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

Útvonaltervezési algoritmusok optimalizálása GPU-n

Szakdolgozat

Készítette
Jost Márk Benedek

Konzulens
Dr. Szegletes Luca

2023. november 20.

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott *Jost Márk Benedek*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot 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, 2023. november 20.

Jost Márk Benedek
hallgató

Kivonat

Különböző hálózatok, legyen az egy szociális média felhasználói hálója, egy ország közúthálózata, de akár egy NYÁK-terven a különböző komponensek vezetékezése során alapvető problémák az úgynevezett útkeresési problémák (Vehicle Routing Problems - VRP). Ezek bizonyítottan NP-nehéz problémák, a megoldásuk egy költséges, ugyanakkor gyakran monoton tevékenység, hiszen sok pontsorozat hossza közül kell kiválasztani a legrövidebbet. Dolgozatom célja a Hangyakolónia Optimalizáció (Ant Colony Optimization - ACO) elvével egy heurisztikus, vagyis valószínűségi megoldás találása polinomidőben. Az ACO nagymértékben párhuzamosítható, melyet azzal tudtam kihasználni, hogy videokártyán összehangolt threadek ezreit voltam képes futtatni az NVIDIA CUDA keretrendszer segítségével. A készült kódot C/C++ nyelven valósítottam meg.

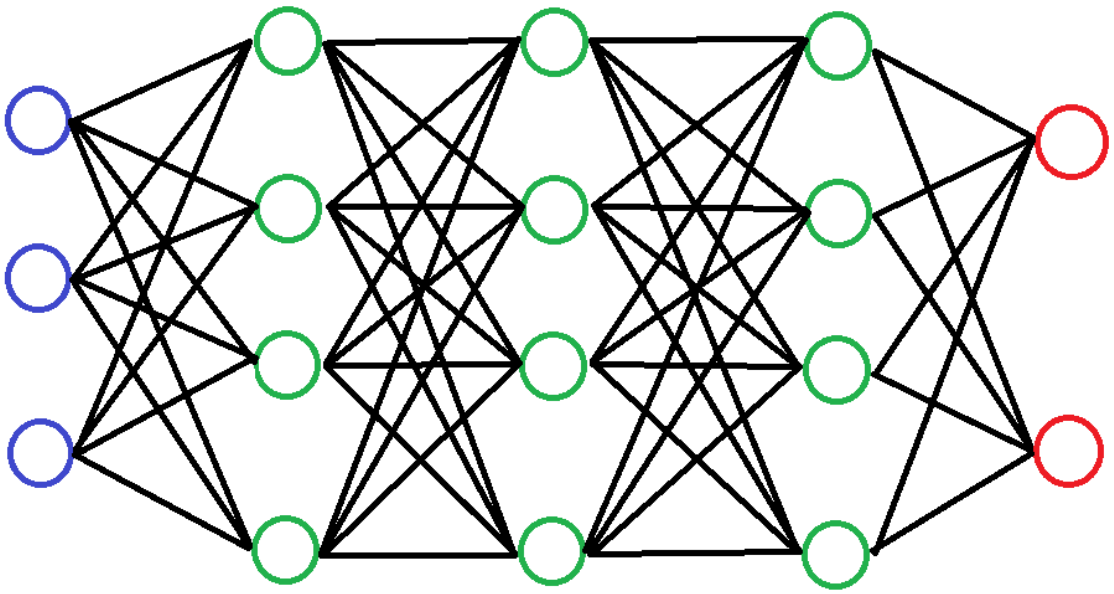
Abstract

This document is a LaTeX-based skeleton for BSc/MSc theses of students at the Electrical Engineering and Informatics Faculty, Budapest University of Technology and Economics. The usage of this skeleton is optional. It has been tested with the *TeXLive* TeX implementation, and it requires the PDF-LaTeX compiler.

1. fejezet

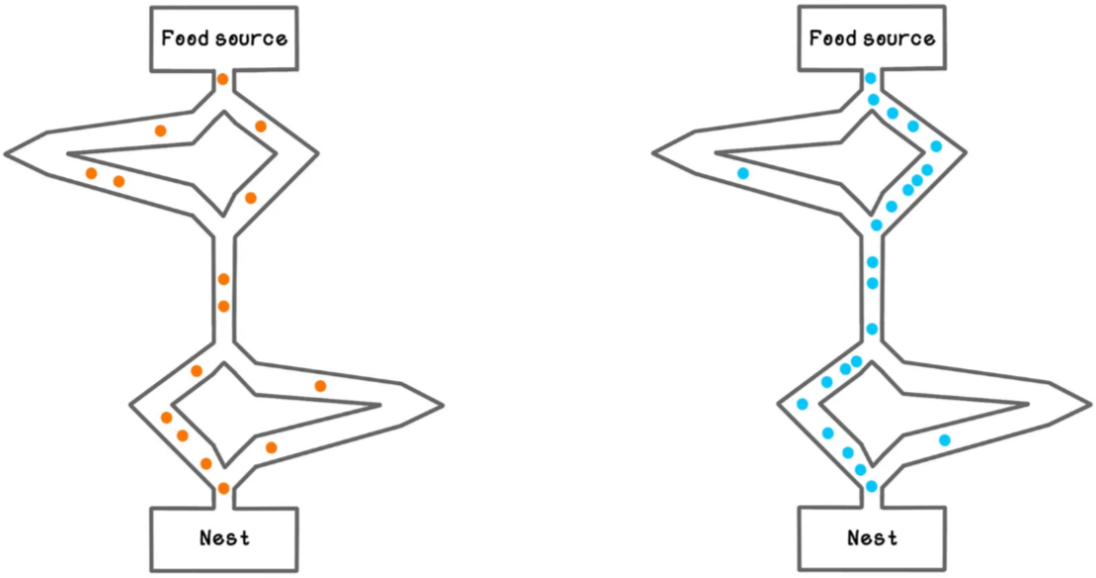
Bevezetés

Konzulensem, Dr. Szegletes Luca már régóta gyűjt videokártyás kódimplementációkat különböző NP-nehéz problémákra. Amikor megbeszéltük, hogy mi lenne a munkám célja, különböző útkeresési algoritmusok megvalósítását kaptam feladatul. A jármű útvonaltervezési algoritmusoknak nagyon sok alesete van, ugyanis végtelenül sokféle feltételeket szabhatunk meg egy bejárás számára: a teljes hossz legyen rövid, az egyes utak legyenek egyenként rövidek, stb. Különböző alkalmazásokhoz nagyon változatos elvárások illenek: például ha egy bútor szállítmányozási cég akarja megtervezni, hogy az aznapi 20 klienséhez milyen sorrendben juttassa el az árukat, már azt is bele kell kalkulálnia, hogy hogyan fognak a termékek elférni egy kamionban (véges kapacitás problémája). Másik példa egy nemzetközi körutazás: a cég vezérigazgatója be szeretné járni egy kampány keretében a cég különböző leányvállalatait, melyek manapság bármely pontján lehetnek: lehet, hogy az egyik üzem Kolumbia közepén, míg egy másik valahol Indiában van. A két ország időzónája nagyon különböző, ezért nem mindegy, hogy az igazgató a nap melyik órájában szeretne találkozni a helyi vezetőkkel (a kliens nem áll korlátlanul rendelkezésre, igazítani kell a fogadó és vendég beosztását). Számtalan hasonló szituáció képzelhető el, a fent említettek csak a legtipikusabbak. Ezen problémákat a számítástudomány már évtizedek óta aktívan kutatja.



1.1. ábra. Többféle modellt is kitaláltak már gépi tanuláshoz, az egyik leghíresebb az ún. neuronháló modell

Napjainkban a mesterséges intelligencia forradalmasította a számítástechnikát a számítási kapacitások soha eddig nem látott bővülése eredményeképpen. Az NVIDIA egy olyan cég, amely hardvergyártóként, videokártyák gyártásával és értékesítésével kezdte működését. Az elmúlt években komoly szerkezetváltozáson ment keresztül, igyekszik némileg közönséget is váltani: korábban a videokártyák legnagyobb felvevőpiaca a videojátékosok voltak. A GPU arra lett kitalálva, hogy különböző 2D-, 3D-s renderelési feladatokban tehermentesítse a CPU-t. Több (fizikai) ezer szálon képes futni. Egy GPU szál cserébe sokkal szűkebb utasításkészleten tud dolgozni, mint egy CPU szál. A GPU szálak elsősorban különböző aritmetikai utasítások végrehajtásában jeleskednek: rendelkeznek hardveres FPU-val (floating point unit - olyan hardver, mely lebegőpontos számokon végzett aritmetikára lett tervezve). A mesterséges intelligenciát használó algoritmusok azért lettek forradalmi vívmányok, mert az algoritmus kiötletelésének nehézségeit nagyrészt képes kivenni a programozók kezéből. Egy klasszikus módon megírt program írása során a kódírónak teljes körű elképzelése kell, hogy legyen arról, hogy hogyan fog eljutni az eredményhez.



1.2. ábra. A hangyák sajátos módon optimalizálják a táplálékszerzést: a Hangyakolónia Optimalizáció segítségével [13]

A gépi tanulás másképp működik: az én esetemben, a **genetikus algoritmusoknál** kell hozzá egy input, és egy hibafüggvény. A programozó megadja, hogy mely bemenetre szeretné ráengedni az algoritmust. Az algoritmus kap még egy kiértékelő függvényt, amellyel számszerű eredményt rendelhet az általa alkotott megoldásokhoz. A gépnek van memóriaterülete, amelyet az alapján írhat, hogy mit tanult. A genetikus algoritmusok megoldásgenerációkat hoznak létre. A gép a hibafüggvény segítségével értékeli az egyes genomokat, majd a legjobban sikerült egyedek alapján készít következő generációt. A folyamat többféle módon is véget érhet:

- a hibafüggvény egy adott hibahatáron belüli megoldást talál
- az algoritmus adott ideig (pl. 2 óra) fut, a végső megoldás az utolsó generáció legjobb megoldása
- a program adott számú iterációt hajt végre, a végső megoldást készítheti külön, a korábban tanultak alapján

A gép kiveszi a programozó kezéből az algoritmizálási feladatot. A kódoló választ egy modellt, amely alapján a gép majd dolgozik. Ha elégedetlenek vagyunk, választhatunk másik modellt, hátha az sikeresebbnek bizonyul.

Szakdolgozatom során a **Hangyakolónia optimalizáció** modelljét alkalmaztam. Ez egy természetből ellesett trükkön alapuló gráfbejárási algoritmus, a hangyák élelemkeresési módszereire hasonlít, melyet a 1.2. ábra szemlélet.

A koncepció központi elemei a **feromonok**, melyek biokommunikációra szolgáló, különböző állatok, rovarok által a kibocsátott kémiai anyagok, amelyek a faj másik egyedéből meghatározott viselkedést váltanak ki.

A hangyák egész nap a hangyaboly környékét járják morzsák, elhullott rovarok után kutatva. Amikor egy dolgozó korábban felfedezetlen élelemforrást talál, kis részével visszaindul a bolyba, és a hazaúton egyfajta testnedvet, **feromont** bocsát ki magából. A többi hangya megérzi a szagot, és a nyomába ered. Ha tényleg táplálékhoz vezetett a feromoncsík, akkor ők is kis részével visszaindulnak az élőhelyükre. A folyamat egészen addig tart, amíg van mit elvinni. Ezután az utólag érkező hangyák azt tapasztalják, hogy elfogyott a táplálék, ezért a visszaúton nem választanak ki feromont. A szél előbb-utóbb elfújja az úton hagyott kémiai anyagokat, ezért nem megy oda több rovar.

1.1. Fejezetek

Bevezetés után a 2. fejezetben először ismertetem a problémák elméleti hátterét. Matematikailag kimondom a feladatokat, valamint bemutatom, milyen - főleg valószínűségszámítási - alapok szükségesek a Hangyakolónia optimalizáció alapos megértéséhez.

A 3. fejezetben bemutatom a GPU programozás technológiai hátterét. Kimondom a GPGPU fogalmát. Példákon keresztül szemléltetem a CUDA szoftvermodell markánsabb fogalmait. Különbséget teszek az egyes szinkronizálási szintek között. Ezek után lépésenként megmutatom, hogyan kell Visual Studio segítségével CUDA programot írni és futtatni.

A 4. fejezetben a kódimplementációimat veszem górcső alá. Részletesen beszámolok az egyes logikai elemek működéséről. Bemutatom a Hangyakolónia Algoritmus megjelenését, és az attól való bizonyos eltéréseket. Részletezem a feromonok nyilvántartását, és a véletlen számok kezelését. Logikailag egymásra építem az algoritmusokat ilyen sorrendben: TSP, VRP, CVRP, CVRPTW.

A 5. fejezet a mérési eredményeimet tartalmazza számtalan táblázat formájában. Igyekeztem minél szélesebb körben lemérni az egyes adathalmazokat. A végén értékelem a kapott eredményeket.

