IMSc önállólabor-pályázat

jelentkezési lap

Név:	**Jost Márk Benedek**
Neptun-kód:	**JXOV52**
Email:	**jost.bence@gmail.com**
Szak:	**Villamosmérnök BSc.**
Specializáció, ágazat/tanszék (ahol a munka készült):	**AUT**
A téma címe:	**Algoritmusok optimalizálása GPU-n**
A téma rövid összefoglalója: **A modern számítástechnikában a hardveres innováció terén egyre inkább háttérbe szorul az órajelfrekvencia növelése, helyette párhuzamosítással lehet nagyobb utasításáteresztő-képességet elérni. Az NVIDIA CUDA keretrendszere segítségével olyan algoritmusokat vizsgáltam, amelyek alkalmasak nagyvolumenű párhuzamos végrehajtásra, ezáltal kihasználva egy speciális célhardver (NVIDIA RTX 3050TI) nyújtotta lehetőségeket.** **Az önálló labor során főleg útvonaltervezési problémákkal (TSP, VRP, CVRP) foglalkoztam, amelyek NP-teljesek, ezért Hangyakolónia algoritmussal közelítettem a futást.**	

Kijelentem, hogy a 2022/23. tanév 2. félévében az IMSc program résztvevőjeként teljesítettem az alapképzés Önálló laboratórium tantárgyát.

Kelt: 2023. 06. 30.

Jost Márk Benedek

……

Aláírás

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

Algoritmusok Optimalizálása GPU-n

BSc Önálló laboratórium

Készítette
Jost Márk Benedek

Konzulens
dr. Szegletes Luca

2023. május 31.

Tartalomjegyzék

1. Összefoglaló

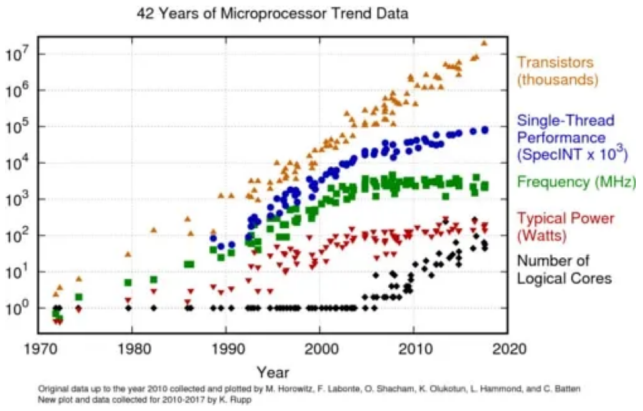
Önálló laboratóriumi tanulmányaim célja az volt, hogy különböző, nagy jelentőséggel bíró algoritmusok futását gyorsítsam videókártyán párhuzamosítás segítségével. Vizsgálódásom során először polinomidejű algoritmusokat, majd bizonyítottan NP-teljes problémákat valósítottam meg, közben kipróbáltam több elviekben különböző ötletet is arra, hogy a gyorsítás ne menjen a végeredmény helyességének kárára.

2. Felhasznált irodalom

Munkám során segítségül használtam különféle irodalmakat is. Az Nvidia CUDA keretrendszer hivatalos dokumentációját követtem a programozási modell alkalmazásához, mely interneten is elérhető bárki számára angol nyelven. Aktívan tanulmányoztam az Nvidia hivatalos blogját, melyet mivel hivatásos Nvidia fejlesztők is szerkesztenek, relevánsnak tekintettem. Továbbá segítségül használtam Jason Sanders és Edward Kandrot urak "CUDA by Example: An Introduction to General-Purpose GPU Programming" című művét.

3. Motiváció

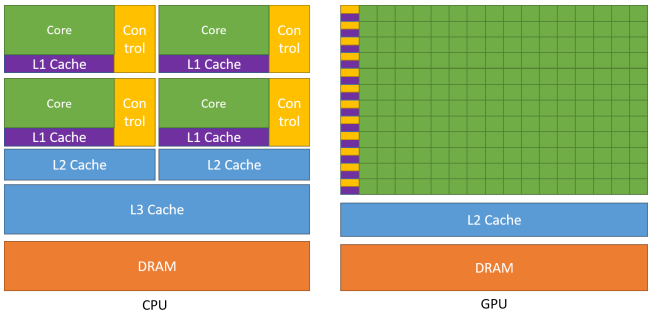
Az 1980-as években megjelentek az első személyi számítógépek (PC-k), melyek központi feldolgozóegységei (CPU) mintegy 1MHz-es belső órajellel működtek. Akkor az volt a számítástechnikai fejlesztők fő célja, hogy az órajelfrekvencia emelésével növeljék a számítási gyorsaságot. Számos kiváló mérnöki megoldás született, talán az egyik legjelentősebb a fáziszárt hurok, ami nélkül képtelenség lett volna 50-60 MHz fölé menni. Nagyjából 30 évvel később elérték, hogy a legtöbb asztali processzor órajele 1GHz és 4GHz között legyen, ami az eredeti PC-k frekvenciájának több ezerszerese. Napjainkban változás látható a fejlesztési trendekben, ugyanis az órajelnövelést a processzorok disszipációja felülről korlátozza. Egyelőre nem tűnik könnyen lehetségesnek 5GHz fölé menni úgy, hogy közben az eszköz helyes működése garantálható legyen. A számítógépgyártók éppen ezért új, alternatív megoldásokat kerestek a számítási teljesítmény növelésére. Legjobb ötletnek a feladatok párhuzamosítása bizonyult. Napjainkban a kutatásoknak két nagy témája van. Egyik a kvantumszámítószépek témája, amit dolgozatomban nem részletezek. A másik a párhuzamosítás. Már a CPU-k fejlesztésénél is megfigyelhető, hogy inkább a minél több processzormag telepítése az iparági trend.



1. ábra. Látható, hogy kb. 2010-re befejeződött a CPU-k órajelfrekvencia-növekedése, helyette egyre nőni kezdtek a magok számai. [1]

A párhuzamosításra azonban sokkal alkalmasabb a grafikus segédprocesszor, a GPU. Amíg a CPU feladata, hogy műveletek egy adott szekvenciáját, és annak minden utasítását a lehető leggyorsabban hajtsa végre, addig a GPU célja minél több szál (akár több ezer) párhuzamos futtatása. A videókártyák előnye akkor válik láthatóvá, ha ugyanazt az utasítást több, nagy adattömbön kell végrehajtani. Ez az úgynevezett SIMD megközelítés (Single Instruction Multiple Data). [9] Az []. ábra szemlélteti, hogy

a GPU-n arányában több tranzisztor van adatfeldolgozásra rendelvelve, cserébe a gyorsítótárazás és a folyamatvezérlés kisebb hangsúlyt kapott.



2. ábra. Látható, hogy a gyorsítótárak és a vezérlés rovására nőtt az adatfeldolgozásra szánt tranzisztorok számára. Ez alkalmas lebegőpontos műveletek nagyfokú párhuzamosítására. [4]

A videókártya sokkal nagyobb utasítás-áteresztőképességet, valamint memória-sávszélességet biztosít, mint a CPU hasonló ár és energiafogyasztás mellett. Egyéb számítási eszközök, például FPGA-k is lehetnek nagyon energiatakarékosak, viszont azok sokkal kevésbé rugalmasan programozhatóak a GPU-knál, és a fejlesztési idő is sokkal hosszabb lesz. [4]

4. Technológiai háttér

Ebben a fejezetben szeretném ismertetni az általam felhasznált szoftver és hardver együttest. Először a grafikus segédprocesszoron történő általános célú programozást tárgyalom, majd bemutatom az Nvidia™ által erre kifejlesztett párhuzamos számítási platformot, a CUDA keretrendszert. Részletezem a megértéshez szükséges fontosabb fogalmakat, valamint bemutatom a CUDA platformon fejlesztéshez elengedhetetlen lépéseket.

