

Az első példában ez a probléma nem állt fenn, mivel a jel minden spektrális komponensét minden időpontban tartalmazta. Az ilyen jeleket **stacionárius** jeleknek nevezzük. A második példa nem stacionárius jel, mert időben változó spektrális komponenseket tartalmaz.

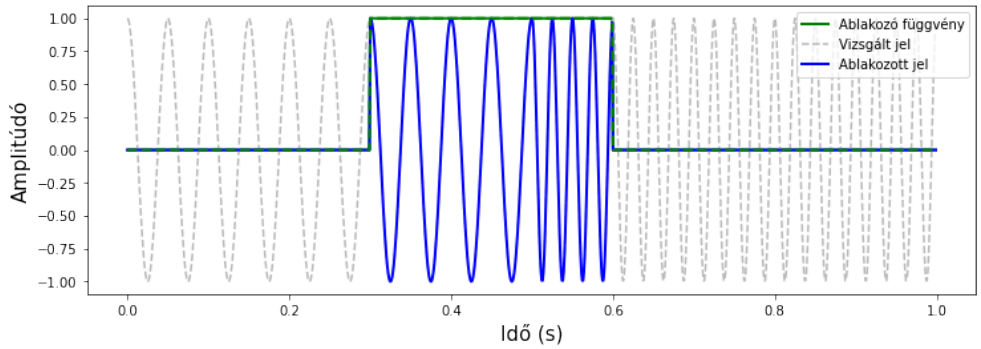
Amennyiben nem vagyunk kíváncsiak a spektrális felbontás időbeliségére, vagy stacionárius jelekkel foglalkozunk, a Fourier-transzformáció megfelelő eszközt nyújt számunkra. Ellenkező esetben más módszert kell alkalmaznunk.

2.2.3. Rövid idejű Fourier-transzformáció (STFT)

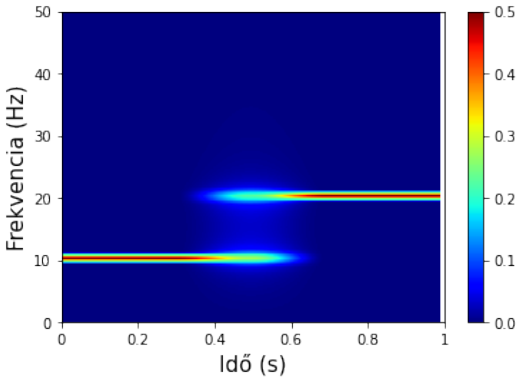
Az előző pontban ismertetett problémára egy kézenfekvő megoldást nyújt a rövid idejű Fourier-transzformáció (Short Time Fourier Transform, STFT). Feltételezzük, hogy egy jel kellően rövid szegmensei stacionáriusnak tekinthetők. Ebben az esetben megtehetjük, hogy a jelet w hosszúságú időszeletekre bontjuk, majd ezeken külön-külön végezzük el a Fourier-transzformációt:

$$STFT(t', f) = \int_{-\infty}^{\infty} x(t) \cdot \omega(t - t') \cdot e^{-2i\pi ft} dt \tag{2.9}$$

Az újonnan bevezetett $\omega(t)$ függvény egy úgynevezett ablakozófüggvény. Az ezzel való szorzás eredményeképp a jelnek csak egy w szélességű tartománya marad változatlan, ezen kívül mindenhol zérus lesz. Azt, hogy az ω által meghatározott időablak hol található, a szintén újonnan bevezetetett t' eltolási paraméter határozza meg. A 2.3 ábra az ablakozó függvény működését mutatja be, a 2.4 pedig egy példa az STFT kimenetére egy nem stacionárius jel esetén.



2.3. ábra. Az ablakozó függvény működése



2.4. ábra. STFT egy nem stacionárius jel esetén

Felmerül a kérdés, hogy hogyan válasszuk meg az időablak w szélességét. Természetesen arra törekszünk, hogy ez a lehető legkisebb legyen, így a részjelek stacionáriusnak tekinthetők és az időbeli felbontás is részletesebb. Sajnos ebben a tekintetben egy fizikai akadályba ütközünk, melyet a *Heisenberg-féle határozatlanságí elv* ír le. Ez jelfeldolgozási viszonylatban azt mondja ki, hogy a jel egy pontos frekvenciájának és a frekvencia pontos előfordulási helyének egyidejű meghatározása nem lehetséges. Más szavakkal egy jel nem reprezentálható egy pontként az idő-frekvencia térben, az időbeli felbontást csak a frekvenciabeli felbontás rovására tudjuk finomítani. [12]

- Keskeny ablak esetén magas időtartománybeli és alacsony frekvenciatartománybeli felbontást kapunk
- Széles ablak esetén alacsony időtartománybeli és magas frekvenciatartománybeli felbontást kapunk

Ez a probléma adja meg a motivációt a multirezolúciós analízis bevezetésére.

2.2.4. Multirezolúciós analízis (MRA)

Bár az idő- és frekvenciabeli felbontás problémája a használt transzformációtól függetlenül fennáll, a multirezolúciós analízis egy alternatív megoldást biztosít a jelek elemzésére. Mint ahogyan azt a neve is mutatja, ebben a megközelítésben a különböző spektrális komponenseket különböző felbontással vizsgáljuk. A multirezolúciós analízis

- Magas frekvenciák esetén magas időtartománybeli és alacsony frekvenciatartománybeli felbontást ad
- Alacsony frekvenciák esetén alacsony időtartománybeli és magas frekvenciatartománybeli felbontást ad

Ez a megoldás olyan esetekben hasznos, amikor a jel rövid ideig tartó magas frekvenciákat és hosszú ideig tartó alacsony frekvenciákat tartalmaz. Szerencsére a gyakorlati alkalmazásban előforduló jelek többsége ilyen típusú. [20]

2.2.5. Wavelet transzformáció [19]

A STFT egy nagy népszerűségnek örvendő alternatívája a wavelet transzformáció, vagy más néven wavelet analízis. A wavelet analízis egy skálázható modulált ablak használatával oldja meg a jel feldarabolási problémáját. Ezt az ablakot végigcsúsztatjuk a jelen és minden egyes pozícióban meghatározzuk a spektrális felbontást. Ezt a műveletet többször

megismételjük úgy, hogy az ablak szélességét minden ciklusban egyre rövidebbre vesszük. A végeredmény a jel különböző felbontású idő-frekvencia reprezentációinak egy gyűjteménye lesz, tehát a wavelet transzformáció is egy multirezolúciós technika.

A wavelet analízis esetén általában nem idő-frekvencia, hanem úgynevezett "time-scale" reprezentációkról beszélünk, ahol a scale (a csúszóablak átméretezési együtthatója) bizonyos értelemben a frekvencia ellentettjének tekinthető.

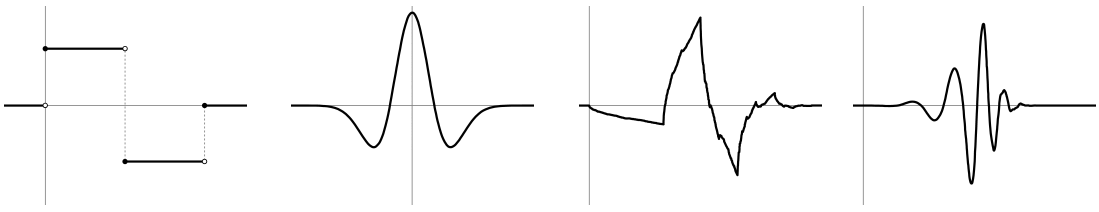
A következő részben a folytonos és a diszkrét wavelet analízist mutatom be, de mindenek előtt ismerjük meg, hogy mik is azok a waveletek.

2.2.5.1. Waveletek

A wavelet transzformáció során a dekompozíció alapját a trigonometrikus függvények helyett speciális függvények, úgynevezett waveletek képezik. A waveletek tulajdonságait különböző matematikai kritériumok határozzák meg, melyek célja kifejezetten annak a biztosítása, hogy ezek a függvények alkalmasak legyenek a későbbiekben bemutatott transzformációk elvégzésére. A fontosabb tulajdonságok formális definíció nélkül a következők:

- Folytonosan differenciálható
- Abszolút értékének integrálja véges
- Az négyzetre emelt függvény integrálja véges
- Admissibility: A Fourier transzformáltja elenyészik a nulla frekvencián. Ez azt jelenti, hogy egy sávszűrő szerű spektruma van. Ez egy nagyon fontos kritérium, amire a továbbiakban építeni fogunk. A nulla frekvencián levő 0 azt is jelenti, hogy a wavelet átlagos értéke az időtartományon 0 kell, hogy legyen. Ez csak úgy lehetséges, ha oszcillál, tehát egy hullám.
- Regularity: A függvénynek kell lennie sima részének és meredek részének is mind idő- mind frekvenciatartományban. Ez egy nagyon összetett probléma, amit az úgynevezett eltűnési tényezővel közelítenek meg. Ez annak a mértéke, hogy egy wavelet a frekvenciatartományban hányszor differenciálható folytonosan a 0 helyen.

Amennyiben egy wavelet kielégíti a fenti tulajdonságokat, akkor alkalmas a wavelet transzformációban való felhasználásra. Ennek köszönhetően a transzformáció definiálásakor nincs szükségünk a konkrét függvény ismeretére, elég csak egy sémát adnunk a műveletre. A 2.5 ábra néhány speciális waveletet mutat be.



2.5. ábra. Nevezetes Waveletek

2.2.5.2. Folytonos Wavelet Transzformáció (CWT)

A folytonos wavelet transzformáció (Continous Wavelet Transform) az alábbi módon definiálható:

$$CWT^{\psi}(s,\tau) = \int x(t)\psi^*_{s,\tau}(t)\,dt \tag{2.10}$$

$$\psi_{s,\tau}(t) = \frac{1}{\sqrt{s}}\psi\left(\frac{t-\tau}{s}\right) \tag{2.11}$$

A 2.10 egyenletben található $*$ a komplex konjugáltat jelöli. Erre a komplex függvények konvolúciós tulajdonságainak való megfelelés érdekében van szükség. Ha feltételezzük, hogy a továbbiakban valós függvényeken dolgozunk, akkor ettől eltekinthetünk.

A $\psi(t)$ függvény egy wavelet, melyet **anya waveletnek** nevezünk, mivel ez egyfajta prototípusként szolgál a 2.11 kifejezésben leírt ablakozó függvények (leány waveletek) előállításához. A τ eltolás funkciója megegyezik a STFT estén használt t' paraméterével. Az s scale (nyújtás, dilatáció) paraméter az ablakozó függvények szélességét befolyásolja az anya wavelet összenyomása vagy nyújtása által és így a vizsgált frekvenciára van kihatással. A 2.11 egyenlet integrálja előtt álló tag gondoskodik arról, hogy a wavelet függvény energiája az átméretezés ellenére is normalizált maradjon.

A wavelet transzformáció alkalmazása a 2.10 kifejezés formájában sok esetben nem praktikus a következő megfontolások miatt:

1. **Redundancia**: a wavelet transzformációt úgy számítjuk, hogy egy folytonosan skálázott függvényt folytonosan tolunk végig egy jelen és számítjuk ki a kettő közti korrelációt. Könnyen látható, hogy ezek a skálázott ablakozófüggvények nem függetlenek, így az eredményül kapott információ nagymértékben redundáns. A legtöbb gyakorlati alkalmazásban ezt a redundanciát el szeretnénk kerülni.

2. **Waveletek száma**: A folytonos skálázásnak és eltolásnak köszönhetően végtelen wavelet szerepel a transzformációban. Ezek számát egy kezelhető mennyiségre szeretnénk csökkenteni.

3. **Numerikus megoldás**: A legtöbb esetben a wavelet transzformációnak nincs analitikus megoldása, ezért azt numerikus módszerekkel kell meghatároznunk. Ebben az esetben a célunk egy kellően gyors algoritmus meghatározása.

2.2.5.3. Diszkrét Wavelet Transzformáció (DWT)

Az előző pontban felsorolt problémákra ad megoldást a diszkrét wavelet transzformáció, melynek alapötlete az, hogy az s és τ paraméterek egy folytonos tartomány helyett csak diszkrét lépésekben változhatnak, az alábbi módon:

$$\left.\begin{aligned} s &\in \{s_0^j\} \\ \tau &\in \{k\tau_0 s_0^j\} \end{aligned}\right\} \text{ ahol } \begin{aligned} &j,k \in \mathbb{Z} \\ &s_0 \in \mathbb{R}, 1 < s_0 \\ &\tau_0 \in \mathbb{R}, 0 < \tau_0 \end{aligned} \tag{2.12}$$

Gyakorlati megfontolások miatt legtöbbször a $s_0 = 2$ dilatációs lépésre és a $\tau_0 = 1$ eltolási faktorra esik a választás, ilyenkor a diszkrét wavelet transzformációban szereplő leánywaveletek az alábbi módon írhatók fel:

$$\psi_{j,k}(t) = \frac{1}{\sqrt{2^j}}\psi\left(\frac{t-k2^j}{2^j}\right) \tag{2.13}$$

Redundancia kiküszöbölése

Megfelelő anyawaveletek választásával elérhető, hogy ne veszítsünk információt a jel szempontjából (a jel visszaállítható a transzformáltjából) és a 2.13 kifejezésben megadott módon előállított waveletek ortogonálisak legyenek, vagyis:

$$\int \psi_{j,k}(t)\psi_{m,n}(t)dt = \begin{cases} 1 & \text{ha } j = m \text{ és } k = n \\ 0 & \text{egyébként} \end{cases} \tag{2.14}$$

Az ilyen waveletek esetén megszűnik a redundancia problémája.

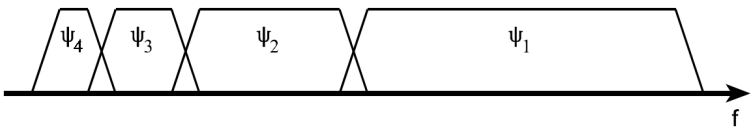
Waveletek számának csökkentése

Annak ellenére, hogy a waveletek folytonos dilatációját és eltolását megszűntettük, a wavelet transzformációban továbbra is (megszámlálhatóan) végtelen waveletet szerepel, hiszen a j és k paraméterek értékére egyelőre nem adtunk korlátot.

Ezek közül az eltolásért felelős k értékre könnyen megadható egy alsó- és egy felső korlát, melyet a vizsgált jel időtartománya határoz meg.

A feladatunk tehát a j dilatációs paraméter határait megállapítani. Ehhez a következő megállapítások adnak segítséget:

- Láthattuk, hogy a waveletek spektruma egy sáváteresztő szűrőként fogható fel
- A wavelet kétszeres szélességre való nyújtása az időben ($s_0 = 2$) azt eredményezi, hogy a frekvenciatartománya felére csökken és a frekvencia komponensei eltolódnak 1/2 faktorral
- Ilyen módon az összenyomott waveleteket egy sáváteresztő szűrőkészletként (filter bank) képzelhetjük el, melyekkel úgy fedhetjük le a véges spektrumot, mint ahogyan az eltolásokkal lefedtük az időtartományt. Ezt szemlélteti a 2.6 ábra.



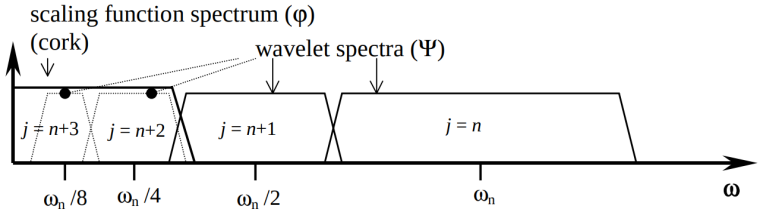
2.6. ábra. A frekvenciatartomány lefedése skálázott waveletek spektrumaival [19]

Megfigyelhetjük, hogy a waveletek egyszeri nyújtásával a frekvenciatartomány lefedetlen részének mindig csak felét takarjuk le. Ebből adódóan a teljes lefedéshez továbbra is végtelen waveletre lenne szükségünk. Hogy ezt kiküszöböljük, meghatározhatunk egy felső korlátot a waveletek számára és az így lefedetlenül maradt tartomány lefedésére definiálhatunk egy aluláteresztő szűrőként használható spektrumot. Ez a spektrum az úgynevezett skálafüggvény spektruma lesz.

A **skálafüggvény** a fentieknek megfelelően felírható egy adott j skála index feletti waveletek összegeként

$$\varphi(t) = \sum_{j,k} \gamma(j,k)\psi_{j,k}(t) \tag{2.15}$$

Ahol $\gamma(j,k)$ a diszkrét wavelet transzformáció kimenete. A skálafüggvény által lefedett spektrumot a 2.7 ábra szemlélteti.



2.7. ábra. A skálafüggvény végtelen wavletet helyettesít [19]

A skálafüggvény bevezetésével megoldottuk a waveletek végtelen számának problémáját. Természetesen a skálafüggvény használatával veszítünk a wavelet transzformáció által előállított információból, ezért ennek szélessége a wavelet transzformáció egy fontos paramétere.

A skálafüggvényekre a waveletekhez hasonló megfelelőségi feltétel fogalmazható meg, mely azt mondja ki, hogy

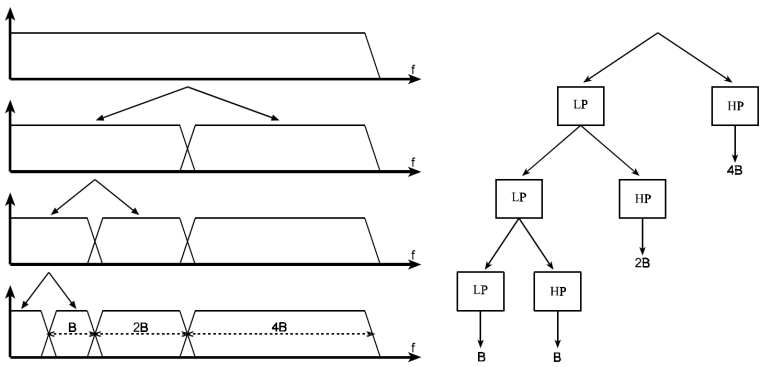
$$\int \varphi(t)\, dt = 1 \tag{2.16}$$

Összefoglalva tehát, ha egy wavelet felfogható egy sáváteresztő szűrőként és a skálafüggvény felfogható egy aluláteresztő sszűrőként, akkor elnyújtott waveletek egy sorozata egy skálafüggvénnyel együtt egy olyan szűrőkészletet alkotnak, mely képes lefedni a teljes tartományt. Ezzel megoldottuk a végtelen wavelet problémáját.

Gyakorlati algoritmus

A diszkrét wavelet transzformációt a gyakorlatban az ismertetett szűrőkészlet elvét felhasználva valósítják meg, de ahelyett, hogy a waveleteket előre definiálnák, egy iteratív módszert alkalmaznak a jel felbontásra, mely az alábbi módon történik:

A jel spektrumát egy alul- és egy felüláteresztő szűrő segítségével két (egyenlő méretű) részre bontjuk. A felüláteresztő szűrő kimenete tartalmazza a legfinomabb részleteket és ha ezekkel megelégszünk, akkor meg is állhatunk. Ekkor két sávunk van. Az aluláteresztő szűrő kimenete viszont még tartalmazhat további részleteket, melyek hasznos információt tartalmaznak a számunkra. Ekkor ezt sávot szintén egy alul- és egy felüláteresztő szűrővel tovább bonthatjuk. Ezt a folyamatot addig ismételhetjük, amíg a felbontásban szereplő frekvenciasávok számával elégedettek nem vagyunk, vagy az adatok nem teszik lehetővé a további felbontást. Ezt a folyamatot ábrázolja a 2.8 ábra.



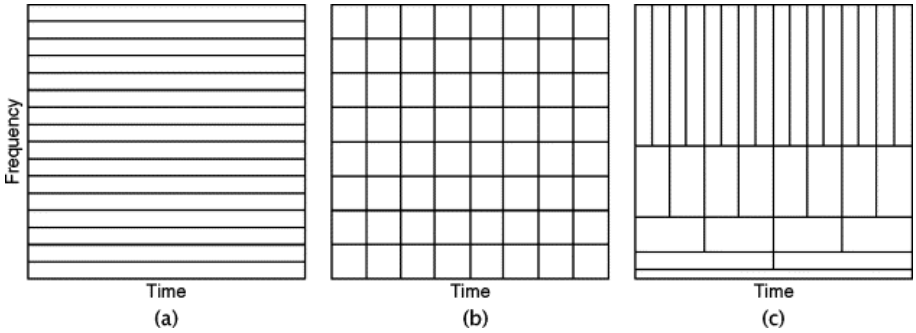
2.8. ábra. A jel spektrumának iteratív felbontása [19]

A folyamat során az aluláteresztő szűrőket a skálafüggvény, a felüláteresztő szűrőket pedig a wavelet függvény szolgáltatja számunkra. (A wavelet függvény a vizsgált frekvenciatartomány felső határán található, ezért felfogható felüláteresztő szűrőként)

Megjegyzendő, hogy a diszkrét wavelet transzformáció név félrevezető, ugyanis a művelet csak a skálázás és az eltolás szempontjából diszkrét. A waveletek és a jel egyaránt folytonosak. Természetesen az ismertetett módszer diszkrét jelek analízisére is kiterjeszthető, ilyenkor diszkrét idejű diszkrét wavelet transzformációról beszélünk.

2.2.5.4. A wavelet analízis összefoglalása

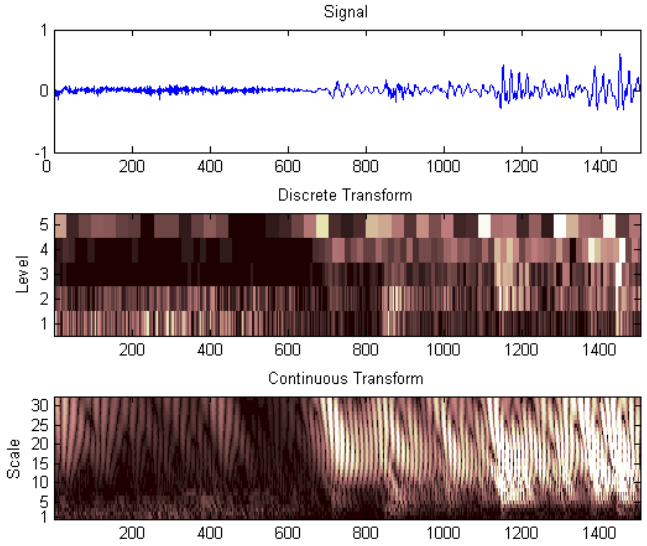
A transzformáció által megvalósított multirezolúciós analízis célját jól szemlélteti a 2.9 ábra, melyen a fent ismertetett módszerek spektrális felbontását szemlélteti.



2.9. ábra. Az ismertetett módszerek idő-frekvencia lefedése.[10]
(a) FT, (b) STFT, (c) DWT

Látható, hogy míg a Fourier transzformáció egyáltalán nem szolgáltat idő információval, a rövid idejű Fourier transzformáció pedig egyenletes idő-frekvencia információt mutat, addig a wavelet transzformáció időbeli felbontása igazodik a különböző frekvenciákhoz.

A 2.10 egy példa egy adott jel wavelet transzformáltjaira.



2.10. ábra. Egy jel diszkrét és folytonos wavelet transzformáltja

A wavelet transzformáció számos alkalmazás esetén hasznos eszköznek bizonyult, mint az adat tömörítés, képfeldolgozás, mintafelismerés, jelfeldolgozás, zajcsökkentés, szűrés, statisztikiai elemzések stb. A következő részben egy hasznos alkalmazást láthatunk az elektronszerkezet számítások területén.

2.3. Kételektron-integrálok wavelet alapú számítása[18]

A 2.1.2 részben bemutatott Fock mátrix elkészítése minden iterációs lépésben $\mathcal{O}\left(N_B^4\right)$ kételektron integrál számítását követeli meg. Ezek az integrálok ráadásul az adott fizikai rendszertől függenek, mivel a különféle atomok atompályái különböző természetűek. Ez a két probléma kiküszöbölhető egy, a wavelet elmélet területéről átvett univerzális skálafüggvény bázis bevezetésével.

Az atomi bázisfüggvényeken végzett számításokhoz (LCAO) képest a wavelet megközelítés egy sokkal gazdaságosabb sémát nyújt, mivel a kételektron-integrálok egyszeri kiszámítása után azok újrahasználhatóak további molekulák esetén. Ezen felül kompakt tartójú skálafüggvényeket használva a legtöbb kételektron-integrál értéke pontosan nulla, így csökkentve a Fock matrix számítási igényét [13].

2.3.1. Skálafüggvények használata

A alfejezet célja bemutatni, hogy hogyan alkalmazhatók a skálafüggvények a kételektron-integrál számítások bázisfüggvényeiként. Az egyik nehézség az integrálok hatalmas mennyisége. Ezért, először azt mutatjuk meg, hogy ez a szám nagy mértékben csökkenthető kompakt tartójú skálafüggvények használatával, valamint sok olyan esetben, amikor az integrálok értéke nullától különböző, egy nagyon jó minőségű közelítést találhatunk.

