Beadandó feladatok

Feladatok a Grafikus processzorok tudományos célú programozása c. óra 2018-as vizsgájához

A vizsgára bocsájtáshoz mindenkinek 1 feladat megoldását kell beküldenie a választott vizsgaidőpontja előtt legalább 1 héttel a gpu (kukac) wigner.mta.hu címre. Aki nem küld be elfogadható megoldást, az nem vizsgázhat és nem szerezhet jegyet. A megoldásnak szabványos C++14/ 17-ben kell elkészülnie, ezért célszerű több fordítóval is ellenőrizni a helyességet. A feladatoknál elvárás, hogy a standard library-t használjuk, ahol lehet és célszerű, továbbá ahol jelöltük a feladatokat VAGY CPU soros és párhuzamos, VAGY GPU párhuzamos, VAGY több optimalizációs szint (pl. blokkosítás) összehasonlításával oldjuk meg. Elvárás, hogy a programok helyesen működjenek, a foglalt memóriát felszabadítsák, ne indexeljék túl a tömböket, ne legyenek bennük race condition-ok, ne segfaultoljanak, stb.

Lehetőség van rá, hogy a teljesen vagy részben megoldott feladatokat ellenőrzésre, vagy segítségkérésre elküldjétek nekünk a végső beadás előtt, amely az értékelésbe nem számít bele, de segít tisztázni félreértéseket, vagy hiányosságokat a megoldásban, javasoljuk, hogy használjátok ki ezt a lehetőséget.

Egyéb hasonló jellegű feladat is választható (pl. mert az valakinek kell/hasznos a szakdolgozatához, tdk-hoz, stb), ezt kérjük előre egyeztessétek!

Szeretnénk, ha minden feladatot csak 1 ember választana. Ahol variációk vannak megjelölve, ott 1 ember 1 variációt válasszon. A választott feladat számát és a variációját küldjétek el email-ben, a már választott feladatok jelzésre kerülnek a weboldalon. Aki változtatni szeretne, az jelezze, hogy melyik helyett melyiket választja, és frissítjük a weboldalon az adatokat.

A feladatok nehézségére igyekszünk utalni a feladatkiírásban, de ez nem feltétlen jó metrika mindenkinek, gondoljátok át, mielőtt feladatot választotok. A feladatkiírás nem feltétlen tartalmaz minden részletet, ha valamit úgy éreztek hiányzik, vagy nincs kellően definiálva, kérdezzetek rá.

1. 1D konvolúciós algoritmus írása [nagyon könnyű]

Írjunk olyan algoritmust, amely hasonló interfésszel rendelkezik, mint a szokásos C++ standard library algoritmusok, amelyekkel egy bemenő tömbön 1 dimenziós konvolúciót számol, és az eredményt visszaírja egy kimeneti tömbbe (az eredmény a bemenet mínusz az ablakszélesség + 1 darab elem méretű).

Például ennek a kódnak működnie kell:
A Variáció:

A template paraméterben megadható neki n darab egész szám (a konvolúció súlyai),

```
std::vector<double> data{1., 2., 3., 4.};
std::vector<double> result(2);
conv1d<1, -2, 1>(data.cbegin(), data.cend(), result.begin());
```

B Variáció:

A template paraméterben megadható neki 1 darab egész szám, az ablakméret, és ekkor a konvolúció súlyai egy másik tárolóból jönnek:

```
std::vector<double> data{1., 2., 3., 4.};
std::vector<double> weights{1., -2., 1.};
std::vector<double> result(2);
conv1d<3>(data.cbegin(), data.cend(), weights.cbegin(), result.begin());
```

Mindkét variációban C++ szálakkal párhuzamosítsuk a műveletet, hasonlítsuk össze a futásidőt a naiv esettel.