1.2. Rövidítések

Dolgozatomban az alábbi rövidítéseket használom:

- ACO - Ant Colony Optimization: Hangyakolónia optimalizáció/algoritmus
- CPU - Processzor
- CUDA - Compute Unified Device Architecture
- CVRP - Capacitive Vehicle Routing Problem: Korlátozott kapacitású járművek útvonaltervezési problémája
- CVRPTW - Capacitive Vehicle Routing Problem with Time Windows: Korlátozott kapacitású, kötött időbeosztású járművek útvonaltervezési problémája

- GPU - videokártya
- min - minimum
- OS - Operating System: Operációs rendszer
- PRNG - Pszeudorandomszám-generátor
- RAM - Random Access Memory
- rep - Repetition: Iteráció
- SIMD - Single Instruction Multiple Data
- TSP - Travelling Salesman Problem: Utazóügynök probléma
- VRP - Vehicle Routing Problems: Járműútvonal-tervezési problémák

2. fejezet

Elméleti háttér

Ahhoz, hogy kellőképpen megértsük a dolgozat által felvetett problémákat, úgy gondolom, hogy szükséges azokat megfelelően, matematikailag tisztán megalapozni. Célom precízen kimondani a megoldandó problémákat, valamint a rájuk alkalmazott különféle technikákat. A témával sokat foglalkozott hallgatótársam, Tóth Márk Andor, munkája számos helyen inspirált [2].

2.1. Utazóügynök probléma

Az utazóügynök probléma (Travelling Salesman Problem - TSP) egy optimalizációs feladat, mely során egy utazónak minél rövidebb úton kell megtennie egy körutat egy adott ponthalmazon.

Precízen fogalmazva: Adott a bemeneten egy G=(V,E) (irányított) gráf,

$$n = |V(G)|, n > 2$$

az állomások száma (a kiindulási állomást beleértve),

$$m = |E(G)|$$

az állomások között futó elérhető utak száma.

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

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

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

az elérhető utak halmaza (Edge),

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

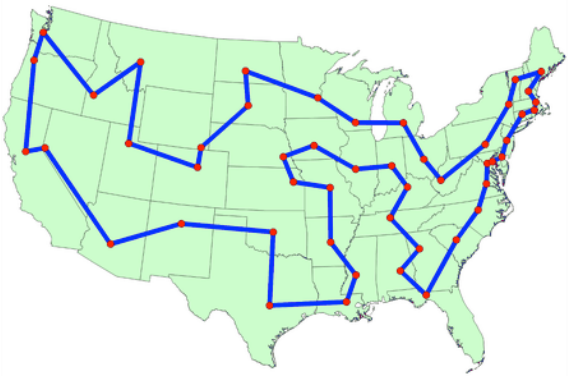
az élekhez rendelt költségfüggvény (Distance).

A kimenet a legkisebb költségű Hamilton-kör G-re, vagyis azon

$$R = (0, v_{i0}, v_{i1} ... 0)$$

bejárás, amely $\forall v_i \in V, v_i \in R$ mindegyik csúcsot tartalmazza, és költsége minimális [2].

A téma 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



2.1. ábra. Egy példa TSP végrehajtására az USA szárazföldi államai fővárosainak körbeutazása [3]

minél jobban minimalizálják 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 lapkára. Az iparban fontos a sebesség, ugyanis ha felére csökkentjük a beültetési időt, akkor akár duplaannyi terméket gyárthatunk le azonos idő alatt. Egy szerelőlemezre alkatrészek százai kerülhetnek, ami nagyon sokféleképpen rendezhető sorba. Természetes igényünk rövid idő alatt gyors útvonalat találni a beültetőfej számára. A TSP-re 2.1. ábrán látható egy látványosabb, vizuális szemléltető példa.

2.2. Járműútvonal-tervezési problémák

A járműútvonal-tervezési probléma (Vehicle Routing Problem - VRP) tekinthető a TSP általánosításának. A problémával korábban hallgatótársam, Tóth Márk Andor is foglalkozott, munkája számos helyen inspirált [2]. A problémát különböző megkötésekkel lehet feltenni az alkalmazás igénye alapján. Ezek lehetnek például:

- járművek maximális száma
- az egyes járművek szállítási kapacitása
- az egyes helyszínekre történő érkezési idő

A következőkben feltételezzük, hogy ha több jármű van, akkor azok egy közös kezdőpontból (0. pont, raktár, warehouse) indulnak. Útjuk során minden pontot legalább egyszer érinteniük kell a járműveknek, egyazon csúcsba nem szállíthat csomagot két autó. A 2.2. ábrán látható egy vizuális szemléltető példa.

Matematikai megfogalmazás : a problémát gráfokkal modellezhetjük.
Legyen

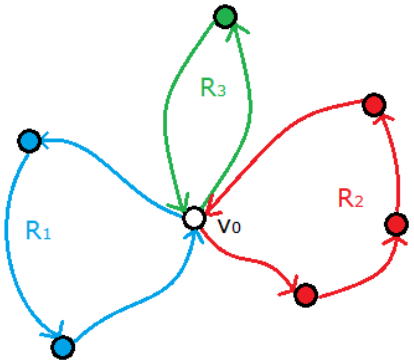
$$G = (V, E)$$

(irányított) gráf, n az állomások száma (a kiindulási állomást beleértve), m az állomások között futó elérhető utak száma, k az rendelkezésre álló járművek maximális száma.

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

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

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



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

az elérhető utak halmaza (Edge),

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

az élekhez rendelt 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),

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

az egyes állomások készenléti ideje,

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

az egyes állomások határideje. Értelemszerűen $T_{max}(v_i) > T_{min}(v_i)$. A járművek sebessége 1 (tetszőleges egység), az idő és távolság egysége ugyanaz.

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.

$$c(R_i) = \sum_{v \in R_i} c(v) \tag{2.1}$$

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 $\exists! R_j : v_i \in R_j$.
- egyik jármű sem szállíthat több árut a megengedettnél, vagyis $l_i \geqslant \sum_{v \in R_i} c(v)$
- a járművek mindegyik állomásra időben megérkeznek: $\forall v_i \in V, v_i \neq v_0$ esetén $T_{min}(v_i) \leqslant t(v_i) \leqslant T_{max}(v_i)$

2.3. Hangyakolónia optimalizáció

A 2.1. és 2.2. fejezetekben ismertetett problémákra az optimális megoldás megtalálása NP-nehéz feladat, tehát nagy csúcs- és élhalmaz mellett nem gazdaságos az eredmény kiszámítása. Annak érdekében, hogy a gyakorlatban használható algoritmust konstruáljunk, valamilyen közelítő megoldást érdemes használni a direkt eljárások helyett. A hangyakolónia optimalizáció (Ant Colony Optimization - ACO) egy heurisztikus alapelv, mely gráfbejárások optimalizálásához képes gyorsan, az optimálishoz nagyon közeli megoldásokat találni. Alkalmas a nagyfokú párhuzamosításra, ezért tökéletes választás az NVIDIA™ CUDA architektúrájával történő, GPU alapú adaptálásra.

Az eljárás a nevéből adódóan a hangyák (Formicidae) természetben is megfigyelhető élelemkeresési módszerén alapszik. Az első felfedező hangyák véletlenszerű útvonalakon haladva keresik az élelemhez vezető utat, majd ha sikerrel jártak, akkor a visszaúton feromonnal jelölik meg az útjukat. A többi hangya a szagokat követi, ezért könnyebben, nagy számban tudnak eljutni az elemózsiához. Ha még maradt étel, ők is visszatérve erősítik a feromon nyomokat. Utánpótlás hiányában annak erőssége idővel gyengül, ami modellezhető exponenciális lecsengéssel. Ez természetes módon biztosítja, 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, egyre kevesebb hangyát vonz.

Az algoritmus futása során nyilvántartunk egy az eredeti gráf topológiájával megegyező, de eltérő élsúlyozású feromongráfot. 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 az élsúly alapján "céltábla elv szerint", véletlenszerűen választ. Úgy kell elképzelni, mintha egy beszínezett darts táblára dobálnánk, és a különböző színekhez az elérhető csúcsok tartoznak, a geometriai valószínűségi mező szerint kisebb-nagyobb valószínűségekkel. Az egyes élek kiválasztásának valószínűsége

$$P_{i,j} = \frac{(\tau_{i,j})^\alpha (d_{i,j})^\beta}{\sum_{l \in N_i^k} (\tau_{i,l})^\alpha (d_{i,l})^\beta} \tag{2.2}$$

ahol N_i^k az algoritmus k-adik lépésében az i-edik csúcsból elérhető szomszédos csúcsok halmaza. Az α és β paraméterek a feromonok és élsúlyok figyelembevételét szabályozzák. A cél az, hogy minél nagyobb valahol a feromon, annál inkább akarjunk oda továbbmenni, illetve minél messzebb van egy adott pont, annál inkább el akarjuk kerülni. Jelen esetben elhanyagoltam az élhosszak egyenként külön figyelembe vételét, ezért $\beta = 0$. Továbbá az egyszerűség kedvéért legyen $\alpha = 1$. Ennek az lesz az előnye, hogy a 2.2. kifejezés nagymértékben leegyszerűsödik:

$$P_{i,j} = \frac{\tau_{i,j}}{\sum_{l \in N_i^k} \tau_{i,l}} \tag{2.3}$$

Miután minden hangya végigment egy úton (legeneráltunk 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. Valamilyen konstans szorzóra még szükség van a feromonértékek adott tartományba szabályozásához, ezért én még az addíciókat megszorzom a gráfban fellelhető átlagos bejárás hosszával. Így a hangsúly nem a konkrét hosszértékeken, hanem inkább az átlagoshoz vagy az optimálishoz viszonyított arányokon lesz.

Az éleken található feromon növelése után mindegyik élt exponenciális jelleggel csökkentem: minden feromon gyengül egy előre beállítandó, konstans szorzóval, ezzel veszem figyelembe a párolgást (ρ). Később úgy tapasztaltam, hogy egy $\rho \approx 0.75$ választás megfelelőnek bizonyult. Tehát az egyes iterációk végén a következő történik egy él feromonjával:

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

Az útkeresés közben mindig fel kell jegyezni az addig megtalált legjobb utat. Az ACO algoritmus egyik előnye, hogy több, hasonlóan jó alternatív utat is képes megtalálni. Ez például térképes útvonaltervezésnél lehet hasznos. Előfordulhat, hogy valami tőlünk független ok miatt a felhasználó egy objektíven nézve enyhén szuboptimális útvonalat akar inkább. Az eljárás javítása érdekében bevezettem, hogy az a hangya egy ún. "Jutalom szorzó"-t kap, aki minden korábbinál rövidebb utat talál. Ez azzal jár, a feromonjához adandó többlet a sokszorosára, például százszorosára változik, így a következő iterációban sokkal nagyobb valószínűséggel fog arra menni a jövő hangyája. A jutalmazási rendszerem rossz beállítások mellett kezdetben félreviheti a hangyákat egy szuboptimális, de lokálisan minimális útvonal felé. Ha ilyen jelenség tapasztalható, akkor át lehet állítani, hogy hányadik iterációtól kezdve kaphassanak az útvonalak "Jutalom szorzó"-t.

Az eljárás adott számú iteráció után leáll, az eredmény lehetőleg az addig megtalált legrövidebb útvonal. A konkrét implementáció során a 4.2. fejezetben látni fogjuk, hogy a helyzet bonyolódhat.

2.4. Alternatív megoldások

Dolgozatomban főleg a Hangyakolónia algoritmussal foglalkoztam, de az nem azt jelenti, hogy ez az egyetlen járható út. Nézzünk meg néhány alternatív megoldási módszert.

2.4.1. „Brute force" algoritmus

A „Brute force" algoritmus lényege, hogy minden lehetséges bejárást megvizsgálunk, és kiválasztjuk a legrövidebb, a konkrét probléma feltétel(rendszer)ének eleget tevő esetet. Magyarul talán "nyers erő" módszerének mondhatnánk, ami kicsit szerencsétlenül hangzik, ezért én a továbbiakban az angol elnevezésével hivatkozok rá. **Ha n db csúcs**ból álló teljes gráfot nézünk, mint ahogy az a valóságban igen gyakori, **akkor n! különböző lehetséges bejárás**t kell összehasonlítani. Kis n esetén még csak-csak elfogadható ez a módszer, viszont ha már $n = 48$ db csúcsunk van, mert szeretnénk bejárni TSP szerint az Amerikai Egyesült Államok 48 összefüggő államának fővárosait (később lesz rá példa), a vizsgálandó esetek száma felugrik $48! \approx 1.24 * 10^{61}$-re. Tegyük fel, hogy csúcskategóriás, 5 GHz-en pörgő szuperszámítógépünk képes átlagosan 1 órajelciklusonként ($2*10^{-10}s$ időközönként, nagyjából lehetetlenül gyorsan) kiszámolni egy út hosszát, még így is kb. $2.5 * 10^{51}s \approx 8 * 10^{43}$ évig vizsgálhatnánk az eseteket. Egy SMD beültetőgép a szerelőlemezre akár alkatrészek százait pakolhatja fel, brute force módszerrel lehetetlen lenne megmondani, hogy milyen sorrendben haladjon. Ha szeretnénk véges időn belül megoldani a problémát, akkor ravaszabbnak kell lennünk.

2.4.2. Held - Karp algoritmus

A Held-Karp algoritmust M. Held és R. Karp alkották meg 1962-ben [12]. Akkoriban még csak gyerekcipőben a számítástudomány és az informatika, ekkor készültek el az első számítógépek. Módszerük szerint egyesével, rendezetten szúrnak be csúcsokat egy egyre növekvő ponthalmazba. Hasonlít a beillesztéses rendezésre, csak kicsit komplexebb. Kihasználja, hogy a stack (magyarul: verem) és a stackpointer megjelenésével a korai számítógépek is képesek voltak már rekurzív programvégrehajtásra.