A bázisfüggvények esetünkben egy valós értékkészletű $s(r)$ anya skálafüggvény eltolt és átméretezett verziói.

$$\chi_\mu(r) \to s_{mk}(r) = 2^{m/2} s(2^m r - k) \tag{2.17}$$

Ebből adódóan a $(mk, ml \mid mq, mt) = I_{kl,qt}^{(m)}$ kételektron-integrálok az alábbi módon írhatók fel:

$$I_{kl,qt}^{(m)} = \int\int dr_1 dr_2 2^{2m} \frac{s(2^m r_1 - k)s(2^m r_1 - l)s(2^m r_2 - q)s(2^m r_2 - t)}{|r_1 - r_2|} \tag{2.18}$$

ahol m a nyújtás, k, l, q, t pedig eltolások egy szabályos rács élei mentén. Az $r_1' = 2^m r_1$ és $r_2' = 2^m r_2$ helyettesítést használva

$$I_{kl,qt}^{(m)} = 2^{2m} \int\int 2^{-m} dr_1' 2^{-m} dr_2' \frac{s(r_1' - k)s(r_1' - l)s(r_2' - q)s(r_2' - t)}{|2^{-m} r_1' - 2^{-m} r_2'|} \tag{2.19}$$

ami a következő alakra hozható:

$$I_{kl,qt}^{(m)} = 2^m \int\int dr_1' dr_2' \frac{s(r_1' - k)s(r_1' - l)s(r_2' - q)s(r_2' - t)}{|r_1' - r_2'|} \tag{2.20}$$

Az $r_1'' = r_1' - k$ és $r_2'' = r_2' - q$ változók bevezetésével

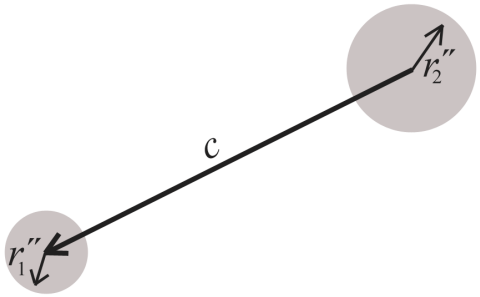
$$I_{kl,qt}^{(m)} = 2^m \int\int dr_1'' dr_2'' \frac{s(r_1'')s(r_1'' + k - l)s(r_2'')s(r_2'' + q - t)}{|r_1'' + k - r_2'' - q|} \tag{2.21}$$

Valójában az integrál a k, l, q, t négyes helyett csak az $a = k - l, b = q - t$ és $c = k - q$ kombinációktól függ. (Ez már korábban bemutatásra került [13])

$$I_{abc}^{(m)} = 2^m \int\int dr_1'' dr_2'' \frac{s(r_1'')s(r_1''+a)s(r_2'')s(r_2''+b)}{|r_1'' - r_2'' + c|} \tag{2.22}$$

Kompakt tartójú skálafüggvények használatával [13], melyek egy véges intervallumon belül elenyésznek $s(r_1'')s(r_1''+a) = 0$, ha $a \geq N$ és $s(r_2'')s(r_2''+b) = 0$, ha $b \geq N$, ahol N a tartó mérete. Ennek köszönhetően az integrálok csak abban az esetben különböznek nullától, amikor a és b kisebb, mint N. Így a kételektron-integrálok száma jelentősen lecsökken. A c paraméter az egyetlen változó, amely tetszőleges egész értéket felvehet.

A fenti megfontolások figyelembe vételével, az a és b értékek kis számok, míg c sok esetben nagy. Megmutatjuk, hogy ezekben az esetekben egy hasznos approximáció adható. Az $s(r_1'')s(r_1''+a)$ szorzat egy "eloszlásként" értelmezhető, amely Coulomb-féle taszítási kölcsönhatásban van a másik, $s(r_2'')s(r_2''+b)$ "eloszlással". Az eloszlások kiterjedése elenyésző a köztük lévő térbeli c távolsághoz képest. Ezt mutatja be a 2.11 ábra:



2.11. ábra. Áttekintő ábra az 2.22 kifejezésben szereplő integrálok kiértékeléséről nagy térbeli c szeparáció esetén

Mivel a 2.22 integrálban szereplő r_1'' változó csak kis térbeli tartományban változik amikor az $s(r_1'')s(r_1''+a)$ szorzat nem nulla, ezért a numerikus értéke kicsi és ugyanez mondható el r_2''-ről is. A nevező közelíthető $|r_1'' - r_2'' + c| \approx |c|$ értékkel, feltéve hogy $|c|$ nagy. Ezt a közelítést alkalmazva a

$$I_{abc}^{(m)} = \frac{2^m}{|c|} \int dr_1'' s(r_1'')s(r_1''+a) \cdot \int dr_2'' s(r_2'')s(r_2''+b) = \frac{2^m}{|c|}\delta_a\delta_b \tag{2.23}$$

kifejezést kapjuk, ahol a Kronecker-delta $\delta_a = 1$ ha $a = 0$, egyébként $\delta_a = 0$. A 2.23 második azonossága az egymáshoz képest eltolt skálafüggvények ortogonalitásából következik. Ez a közelítés kis c esetén pontatlan, de a komplikált számításokat kerülhetünk el a segítségével.

2.4. Integrálszámítás a gyakorlatban

A 2.1 fejezetben megismertük a kételektron-integrálokat, majd a 2.2 szekcióban a wavelet transzformációk elméletébe nyertünk betekintést. A 2.3 szakaszban láthattuk, hogy a waveletek skálafüggvényei hogyan alkalmazhatóak a kételektron-integrálok számítására. Ezzel eljutottunk a függvényhez, amely a továbbiakban a vizsgálatunk tárgyát képezi:

$$I_{abc}^{(m)} = 2^m \int\int dr_1'' dr_2'' \frac{s(r_1'')s(r_1''+a)s(r_2'')s(r_2''+b)}{|r_1'' - r_2'' + c|} \tag{2.22}$$

Látható, hogy a magasabb dimenziótól eltekintve egy "hagyományos" határozott integrálszámítási problémával állunk szemben, így az optimalizációs törekvések érdekében érdemes megvizsgálnunk a számítógépes integrálszámítás témakörét.

2.4.1. Algebrai és numerikus megközelítés

A tudományos és mérnöki munka során egy probléma levezetése általában egy- vagy több megoldandó egyenletre vezet. Ezek megoldására általában két szemléletmódot alkalmazhatunk.

- A ***numerikus analízis*** egy kiforrott terület. Eszközei jól használható matematikai módszerek, melyekkel egy egyenlet kiértékelését végezhetjük el, annak egy **közelítő** értékét határozhatjuk meg.
- Ezzel szemben a ***számítógép-algebra*** a számítástudomány egy jóval fiatalabb ága. Módszerei magasabb absztrakciós szintű matematikai objektumokon végeznek algebrai átalakításokat. Konkrét számértékek helyett szimbolikus műveleteket végeznek és szimbolikus, **pontos** eredményt szolgáltatnak.

A továbbiakban ezt a kettő megközelítést részletezem. Először a számítógép-algebra témakörét mutatom be ismertető jelleggel, majd a numerikus integrálszámításra térek át. Ennek néhány eszközére - melyeket a megoldás során is alkalmazni fogok - részletesebb rálátást kívánok nyújtani.

2.4.2. Számítógép-algebra

A számítógép-algebra a matematikának és ezen belül a számítástudománynak az az ága, mely olyan algoritmusok tervezésével, elemzésével és implementációjával foglalkozik, melyek matematikai kifejezéseket manipulálnak és elemeznek. Míg a "hagyományos" algebrában a megfigyelés tárgyai egy algoritmus működését biztosító általános szabályok, addig a számítástudományban ez maga az algoritmus, annak minden aspektusával, beleértve a hatékonyságát és implementációját, vezérlési struktúráit stb. A számítógép-algebra szisztematikusan tanulmányozza a hatékonytalanság forrásait és alternatív algebrai módszerek után keres, hogy javítsa, vagy teljesen lecserélje az algoritmusokat.[4]

Az algebrai algoritmusok egyszerű, formális specifikációval rendelkeznek, vannak helyességi bizonyításaik és az objektumaik pontosan ábrázolhatóak a számítógép memóriájában. Ennek köszönhetően az algebrai számítások pontosságvesztés nélkül elvégezhetők. Magától értetődő, hogy a számítógép-algebra nagy érdeklődést vonz a számítástudomány, matematika és a mérnöki munka különböző területeiről. Az ötlet, hogy használjunk számítógépeket nem numerikus számítások elvégzésére viszonylag régi, de ennek a tényleges megvalósítása viszonylag új.

2.4.2.1. Számítógép-algebrai rendszerek

Azokat a konkrét alkalmazásokat, melyek a számítógép-algebra módszereire építve végeznek számításokat, számítógép-algebrai rendszereknek (CAS - Computer Algebra System), vagy szimbólum manipulációs rendszereknek nevezzük. Az a tulajdonságuk, hogy képesek szimbolikus matematikai objektumok manipulálására, számos lehetőséget nyit meg számukra. A legtöbb olyan algebrai művelet elvégzése, melyekkel az ember a középiskolai és egyetemi tanulmányai során találkozik, megvalósítható ezekkel a rendszerekkel. Tulajdonképpen bármely algebrailag felépíthető matematikai művelet elméletileg megoldható egy algebrai rendszer segítségével is. [7]

A számítógép-algebrai rendszerek legnagyobb előnye, hogy nagy komplexitású algebrai számítások elvégzésére képesek. Ennek köszönhetően nem feltétlenül kell a klasszikus tankönyvi algoritmusokat használniuk, melyek gyakran hatékonytalannak bizonyulnak. Kettő racionális együtthatójú polinom legnagyobb közös osztójának meghatározása történhet az Euklideszi algoritmussal, de ha ezt nagyobb méretű polinomokra szeretnénk alkalmazni, akkor azzal szembesülünk, hogy a számítás köztes kifejezéseinek mérete exponenciálisan nő. Ennek következtében az algoritmus gyorsan praktikátlanná válik ahogyan a probléma mérete nő. Az ismert algoritmusok nagy része ebbe a kategóriába tartozik. Megjegyzendő, hogy azok a programok, amik tisztán numerikus számításokat végeznek, nem szenvednek ettől a problémától, mivel azok fix méretű számokat tartanak nyilván a matematikai kifejezések reprezentációja helyett.

Számos számítógép-algebrai rendszer létezik, ilyenek például a Maple, MuPAD, Macsyma, Reduce, Axiom vagy a Wolfram Mathematica.

2.4.2.2. Kapcsolat a numerikus analízissel

Az összehasonlítás érdekében hasznos lehet egy konkrét példán szemléltetni a kontrasztot a numerikus és szimbolikus számítások között. Az úgynevezett Chebysev polinomok az alábbi rekurzív módszerrel definiálhatók:

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_k(x) &= 2xT_{k-1}(x) - T_{k-2}(x) \text{ ha } k \geq 2 \end{aligned} \tag{2.24}$$

Az alábbi táblázat pedig azt mutatja be, hogy egy szimbolikus és egy numerikus módszer adott k érték mellett milyen kimenetet adna a 2.24 definícióra.

k	Szimbolikus	Numerikus ($x = 0.30$)
0	1	1.0000
1	x	0.3000
2	$2x^2 - 1$	-0.8200
3	$4x^3 - 3x$	-0.7920
4	$8x^4 - 8x^2 + 1$	0.3448

A számítógép-algebra sok esetben érdekesebb válaszokat nyújt, mint a tradícionális számítási módszerek. Egy matematikai elemzés során sokkal több információt kaphatunk egy x-re vonatkozó kifejezésből, mint a konkrét x helyeken felvett függvényértékek halmazából.

Számos esetben a számítógép-algebrai számítás kimenete egy képlet, amit aztán numerikus eljárások bemeneteként használunk fel. Vegyük például a gyök keresés problémáját. A numerikus módszerek véges precíziójú aritmetikát használnak és a közelítés elvén alapulnak. Egy numerikus gyök kereső ebből adódóan nem feltétlenül képes a gyökök elkülönítésére, csak "klasztereket" képes meghatározni, melyen belül a gyökök egy bizonyos távolságon belül találhatók. Ilyen esetekben célravezető lehet az egyes gyökök szimbolikus megoldása, majd a konkrét számértékek numerikus meghatározása. Elméletileg lehetséges és érdemes a numerikus algoritmusokat az algebrai algoritmusokkal azonos szigorúsággal definiálni, de az ehhez szükséges részletek negatívan hatnak az algoritmus és a konkrét matematikai probléma közti megfeleltethetőségre. [7]

Az alkalmazandó módszer megválasztásakor azt is érdremes mérlegelni, hogy egy algebrai algoritmus esetén a pontosságnak költsége van, ami általánosságban jóval nagyobb végrehajtási időt és tárhely igényt jelent a numerikus megoldással szemben. Természetesen

vannak esetek, amikor egy approximációnak nincs értelme, ilyenkor elkerülhetetlen a szimbolikus megoldás.

2.4.2.3. Szimbolikus integrálszámítás [21]

Vizsgáljuk meg, hogy a számítógép-algebrai módszerek hogyan alkalmazhatók az integrálszámítás problémájára. Nevezzük *kifejezésnek* az olyan matematikai entitásokat, melyek csak racionális törtekből és olyan elemi függvényekből épülnek fel, mint a *sin*, *cos*, *exp*, *log* stb. Legyen adott egy f kifejezés. A célunk az, hogy meghatározzuk azt a

$$g = \int f = \int f(x)\,dx \tag{2.25}$$

integrált, melyre g is egy kifejezés.

A szimbolikus integrálszámítás problémájának egy gyakorlati megközelítése lehet az, ha rengeteg képletet felsorakoztatunk bizonyos speciális függvényekhez. Ilyenek a kalkulus alapvető trükkjei, mint például a helyettesítés és a parciális integrálás. Ebben az esetben táblázatos módon kikereshetjük az illeszkedő megoldást.

A szimbolikus integrálszámítás egyik akadálya, hogy a 2.25 képletben definiált g kifejezéseknek nem feltétlenül létezik. Erre egy példa Liouville tétele, amely éppen azt mondja ki, hogy az $e^{(x^2)}$ kifejezésnek nem létezik olyan integrálja, ami csak racionális törtekből valamint *sin*, *cos*, *exp* és *log* függvényekből tevődik össze.

2.4.3. Numerikus integrálszámítási módszerek

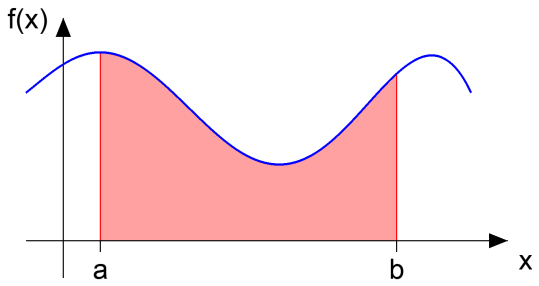
A numerikus integrálszámítási módszerek megismerése előtt vizsgáljuk meg azokat a fontosabb szempontokat, amik a közelítő megoldások alkalmazását általánosságban indokolttá tehetik:

1. Bizonyos egyenletekre nem lehetséges zárt formulájú megoldást találni. Erre láthattuk az $\int e^{(x^2)}\,dx$ kifejezés példáját a 2.4.2.3 szekcióban, de természetesen több egyenlet létezik, melyekről ez szintén bizonyítotható.

2. Felmerülhet a hatékonyság kérdése. A lineáris egyenletrendszerek megoldására jól bevált eljárásaink vannak, azonban a gyakorlatban gyakran találkozhatunk olyan méretű egyenletrendszerekkel, melyek megoldását a korszerű hardvereken futtatva is napokba, vagy hetekbe telne meghatározni. Ilyenek például a geológiai vagy meteorológiai számítások. Bár egy időjárás előrejelzés pontos kiszámítása erőforrásigényes lenne, a numerikus módszerekkel mégis kellő pontosságú közelítést adhatunk.

3. Számos gyakorlati esetben már a kiindulási alapunk sem egy szimbolikus kifejezés, csak adatok egy mintavételezett sorozata (megfigyelések, mérések)

A bekezdés további részeiben határozott integrálszámítás problémájára vonatkozó numerikus megközelítést ismertetem. A határozott integrálszámítás nem formális jelentése a függvénygörbe alatti terület megállapítása.



2.12. ábra. Görbe alatti terület

Mivel a határozott integrál definíciója magában foglalja a numerikus integrálszámítási módszerek alapgondolatait, ezért érdemes ennek a megismerésével kezdenünk [6].

- Az integrálandó függvényünk legyen $f : [a,b] \Rightarrow \mathbb{R}$ és f legyen korlátos $[a,b]$ intervallumon
- Osszuk fel az $[a,b]$ intervallumot n részre és jelöljük az osztópontokat x_k -val

  $$k = 0, 1..., n;\ a = x_0 < x_1 ... < x_{n-1} < x_n = b$$

  A k-adik részintervallum jelölése $I_k = [x_k - 1, x_k]$, hossza $\Delta x_k = x_k - x_{k-1} > 0$
- $[a,b]$ egy felosztása : $F = \{I_k : k = 1, 2, ..., n\}$
- Definiáljuk az F felosztáshoz tartozó alsó- és felső közelítőösszeget az alábbi módon:

  $$(a.k.\ddot{o}) = s_F = \sum_{k=1}^{n} m_k(x_k - x_{k-1}) = \sum_{k=1}^{n} m_k \Delta x_k \text{ , ahol } m_k = \inf_{I_k}\{f(x)\}$$

  $$(f.k.\ddot{o}) = S_F = \sum_{k=1}^{n} M_k(x_k - x_{k-1}) = \sum_{k=1}^{n} M_k \Delta x_k \text{ , ahol } M_k = \sup_{I_k}\{f(x)\}$$

- Belátható, hogy $s_F \leq S_F$
- Belátható, hogy ha az F felosztást úgy finomítjuk, hogy az eddigi osztópontok mellé egy új osztópontot veszünk fel, akkor az *alsó közelítő összeg* nem csökkenhet, a *felső közelítő összeg* pedig nem nőhet, tehát ha a leszármaztatott felosztás jelölése F', akkor

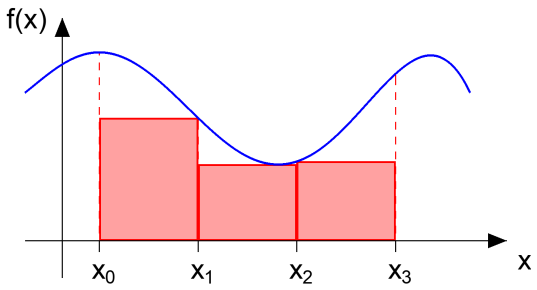
  $$s_F \leq s_{F'} \leq S_{F'} \leq S_F$$

- Belátható, hogy létezik $sup\{s_F\} = h$ és $inf\{S_F\} = H$

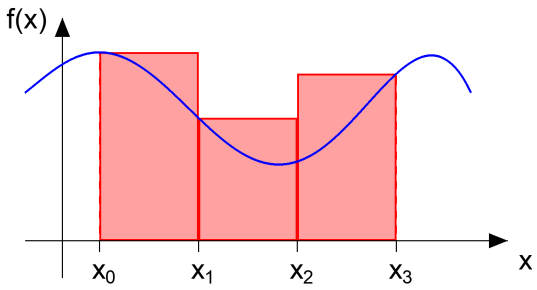
Azt mondjuk, hogy az f függvény az $[a,b]$ intervallumon Riemann szerint integrálható, ha $h = H = I$. Ezt a közös I számot az f függvény $[a,b]$-beli határozott integráljának nevezzük és

$$I = \int_a^b f(x)\,dx$$

módon jelöljük.



2.13. ábra. Alsó közelítő összeg (s_F)



2.14. ábra. Felső közelítő összeg (S_F)

Látható tehát, hogy a határozott integrál definíciójában a függvény integrálási tartományát szakaszokra osztottuk. Két osztópont között a függvényt egy téglalapnak tekintettük. Arra jutottunk, hogy a felosztás folyamatos finomításával a tényleges integrál értékéhez közelítünk. Általában nincsen lehetőségünk tetszőlegesen finom felosztásra, ezért a numerikus integrálszámítási módszerek azon az elven alapulnak, hogy az integrálandó függvényt néhány pontban kiértékeljük majd ezeket a függvényértékeket súlyozzuk és összegezzük hogy megkapjuk az integrál egy közelítő értékét. A mintavételezés és a súlyozás miként je az adott módszerek sajátossága.

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n} w_n f(x_i) \tag{2.26}$$

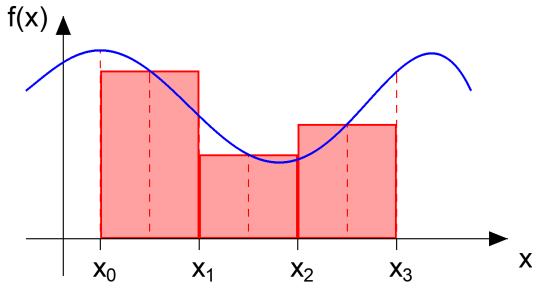
A továbbiakban néhány gyakori numerikus integrálási szabályt ismertetek, melyeket más néven kvadratúra szabályoknak is nevezünk.

2.4.3.1. Középpont szabály

A fenti definícióban az integrál értékére egy alsó- és egy felső becslést is adhattunk, mivel minden részintervallumban kiválaszthattuk a függvény minimum és maximum értékét. Ez a gyakorlatban nem praktikus, célszerűbb a közelítő téglalap magasságának egy könnyebben definiálható (és számítható) értéket megadni.

A középpont szabály a téglalapok magasságának az adott intervallum felezőpontján felvett értéket veszi, tehát

$$\int_a^b f(x)\,dx \approx \sum_{i=0}^{n-1} hf\left(\frac{x_i + x_{i+1}}{2}\right) \tag{2.27}$$



2.15. ábra. Középpont szabály

2.4.3.2. Riemann integrál[11]

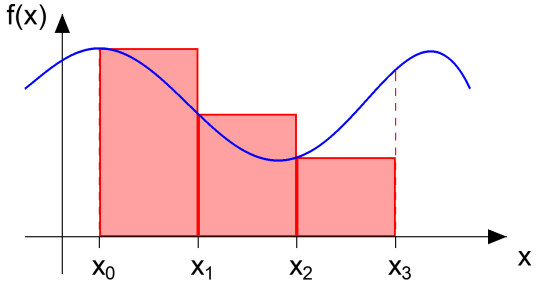
Kézenfekvő választás lehet a téglalap magasságának a szakasz két végpontjának valamelyikét, x_i-t vagy x_{i+1} -et választani. Ennek megfelelően rendre bal- illetve jobboldali Riemann integrálról beszélünk.

$$\int_a^b f(x)\,dx \approx \sum_{i=0}^{n-1} hf(x_i) \tag{2.28}$$

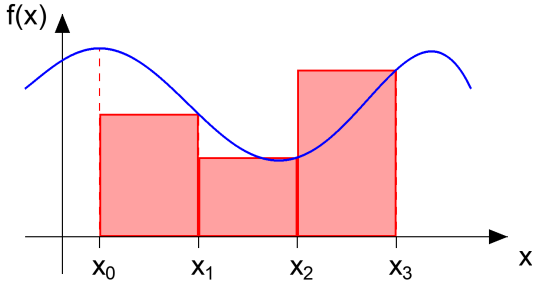
vagy

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n} hf(x_i) \tag{2.29}$$

ahol $h = x_{i+1} - x_i$



2.16. ábra. Baloldali Riemann integrál



2.17. ábra. Jobboldali Riemann integrál

2.4.3.3. Trapézszabály

Az egyik leggyakrabban alkalmazott numerikus integrálási módszer. A trapézszabály azzal a feltételezéssel él, hogy az f függvény az $[x_i, x_{i+1}]$ intervallumon lineáris, tehát az integrálja itt egy trapézzal közelíthető:

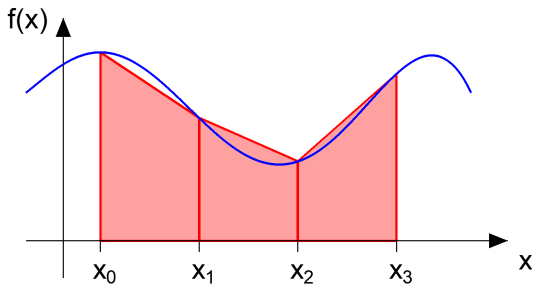
$$\int_{x_i}^{x_{i+1}} f(x)\,dx \approx \frac{1}{2}h_i[f(x_i) + f(x_{i+1})], \text{ ahol } h_i = x_{i+1} - x_i \tag{2.30}$$

Ebből levezethető a teljes integrál közelítése

$$\int_a^b f(x)\,dx \approx \frac{1}{2}\sum_{i=1}^{n-1} h_i[f(x_i) + f(x_{i+1})] \tag{2.31}$$

Illetve, ha feltételezzük, hogy az $[x_i, x_{i+1}]$ intervallumok egyenlő nagyságúak, azaz $x_i = a + i\Delta x$ és $\Delta x = h = h_i = \frac{b-a}{n}$, akkor

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\sum_{i=1}^{n-1} f(x_i) + f(x_{i+1}) \tag{2.32}$$



2.18. ábra. Trapézszabály

A trapézszabály a gyakorlati alkalmazások többségében kielégítő megoldást nyújt. Azokban az alkalmazásokban, ahol nagyobb pontosságra van szükségünk, mint amit a trapézszabály nyújtani tud, magasabb fokú integrálási sémák elérhetőek. A legnépszerűbb ezek közül a Simpson szabály, ami parabolikus interpoláción alapul.