2. 2D csúszóablak transzformációs algoritmus írása [nagyon könnyű]

Írjunk olyan algoritmust, amely bemenetéül vár egy két dimenzióként értelmezett tárolót (pl. std::vector) egy fordítás időben ismert értéket, az ablakszélességet a két dimenzióban (w_1, w_2), továbbá egy lambda függvényt, amely a kétdimenziós tároló egy $w_1 \times w_2$ méretű darabját kapja meg és egyetlen értéket ad vissza, amelyet az algoritmus egy másik tárolóba ír bele. Például az alábbi kód minden 3x4-as ablakra kiszámolja az ablak elemeinek összegét:

```
size_t n, m;
std::vector<double> input(n*m);
std::vector<double> output(n*m);
sliding_window_2d<3, 4>(input, output, n, m, [](std::array<double, 3*4> const& elems )
{
   return std::accumulate(elems.begin(), elems.end(), 0.0);
});
```

Azért, hogy az elemszám stimmeljen, periodikus határfeltételt használjunk!

3. Adatsor statisztikai vizsgálata [nagyon könnyű]

Olvassunk be egy adatsort tartalmazó fájlt, és számítsuk ki rajta a következőket, lehetőleg standard library algoritmusok segítségével, ahol lehet:

Átlag, Szórás, Ferdeség, Kurtózis, Medián, a percentilisek (mind a száz).

C++ szálakkal párhuzamosítsuk a műveletet, hasonlítsuk össze a futásidőt a naiv esettel.

4. Kártya keverés vizsgálata [könnyű]

Definiáljunk egy típust, ami képes reprezentálni egy kártyalap tulajdonságait egy kiválasztott fajta pakliból. (a kártya színeire pl. használhatunk enumokat). Definiáljunk egy rendezést a kártya tulajdonságai felett (pl.: először szín szerint, azon belül érték szerint), és hozzunk létre egy pakli kártyát és rendezzük eszerint a reláció szerint.

Definiáljunk egy keverő függvényt (**VARIÁCIÓ**: több ember is választhatja ezt a feladatot, ha szignifikánsan más keverési eljárást valósít meg), amely például a következő kézi keverési stílust valósítja meg:

- Minden keverési lépésben kiválasztunk egy random (normál eloszlás, paraméter az átlag és a szórás) résztartományt a teljes paklából (a tartomány nem lehet nagyobb, mint a pakli), a tartományt kivágjuk, és a pakli végére áthelyezzük (használjunk STL algoritmusokat!)
- Ezt a keverési lépést megismételjük n alkalommal.

Vizsgáljuk meg, hogy kiválasztott „érdekes" mintázatok (pl. 5 egyforma szín egymás után) milyen gyorsan (n függvényében) érik el teljesen random pakliban mérhető előfordulási valószínűségeiket a keverés paramétereinek függvényében. Használjunk standard algoritmusokat a referencia keverésre (shuffle, de NE a deprecated változatokat!), és amire csak lehet.

Csináljunk egy olyan változatot is, ahol a paklit szétszedjük *k* részre, és külön keverjük (külön szálakon), majd egyesítjük, és még *m* keverést végzünk. Hogyan változnak az eredmények, ha összehasonlítjuk az n keveréses változatot a megfelelő $n = l(k) + m(k)$ esettel?

5. Tenzor összeejtések [közepesen nehéz]

Adott egy kiszámolandó formula:

A. VARIÁCIÓ:

$$b_{ijk} = \sum_x \sum_y \sum_z a_{xyz}\, M^{(1)}_{ix} M^{(2)}_{jy} M^{(3)}_{kz}$$

B. VARIÁCIÓ:

$$b_{jk} = \sum_y \sum_z a_{jyz}\, c_{kyz}$$

C. VARIÁCIÓ:

$$b_{xyz} = \sum_k \sum_l a_{xyk}\, M_{kl} c_{yzl}$$

A tömböket -1, 1 közötti véletlen számokkal inicializáljuk. Az optimalizáció során minden olyan átrendezés megengedett, amely a végeredményt elvileg nem változtatja meg, tehát pl.: összeadás, szorzások sorrendjének megváltoztatása, memória elrendezés (különös tekintettel az indexek tárolási sorrendjére).

Az optimalizált eseteket mindig hasonlítsuk össze egy naív referencia implementációval, hogy az eredmény helyességéről meggyőződjünk (adott tolerancián belül).