4.1. 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. [9]

4.2. CUDA

A választásom a CUDA platformra esett. A CUDA (Compute Unified Device Architecture) egy, az NVIDIA által fejlesztett párhuzamos számítási platform, amely szoftveres támogatást nyújt az ezzel kompatibilis grafikus feldolgozóegységek általános célú programozására. A gyártó bővítményt is adott ki, mely a Visual Studio nevű fejlesztői környezetbe importálható. Telepítés után ha új projekt létrehozását választjuk (File/New/Project), akkor "CUDA [verziószám] Runtime" néven kiválasztható a projekt típusának a CUDA. A programozás C vagy C++ nyelven történhet, melyhez minimális nyelvi kiegészítéseket tettek, hogy többek között a szálkezelés egyszerűbbé váljon. A CUDA használatához elérhető egy rendkívül kiterjedt dokumentáció a gyártó weboldalán. [4]

4.3. Programozási modell

A továbbiakban összefoglalom a legfontosabb fogalmakat úgy, hogy ismertetem, hogyan lettek megvalósítva C++ nyelven.

4.3.1. Kernel, és a többi függvénytípus

A programozó speciális függvényeket definiálhat, melyeket kernelnek nevezünk. A kernel létesít kapcsolatot a CPU (host) és GPU (device) között úgy, hogy előbbi meghívja a függvényt, majd átadja

utóbbinak a vezérlést, tehát a kernel a videókártyán fut. Minden egyes kernel példányt egy számára megadott szál hajt végre. A kernel a "__*global*__" kulcsszóval definiálható. Ezt a függvény fejléc elé kell írni, ekkor tudja a szoftverkörnyezet, hogy mostantól GPU-kódként értelmezze. (Megjegyzendő, hogy egyéb kulcsszavak is léteznek. Egyik a "__*host*__", ami jelzi, hogy CPU által hívott, majd ugyanúgy általa végrehajtandó kód következik. Ha nem adunk meg egy függvény elé kulcsszót, akkor azt tiszta CPU kódként értelmezi, úgy, mintha nem is lenne a szoftverkörnyezet mögött a CUDA platform. Másik a "__*device*__", amely tisztán GPU függvényt jelez. A két kulcsszó vegyíthető: amennyiben azt írjuk, hogy "__*global*__ __*host*__", a fordító ezt minden egyes híváskor a végrehajtó saját kódjának tekinti, vagyis nem hajt végre vezérlésátadást. Utóbbi hasznosítható például függvénykönyvtárak GPU-ra kiterjesztésére.

Az, hogy a kernelt egy adott híváskor hány CUDA szálon szeretnénk futtatni, az új nyelvi elemként megjelenő «< · · · »> végrehajtási konfiguráció szintaxissal specifikálható. Sajnos a Visual Studio még szintaxishibaként kezeli [verziószám], ezért a programozó érdemes, hogy odafigyeljen, milyen IntelliSense hibaüzeneket vesz figyelembe. Minden, a kernelt végrehajtó szál egy egyedi thread azonosítót kap, mely a beépített threadIdx változón keresztül érhető el a kernelből.

Példa: A hivatalos dokumentáció az alábbi példát adja kernel definícióra. A kódrészlet az N méretű A és B vektorok összeadását végzi 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);
    ...
}
```

5. Mátrixműveletek vizsgálata

Dolgozatom célja minél jobb módszereket találni olyan algoritmusokra, amik gráfokon futnak. A gráfok egyik legalapvetőbb megadása a szomszédossági mátrixával történő reprezentáció. Ennek előnye, hogy az egyes csúcsokhoz tartozó élek könnyen, direkt lekérdezhetőek, ezért a rajtuk végzett műveletek gyorsabbak lehetnek. Hátránya azonban, hogy minél kevesebb él van a gráfban, annál helypazarlóbbá válik ez a megadás. Később tekintünk egyéb megadási módokat is, de előbb érdemes megvizsgálnunk, adott tárolási mód mellett a futásidő hogyan változik különböző programozási paradigmák esetén.

5.1. Implementációs választási lehetőségek

Input: Szükségünk van inputmátrix megadására. Kezdetleges C programokban ezt úgy tettük meg, hogy a felhasználótól kértük az adatok begépelését konzolba. Ez nem szép megoldás, nagy méret esetén pedig egyszerűen kivitelezhetetlen. Programom egy fájlból olvassa be az adatokat, mely fájl nevét pedig parancssori argumentumban várja. Ennek a C típusú megvalósítása például ilyen módon történhet:

```
// Variables for reading file
FILE* pFile;    // File pointer
int fileNameIdx;
bool foundMatrix = false;    // For error handling
int size;    // Number of matrix rows (=matrix coloumns)
```

```
int i;  // Iterator
for (i = 1; i < argc; ++i)  // Processing command line arguments
{
    // Command Line Syntax: ... --m [file_name]
    if ((strcmp(argv[i], "--m") == 0))
    {
        pFile = fopen(argv[++i], "r");  // Opening file to read
        if (pFile == NULL)  // Unable to open file
        {
            return -1;
        }
        fileNameIdx = i;
        foundMatrix = true;
    }
}
if (!foundMatrix)
{
    fprintf(stderr, "Required Command Line Syntax: --m [data_file]\n");
    return -1;
}

// Read data
// ...

if (fclose(pFile) != 0) // Unable to close file
{
    return -1;
}

// Process data
// ...
```

Adattárolás módja: A mátrixelemeket valahogyan el kell tárolnunk ahhoz, hogy műveleteket végezhessünk rajtuk. Nem mindegy, hogy milyen adattípust kap egy elemet tároló változó. Integer jelen esetben nem alkalmas választás, mert az input lehet nem egész számokból álló mátrix is. Válasszunk lebegőpontos ábrázolást, ebből a két népszerű szabványos példa a float és a double. Ahhoz, hogy a későbbiekben össze tudjuk hasonlítani a két opciót, vezessük be az alábbi define-t:

```
// Define for used data types: now chosen double
#define DATATYPE double
```

Most éppen double-t választottunk használt adattípusnak. Ha változtatni szeretnénk, elég itt, egy helyen átírni a kódot pl. float-ra.

Másik fontos megfontolás az adatstuktúra megválasztása. Statikus tömböt nem használhatunk, mert fordítási időben nem ismert az input mérete. Láncolt listákat device kódban nem támogat jelenleg a CUDA keretrendszer (létezik a Thrust nevű kezdeményezés [7], amely host kódban enged device-on listamemória használatára, de most ennek használatára nem térek ki). Én dinamikus tömbben tárolom a mátrixot. Mátrixelemeket lehet egy- illetve kétdimenziós tömbben is tárolni. Egydimenziós tömb foglalására példakód:

```
// size read from file
DATATYPE* Matrix = (DATATYPE*)calloc(size * size, sizeof(DATATYPE));
```

Blokkszám: Szinkronizáció szempontjából nagyon nem mindegy, hogy hány konkurens blokkban fut programunk. (Lásd grid szinkonizáció, szükséges ide másolni később!). Jelenleg minden mátrixelemre jut egy külön szál, ezért most nincs ráhatásunk a blokkok számára (blokkméretnek érdemes a lehető legnagyobbat választani), más alkalmazások esetén erre figyelnünk kell.

5.2. Determinás számítása

Tekintsünk egy alapvető lineáris algebrai fogalmat, a determinánst. A determinánsszámítás egy művelet, ami egy NxN-es (négyzetes) mátrixhoz egy skalárszámot rendel. Egy mátrix determinánsának

kiszámítására számos módszer létezik, én most Gauss-eliminációt alkalmazok.
Matematikából ismertek a determinás alábbi tulajdonságai (teljesség igénye nélkül):

1. ha egy mátrix egyik sorához (vagy oszlopához) egy másik sor (vagy oszlop) skalárral való szorzatát adjuk, az a determináns értékét nem változtatja
2. két sor (vagy oszlop) felcserélése esetén a determináns a (-1)-szeresére változik
3. felsőháromszög-mátrix determinánsa kiszámolható a főátlóbeli elemek összeszorzásával

Előbbi tulajdonságok alapján ha Gauss-elimináció segítségével felsőháromszög-mátrixszá alakítjuk az eredeti mátrixot, annak determinánsa a főátlóbeli elemek szorzata. Megtörténhet, hogy az elimináció közben az aktuális vezérelem 0. Ekkor ha van az adott oszlopban alatta nemnulla elem, akkor sorcserét végzünk. Ha nincs alatta nemnulla, akkor a determináns nulla.

5.2.1. Device kód

Az alábbi GPU kódhoz illeszthető kiszolgáló CPU kód. A teljes kód megtekinthető a mellékelt github repo-ban (majd később lesz hozzáadva).

Segédfüggvényem 0 vezérelem esetén megvizsgálja, hogy ki lehet-e cserélni az adott sort egy alatta levővel.