2.4.3.4. Simpson szabály

Az úgynevezett 1/3-os Simpson szabály az intervallum három pontjára egy parabolát illeszt és az eredeti függvény határozott integrálját a parabola határozott integráljával becsli. Levezethető, hogy ez a megoldás minden esetben ekvivalens a következő kifejezéssel:

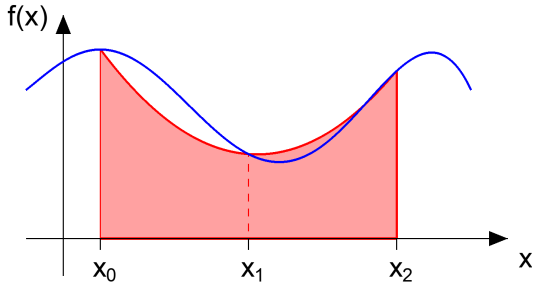
$$\int_a^b f(x)\,dx \approx \frac{b-a}{6}[f(a) + 4f(\frac{a+b}{2}) + f(b)] \tag{2.33}$$

Ezzel a módszerrel jól közelíthetünk egy függvényt, amennyiben az az $[a, b]$ intervallumon kellően sima. Amennyiben egy bonyolultabb függvényt szeretnénk integrálni, akkor az összetett Simpson szabályt kell alklamaznunk, ami az előző módszerekhez hasonlóan a műveletet az egyes részintervallumokra alkalmazza. A Simpson szabály képlete egy h távolságot felölelő ponthármasra nézve

$$\int_{x_1}^{x_3} f(x)\,dx \approx h[\frac{1}{3}f(x_1) + \frac{4}{3}f(x_2) + \frac{1}{3}f(x_1)] \tag{2.34}$$

Ezt a teljes függvényre kifejtve a következő kifejezést kapjuk:

$$\int_{x_1}^{x_n} f(x)\,dx \approx \frac{h}{3}[f(x_1) + 4f(x_2) + 2f(x_3) + 4f(x_4) + ... + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)] \tag{2.35}$$



2.19. ábra. Simpson szabály

A Simpson szabály páratlan számú ponton használható, de kiterjeszthető páros számú felosztásra is ha az egyik szakaszt valamely más módszerrel, például a trapéz módszerrel értékeljük ki.

2.4.3.5. Egyéb megoldások

Az eddigiekben ismertetett formulákat összefoglaló néven Newton-Cotes kvadratúráknak nevezzük. A Simpson szabályban található interpolációnál magasabb rendű formulákat is alkalmazhatunk, de a gyakorlatban ezek használata nem jellemző. Amikor ezek a módszerek nem képesek kielégítő eredményt nyújtani, a gyakorlatban általában más megközelítést alkalmazó módszereket használnak.

Ilyen például a népszerű Gauss interpoláció. Mint kiderül, ha engedünk abból a feltételből, hogy az intervallumot egyenlő távolságokra kell osztani, akkor tudunk olyan speciális osztópontokat és súlyokat választani, melyek nagyon nagy pontosságot eredményezhetnek. Ez a Gauss-féle integrálási módszer alapja, amely az alábbi formájú integrálokon alapul

$$\int_a^b W(x)f(x)\,dx \tag{2.36}$$

és a következő összeggel közelíthető:

$$\sum_{i=1}^{N} w_i f(x_i) \tag{2.37}$$

A w_i súlyok és az x_i osztópontok megválaszthatók olyan módon egy adott $W(x)$ mellett, hogy az eredmény pontos eredményt ad, amennyiben $f(x)$ egy polinom. Ha $f(x)$ jól közelíthető egy polinomfüggvénnyel, akkor ez a képlet szintén nagy pontosságot ad. Különböző $W(x)$-ek esetén különböző választásokat adhatunk a súlyoknak és az osztópontoknak, melyek a Gauss módszer speciális eseteit definiálják. A $W(x) = 1$ esetet például "Gauss-Legendre" integrálnak nevezzük, $W(x) = (1 - x^2)^{-\frac{1}{2}}$ esetén pedig "Gauss-Chebyshev" integrálról beszélünk.

2.4.3.6. Többdimenziós integrálok

A többdimenziós függvények integrálása számos problémával jár. Ahhoz, hogy megfelelő pontosságú közelítést adhassunk, a numerikus integrálás a függvénynek akár több tíz- százezer pontban való kiértékelését is megkövetelheti. Amennyiben az integrálandó tartomány határai nem szabályosak, az nagy mértékben bonyolíthatja a procedúrát. Ennek megfelelően amikor lehetséges, a többdimenziós technikák alkalmazása előtt megpróbálhatjuk csökkenteni az integrál dimenzióit. Erre egy példa lehet, amikor egy gömbszimmetrikus függvényt integrálunk egy gömb régió felett. Gömbi koodrinátarendszerre váltva az integrálunk egydimenziósra redukálható. Amennyiben ezt nem tudjuk megtenni és a függvény egyébként "egyenletes", tehát nincsenek erősen kicsúcsosodó területei, valamint nem igénylünk nagy pontosságot, megpróbálkozhatunk a Monte-Carlo technikákkal.

Amennyiben mégis a többdimenziós integrál elvégzésére hagyatkozunk, azt az egydimenziós módszerekkel közelíthetjük a következő módon. Tegyük fel, hogy a többdimenziós integrálunk az alábbi alakú:

$$I = \int_{x_1}^{x_2} dx \int_{y_1(x)}^{y2(x)} dy \int_{z_1(x,y)}^{z_2(x,y)} f(x,y,z)\,dz \tag{2.38}$$

Ha azt mondjuk, hogy

$$G(x,y) = \int_{z_1(x,y)}^{z_2(x,y)} f(x,y,z)\,dz \tag{2.39}$$

és

$$H(x) = \int_{y_1(x)}^{y_2(x)} G(x,y)\,dy \tag{2.40}$$

akkor

$$I = \int_{x1}^{x2} H(x)\,dx \tag{2.41}$$

A harmadik, I integrál kiértékelhető valamely egydimenziós technikával. Ebben az esetben azonban szükségünk van $H(x)$ értékeire bizonyos x helyeken. Hasonló módon a $H(x)$ kiszámítása bizonyos $G(x,y)$ értékek meghatározását követeli meg. $G(x,y)$ pedig csak úgy számítható ki, ha elvégezzük az integrált minden olyan x és y értékre, melyek előkerültek az I és H integrálok kiértékelésekor. Láthatjuk, hogy az ilyen módon egymásba ágyazott integrálok az alsóbb szinteken lévő kifejezések egyre nagyobb- és nagyobb számú meghatározását igénylik.

2.4.3.7. Monte-Carlo módszer

Az előző részben megemlítettük, hogy kellően sima függvények esetén a Monte-Carlo módszert is alkalmazhatjuk az integrál közelítésére. Ez a megoldás az értékek véletlenszerű mintavételezésén alapul. Tegyük fel, hogy van egy f integrálunk a V tér felett és választunk N darab pontot ebben a térben. Ekkor a módszer alkalmazásakor a következő közelítést adjuk:

$$\int\int\int_V f\,dV \approx V\frac{1}{N}\sum_{i=1}^{N} f(x_i) \tag{2.42}$$

Érdemes megjegyezni, hogy a módszer hibája $\sqrt{1/N}$ -el arányos, ami az egyik nagy hátránya: A hiba az N növelésével nagyon lassan csökken. Ennek ellenére a Monte Carlo integrálás a legkézenfekvőbb megoldás olyan esetekben, amikor az integrálás területe összetett.[22]

A módszer egy általánosítását gyakran alkalmazzák olyan esetekben, amikor a függvényértékek számításánál egyszerűbb azt meghatározni, hogy egy adott pont az integrált képező térnek eleme-e. Ekkor a véletlenszerűen választott pontok között az integrál területén belüli és azon kívüli pontok aránya megadja az integrál arányát a vizsgált térre vetítve.

3. fejezet

Technológia

Ebben a fejezetben a dolgozatom technológiai alapjául szolgáló hardveres és szoftveres megoldásokat kívánom bemutatni. Először a grafikus processzor általános célú programozását mutatom be, majd az erre használt párhuzamos számítási platformot, a CUDA keretrendszert ismertetem. A CUDA megismerése során rálátást nyerünk a témakörrel kapcsolatos fontosabb fogalmakra, mely megalapozza az ezt követő szekciót, ahol egy nyílt párhuzamos számítási keretrendszert, az OpenCL-t mutatom be. A fejezet végén ismertetem azokat a lépéseket, melyek a CUDA platformon való fejlesztés elkezdéséhez szükségesek.

3.1. GPGPU

A GPGPU (general-purpose computing on graphics processing units) egy olyan szoftverfejlesztési gyakorlat, melynek során a grafikus feldolgozóegységet (GPU) általános célú számítási műveletek elvégzésére használjuk.

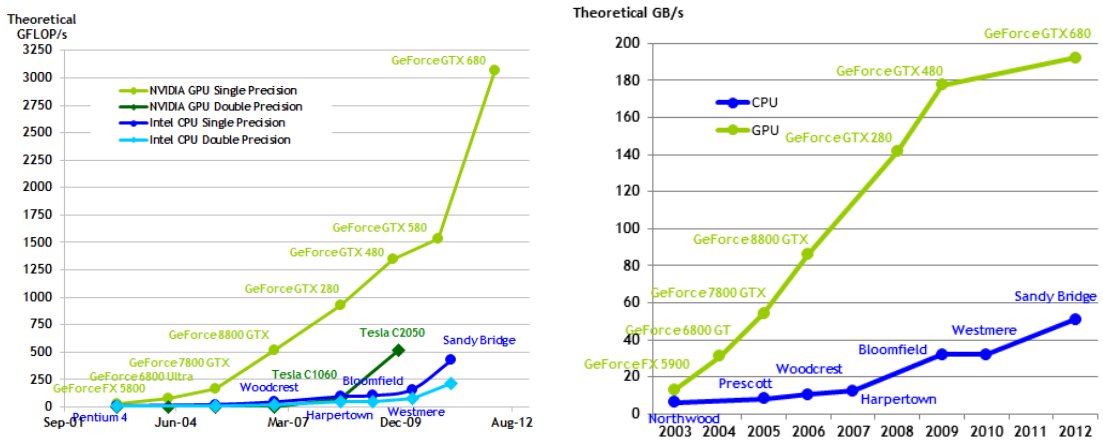
3.1.1. Motiváció

Az első szuperszámítógépek megjelenése óta a párhuzamos számítástechnika nagy fejlődésen ment keresztül és az idők folyamán új lehetőségek és kihívások jelentek meg. Míg a múlt évszázad végén a párhuzamos végrehajtásra masszívan párhuzamos processzorokat (ezek különálló processzorok, vagy különálló számítógépek) vagy transputereket (párhuzamos feladatvégzésre specializált mikroprocesszor) illetve bizonyos feladatokra dedikált vektorszámítógépeket használtak, manapság a boltok polcain megtalálható számítógépek ezeknél jóval magasabb FLOPS (floating point operations pers second) értékekkel rendelkeznek. [16]

Az órajelfrekvencia majdnem négy nagyságrenddel növekedett az első Intel 8088 processzor és a mai proceszorok között, a tranzisztorok száma pedig 29000-ről 30.5 milliárdra nőtt (AMD Threadripper PRO 3995WX). A megnövekedett órajelfrekvenciának van egy fontos mellékhatása - a processzorok hőleadása. Hogy ezen a problémán túllépjenek, a processzor gyártóknak egy új szemléletmóddal kellett előállniuk - a többmagos processzorokkal. Az Intel és az AMD egyaránt kínálnak többmagos processzorokat, melyek manapság már hétköznapinak számítanak az asztali számítógépekben. A többmagos processzorok a normál asztali rendszereket ténylegesen párhuzamos rendszerekké tették. Bár a többmagos processzorok nagyon magas számítási teljseítménnyel bírnak, az új alkalmazások egyre nagyobb és nagyobb teljesítményt követelnek meg. [16]

A fejlesztők már a 2000-es évek elején meglátták a lehetőséget a grafikus feldolgozóegységek általános célú alkalmazásokra való használatában. Ekkor a grafikus

processzorok feldolgozóegységei még kizárólag grafikai alkalmazásora voltak tervezve, így az általános célú programokat csak egy ezt kihasználó megoldással, a renderelési folyamat részeként, úgynevezett shaderek segítségével lehetett definiálni. A gyártók felismerték a generikus szoftverek futtatására való igényt, így különböző megoldásokat tettek lehetővé a hardvereik használatára, mint az NVIDIA CUDA platform, vagy az AMD APP keretrendszer. Ennek köszönhetően napjainkra a GPU programozási modellek nagy fejlődésen mentek keresztül és nagyon sok alkalmazást portoltak át ezek használatára, így néhány nagyságrendbeli gyorsulást elérve a CPU alapú változataikhoz képest. Napjainkra a GPU-k ténylegesen általános célú párhuzamos processzorok, melyek nagyon magas FLOPS rátákat képesek elérni.



3.1. ábra. Különböző CPU (Intel) és GPU (NVIDIA) termékek műveleti rátája és memória sávszélessége az évek során[14]

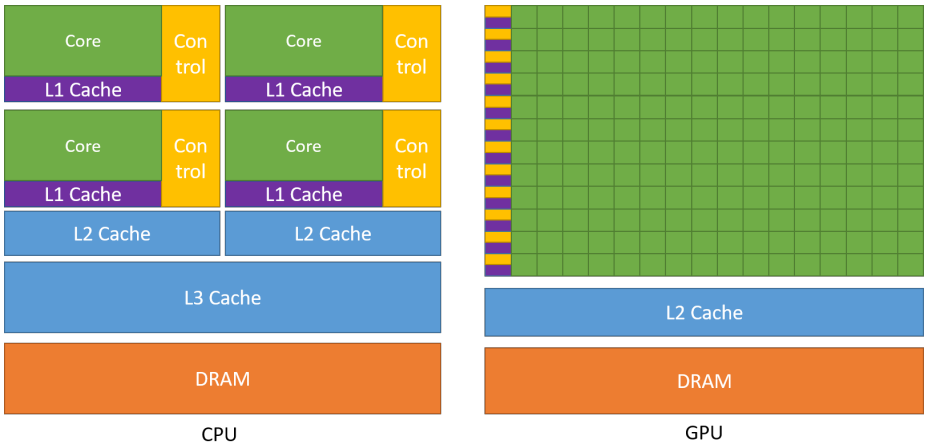
A 3.1 ábra jól mutatja a hagyományos és grafikus processzorok fejlődési ütemét és ezek egymáshoz való viszonyát. Mivel ezek az adatok egy évtized után elavultnak tekinthetők, nézzük meg kettő korszerű, asztali felhasználásra szánt eszköz specifikációit:

Típus	Termék	Kiadás ideje	Műveleti ráta (GFLOPS)	Memória sávszélesség (GB/s)
GPU	Nvidia GeForce RTX 3090 Ti [15]	2022.03.29	40000	1008.3
CPU	AMD Ryzen Threadripper PRO 3995WX [2] [1]	2020.07.14	278.3	204.8

Ezekből az értékekből látható, hogy az elmúlt tíz évben a grafikus processzorok műveleti sebessége már teraFLOP-ban mérhető illetve az elméleti memória sávszélesség is 1000GB/s felett van. A CPU-val való összevetés szempontjából továbbra is fennál, hogy ezekben a mérőszámokban napjainkban magasan túlszárnyalják a központi feldolgozóegységeket.

Az ok, amiért a grafikus processzorok még a többmagos "hagyományos" processzoroknál is nagyobb számítási teljesítménnyel rendelkeznek az, hogy ezek különböző típusú feladatok ellátására lettek tervezve, ezért felépítésükben jelentős eltérések vannak. Amíg a CPU arra lett tervezve, hogy műveletek egy szekvenciáját (szál) hajtsa végre a lehető leggyorsabban és ezekből néhány tizet képes párhuzamosan kezelni, addig a GPU célja az, hogy szálak ezreit hajtsa végre párhuzamosan (feláldozva az egyszálú végrehajtás sebességét a nagyobb áteresztőképesség elérése érdekében). Mivel a GPU nagymértékben

párhuzamos számítások (highly parallel computations) elvégzésére specializálódott, ezért ezen több tranzisztor van az adatokon végzett műveletek elvégzésére allokálva, a gyorsítótárazásra és folyamatvezérlésre használt tranzisztorok számának rovására. Ezt szemlélteti a 3.2 ábra. [5]



3.2. ábra. A CPU és a GPU tranzisztorkiosztása tükrözi a felhasználási céljukat [5].

Az adatfeldolgozásra (például lebegőpontos műveletek végzésére) szentelt erőforrások előnyösek a párhuzamos számítások szempontjából. A GPU a memóriahozzáférés késleltetéseit a nagy számítási sebességel képes elrejteni. Egy alkalmazásnak általában egyaránt vannak párhuzamos és szekvenciális részei, ezért ezek általában a grafikus és hagyományos processzorok együttesére épülnek. Az ilyen rendszereket heterogén számítási rendszereknek nevezzük. A nagymértékben párhuzamos alkalmazások kihasználhatják a GPU masszívan párhuzamos természetét, hogy ezáltal jobb teljesítményt érjenek el.[5]

A grafikus processzorok olyan problémák megoldására vannak tervezve, melyek úgynevezett adatpárhuzamos számításokként formulálhatók - amikor ugyanazokat az utasításokat hajtjuk végre párhuzamosan több adatelemen és nagyobb arányban használunk aritmetikai műveleteket, mint memóriahozzáféréseket. Ez az úgynevezett SIMD megközelítés (Single Instruction Multiple Data). Mivel ugyanazok az utasítások hajtódnak végre minden adatelemen, ezért nincs szükség összetett vezérlésre és az adatok gyorsítótárazásának hiányából adódó memóriakésleltetést az aritmetikai műveletek gyorsasága ellensúlyozza.[16]

Az adatpárhuzamos programozási paradigma számos alkalmazásban megtalálható, mint a 3D renderelés, képfeldolgozás, mintafelismerés, videó kódolás és dekódolás, lineáris algebrai eljárások, biológiai számítások, pénzügyi és gazdasági számítások. Mindezek az alkalmazások nagymértékű gyorsulást érhetnek el azáltal, hogy az adatelemeiket párhuzamos feldolgozószálakra képezik le, melyeket a GPU párhuzamosan hajt végre.

3.2. CUDA

A CUDA (Compute Unified Device Architecture) egy, az NVIDIA által fejlesztett párhuzamos számítási platform és API, amely lehetővé teszi a fejlesztők számára, hogy az ezzel kompatibilis grafikus feldolgozóegységet általános célú számításokra használják. Mindehhez egy szoftverkörnyezet is biztosítanak, amely segítségével C, vagy magasabb szintű C++ nyelven készíthetünk programokat a grafikus processzorra. A CUDA

használatával kapcsolatban egy rendkívül kiterjedt leírás érhető el a gyártó weboldalán[5], a továbbiakban a legfontosabb alapvető koncepciókat foglalom össze.

3.2.1. Skálázható programozási modell

Láthattuk, hogy a többmagos CPU-k és sokmagos GPU-k megjelenésének köszönhetően a mainstream processzor chipek párhuzamos rendszerekké váltak. A kihívás ezzel kapcsolatban az, hogy olyan szoftverket készíthessünk, melyek képesek transzparens módon skálázódni a számukra elérhető processzormagok kihasználásával. A CUDA párhuzamos számítási modell arra lett kifejlesztve, hogy megoldja ezt a problémát és emellett alacsony tanulási görbét nyújtson az olyan fejlesztők számára, akik a hagyományos programozási nyelvek területén mozognak otthonosan.

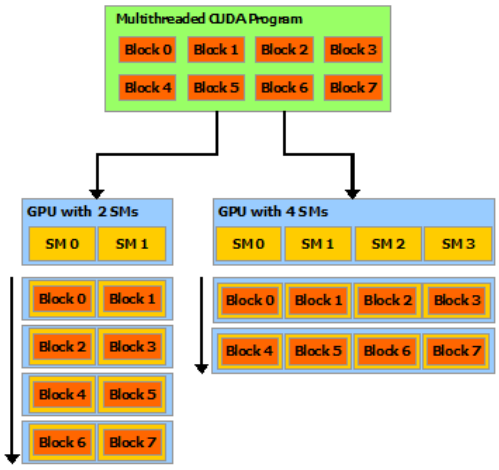
A megoldás központjában három absztrakció áll, melyek a következők:

- A szálak csoportosításának hierarchiája
- A szálak közti megosztott memória
- A szálak közti szinkronizáció (barrier synchronization)

Ezek az eszközök minimális nyelvi kiegészítésként érhetők el a fejlesztők számára. Általuk az adatok és szálak párhuzamosítása egyszerűen finomhangolható, a programozót nagymértékben segítik abban, hogy a programot olyan kisebb alproblémákra bontsák, melyek függetlenül megoldhatók párhuzamos szálak blokkjainak segítségével, és ezeket az alproblémákat tovább finomítsák oly módon, hogy azok megoldhatók legyenek a blokkokon belüli szálak kooperációjával.

Ez a fajta felbontás megtartja a programnyelv kifejezőerejét miközben a szálak együttműködését és az automatikus skálázódást egyaránt lehetővé teszi. A szálakat csoportosító blokkok a processzorban található bármely szabad feldolgozóegységen (multiprocesszoron), bármilyen sorrendben futtathatók. Ennek köszönhetően a lefordított CUDA program tetszőleges számú multiprocesszoron futtatható, ezek tényleges számát csak a futtató keretrendszernek szükséges ismernie.

Az automatikus skálázódásra mutat példát a 3.3 ábra. A többszálú CUDA programnak csak a futtatandó blokkokat kell definiálnia, ezek végrehajtása a rendelkezésre álló hardver erőforrások függvényében történik



3.3. ábra. Automatikus skálázódás

Összefoglalva, egy GPU úgynevezett Streaming Multiprocesszorok (SM) köré épül. Egy többszálú program szálak blokkjaira partícionálható, amik egymástól függetlenül képesek végrehajtódni. Ennek köszönhetően egy több multiprocesszorral rendelkező GPU automatikusan kevesebb idő alatt képes végrehajtani ugyanazt a feladatot, mint egy kevesebb multiprocesszorral rendelkező GPU.

3.2.2. Progamozási modell

A CUDA különböző intuitív nyelvi elemekkel egészíti ki a C/C++ nyelvet, melyek segítségével a párhuzamos végrehajtást konfigurálhatjuk.

3.2.2.1. Kernel

A programozó speciális függvényeket definiálhat, melyeket **kernelnek** nevezünk. Ezek a párhuzamos végrehajtás egységei. Egy kernelfüggvény meghívásakor definiáljuk, hogy azt hány példányban szeretnénk futtatni. Minden egyes kernel példányt egy számára dedikált szál hajt végre.

A kernelt a *__global__* kulcsszóval definiálhatjuk. Az, hogy a kernelt egy adott híváskor hány CUDA szálon szeretnénk futtatni, a $<<<\cdots>>>$ végrehajtási konfiguráció szintaxissal specifikálható. Minden, a kernelt végrehajtó szál egy egyedi thread azonosítót kap, mely a beépített *threadIdx* változón keresztül érhető el a kernelből.

Az alábbi kódrészlet a threadIdx változó használatával az N méretű A és B vektorokat adja össze és az eredményt a C vektorban tárolja:

```
//Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

3.2.2.2. Thread blokk

A fenti példában látható, hogy a *threadIdx* struktúra x tagját haszáltuk a szál azonosítására. Erre azért van szükség, mivel a kényelmes használat érdekében a szálak egy-, kettő-, vagy háromdimenziós logikai egységbe szervezhetők, amit **blokknak** nevezünk. Ennek megfelelően a *threadIdx* is egy több komponensből álló vektor lehet. Ez a megoldás egy természetes módot nyújt arra, hogy vektorok, mátrixok vagy testek elemein hajtsunk végre számításokat.

A szál indexe és a azonosítója között egyértelmű megfeleltetés létesíthető. Egydimenziós blokk esetén megegyeznek; kétdimenziós (D_x, D_y) méretű blokk esetén az (x, y) indexű thread azonosítója $(x + yD_x)$; háromdimenziós (D_x, D_y, D_z) méretű blokk esetén az (x, y, z) indexű thread azonosítója $(x + yD_x + zD_xDy)$

A következő kódrészlet az $N \times N$ méretű A és B mátrixok elemeit adja össze és az eredményt a C mátrixban tárolja:

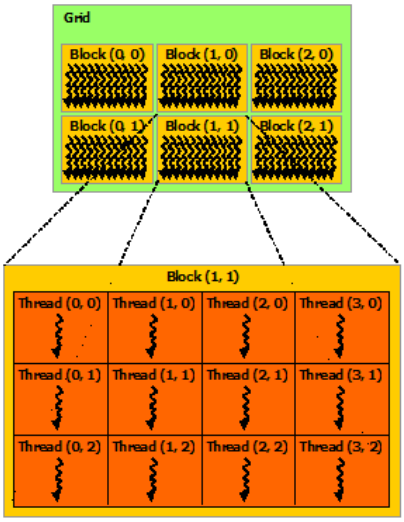
```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Az egy blokkba tartozó szálak száma limitált, mivel ezeknek közös processzormagon kell futniuk és a mag korlátos memória-erőforrásain kell osztozniuk. A jelenlegi GPU-kon egy blokk legfeljebb 1024 thread futtatására alkalmas, viszont a kernel több egyenlő méretű blokkban futtatható, ezért a szálak száma egyenlő a blokkonkénti szálak és a blokkokszám szorzatával.