X. Variáció: Próbáljuk meg több féle csoportosítással (blokkosítással) elvégezni a műveletet, hogy cache barátabb legyen és hasonlítsuk össze a futásidőket a naív és a blokkosított esetben,

Y. Variáció: C++ szálakkal párhuzamosítsuk a műveletet, hasonlítsuk össze a futásidőt a naiv esettel.

Z. Variáció: GPU Párhuzamosítsuk a feladatot

(itt a variációk értelem szerűen direkt szorzatot képeznek, egyik ember választhatja a **BY** párosítást, a másik az **AZ**-t, stb.)

6. Gyors konvolúció (link) [közepesen nehéz]

Valósítsuk meg a linkelt cikkben bemutatott Winograd-féle gyors konvolúciót a következő paraméterek esetére:

A bemenő kép I 512 x 512 méretű, 32 csatornával, a szűrők F 3 x 3 méretűek 32 csatornával, és 128 darab van belőlük, és a kimenő kép O 510 x 510 méretű és minden szűrőhöz tartalmaz egy értéket, azaz 128 eleme van minden pontban. Tehát a kiszámolandó formula:

$$O_{x,\ y,\ n} = \sum_{c} \sum_{a} \sum_{b} I_{x+a,\ y+b,\ c} \cdot F_{x+a,\ y+b,\ c,\ n}$$

A tömböket -1, 1 közötti véletlen számokkal inicializáljuk. Az optimalizáció során minden olyan átrendezés megengedett, amely a végeredményt elvileg nem változtatja meg, tehát pl.: összeadás, szorzások sorrendjének megváltoztatása, memória elrendezés (különös tekintettel az idexek tárolási sorrendjére).

Az optimalizált eseteket mindig hasonlítsuk össze egy naív referencia implementációval, hogy az eredmény helyességéről meggyőződjünk (adott tolerancián belül).

X. Variáció: Próbáljuk meg több féle csoportosítással (blokkosítással) elvégezni a műveletet, hogy cache barátabb legyen és hasonlítsuk össze a futásidőket a naív és a blokkosított esetben,

Y. Variáció: C++ szálakkal párhuzamosítsuk a műveletet, hasonlítsuk össze a futásidőt a naiv esettel.

Z. Variáció: GPU Párhuzamosítsuk a feladatot

7. Fáljok méreteinek statisztikai elemzése [könnyű]

Figyelem, ezt a feladatot csak akkor válasszuk, ha a fordítónk már támogatja az std::filesystem könyvtárat!

http://en.cppreference.com/w/cpp/filesystem

A C++17 std::filesystem alrendszerének segítségével lehetőség nyílik elegánsan, iterátorokkal bejárni a fájlrendszert. Írjunk olyan programot (osztályt), amely kap egy könyvtárra mutató elérési utat, és lefelé bejárja a mappaszerkezetet, kigyűjti a fájlokat és mappákat, és statisztikai lekérdezéseket tesz lehetővé a kapott adatokon (ne kelljen újra bejárni ezekhez a mappákat!). Lekérdezések pl.: fájlok méretének eloszlása, mappák elemszámának eloszlása, melyik (elérési út!) az *n* legnagyobb fálj, melyik az *n* legnagyobb mappa, v legtöbb elemű mappa, mekkora az átlagos / medián fájlméret, stb.

Ahol lehet, használjunk algoritmusokat!

8. Blokkos lineáris tároló implementálása [közepesen nehéz]

A láncolt lista (std::list) és a vektor (std::vector) két végletként tekinthető a lineáris tárolás szempontjából, ugyanis el lehet képzelni egy olyan tárolót, amely láncolt blokkokat tárol, ahol minden blokk *b* darab elemet tárol folytonosan a memóriában (mint a vektor), és a blokkok pointerekkel mutatnak egymásra (mint a listában). Ha *b=1*, akkor visszakapjuk a listát, ha *b=n,* ahol *n* az elemszám, akkor a tároló ugyan az, mint a vektor. Implementáljuk ezt a tárolót, b legyen template paraméter, és írjuk meg rá a legszükségesebb konstruktorokat, tagfüggvényeket, begin, end metódusokat.