```
// Megkeresi a vezérelem alatti első nem nulla elemet az oszlopban
// Visszatérési értéke, hogy talált-e
__device__ bool firstNotZero(DATATYPE* Matrix, int size, int k, int* idx)
{
    int i;
    for (i = k + 1; i < size; ++i) {
        if (Matrix[i * size + k]) {
            *idx = i;
            return true;
        }
    }
    return false;
}
```

Az 1x1-es Gridek esetén ez a kernel hívandó. Látható, hogy több egyszerűsítés is meg van benne téve, ami kihasználja a Grid méretét.

```
// 1x1-es Grid
// Grid: 1x1
__global__ void detKernel_1Block(DATATYPE* Matrix, int size, DATATYPE* det)
{
    cooperative_groups::thread_block block = cooperative_groups::this_thread_block();
    int i = blockIdx.x * blockDim.x + threadIdx.x;  // oszlopváltozó
    int j = blockIdx.y * blockDim.y + threadIdx.y;  // sorváltozó
    if (i >= size || j >= size)
        return;
    int k;
    int idx;    // Sorcserénél használt változó
    DATATYPE temp;

    __shared__ int sign;
    __shared__ bool fullZeroColoumn;

    // Kezdeti érték adás a 0. thread által
    if (i == 0 && j == 0)
    {
        sign = 1;
        fullZeroColoumn = false;
    }

    // Gauss elimináció
    for (k = 0; k < size - 1; ++k)
    {
```

```
        //a már nem kellő szálak kiléphetnek
        if (i < k || j < k)
            return;
        block.sync();

        // Mi van akkor, amikor a vezérelem 0?
        if (Matrix[k * size + k] == 0) {
            // Keresünk másik sort, ahol nem 0 vezérelem van  különben det=0
            // Mindig csak az egyik szálon teszteljük a vezérelem 0 voltát  mert
    megosztott változót állítunk
            if (i == k && j == k)
            {
                fullZeroColoumn = !firstNotZero(Matrix, size, k, &idx);

                if (fullZeroColoumn) {
                    // Csupa 0 oszlopot találtunk
                    *det = 0;
                }
                sign = -sign;
            }

            // Bevárjuk a [k][k] threadet, hogy megfelelőre állítsa a változót
            block.sync();
            // A többi threadet is értesítjük arról  ha kész vagyunk; értéket már nem
    kell állítaniuk
            if (fullZeroColoumn)
                return;

            // Kicseréljük a  k. és idx. sort: ilyenkor a determináns a -1 -szeresére v
    áltozik

            // 1 dimenziós párhuzam, mert vektorművelet
            block.sync();
            if (i == k) {
                temp1 = Matrix[k * size + j];
                temp2 = Matrix[idx * size + j];
                Matrix[k * size + j] = temp2;
                Matrix[idx * size + j] = temp1;
            }
            // Sorcsere után újabb szinkronizáció
            block.sync();
        }

        // Nem nulla a vezérelem, kezdődhet a Gauss elimináció,
        // a k-adik oszlopot felesleges kinullázni, többet nem kellenek
        if (i > k && j > k)
            Matrix[i * size + j] -= Matrix[i * size + k] / Matrix[k * size + k] *
    Matrix[k * size + j];
    }
    // Determináns a főátlóbeli elemek szorzata alapján
    if (i == size - 1 && j == size - 1)
    {
        *det = sign;
        for (k = 0; k < size; ++k)
            *det *= Matrix[k * size + k];
    }
};
```

A több blokkból álló gridek esetében a multiblokk kernel hívandó.
Itt a szinkronizációért a *grid.sync()* API hívás felelős. A függvény használ globális változókat annak érdekében, hogy elérje mindegyik thread blokk. Ez nem feltétlen szerencsés megközelítés, másik lehetséges megoldás az, hogy függvényparaméterként adjuk át őket.

```
// Számoljuk a sorcserék számának paritását
__device__ int global_sign;
// Ha bármikor csupa 0 oszlopot találunk, tudjuk, hogy 0 a determináns
```

```
__device__ bool global_fullZeroColoumn;

// Grid: >1x1
__global__ void detKernel_multiBlock(DATATYPE* Matrix, int size, DATATYPE* det) {
    // Szinkronizációs változó a teljes griden belül
    cooperative_groups::grid_group grid = cooperative_groups::this_grid();
    int i = blockIdx.x * blockDim.x + threadIdx.x;  // oszlopváltozó
    int j = blockIdx.y * blockDim.y + threadIdx.y;  // sorváltozó

    int k;
    int idx;    // Sorcserénél használt változó
    DATATYPE temp1, temp2;
    if (!grid.is_valid())
        return;
    grid.sync();

    // Kezdeti érték adás a 0. thread által
    if (i == 0 && j == 0) {
        global_sign = 1;
        global_fullZeroColoumn = false;
    }

    for (k = 0; k < size - 1; ++k)
    {
        grid.sync();

        // Mi van akkor  amikor a vezérelem 0?
        if (Matrix[k * size + k] == 0)
        {
            // Keresünk másik sort, ahol nem 0 vezérelem van  különben det=0
            // Mindig csak az egyik szálon teszteljük a vezérelem 0 voltát, mert
    megosztott változót állítunk
            if (i == k && j == k) {

                global_fullZeroColoumn = !firstNotZero(Matrix, size, k, &idx);

                if (global_fullZeroColoumn) {
                    // Csupa 0 oszlopot találtunk
                    *det = 0;
                }
                global_sign = -global_sign;
            }
            // Bevárjuk a [k][k] threadet, hogy megfelelőre állítsa a változót
            grid.sync();
            // A többi threadet is értesítjük arról, ha kész vagyunk; értéket már nem
    kell állítaniuk
            if (global_fullZeroColoumn)
                return;

            grid.sync();
            if (i == k) {
                temp1 = Matrix[k * size + j];
                temp2 = Matrix[idx * size + j];
                Matrix[k * size + j] = temp2;
                Matrix[idx * size + j] = temp1;
            }
            // Sorcsere után újabb szinkronizáció
            grid.sync();
        }
        // Nem nulla a vezérelem, kezdődhet a Gauss elimináció
        // a k-adik oszlopot felesleges kinullázni, többet nem kellenek
        if (i > k && j > k && i < size && j < size) // diagnosztika végett nem j>=k
    lehetséges
            Matrix[i * size + j] -= Matrix[i * size + k] / Matrix[k * size + k] *
    Matrix[k * size + j];
    }
```

```
    if (i == size - 1 && j == size - 1)
    {
        *det = global_sign;
        for (k = 0; k < size; ++k)
            *det *= Matrix[k * size + k];
    }
};
```

5.2.2. Tesztelések

A profilozást az NVIDIA Nsight™ Compute programjának segítségével végeztem.
Verziószám: 2023.1.0.0 (build 32451174) (public-release)

A determinánsszámító programomat többféle módon teszteltem: több inputra megnéztem a futásidőt, a pontos eredménytől való százalékos eltérést (referenciának a Wolfram Alpha [8] nevű internetes számológépekkel, illetve Matlabbal ellenőriztem): egydimenziós tárolás float adattípusokon, egydimenziós tárolás double adattípusokon, kétdimenziós tárolás float adattípusokon, és végül kétdimenziós tárolás double adattípusokon. Az eredményeket az alábbi táblázat tartalmazza.

Az eredmény összehasonlítását automatizálhatjuk azzal, hogy a kiegészítő program saját maga hasonlítja össze a GPU által számolt eredményt az elméleti helyes eredménnyel. Én parancssori argumentumba adom meg opcionálisan a helyes eredményt, és ekkor %-ban megadva kiírja a program a relatív hibát. Ennek egy fajta implementálása:

```
bool givenSolution = false;
double solution;
int i;  // Iterator
for (i = 1; i < argc; ++i)  // Processing command line arguments
{
    // Command Line Syntax: ... --s [Solution]
    if ((strcmp(argv[i], "--s") == 0) || (strcmp(argv[i], "--Solution") == 0))
    {
        if (sscanf(argv[++i],"%lf", &solution) != 1) {
            fprintf(stderr, "Unable to read given solution!\n");
        }
        else {
            printf("Given solution : %f\n", solution);
            givenSolution = true;
        }
    }
}