Jelöljük a csúcsokat $V = (v_1, v_2, \dots v_n)$ -nel, v_1 önkényesen kijelölhető kezdőpont. Legyen S halmaz a csúcsok valamely, a kezdőcsúcsot nem tartalmazó részhalmaza: $S \subseteq \{v_2, ...v_n\}$. Legyen g(S,e) v_1-ből az S összes elemén keresztül az e $\neq v_1$ csúcsban végződő legrövidebb út hossza. Az u-ból a v-be mutató él költsége d(u,v). Lépésenként kiszámítjuk a g(S,e) értékeket kezdve a kis S-ekre.

Példák:

- $\forall e : g(\emptyset, e) = d(1, e)$
- $g(\{2\}, 4)$ csak az összhossza az $1 \rightarrowtail 2 \rightarrowtail 4$ útnak
- $g(\{2, 3\}, 4)$ az ($1 \rightarrowtail 2 \rightarrowtail 3 \rightarrowtail 4$) és ($1 \rightarrowtail 3 \rightarrowtail 2 \rightarrowtail 4$) utak rövidebbikének a költsége

Amikor már 3 vagy több pontot tartalmaz az S halmaz, a lehetséges utak száma drasztikusan megnő, de egy ügyes trükk felhasználásával csak néhányat kell figyelembe venni a legrövidebb út keresése érdekében. Vegyük észre, hogyha az $(1 \rightarrowtail 4 \rightarrowtail 3 \rightarrowtail 2)$ út rövidebb, mint az $(1 \rightarrowtail 3 \rightarrowtail 4 \rightarrowtail 2)$, akkor az $(1 \rightarrowtail 4 \rightarrowtail 3 \rightarrowtail 2 \rightarrowtail 5)$ út is rövidebb lesz, mint a $(1 \rightarrowtail 3 \rightarrowtail 4 \rightarrowtail 2 \rightarrowtail 5)$.

Általánosan: tegyük fel, hogy a k-adik lépésben $S = \{s_1, ...s_k\}$. Jelöljük $1 \leqslant i \leqslant k$-ra $S_i = S - \{s_i\} = \{s_1...s_{i-1}, s_{i+1}...s_k\}$. Ha a v_1-ből a legrövidebb út e-be S-en keresztül úgy vezet, hogy annak s_i az utolsó előtti eleme, akkor következik, hogy v_1-ből a legrövidebb út s_i-be S_i-n keresztül vezet. Ez azt jelenti, hogy a k-adik lépésben elég k db S halmazt továbbvinnünk, mert csak azok lehetségesek legrövidebb utat adni.

2.4.2.1. Az eljárás Értékelése

A Held-Karp algoritmus exponenciális idejű, $O(2^n n^2)$, ami nagyságrendekkel jobb, mint a brute-force módszer az ő $O(n!)$ faktoriális idejével. Előnye a Hangyakolónia algoritmussal szemben, hogy determinisztikus algoritmus révén a végén mindig a legrövidebb utat adja eredményül. Komoly hátránya, hogy rekurziót alkalmaz, ami kedvezőtlen a programvégrehajtás szempontjából: sok rekurzív függvényhívás ugyanis megterheli a stacket, sok felesleges másolás történik. Belátható, hogy ha különböző, adott esetekben bonyolult feltételekkel keresünk utakat, akkor az algoritmus elveszíti alapelvét, nem elég mindig az előző ciklus legjobbjaiból kiindulni. Ezzel a problémával az ACO is szembesül, de kevésbé van rá kihatással. Erre még később visszatérek a 5.3.3 fejezetben, ahol a feltételek bonyolódásával foglalkozom.

3. fejezet

Technológiai háttér

Ebben a fejezetben szeretném ismertetni a GPU porgramozáshoz felhasznált szoftver- és hardveregyüttest. Először a grafikus segédprocesszoron történő általános célú programozást tárgyalom, majd bemutatom az Nvidia^{TM} által erre kifejlesztett párhuzamos számítási platformot, a CUDA keretrendszert. Részletezem a megértéshez szükséges fontosabb fogalmakat, valamint röviden bemutatom, hogy lehet Visual Studio segítségével CUDA platformon C++ nyelven többszálú programot fejleszteni. A fogalmak összegyűjtéséhez nagymértékben ihletett nyújtott többek között Király Gábor diplomamunkája, mely hasonló témakörrel foglalkozik. [16]

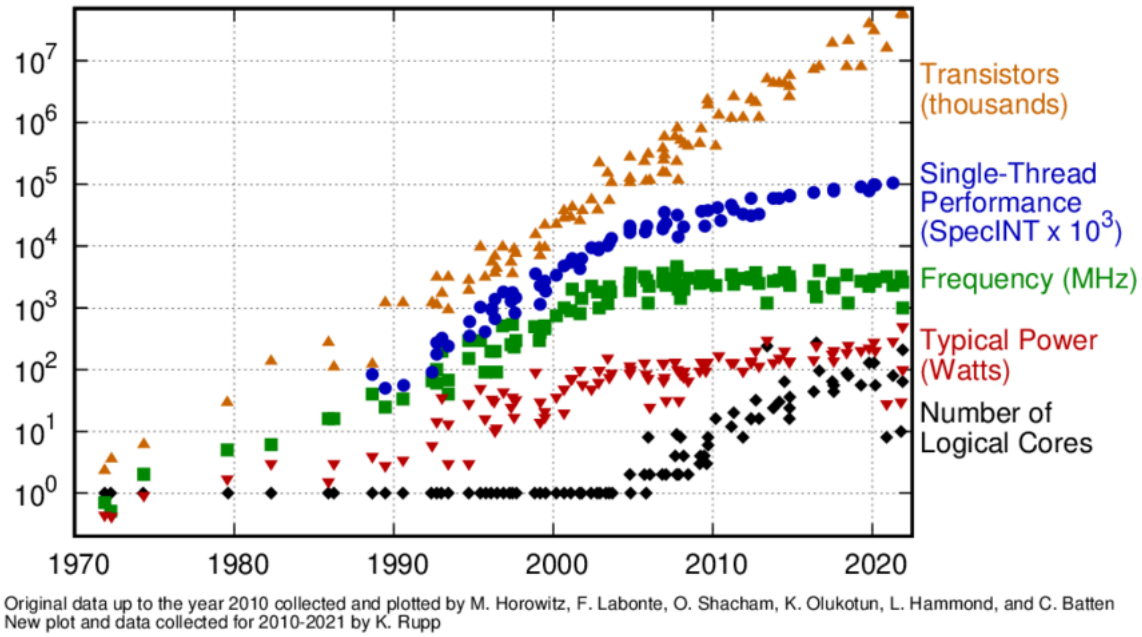
3.1. GPGPU

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. [16] Korábban a GPU-t azért találták fel, hogy a grafikus felületek elterjedése után a renderelési, 2D-s vagy 3D-s megjelenítési feladatok terén tehermentesítse a CPU-t. Később kiderült, hogy a GPU alkalmas általánosabb megközelítésekre is, bizonyos aritmetikai utasítások kifejezetten hatékonyan tudnak működni rajta.

3.1.1. Motiváció

Az 1980-as években megjelentek az első személyi számítógépek (PC-k), melyek központi feldolgozóegységei (CPU) kezdetekben néhány MHz-es belső órajellel működtek. Akkor az volt a számítástechnikai fejlesztőmérnökök fő eszköze a számítási gyorsaság növelésére, hogy az órajelfrekvenciát növelték. Ez frekventáltabb utasítás-végrehajtást biztosított, és evidens volt, hogy a nagyobb frekvencia nagyobb számítási erővel is jár. Számos kiváló mérnöki megoldás született, ezek közül talán az egyik legjelentősebb találmány a fáziszárt hurok (Phase-Locked Loop - PLL). A PLL egy olyan Szabályozható hurok, amely (a részleteket mellőzve, nem tárgya dolgozatomnak) egy bemeneti referenciafrekvenciát tud megsokszorozni. Nélküle gyakorlatilag képtelenség lett volna felhasználói szinten 50-60 MHz fölé menni a személyi számítógépek belső órajelénél. Nagyjából 30 évvel később elérték a hardverfejlesztők, hogy a legtöbb asztali processzor órajele 1GHz és 4GHz között legyen képes működni, ez az eredeti PC-k operatív belső frekvenciájának több, mint az ezerszerese. Napjainkban változás látható a fejlesztési trendekben, ugyanis az órajelnövelést a processzorok hődisszipációja felülről korlátozza. Egyelőre nem tűnik könnyen lehetségesnek 5GHz fölé menni úgy, hogy közben az eszköz helyes működése garantálható legyen. A különböző hűtési technológiák (léghűtés, vízhűtés) bizonyos fokig tudnak javítani a sebességen, viszont nagyságrendeket ugrani velük sem lehetséges. A számítógépgyártók

mai napig új, alternatív megoldásokat keresnek a számítási teljesítmény növelésére. Napjainkban a kutatásoknak két nagy témája van. Egyik a kvantumszámítógépek témaköre, amit dolgozatomban nem részletezek. Másik aktívan vizsgált lehetőség a párhuzamosítás minél több szálon. Már a CPU-k fejlesztésénél is megfigyelhető, hogy inkább a minél több processzormag telepítése az iparági trend.



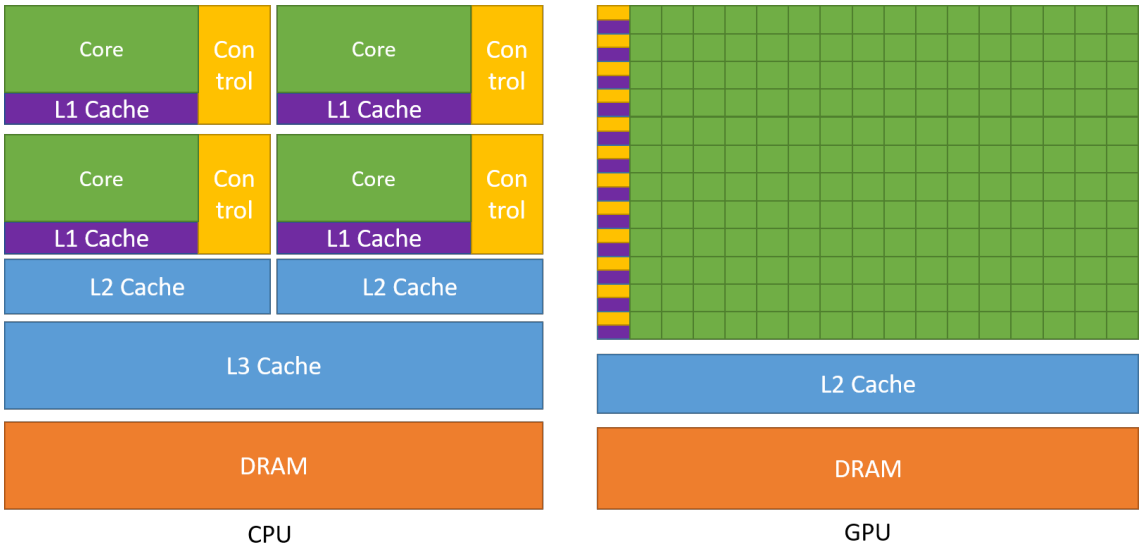
3.1. ábra. Látható, hogy kb. 2010-re befejeződött a CPU-k órajelfrekvencia-növekedése, helyette egyre nőni kezdett az egy házon belüli magok száma. [1]

Párhuzamosításra legalkalmasabb a grafikus segédprocesszor, a GPU, hiszen felépítéséből adódóan erre készült. Amíg a CPU feladata az, hogy műveletek egy adott szekvenciáját, és annak minden utasítását a lehető leggyorsabban hajtsa végre, addig a GPU célja minél több szál (akár több ezer, vagy több tízezer) párhuzamos futtatása. A videokártyák előnye akkor válik láthatóvá, ha ugyanazt az utasítást több nagy adattömbön kell végrehajtani. Ez az úgynevezett SIMD megközelítés (Single Instruction Multiple Data). [16] Az 3.2. ábra szemlélteti, hogy a CPU-hoz képest a GPU-n arányaiban több tranzisztor van adatfeldolgozásra rendelve, cserébe a gyorsítótárazás és a folyamatvezérlés (feltételkiértékelések, ciklusszervezések) kisebb hangsúlyt kapott.

A videokártya sokkal nagyobb utasítás-áteresztőképességet, valamint memóriasávszélességet biztosít, mint a CPU hasonló ár és energiafogyasztás mellett. Egyéb számítási eszközök, mint az FPGA-k is lehetnek nagyon energiatakarékosak, viszont azok sokkal kevésbé rugalmasan programozhatóak, mint a GPU-k, ezért a fejlesztési idő sokkal hosszabb lesz és az alkalmazást nehezebb karbantartani. [7]

3.2. CUDA

Többféle keretrendszer is megvalósítja a GPGPU szabta alapelveket. Munkám során a CUDA (Compute Unified Device Architecture) rendszerét használtam. A CUDA egy, az NVIDIA által fejlesztett párhuzamos számítási platform és API (felhasználói interfész), amely szoftveres támogatást nyújt az ezzel kompatibilis grafikus feldolgozóegységek általános célú programozására [16].