3.2.2.3. Grid

A szálakat tartalmazó blokkok szintén egy-, kettő-, vagy háromdimenziós szervezeti egységben hozhatók létre, melyet **gridnek** nevezünk. A gridben található blokkok számát általában a feldolgozandó adat méretei diktálják. Ez tipikusan meghaladja a rendszerben található processzorok számát, de ez nem probléma, hiszen az automatikus skálázódásnál már láthattuk, hogy a blokkok egymástól teljesen függetlenül végrehajthatók, így csak ütemezés kérdése, hogy minden blokkot lefuttassunk a rendelkezésre álló processzormagokon.



3.4. ábra. Kétdimenziós blockok egy kétdimenziós gridben

Az egy blokkban található szálak és a gridben található blokkok száma *int*, vagy speciális *dim3* típussal specifikálható a $<<<\cdots>>>$ hívási szintaxisban.

A grid minden blokkja egy egy-, kettő-, vagy háromdimenziós indexszel azonosítható a kernel futásán belül, amely a már megimsert *threadIdx* változóval teljesen analóg

módon működő *blockIdx* változón keresztül érhető el. A blokk és a grid dimenziói szintén hozzáférhetőek a kernelből a *blockDim* és a *gridDim* változók segítségével.

Az előző *MatAdd()* példát az alábbi módon egészíthetjük ki több blokk használatára:

```
//Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    //Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

A példában egy blokk mérete 16×16 (256 thread). A grid elegendő blokkal lett létrehozva ahhoz, hogy minden mátrixelemre ugyanúgy egy szál jusson, mint eddig. Az egyszerűség kedvéért a példa feltételezi, hogy a gridben található threadek száma minden dimenzióban maradék nélkül osztható a blokkonkénti threadek számával.

Továbbra se feledjük, hogy a blokkoknak függetlenül kell végrehajtódniuk! Lehetségesnek kell lennie hogy bármilyen sorrendben, párhuzamosan, vagy akár sorosan hajtsuk végre őket, ugyanis ez a függetlenségi követelmény teszi lehetővé a magok számával való automatikus skálázódást. Példának okául oda kell figyelnünk arra, hogy két különböző blokk szálai ne írjanak ugyanarra a memóriaterületre, különben nem lehetünk biztosak abban, hogy végül melyik eredmény kerül be oda.

3.2.2.4. Blokkon belüli szinkronizáció

Az egy blokkon belül található szálak képesek egymással együttműködni. Erre egyrészt az úgynevezett *megosztott memóriaterületet* használhatják, másrészt a végrehajtásukat tudják szinkronizálni az itt végzett memóriaműveletek koordinációja érdekében.

A szálak összehangolása úgy lehetséges, hogy a kernelben szinkronizációs pontokat helyezthetünk el a *___syncthreads()* beépített függvény hívásával. A *___syncthreads()* egy "sorompóként" viselkedik, amit az adott blokk minden threadjének el kell érnie, mielőtt azok továbbhaladhatnának.

A megoldás hatékonysága érdekében a megosztott memória egy kis késleltetésű memória a processzormagok mellett, a *___synchtreads()* függvény pedig könnyűsúlyú, hívása nem jár nagy költséggel.

3.2.2.5. Memóriatípusok

A CUDA szálak több különböző memóriaterületen kezelhetnek adatokat, melyek más-más felhasználási módokkal és ennek megfelelően különböző tulajdonságokkal rendelkeznek. Ezek a memóriatípusok az alábbiak:

1. **Regiszterek**: A kernelben deklarált változók automatikusan az adott szálhoz tartozó regiszterekben jönnek létre. A regiszterek száma véges, a korszerű

architektúrákon 255 darab, egyenként 32 bites regiszter érhető el az egyes szálak számára. A regiszterek elérése nem igényel extra órajelciklust. A kernelben definiált tömbök, illetve a regiszterek túlcsordulása esetén a további változók a lokális memóriában jönnek létre.

2. **Lokális memória**: A szálak lokális memóriája valójában a globális memória egy szelete, melynek allokációját és címzését a keretrendszer transzparensen átvállalja a fejlesztőtől.

3. **Megosztott memória**: Az egy blokkon belüli szálak hozzáférnek egy közös memóriaterülethez, melyen keresztül együtt tudnak működni. A memórián keresztüli összehangolt működéshez a szálak szinkronizációjára van szükség (___syncthreads()). A megosztott memória mérete a korszerű eszközökön 164kb és a lokális memóriával szemben a blokkot végrehajtó processzormagon található. Sávszélessége a globális memória sávszélességének akár húszszorosa [9]

4. **Globális memória**: Minden thread hozzáfér az eszköz DRAM tartományán allokált területekhez. Tipikus mérete napjainkban 8-12 GB. Mint azt a 3.1 részben láthattuk, a korszerű grafikus processzorok elméleti globális memória sávszélessége az 1 TB/s-os értéket közelíti.

A felsoroltakon túl lehetőség van kettő olyan memóriaterület használatára, melyek egy szál számára "csak olvasható" módon érhetők el:

5. **Konstans memória**: A konstans memória szintén a globális memóriában van tárolva, de egy úgynevezett konstans gyorsítótár tartozik hozzá. Az itt található elemek első elérése a globális memória elérésének késleltetésével történik, ezután az elérés időigénye egységnyi. A konstans memória mérete 64kb.

6. **Textúra memória**: A textúra memória elemei háromdimenziós struktúrába rendezhetők és a processzor a térbeli lokalitásnak megfelelő cache algoritmust alkalmaz rájuk. A textúra memória különböző címzési módokat és szűrési funkciókat tesz lehetővé bizonyos specifikus adatstrukturákhoz.

3.2.2.6. Heterogén programozás

A CUDA programozási modell feltételezi, hogy a CUDA szálak egy fizikailag különálló eszközön (device) hajtódnak végre, ami a C/C++ programot futtató gazdagép (host) társprocesszoraként működik. Ez a helyzet áll fenn akkor, amikor a kernelek a GPU-n hajtódnak végre, a maradék funkcionalitás pedig a CPU-n fut.

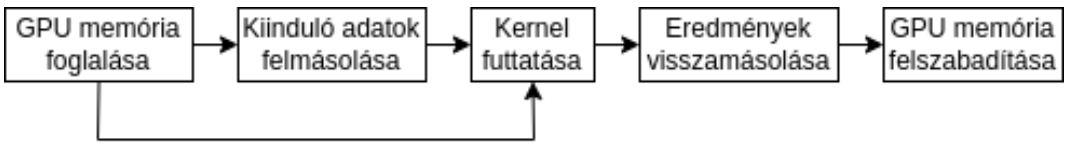
A CUDA programozási modell ezen felül feltételezi, hogy a host és a device egyaránt a saját memóriaterületüket tartják karban a DRAM-ban, amit host memóriának és device memóriának nevezünk.

Ennek a legfontosabb következménye, hogy a host program nem férhet hozzá a device memória területeihez és ez fordítva sem lehetséges. Az egyes memóriaterületek lefoglalása, felszabadítása és a memóriaterületek közti adatátvitel a CUDA runtime felé intézett speciális API hívásokon keresztül történik, melyek közül a legfontosabbakat a következő táblázatban foglaltam össze:

Memória	Művelet	Példa szintaxis (deklaráció vagy runtime API hívás)
Regiszter	Használatuk a szokásos módon történik, transzparens kezelésükről a keretrendszer gondoskodik.	
Lokális		
Megosztott	deklaráció (a kernelben)	___shared___ int s[64];
Globális	allokáció	cudaMalloc((void**)dev_ptr, 64 * sizeof(int));
	deallokáció	cudaFree(dev_ptr);
	adatok másolása (host to device)	cudaMemcpy(dev_ptr, host_ptr, 64 * sizeof(int), cudaMemcpyHostToDevice);
	adatok másolása (device to host)	cudaMemcpy(host_ptr, dev_ptr, 64 * sizeof(int), cudaMemcpyDeviceToHost);
Konstans	allokáció (a hoston)	___constant___ int c[64];
	adatok másolása (host to device)	cudaMemcpyToSymbol(c, dev_ptr, 64 * sizeof(int));

* A táblázat nem tartalmazza a textúra memóriát. Ez egy speciális témakör, melynek ismertetése túlmutat a dolgozat keretein. A megoldásom során nem használtam a textúra memória által nyújtott lehetőségeket.

A heterogén rendszer által meghatározott kényszerek a 3.5 ábrán látható magas szintű struktúrát követelik meg a párhuzamosságra felkészített rendszerektől.



3.5. ábra. Egyszerű program munkafolyamata egy heterogán rendszerben

3.2.2.7. Speciális műveletek

Atomi műveletek
Az atomi függvények egy read-modify-write atomi műveletet hajtanak végre a megosztott, vagy a globális memória adatán. Például az atomicAdd() beolvas egy értéket a megosztott, vagy globális memória adott címéről, hozzáad egy számot és az eredményt visszaírja ugyanarra a címre. A művelet abból a szempontból atomi, hogy garantáltan interferencia nélkül hajtódik végre a többi thread szempontjából. Máshogy fogalmazva, a többi thread nem férhet hozzá az adott memóriacímhez, amíg ez a művelet be nem fejeződik. Az atomi műveletek nem használhatók memória "sorompóként", mivel nem határoznak meg szinkronizációs vagy sorrendbeli kényszereket a memóriaműveletekre. Az atomi függvények csak a device kódban használhatók.

Matematikai függvények
A CUDA bizonyos standard matematikai függvényeket tesz elérhetővé, melyek a device és a host kódban egyaránt használhatók. (sqrt(x), log(x), sin(x) stb.)

Ezek alternatívájaként használhatók az úgynevezett intrinsic függvények, melyek csak a device kódból hívhatók és a standard függvények kisebb precíziójú, de gyorsabb verzióit valósítják meg. A nevük az eredeti függvénnyek prefixált neveivel egyezik meg (___sqrt(x), ___log(x), ___sin(x), stb). Ezek a függvények azért gyorsabbak, mert kevesebb natív instrukcióra képeződnek le.

3.2.2.8. Adatpárhuzamos végrehajtás

Warpok

A multiprocesszor a kerneleket 32 párhuzamos szálból álló egységekben, úgynevezett **warpokban** példányosítja, ütemezi és futtatja. Az egy warpba tartozó szálak egyszerre indulnak, ugyanazon az utasításcímen. Ettől függetlenül saját utasítás számlálójuk és regiszter állapotuk van, ezért tetszőlegesen elágazhatnak és függetlenül végrehajtódhatnak.

Amikor egy multiprocesszor megkap egy vagy több thread blokkot végrehajtásra, ezeket először warpokba partícionálja. Minden ilyen warp végrehajtása egy úgynevezett warp ütemezőn keresztül történik. Egy blokk warpokra bontása mindig ugyanolyam módon történik. Az egyes warpok egymást követő azonosítóval rendelkező szálakat tartalmaznak, ahol az első warp tartalmazza a 0 azonosítójú szálat. A 3.2.2.2 szekcióban láthattuk, hogy a thread azonosítók hogyan viszonyulnak a blokkon belüli indexhez.

A warp alapú ütemezés motivációja az, hogy egy warp egyidőben mindig egy közös utasítást hajt végre. Ebből adódik, hogy a legnagyobb hatékonyságot akkor érhetjük el, ha a warpban található 32 szálnak megegyezik a végrehajtási útvonala. Ha az egyes threadek egy adatfüggő feltételes úton divergálnak egymástól, akkor a warp minden utat lefuttat, miközben letiltja azokat a szálakat, melyek nem az éppen végrehajtott útvonalon állnak. Ez az elágazási divergencia csak egy warpon belül történik, a különböző warpok egymástól függetlenül hajtódnak végre, nem számít hogy azok megegyező vagy különböző végrehajtási útvonalat járnak be.

Egy adott szálat aktív, vagy inaktív (letiltott) szálnak nevezünk annak függvényében, hogy az részt vesz-e a warp által végrehajtott aktuális utasításban. Egy thread különböző okokból lehet inaktív, például előbb visszatérhetett, mint a warp többi szála; más útvonalon haladt tovább, mint a warp által éppen végrehajtott útvonal; vagy egy egyike egy olyan blokk utolsó threadjeinek, melyben a szálak száma nem a warp méret többszöröse (üres threadek).

Ha egy, a warp által végrehajtott nem atomi utasítás a globális vagy megosztott memória egyazon területére ír több, mint egy szálban, a tényleges írási műveletek száma a hardveres megvalósítástól függ. Az, hogy melyik thread végzi el az utolsó írást, nem definiált.

Ha egy, a warp által végrehajtott atomi utasítás a globális vagy megosztott memória egyazon területéről olvas, módosít és ír több, mint egy szálban, minden olvasási, módosítási és írási művelet sorosítva megtörténik, de a végrehajtási sorrend nem definiált.

SIMT architektúra

A warpok által megvalósított megoldást SIMT (Single Instruction Multiple Thread) architekturának nevezzük, ami hasonlít a SIMD (Single Instruction Multiple Data) szervezéshez abból a szempontból, hogy egyetlen utasítás vezérel több feldolgozó elemet. Egy kulcs különbség azonban az, hogy a SIMD utasítások a teljes rendszerre kiterjednek, a SIMT utasítások viszont egyetlen szál végrehajtási és elágazási viselkedését specifikálják. A SIMD alapú megoldásokkal ellentétben a SIMT lehetővé teszi a programozók számára, hogy thread szintű párhuzamos kódot írjanak egymástól független szálakra vagy adatpárhuzamos kódot írjanak koordinált szálakra. Az algoritmus helyességének tekintetében a programozó figyelmen kívül hagyhatja a SIMT viselkedést, mégis jelentős teljesítménybeli javulást érhet el azzal, ha kihasználja, hogy a kód ritkán követeli meg az egy warpon belüli szálak divergenciáját. Ez analóg a hagyományos kódok esetén a cache használatával: amikor a célunk a helyes működés elérése, akkor eltekinthetünk tőle, ellenben ha a célunk a csúcsteljesítmény, akkor a gyorsítótár kihasználása érdekében a lokalitástudatos programozás elengedhetetlen.

3.2.3. Programozási interfész

Az előző szekciókban láthattuk, hogy a CUDA C/C++ egy egyszerű megoldást biztosít a feladatok grafikus processzoron való végrehajtásához. Ez a megoldás a programozási nyelv kiegészítéseire és egy runtime könyvtár használatára épül.

A bemutatott nyelvi kiegészítések lehetővé teszik a programozó számára, hogy a kerneleket C++ függvényként definiálja és ennek meghívásakor néhány új szintaktikai elem használatával meghatározza az ezt futtató grid és blokk dimenzióit. Ahhoz, hogy ezek a bővített funkciók érvényre jussanak, a forrásfájlokat az nvcc fordítóval kell lefordítanunk.

Az nvcc-vel fordított kódok vegyesen tartalmazzák az úgynevezett host kódot és a device kódot. Ennek megfelelően az nvcc munkafolyamata az alábbi:

- A host és device kód szétválasztása
- A device kód lefordítása bináris formába
- A host kód módosítása, lecserélve benne a <<< ... >>> szintaxist a szükséges CUDA runtime hívásokkal, amik betöltik és elindítják a lefordított kernelt

A módosított host kód kimenete egy C++ kód, amit manuálisan lefordíthatunk egy másik eszközzel, de megengedhetjük az nvcc-nek is, hogy a fordítás utolsó szakaszában lefuttassa rá az alapértelmezett host fordítót (pl. gcc). Az alkalmazások ezt követően képesek linkelni a lefordított host kódot, vagy a CUDA driver API használatával betölthetik és futtathatják a bináris formátumú kernel kódot.

3.2.3.1. Compute Capability

Egy eszköz számítási képessége (compute capability) egy verziószámmal írható le, melyre gyakran "SM version" (Streaming Multiprocessor Version) néven is hivatkoznak. Ez a verziószám azonosítja a GPU hardver által támogatott funkciókat. A CUDA alkalmazások futásidőben hozzáférhetnek az adott eszköz compute capability-jéhez és ennek függvényében elágazhatnak a használható eszközkészlet tekintetében.

A számítási képességet jelölő verziószám X.Y formátumú, ahol az X szám a GPU alap architektúrájának feleltethető meg (Volta, Pascal, Tesla, stb), az Y szám pedig az architektúra inkrementális fejlesztéseit, és ezzel potenciálisan új elérhető funkciókat jelöl.

Az egyes számítási képességeken támogatott funkciók listája megtalálható az NVIDIA CUDA útmójának mellékletében [5].

3.3. OpenCL

Az előző fejezetben megismerhettük a CUDA párhuzamos számítási platformot, mely egy megoldást nyújt az NVIDIA által gyártott grafikus processzorok általános célú programozására. A CUDA a legnépszerűbb, de nem az egyetlen megoldás, amely a GPGPU programozást lehetővé teszi. A különböző gyártók különböző interfészeket nyújtanak a saját hardvereik használatához. A platformfüggetlenség céljából felmerült az igény egy olyan szabvány készítésére, amely független a hardveres megoldásoktól.

Az OpenCL (Open Computing Language) az első nyílt szabvány heterogén rendszerek általános célú párhuzamos programozására. A szabvány egy egyedi programozási környezetet igyekszik nyújtani a szoftverfejlesztők számára, hogy azok hordozható kódot írhassanak többmagos CPU-val, vagy GPU-val rendelkező szerverekre, laptopokra, asztali számítógépekre. [16]

Az OpenCL forráskódok kettő részre oszthatók: az egyik rész a device program, mely a GPU-t használja párhuzamos számításra, a másik pedig a hoston futó kód. A fejlesztők a speciális kernelfüggvényeket az OpenCL programozási nyelv felhasználásával definiálhatják, ami a C nyelv egy kiterjesztése. A kernelek futtatását a grafikus feldolgozóegység ütemezi. A host program C vagy C++ nyelven írható, és magát az OpenCL környezetet vezérli, a kernelek futtatását menedzseli.

Az OpenCL a SIMT (Single Instruction Multiple Thread) modellt használja a végrehajtásra. Ez azt jelenti, hogy pontosan ugyanaz az utasítás fut az egyes párhuzamos szálakon, viszont minden szál különböző adatokon dolgozik. A CUDA threadek OpenCL-beli megfelelője az úgynevezett **work-item**, ami az alapvető végrehajtási egység. A work-itemek szintén rendelkeznek egy azonosítóval, ami a kernelfüggvényekben elérhető és felhasználható az adott work-item által kezelt adatelemek azonosítására. Amikor egy kernelt elindítunk, azzal egyidőben a hívás során definiált számú work-itemek is létrejönnek. A work-itemek együttműködhetnek az úgynevezett **work-groupokon** belül, mely a work-itemek egy- kettő-, vagy háromdimenziós szervezését teszi lehetővé, tehát a CUDA blokk fogalmának feleltethető meg. A work-groupok azonosítója is elérhető a kernelben, valamint ezek is háromdimenziós struktúrába rendezhetők, amit az OpenCL esetén **ND-Range**-nek hívunk.

Látható, hogy az OpenCL programozási modellje nagy mértékben hasonlít a CUDA megvalósítására, viszont az OpenCL nem csak grafikus processzorok kezelésére alkalmazható, akár a többmagos processzorokat, vagy a többprocesszoros rendszereket is felhasználhatja a párhuzamos végrehajtás elvégzésére. Érdemes hozzátennünk, hogy az OpenCL nem egy implementáció, hanem egy interfész, amit a különböző gyártók implementálhatnak. A CUDA-nak is létezik az OpenCL szabvány megvalósítására készített "becsomagolt" megoldása. Azonban az ilyen implementációkról általában elmondhatjuk, hogy kevésbé hatékonyak, mint a gyártó által készített natív megoldások. Ebből adódóan az OpenCL használata csak azokban az esetekben célszerű, amikor ténylegesen szükségünk van az általa nyújtott, különböző rendszerek közti kompatibilitásra.

Természetesen az OpenCL-en kívül több nyílt keretrendszer is létezik, melyek különböző eltérésekkel bizonyos specifikus problémákra próbálnak megoldást nyújtani. Ilyen például az OpenMP, amely a különböző eszközök között egy közösen elérhető memóriaterület virtualizációval oldja meg a heterogén rendszereken való párhuzamosítás problémáját, vagy az OpenACC, amely főleg a tudományos és mérnöki köröket célozza meg azáltal, hogy a párhuzamosítás vezérlését magasabb absztrakciós szintű eszközökkel teszi lehetővé.

3.4. Fejlesztési környezet

Ebben a fejezetben azokat a lépéseket kívánom bemutatni, melyek a CUDA környezetben való fejlesztés elkezdéséhez és az elkészült programok futtatásához szükségesek.

3.4.1. Távoli eszköz

Mivel a saját számítógépem grafikus processzora nem CUDA kompatibilis, ezért a programjaim futtatásához egy külső szolgáltatás igénybevételére volt szükségem. Számos szolgáltató kínál olyan felhő alapú megoldást, amely segítségével CUDA alapú programokat futtathatunk egy távoli virtuális számítógépen. (Amazon AWS, Google Cloud, DigitalOcean, Oracle, IBM, stb.)

A fejlesztés során egy, a Google Cloud Platformon létrehozott szervert (Compute Engine példányt) használtam, melynek néhány fontosabb specifikációja a következő:

OS: Ubuntu 20.04.4 LTS (x64)
GPU: 1 x NVIDIA Tesla V100
DISK: 50GB HDD
CPU: 2 Cores, Intel Haswell architecture
RAM: 8GB

Többek között a költséghatékonyság miatt döntöttem emellett a szolgáltatás mellett, ugyanis a virtuális gép létrehozásakor lehetőségem volt arra, hogy működési modellként egy úgynevezett "preemtív" opciót válasszak. Ennek az a tulajdonsága, hogy a példány futását a szolgáltató az erőforrásszükséglet kielégítése érdekében bármikor felfüggesztheti. Természetesen ennek engedélyezése a szolgáltatásért fizetendő összeg csökkentését vonja magával. Amikor a szerver nincs használatban, csak a tárhelyért kell fizetni.

A preemptív modell azért volt előnyös számomra, mivel a munkám jellegéből adódóan a kódok futtatására csak bizonyos időközönként, egy-egy algoritmus elkészültét követően volt szükségem.

3.4.1.1. CUDA telepítése

Ebben a bekezdésben azt szeretném bemutatni, hogy az imént ismertetett, frissen konfigurált virtuális szerveren milyen lépések szükségesek ahhoz, hogy a CUDA programok fordításához és futtatásához szükséges feltételeket megteremtsük.

Első lépésben frissítsük a rendszerünk csomag információit!

```
ze@thesis-example:~$ sudo apt update
[...]
Building dependency tree
Reading state information... Done
All packages are up to date.
```

Ellenőrizzük, hogy a grafikus processzorunk valóban csatlakoztatva van-e a virtuális gépünkhöz!

```
ze@thesis-example:~$ lspci | grep -i nvidia
00:04.0 3D controller: NVIDIA Corporation GV100GL [Tesla V100 SXM2 16GB] (rev a1)
```

A https://developer.nvidia.com/cuda-gpus oldalon ellenőrizhetjük, hogy a kártya valóban CUDA kompatibilis. Az oldal továbbá információt ad a Compute Capability-ről is, ami esetünkben 7.0.
A következő paranccsal ellenőrizhetjük a rendszerünk hardver architektúráját és az operációs rendszer verzióját.

```
ze@thesis-example:~$ uname -m && cat /etc/*release
x86_64
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=20.04
DISTRIB_CODENAME=focal
DISTRIB_DESCRIPTION="Ubuntu 20.04.4 LTS"
[...]
```

A Cuda Toolkit-tel való fejlesztéshez szükségünk van a gcc fordítóra is. Ellenőrizzük, hogy telepítve van-e:

```
ze@thesis-example:~$ gcc --version

Command 'gcc' not found, but can be installed with:

apt install gcc
Please ask your administrator.
```

Esetünkben a gcc nincs telepítve, ezért ezt külön kell megtennünk a következő parancs segítségével:

```
sudo apt install build-essential
```

A build-essential csomag több olyan függőséget tartalmaz, melyek elengedhetetlenek a C, C++ fordításhoz (gcc, g++, libc6-dev, make, dpkg-dev). Ellenőrizzük ismét, hogy a gcc elérhető-e!

```
ze@thesis-example:~$ gcc --version
gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
[...]
```

Telepítsük a kernelhez tartozó fejléceket! Ezek olyan komponensek, melyek a Linux kernel által nyújtott API interfészeket definiálják. Ezeken keresztül képesek a driverek és egyéb alacsony szintű műveleteket végző szoftverek, mint a CUDA Toolkit programjai az operációs rendszer funkcióit használni.

```
ze@thesis-example:~$ sudo apt-get install linux-headers-$(uname -r)
[...]
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
```

Esetünkben a kernel header-ök telepítve voltak.
A következő lépés a Cuda Toolkit letöltése. Ezt az NVIDIA által nyújtott webes felületen (https://developer.nvidia.com/cuda-downloads) tehetjük meg, ahol a rendszerünk specifikációjának megadásával megkapjuk a számunkra alkalmas verzió telepítési útmutatóját. A runfile alapú installációt választottam, ami egy script letöltéséből és annak futtatásából áll.