// Process data
// ...

if (givenSolution) {
        double errorPercentage = abs((double)calculatedSolution - solution) / solution
    * 100;
        printf("Difference from correct solution: %.12f %%\n", errorPercentage);
    }
```

A mérések eredményét az alábbiakban táblázatosan tüntettem fel:

5x5-ös egészértékű mátrix					
	Futásidő (µs)	Végeredény	Relatív hiba (µ%)	Compute TP (%)	Occupancy (%)
float, 1D	5.18	-506992	0	0.24	15.63
float, 2D	5.31	-506992	0	0.24	11.31
double, 1D	7.94	-506992	0	0.94	11.14
double, 2D	7.87	-506992	0	1.02	8.46

1. táblázat. 5x5-ös egészértékű mátrix, det = -506992 (Matlab)

10x10-ös valósértékű mátrix					
	Futásidő (µs)	Végeredény	Relatív hiba (µ%)	Compute TP (%)	Occupancy (%)
float, 1D	6.69	-7065577472.00	72.6	0.49	20.5
float, 2D	8.32	-7065577472.00	72.6	0.18	5.99
double, 1D	16.61	-7065572342.25	0	1.86	13.52
double, 2D	17.66	-7065572342.25	0	1.75	12.65

2. táblázat. 10x10-es valósértékű mátrix, det = -7065572342.249096 (Matlab)

18x18-as mátrix					
	Futásidő (µs)	Végeredény	Relatív hiba (µ%)	Compute TP (%)	Occupancy (%)
float, 1D	11.20	-7065577472.00	22980.3	0.74	23.36
float, 2D	11.49	-7065577472.00	22980.3	0.76	24.52
double, 1D	35.46	-7065572342.25	0.0000067	2.93	22.37
double, 2D	35.90	-7065572342.25	0.0000067	2.83	21.89

3. táblázat. 18x18-as véletlenszerű mátrix, det = -6.271444603206407e+19 (Matlab)

5.3. Egy lépéssel tovább: Mátrix inverz számítás

Az előző fejezetben taglalt determinánsnak sok alkalmazása van a lineáris algebrában, az egyik az, hogy felhasználható egy mátrix inverzének meghatározása során. Matematikából ismert képlet:

$A^{-1} = (adjA)/(detA)$, ahol A^{-1} az A mátrix inverze, adj A az adjungáltmátrix, azaz az eredeti mátrix előjeles aldeterminánsaiból alkotott mátrix transzponáltja, det A pedig a mátrix determinánsa.

Én jelen esetben egy másik algoritmust alkalmaztam: az LU dekompozíció alapú mátrixinvertálást. Elvileg gyorsabb algoritmus, cserébe speciális esetekben nem működik (amikor az inputnak nem létezik LU dekompozíciója, csak PLU dekompozíciója, P a permutációmátrix, nem foglalkozunk most vele). Ilyen esetben a program figyelmeztet, hogy nem tud működni. Továbbfejlesztési lehetőség, hogy amikor nem létezik LU felbontás, olyankor egy másik (potenciálisan lassabb) algoritmus fut le.

LU dekompozíció Egy matematikai művelet, mely során egy négyzetes mátrixot felbontunk egy felsőháromszögmátrix (U), valamint egy alsóháromszögmátrix (L) szorzatára.

Az LU dekompozíció a következőképpen alkalmas invertálás számítására: [kifejtés később]

Belátható, hogy az így végzett invertálás $O(n^3)$-ös algoritmust eredményez. Algoritmusom során megpróbálok hol n, hol n^2 számú műveletet végezni, ezáltal gyorsítani a futást. Neminvertálható mátrixoknak is létezhet LU felbontása, ezért az invertáló kernel előtt lefuttatom az előzőekben bemutatott determináns számító kernelt azért, hogy elkerüljünk különböző hibázási lehetőségeket.

5.3.1. Device kód

Az alábbi GPU kódhoz illeszthető kiszolgáló CPU kód. A teljes kód megtekinthető a mellékelt github repo-ban [majd később lesz hozzáadva].

Egy blokkos esetben alábbi kernel futtatandó:

```
__global__ void inversionKernel_1Block(DATATYPE* Matrix, DATATYPE* InvMatrix,
                                       DATATYPE* L, DATATYPE* U, DATATYPE* Z, int size)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;  // oszlopváltozó
    int j = blockIdx.y * blockDim.y + threadIdx.y;  // sorváltozó
    if (i >= size || j >= size)
        return;

    thread_block block = this_thread_block();
```

```
// 1. lépés: LU dekompozíció

// Előformázás
U[i * size + j] = Matrix[i * size + j];
if (i < j)
    L[i * size + j] = 0;
else if (i == j)
    L[i * size + i] = 1;

block.sync();

// Gauss elimináció az U mátrixban
for (int m = 0; m < size; m++)
{
    if (U[m * size + m] == 0)
    {
        if (i == 0 && j == 0)
        {
            printf("Inversion not calculatable with LU Decomposition!\n");
        }
        return;
    }
    DATATYPE x = U[i * size + m] / U[m * size + m];     // Ez soronként ugyanaz
    if (j == m && i > m)
    {
        // Ha a nullázandó oszlopban vagyunk
        U[i * size + j] = 0;
        L[i * size + j] = x;
    }
    else if (i > m && j > m) {
        U[i * size + j] -= x * U[m * size + j];
    }
    block.sync();
}

// Felbontottuk a mátrixot egy felsőháromszög-, és egy alsóháromszögmátrixra
int k;
int l;

Z[i * size + j] = 0;
block.sync();

// 1 dimenzios párhuzamosítás (szinkronizálások mindenkire vonatkoznak)
// size db teljesen független műveletvégzés Matrix[][i] oszlopvektorokon

// 2. lépés: = (oszlopvektor, azért van i és j fordítva) meghatározása
if (j == 0)
{
    Z[i * size + i] = 1;
}
block.sync();

if (j == 0)
{
    for (k = i + 1; k < size; k++)
    {
        for (l = 0; l < k; l++)
        {
            Z[k * size + i] -= Z[l * size + i] * L[k * size + l];
        }
    }
}
block.sync();

// Elkészült Z
```

```
    // 3. lépés: InvMatrix[][i] oszlopvektor meghatározása, módszer backpropagation
    InvMatrix[j * size + i] = Z[j * size + i];
    block.sync();
    if (j == 0)
    {
        for (k = size - 1; k >= 0; k--)
        {
            for (l = k + 1; l < size; l++)
            {
                InvMatrix[k * size + i] -= U[k * size + l] * InvMatrix[l * size + i];
            }
            InvMatrix[k * size + i] /= U[k * size + k];
        }
    }
}
```

A több blokkos kivitel hasonlóan megírható, teljes kód megtalálható a repo-ban. (később adom hozzá)

5.3.2. Tesztelések

A teszteléshez inputnak különböző méretű, matlab által véletlenszerűen generált mátrixokat használtam. Megvizsgáltam, hogy az adattípus, valamint a blokkszám hogyan befolyásolja a futásidőt és az eredményt. Blokkszámot úgy lehet befolyásolni, ha módosítjuk a CUDA rendszerre jellemző blokkméretet. (Megjegyzés: ezt már azért is megéri megtenni, mert különböző architektúrákon kisebb vagy nagyobb lehet egy blokkon belül maximálisan futtatható szálak száma, ezáltal más-más kártyákon jelentősen eltérő teljesítményt tapasztalhatunk).

Összehasonlítási alapon matlabban is elvégeztem a beépített inv(Matrix) utasítással a műveletet, és összehasonlítottam az eredményeket. A matlab eredményét tekintem referenciának.

Az eredmény összehasonlítását automatizálhatjuk azzal, hogy a kiegészítő program saját maga hasonlítja össze a GPU által számolt eredményt az elméleti helyes eredménnyel. Én parancssori argumentumba adom meg opcionálisan a helyes eredményt, és ekkor %-ban megadva kiírja a program az átlagos, illetve maximális (a mátrixelemekre vett) relatív hibát. Ennek egy fajta C nyelvű implementálása (ha matlab segítségével kiírjuk fájlba az általa számolt eredményt):

```
FILE* pSolutionFile = NULL;
bool givenSolution = false;
double* Solution;

int i;  // Iterator
for (i = 1; i < argc; ++i)  // Processing command line arguments
{
    // Command Line Syntax: ... --s [SolutionFileName]
    if ((strcmp(argv[i], "--s") == 0) || (strcmp(argv[i], "--Solution") == 0))
    {
        pSolutionFile = fopen(argv[++i], "r");
        if (pSolutionFile == NULL)
        {
            fprintf(stderr, "Unable to open file \"%s\"", argv[i]);
            return -1;
        }
        else
        {
            givenSolution = true;
        }


    }
}

// ...