3.2. ábra. Látható, hogy a gyorsítótárak és a vezérlés rovására nőtt az adatfeldolgozásra szánt tranzisztorok számára. Ez alkalmas lebegőpontos műveletek nagyfokú párhuzamosítására. [7]

A programozás C vagy C++ nyelven történhet, melyet minimális nyelvi kiegészítésekkel bővítettek, többek között a szálkezelés rendszerszintű használata érdekében. A CUDA programozás tanulásához elérhető egy felettébb kiterjedt dokumentáció a gyártó weboldalán, melyet folyamatosan frissítenek. [7]

3.2.1. Programozási modell

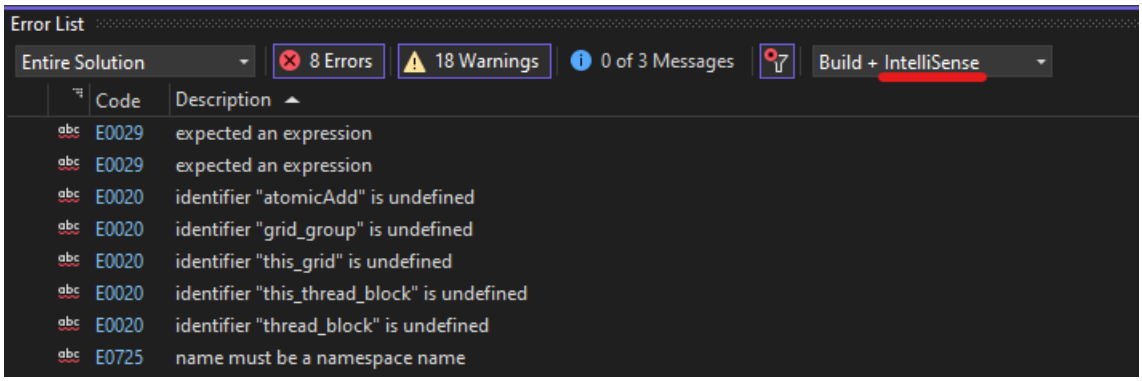
A továbbiakban összefoglalom a legfontosabb fogalmakat úgy, hogy ismertetem, azok hogyan lettek megvalósítva C++ programozási nyelven.

3.2.1.1. Kernel, és a többi függvénytípus

A programozó speciális függvényeket definiálhat, melyeket kernelnek nevezünk. A kernel létesít kapcsolatot a CPU (host) és GPU (device) között. A CPU hívja meg a függvényt, amit aztán a GPU hajt végre, tehát a kernelfüggvény törzse a videokártyán fut végig. Minden egyes kernel példányt egy számára megadott szál hajt végre. A kernel a "___*global*___" kulcsszóval definiálható. Ezt a függvény fejléce elé kell írni, ettől fogja tudni a szoftverkörnyezet fordítóprogramja, a compiler, hogy mostantól GPU kódként kell értelmezze a programot. Minden, a kernelt végrehajtó szál egy egyedi thread azonosítót kap, mely a beépített threadIdx változón keresztül érhető el a függvényen belül.

Egyéb kulcsszavak is léteznek. Egyik a "___*host*___", ami azt jelzi, hogy CPU által hívott, majd ugyanúgy általa futtatandó kódrészlet következik. Ha nem adunk meg egy függvény elé kulcsszót, akkor azt a preprocesszor host függvénnyé írja át, tiszta CPU kódként értelmezi, mintha nem is lenne a szoftverkörnyezet mögött a CUDA platform. Másik használható jelző a "___*device*___", amely tisztán GPU függvényt jelez. A két kulcsszó vegyíthető: amennyiben azt írjuk, hogy "___*device*___ ___*host*___", a fordító ezt minden egyes híváskor a végrehajtó saját kódjának tekinti, vagyis nem hajt végre vezérlésátadást az eszközök között. Hasznosítható például függvénykönyvtárak GPU-ra kiterjesztésére.

Azt, hogy a kernelt felhívásakor hány CUDA szálon szeretnénk futtatni, az új nyelvi elemként megjelenő $<<<\cdots>>>$ végrehajtási konfiguráció szintaxissal tudjuk specifikálni. Sajnos a Visual Studio még szintaxishibaként kezeli a mostani, CUDA 12.3 verzióban, ezért a programozónak érdemes odafigyelni, hogy milyen IntelliSense hibaüzeneteket vesz figyelembe. A 3.3. ábrán például olyan fogalmakra jelez a fejlesztői környezet, amelyek a keretrendszer adta bővítményekben vannak definiálva.



3.3. ábra. Bizonyos hibaüzeneteket azért kaphatunk, mert a Visual Studio által használt Intellisense nem ismeri a CUDA által tett kiegészítéseket. Ezeket figyelmen kívül lehet hagyni

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:

```
#define N 1024

// 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);
  ...
}
```

Megfelelő CPU programkörnyezet hozzáadásával ellenőrizhető, hogy a példában adott kernel tényleg helyes eredményt ad.

3.2.1.2. Szálkezelés

Ahhoz, hogy CUDA programunk megfelelően működjön, a szálainkat rendszereznünk kell.

Most bemutatom, hogy CUDA API-ban milyen típusú szálak léteznek, és ezeket hogyan lehet különböző szintű szinkronizációkba hozni. Ezen fejezetrészhez intenzíven tanulmányoztam az NVIDIA hivatalos fórumának bejegyzéseit, amit mivel az ott dolgozó fejlesztőmérnökök szerkesztenek, relevánsnak tekintettem. Az egyik leghasznosabb cikket a "Cooperative Groups" nevű bővítményről találtam, a példákat belőle idézem [4].

Hatékony párhuzamosított programnak három alappillére van: [16]

1. *Szálak csoportosításának hierarchiája*: MultiGrid - Grid - Blokk - Szál
2. *A szálak között megosztott memória: shared memory*
3. *A szálak közti szinkronizáció*

Foglalkozzunk jelen esetben a 3. ponttal, a szinkronizálással. Szinkronizáció szükséges ahhoz, hogy elosztott programunk biztonságos, fenntartható és moduláris legyen. A CUDA 9 bevezette az ún. **Kooperatív csoport** nevű absztrakciót (angolul Cooperative Groups) [5], amely erős támogatást nyújt a kerneleknek ahhoz, hogy dinamikusan alakíthassanak ki csoportokat a szálak között.

Korábban a CUDA API egy nagyon egyszerű, de csak speciális esetekben működő megoldást biztosított a szinkronizáció kérdésére, a blokkon belüli "barrier" szinkronizációt: A "___syncthreads()" függvény addig nem engedte tovább futni a szálakat, amíg **a blokkon belül** minden még futó szál el nem jutott az adott pontig. Belátható, hogy nagy szál szám mellett ez nem elég, ugyanis egy blokkon belül jelenlegi GPU-kon legfeljebb 1024 szál futhat. Ha mi több, mint 1024 threadből álló programot írnánk, azaz több Streaming multiprocessor (SM) futna egymással párhuzamosan, akkor ezek összehangolását eddig nem tudtuk volna megfelelő szoftveres támogatással elvégezni. Másik probléma az, hogy ha a szálainknak csak egy kis, adott számú (például 4 vagy 32, de tipikusan 2-hatvány) részhalmazát akarjuk összehangolni akkor korábban azt sem tudtuk szépen megoldani.

Leegyszerűsítve a Cooperative Groups egy API Support Package, ami szálak csoportosítását és szinkronizálását segíti CUDA programokban. A Package nagy része az összes CUDA 9-el kompatibilis GPU-ra működik, azaz Kepler és későbbi architektúrákkal (Compute Capability 3.0+) kompatibilis.

Ahhoz, hogy élhessünk a csomag nyújtotta lehetőségekkel, be kell illeszteni az alábbi headert .cu vagy .cuh kiterjesztésű fájlunk fejlécébe.

```
#include <cooperative_groups.h>
```

A típusok és interfészek a "cooperative_groups" C++ névtérben vannak definiálva, így mindig prefixként ki kell írjuk, hogy "cooperative_groups::", vagy betöltjük a névteret a "using" direktívával. Én a munkám során a "using" megoldást választottam.

```
using namespace cooperative_groups; // Névtér betöltése
using cooperative_groups::thread_group; // stb.
namespace cg = cooperative_groups; // Használhatunk rövid aliast is
```

Thread csoportok Az egyik legfontosabb típus a csomagon belül a "thread_group" típus, ami threadek, azaz szálak csoportját tudja kezelni. Ezt örökölteti le az összes, később tárgyalandó csoport objektum. Alapvető függvények rájuk:

- Megkaphatjuk a csoport méretét, azaz a benne futó szálak számát a size() metódussal. Használható túlcímzés elleni védelemre

```
unsigned size();
```

- Megkaphatjuk a hívó thread indexét (0 és size()-1 közötti) a thread_rank() metódussal

```
unsigned thread_rank();
```

- Megvizsgálhatjuk a csoport érvényességét az is_valid() függvénnyel

```
bool is_valid();
```

Thread csoportokon végrahajtható kollektív (egyenhatású) műveletek akár kiszedhető - A thread csoportok megadják a lehetőséget, hogy együttesen hajtsunk rajtuk végre műveleteket. Legegyszerűbb operációink egyike a szinkronizálás, ami annyit tesz, hogy a csoport tagjait nem engedi túl egy műveletsoron addig, míg minden tagja el nem jut az adott pontig. Az összes thread csoport fajta támogatja a szinkronizálást, azonban mindegyik kicsit másképp.

Egy adott g csoporthoz tartozó szálakat a kollektív sync() metódussal, vagy g-re a cooperative_groups::synchronize() függvényt meghívva szinkronizálhatjuk. Ezek a már korábban emlegetett barrier szinkronizációt hajtják végre.

```
g.sync();               // g szinkronizálása
cg::synchronize(g); // ekvivalens megoldás
```

A következő egyszerű példában egy device (GPU-n futtatott) függvény szerepel kooperatív csoportok felhasználásával [4]. Amikor egy thread csoport meghívja, a szálak kiszámolják a szálanként kapott értékek összegét a csoporton belül.

```
using namespace cooperative_groups;
__device__ int reduce_sum(thread_group g, int *temp, int val)
{
  int lane = g.thread_rank();

  // Minden iteráció megfelezi az aktiv szálak számát
  // Minden szál hozzáadja a saját sum[i]-jét sum[lane+i]
  for (int i = g.size() / 2; i > 0; i /= 2)
  {
    temp[lane] = val;
    g.sync(); // wait for all threads to store
    if(lane<i) val += temp[lane + i];
    g.sync(); // wait for all threads to load
  }
  return val; // note: only thread 0 will return full sum
}
```

Thread Blokk Az első thread csoport fajta a Thread blokk. A Cooperative Groups azért vezette be ezt az adattípust, hogy explicit reprezentálja a CUDA programozás azonos nevű, egyik fontos fogalmát. A szálak egy-, kettő-, vagy háromdimenziós logikai egységbe szervezhetők, amit **blokk**nak nevezünk. Ez a megoldás egy természetes módot nyújt arra, hogy vektorok vagy mátrixok elemein hajtsunk végre számításokat. Az egy blokkba tartozó szálak számát havdveres megfontolások felülről korlátozzák: mivel ezeknek a threadeknek közös processzormagon kell futniuk és a mag korlátos memória-erőforrásain kell osztozniuk, nem fogalhatnak el túl nagy helyet. A jelenlegi GPU-k egy blokkban legfeljebb 1024 thread futtatását támogatják, viszont a kernel több egyenlő méretű blokkban futtatható, ezért a szálak száma egyenlő a blokkonkénti szálak száma és a blokkszám szorzatával. [16] Egy thread blokk példánya az alábbi módon inicializálható:

```
thread_block block = this_thread_block();
```

Azon threadek, melyek beépített CUDA blockIdx értékei megegyezőek, ugyanazon thread blokkba tartoznak. A blokkok szinkronizálása nagyon hasonló a korábban említett __syncthreads() metódushoz. A következő kódok mind ugyanolyan hatást érnek el: (feltéve, ha a thread blokk összes szála elér odáig)

```
__syncthreads();
block.sync();
cg::synchronize(block);
this_thread_block().sync();
cg::synchronize(this_thread_block());
```

A ”thread_block” adattípus kiterjeszti a ”thread_group” interfészt két, blokk-specifikus tagfüggvénnyel. Ezek megfeleltethetőek a CUDA API blockIdx és threadIdx tagváltozóinak.

```
dim3 group_index(); // 3-dimenziós blokk index griden belül
dim3 thread_index(); // 3-dimenziós thread index blokkon belül
```

Az alábbi egy egyszerű kernel, ami a korábban látott ”reduce_sum()” device függvényt használja egy tömb elemeinek összegzésére. Gyorsítás érdekében több, párhuzamosan elvégzett rész összeadással kezd (thread_sum()). A kernel thread blokkokat használ az összegzésre, és az atomikus ”atomicAdd()” metódussal adja össze a részösszegeket.

```
__device__ int thread_sum(int *input, int n)
{
  int sum = 0;

  for(int i = blockIdx.x * blockDim.x + threadIdx.x;
  i < n / 4;
  i += blockDim.x * gridDim.x)
  {
    int4 in = ((int4*)input)[i];
    sum += in.x + in.y + in.z + in.w;
  }
  return sum;
}

__global__ void sum_kernel_block(int *sum, int *input, int n)
{
  int my_sum = thread_sum(input, n);

  extern __shared__ int temp[];
  auto g = this_thread_block();
  int block_sum = reduce_sum(g, temp, my_sum);

  if (g.thread_rank() == 0) atomicAdd(sum, block_sum);
}
```

Az előbbi függvényt futtathatjuk például egy 16 millió elemszámú tömb kiértékelésére.

```
int n = 1<<24;
int blockSize = 256;
int nBlocks = (n + blockSize - 1) / blockSize;
int sharedBytes = blockSize * sizeof(int);

int *sum, *data;
cudaMallocManaged(&sum, sizeof(int));
cudaMallocManaged(&data, n * sizeof(int));
std::fill_n(data, n, 1); // Adatok feltöltése
cudaMemset(sum, 0, sizeof(int));

// Kernelhívás
sum_kernel_block<<<nBlocks, blockSize, sharedBytes>>>(sum, data, n);
```