```
ze@thesis-example:~$ wget https://developer.download.nvidia.com/compute/cuda/11.6.2/local_installers/
     cuda_11.6.2_510.47.03_linux.run
[...]
ze@thesis-example:~$ sudo sh cuda_11.6.2_510.47.03_linux.run
```

A telepítés során el kell fogadnunk az általános szerződési feltételeket (EULA), majd kiválaszthatjuk a telepítendő komponenseket (Driver, Toolkit, Samples, Demo Suite, Documentation)

```
ze@thesis-example:~$ sudo sh cuda_11.6.2_510.47.03_linux.run

===========
= Summary =
===========

Driver:   Installed
Toolkit:  Installed in /usr/local/cuda-11.6/

Please make sure that
 -   PATH includes /usr/local/cuda-11.6/bin
 -   LD_LIBRARY_PATH includes /usr/local/cuda-11.6/lib64, or, add /usr/local/cuda-11.6/lib64 to /etc/
     ld.so.conf and run ldconfig as root

To uninstall the CUDA Toolkit, run cuda-uninstaller in /usr/local/cuda-11.6/bin
To uninstall the NVIDIA Driver, run nvidia-uninstall
Logfile is /var/log/cuda-installer.log
```

Mint ahogyan azt a telepítő kimenete is mutatja, a telepítés befejeztével be kell állítanunk a PATH és LD_LIBRARY_PATH környezeti változókat. Ezeket permanenen elérhetővé tehetjük minden felhasználó részére, ha létrehozunk egy scriptet az /etc/profile.d könyvtár alatt. A /etc/profile.d/cuda-environment.sh fájl tartalma a következő:

```
export PATH=/usr/local/cuda-11.6/bin${PATH:+:${PATH}}
export LD_LIBRARY_PATH=/usr/local/cuda-11.6/lib64 ${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
```

Ez a script a következő bejelentkezéskor lefut, de a parancsokat jelenleg manuálisan is végrehajthatjuk.
A telepítést ezennel befejeztük, már csak a validáció van hátra.
Ellenőrizzük, hogy az NVIDIA System Management Interface (NVIDIA-SMI) felismerte-e a videokártyát!

```
nvidia-smi
Mon Apr 25 20:27:39 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 510.47.03    Driver Version: 510.47.03    CUDA Version: 11.6     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   36C    P0    37W / 300W |      0MiB / 16384MiB |      1%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Az nvidia-smi parancs élő infomrációt ad a rendszerhez csatolt GPU-k állapotáról, kihasználtságáról.

A CUDA kipróbálásának legegyszerűbb módja a CUDA által nyújtott példaprogramok használata. Ezek korábban a Cuda Toolkit részei voltak, jelenleg egy github repositoryban

érhetők el, amit az alábbi paranccsal klónozhatunk:

```
ze@thesis-example:~$ git clone https://github.com/NVIDIA/cuda-samples.git
```

Az alábbiakban a deviceQuery nevű példaprogramot fordítom le a make (és közvetetten az nvcc) programmal, majd futtatom azt.

```
ze@thesis-example:~$ make -C cuda-samples/Samples/1_Utilities/deviceQuery
[...]
ze@thesis-example:~$ ./cuda-samples/bin/x86_64/linux/release/deviceQuery
./cuda-samples/bin/x86_64/linux/release/deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Tesla V100-SXM2-16GB"
  CUDA Driver Version / Runtime Version          11.6 / 11.6
  CUDA Capability Major/Minor version number:    7.0
  Total amount of global memory:                 16161 MBytes (16945709056 bytes)
  (080) Multiprocessors, (064) CUDA Cores/MP:    5120 CUDA Cores
[...]
```

Megállapíthatjuk, hogy a telepítés sikeresen megtörtént.

4. fejezet

Megvalósítás

Az alábbi fejezetben a kételektron-integrálok wavelet alapú számításának optimalizálásával kapcsolatos elvégzett munkámat mutatom be. A probléma felvezetése után a CUDA alapú implementációimat ismertetem, majd ezt követően a numerikus kvadratúrákkal kapcsolatos kísérletet készítem elő. Az eredmények bemutatását a következő fejezetben teszem meg.

4.1. A probléma definiálása

A 2.3.1 részben megismerkedhettünk Dr. Szegletes Luca és Dr. Pipek János módszerével, mely lehetőséget ad a kételektron-integrálok wavelet alapú számítására. Megtudhattuk, hogy a módszer legnagyobb előnye abban rejlik, hogy eredményei újrahasználhatók. Ebből adódóan, amennyiben lehetőségünk lenne ezek előzetes meghatározására, akkor használatukkal a kvantummechanika elektronszerkezet számításait jelentősen felgyorsíthatnánk. Sajnos abba az akadályba ütközünk, hogy ezen értékek meghatározása egy rendkívül idő- és számításigényes művelet. A kitűzött célunk ennek megfelelően az, hogy a módszer futásidejét párhuzamos számítások segítségével csökkentsük, egy olyan módszert találjunk, mely a 2.22 függvény értékeit a lehető leghatékonyabb módon képes meghatározni:

$$I_{abc}^{(m)} = 2^m \int\int dr_1'' dr_2'' \frac{s(r_1'')s(r_1''+a)s(r_2'')s(r_2''+b)}{|r_1''-r_2''+c|} \tag{2.22}$$

ahol az $r_1''(m_1,n_1,o_1)$, $r_2''(m_2,n_2,o_2)$, $a(a_1,a_2,a_3)$, $b(b_1,b_2,b_3)$ és $c(c_1,c_2,c_3)$ értékek háromdimenziós vektorok. a_i és b_i, valamint m_i, n_i és o_i pozitív egészek, c_i pedig tetszőleges valós szám. Az s egy diszkrét, kompakt tartójú wavelet skálafüggvény, a tartó hossza N. A 2.3.1 fejezetben ismertetett okokból kiköthetjük, hogy $a_i, b_i < N$, valamint figyelmen kívül hagyhatjuk azokat az eseteket is, amikor a c értéke kellően nagy.

4.1.1. Előzetes eredmények

A témával kapcsolatban született egy TDK dolgozat 2013-ban[3]. Horváth Kristóf Attila dolgozatában kettő különböző megvalósítást mutatott be, a CUDA és a nyílt forráskódú BOINC (Berkeley Open Infrastucture for Network Computing) keretrendszerek használatával. A két dolgozat eredményei sajnos nem összevethetők, mivel a megoldás során különböző felbontású skálafüggvényeket használtunk. A TDK munka egyetlen eredménye, amely lehetőséget ad az összehasonlításra az, hogy az általam a későbbiekben maximális számításigényű pontként hivatkozott adatelemet 42 nap alatt sikerült kiszámítani. Később láthatjuk, hogy ezen az értéken képesek vagyunk javítani.

4.1.2. A probléma diszkretizálása

Ahhoz, hogy a 2.22 kifejezésben szereplő integrál számítására numerikus megoldást adhassunk, először diszkretizálnunk kell azt. Az alábbiakban az ehhez szükséges lépéseket foglalom össze.

A skálafüggvény mintavételezése

Láthatjuk, hogy az integrál értéke (a bemeneti paramétereket leszámítva) csak a választott s skálafüggvénytől függ. Ezt egy diszkrét függvényként kell reprezentálnunk, a következő tulajdonságokkal:

- A skálafüggvény kompakt tartójú, a tartójának hossza N. Mivel ezen a tartományon kívül a függvény értéke mindenütt 0, így a skálafüggvényt elegendő csak a tartó intervallumán mintavételezni.
- A mintavételezés frekvenciája $1/2^m$. Ez azt jelenti, hogy egy egység hosszúságú tartomány pontjainak száma $2^m + 1$, az N hosszúságú tartomány pontjainak száma pedig $S = N \cdot 2^m + 1$. A $+1$ tag az intervallum kezdő- vagy végpontjának a mintavételezéséből adódik.

A számláló egyszerű tagjai

Első lépésben a számlálóban található $s(r'')$ alakú tagokat alakítsuk át diszkrét változatra! A 2.3.1 fejezetben bemutatott folytonos r értéket helyettesítsük egy diszkrét R indexszel! Ekkor az r' és r'' analógiájára előállíthatjuk az R' és R'' értékeket és ezekre a következő megszorításokat tehetjük:

$$\begin{array}{ll} (2.17 \text{ egyenlet}) \; s[2^m R - k] & \text{miatt } R \in [\frac{k}{2^m}, \frac{S+k}{2^m}) \\ (2.19 \text{ egyenlet}) \; R' = 2^m R & \text{miatt } R' \in [k, S+k) \\ (2.21 \text{ egyenlet}) \; R'' = R' - k & \text{miatt } R'' \in [0, S) \end{array}$$

Ennek megfelelően az $s(r'') \Rightarrow s[R''], R'' \in [0, S)$ átalakítást alkalmazhatjuk.

A számlálóban található eltolások

A számlálóban található $s(r'' + a)$ formájú kifejezések átalakításánál az előző pontban bemutatott R'' értéket vehetjük alapul, azonban figyelembe kell vennünk, hogy a folytonos függvényben való a hosszúságú eltolás a diszkrét indexben $a \cdot 2^m$ elemmel való eltolásként jelentkezik.

Az $s(r'' + a)$ kifejezést így az $s[R'' + a \cdot 2^m]$ taggal helyettesíthetjük, és így adott a érték mellett az R''-re egy új megszorítást kapunk, miszerint $R'' \in [0, S - a \cdot 2^m)$.

A nevezőben szereplő távolság

A nevezőben található $r_1'' - r_2''$ tag esetén az előző pontban alkalmazott logikához hasonlóan kell eljárnunk, viszont itt a diszkrét indexekből kell visszavezetnünk az indexek által reprezentált helyek közti távolságot.

Ezt az $r_1'' - r_2'' \Rightarrow (R_1'' - R_2'') \cdot \frac{1}{2^m}$ helyettesítéssel tehetjük meg.

Kvadratúra alkalmazása

A fenti megfontolások alkalmazásával gondoskodtunk a függvény mintavételezéséről. Ezt már csak azzal szükséges kiegészíteni, hogy valamely megismert numerikus integrálási approximációt alkalmazzuk. Ezek közül a legegyszerűbb a 2.29 egyenletben leírt jobboldali Riemann integrál.

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n} hf(x_i) \tag{2.29}$$

A kvadratúrát az R_1'' és R_2'' vektorok kifejtéséből adódó hatdimenziós esetre alkalmazva a teendőnk annyi, hogy az összegzett eredményeket a mintavételezett szakaszok $h = \frac{1}{2^m}$ hosszaival, tehát összességében $h^6 = \left(\frac{1}{2^m}\right)^6$ -nal felszorozzuk.

Numerikus alak

A felsorolt átalakításokat alkalmazva és a háromdimenziós vektorokat kifejtve eljutunk az integrál végleges, numerikus formába alakított verziójához:

$$I_{abc}^{(m)} = 2^m \sum_{m_1=1}^{m_1Max} \sum_{n_1=1}^{n_1Max} \sum_{o_1=1}^{o_1Max} \sum_{m_2=1}^{m_2Max} \sum_{n_2=1}^{n_2Max} \sum_{o_2=1}^{o_2Max} \left[\frac{s(m_1)s(m_1+a_1\cdot 2^m)s(n_1)s(n_1+a_2\cdot 2^m)s(o_1)s(o_1+a_3\cdot 2^m)}{\sqrt{(\frac{m_1-m_2}{2^m}+c_1)^2+(\frac{n_1-n_2}{2^m}+c_2)^2+(\frac{o_1-o_2}{2^m}+c_3)^2}} \right] \cdot \left(\frac{1}{2^m}\right)^6 \tag{4.1}$$

ahol

$$a_i, b_i, m_i, n_i, o_i \in \mathbb{Z}^{0+} \qquad a_i, b_i < N \qquad c_i \in \mathbb{R}$$

valamint

$$\begin{aligned} m_1Max &= S - a_1\cdot 2^m \\ n_1Max &= S - a_2\cdot 2^m \\ o_1Max &= S - a_3\cdot 2^m \\ m_2Max &= S - b_1\cdot 2^m \\ n_2Max &= S - b_2\cdot 2^m \\ o_2Max &= S - b_3\cdot 2^m \end{aligned}$$

4.1.3. A probléma mérete számokban

A 4.1 egy általános formula a wavelet alapú kételektron-integrálok numerikus számításához. A további munkához, valamint az előzetes elemzéséhez egy konkrét skálafüggvény behelyettesítése szükséges.

Az implementáció során a Daubechies-6 wavelet skálafüggvényét használtam, mely az alábbi tulajdonságokkal rendelkezik:

- A tartó mérete $N = 5$, ezért $a_i, b_i \in \{0, 1, 2, 3, 4\}$
- A mintavételezési frekvencia 2^6, tehát $m = 6$. Két pont közti távolság $dr = \frac{1}{2^m} = \frac{1}{2^6} = 0.015625$.
- A mintavételezett pontok száma $S = N \cdot 2^m + 1 = 5 \cdot 2^6 + 1 = 321$

Ezen paraméterek kikötése után már különböző megállapításokat tehetünk az integrál számításigényére.

Számításigény

Az integrál belsejében szereplő kifejezésre a továbbiakban **elemi művelet** vagy **hányados** néven fogok hivatkozni. **Számláló** és **nevező** alatt általában ennek a kifejezésnek a számlálóját és nevezőjét értem.

Megfigyelhetjük, hogy az integrál értékének meghatározásához szükséges elvégzendő elemi műveletek száma nem állandó, mivel az m_iMax, n_iMax, o_iMax értékek a bemeneti a_i és b_i változók függvényei. A c érték nem befolyásolja a számítás műveletigényét.

- A legkisebb számításigényű értékek azok, melyekre $a_i = b_i = 4$. Az ilyen pontok számításához szükséges elemi műveletek száma $(320 - 4 \cdot 2^6)^6 \approx 6.87 \cdot 10^{10}$. Ezekre a függvényértékekre a továbbiakban **minimális számításigényű pontként** hivatkozok.
- Hasonlóan **maximális számításigényű pontnak** nevezem azokat az eseteket, amikor $a_i = b_i = 0$. Ezek megállapításához $1.07 \cdot 10^{15}$ darab elemi műveletet kell kiszámítanunk.
- Egy adott c_i érték mellett a függvényértéket az a_i és b_i minden lehetséges értékére megállapítva összesen $7.83 \cdot 10^{17}$ hányados elvégzését kell megtennünk
- Az előző pont alapján egy **átlagos számításigényű pont** meghatározásához $7.83 * 10^{17}/5^6 = 5.01 \cdot 10^{13}$. Ez pontosan megegyezik az $a_i = b_i = 2$ esettel.

Az fenti pontokban meghatározott műveletek időigényére az egyes implementációk esetén rendre a $t_{\mathbf{min}}$, $t_{\mathbf{max}}$, $t_{\mathbf{all}}$ és $t_{\mathbf{avg}}$ jelöléseket alkalmazom.

4.2. Kételektron-integrálok számítása a CUDA platformon

Ebben az alfejezetben a kételektron-integrálok numerikus számítására készített különböző implementációimat ismertetem. A megoldásokat inkrementális módon mutatom be, kitérve a különböző verziók közötti változtatások motivációira, valamint az egyes módszerek futásidejének becslésére. A megoldások pontosságát a 5. fejezetben ismertetem.

Az összehasonlíthatóság érdekében minden algoritmust a Google Cloud Platformon létrehozott szerveren futtattam, melynek fontosabb host és device specifikációit a 4.1 és 4.2 táblázatok foglalják össze.

CPU	
Modell	Intel® Xeon® Processor E3-1268L v3
Órajelfrekvencia	2300 MHz
Processzormagok száma	4
Memória	
Modell	ismeretlen
Méret	15360 MB

4.1. táblázat. Host hardverspecifikáció

GPU	
Modell	Tesla V100-SXM2-16GB
Órajelfrekvencia	1530 MHz
Compute Capability	7.0
Multiprocesszorok száma	80
CUDA magok száma multiprocesszoronként	64
Globális memória mérete	16161 MB
Konstans memória mérete	64 KB
Blockonkénti megosztott memória mérete	48 KB
Blockonkénti regiszterek száma	65536
Warpok mérete	32
Blockonkénti threadek maximális száma	1024
Block maximális dimenziói (x, y, z)	(1024, 1024, 64)
Grid maximális dimenziói (x, y, z)	(2147483647, 65535, 65535)

4.2. táblázat. Device hardverspecifikáció

4.2.1. CPU alapú implementáció

Az első megoldásunk a probléma egyszerű megvalósítása párhuzamosítás nélkül, CPU-n. Érdemes megismerkednünk a megvalósítás pszeudokódjával, mivel ez nagymértékben elősegíti a megoldandó feladat megértését.

```
bemenet: a1, a2, a3, b1, b2, b3, c1, c2, c3
kimenet: a kételektron-integrál adott helyen felvett értéke

s = read_scaling_function()
S = 321
dr = 1/64
dr6 = dr^6

sum = 0
for (m1 = 1; m1 < S - a1 * 64; m1++) {
  for (n1 = 1; n1 < S - a2 * 64; n1++) {
    for (o1 = 1; o1 < S - a3 * 64; o1++) {
      for (m2 = 1; m2 < S - b1 * 64; m2++) {
        for (n2 = 1; n2 < S - b2 * 64; n2++) {
          for (o2 = 1; o2 < S - b3 * 64; o2++) {
            nevező = sqrt(
              pow((m1 - m2) * dr + c1, 2) +
              pow((n1 - n2) * dr + c2, 2) +
              pow((o1 - o2) * dr + c3, 2)
            )
            if (nevező != 0) {
              számláló = s[m1] * s[m1 + a1 * 64] * s[n1] * s[n1 + a2 * 64] * s[o1] * s[o1 + a3 * 64]
                       * s[m2] * s[m2 + b1 * 64] * s[n2] * s[n2 + b2 * 64] * s[o2] * s[o2 + b3 * 64]
              sum += számláló / nevező
            }
          }
        }
      }
    }
  }
}

result = sum * dr6
```

A megoldás első részében beolvassuk a skálafüggvényt és inicializáljuk a számításhoz szükséges változókat. Ezt követően hat egymásba ágyazott ciklus segítségével végezzük

el az összegzést. A legbelső ciklusban megállapítjuk a nevező értékét és amennyiben az nem nulla - tehát az osztás értelmezhető -, kiszámítjuk a számlálót is és a hányadosukat hozzáadjuk az eddigi eredményhez, amit a *sum* változóban gyűjtünk. A kvadratúrához tartozó szorzást kiemelhetjük, azt elegendő a folyamat befejeztével elvégezni.
A tényleges implementáció kis mértékben eltér az itt feltüntetett algoritmustól, mivel a számlálóhoz és a nevezőhöz tartozó részeredményeket már azokban a külső ciklusokban előállítja, melyekben az ehhez szükséges változók rendelkezésre állnak.

Időigény

A minimális számításigényű pontok esetén az átlagos kiértékelési idő 4199.94 másodperc volt. Feltételezhetjük, hogy az egyes pontok számítási ideje a kiszámítandó elemi műveletek számával egyenesen arányos. Ennek megfelelően a 4.1.3 részben ismertetett speciális értékekre az alábbi eredményeket kapjuk:

t_{min}	t_{max}	t_{avg}	t_{all}
4199.94 s	$6.56 \cdot 10^7$ s (759.54 nap)	$3.06 \cdot 10^6$ s (35.44 nap)	$4.78 \cdot 10^{10}$ s (1516.00 év)

Az eredményekből jól látható, hogy a probléma méretéből adódóan a hagyományos, szekvenciális algoritmusok használata nem célravezető.

4.2.2. Naiv GPU implementáció

Az első GPU alapú megoldás alapötlete az, hogy az előző megoldásban látott iteráció belsejében található számításokat egy-egy külön szálon végezzük el. Ezt egyelőre egy egyszerű konfigurációban szeretnénk megtenni, ezért egydimenziós gridet és egydimenziós blokkokat használunk.

Az egy blokkban lévő szálak számának a maximális 1024-et választjuk. Minden szál egy elemi művelet elvégzését és az eredmény globális memóriában való tárolását végzi. Az összes lehetséges elemi műveletek, és így a szükséges szálak száma $320^6 \approx 1.07 \cdot 10^{15}$. Ekkora mennyiségű szálat kettő korlát miatt nem vagyunk képesek egy darab kernelhívásban futtatni:

- A művelet elvégzéséhez szükséges blokkok száma $\frac{320^6}{1024} \approx 1.04 \cdot 10^{12}$. A grid x dimenziójában található blokkok száma viszont legfeljebb csak $2^{31} - 1 \approx 2.15 \cdot 10^9$ lehet. Ebből következően 500.001 kernel futtatásával lennénk képesek a teljes művelet elvégzésére.
- A másik akadály a globális memória mérete. Mivel minden szál eltárol egy 8 bájt méretű double értéket, az összes eredmény eltárolása $320^6 * 8$ bájtot, vagyis 8 millió gigabájtot venne igénybe.

A fenti problémák megoldására rögzítsük le az egy kernelfuttatás során használt globális memória méretét 8GB-ban. Ekkora területen 1073741824 szál eredménye fér el, ami 1048576 blokk használatával indítható. Látható, hogy ezzel mind a kettő felmerülő problémát megoldottuk. Az egy pont kiszámításához végrehajtandó kernelhívások száma 1024000000.

Megvalósítás

A kernel indítása előtt lefoglaljuk a megfelelő memóriaterületeket a host és device memóriában. Az s skálafüggvényt a globális memóriaterületen helyezzük el, hogy a kernelből elérhessük ezeket értékeket.

Ezt követően elindítunk egy ciklust, amely a kernel több lépésben való futtatásáért felel. A globális memóriában előállított eredményeket minden kernelhívás után átmásoljuk a host memóriába és ott összegezzük azokat.

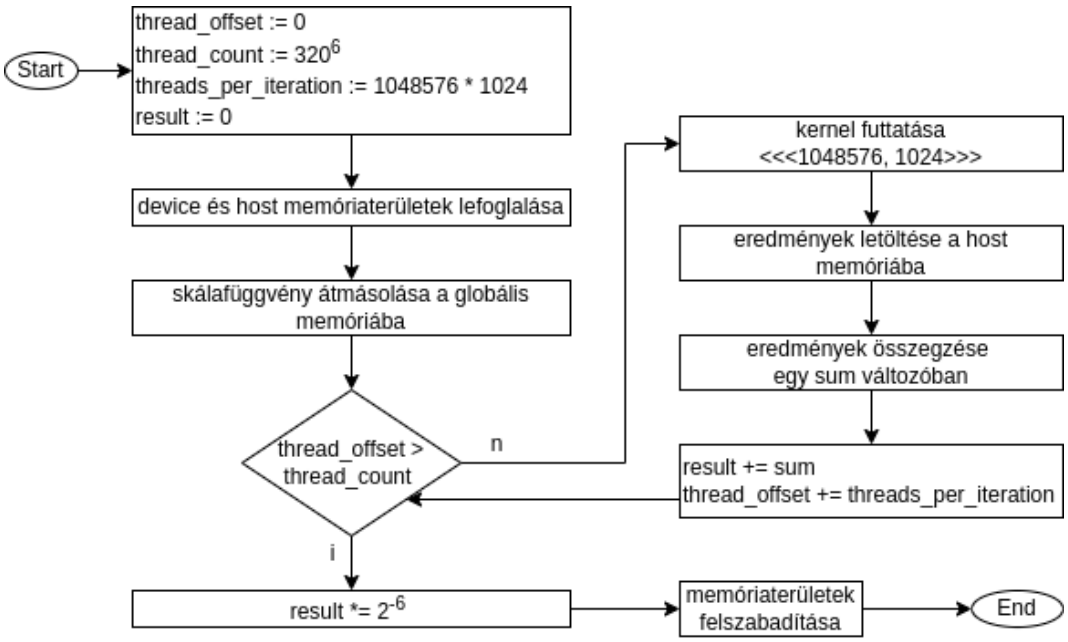
A végrehajtás során nyilvántartjuk a már lefuttatott szálak számát egy *thread_offset* változóban, amely kettős célt szolgál. Egyrészt a folyamat előrehaladását regisztrálja, másrészt egy adott szál által elvégzendő elemi művelet azonosítását teszi lehetővé. A kernel működését az alábbi pontokban foglalhatjuk össze:

- A kernel paraméterként megkapja a *thread_offset* értéket.
- Minden szál megállapítja a saját globális azonosítóját a *thread_offset*, valamint a kernelbenhívásban elfoglalt helye alapján:

$$global\_id = thread\_offset + blockIdx.x * blockDim.x + threadIdx.x$$

- A globális azonosító ismeretében a modulo és osztás operátorok használatával kiszámíthatók az m_i, n_i és o_i értékek.
- Ezt követően a szál megvizsgálja, hogy az általa képviselt elemi művelet elvégzése szükséges-e az adott integrál kiszámításához. Ha igen, akkor elvégzi a számítást és az eredményt eltárolja a megfelelő memóriaterületen. Ha nem, akkor a 0 értéket írja a memóriába, ezzel felülírva az esetleg ott található memóriaszemetet.

A megoldás magas szintű működését a 4.1 folyamatábra mutatja be.



4.1. ábra. A naiv GPU megoldás működése

Időigény

A megoldásnak több olyan jellemzője van, ami alapján magától értetődő a hatékonytalansága:

- Minden integrálérték számítása esetén a maximális számú szálat indítjuk, ezért sok esetben rengeteg olyan szál van, ami feleslegesen fut.
- Nagyszámú kernelindítást végez, ami egy viszonylag költséges művelet.
- Minden iterációban átmozgat 8GB memóriát a device és host között, majd az eredményeket lineárisan összegzi.