Hasonlítsuk össze különböző reprezentatív *b*-k esetére a következő eljárások átlagos idejeit (az időméréshez az std::chrono szolgáltatásait használjuk!):

- Véletlenszerű indexelés
- Elemek beszúrása a tároló elejére
- Elemek beszúrása a tároló végére
- n egymásutáni elem törlése a tároló random pozíciójából

9. String pooling egy kis extrával [haladó]

String poolingnak nevezzük azt, amikor nem azt csináljuk, mint az std::string, hogy minden példánynak foglalunk egy darab memóriát a heap-en és abban tárolunk karaktereket, hanem van egy globális tároló objektum (a string pool), amely minden a programban éppen használt string-et tárol csak 1x, egy fa struktúrában, és minden string objektum csak egy index / pointer ebbe a tárolóba. Ennek a tárolásnak az előnye, hogy a string összehasonlítás konstans idejű: ha két ilyen stringünk ugyan oda mutat a poolban, akkor biztosan azonosak, és ezt csupán az indexük / pointerük összehasonlításával el lehet dönteni. Cserébe

azonban, új string létrehozása, vagy meglevő módosítása költséges (meg kell keresni a helyét a poolban, és ott létrehozni), továbbá nem thread safe! (shared mutable state!).

Implementáljuk ezt a tárolót, hogy ugyan azokat a szolgáltatásokat adja, mint az std::string, és legyen szálbiztos, azaz race-condition mentesen biztosítsa, hogy a pool módosítását egyszerre csak 1 szál végezhesse.

Az extra pedig: egészítsük ki a tárolás logikáját azzal, hogy két string konkatenációját (egymás után fűzését) új string letárolása nélkül reprezentálja: tehát ha „ab” és „cd” már szerepel a poolban, de „abcd” még nem, akkor „ab” + „cd” ne hozzon létre új bejegyzést, csak a két forrás string azonosítóját tárolja. Ha egy ilyen módon létrehozott stringhez egy újabbat fűznénk hozzá, akkor az argumentumokat berakja a poolba, tehát mindig csak a legkülső konkatenáció marad „elvégzetlenül”.

10. Mértékegységek és prefixumok követése [könnyű]

Csináljunk egy olyan osztályt, amely adott T típust tárol, de követi azt is, hogy 3 kiválasztott SI alapmennyiség (pl. hosszúság, idő, tömeg) milyen kombinációját jelenti. Ezt pedig úgy éri el, hogy tárolja a három mennyiség kitevőjét. Tehát pl. a sebesség reprezentálása:

```
template<typename T, int L, int T, int M> class ValueWithDimension;

template<typename T> using Velocity = ValueWithDimension<T, 1, -1, 0>;
```

Írjuk meg erre az osztályra az alapműveleteket, úgy, hogy összeadni, kivonni csak olyan mennyiségeket lehessen, amelyek azonos mértékegységűek! Szorzásnál, osztásnál pedig megfelelően transzformálódjanak a kitevők.

Terjesszük ki az osztályt azzal a funkcionalitással, hogy az ismert SI prefixumokat is tárolja és követi a mennyiségekre, és a műveleteknél igyekszik a legjobb köztes prefixumot választani (pl. 1km + 1mm -> 1000.001 m).

11. Gauss elimináció (link) [közepesen nehéz]

A bemenet egy N x N-es nagymátrix és egy N hosszú vektor két független file-ban. Beolvasás után oldjuk meg az egyenlet rendszert, és írjuk ki a megoldás vektort egy új file-ba. A teljesen naiv implementáció mellett valósítsuk meg C++ szálakkal a párhuzamos implementációt és hasonlítsuk össze a futásidőket.

12. 2D-s kép hisztogramm kiegyenlítése (link) [könnyű]

A bemenet egy 2D-s adattábla (0 és 255 közötti értékekkel), aminek ki kell számolni a hisztogramját, majd a linken látható módon korrigálni az értékeket és visszaírni fileba az új képet (textfile).