Csoport Partíciók A Cooperative Groups kódcsomag megengedi, hogy már meglévő csoportjainkat rugalmasan particionáljuk új, kisebb egységekre, ezáltal kiegyensúlyozhatjuk párhuzamos algoritmusainkat. A cooperative_groups::tiled_partition() függvény felosztja az adott thread blokkot több részegységre. Következik egy példa, amely a teljes thread blokkot 32 szálú részegységekre bontja. A 32 gyakori felosztás, ugyanis a warpok (a CUDA szoftvermodell egyik alapfogalma, most nem térek rá ki részletesebben) méretével megegyező.

```
thread_group tile32 = cg::partition(this_thread_block(), 32);
```

Egy másik példa partíciók létrehozására, amit az NVIDIA CUDA Toolkit Documentation (C.4.2.1. Thread Block Tile) [6] említ.

```
/// The following code will create two sets of tiled groups, of size 32 and 4 respectively:
/// The latter has the provenance encoded in the type, while the first stores it in the handle
thread_block block = this_thread_block();
thread_block_tile<32> tile32 = tiled_partition<32>(block);
thread_block_tile<4, thread_block> tile4 = tiled_partition<4>(block);
```

3.2.2. Moduláris programszerkesztés

A kooperatív csoportok használata nem csak gyors, de hasznos is tud lenni. A kódcsomag ereje a modularitás, amikor a csoportot explicit átadjuk függvények között, és konzisztensen hivatkozunk annak méretére. Ez segít kritikus versenyhelyzetek, illetve holtpontok elkerülésében, mert nem teszünk hibás következtetéseket elágazó függvényhívások között. Az alábbi egy elkerülendő példa hibás szinkronizálásra.

```
__device__ int sum(int *x, int n)
{
  ...
  __syncthreads();
  ...
  return total;
}

__global__ void parallel_kernel(float *x, int n)
{
  if (threadIdx.x < blockDim.x / 2)
  sum(x, count);  // hiba: threadek fele nem hívja meg a függvényt
  // __syncthreads() => holtpont
}
```

A példában a threadeknek csak a fele hívja meg a sum() függvényt, ami tartalmaz "*__syncthreads()*" utasítást. A thread blokk nem minden threadje éri el a "*__syncthreads()*"-et, így holtpont alakul ki, mivel a "*__syncthreads()*" gátat képez addig, míg minden blokkon belüli thread el nem éri. Amennyiben alkalmazzuk a kooperatív csoportok adta lehetőségeket, ez a hiba nehezebben elkövethető. Fontos átadni a csoport típust, mint paramétert a hívandó függvénynek, és ekkor azon a csoporton végzünk szinkronizációt. Alapszabály, hogy tiszta GPU függvényben nem hivatkozunk közvetlenül a kernel szálaira, különben hasonló hibákba ütközhetünk.

```
// Nyilvánvaló, hogy a teljes blokk meg kell hivja
// Van benne sync utasítás, ami különben holtpontot okozna
__device__ int sum(thread_block block, int *x, int n)
{
  ...
  block.sync();
  ...
  return total;
}

__global__ void parallel_kernel(float *x, int n)
{
  sum(this_thread_block(), x, count); // nincs elágazó függvényhívás
}
```

3.2.3. Grid csoport

Ez a csoport objektum reprezentálja az összes szálat, melyek közös grid alatt futnak. A *sync()* operációt kivéve minden API elérhető mindig, azonban ahhoz, hogy griden belül szinkronizálhassunk, a speciális "cooperative launch API" használatára van szükség. Egy grid csoport példánya az alábbi módon inicializálható:

```
grid_group grid = this_grid();
```

A "grid_group" adattípus kiterjeszti a "thread_group" interfészt két, blokk-specifikus tagfüggvénnyel. Ezek megfeleltethetőek a CUDA API blockIdx és threadIdx tagváltozóinak.

```
dim3 block_index();  // 3-dimenziós blokk index griden belül
dim3 thread_index(); // 3-dimenziós thread index blokkon belül
```

3.2.3.1. Teljes Grid csoporton belüli szinkronizáció

A kooperatív csoportok bevezetése előtt a CUDA programozási modellja csak thread blokkon belüli szervezésre nyújtott natív támogatást. A régebbi gyakorlat az volt, hogy a kernelt felbontottuk több kisebb alkernelre, majd azon pontokon, ahol grid szintű szinkronizációra vágytunk, befejeztük az adott alkernelt, és hívtuk az újat. Ennek a módszernek "CPU Szinkronizáció" vagy "Implicit Szinkronizáció" a neve.

A módszer több szempontból problémás. Először is egy GPU kernel hívása sok erőforrást igényel. Sok egymás utáni hívás miatt lassabb lesz a program, *ad absurdum* jobban járunk, ha bele sem vágunk a GPU programozásba, és az egész kódot tisztán CPU-ra írjuk. Másrészt ha az eredeti függvényben ciklusiterációnként akarnánk szinkronizálni, akkor Implicit szinkronizációs módszer mellett a kernelhívásokat kéne CPU cikluson belülre helyezni. Ez még jobb esetben átláthatatlan és fenntarthatatlan kódot eredményez, de rosszabb esetben a grafikus kártya meghibásodását is okozhatja a sűrű kernelhívás miatt.

A probléma tehát adott, de nézzük, hogy a kooperatív csoportok hogyan jelentenek erre megoldást.

Ahhoz, hogy grid csoporton belül szinkronizáljunk, első ránézésre elég a *grid.sync()* függvényt használnunk, mint ahogy azt a thread blokkon belül is tettük.

```
grid_group grid = this_grid();
grid.sync();
```

A főbb különbséget ott tapasztaljuk, amikor a kernel hívására kerül sor. A szokásos < < <...> > > konfigurációs szintaktika helyett a CUDA runtime API hívást kell végrehajtani (vagy annak driver megfelelőjét) [8]

Többféle módszer van a megfelelő több blokkos API hívásra, ezek közül nézzünk párat. Ahhoz, hogy biztosítsuk a thread blokkok megfelelő együttműködését a GPU-n, a blokkok száma előre megfontolandó. Ahhoz, hogy annyi blokkot futtassunk, ahány Streaming Multiprocessor (SM) van beépítve a rendszerbe, az alábbi kód alkalmas:

```
int device = 0;
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, device);
// initialize , then launch
cudaLaunchCooperativeKernel((void*)my_kernel, deviceProp.multiProcessorCount,numThreads,args);
```

Ajánlott megvizsgálnunk, hogy a kernel hívásához mekkora maximális blokkPerSM aránnyal dolgozhatunk (legfeljebb hány blokk fér el egy SM-en). Tegyük, fel, hogy szükségünk van 128 szálra, akkor így járhatunk el.

A CUDA documentation az alábbi példakódhoz hasonló eljárást javasol:

```
/// This will launch a grid that can maximally fill the GPU, on the default stream with kernel
    arguments
int numBlocksPerSm = 0;
// Number of threads my_kernel will be launched with
int numThreads = 128;
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, dev);
cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocksPerSm, my_kernel, numThreads, 0);
// launch
void *kernelArgs[] = { /* add kernel args */ };
dim3 dimBlock(numThreads, 1, 1);
dim3 dimGrid(deviceProp.multiProcessorCount*numBlocksPerSm, 1, 1);
cudaLaunchCooperativeKernel((void*)my_kernel, dimGrid, dimBlock, kernelArgs);
```

Érdemes futtatás előtt ellenőrizni, hogy grafikus kártyánk egyáltalán támogatja-e a grid szinkronizációt. Ennek komoly feltételei vannak:

- Csak 6.0 compute capability eszközök támogatottak
- Támogatott Windows operációs rendszer - Aktuális verziók: 8.1/10/11 [14]
- MPS ellátott Linux rendszer esetén csak 7.0 compute capability eszközök támogatottak

Az alábbi kód a cudaDevAttrCooperativeLaunch eszköz attribútumot vizsgálja meg.

```
int dev = 0;
int supportsCoopLaunch = 0;
cudaDeviceGetAttribute(&supportsCoopLaunch, cudaDevAttrCooperativeLaunch, dev);
```

Ez 1-be állítja a supportsCoopLaunch flaget ha a művelet támogatott a 0-s eszközön (egy grafikus kártyás rendszeren ez az alapértelmezett eszköz).

3.2.3.2. Program felkészítése több blokkos futtatásra

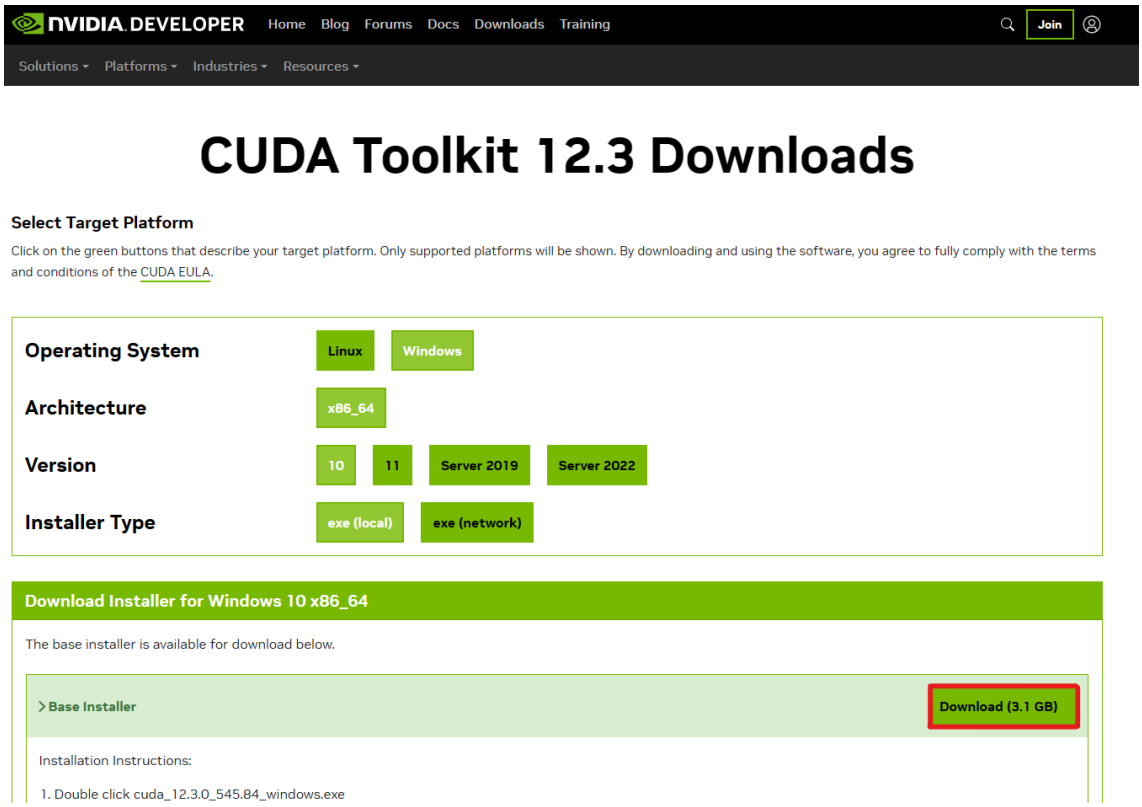
Mint azt már tárgyaltuk, bizonyos thread szám felett (jelenlegi GPU-kon 1024) a kerneleket már több blokkon szükséges futtatni hardveres okokból kifolyólag. Amennyiben elég egy blokk használata, márpedig számos alkalommal ez a helyzet, ki tudunk használni olyan trükköket gyorsításra, mint például a gyors megosztott memória használata a globális device memória helyett vagy a warpok hatékonyabb kihasználása. Előnyös lehet, ha algoritmusunkat egy és több blokkos használatra is megírjuk. Így képesek leszünk kezelni nagy thread számokat is, míg ki szám esetén kihasználjuk a lehetséges optimalizálási tulajdonságokat.

3.2.4. CUDA használata Visual Studio alatt

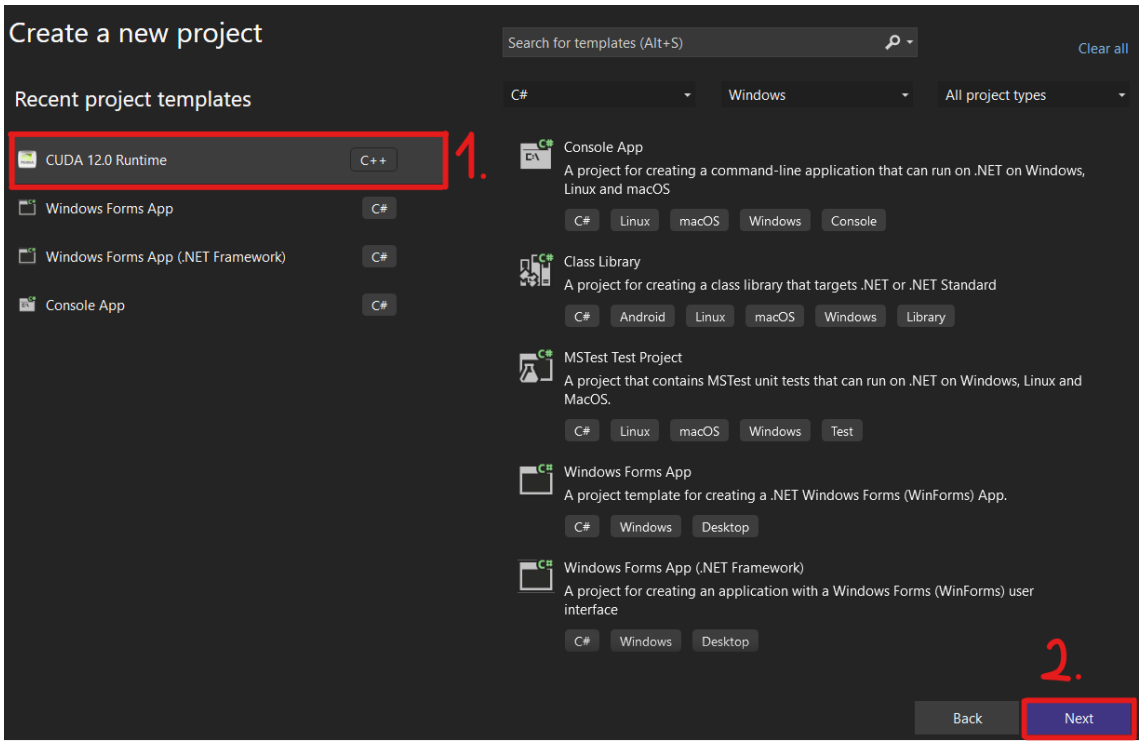
Szeretnék röviden leírást nyújtani az első CUDA nyelven megírt program létrehozásához.