Ezeket az állításokat tükrözi az időigény is, ami olyan magas, hogy csak az iterációk sebességéből tudunk rá becslést adni. Egy maximális számításigényű pont 1000 ciklusának lefutása átlagosan 5229.12 másodpercbe, így a teljes végrehajtás megközelítőleg $5.35 \cdot 10^9$ másodpercbe telik.

Az egyéb pontok időigényének megállapítására nem alkalmazhatjuk az egyszerű, műveletarányos közelítést, mivel a megoldásban nem az elemi műveletek elvégzése jelenti a fő költséget. Azonban a ciklusidők méréséből sem célszerű következtetni, mivel a maximálisnál kisebb számításigényű pontok esetén egy bizonyos iterációszám után csak olyan kernelek indulnak, melyekben egy szál sem végez érdemi számítást, tehát ezekben az esetekben az iterációk hossza nem egyenletes. Ez a megállapítás viszont lehetőséget ad egy nagyvonalú approximációra, melyet a következő megfontolások alapján tehetünk meg:

- A *thread_offset* manuális eltolásával megállapítható, hogy a számítást nem végző iterációkból 1000 darab lefutása 4743.13 másodpercbe telik. Ezt az értéket vehetjük egy konstans költségnek, amely minden pont számításánál jelen van.
- A maximális számításigényű pont ciklusait ettől az különbözteti meg, hogy ott minden szál elemi műveletet végez (kivéve azokat az eseteket, amikor a nevező 0, de ettől eltekinthetünk).
- Ebből az következik, hogy a kettő érték közti különbség kizárólag a műveletvégzésből adódik, ami már arányosítható a műveletek számával. Ezt az arányosított értéket a konstans taghoz adva megkaphatjuk a különböző pontokra adott becslést, melyet az alábbi táblázat foglal össze.

t_{min}	t_{max}	t_{avg}	t_{all}
$4.86 \cdot 10^9$	$5.35 \cdot 10^9$ s	$4.88 \cdot 10^9$ s	$7.63 \cdot 10^{13}$ s
(153.91 év)	(169.68 év)	(154.65 év)	(2.41 millió év)

Látható, hogy az értékek jelentősen romlottak a CPU alapú megoldáshoz képest. Az eredményekből azt a tanulságot vonhatjuk le, hogy az optimalizáció nem bízható egyszerűen arra, hogy külön szálon futtatjuk a számításokat. Ahhoz, hogy jó eredményeket érjünk el, a megoldást alaposan át kell gondolni.

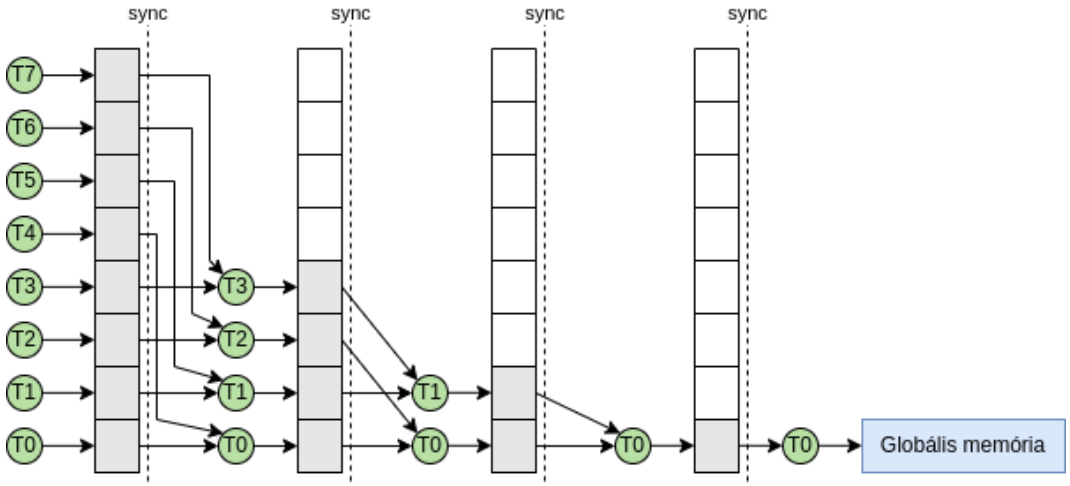
4.2.3. Device összeadás

Az előző megoldás egyik legköltségesebb aspektusa az volt, hogy minden iterációban egy 8GB méretű memóriablokkot mozgattunk át a device memóriából a host memóriába és ott

összegeztük azt. Ennek elkerülése érdekében érdemes a lehető legtöbb műveletet a GPU memóriájában elvégezni mielőtt visszatérnénk az eredménnyel.

A globális memóriában előállított adatok párhuzamos összeadására (redukciójára) egy nagyon hatékony megoldás adható, ami az egy blokkon belül található szálak együttműködésére épít a megosztott memórián keresztül. A megoldás működését az alábbiakban részletezem.

Párhuzamos redukció



4.2. ábra. Párhuzamos redukció működése

A párhuzamos redukció megvalósítására definiálhatunk egy kernelt, melyben a szálak a redukálandó adathalmaz egy-egy elemének feleltethetők meg. A művelet kezdetén minden szál elhelyezi a hozzá tartozó értéket a megosztott memória hozzá tartozó területére. Ezt követően egy iteratív folyamatba kezdünk, melynek első lépésében az aktív szálak felét elhagyjuk, majd a maradék szál segítségével egy-egy számpár összeadását végezzük el oly módon, hogy minden szál a saját értékét és egy "elhagyott" szálhoz tartozó értéket összegez, az így kapott eredményt pedig visszaírja a saját megosztott memóriaterületére. Az iterációk végén szinkronizációt végzünk, ezzel biztosítva azt, hogy minden számítás eredménye rendelkezésre álljon a következő lépéshez. Az eljárás akkor szakad meg, amikor már csak egy szál marad, ami végül elhelyezi az eredményt a globális memóriában.

Az ilyen módon megvalósított párhuzamos összegzéssel kapcsolatban a következő megállapításokat tehetjük:

- A redukció N darab számot $log_2(N)$ lépésben összegez.
- A kooperáló szálak a teljes blokkra nézve egyetlen összeget állítanak elő, ezért a kimenetek száma a blokkok számával egyenlő.
- Az algoritmus működéséhez szükséges, hogy a blokkon belüli szálak száma 2 hatványa legyen.

Megvalósítás

Létrehoztam az imént ismertetett kernelt. A redukciót végző blokkon belüli szálak számának ismét a maximális 1024-et választottam. Az előző implementációban szereplő

másolás elé kettő redukciós műveletet illesztettem be. Az első redukció az elemi műveletek számítását végző kernel 1073741824 elemű kimenetét 1048576-ra, a második pedig ezt 1024-re csökkenti. További GPU alapú összegzést nem végeztem, hiszen a kernel indítása is költségekkel jár. Ennek megfelelően a megmaradt adathalmazt (8KB) az eddigi megoldáshoz hasonlóan a host memóriába letöltve összegeztem.

Időigény

Bár a számítás elvégzéséhez szükséges idő továbbra is rendkívül magas, látható, hogy az előző megoldáshoz képest jelentős javulást értünk el. A becslést ismét a már megismert módon végeztem el.

$t_{\min}$	$t_{\max}$	t_{avg}	t_{all}
$4.17 \cdot 10^7$	$1.32 \cdot 10^8$ s	$4.59 \cdot 10^7$ s	$7.17 \cdot 10^{11}$ s
(482.61 nap)	(1525.80 nap)	(531.21 nap)	(22725.36 év)

Érdemes megfigyelni, hogy míg az előző implementáció esetén az egyes végrehajtási idők között nem volt számottevő eltérés, addig az új eredmények a különböző pontok esetén jóval nagyobb kontrasztot mutatnak. Ez a jelenség is arra utal, hogy a memóriatranszfer csökkentésével az elemi műveletek száma sokkal hangsúlyosabb tényezővé vált teljes költségben.

4.2.4. Blokkok számának növelése

A párhuzamos redukció bevezetésével megismerhettünk egy olyan mechanizmust, ami lehetővé teszi, hogy a megosztott memória használatával a kapott eredményeket blokk szinten összegezzük.

Ebben a változatban, azt a módosítást végeztem el, hogy az első szintű redukciót már az elemi műveletek számítását végző kernelben elvégeztem, így az eredmények tárolásához szükséges memória jelentősen lecsökken. Ezt a tényt a másik irányból megközelítve megállapíthatjuk, hogy ugyanakkora memóriaterületen jóval több számítás eredményét vagyunk képesek eltárolni. Ha továbbra is a 8GB-os memóriaterületnél maradunk, akkor a blokkok számát az eddigi 1048576-ról 1073741824-re növelhetjük és ennek köszönhetően az iterációk és az ezzel járó kernelhívások száma 1024000000-ról pontosan egymillióra csökken. A blokkszám növelése miatt nem kell hátrányos változásra készülnünk, hiszen a multiprocesszorok telítődése után a végrehajtási idő a blokkok számával egyenesen arányos.

Időigény

Szintén az eddigi becslést alkalmazva az alábbi eredményeket kapjuk:

$t_{\min}$	$t_{\max}$	t_{avg}	t_{all}
$2.54 \cdot 10^7$	$1.37 \cdot 10^8$ s	$3.06 \cdot 10^7$ s	$4.78 \cdot 10^{11}$ s
(294.34 nap)	(1582.30 nap)	(354.36 nap)	(15159.32 év)

Látható, hogy a kisebb számításigényű pontok esetén jóval nagyobb javulást értünk el, hiszen ezek esetén az egy számítási műveletre jutó kernelindítások száma nagyobb volt, így annak jobban érvényesült a hatása.

Bár az első GPU alapú implementációhoz képest az átlagos pont számításának tekintetében százötvenszeres gyorsulást értünk el, a jelenlegi megoldás továbbra is hatékonytalan, a CPU alapú megoldásnál is lassabb. Belátható, hogy a naiv megoldás iterációs alapgondolata problémás, az optimálisabb végrehajtás érdekében a kernel futtatására egy jobb megoldást kell kell keresnünk.

4.2.5. Kernel dimenziók finomhangolása

Az eddigi megvalósításokban az elemi műveletek számítása kapcsán két komolyabb problémát állapíthatunk meg. Egyrészt nagyon nagy számú felesleges szálat indítottunk, másrészt az m_i, n_i és o_i "ciklusváltozók" meghatározására alkalmazott osztáson és maradékképzésen alapuló módszer költséges volt. Ebben a részben megmutatom, hogy a kernel futtatási paramétereinek megfelelő megválasztásával mindkettő problémát megoldhatjuk.

Mint azt megismertük, egy blokk szálait háromdimenziós strukturába rendezhetjük, valamint az így kialakított blokkokat is lehetőségünk van egy háromdimenziós gridben létrehozni. Ez alapján kézenfekvő lehet az a megoldás, hogy a kernelt egy ($m_1Max - 1$, $n_1Max - 1$, $o_1Max - 1$) méretű gridben és ($m_2Max - 1$, $n_2Max - 1$, $o_2Max - 1$) méretű blokkokban indítsuk. (*Megjegyzés: a -1 tagok abból adódnak, hogy a Riemann kvadratúra miatt a nulladik elemet nem számítjuk bele az integrálba*). Ebben az esetben bármely szál egyértelműen azonosíthatja az általa kezelt m_i, n_i, o_i ciklusváltozókat a blockIdx és threadIdx strukturákon keresztül.

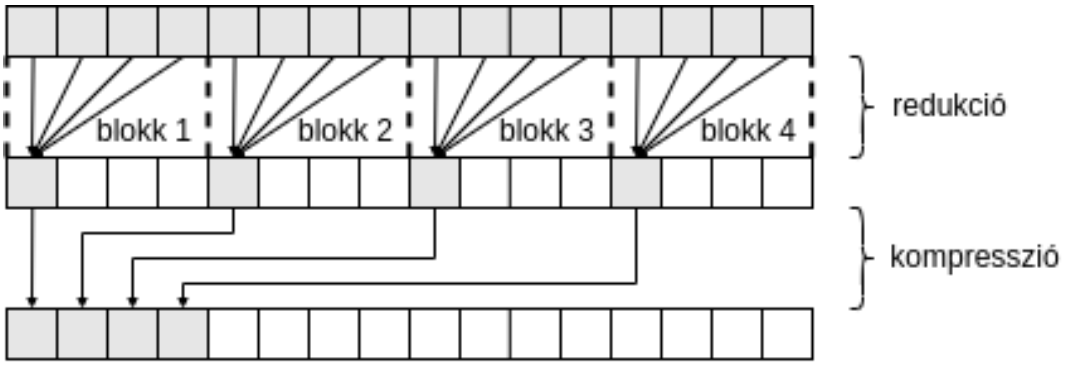
Ez a megoldás azonban így nem megvalósítható, mivel az $(m_2Max - 1) * (n_2Max - 1) * (o_2Max - 1)$ szorzat értéke, tehát a blokkon belüli szálak száma már a minimális számításigényű pontok esetén is 262144, ami meghaladja a maximális 1024-et. Ezt a problémát első közelítésben megoldhatjuk úgy, hogy az m_1 és n_1 értékek alapján a host kódban iterálunk, a kernelt pedig ezen belül ($o_1Max - 1$, $m_2Max - 1$, $n_2Max - 1$) méretű gridben, egydimenziós $o_2Max - 1$ méretű blokkokban futtatjuk. A kernel paraméterként megkapja az aktuális m_1 és n_1 értéket, így ezeket a többi változóhoz hasonlóan közvetlenül elérheti.

Ahhoz, hogy az első redukciót továbbra is el tudjuk végezni a kernelben, arra van szükségünk, hogy a blokkban lévő szálak száma kettő hatványa legyen. Ez azonban nem minden esetben igaz az $o_2Max - 1$ méretre. Ezért ehelyett a blokkméret helyett azt a legkisebb kettő hatványt választjuk, amely minden esetben nagyobb az $o_2Max - 1$ értéknél. Ez a szám az 512. Az extra szálak nem végeznek műveletet, a redukció előtt csak egy nullát írnak a megosztott memóriába.

Egy szerencsés véletlen, hogy az $o_2Max - 1$ érték minden esetben a 32 többszöröse (belátható, hogy ez minden $m > 4$ skálafüggvény felbontás esetén teljesül), valamint az 512 is a 32 többszöröse. Ez azt jelenti, hogy a blokkokban lévő "hasznos" és "felesleges" szálak sosem keveredtek a warpokban, így a plusz szálak nem okoznak divergenciát a kernel végrehajtásában, így várhatóan a teljesítményre gyakorolt hatásuk sem jelentős.

Egy kernelfuttatás eredményei legfeljebb $320^3 \cdot 8$ bájtot, azaz 250MB-ot foglalnak a globális memóriában, ami az eddigi megoldáshoz képest szintén jelentős javulást jelent, azonban egy fontos változás az eddigi megvalósításhoz képest, hogy a blokkok száma, így az előállított adatok mennyisége is változó. Ennek következtében az elemi műveletet végző kernel utáni összegzésre egy ehhez igazodó megoldást kell adnunk, melyet én egy rekurzív összegző függvény bevezetésével tettem meg. Ez egy bizonyos elemszámig ismételt GPU alapú redukcióval csökkenti az eredmény méretét a globális memóriában, majd a hostra átmásolva összegzi azt. Ahhoz, hogy elkerüljem a ciklikus memóriafoglalást és felszabadítást, az új összegző algoritmust úgy valósítottam meg, hogy a redukciók során elkészített részösszegeket a forrás memóriaterületre írja vissza. Ez azonban azt

eredményezte, hogy a memóriaterületen az eredmények nem egymást követő indexeken álltak elő, ezért készítettem egy további kernelt, ami az így előállított eredmények összetömörítését végzi. A 4.3 ábra egy egymást követő redukciós és tömörítési műveletpárt mutat be.



4.3. ábra. A dinamikus összegzés két lépése

A megoldásban az említetteken felül egy további említésre érdemes módosítást végeztem. Az s függvényt az eddigi globális memória helyett a konstans memóriában helyeztem el, így várhatóan a skálafüggvény értékeinek elérése a gyorsítótárazásnak köszönhetően jóval optimálisabb lesz.

Időigény

Ezekkel a módosításokkal sikerült elkészítenünk az első olyan verziót, amely jóval túlszárnyalja a CPU alapú implementációt. A minimális ás átlagos pont számítási idejét képes voltam kísérletileg meghatározni, a maximális számításigényű pont esetén pedig egy nagyon jó minőségű becslés adható. Továbbra is iterációkban dolgozunk és jelen esetben minden iteráción belül azonos mennyiségű műveletet hajtunk végre. Ennek köszönhetően az iterációk sebességéből következtehetünk a teljes végrehajtás sebességére.

Ezek alapján az alábbi értékek adódtak az egyes időigényekre:

$t_{\min}$	$t_{\max}$	t_{avg}	t_{all}
18.18 s	$1.05 \cdot 10^5$ s (29.09 óra)	5820.01 s (1.62 óra)	$9.10 \cdot 10^7$ s (1052.52 nap)

4.2.6. Iteráció a kernelben

Az előző megoldás során láthattuk, hogy a kernel futtatási paramétereiben nem voltunk képesek átadni az egyes iterációs változók értékeit. Ezt úgy küszöböltük ki, hogy kettő változó szerint a host kódban iteráltunk és a kernelt több példányban futtattuk.

Ezzel kapcsolatban azonban egy másik megoldás is felmerülhet, melynek során, egy integrál meghatározását egyetlen kernelhívással végezzük el és a ciklust a kernel belsejében valósítjuk meg. Vizsgáljuk meg ezt a megoldást!

Az új grid dimenziója ($m_1Max - 1$, $n_1Max - 1$, $o_1Max - 1$), a threadek száma továbbra is 512, de a threadIdx immár az m_1Max-1 paramétert jelöli. A kernel kiszámítja az n_2Max és o_2Max paramétereket és ennek megfelelően két egymásba ágyazott ciklus segítségével elvégzi az összegzést. Tehát amíg az eddigi megoldásokban egy szálon egy

darab elemi műveletet végeztünk el, ebben a megoldásban m_2Max*o_2Max elemi műveletet végzünk.

Időigény

A minimális és átlagos számításigényű pontok meghatározását ismét képesek voltunk közvetlenül lemérni, azonban az iterációk kiküszöbölésével elvesztettük az információt a maximális számításigényű pontok futásidejéről. Ebben az esetben az a felismerés adja a becslés alapját, hogy mivel az egyes szálak azonos számú műveletet végeznek el, a blokkok számának növelésével egyenes arányosan változik a futásidő. A blokkok számát manuálisan lekorlátozva és a végrehajtás sebességét megmérve ismét egy jó közelítést adhatunk a tényleges értékre. Az eredményeket az alábbi táblázat foglalja össze:

$t_{\min}$	$t_{\max}$	t_{avg}	t_{all}
7.66 s	$4.08 \cdot 10^4$ s (11.33 óra)	2387.78 s (39.80 perc)	$3.73 \cdot 10^7$ s (431.82 nap)

Látható, hogy a kernelbe bevitt iterációk segítségével kétszeres gyorsulást értünk el, ami annak köszönhető hogy az egyes szálak még így is nagyrészt megegyező végrehajtási utat járnak be, ezért a warpokban hatékonyan végrehajthatók. Ezen túl a költséges iteratív kernelhívást is kiküszöböltük.

4.2.7. Számláló cache

Az eddigi számításaink során az elemi műveletet az eredeti formájában végeztük el, a megoldások kizárólag a párhuzamos végrehjatás megfelelő ütemezésével foglalkoztak. A jelenlegi és a következő megoldások egy-egy gyorsítótárazási lehetőséget mutatnak be, rendre a számláló és a nevező értékeire. A gyorsítótárazás lényege, hogy bizonyos számításokat előre elvégzünk, így a tényleges műveletvégzés során ezeket csak a megfelelő memóriaterületről kell kiolvasnunk.

A számláló gyorsítótárazása során az

$$s[m_1]s[m_1 + a_1 \cdot 2^6]s[n1]s[n_1 + a_2 \cdot 2^6]s[o_1]s[o_1 + a_3 \cdot 2^6]$$

értékeket eltárolását tűzzük ki célul, melyek természetesen megegyeznek az

$$s[m_2]s[m_2 + b_1 \cdot 2^6]s[n_2]s[n_2 + b_2 \cdot 2^6]s[o_2]s[o_2 + b_3 \cdot 2^6]$$

kifejezés lehetséges értékeivel. Ennek megfelelően a továbbiakban csak az a értékekhez kapcsolódó példákkal élek, de feltételezhetjük, hogy azok a b értékekre is igazak.

Első megközelítésben úgy tűnhet, hogy a kifejezés eltárolása nem lehetséges, mivel az m_1, n_1, o_1 változók értékkészlete 320, elemű, az a_i változóké pedig 5 elemű. Ebből adódóan egy $[a_1, a_2, a_3, m_1, n_1, a_2]$ index elemeinek száma $320^3 \cdot 5^3 = 4096000000$ lenne, ami double értékeket használva $30.51GB$, ami meghaladja a globális memória méretét.

Megállapíthatjuk azonban, hogy az a_1, m_1 páros lehetséges kombinációinak száma a köztük levő összefüggés ($m_1Max = 320 - a_1 \cdot 2^6$) miatt ennél jóval kevesebb,

$$\sum_{a=0}^{4} 320 - a \cdot 2^6 = 320 + 256 + 192 + 128 + 64 = 960$$

Ezt felhasználva, már csak 960^3 lehetőségről beszélhetünk, ami 6.59GB memóriaterületen eltárolható.

A cache építése két lépésből áll. Az első lépésben létrehozunk egy "numIdx" indexet, amely minden $a1, m1$ párhoz egy azonosítót rendel, tehát leképezi az említett [0, 960) tartományra:

$$\text{numIdx}(a_1, m_1) = \text{<a számpár azonosítója>}$$

Ezt a numIdx indexet fogjuk a cache címzésére használni, ezért eltároljuk a konstans memóriában.

Ezt követően felépítjük a tényleges gyorsítótárat. Ehhez egy kernel segítségével végigiterálunk az összes $a_1, a_2, a_3, m_1, n_1, o_1$ lehetőségen és az ezekhez tartozó számláló értéket eltároljuk a

$$\text{numerator\_cache}[\text{numIdx}(a_1, m_1)][\text{numIdx}(a_2, n_1)][\text{numIdx}(a_3, o_1)]$$

helyen. Az integrált számító kernelünkben ezt úgy használhatjuk fel, hogy a konstans memóriából kikeressük a $\text{numIdx}(a_1, m_1), \text{numIdx}(a_2, m_2), \text{numIdx}(a_3, o_1)$, valamint a $\text{numIdx}(b_1, m_2), \text{numIdx}(b_2, m_2), \text{numIdx}(b_3, o_2)$ értékeket, majd a cache hozzájuk tartozó helyein található kettő érték szorzatából előállítjuk a számlálót.

Időigény

Az egyes pontok számítási idejére az alábbi értékek adódtak:

t_{min}	t_{max}	t_{avg}	t_{all}
8.39 s	$4.55 \cdot 10^4$ s (12.64 óra)	2598.62 s (43.31 perc)	$4.06 \cdot 10^7$ s (469.95 nap)

Az eredmények alapján teljesítménybeli visszaesést tapasztaltunk, ami könnyen magyarázható, ha megizsgáljuk a számláló megállapításához szükséges műveletek számát, amit a 4.3 táblázat foglal össze.

	Cache nélkül	Cache használatával
Konstans memória olvasás	6	6
Globális memória olvasás	-	2
Integer összeadás	6	10
Integer szorzás	-	10
Double szorzás	11	1

4.3. táblázat. Számláló meghatározásának műveleti igénye

A cache használatával tíz lebegőpontos szorzást spóroltunk meg, de helyette tíz egész szorzást, négy egész összeadást és ami a legkritikusabb, kettő globális memória elérést vezettünk be. Ezek végrehajtása a tapasztalat szerint több időt vesz igénybe. A következő megoldásban nem alkalmazzuk a számláló gyorsítótárazását.

4.2.8. Nevező cache

Az előző megoldásból azt a következtetést vonhatjuk le, hogy a cache használata olyan esetekben célravezető, amikor annak elemei gyorsan elérhetők és kellően költséges műveleteket váltanak ki.

A nevező értékeinek számításakor egy hasznos optimalizációt fedezhetünk fel. Megállapíthatjuk ugyanis, hogy az $n_1 - m_1$ és ezzel analóg kifejezések lehetséges értékkészlete a $[-320, 320]$ tartomány, melynek számossága 641. Ebből az következik, hogy adott c_1, c_2, c_3 értékek mellett a nevező csak $621^3 = 263374721$ különböző értéket vehet fel, ami 1.96GB tárterületen ábrázolható.