// Reading solution values
```

```
if (givenSolution)
{
    Solution = (double*)malloc(size * size * sizeof(double));
    for (int ii = 0; ii < size; ++ii)
    {
        double temp;

        for (int jj = 0; jj < size; ++jj)
        {
            if (fscanf_s(pSolutionFile, "%lf ", &temp) != 1)
            {
                fprintf(stderr, "Error reading SolutionMatrix(%d %d)\n", ii, jj);
                fclose(pFile);
                fclose(pSolutionFile);
                return -1;
            }
            Solution[ii * size + jj] = temp;
        }
        //fscanf_s(pFile  "\n");
    }
    fclose(pSolutionFile);
}

// Calculations ...

// Testing the punctuality of the results
if (givenSolution)
{
    double sumError = 0.0;
    double maxError = 0.0;

    for (i = 0; i < size * size; i++)
    {
        double erroruPercentage = abs(((double)InvMatrix[i] - Solution[i]) / Solution[i
    ]) * 100000000;
        sumError != erroruPercentage;
        if (erroruPercentage > maxError)
            maxError = erroruPercentage;
    }
    sumError /= size * size;
    printf("Average difference from correct solution: %.12f u%%\n", sumError);
    printf("Max difference from correct solution: %.12f u%%\n", maxError);
}
```

Mikroszázalékban adom meg a hibát (10^{-8}) azért, mert a double esetében gyakran ilyen pontossággal kell összehasonlítást végezni ahhoz, hogy eltérést tapasztaljunk. A mérések eredményét az alábbiakban táblázatosan tüntettem fel:

5x5-ös mátrix					
	Futásidő (µs)	Átlagos Relatív hiba (µ%)	Max. Relatív hiba (µ%)	Compute TP (%)	Occupancy (%)
float, 1blokk	7.49	22.18	0	260.39	4.12
double, 1blokk	13.34	0.0000001	0.0035	0.94	3.74

4. táblázat

5.4. Eredmények értékelése

Adattípus Az eredmények alapján sokat gyorsíthat az eljáráson, hogyha egyszeres pontosságú lebegőpontos számokkal float-tal dolgozunk. Ilyenkor az eredmény pontosságára oda kell viszont figyelni. Főleg kisebb memóriájú videokártyák esetén lehet lényeges a spórolás: például ha szükségünk van 10 millió változóra, és a double 8bájtos, a float pedig 4 bájtos, akkor float használatával 40millió bájtot,

10x10-es mátrix					
	Futásidő (µs)	Átlagos Relatív hiba (µ%)	Max. Relatív hiba (µ%)	Compute TP (%)	Occupancy (%)
float, 1blokk	16.67	380.22	6084.2	0.35	6.7
float, 4blokk	42.02	380.22	6084.2	0.47	5.93
double, 1blokk	32.93	0.000001	0.00002	1.22	6.46
double, 4blokk	56.35	0.000001	0.00002	1.36	5.72

5. táblázat

18x18-as mátrix					
	Futásidő (µs)	Átlagos Relatív hiba (µ%)	Max. Relatív hiba (µ%)	Compute TP (%)	Occupancy (%)
float, 1blokk	39.07	66594.95	6577386.35	0.5	17.59
float, 4blokk	77.12	66594.95	6577386.35	0.57	5.81
double, 1blokk	93.54	0.000030	0.0035	1.84	17.73
double, 4blokk	114.72	0.000030	0.0035	2.01	7.35

6. táblázat

32x32-es mátrix					
	Futásidő (µs)	Átlagos Relatív hiba (µ%)	Max. Relatív hiba (µ%)	Compute TP (%)	Occupancy (%)
float, 1blokk	159.42	2527.87	931314.59	0.43	55.53
float, 4blokk	171.58	527.87	931314.59	0.70	14.82
double, 1blokk	338.59	0.0000028	0.00078	2.21	55.23
double, 4blokk	286.66	0.0000028	0.00078	2.86	14.27

7. táblázat. Alapvetően 1024 szálat tartalmazhat egy blokk, ezért most kipróbáltam a lehető legnagyobb blokkmérettel az algoritmust. Double esetében megfigyelhető, hogy már gyorsabb a futás, ha 4 db 256 szálú blokkra szét van szedve a rendszer

50x50-es mátrix					
	Futásidő (µs)	Átlagos Relatív hiba (µ%)	Max. Relatív hiba (µ%)	Compute TP (%)	Occupancy (%)
float, 4blokk	375.30	12960.1	1422408.01	0.95	58.07
float, 16blokk	313.82	12960.1	1422408.01	1.92	15.35
double, 4blokk	787.04	0.000024	0.0046	4.92	58.13
double, 16blokk	569.70	0.000024	0.0046	7.04	15.11

8. táblázat

75x75-ös mátrix					
	Futásidő (µs)	Átlagos Relatív hiba (µ%)	Max. Relatív hiba (µ%)	Compute TP (%)	Occupancy (%)
float, 9blokk	699.04	11271.20	6189999.02	1.49	37.35
float, 16blokk	656.03	11271.20	6189999.02	2.05	24.8
double, 9blokk	1314.2	0.000037	0.021	7.70	36.92
double, 16blokk	1184.1	0.000037	0.021	9.39	24.33

9. táblázat

azaz mintegy 38MB memóriát takaríthatunk meg.

80x80-as mátrix					
	Futásidő (µs)	Átlagos Relatív hiba (µ%)	Max. Relatív hiba (µ%)	Compute TP (%)	Occupancy (%)
float, 9blokk	900.99	56800.79	64265928.6	1.48	60.81
float, 16blokk	747.10	56800.79	64265928.6	2.03	24.84
double, 9blokk	1752.6	0.00013	0.087	7.70	60.33
double, 16blokk	1386.7	0.00013	0.087	8.74	24.51

10. táblázat. Észrevevendő, hogy float használatával előfordul akár 64%-os hiba is $\Longrightarrow$ Ilyen nagy mátrixok számítására már alkalmatlan az egyszeres pontosságú lebegőpontos számábrázolás

Tömb dimenzió Kevéssel ugyan, de gyorsabb futású a program, ha egydimenziós tömböt használunk. Ez nagyobb odafigyelést igényel, viszont eredményes lesz, ha megtesszük.

Blokkszám, blokkméret Számomra meglepő tapasztalás volt, hogy nagy adathalmazokon gyorsabban futottak az algoritmusok, ha több, de kisebb párhuzamos blokkot alkalmaztam.