CUDA Extension letöltése A gyártó bővítményt adott ki, mely a Visual Studio nevű fejlesztői környezetbe importálható. A CUDA legfrissebb verziója [az alábbi webhelyen] tölthető le (A link később változhat). Itt lehet tájékozódni a program használati feltételeiről is. Főbb információk: néhány (3-4) GB tárhelyre, illetve 64 bites Linux vagy Windows operációs rendszerre van szükség.

Új projekt létrehozása Telepítés után ha új projekt létrehozását választjuk (File/-New/Project), akkor ”CUDA [verziószám] Runtime” néven kiválasztható a projekt típusának a CUDA. Adjunk neki egy nevet és egy elérési mappát, és létre is jön a projektünk.



3.4. ábra. 1. lépés: A CUDA letöltése



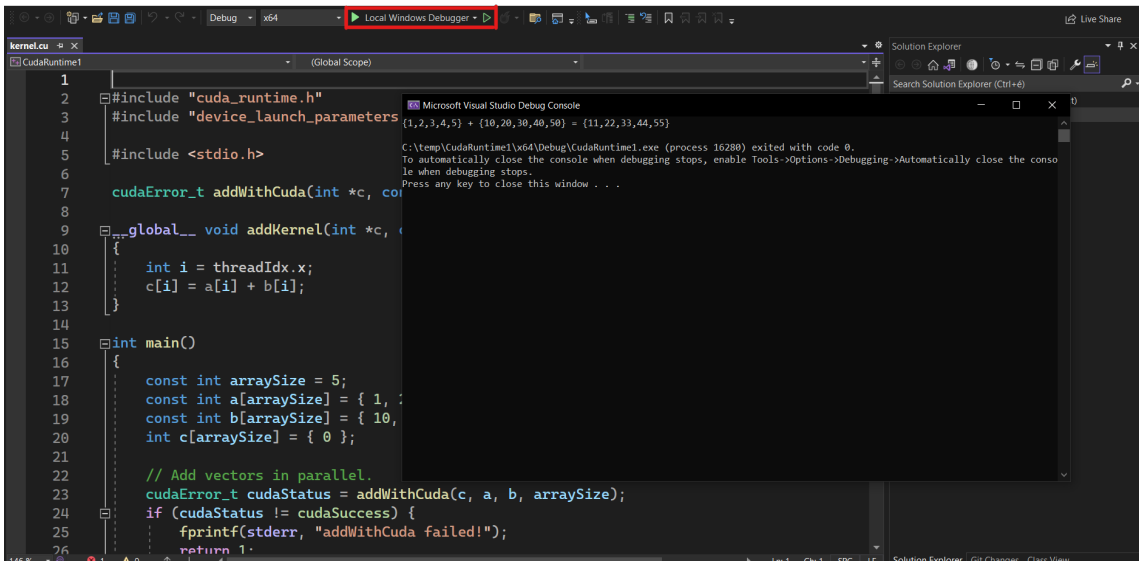
3.5. ábra. 2. lépés: Új CUDA projekt létrehozása

Példakód futtatása A template egy példaprogramot tartalmaz, amely két vektor összeadását végzi el videokártyán.

A kernel függvény nagyon egyszerű, mindössze 2 sorból áll: [7]

```
__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}
```

A ”Local Windows Debugger gomb” megnyomásával lefut a kód, és meg is kapjuk az eredményt. Ezután már írhatunk saját kódot is.



3.6. ábra. 3. lépés: Példakód futtatása

4. fejezet

Implementáció

Ebben a fejezetben vizsgáljuk meg a témában végzett programozói munkámat. Összesen 4 algoritmussal foglalkoztam, melyeket logikailag egymásra építve kezeltem: TSP → VRP → CVRP → (C)VRPTW

4.1. Hangyakolónia algoritmus

A munkám során megvalósított algoritmusok mindegyikéhez a hangyakolónia optimalizációt használtam.

4.1.1. Adatstruktúrák

Az útvonaltervezési problémák gráfokon futnak, ezért elsődleges feladatom annak eldöntése volt, hogy milyen formában tároljam a gráfokat. Az egyik legalapvetőbb megadási mód, ami én is mindenhol alkalmaztam, az a szomszédossági mátrix reprezentáció. Előnye, hogy az egyes csúcsokhoz tartozó élek direkt lekérdezhetőek, ezért a rajtuk végzett műveletek gyorsabbak lehetnek. Hátránya azonban, hogy minél kevesebb él van a gráfban, annál pazarlóbbá válik ez a megadás. Ismertek más fajta tárolási módok is, de azokkal most nem foglalkoztam. [2] Az adattípusok megválasztásában eltérőek az egyes megvalósítások: az első TSP verzióban például minden lebegőpontos számot *double* típusú változóban tárolok, míg a többi algoritmusnál *float* adattípust alkalmazok. A váltás hátterében az áll, hogy kipróbálva az első algoritmust azt tapasztaltam, hogy nem javít érdemben a valószínűségek dupla pontosságú számítása, cserébe sokat lehet spórolni a futásidőn a felbontás csökkentésével.

4.1.2. Feromonok nyilvántartása

A feromonokat az élekhez rendelem, ezért egy, az eredeti gráf topológiájával megegyező, teljes gráfban tárolom. A gráfélek eltárolása már másképpen történhet.

4.1.2.1. TSP

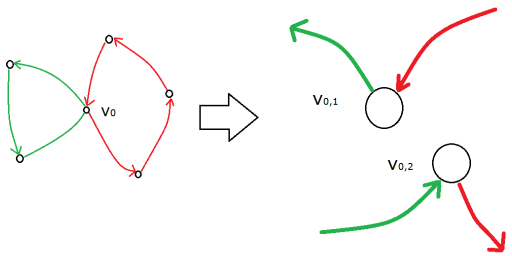
A TSP során egy NxN-es mátrix által adott feromongráfon egyértelműen végrehajtható mohó algoritmus, vagyis olyan döntéshozatal-sorozat, hogy minden csúcsból abba a következő pontba megyünk, ahova a legtöbb feromon vezet (kivéve ha az a kezdőcsúcsba vezet, és még maradtak bejáratlan csúcsok). Hogy megtartsam az implementáció egyszerűségét, a TSP során tényleg NxN-es 2 dimenziós mátrixban tárolom a feromonértékeket. Tehát az n csúcsú TSP feromonmátrix a kezdőpillanatban a következő:

$$\begin{pmatrix} 0 & P_{0,1} & P_{0,2} & \cdots & P_{0,n-1} \\ P_{1,0} & 0 & P_{1,2} & \cdots & P_{1,n-1} \\ \vdots & & \ddots & & \\ P_{n-1,0} & P_{n-1,1} & P_{n-1,2} & \cdots & 0 \end{pmatrix}$$

Ahol $P_{i,j}$ az i-ből j-be él kezdeti feromonértéke. $\forall i \in V(G) : P_{i,i} = 0$, mert a gráfunk egyszerű, nincs benne hurokél. Praktikusan $P_{i,j} = 0$ ha az input gráfban $v_{i,j}$ nincs behúzva. Ekkor már az első próbálkozások sem próbálnak meg arra menni. Ez a kikötés nem kötelező, előbb-utóbb ezen éleket úgyis elhalnának a feromonok.

4.1.2.2. VRP variánsok

A VRP esetében kicsit bonyolultabb a helyzet. Ha felírjuk egymás után az egyes járművek bejárási sorrendjét azt láthatjuk, hogy olyan, mintha több TSP-t hajtottunk volna végre egymás után. Képzeletben majdnem ez is történt. Úgy álltam hozzá a több útvonalhoz, hogy fejben kifeszítettem őket egy nagy körré. A kezdőcsúcs helyére gondolhatjuk azt, hogy k db csúcs van ott, akik egymástól 0 távolságra vannak. A gondolatmenetemet a 4.1. ábrán illusztrálom.



4.1. ábra. A VRP visszavezethető a TSP-re, ha az egyes utakat képzeletben összefűzzük egy nagy körré

Egy n csúcsot k járművel bejáró VRP visszavezethető egy n+k-1 csúcsú TSP-re. Ha egy jármű nem vesz részt a folyamatban, akkor olyan, mintha (v_0, v_0) útvonalat tenne meg egy hurokélen haladva. Ezt most megengedjük, ezért a feromonokat tartalmazó mátrixot kibővítjük. A $v_{0,k}$ csúcsra vonatkozó sor 0. eleme a $v_{0,k}$-ból $v_{0,k+1}$-be mutató él feromonértéke. Az első n sor ugyanolyan, mint a TSP esetén, ezzel biztosítva, hogy további járműveket belevéve a rendszerbe a TSP-ből át tudunk menni VRP-be.

Minden sor egy adott kiindulási csúcsra vonatkozik. Az i. sorban a kiindulási csúcs, ha 0 < i < n+k-1:

- i = 0 : $v_{0,0}$
- 0 < i < n : v_i
- n <= i < n + k - 1 : $v_{0,i-(n-1)}$

Tehát a k járművel n csúcson futó VRP feromonmátrix a kezdőpillanatban a következő:

$$\begin{pmatrix} P_{v_{0,0},v_{0,1}} & P_{0,1} & P_{0,2} & \cdots & P_{0,n-1} \\ P_{1,0} & 0 & P_{1,2} & \cdots & P_{1,n-1} \\ \vdots & & \ddots & & \\ P_{n-1,0} & P_{n-1,1} & P_{n-1,2} & \cdots & 0 \\ P_{v_{0,1},v_{0,2}} & P_{v_{0,1},1} & P_{v_{0,1},2} & \cdots & P_{v_{0,1},n-1} \\ \vdots & & & & \\ P_{v_{0,i},v_{0,i+1}} & P_{v_{0,i},1} & P_{v_{0,i},2} & \cdots & P_{v_{0,i},n-1} \\ \vdots & & & & \\ P_{v_{0,k-1},v_{0,0}} & P_{v_{0,k-1},1} & P_{v_{0,k-1},2} & \cdots & P_{v_{0,k-1},n-1} \end{pmatrix}$$

Feltételek megnyilvánulása: az, hogy milyen egyéb feltételeket tűzünk még ki a feladatba, a feromonok nyilvántartásának módját nem befolyásolja. Csak ott módosít, hogy hiába vezet út az adott körön, ha például a járművel kapacitásfeltétele sérül. Olyankor egyáltalán nem veszem figyelembe az adott megoldást.