Ha az $n_1 - m_1$ értéket használjuk a cache indexelésére, akkor sokkal gazdaságossabbann járhatunk el, mint a számláló cache esetében, ugyanis itt a $numIdx$ használata helyett elegendő csak egy 320 méretű eltolást alkalmaznunk és megkapjuk az adott $n_1 - m_1$ különbséghez tartozó azonosító indexet. Ennek köszönhetően építhetünk egy nevező gyorsítótárat, mely az alábbi elemeket tárolja:

$$\text{denominator\_cache}[m_1 - m_2 + 320][n_1 - n_2 + 320][o_1 - o_2 + 320] =$$

$$\frac{1}{\sqrt{\left(\frac{m_1-m_2}{2^6} + c_1\right)^2 + \left(\frac{n_1-n_2}{2^6} + c_2\right)^2 + \left(\frac{o_1-o_2}{2^6} + c_3\right)^2}}$$

Már a konstrukcióból is látható, hogy az index használatával egy kellően összetett kifejezést váltunk ki. A cache használatát elősegíthetjük azzal, hogy az eddigi kernel dimenziók helyett áttérünk az $m_1, m_2, n_1, n_2, o_1, o_2$ sorrendre, így a kernelben található ciklus során a cache index előállításához szükséges változók korábban előállnak.

A megoldás nagy előnye, hogy a gyökös kifejezés ellentettjét tároljuk, így az osztást egy szorzásra cserélhetjük, valamint a nullával való osztás ellenőrzését kiszervezhetjük a cache építés pillanatára. Amikor a gyökös kifejezés nulla és a művelet nem értelmezhető, egy nulla szorzótényezőt tárolhatunk a gyorsítótárban, ezzel érvénytelenítve a megfelelő elemi műveletek eredményét. Ennek az a következménye, hogy az elemi művelet számítása során nincs szükség elágazásra, elegendő csak a számláló értékének szorzása a nevező gyorsítótárazott értékével.

Időigény

A számláló gyorsítótárazásával ellentétben ez a megoldás minden c érték mellett a cache újraszámítását igényli, ami 0.38 másodpercet vesz igénybe. Az alábbi értékek egy-egy integrál meghatározását becslik, ezért ezt a cache építést is magukba foglalják. Ennek ellenére jól látható, hogy határozott javulást értünk el az előző megoldásokhoz képest. Az eddigiektől eltérően, a maximális számításigényű pontot is gyakorlati méréssel határoztam meg.

$t_{\min}$	$t_{\max}$	t_{avg}	t_{all}
1.54 s	$3.34 \cdot 10^4$ s (9.28 óra)	1421.37 s (23.69 perc)	$2.22 \cdot 10^7$ s (257 nap)

Az eddigi megoldások során a célunk mindig egyetlen érték meghatározása volt, azonban nagymértékű optimalizációra van lehetőségünk, ha egyszerre több megoldást számítunk ki. Egy ilyen megoldást mutatok be a 4.2.12 részben, viszont ezt a hardver erőforrások korlátai miatt a probléma kettődimenziós változatán szemléltetem. Ehhez

először vizsgáljuk meg a 2D probléma CPU és egyszerű GPU alapú megvalósítását az összehasonlíthatóság érdekében.

4.2.9. 2D probléma CPU alapú megvalósítása

A 2D problémát a 4.1 numerikus integrálhoz hasonnlóan definiálhatjuk, a különbség csak az egyes vektorok méretéből adódik.

$$I_{abc}^{(m)} = 2^m \sum_{m_1=1}^{m_1Max} \sum_{n_1=1}^{n_1Max} \sum_{m_2=1}^{m_2Max} \sum_{n_2=1}^{n_2Max} \left[\frac{s(m_1)s(m_1 + a_1 \cdot 2^m)s(n_1)s(n_1 + a_2 \cdot 2^m)}{s(m_2)s(m_2 + b_1 \cdot 2^m)s(n_2)s(n_2 + b_2 \cdot 2^m)} \right] \cdot \left(\frac{1}{2^m} \right)^4 \tag{4.2}$$

ahol

$$a_i, b_i, m_i, n_i, \in \mathbb{Z}^{0+} \quad a_i, b_i < N \quad c_i \in \mathbb{R} \qquad \text{valamint} \qquad \begin{array}{c} m_1Max = S - a_1 \cdot 2^m \\ n_1Max = S - a_2 \cdot 2^m \\ m_2Max = S - b_1 \cdot 2^m \\ n_2Max = S - b_2 \cdot 2^m \end{array}$$

A megvalósítás ennek megfelelően a 4.2.1 részben bemutatott 3D implementáció egyszerűsített változata, az eddigi 6 helyett 4 egymásba ágyazott iteráció segítségével.

Időigény

A kétdimenziós esetekben a kevesebb szükséges számításnak köszönhetően minden érték becslés nélkül meghatározható, a futási időkre az alábbi értékeket kaptam:

t_{min}	t_{max}	t_{avg}	t_{all}
0.21 s	121.44 s	17.12 s	10601.09 s

4.2.10. Egyszerű 2D GPU implementáció

Az egyszerű GPU implementáció a 3D megvalósítások közül a 4.2.5 részben bemutatott megoldás megfelelője.

Egy fontos különbség, hogy a kétdimenziós esetben nincs szükség iterációra sem a kernelen belül, sem azon kívül. Ez annak köszönhető, hogy a négy ciklusváltozót felhasználhatjuk a grid és blokk dimenzióiként.

Időigény

Láthatjuk, hogy a párhuzamos számítás a megoldást nagymértékben felgyorsította.

t_{min}	t_{max}	t_{avg}	t_{all}
0.0039 s	0.80 s	0.13 s	84.06 s

4.2.11. 2D probléma cache alkalmazásával

Miután megismertük az egyszerűbb algoritmusok futásidejét, vizsgáljuk meg azt megoldást, ami elvezet minket több pont együttes számításához.

A megoldás alapötlete a 4.2.8 fejezetben bemutatott nevező cache. Láthattuk, hogy a nevező értékét egy adott c érték mellett felírhatjuk az $m_1 - m_2$ és $n_1 - n_2$ kifejezések függvényeként, és ezt fel is használtuk a gyorsítótárazásra. Belátható, hogy előnyös lenne a számunkra, ha a számláló értékeit is úgy gyorsítótárazhatnánk, hogy az eléréshez ugyanezt az indexet használjuk. A továbbiakban megmutatom, hogy hogyan lehetséges egy ilyen cache elkészítése.
Vegyük az a, b, c paramétereket konstansnak, ekkor az integrálunk

$$I = \sum_m \sum_n \frac{f(m)f(n)}{g(m,n)} \tag{4.3}$$

alakú, ahol a g nevező értékkészlete egy véges elemű halmaz. Ennek számossága legyen k és az i-edik elemét jelöljük g_i-vel. Ekkor az egyenletünket felírhatjuk az alábbi formában:

$$I = \sum_{i=1}^{k} \frac{1}{g_i} \left(\sum_m \sum_n f(m)f(n) \middle| g(m,n) = g_i \right) \tag{4.4}$$

Ez azt jelenti, hogy "szétválaszthatjuk" a számlálót és a nevezőt és ilyen módon a nevező minden lehetséges értékéhez meghatározhatunk egy együtthatót, amivel az adott összegben részt vesz. Ez az együttható pedig az adott nevezőértékhez tartozó számlálók összegeként áll elő.

Az alábbi összefüggés az ilyen módon definiált gyorsítótár felépítéséhez nyújt alapot:

$$\left(\sum_m \sum_n f(m)f(n) \middle| g(m,n) = g_i \right) = \left(\left[\sum_m f(m) \right] \cdot \left[\sum_n f(n) \right] \middle| g(m,n) = g_i \right) \tag{4.5}$$

Ez az azonosság lehetőséget ad arra, hogy a gyorsítótárat kettő lépésben építsük fel. Először elkészíthetünk egy egydimenziós változatot, majd ezt felhasználhatjuk a kétdimenziós cache felépítésére.

A leírást azért kezdtük a bemeneti változók fixálásával, mert a számláló cache értékei az a és b paraméterektől függenek. A kétdimenziós probléma mérete azonban lehetőséget ad arra, hogy a gyorsítótárat a teljes a, b tartományra előre felépítsük. Az a_i, b_i kombinációk száma 5^4, az $m_1 - m_2$ és $n_1 - n_2$ kombinációk száma pedig 640^2. Ebből következően a teljes cache tárolása $5^4 \cdot 640^2 \cdot 8$ bájtot, azaz 1.9GB-ot vesz igénybe.

A nevezőhöz a háromdimenziós módszernél megismert gyorsítótárat készítjük el. Mivel adott a és b értékek mellett a számlálót felírhatjuk $m_1 - m_2$ és $n_1 - n2$ függvényeként, valamint a nevezővel ugyanezt megtehetjük egy adott c érték mellett, ezért az eredeti $m_1Max * n_1Max * m2_Max * n_2Max$ elemi művelet elvégzését átalakítottuk 640^2 darab szorzás elvégzésére.

Ennek megfelelően megtehetjük, hogy a kernel futtatási konfigurációjában az eddig használt m_i, n_i értékek helyett az $m_1 - m_2 + 320$ és az $n_1 - n_2 + 320$ cache indexekkel "iteráljunk". Az így definiált kernel szálainak tevékenysége a kettő indexnek megfelelő értékek eléréséből, valamint ezek összeszorzásából áll.

Időigény

Láthatjuk, hogy a módszer nagyon jó eredményeket ért el, és megvan az az előnye, hogy az egyes pontok számításigénye közel azonos. A különbségek az ütemezés és műveletvégzés CUDA implementációjának tudhatók be.

t_{min}	t_{max}	t_{avg}	t_{all}
0.000071 s	0.000071 s	0.000080 s	0.049 s

4.2.12. Pontok együttes számítása

A legutóbbi pontban láthattuk, hogy a teljes számláló gyorsítótár elfér a memóriában, valamint a kernelfuttatást is négydimenziósról kettőre csökkentettük. Ez lehetőséget ad arra, hogy egy új kernelfuttatási konfigurációban a grid egy dimenziójaként átadjuk a szálaknak az a_i, b_i eltolások lehetséges értékeit, tehát minden szál különböző a_i és b_i értékekhez tartozó szorzatot számoljon.

Ilyen módon egy adott c esetén a nevező cache kiszámítása után egyetlen kernelhívás segítségével az összes lehetséges a_i, b_i bemenetre visszakapjuk az eredményt.

Időigény

Ennek a megoldásnak az a különlegessége, hogy mivel minden pontot egyszerre határozunk meg, ezért minden pont kiszámítása ugyanakkora számításigényű, valamint megegyezik az összes pont megállapításának számítási idejével.

t_{min}	t_{max}	t_{avg}	t_{all}
0.0064 s	0.0064 s	0.0064 s	0.0064 s

Mivel az összes pontot 0.0064 másodperc alatt képesek vagyunk meghatározni, megállapíthatjuk, hogy a CPU alapú megoldáshoz viszonyítva az egy c értékhez tartozó pontok számítási idejének tekintetében több, mint 1.6 milliószoros javulást értünk el.

4.3. Integrálszámítási módszerek

Láthattuk, hogy az integrál értékének meghatározása nagy mértékben gyorsítható a különböző CUDA alapú implementációkkal. Emellett azt is megismerhettük, hogy a nagy számítási igény forrása az egy darab érték meghatározásához kiszámítandó elemi műveletek nagy mennyisége.

Felmerül a kérdés, hogy lehetséges-e az m_i, n_i, o_i értékek ritkább mintavételezése mellett különböző approximációkat alkalmazni és ezáltal gyorsítani a megoldás menetét. Az előző részben bemutatott implementációk során a jobboldali Riemann integrált használtam. Ennek az a legnagyobb előnye, hogy könnyen számítható, mivel egy szakaszon vett integrálérték közelítésére csak egy-egy függvényértéket használ az egyéb megismert módszerekkel szemben, melyek ezt kettő vagy több érték súlyozásával teszik meg. Ebből adódóan érdemes megvizsgálni, hogy ezek a módszerek milyen eredményeket érnek el.

Ebben a részben a méréshez kapcsolódó különböző megfontolásokat ismertetem, az eredményekre az 5. fejezetben térek ki.

4.3.1. A mérés terve

A célunk tehát az, hogy a kételektron-integrálban szereplő m_i, n_i, o_i értékek különböző felbontású mintavételezései mellett az integrál számítására különböző kvadratúrákat alkalmazva megállapítsuk azok eredményét és ezt összehasonlítsuk az eddigi implementációk által számított értékekkel.

Sajnos a különböző kvadratúrák megvalósítása magasabb dimenzióban, párhuzamos számítást alkalmazva nem magától értetődő, ezért úgy döntöttem, hogy a mérést hagyományos módon, a host számítógépen végzem el. Ebben a tekintetben az egyes algoritmusok futásideje nem releváns, viszont lehetőségünk van a különböző eredmények kiértékelésére.

A megvalósítás azon alapul, hogy a kételektron-integrál számítása során az elemi műveletek eredményeit könnyen elő tudjuk állítani. Ahelyett viszont, hogy ezeket összegeznénk, letöltjük őket a host memóriába. Ilyen módon a már rendelkezésre álló diszkrét adathalmazon egyszerűen elvégezhetjük a kívánt számításokat.

4.3.2. Implementációs megfontolások

Az ismertetett okokból adódóan arra van szükségünk, hogy különböző kvadraturákat megvalósító függvényeket implementáljunk, melyek képesek magasabb dimenziójú adathalmazok kiértékelésére. A mérés elvégzéséhez az alábbi numerikus integrálszámítási módszereket implementáltam:

- Jobboldali Riemann-integrál
- Trapéz szabály
- Simpson-módszer
- Monte-Carlo alapú becslés

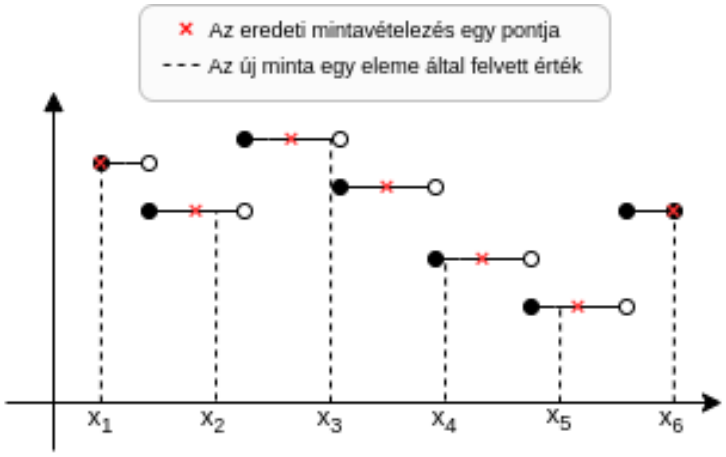
Az első három módszer a magasabb dimenziójú integrálszámítást rekurzióval oldja meg. Paraméterként megkapják az adathalmazt és annak méreteit, majd az aktuálisan számítandó függvényértéket a saját maguk által definiált művelet eggyel alacsonyabb dimenziójú problémájaként számítják ki. Amikor a folyamat elér az egydimenziós problémához, a hagyományos módszert alkalmazzuk. Ez a megoldás természetesen nem optimális, de annak eldöntésére alkalmas, hogy érdemes-e az egyes módszerekkel foglalkoznunk a jövőben.

A Simpson-módszer sajátossága, hogy működéséhez páratlan számú pontot igényel. Ezt a problémát úgy küszöböltem ki, hogy amennyiben páros számú pontot kell számolnia, az utolsó szakaszt a trapéz szabály segítségével közelíti.

A Monte-Carlo alapú integrálszámítás általam készített implementációjának sajátossága, hogy azt egymás utáni, progresszív mintavételezésre készitettem fel. Ez azt jelenti, hogy minden új felbontás (amit ebben az esetben több véletlenszerű pont mintavételeként értelmezek) esetén az eddig meglévő véletlenszerű ponthalmazt egészíti ki a szükséges mennyiségű új pont hozzávételével. Erre azért volt szükség, mivel a módszer nagy mennyiségű adatra alkalmazva nagyon lelassult, feltehetőleg a nagy számú, véletlenszerű memóriaelérések következtében.

Az utolsó implementációs megfontolás a felosztások működésével kapcsolatos. A Monte-Carlo integrálást leszámítva a módszerek a teljes tartomány egyenletes mintavételezésén alapulnak. Az előállított adathalmazra különböző, az eredtinél ritkább mintavételezést szerettem volna alkalmazni. Ennek velejárója, hogy ugyanazt a tartományt különböző méretű részekre osztva, azt osztópontok nem mindig az eredeti osztópontokra

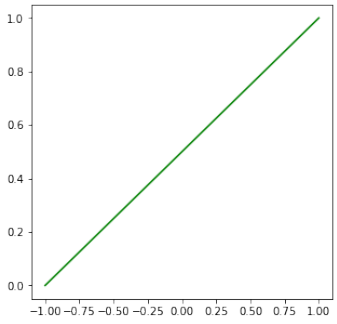
esnek. Ezeken a helyeken azonban nem ismerjük a függvényértéket. A megoldás során ilyen esetekben egy adott helyen felvett értéknek a legközelebb eső mintavételezett értéket választottam. Ezt szemlélteti az alábbi ábra:



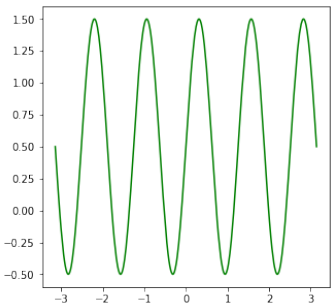
4.4. ábra. Az újonnan mintavételezett minta pontjai a hozzájuk legközelebb áló pont értékét veszik fel

4.3.3. Ellenőrzés

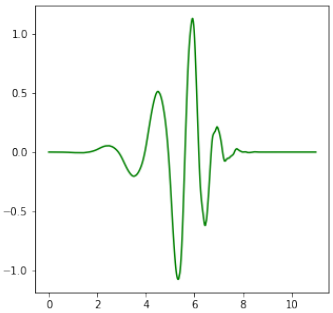
Ahhoz, hogy az egyes megoldások eredményét ellenőrizhessem, olyan különböző egy- és kétdimenziós függvényeken teszteltem őket, melyek esetén a tényleges integrál értéke analitikusan meghatározható, vagy legalábbis előre ismert. Ezek a függvények az alábbiak voltak:



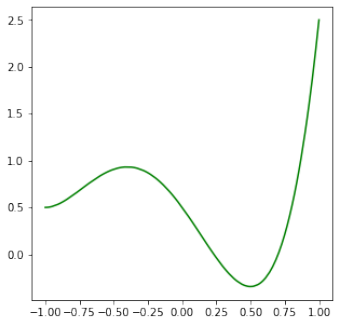
(a) $y = 0.5x + 0.5$



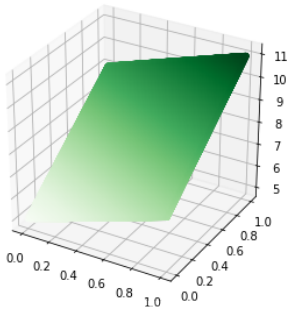
(b) $sin(5x) + 0.5$



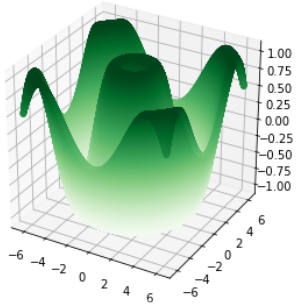
(c) Daubechies-6 wavelet



(d) $y = 2.5x^4 + 3x^3 - 1.5x^2 - 2x + 0.5$

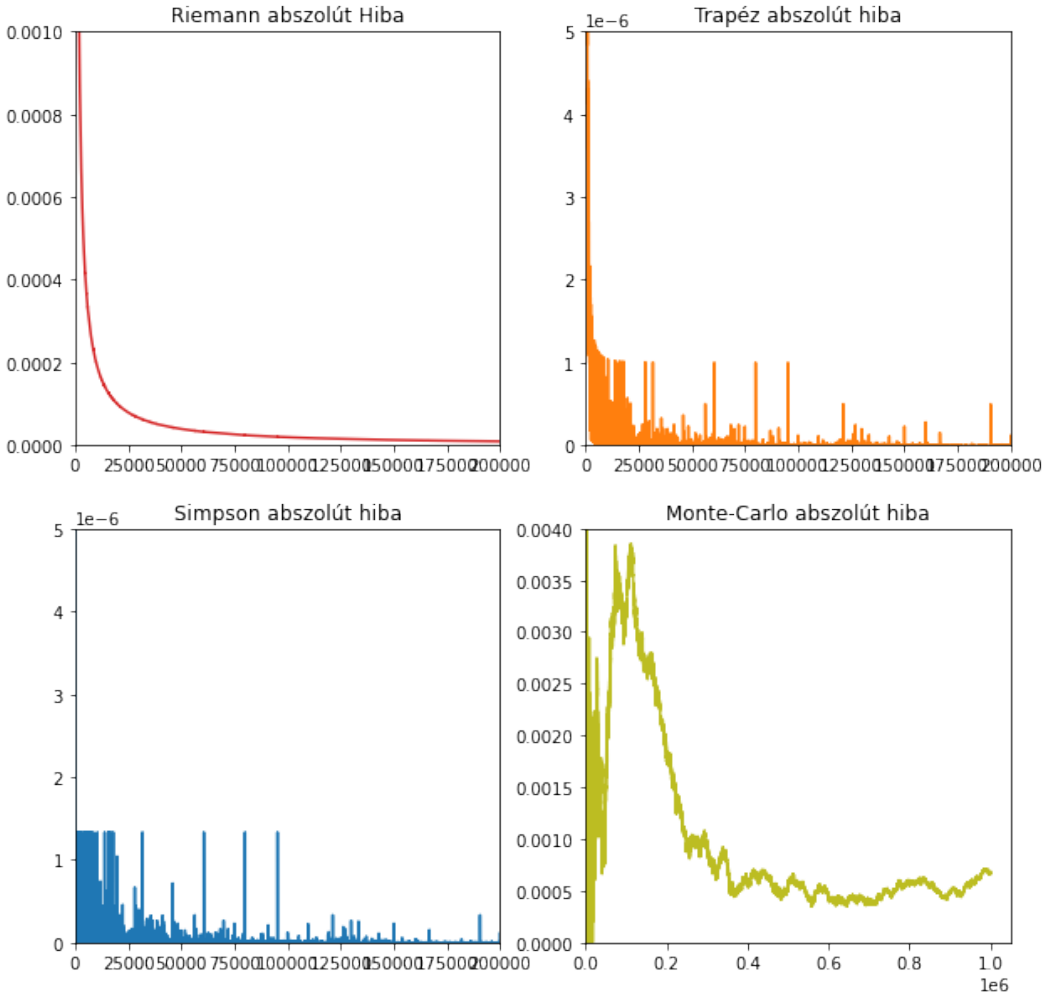


(e) $z = 2x + 4y + 5$



(f) $z = sin(\sqrt{x^2 + y^2})$

Az egyes függvényeket egymillió pontban mintavételeztem, ami a kétdimenziós esetekben dimenziónként ezer pontot jelent. Ezt követően a különböző kvadratúrákat különböző felbontással futtatva megállapítottam, hogy az eredmények valóban a megfelelő értékhez tartanak. A lefutás egy példáját mutatja a 4.6 ábra. Az egyes diagramok az abszolút hiba alakulását ábrázolják a mintavételezett pontok függvényében.



4.6. ábra. A negyedfokú polinom integrálása különböző módszerekkel, különböző mintavételezéssel

Érdemes megjegyezni, hogy az egyes diagramok különböző nagyságrendeket ábrzolnak. Az ábrán többek között az figyelhető meg, hogy a Riemann módszerhez képest a Simpson- és a trapéz módszerek jóval kevesebb lépésben közelítik meg az integrál tényleges értékét, ezt követően pedig csak kisebb eltéréseket produkálnak, ami a műveletek pontatlanságából is adódhat. Látható, hogy a Monte-Carlo módszer is a várható kezdeti egyenetlenségek után lassan a cél érték felé közelít.

5. fejezet

Kiértékelés

Ebben a fejezetben a különböző implementációk teljesítményét és hibáját, valamint az előző fejezetben bemutatott integrálszámítási módszerek eredményeit mutatom be.

5.1. Implementációk eredményei

5.1.1. Számítás helyessége

Mivel a számítással kapcsolatban nem álltak rendelkezésre előzetes eredmények, ezért a hiba megállapításánál a CPU alapú megoldást vettem alapul. A CPU alapú implementáció átlátható, ezért könnyen meggyőződhetünk arról, hogy valóban az elvárt működést valósítja meg. A módszer hátránya, hogy a megoldás nagyon lassú, ezért a referencia eredményhalmazt csak bizonyos korlátok között voltam képes előállítani.

A mérés elvégzéséhez 100 darab minimális számításigényű pontban ($a_i, b_i = 4$) állapítottam meg a függvényértéket, ez mintegy 116 órát vett igénybe. A c_i értékeket egy -5 és 5 közti intervallumon vett egyenletes eloszlással generáltam.

Az arra alkalmas implementációk mindegyikét lefuttattam erre a bementi adathalmazra, így képes voltam összehasonlítást adni az eredményekre.

5.1.2. Számítási idő mérése

Az egyes algoritmusok futásidejének mérésére a time.h standard C könyvtár clock() függvényét használtam. Bizonyos esetekben a hosszú futásidő miatt nem volt lehetőségem a teljes időigény megállapítására. Ilyenkor a ciklusokba ágyazott, bizonyos iterációnkénti időméréssel, vagy a blokkok számának manipulálásával állapítottam meg a becsült értéket. Ezeket a módszereket az implementációk bemutatása során ismertettem. Azzal a feltételezéssel éltem, hogy az idő mérése és kiírása elhanyagolható számítási- és időtöbblettel jár a kernel futásához képest.