6. Genetikus algoritmusok vizsgálata

Ebben a fejezetben útkeresési problémákkal foglalkozok, amelyek NP-teljes problémák, ezért úgynevezett genetikus algoritmusokkal igyekszem közelíteni a végeredményt.

A mérést végző számítógép specifikációja A mérést a saját laptopomon végeztem.

- CPU : Intel 15-11300H
- GPU : NVIDIA GeForce RTX 3050 Ti
- RAM : 24GB DDR4
- Operációs Rendszer : Windows 10

A jövőben tervezem erősebb hardver mellett is tesztelni az algoritmusokat.

6.1. Fogalmak ismertetése

Jármű útvonaltervezési probléma [2] A jármű útvonaltervezési probléma (Vehicle Routing Problem, továbbiakban: VRP) egy fajta optimalizációs probléma, mely során járművеknek (itt: a teljes) útiköltség minimalizálásával kell végigjárniuk megadott célpontokat. A problémát különböző megkötésekkel lehet feltenni az alkalmazás igénye alapján. Ezek lehetnek például:

- járművek maximális száma
- az egyes járművek szállítási kapacitása
- az egyes járművek által megtehető maximális úthossz

Az egyszerűség kedvéért a járművek ugyanabból a pontból indulnak (0. gráfcsúcs, warehouse), a kiindulási pontot leszámítva minden állomást legalább egyszer érinteniük kell és az útjuk végén vissza kell térniük a közös kiindulópontba. A járművek minden állomáson pontosan egyszer rakhatnak le árut, és részleges kiszállítás nincs megengedve. A 3. ábrán látható egy vizuális szemléltető példa.

Matematikai megfogalmazás : a problémát gráfokkal modellezhetjük.
Legyen G = (V,E) gráf, n az állomások száma (a kiindulási állomást beleértve), m az állomások között futó elérhető utak száma, k az elérhető járművek maximális száma.

$$V = (v_0, v_1, \dots v_n)$$

állomások halmaza (Vertex), $0 \in V$ a kiindulási állomás,

$$E = (e_1, e_1, \dots e_m)$$

elérhető utak halmaza (Edge),

$$D : E(G) \mapsto \mathbb{Z}^+$$

költségfüggvény (Distance),

$$L = (l_1, l_2, \dots l_k)$$

a járművek szállítási kapacitása (Load capacity),

$$C : V(G) \setminus \{v_0\} \mapsto \mathbb{R}^+$$

az egyes állomások áruigénye (Claim) .

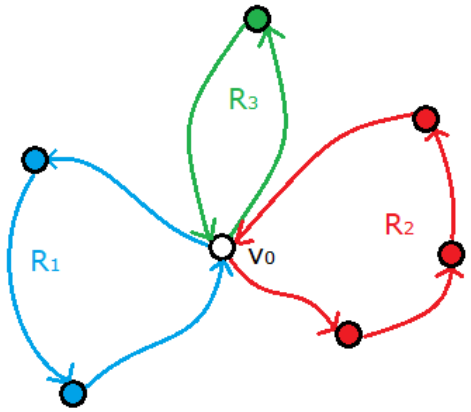
Az élek azonosítása érdekében éljünk a következő jelöléssel: e_{ij} a v_i-ből v_j-be mutató él, és $d_{i,j}$ az $e_{i,j}$ költsége (itt : távolság, distance). Adott továbbá minden csúcshoz a c_i áruigény, ami ki kell elégíteni (ez a valóságban lehet db, kg, stb.).Adott minden csúcshoz a c_i áruigény, amit ki kell elégíteni (ez a valóságban lehet darab, kg, stb.). Legyen l_i az i-edik jármű szállítási kapacitása.

Állítsuk elő útvonalak (Route) olyan $R_i = (0, v_{i_1}, v_{i_2}, \dots 0)$ listáit, ahol az i-edik jármű azon útvonalát adja meg, amelyet alkotó élek $(e_{0,i_1}, e_{i_1,i_2}, \dots e_{i_m,0})$. Az útvonal költsége az azt alkotó élek összköltsége.

$$c(R_i) = \sum_{v \in R_i} c(v) \tag{1}$$

A cél azon $R_1, R_2, ...R_k$ útvonalak megtalálása, amelyekre a következők igazak:

- összköltségük minimális
- kiindulási és végpontjuk a 0. állomás
- a kiindulási csúcsot leszámítva minden csúcsot pontosan egyszer tartalmaznak, vagyis $\forall v_i \in V, v_i \neq v_0$ esetén $\exists! R_j : v_i \in Rj$.
- egyik jármű sem szállíthat több árut a megengedettnél, vagyis $l_i \geqslant \sum_{v \in R_i} c(v)$



3. ábra. Egy példa VRP végrehajtására n=7 csúcsú gráfon, k=3 járművel

Utazóügynök probléma [2] Az utazóügynök probléma (Travelling Salesman Problem, TSP) a VRP egy speciális esete, amikor egyetlen jármű van és nincsenek egyéb korlátozó feltételek. A nevét onnan kapta, hogy a XX. században utazó porszívóügynökök autóval járták az Egyesült Államok útjait kereskedés céljából. Az olajválság során megdrágult a járművek működtetéséhez szükséges üzemanyag, és hirtelen megnőtt az igény arra, hogy csökkentsék a megtett út hosszát egy-egy üzleti út során.
A problémának azóta több alkalmazása is lett, ebből a villamosmérnöki gyakorlathoz egyik legközelebb az SMD beültetőgép bejárása áll. A gép feladata, hogy egy adott nyomtatott áramköri terv alapján lepakolja az alkatrészeket a hordozó lemezre. Az iparban fontos a sebesség, ugyanis ha felére csökkentjük a beültetési időt, akkor akár duplaannyi terméket gyárthatunk le azonos idő alatt.

Hangyakolónia algoritmus [2] Az előbb bemutatott problémákra az optimális megoldás megtalálása NP-nehéz feladat, emiatt nagy csúcs- és élhalmaz mellett nem gazdaságos a kiszámítása. Annak érdekében, hogy a gyakorlatban használható algoritmust konstruáljunk, valamilyen közelítő, heurisztikus megoldást érdemes használni a legjobb megoldást megtaláló algoritmusok helyett. A hangyakolónia algoritmus (Ant Colony Optimization, továbbiakban: ACO) egy heurisztikus algoritmus, amely amellett, hogy képes az optimálishoz nagyon közeli megoldásokat biztosítani, alkalmas a nagyfokú párhuzamosításra. Utóbbi tulajdonsága miatt alkalmas az NVIDIA$^{\text{TM}}$ CUDA architektúrájával történő adaptálásra.