A Variáció: több szálon oldjuk meg

B Variáció: GPU párhuzamosan

13. 2D-s korrelációs függvény számolása [könnyű]

Számítsuk ki egy 2D-s adattáblán (D) a következő korrelációs függvény közelítő értékét:

$$C(\vec{p},\vec{q}) = \langle D(\vec{p}) \cdot D(\vec{q}) \rangle, \qquad \text{ahol } D(\vec{p}) \geq t \text{ és } D(\vec{q}) \geq t,$$

ahol t egy felhasználó által adott küszöb. Ehhez lényegében 1D-s hisztogramot kell készítenünk (amit $d = |\vec{p} - \vec{q}|$), amibe minden lehetséges pixel párra növelünk egy számlálót (darabszám), továbbá megvizsgáljuk a feltételeket, és ha teljesülnek, akkor 1-el növeljük a hisztogramban tárolt értéket, végül a binekben felalmozott értéket leosztjuk a vizsgált párok darabszámával.

X. Variáció: Próbáljuk meg több féle csoportosítással (blokkosítással) elvégezni a műveletet, hogy cache barátabb legyen és hasonlítsuk össze a futásidőket a naív és a blokkosított esetben,

Y. Variáció: C++ szálakkal párhuzamosítsuk a műveletet, hasonlítsuk össze a futásidőt a naiv esettel.

Z. Variáció: GPU Párhuzamosítsuk a feladatot

14. Egyenes keresés 2D-s Hough transzformációval, grafikus gyorsítással (link) [haladó]

Használjunk OpenGL-t, vagy DirectX-et arra, hogy kissé átlátszó egyenesek húzásával a framebufferben, megvalósítsuk a linken leírt Hough transzformációt, és ezzel zajos képen egyeneseket keressünk meg.

Olvassunk be egy képet textúraként VAGY generáljunk egy olyan textúrát, amelyen néhány ismert paraméterezésű egyenes van zajjal terhelten. Minden pixelhez a fényességének megfelelő (de leskálázott) átlátszósággal húzzunk egy egyenest egy másik, kezdetben fekete textúrára. A rajzolás végeztével olvassuk vissza a textúrát és keressük meg a felhasználó által adott küszöb feletti pixeleket a transzformált képen, és írjuk ki egy fáljba.

15. 2D-s Klaszterezés [közepesen nehéz]

Adott egy 2D-s adattábla, lebegőpontos értékekkel, amelyben klasztereket kell keresni, azaz azon pixeleknek az összefüggő (legközelebbi 4 szomszédot tekintve) halmazát, amelyek értéke egy megadott értéktől (x) csak egy megadott hibával (dx) tér el. A számítást párhuzamosítsuk a lehető legjobban, a klasztereket írjuk ki egy kimeneti fájlba.

X. Variáció: C++ szálakkal párhuzamosítsuk a műveletet, hasonlítsuk össze a futásidőt a naiv esettel.

Y. Variáció: GPU Párhuzamosítsuk a feladatot

16. Egyenesek illesztése párhuzamos bootstrap módszerrel (link, link) [könnyű]

A bemeneti fájlból olvassuk be a ponthalmazunkat, ami x-y koordináta párok sorozata. Valósítsuk meg az egyenes illesztést a linken szereplő formulákkal. Határozzuk meg az egyenes paramétereinek hibáját bootstrap módszerrel (második link), azaz: véletlenszerűen hagyjunk el pontokat és ismételjük meg az illesztést, mindezt egyszerre több szálon végezve. A paraméterekre kapott értékeket hisztogramozzuk és írjuk ki egy eredmény fileba.

17. Háromszög raszterizáció (link) [könnyű]

Egy bemeneti fileból olvassunk be háromszög csúcspontokat +1 lebegőpontos értékkel csúcsonként (1 sor 3 x 3 adat) és vegyünk fel egy 2D-s rácsot, ami tartalmazza az összes háromszöget. A háromszögekre számoljuk ki a bennfoglaló téglalapot a rácson (a legkisebb téglalap, ami a teljes háromszöget tartalmazza), és annak minden pontjára végezzük el a következőt: számoljuk át a pontot baricentrikus koordinátákra (l. link), döntsük el, hogy belül van-e a háromszögben, és ha igen, akkor interpoláljuk az extra paramétert és írjuk be a rács megfelelő cellájába. Próbáljunk ki többféle elrendezést a ciklusok kiszámolására, főleg nagy méretű háromszögek esetére. Végül a rácsot írjuk ki egy fileba.