5.1.3. Eredmények

A háromdimenziós implementációk teljesítményét az 5.1 táblázat foglalja össze.

#	Implementáció	t_{min} (s)	t_{max} (s)	t_{avg} (s)	t_{all} (s)
1	**CPU**	4199.95	$6.56 \cdot 10^7$ *	$3.06 \cdot 10^6$ *	$4.78 \cdot 10^{10}$ *
2	**Naiv GPU**	$4.86 \cdot 10^9$ *	$5.35 * 10^9$ *	$4.88 \cdot 10^9$ *	$7.63 \cdot 10^{13}$ *
3	**Device összeadás**	$4.17 \cdot 10^7$ *	$1.32 * 10^8$ *	$4.59 \cdot 10^7$ *	$7.17 \cdot 10^{11}$ *
4	**Blokkok számának növelése**	$2.54 \cdot 10^7$ *	$1.37 * 10^8$ *	$3.06 \cdot 10^7$ *	$4.78 \cdot 10^{11}$ *
5	**Kernel dimenziók finomhangolása**	18.18	$1.05 \cdot 10^5$ *	$5.82 \cdot 10^3$	$9.10 \cdot 10^7$ *
6	**Iteráció a kernelben**	7.66	$4.08 \cdot 10^4$ *	$2.39 \cdot 10^3$	$3.73 \cdot 10^7$ *
7	**Számláló cache**	8.39	$4.55 \cdot 10^4$ *	$2.60 \cdot 10^3$	$4.06 \cdot 10^7$ *
8	**Nevező cache**	1.54	$3.34 \cdot 10^4$	$1.42 \cdot 10^3$	$2.22 \cdot 10^7$ *

5.1. táblázat. Az egyes 3D implementációk futási ideje a különböző speciális pontok esetén

A táblázatban csillaggal jelölt értékek becslésekkel lettek előállítva. Láthatjuk, hogy az átlagos számításigényű pontok tekintetében a CPU alapú implementációhoz képest a végső megoldásunkkal több, mint kétezerszeres gyorsulást sikerült elérnünk. Ez azonban sajnos továbbra sem teszi lehetővé a függvényértékek nagy mennyiségben való előzetes meghatározását.

Ezt jól szemlélteti, ha kikötjük, hogy a függvény értékeit minden a_i és b_i értékre szeretnénk meghatározni úgy, hogy a c_i értékeket egy $128 \times 128 \times 128$ méretű rács pontjaiként határozzuk meg. A számítás időigénye a leggyorsabb megoldásunkkal is majdnem másfélmillió évig tartana.

Ennek ellenére az egyedi értékek meghatározásának szempontjából mindenképpen haszonos optimalizációt végeztünk, hiszen a legnagyobb számításigényű pontokhoz az eredetileg kettő évig tartó kalkuláció helyett már kevesebb, mint 10 óra alatt hozzájuthatunk. Ezt az időigényt becslés helyett az érték tényleges kiértékelésével kaptam, így megállapítható, hogy a 4.1.1 részben bemutatott előzetes eredményhez viszonyítva több, mint százszoros javulást értünk el.

#	Implementáció	t_{min} (s)	t_{max} (s)	t_{avg} (s)	t_{all} (s)
9	**2D - CPU**	0.21	121.44	17.12	10601.09
10	**2D - Egyszerű GPU**	0.0039	0.80	0.13	84.06
11	**2D - Cache alkalmazása**	0.000071	0.000071	0.000080	0.049
12	**2D - Pontok együttes számítása**	0.0064	0.0064	0.0064	0.0064

5.2. táblázat. Az egyes 2D implementációk futási ideje a különböző speciális pontok esetén

A kétdimenziós probléma esetén jóval nagyobb gyorsulást sikerült realizálnunk. Az átlagos pontok számítási idejét az egyenkénti számításokkal is ketőssázezred részére csökkentettük, de mutattunk egy olyan megoldást is, amivel a szisztematikus számítások több, mint másfélmilliószoros gyorsulást értek el. A 3D eredményeknél bemutatott analógiával élve amíg egy 128×128 méretű rács számítási ideje eredetileg 5.5 év volt, ezt most 1.5 perc alatt tudjuk elvégezni.

A CUDA alapú implementációk kapcsán vizsgáljuk meg az utolsó táblázatot, amely az egyes módszerek pontosságát mutatja meg a CPU alapú implementáció eredményeihez viszonyítva, a száz darab érték alapján meghatározva.

#	Implementáció	$h_{\mathbf{avg}}$ (%)	$h_{\mathbf{max}}$ (%)
1	**CPU**	-	-
2	**Naiv GPU**	-	-
3	**Device összeadás**	-	-
4	**Blokkok számának növelése**	-	-
5	**Kernel dimenziók finomhangolása**	$3.53 \cdot 10^{-7}$	$3.68 \cdot 10^{-6}$
6	**Iteráció a kernelben**	$3.87 \cdot 10^{-7}$	$4.04 \cdot 10^{-6}$
7	**Számláló cache**	$4.08 \cdot 10^{-7}$	$6.67 \cdot 10^{-6}$
8	**Nevező cache**	$1.19 \cdot 10^{-6}$	$5.11 \cdot 10^{-5}$
9	**2D - CPU**	-	-
10	**2D - Egyszerű GPU**	$7.77 \cdot 10^{-10}$	$8.60 \cdot 10^{-9}$
11	**2D - Cache alkalmazása**	$1.94 \cdot 10^{-9}$	$3.02 \cdot 10^{-8}$
12	**2D - Pontok együttes számítása**	$1.99 \cdot 10^{-9}$	$3.25 \cdot 10^{-8}$

5.3. táblázat. A különböző implementációk által számított eredmények átlagos és maximális relatív hibája

A táblázatból kiolvasható, hogy az átlagos és maximális relatív hibarétékek egyaránt nagyon alacsonyak. Ezek alapján az értékek alapján kijelenthetjük, hogy a párhuzamos implementációk ekvivalensek a CPU alapú implementációval.

5.2. Integrálszámítási módszerek eredményei

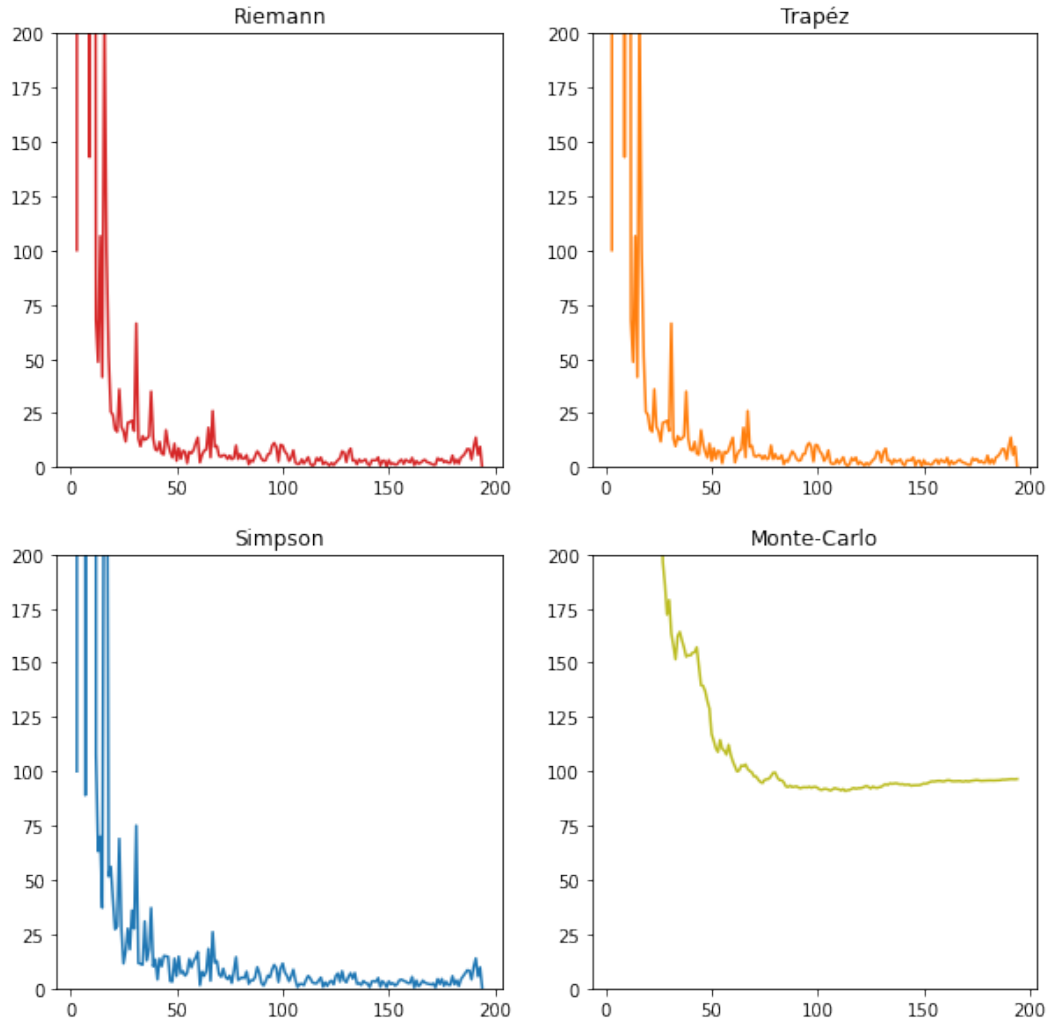
A 4.3 részben ismertettem a módszert, amellyel a különböző kvadratúrák használatával való optimalizáció lehetőségét vizsgáltam. A mérés célja az volt, hogy megállapítsam, vajon különböző numerikus integrálszámítási közelítések használatával lehetséges-e megoldani azt, hogy az m_i, n_i, o_i értékeket nem a teljes értelmezési tartományukon, hanem annak csak egy részhalmazán mintavételezzük és így számítjuk ki az integrál értékét az eredmény minőségének romlása nélkül.

Ehhez a lehető legnagyobb olyan ponthalmazt szerettem volna előállítani a memóriában, ami az m_i, n_i és o_i értékei szerint egy szabályos hiperkocka. Azt találtam, hogy a legnagyobb ilyen adathalmazt a kétdimenziós változat és $a_i = b_i = 2$ esetben érhetem el. Ilyenkor az m_i és n_i értékek száma $321 - 2 \cdot 2^6 = 193$. Ez összesen 193^4 adatelemet jelent, ami 8 bájtos double értékekben tárolva 10.33 GB.

A mérés eredményét ebben az esetben is egy, a CPU implementációval kiértékelt adathalmazhoz viszonyítom, melyet 50 különböző értékre határoztam meg.

A számítás menete az volt, hogy mind az ötven integrál esetén előállítottam az elemi műveletek eredményét a device memóriában egy kernel segítségével. Ezt követően az adathalmazt visszamásoltam a host memóriába. Itt egy 2 felbontású felosztástól indulva egyesével a maximális 193 értékig minden lehetséges rács mentén mintavételeztem a hiperkockát és meghatároztam a különböző módszerekkel számított integrál értékeket.

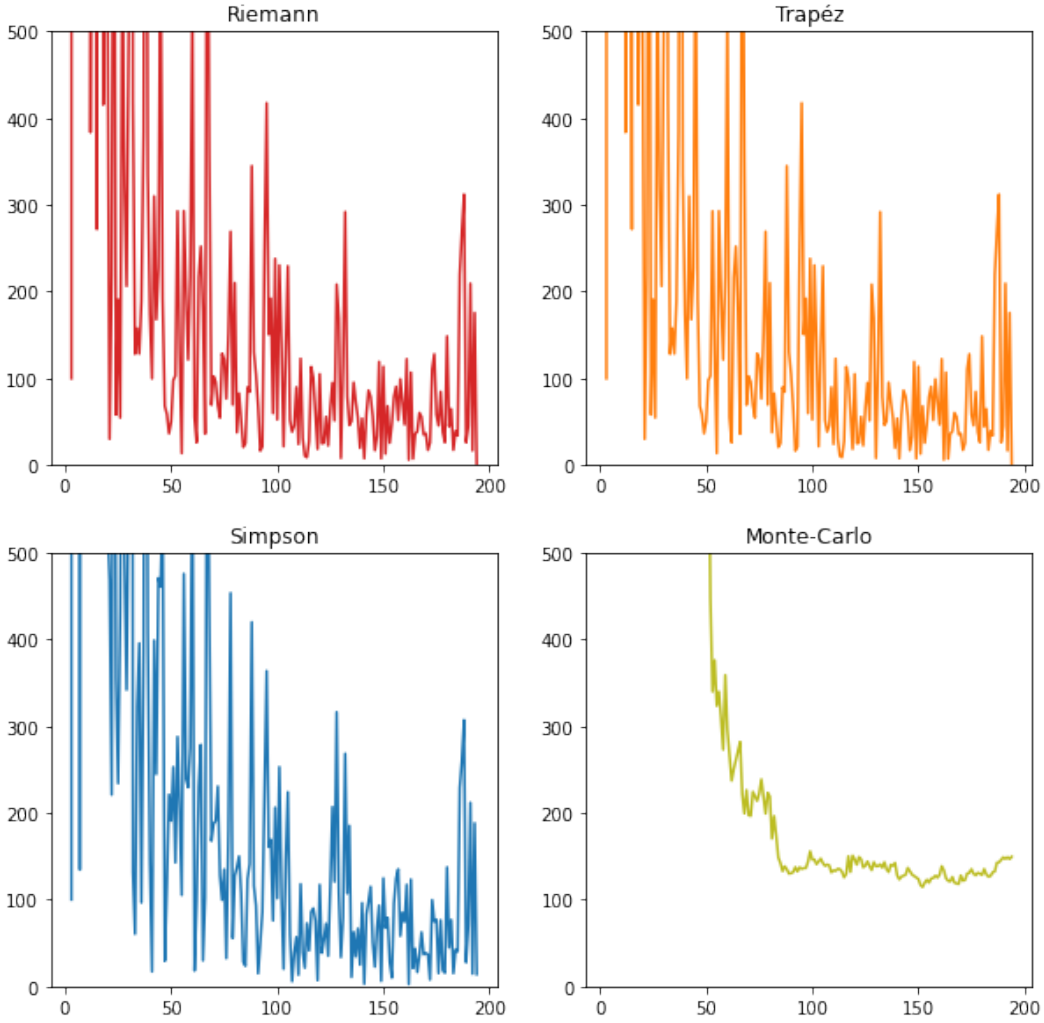
Ezekből az adatokból készültek el a következő ábrák, melyek az egyes módszerek átlagos, valamint maximális relatív hibáit ábrázolják az ötven számításra nézve adott felbontás mellett.



5.1. ábra. Kvadratúrák átlagos relatív hibája (%) a felosztás függvényében

A diagramok alapján az első szembetűnő észrevétel az, hogy a Monte-Carlo módszer a 100%-os relatív hibához tart. Ez annak a következménye, hogy az elemi műveletek eredményei nem elég egyenletesek, a véletlenszerű mintavétel nagyobb valószínűséggel választ olyan pontokat, melyek a 0 értékhez közeliek. Ennek eredményeképp a becsült integrál értéke a 0-hoz tart, ami 100%-os relatív hibát jelent.

Az egyes integrálszámítási módszerek nagyon hasonló eredményeket adnak és a nullához közelítenek, de az átlagos hiba sajnos az 5-10%-os tartományban mozog, ami magasnak számít. Ebből arra a következtetésre juthatunk, hogy nem érdemes az integrál értékek alulmintavételezése. Ezt támasztja alá a következő ábra is, amely az ötven függvényre tekintett maximális relatív hibákat ábrázolja a felbontás függvényében:



5.2. ábra. Kvadratúrák maximális relatív hibája (%) a felosztás függvényében

Ezekből az eredményekből azt láthatjuk, hogy a maximális relatív hiba hektikusan változik a felbontások függvényében. Még viszonylag nagy felbontás mellett is vannak olyan függvények, melyekre a különböző kvadratúrák hibája az eredeti érték háromszorosa.

Az integrálszámítási módszerekkel végzett kísérletünk konklúziója az, hogy az optimalizációt nem érdemes ezek segítségével elvégezni, mivel jelentős hiával járhat az alkalmazásuk.

6. fejezet

Összefoglalás

Dolgozatom első részében bemutattam a Hartree-Fock módszert, mely egy gyakran alkalmazott technika kvantummechanikia elektronszerkezet számítások során. Megtudhattuk, hogy a módszer legszámításigényesebb része az elektronok közötti kölcsönhatást leíró kifejezések, az úgynevezett kételektron-integrálok meghatározása. A problémát súlyosbítja, hogy a kételektron-integrálokat minden esetben újra kell számolni, mivel a számítás alapjául az adott fizikai rendszerre jellemző bázisfüggvényeket használnak.

Ezt követően a wavelet transzformáció elméletén keresztül bemutattam a waveleteket, melyekre építve megismerhettük a Dr. Szeglets Luca és Dr. Pipek János által megalkotott módszert, ami a kételektron-integrálok számítását wavelet skálafüggvények használatával oldja meg. A számítás előnye abban rejlik, hogy az ilyen módon meghatározott kételektron-integrálok újrafelhasználhatók, tehát ha lehetőségünk lenne arra, hogy ezeket előre meghatároznunk, akkor a Hartree-Fock módszer során elegendő lenne az eredmény keresését elvégeznünk. A számítást azonban megnehezíti, hogy a módszer továbbra is meglehetősen számításigényes, ezért egy optimális megoldást kell keresnünk a gyakorlati megvalósításra.

A wavelet alapú módszer nagy mennyiségű, a számítás tekintetében hasonló részeredmény megállapítását vonja magával. Kézenfekvő, hogy a megoldásra valamilyen párhuzamos számítási technológiát alkalmazzunk, ezért bemutattam az általános célú grafikus programozást, valamint az NVIDIA CUDA keretrendszer fontosabb koncepcióit.

Ezt követően a számítógépen való analitikus és numerikus integrálszámítás témakörét foglaltam össze, ezzel megalapozva a megoldásunk elméleti hátterét.

A következő fejezetben levezettem a megoldandó problémát és bemutattam az ezzel kapcsolatos előzetes eredményeket. A probléma rövid elemzése után inkrementális módon bemutattam az optimalizációra készített CUDA alapú megoldásaimat, valamint bemutattam egy kísérlet tervét azzal kapcsolatban, hogy a különböző numerikus kvadratúrák használatával lehetőségünk van-e a szükséges kiszámítandó részeredmények ritkítására.

Az eredmények kiértékelése során a CUDA alapú implementációkkal kapcsolatban megállapítottam, hogy azokkal három nagyságrendbeli gyorsulást érhetünk el a CPU alapú megoldáshoz képest, miközben az eredmények precíziója nem változik számottevően. A kvadratúrákon végzett kísérlet eredményei arra utalnak, hogy a wavelet alapú módszer nem gyorsítható a mintavételezés ritkításával.

7. fejezet

Továbbfejlesztési lehetőségek

A kételektron-integrálok számítása során előállított összegzendő részeredmények a c paraméter függvényében tetszőlegesen kicsi értékek lehetnek. Bár a dupla lebegőpontos számábrázolás akár 10^{-324} nagyságrendbeli értékek tárolását is lehetővé teszi, az ezeken végzett összeadás művelete az operandusok közös exponensre való hozásán alapul, így a precízióvesztés általában elkerülhetetlen. Érdemes lehet megvizsgálni, hogy magasabb pontosságú (kiterjesztett, négyszeres) lebegőpontos reprezentáció esetén milyen eredmények születnek. Az implementációt megnehezíti, hogy ezekre a típusokra nincs natív C és CUDA támogatás, így egyéb, a dobule összeadás pontosságának növelését célzó különböző technikák, mint például Kahan összegzési algoritmus vizsgálata is felmerülhet alternatívaként.

Bár a dolgozatban bemutatott technikákkal jelentős gyorsulás érhető el, ezek továbbra sem teszik lehetővé a függvényértékek nagy mennyiségben való számítását. Arra azonban alkalmasak lehetnek, hogy gyakorlati alkalmazásba ültessük őket és így az eredményt a hagymányos kvantummechanikai számítások kimeneteivel vessük össze. Amennyiben ez sikeresnek bizonyul, különböző architekturákat képzelhetünk el, melyek átmenetileg képesek lehetnek az igények kiszolgálására amíg nem sikerül jelentősebb áttörést elérni a számítás optimalizációjával. Egy ilyen megoldás lehet az eredmények szolgáltatás alapú kiszolgálása. Ekkor a már kiszámított eredményeket eltárolhatnánk és szükség esetén újra felhasználhatnánk.

A kiszámítandó integrálok mennyiségének csökkentése érdekében érdemes lehet megvizsgálni, hogy a már meghatározott függvényértékek közti interpolációval milyen eredményeket érhetünk el.

Amennyiben a célunk továbbra is nagymennyiségű függvényérték meghatározása, érdemes lehet a 4.2.12 részben bemutatott algoritmus alkalmazása a háromdimenziós problémára. Bár az erőforráskorlátok miatt ez a módszer nem ültethető át egy az egyben, arra képes lehet, hogy az a és b bemeneti változók által meghatározott tér kisebb részeit lefedve végezzünk el több egyidejű számítást.

Köszönetnyilvánítás

Hálás köszönetemet szeretném kifejezni konzulensemnek, Dr. Szegletes Lucának a dolgozat alapjául szolgáló módszer kidolgozásáért, valamint a munkám során nyújtott készséges segítségéért és hasznos tanácsaiért.

Irodalomjegyzék

[1] Amd ryzen threadripper pro 3995wx - benchmark, test and specs. URL https://www.cpu-monkey.com/en/cpu-amd_ryzen_threadripper_pro_3995wx.

[2] Amd ryzen threadripper pro 3995wx. URL https://www.cpubenchmark.net/cpu.php?cpu=AMD%2BRyzen%2BThreadripper%2BPRO%2B3995WX&id=3837.

[3] Horváth Kristóf Attila: Kételektron-integrálok wavelet alapú számítása párhuzamos programozás segítségével. 2013. 10.

[4] B Buchberger – G E Collins – R Loos (szerk.): *Computer algebra: Symbolic and algebraic computation*. 1982. kiad. Vienna, Austria, 2013, Springer.

[5] Cuda c++ programming guide.
URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[6] Pataki Gergely Tasnádi Tamás Fritz Józsefné, Kónya Ilona: *Matematika 1.* 2011, Typotex Kiadó. ISBN 9789632794457.

[7] K.O. Geddes – S.R. Czapor – G. Labahn: *Algorithms for Computer Algebra*. 2007, Springer US. ISBN 9780585332475.
URL https://books.google.hu/books?id=9fOUwkkRxT4C.

[8] Norbert Hesselmann: *Digitális jelfeldolgozás*. Budapest, 1985, Műszaki Könyvkiadó.

[9] Zhe Jia – Marco Maggioni – Benjamin Staiger – Daniele Paolo Scarpazza: Dissecting the NVIDIA volta GPU architecture via microbenchmarking. *CoRR*, abs/1804.06826. évf. (2018). URL http://arxiv.org/abs/1804.06826.

[10] Nasser Kehtarnavaz: Chapter 7 - frequency domain processing. In Nasser Kehtarnavaz (szerk.): *Digital Signal Processing System Design (Second Edition)*. Second edition. kiad. Burlington, 2008, Academic Press, 175–196. p. ISBN 978-0-12-374490-6. URL https://www.sciencedirect.com/science/article/pii/B9780123744906000076.

[11] Qingkai Kong – Timmy Siauw – Alexandre M Bayen: *Python programming and numerical methods : a guide for engineers and scientists*. San Diego, CA, 2021, Academic Press.

[12] Alfred Mertins – Dr Mertins: Signal analysis: Wavelets, filter banks, time-frequency transforms and applications. 2001. 11.

[13] Szilvia Nagy – Janos Pipek: Multiresolution analysis of density operators, electron density, and energy functionals. *International Journal of Quantum Chemistry*, 84. évf. (2001. 01), 523 – 529. p.

[14] Nvidia cuda c programming guide, version 4.2, 2012. Apr.
URL https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

[15] Nvidia geforce rtx 3090 ti specs.
URL https://www.techpowerup.com/gpu-specs/geforce-rtx-3090-ti.c3829.

[16] Bogdan Oancea – Tudorel Andrei – Raluca Mariana Dragoescu: Gpgpu computing. 2014. URL https://arxiv.org/abs/1408.6923.

[17] F.L. Pilar: *Elementary Quantum Chemistry*. Dover Books on Chemistry Series sorozat. 2001, Dover Publications. ISBN 9780486414645.
URL https://books.google.hu/books?id=XpGM7r69LdkC.

[18] István Vajk – Luca Szegletes – János Pipek: *Evaluation of coulombic integrals in electron structure calculations using a wavelet approach*. 2012, BUTE Department of Automation and Applied Informatics, 70–81. p.

[19] C. Valens: A really friendly guide to wavelets. 1999.

[20] Brani Vidakovic – Peter Mueller: Wavelets for kids: A tutorial introduction. Jelentés, 1991, Duke University.

[21] Joachim von zur Gathen – Jürgen Gerhard: *Modern Computer Algebra*. 3. kiad. 2013, Cambridge University Press.

[22] Stefan Weinzierl: Introduction to monte carlo methods, 2000.