Az algoritmus a nevéből adódóan a hangyák (Formicidae) természetben is megfigyelhető élelemkeresési módszerén alapszik. Az első hangyák véletlenszerű (random) útvonalakon haladva keresik az élelemhez vezető utat, majd a visszaúton a bolyba feromonnal jelölik meg az útjukat. Ez alapján a később induló hangyák a feromonok erőssége alapján már megfontoltabb döntéseket hozhatnak a bejárást illetően. A hátrahagyott feromon erőssége idővel gyengül (modellezhető exponenciális lecsengéssel), ezzel biztosítva, hogy a nem optimális útvonalak (az élelemhez vezet, de már van nála rövidebb) maguktól elhaljanak. Látható, hogy olyan él, ami sok ideig nem kap feromon utánpótlást, exponenciális jelleggel lecseng.

Az algoritmus futása során nyilvántartunk egy feromongráfot. A feromongráf és az eredeti gráf topológiája megegyezik, de az élek súlyozása eltérő. Legyen Ph(V,E) gráf, amiben az élek súlyai $e_{i,j} \rightarrow \tau_{i,j}$.

Gráfbejárás során egy v_i-n álló hangya a továbblépéséhez a lehetséges kimenő élek közül a feromon és és az élsúly alapján súlyozottan, véletlenszerűen választ. Az egyes élek kiválasztásának valószínűsége

$$P_{i,j} = \frac{(\tau_{i,j})^{\alpha}(d_{i,j})^{\beta}}{\sum_{l \in N_i^k}(\tau_{i,l})^{\alpha}(d_{i,l})^{\beta}} \tag{2}$$

ahol N_i^k az algoritmus k-adik lépésében az i-edik csúcsból elérhető szomszédos csúcsok halmaza. Az α és β a feromonok és élsúlyok figyelembevételét szabályozza. Például minél nagyobb valahol a feromon, annál inkább akarunk oda továbbmenni, illetve minél messzebb van egy adott pont, annál inkább el akarjuk kerülni. Saját megvalósításom esetén gyorsítás céljából elhanyagoltam ezen ponton az élhosszak figyelembe vételét, ezért $\beta = 0$. Továbbá az egyszerűség kedvéért legyen $\alpha = 1$. Ennek egy előnye, hogy a (2). egyenlet a következővé egyszerűsödik:

$$P_{i,j} = \frac{\tau_{i,j}}{\sum_{l \in N_i^k} \tau_{i,l}} \tag{3}$$

Miután minden hangya végigment egy úton (legenerált egy csúcssorrendet, legyen az akár lehetséges, akár nem) értékeljük az útvonalakat. A teljesíthető útvonalak esetén a élek feromonszintjét a útvonal hosszával fordítottan arányosan $\beta = -1$ megnövelem. Ez biztosítja, hogy a rövidebb útvonalak nagyobb feromonszinttel rendelkezzenek, ezáltal több hangya menjen előbb-utóbb olyan irányba.

Az éleken található feromon növelése után mindegyik élt exponenciális jelleggel csökkentem: minden feromon gyengül egy előre beállítandó, konstans szorzóval, ez a párolgás ρ. Összességével minden iteráció végén a következő történik egy él feromonjával:

$$\tau_{i,j} \leftarrow (1 - \rho)\tau_{i,j} + \Delta\tau_{i,j}$$

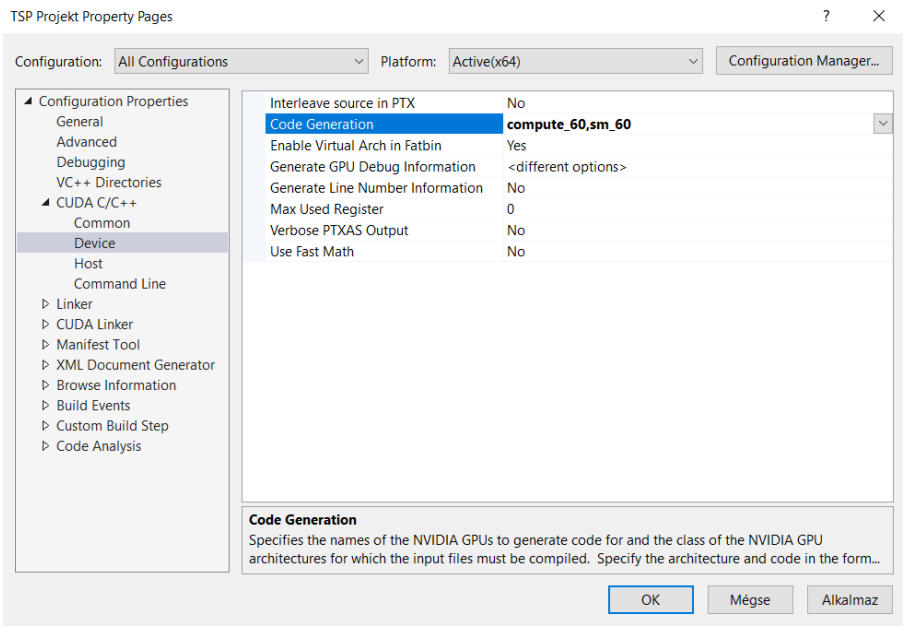
Az útkeresés közben Az mindig fel kell jegyezni a legjobb addigi megtalált utat. Az ACO algoritmus egyik előnye, hogy több, hasonlóan jó alternatív utat is képes megtalálni. Ez például térképes útvonaltervezésnél lehet hasznos. Az algoritmus javítása érdekében bevezettem, hogy az a hangya, aki minden korábbinál rövidebb utat talál, az egy ún. "Jutalom szorzó"-t kap. Ez azt jelenti, hogy a feromonjához adandó többlet a szorzószorosára változik, így a következő iterációban sokkal nagyobb valószínűséggel fog arra menni a jövő hangyája. Ez a jutalmazási rendszer rossz beállítások mellett kezdetben félreviheti a hangyákat egy rossz útvonal felé, ezért állítható, hogy hányadik iterációtól kezdve kaphassanak az útvonalak "Jutalom szorzó"-t.

6.2. TSP első verzió

Ahogy az előző alfejezetben részleteztem, a Jármű útvonaltervezési problémák legalapszintűbb tagja az Utazóügynök probléma (továbbiakban: TSP), ezért ezzel érdemes kezdenie annak, aki szeretne elmélyülni a problémakörben. Legyen a fejlesztői környezet Visual Studio 2022.

6.2.1. Új projekt létrehozása

Új projekt létrehozásához a projekt varázslónál válasszuk a "CUDA [verziószám] Runtime" nevű templatet. Aktuális verziószám "12.0". A projekt létrehozása során érdemes pár dolgot módosítani a projekt konfigurációjában. Szeretnénk használni a Cooperative groups grid szinkronizáció [3] funkciót, amihez Compute Capability 6.0+ szükséges. Hogy a mi videókártyánk alkalmas-e ilyen algoritmus futtatására, azt a GPU gyártó weboldalán [6] ellenőrizhetjük. A legkisebb olyan GPU, ami alkalmas számunkra, a GTX1060 (Pascal architektúra). Amennyiben alkalmas a kártyánk, válasszuk a Visual Studio-n belül Solution Explorerben a projektünket, jobb klikk, majd "Properties"/"Configuration Properties"/"CUDA C/C++"/"Device" fülben írjuk át a Code Generation paramétert a 4. ábrának megfelelően.



4. ábra. Visual Studio konfiguráció beállítása. Érdemes a lehető legkisebb feltételekkel (6.0) beállítani, hogy régebbi kártyákon is lehessen futtatni

6.2.2. A kód elemzése

Bevezettem egy globális változót a hangyák (threadek) számának állítására. Ez későbbi verziókban módosulhat, például a felhasználó adhatja meg, hogy hány hangyát szeretne. Jelenleg konstans:

```
// Number of threads = number of ants
const int ants = 1024;
```

Van még néhány olyan algoritmus paraméter, ami a futás során változatlan, viszont értékük befolyásolja a futásidőt, illetve a végeredményt. Kiszerveztem őket makróba, hogy csak egy helyen kelljen változtatni.

```
// Repetition constants
#define REPETITIONS 10
#define RANDOM_GENERATIONS 20
#define FOLLOWER_GENERATIONS 500

// Pheromone matrix constants
#define RHO 0.75  // Reduction ratio of previous pheromon value
#define REWARD_MULTIPLIER 100   // Reward multiplier after finding a shortest path
#define INITIAL_PHEROMONE_VALUE 1000    // Initial value of elements in the Ph. matrix


#define SERIAL_MAXTRIES 10    // Number of serial processes
```

Először a "Repetition" konstansokról: Az algoritmus így működik: Csináld REPETITIONS alkalommal, hogy jön RANDOM_GENERATIONS iteráció olyan hangya, aki teljesen véletlenszerűen végigmegy a gráfon, majd őket követi FOLLOWER_GENERATIONS iterációban olyan hangya, aki a feromonom alapján dönti el az útját. Minél több iterációt hajtunk végre, annál több utat vizsgálunk meg, és elvileg annál pontosabb lehet a végeredmény. Az előbbiektől némileg elkülönül a SERIAL_MAXTRIES, ami azért felel, hogy többször lefuttathassuk egymás után a GPU kernelt. Mivel az ACO egy heurisztikus algoritmus, futásonként más és más eredményt szolgáltat. Éppen ezért érdemes lehet többször (például 10-szer) egymás után végrehajtani az algoritmust, és megvizsgálni az egyes eredményeket. Ilyenkor végső eredménynek célszerű a legrövidebb megoldást venni.

6.2.3. Tesztelések

Teszteléshez szükségem volt ismert eredményű adathalmazokra. A Floridai Állami Egyetem weboldalán [5] elérhető bárki számára több adathalmaz, különböző adatstruktúrában. Nekem a [fájlnév]_d.txt nevű fájlok voltak hasznosak, ugyanis abban megtalálhatóak a szomszédossági mátrix költségei mátrix alakban. Az itt található 6 adathalmazon végigfuttattam az algoritmusomat több konfigurációban. Mindig 10-szer ismételtem meg a futást, és képeztem az eredmények átlagát (számtani középpel), illetve minimumát. Nagy adathalmaz esetén hosszú a futásidő profilozó módban, ezért időmérés céljából egyszer futtattam újra ugyanazon beállításokkal. A TSP első verziójában a feromon mátrix és az elsúlyok tárolása csak double formátumban történik. Az összehasonlíthatóság érdekében egy iteráció során 20 random hangya és 500 követő hangya fut. A kezdeti feromon érték 1000, az elnyomási tényező $\rho = 0.75$, a jutalmazási tényező 100, amely csak a 2. iterációtól érvényes (ha van).

FIVE : 5 csúcs, minimális út : 19, átlagos út : 24			
	Futásidő (ms)	Végeredény átlag	Végeredmény min.
1 rep			
32 ant	7,84	20,2	19
256 ant	19,1	20,8	19
1024 ant	76,1	20,8	19
10 rep			
256 ant	137,3	19	19
1024 ant	424	19	19

11. táblázat. Elsőnek egy kicsi, 4 állomásból (és a 0. kiindulási pontból) álló gráfon próbáltam ki az algoritmust.

A 17 vagy nála nagyobb gráfok esetében az 1 iterációs algoritmusok már olyan rosszul teljesítettek, hogy csak 10 iterációs eseteket soroltam fel a hangyák számának függvényében.

P01 : 15 csúcs, minimális út : 291, átlagos út : 662			
	Futásidő (s)	Végeredény átlag	Végeredmény min.
1 rep			
1024 ant	0,925	370,2	291
2048 ant	1,18	365	327
4096 ant	1,20	359,7	332
10 rep			
1024 ant	11,39	350,4	291
2048 ant	11,47	328	295
4096 ant	11,59	336,8	291

12. táblázat

GR : 17 csúcs, minimális út : 2085			
	Futásidő (s)	Végeredény átlag	Végeredmény min.
10 rep			
1024 ant	14,85	2391,3	2151
2048 ant	14,88	2363,4	2085
4096 ant	14,97	2279,6	2097
8192 ant	15,15	2306,4	2207
16384 ant	15,49	2250,5	2085

13. táblázat

FRI26 : 26 csúcs, minimális út : 937, átlagos út : 2693			
	Futásidő (s)	Végeredény átlag	Végeredmény min.
10 rep			
1024 ant	39,72	1386,1	1249
2048 ant	39,73	1367,1	1221
4096 ant	39,82	1227,6	1121
8192 ant	40,14	1158,7	1102
16384 ant	40,70	1132,1	1075

14. táblázat

DANTZIG42 : 42 csúcs, minimális út : 699, átlagos út : 3110,5			
	Futásidő (s)	Végeredény átlag	Végeredmény min.
10 rep			
1024 ant		1735,7	1554
2048 ant		1420	1252
4096 ant		1263,8	1106
8192 ant		1082,6	928
16384 ant	114,34	987,7	906

15. táblázat

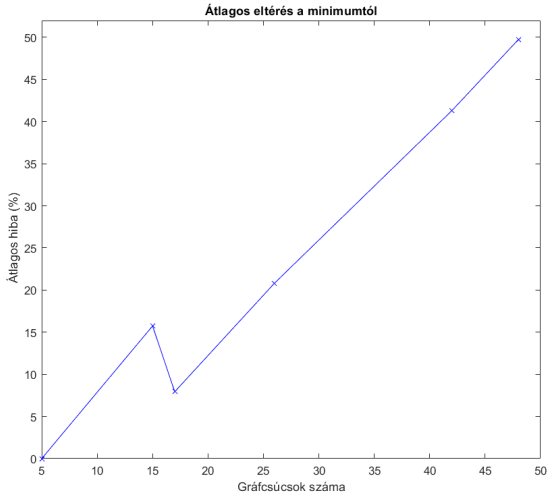
6.3. TSP Továbbfejlesztett verzió

Az előző (6.2.) fejezetben ismertettem a TSP algoritmusra készült első megoldásomat, amely kezdetlegességéből adódóan több szempontból is hiányos volt. A főbb motivátorok ezek voltak egy új, jobb verzió elkészítésére:

- A hangyák, vagyis a szálak száma konstans volt, ami nem jó, ha különböző méretű és bonyolultságú gráfon kívánunk minimális bejárást találni
- Az első verzió kötelezően double-t használt a változók nagy részének tárolására (a sorszámozást

ATT48 : 48 csúcs, minimális út : 33523, átlagos út : 157686,9			
	Futásidő (s)	Végeredény átlag	Végeredmény min.
10 rep			
1024 ant			
2048 ant			
4096 ant			
8192 ant	152,82		
16384 ant	153,78	50197,2	47387

16. táblázat



5. ábra. Matlabban ábrázoltam a gráfcsúcsok számának függvényében az elméleti minimumhoz képesti átlagos hibát.

leszámítva). Belátható, hogy például a feromonmátrixon történő pontos műveletvégzés nem célunk, a mi feladatunk gyorsan rövid utat találni. Márpedig a 5. fejezetben láttuk, hogy nagyon sokat gyorsíthat futásidőben, ha double helyett float-ot használunk adatípusnak.

- A különböző útvonalkeresési algoritmusok (TSP, VRP, CVRP stb.) sokban hasonlítanak, hasonló algoritmusok léteznek rájuk. Célom volt, hogy egységes kódot írjak azért, hogy később kis módosításokkal átvihetőek legyenek egymásba a különböző kódok.

- Az idő haladtával több tapasztalatot szereztem a CUDA programozásban, és számos helyen találtam gyorsítási lehetőségeket az előző projektben. Nem akartam megbontani a már kész forrásfájlokat, ezért inkább új projektben valósítottam meg új ötleteimet.

Több változást eszközöltem. Bevezettem egységes szintaktikát a paraméterlisták megadására. A videokártyán nem lehet közvetlenül dinamikus memóriát foglalni, ezért minden ideiglenes tömböt paraméterben adok át. A bevezetett struktúrák átláthatóbbá és rövidebbé teszik a függvény fejléceket. A mezők neveiben azt a nevezéktant tartottam, hogy mátrix változók FistSecond...Last típusú, egyáb (skalár vagy vektor) változók firstSecond...Last nevet kaptak.

```
// Struct for Main CUDA function call
  typedef struct {
    DATATYPE* Dist;
    DATATYPE* Pheromone;
    int* route;
    bool* foundRoute;
    int antNum;
    size_t size;
  } TSP_AntCUDA_ParamTypedef;


  // Struct for kernel call
  typedef struct {
    DATATYPE* Dist;      // Cost function input
    DATATYPE* Pheromone;
    int* route;          // Sequence output
    int size;         // Number of graph vertices
    bool* foundRoute;    // Existence output
    int* antRoute;       // Temp array
    int antNum;          // Number of ants
    curandState* state; // CURAND random state
  } TSP_AntKernel_ParamTypedef;


  // Variables allocated in global memory for communication between different thread
    blocks
  typedef struct {
    bool* invalidInput;   // Variable used to detecting invalid input
    bool* isolatedVertex;  // Variable used to detecting isolated vertex (for
    optimization purposes)
    DATATYPE* averageDist;
    DATATYPE minRes;     // Minimal found Route distance
  } TSP_AntKernel_Global_ParamTypedef;


  typedef struct {
    // Repetition constants
    unsigned repetitions;
    unsigned randomGenerations;
    unsigned followerGenerations;
    int maxTryNumber;    // Follower ants use this to stop weighted roulette
    // Pheromone matrix constants
    float Rho;
    float rewardMultiplier;
    DATATYPE initialPheromoneValue;
  } TSP_AntKernel_Config_ParamTypedef;
```

A TSP_AntCUDA_ParamTypedef típusú struktúrát a main host függvény kapja meg. A TSP_AntKernel_Par típusút a kernel kapja meg. Több blokkos kernelben nem működik a __shared__ típusú változóattribútum, ezért globális memóriából kell pótolni. Ekkor a device függvény kap egy TSP_AntKernel_Global_ParamTy típusú struktúrát. Az első verzióban csak makrókkal szerepeltek konfigurációs paraméterek, most lett rá TSP_AntKernel_Config_ParamTypedef típusú struktúra.

Hivatkozások

[1] *55th Anniversary of Moore's Law*. https://www.infoq.com/news/2020/04/Moores-law-55/. Accessed: 2020-04-27.

[2] Tóth Márk Andor. "Algoritmusok Optimalizálása GPU-n". diplomaterv. Budapesti Műszaki Egyetem, 2022.

[3] *Cooperative Groups - Grid Synchronization*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#grid-synchronization.

[4] *CUDA C++ Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 2020-04-27.

[5] *Datasets fot Traveling Salesperson Problem - Florida State University*. https://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html.

[6] *NVIDIA - Your GPU Compute Capability*. https://developer.nvidia.com/cuda-gpus.

[7] *Thrust, the CUDA C++ template library*. https://docs.nvidia.com/cuda/thrust/.

[8] *Wolfram Alpha online calculator*. https://www.wolframalpha.com.

[9] Király Zoltán. "Kvantummechanikai számítások optimalizálása". diplomaterv. Budapesti Műszaki Egyetem, 2022.