18. 2D kontúrvonalak kiszámolása Marching Squares algoritmussal (link). [közepesen nehéz]

A bemenet egy 2D-s adattábla, és egy másik fileban értékek sorozata, amik kontúr szinteket jelentenek. Több szál segítségével minden egyes kontúr szinthez számítsuk ki az összes kontúrvonalat az adattáblából és írjuk ki egy file-ba (azaz: a file struktúrája: minden sor egy kontúrvonal, ahol az első érték, hogy milyen szinthez tartozik, utána pedig 2D-s koordinátapárok listája szerepel).

X. Variáció: C++ szálakkal párhuzamosítsuk a műveletet, hasonlítsuk össze a futásidőt a naiv esettel.

Y. Variáció: GPU Párhuzamosítsuk a feladatot

19. 2D sugárkövetés [haladó]

Számoljunk szivárványt 2D-ben: egy párhuzamos sugárnyaláb érkezik egy körre, ahol a sugarakat az irányvektoruk és a λ hullámhosszuk jellemzi. A sugarak egy n_1 törésmutatójúközegben terjednek, a kör belseje viszont n_2 a törésmutatójú. A sugarak a Snellius-Descartes törvény szerint törnek, illetve teljes visszaverődést szenvednek a körön. Addig kövessük a sugarakat, amíg el nem hagyják a kört, és ekkor készítsünk 2D hisztogramot a hullámhossz-irányszög szerint. Az RGB értékeket a hullámhosszból a Color Matching Function-ökkel való átfedésből számítsuk:

#wavelength, r, g, b

360, 0.000129900000, 0.000003917000, 0.000606100000

365, 0.000232100000, 0.000006965000, 0.001086000000

370, 0.000414900000, 0.000012390000, 0.001946000000

375, 0.000741600000, 0.000022020000, 0.003486000000

380, 0.001368000000, 0.000039000000, 0.006450001000

385, 0.002236000000, 0.000064000000, 0.010549990000

390, 0.004243000000, 0.000120000000, 0.020050010000

395, 0.007650000000, 0.000217000000, 0.036210000000

400, 0.014310000000, 0.000396000000, 0.067850010000

405, 0.023190000000, 0.000640000000, 0.110200000000

410, 0.043510000000, 0.001210000000, 0.207400000000

415, 0.077630000000, 0.002180000000, 0.371300000000

420, 0.134380000000, 0.004000000000, 0.645600000000

425, 0.214770000000, 0.007300000000, 1.039050100000

430, 0.283900000000, 0.011600000000, 1.385600000000

435, 0.328500000000, 0.016840000000, 1.622960000000

440, 0.348280000000, 0.023000000000, 1.747060000000

445, 0.348060000000, 0.029800000000, 1.782600000000

450, 0.336200000000, 0.038000000000, 1.772110000000

455, 0.318700000000, 0.048000000000, 1.744100000000

460, 0.290800000000, 0.060000000000, 1.669200000000

465, 0.251100000000, 0.073900000000, 1.528100000000

470, 0.195360000000, 0.090980000000, 1.287640000000

475, 0.142100000000, 0.112600000000, 1.041900000000

480, 0.095640000000, 0.139020000000, 0.812950100000

485, 0.057950010000, 0.169300000000, 0.616200000000

490, 0.032010000000, 0.208020000000, 0.465180000000

495, 0.014700000000, 0.258600000000, 0.353300000000

500, 0.004900000000, 0.323000000000, 0.272000000000

505, 0.002400000000, 0.407300000000, 0.212300000000

510, 0.009300000000, 0.503000000000, 0.158200000000

515, 0.029100000000, 0.608200000000, 0.111700000000

520, 0.063270000000, 0.710000000000, 0.078249990000

525, 0.109600000000, 0.793200000000, 0.057250010000

530, 0.165500000000, 0.862000000000, 0.042160000000

535, 0.225749900000, 0.914850100000, 0.029840000000

540, 0.290400000000, 0.954000000000, 0.020300000000

545, 0.359700000000, 0.980300000000, 0.013400000000

550, 0.433449900000, 0.994950100000, 0.008749999000

555, 0.512050100000, 1.000000000000, 0.005749999000

560, 0.594500000000, 0.995000000000, 0.003900000000

565, 0.678400000000, 0.978600000000, 0.002749999000

570, 0.762100000000, 0.952000000000, 0.002100000000

575, 0.842500000000, 0.915400000000, 0.001800000000

580, 0.916300000000, 0.870000000000, 0.001650001000

585, 0.978600000000, 0.816300000000, 0.001400000000

590, 1.026300000000, 0.757000000000, 0.001100000000

595, 1.056700000000, 0.694900000000, 0.001000000000

600, 1.062200000000, 0.631000000000, 0.000800000000

605, 1.045600000000, 0.566800000000, 0.000600000000

610, 1.002600000000, 0.503000000000, 0.000340000000

615, 0.938400000000, 0.441200000000, 0.000240000000

620, 0.854449900000, 0.381000000000, 0.000190000000

625, 0.751400000000, 0.321000000000, 0.000100000000

630, 0.642400000000, 0.265000000000, 0.000049999990

635, 0.541900000000, 0.217000000000, 0.000030000000

640, 0.447900000000, 0.175000000000, 0.000020000000

645, 0.360800000000, 0.138200000000, 0.000010000000

650, 0.283500000000, 0.107000000000, 0.000000000000

655, 0.218700000000, 0.081600000000, 0.000000000000

660, 0.164900000000, 0.061000000000, 0.000000000000

665, 0.121200000000, 0.044580000000, 0.000000000000

670, 0.087400000000, 0.032000000000, 0.000000000000

675, 0.063600000000, 0.023200000000, 0.000000000000

680, 0.046770000000, 0.017000000000, 0.000000000000

685, 0.032900000000, 0.011920000000, 0.000000000000

690, 0.022700000000, 0.008210000000, 0.000000000000

695, 0.015840000000, 0.005723000000, 0.000000000000

700, 0.011359160000, 0.004102000000, 0.000000000000

705, 0.008110916000, 0.002929000000, 0.000000000000

710, 0.005790346000, 0.002091000000, 0.000000000000

715, 0.004109457000, 0.001484000000, 0.000000000000

720, 0.002899327000, 0.001047000000, 0.000000000000

725, 0.002049190000, 0.000740000000, 0.000000000000

730, 0.001439971000, 0.000520000000, 0.000000000000

735, 0.000999949300, 0.000361100000, 0.000000000000

740, 0.000690078600, 0.000249200000, 0.000000000000

745, 0.000476021300, 0.000171900000, 0.000000000000

750, 0.000332301100, 0.000120000000, 0.000000000000

755, 0.000234826100, 0.000084800000, 0.000000000000

760, 0.000166150500, 0.000060000000, 0.000000000000

765, 0.000117413000, 0.000042400000, 0.000000000000

770, 0.000083075270, 0.000030000000, 0.000000000000

775, 0.000058706520, 0.000021200000, 0.000000000000

780, 0.000041509940, 0.000014990000, 0.000000000000

785, 0.000029353260, 0.000010600000, 0.000000000000

790, 0.000020673830, 0.000007465700, 0.000000000000

795, 0.000014559770, 0.000005257800, 0.000000000000

800, 0.000010253980, 0.000003702900, 0.000000000000

805, 0.000007221456, 0.000002607800, 0.000000000000

810, 0.000005085868, 0.000001836600, 0.000000000000

815, 0.000003581652, 0.000001293400, 0.000000000000

820, 0.000002522525, 0.000000910930, 0.000000000000

825, 0.000001776509, 0.000000641530, 0.000000000000

830, 0.000001251141, 0.000000451810, 0.000000000000

X. Variáció: C++ szálakkal párhuzamosítsuk a műveletet, hasonlítsuk össze a futásidőt a naiv esettel.

Y. Variáció: GPU Párhuzamosítsuk a feladatot

20. Háromszögek és egyenesek metszéspontjainak kiszámolása [könnyű]

A bemenet 2 file, az egyik háromszögeknek a 3D-s koordinátáit tartalmazza (3 csúcspont xyz), a másik egyeneseknek a 3D-s koordinátáit (xyz kiindulási pont és xyz irányvektor, ami nem feltétlen normált). A feladat: minden egyeneshez meghatározni a kiindulási ponttól az irányvektor irányában legközelebbi háromszöggel vett metszéspontot és kiírni a kimeneti file-ba (egyenes kezdőpontja és a megtalált metszéspont koordinátái soronként).