4.1.3. A rulettkerék algoritmus

Az algoritmusaimban többször feltűnik a random számok generálásának problémája. Hallgatótársam, Tóth Andor Márk sokat foglalkozott a témával, és diplomamunkájából sokat tanulhattam a témában [2]. Valószínűségszámításból tudjuk, hogy igazi véletlen számok nem léteznek. Elméletileg nincs is semmi értelme ennek a fogalomnak. Nézzük például a klasszikus példát: a pénzérme feldobását, ami a földre érve fejjel vagy írással felfelé landol (vagy egyes nagy sikerű filmekben a sirály még a földet érés előtt elkapja és elviszi). Ha a tér minden egyes pontjában pontosan ismernénk a különböző légnyomásértékeket, valamint tökéletesen tisztában lennénk a feldobás dinamikai jellemzőivel, mindig pontosan tudnánk, hogy melyik oldalára fog esni az érme. A véletlen illúzióját ezen ismeretek hiánya okozza. Póker közben feltehetően nem ismerjük a keverő pontos technikáját, ezért az osztott lapok jó esetben véletlenszerűnek hatnak. A számítógépek alapesetben determinisztikus gépek, hiszen adott utasításokat hajtanak végre. A CUDA az úgynevezett PRNG (pszeudorandomszám-generátor) elvét követi. A generált szám tökéletesen meghatározható egy kezdeti érték alapján, ezért kell egy seed, ami viszont már független lehet a számítógéptől. Én a klasszikus megoldást választottam, miszerint az 1970. január 1. éjfél óta eltelt másodpercek számát vettem seednek. GPU oldalon a CuRand függvénykönyvtár videokártyás támogatást biztosít pszeudorandom számok generálására, ehhez a következő egyszerű kernelt kell futtatni:

```
// Initializes a random seed for each different threads
__global__ void setup_kernel(curandState* state, unsigned long seed)
{
  int id = blockIdx.x * blockDim.x + threadIdx.x;
  curand_init(seed, id, id, &state[id]);
}
```

A kernelt CPU oldalról kell felhívni a kiválasztott seeddel.

```
srand(time(0)); // Need seeds for random solutions

...

setup_kernel <<< BlockNum, threadPerBlock >>> (d_kernelParams.state, time(NULL) * rand());
```

4.2. A végeredmény számítása

Mind minden algorimtus esetében, itt is külön figyelmet kell fordítanunk arra, hogy a végső eredményt milyen formában állítjuk elő. A legkézenfekvőbb gondolat az, hogy út közben, ha minden korábbinál rövidebb megoldást találunk, akkor azt feljegyezzük, és a végén ami a legrövidebb megoldás, kinevezzük outputnak. Ezzel a megközelítéssel elméletben nincsen gond. Gyakorlatban amikor a programjaimat teszteltem, nagyon gyakran hibás végeredményt kaptam így. A hiba detektálása során arra jöttem rá, hogy a GPU nagyon gyakran hibázik, főleg adatmozgatások során. Ez azzal járt, hogy néha rossz eredményt, néha pedig semmit sem töltött a erre kijelölt tömbbe. Felismerésem után egyéb módot is kitaláltam. A megadott számú iteráció után ott marad a feromongráf, ami alapján a program önmagát tanította. Ebben a gráfban végrehajthatunk egy mohó algoritmust: kiindulva az előírt kezdőcsúcsból következő csúcsnak a megmaradtak közül mindig azt a csúcsot választjuk, ahova a legtöbb feromon vezet. Ekkor közvetlenül a kijelölt eredménytömbbe dolgozok, megkerülve a kritikusnak tapasztalt adatmozgatást. Hogy ne írjak felül jó megoldást, a mohó keresést csak akkor hajtom végre, ha a tömbben talált szekvencia érvénytelen.

```
// Choosing path with greedy algorithm if we dont have a valid answer
if (!validRoute(&params))
{
  // Mostly occurs when it did not find any routes, but we also prepare for corrupted memory
  printf("Need to find route in greedy mode!\n");
  greedySequence(&params);
}
```

4.3. TSP első verzió

Korábban már volt róla szó, de az útvonatkeresési algoritmusok ezen családjának legegyszerűbb tagja a TSP, ezért ezzel érdemes kezdenie annak, aki szeretne elmélyülni a problémakörben. Én is így tettem. Amikor még az Önálló laboratóriumi munkám során megismertem a genetikus algoritmusokat, az első feladatom a TSP megvalósítása volt CUDA nyelven. Azért tartom fontosnak kiemelni a TSP programom első verzióját, mert elsőkörben teljesen rendszertelenül álltam a problémához, nem tekintettem a feladatra egy nagyobb téma részeként. Ez később nehezen bővithető lett volna, ezért a következő verziókat teljesen a nulláról kellett felépítenem.

A mellékletben elérhető a teljes TSP_v1 kód, jelenleg néhány fontosabb elemet/fogalmat szeretnék kiemelni belőle. Bevezettem egy globális változót a hangyák (threadek) számának állítására. Jelenleg konstans, amennyiben a felhasználó egyéb számú párhuzamosítást szeretne, sajnos még bele kell nyúljon a forráskódba:

```
// 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

// Pheromon matrix constants
#define RHO 0.75  // Reduction ratio of previous pheromon value
#define REWARD_MULTIPLIER 100   // Reward multiplier after finding a shortest path until then
```

```
#define INITIAL_PHEROMONE_VALUE 1000    // Initial value of elements in the Pheromone
    matrix

#define SERIALMAXTRIES 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 feromon 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, ezért érdemes lehet többször (például 10-szer) egymás után végrehajtani. Ilyenkor végső eredménynek célszerű a legrövidebb megoldást venni. Sajnos a tesztelések során azt tapasztaltam, hogy a GPU főleg amikor több szálon dolgozik, mint ami fizikailag elérhető CUDA core formájában, gyakran hibázik, és vagy egyáltalán nem, vagy rosszul ír át memóriaterületet. Azon memóriaterület, mely tárolja azt aktuális minimális szekvenciát, rendkívül ki van téve hasonló hibázásoknak. Elképzelhető, hogy a programfutás eredetileg tervezett végén érvénytelen adat lenne az eredmény. Ilyenkor mentő ötletként végrehajtok egy mohó algoritmust a feromongráfon. A kezdőcsúcs a v_0, és a soron következő csúcs mindig az a megmaradt pont lesz, ahova a legtöbb feromon vezet.

```
if (!validRoute(&params)) {
  greedySequence(&params);
}
```

4.4. TSP második verzió

Amikor másodszorra álltam neki a TSP programozásának, már számos tapasztalattal bírtam. Első feladatomnak éreztem a felhasználóra bízni, hogy hány szállal kívánja használni az alkalmazást. Az ants változó már nem konstans, és parancssori argumentummal befolyásolható.

```
// Command Line Syntax: ... --ants [number of ants]
if ((strcmp(argv[i], "-a") == 0) || (strcmp(argv[i], "--ants") == 0))
{
  if (sscanf(argv[++i], "%d", &ants) != 1) {
    fprintf(stderr, "Unable to read ant number!\n");
  }
  else {
    printf("Given ant number : %d\n", ants);
  }
}
```

Az első verzió nagyon lassúra sikerült, ezért több módon is igyekeztem gyorsabbnak lenni. Azt találtam, hogy nem olyan fontos a feromonok pontos tárolása, ezért *double* helyett *float* adattípusra váltottam. Sok függvényhívást alkalmazok, ezért kritikusnak bizonyult a paraméterátadások gyorsítása is. A kernel által használt változókat egy általam definiált struktúrába szerveztem. Ilyen struktúrák pointereit adom csak át függvények között, ezzel kevesebb utasítást vesznek el a vezérlésátadások. A random generált szekvenciákat úgy állítom elő hamarabb, hogy a csúcsok egyesével sorsolása helyett csak n db véletlen csúcscserét alkalmazok. Arról, hogy ezek mekkora sebességnövekedést okoztak, a 5.3. részben írok.

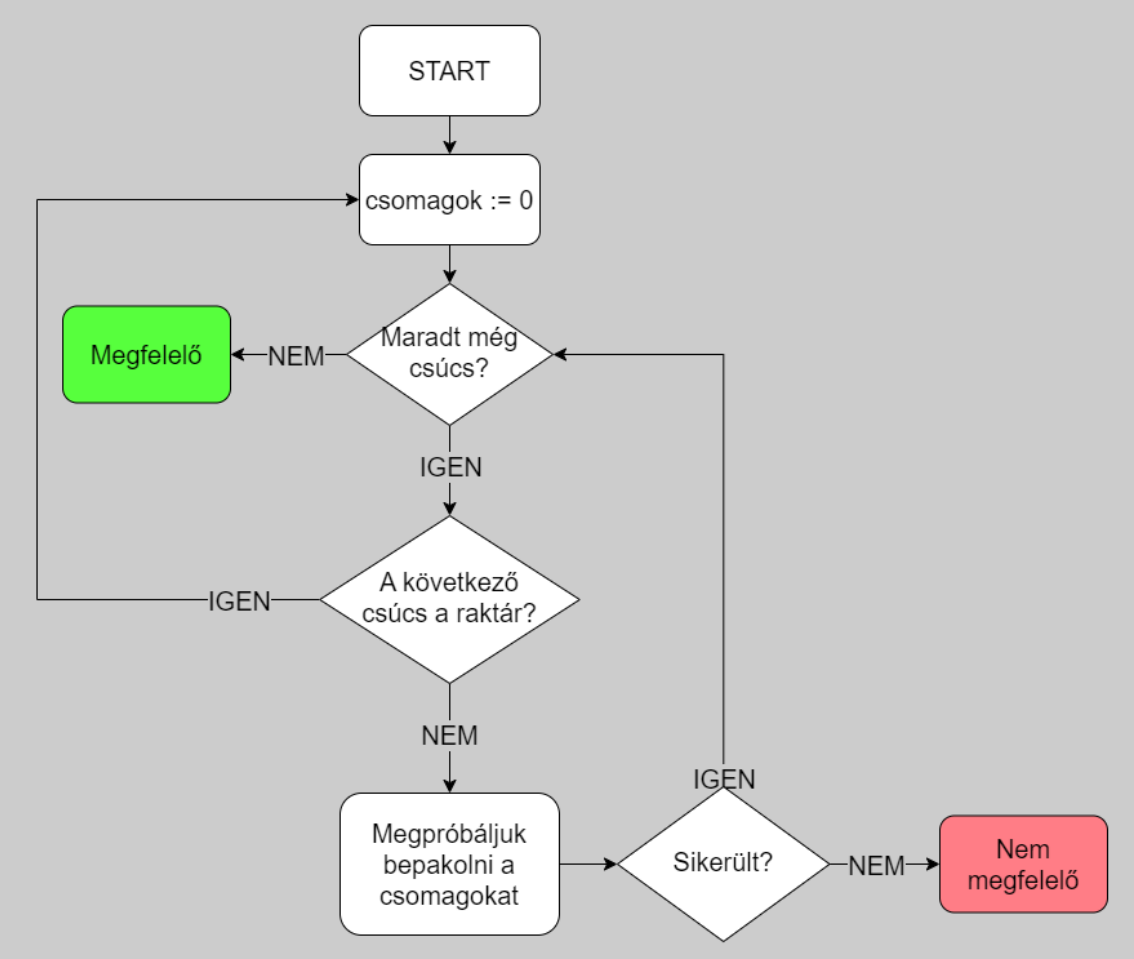
4.5. VRP

A VRP újítása, hogy több kis alegységre bomlik a körút, mert több járművet használhatunk. A 4.1.2.2. részben mutatottak alapján az n csúcsú k járműves VRP-t visszavezettem egy n+k-1 csúcsú TSP-re. A kódimplementációban is látszik, hogy ezen ötlet után elég a korábbi kódot használnunk. Jól jött, hogy a TSP v2-ben bővíthetőre lett megcsinálva a parametrizálás.