X. Variáció: C++ szálakkal párhuzamosítsuk a műveletet, hasonlítsuk össze a futásidőt a naiv esettel.

Y. Variáció: GPU Párhuzamosítsuk a feladatot

21. 2D Ising modell szimuláció Metropolis algoritmussal (link, link) [közepesen nehéz]

Vegyünk fel egy 2D-s rácsot periodikus határfeltétellel, aminek minden pontjában spinek vannak (s_i = +1, -1 állapot), a kölcsönhatás csak a 8 legközelebbi szomszéddal van, erőssége J, a spinek mágneses momentuma μ, a külső mágneses tér erőssége H. Léptessük a rácsot különböző hőmérsékletek esetén Metropolis algoritmussal, végül ábrázoljuk a hőmérséklet függvényében a mágnesezettséget: $M = \mu \sum_i s_i$. A kapott

görbét írjuk ki egy file-ba. Hasonlítsuk össze a számolás sebességét különböző blokkosítási sémák, illetve bejárások esetén.

22. Cahn-Hilliard egyenlet szimulációja (link) [közepesen nehéz]

Vegyünk fel egy 2D-s periodikus rácsot ($N \times N$ cella), minden cellában egy random $[-1.0; +1.0]$ közötti lebegőpontos értékkel. Véges differencia módszerrel, több szálon, kis lépésközzel léptessük a rácsot, és minden lépésben közelítsük a kétpont korrelációs függvényt random mintával: $C(r) = < P, Q >$, ahol P és Q két rácspont, aminek a távolsága r ($r = 0 .. N$). Határozzuk meg a függvény közelítő levágási helyét minden időpillanatban és írassuk ki file-ba az eredményt időlépésenként.

23. Monte-Carlo integrálás (link) [nagyon könnyű]

Írjunk integrátort, ami az alábbi függvénnyel hívható, és többszálra szétosztva kiszámolja a 3D-s integrál közelítő értékét. Az első argumentum az integrandus, a második az integrálási tartomány (igazat ad vissza, ha a pont benne van a tartományban), az utolsó 6 az integrálási tartomány bennfoglaló téglalapja.

```
MonteCarlo( [](auto x, auto y, auto z){ return exp(-x*x-y*y-z*z); },
            [](auto x, auto y, auto z)->bool{ return x*x+y*y+z*z<16.0; },
             -4.0, 4.0, -4.0, 4.0, -4.0, 4.0);
```

C++ szálakkal párhuzamosítsuk az integrátort.

24. Térpartícionált N-test szimuláció [haladó]

Írjuk meg az órán látott N-test szimuláció térpartícionált változatát, azaz osszuk fel a szimulációs teret kisebb blokkokra, amiken belül (illetve a legközelebbi szomszédos blokkok részecskéivel) egzakt a kölcsönhatás, de a távolabbi blokkok csak a tömegközéppontjaikon keresztül hatnak kölcsön. Amit lehet, számoljunk több szálon!

25. Constructive Solid Geometry (link) [könnyű]

Írjunk programot, amelyben a felhasználó képes pár különböző méretű és elhelyezkedésű CSG primitív (pl. kocka, hasáb, gömb, gúla) unióinak és metszeteinek tetszőleges kombinációját létrehozni, egy fában tárolni, majd egy bemeneti térpontról eldönteni, hogy kívül, vagy belül esik-e az összerakott alakzaton. Az alakzat építésekor keletkezett fa struktúra bejárásakor indíthatunk új szálakat a párhuzamosítás megvalósítására.

26. Minimális befoglaló térfogat megkeresése [könnyű]

Írjunk egy függvényt, amely képes egy 3D-s csúcspont halmazról megmondani, hogy egy bemeneti geometriai primitív (gömb, kocka, hasáb, gúla) mely paraméterezése képes minimális térfogattal befoglalni az adott csúcspont halmazt befoglalni. Legyen képes továbbá az adott alakzatok paramétereinek listáját/tömbjét visszaadni, ami térfogat szerint csökkenő sorrendbe van rendezve. Párhuzamosítsuk a számolást.