```
// Reduction to TSP
___host___ ___device___ inline int RouteSize(int size, int maxVehicles)
{
  return size + maxVehicles - 1;
};
```

4.6. CVRP

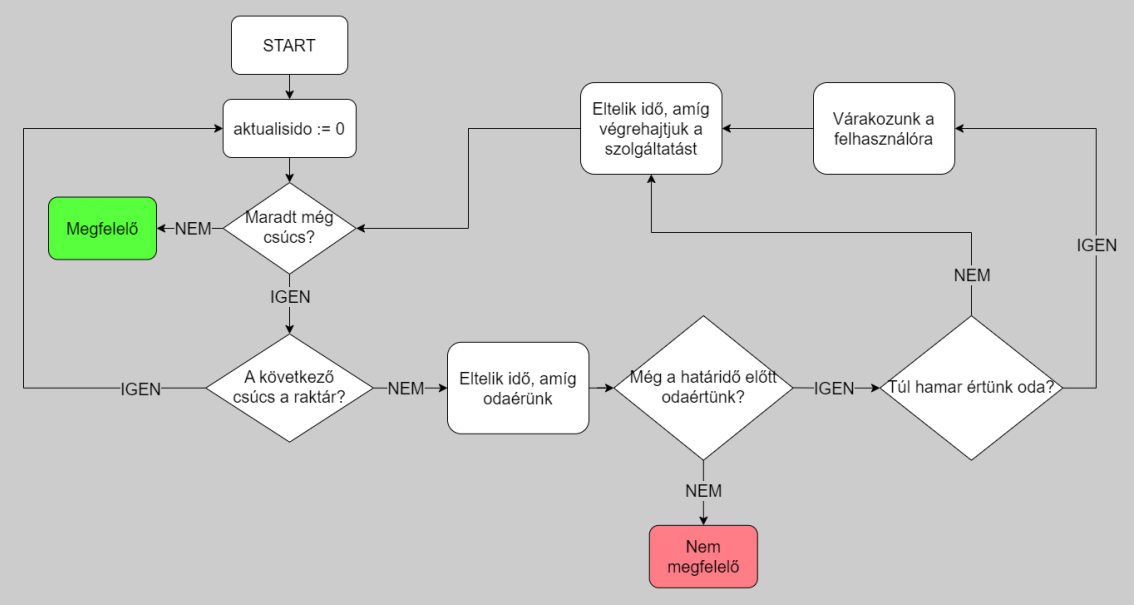
A CVRP azt jelenti, hogy feltételül kaptuk a járművek korlátozott kapacitásának számításba vételét. A VRP-t egy kapacitásviszonyokat kiértékelő függvénnyel egészítettem ki, melynek logikáját a 4.2. ábrán látható folyamatábra szemlélteti.



4.2. ábra. A CVRP során használt, kapacitásfeltételt kiértékelő függvény folyamatábrája

4.7. CVRPTW

A CVRPTW során a kapacitáskorlátok mellett az ügyfél által megszabott határidők is további nehézséget okoznak. A CVRP-t egy időablakokat kiértékelő függvénnyel egészítettem ki, melynek logikáját a 4.3. ábrán látható folyamatábra szemlélteti.



4.3. ábra. A VRPTW során használt, időablak-feltételt kiértékelő függvény folyamatábrája

5. fejezet

Eredmények

A méréseket az alábbi paraméterekkel rendelkező számítógépen végeztem:

- CPU: 11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz
- GPU: NVIDIA GeForce RTX 3050Ti
- RAM: 24 GB DDR4
- OS: Windows 10 Education 22H2
- Compiler: NVCC V12.0.140

5.1. A mérések menete

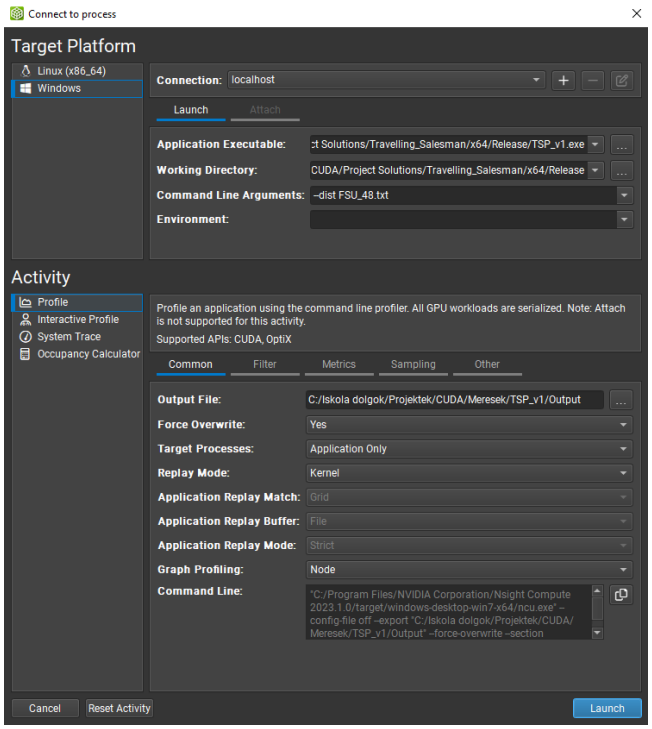
Az algoritmusokhoz készítettem CPU és GPU kódot is. Az algoritmusok érdemi részét végig GPU-n végeztem. CPU-n a kezdeti értékeket állítottam be, különböző diagnosztikai kiíratásokat végeztem, illetve a parancssori argumentumok feldolgozását. A futásidők méréséhez az NVIDIA által készített Nsight Compute(R) programot használtam, mely képes CUDA kerneleket elemezni több szempontból is: Futási idő, Memória kihasználtsága, Grid mérete, szálanként regiszterek száma és még tovább. A program futását 5.2. ábrán ábrázoltam.

Számomra a legfontosabb a futásidő volt, ezt több különböző bemenetre és konfigurációra lemértem, majd táblázatosan összegyűjtöttem.

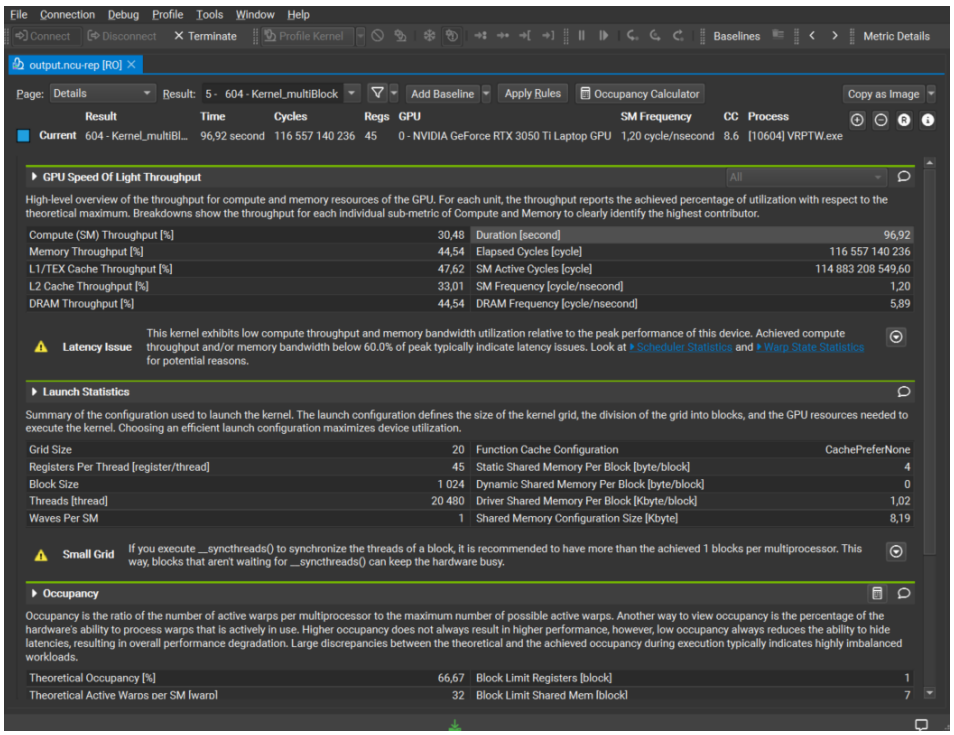
5.2. Mérési eredmények

A továbbiakban táblázatokba szedtem az egyes algoritmusokon végzett profilozó méréseim eredményeit. A futásidők időegysége másodperc, mert ilyen nagyságrendben mozogtak az értékek. Többször lefuttattam a programokat ugyanolyan paraméterek mellett, ebből lejegyeztem az átlagos eredményeket (10 futás számtani közepe) és a legkisebb kapott megoldást.

A CVRPTW mérése során tapasztaltam először olyat, hogy néha a program egyáltalán nem talált megoldást. Ott új oszlopot vettem fel ”Hiba (%)” névvel. Azt mutatja, hogy a programfuttatások hány százalékában nem sikerült egyáltalán megoldást találni. Ilyenkor az átlagolást a sikeres futtatásokból számoltam.



5.1. ábra. Az NVIDIA Nsight Compute programban egy mérés összeállításához be kell állítani a mérendő exe fájlt, a munkamappát, illetve a szükséges parancssori argumentumokat



5.2. ábra. Az NVIDIA Nsight Compute program adatok széles tárházával látja el a programozót

5.2.1. TSP első verzió

Teszteléshez szükségem volt ismert eredményű adathalmazokra. A Floridai Állami Egyetem weboldalán [10] elérhető bárki számára több adathalmaz, különböző adatstruktúrában. Nekem a [fájlnév]_d.txt nevű fájlok voltak hasznosak, ugyanis abban megtalálhatóak a szomszédossági mátrix költségei táblázatos 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 élsúlyok tárolása csak double formátumban történik. Az összehasonlíthatóság érdekében egy iteráció során 20 random, és 500 tudatos hangya fut. A kezdeti feromon érték 1000, az elnyomási tényező $\rho = 0.75$, a jutalmazási arány 100, amely csak a 2. iterációtól érvényes (ha van).

Az egyes mérések eredményei a 5.1 - 5.6. táblázatokból olvashatóak ki.

FIVE : 5 csúcs, minimális út : 19, átlagos út : 24			
	Futásidő (ms)	Végeredmény átlag	Végeredmény min.
1 rep			
32 ant	7,84	20,2 (+6,3%)	19
256 ant	19,1	20,8 (+9,5%)	19
1024 ant	76,1	20,8 (+9,5%)	19
10 rep			
256 ant	137,3	19	19
1024 ant	424	19	19

5.1. táblázat. Elsőnek egy kicsi, a 0. kiindulási ponton kívül 4 állomásból álló gráfon próbáltam ki az algoritmust.

P01 : 15 csúcs, minimális út : 291, átlagos út : 662			
	Futásidő (s)	Végeredmény átlag	Végeredmény min.
1 rep			
1024 ant	0,925	370,2 (+27,2%)	291
2048 ant	1,18	365 (+25,4%)	327 (+12,4%)
4096 ant	1,20	359,7 (+23,6%)	332 (+14,1%)
10 rep			
1024 ant	11,39	350,4 (+20,4%)	291
2048 ant	11,47	328 (+12,5%)	295 (+1,4%)
4096 ant	11,59	336,8 (+15,7%)	291
16384 ant	13,21	326,2 (+12,1%)	295 (+1,4%)
20480 ant	13,46	323,4 (+11,1%)	291

5.2. táblázat. 15 csúcsú gráfon futtatott TSP v1

A 17 vagy nála nagyobb gráfok esetében az 1 iterációs algoritmusok már olyan rosszul teljesítettek, hogy a méréseket legalább 10 iterációval folytattam.

GR : 17 csúcs, minimális út : 2085			
	Futásidő (s)	Végeredmény átlag	Végeredmény min.
10 rep			
1024 ant	14,85	2391,3 (+14,7%)	2151 (+3,2%)
2048 ant	14,88	2363,4 (+13,5%)	2085
4096 ant	14,97	2279,6 (+9,3%)	2097 (+0,6%)
8192 ant	15,15	2306,4 (+10,6%)	2207 (+5,9%)
16384 ant	15,49	2250,5 (+7,9%)	2085
20480 ant	15,68	2221,7 (+6,6%)	2085

5.3. táblázat. 17 csúcsú gráfon futtatott TSP v1

FRI26 : 26 csúcs, minimális út : 937, átlagos út : 2693			
	Futásidő (s)	Végeredmény átlag	Végeredmény min.
10 rep			
1024 ant	39,72	1386,1 (+47,9%)	1249 (+33,3%)
2048 ant	39,73	1367,1 (+45,9%)	1221 (+30,3%)
4096 ant	39,82	1227,6 (+31%)	1121 (+19,6%)
8192 ant	40,14	1158,7 (+23,7%)	1102 (+17,6%)
16384 ant	40,70	1132,1 (+20,8%)	1075 (+14,7%)
20480 ant	42,72	1152,3 (+23%)	1097 (+17,1%)

5.4. táblázat. 26 csúcsú gráfon futtatott TSP v1

DANTZIG42 : 42 csúcs, minimális út : 699, átlagos út : 3110,5			
	Futásidő (s)	Végeredmény átlag	Végeredmény min.
10 rep			
1024 ant	115,41	1735,7 (+148%)	1554 (+122%)
2048 ant	113,94	1420 (+103%)	1252 (+79%)
16384 ant	114,34	987,7 (+41%)	906 (+30%)
20480 ant	113,55	931,6 (+33%)	879 (+26%)

5.5. táblázat. 42 csúcsú gráfon futtatott TSP v1

ATT48 : 48 csúcs, minimális út : 33523, átlagos út : 157686,9			
	Futásidő (s)	Végeredmény átlag	Végeredmény min.
10 rep			
1024 ant	156,7	79993,2 (+139%)	75601 (+126%)
8192 ant	152,82	56097,9 (+67%)	51139 (+53%)
16384 ant	153,78	50197,2 (+50%)	47387 (+41%)
20480 ant	154,28	46360,8 (+38,3%)	43167 (+28,77%)

5.6. táblázat. 48 csúcsú gráfon futtatott TSP v1

5.2.2. TSP második (konzisztens) verzió

Azért, hogy az előző fejezetben látott első verzióval érdemben össze tudjam hasonlítani a mostani verziót, hasonló iterációs számokat választottam: 1 rep-en belül 20 random hangyát követ 500 tudatos hangya. Az adathalmaz is az előbbi [10] helyről származó csomag. A számértékeket összevetve szembetűnő a gyorsulás az előző verzióhoz képest. A legfőbb szerepe ebben úgy gondolom, hogy az adatok double-ről float-ra cserélése, illetve a függvények között szigorú pointer alapú paraméterátadás jelentette.

Az egyes mérések eredményei a 5.7 - 5.11. táblázatokból olvashatóak ki.

P01 : 15 csúcs, minimális út : 291, átlagos út : 662			
	Futásidő (s)	Végeredmény átlag	Végeredmény min.
10 rep			
1024 ant	2,69	320,0 (+10%)	307 (+5,5%)
2048 ant	2,82	307,6 (+5,7%)	291
4096 ant	2,97	305,4 (+4,9%)	291
8192 ant	3,19	300,2 (+3,2%)	291
12288 ant	3,50	296,4 (+1,9%)	291
16384 ant	3,73	294,6 (+1,2%)	291
20480 ant	3,92	295,4 (+1,5%)	291
30 rep			
1024 ant	8,05	301,6 (+3,6%)	291
20480 ant	13,37	291	291

5.7. táblázat. 15 csúcsú gráfon futtatott TSP v2

GR : 17 csúcs, minimális út : 2085			
	Futásidő (s)	Végeredmény átlag	Végeredmény min.
10 rep			
1024 ant	3,54	2246 (+7,7%)	2094 (+0,4%)
2048 ant	3,69	2237,7 (+7,3%)	2207 (+5,9%)
4096 ant	3,69	2196,5 (+5,3%)	2142 (+2,7%)
8192 ant	3,89	2211,5 (+6,1%)	2170 (+4,1%)
16384 ant	4,37	2179,2 (+4,5%)	2129 (+2,1%)
20480 ant	4,55	2175,1 (+4,3%)	2134 (+2,4%)
30 rep			
20480 ant	13,38	2146,5	2103

5.8. táblázat. 17 csúcsú gráfon futtatott TSP v2

FRI26 : 26 csúcs, minimális út : 937, átlagos út : 2693			
	Futásidő (s)	Végeredmény átlag	Végeredmény min.
10 rep			
1024 ant	9,00	1393 (+48,7%)	1347 (+43,8%)
2048 ant	9,23	1363,1 (+45,5%)	1288 (+37,5%)
4096 ant	9,43	1215,3 (+29,7%)	1163 (+24,1%)
8192 ant	9,72	1136,9 (+21,3%)	1055 (+12,6%)
16384 ant	10,82	1104,9 (+17,8%)	1007 (+7,5%)
20480 ant	11,61	1111,7 (+18,6%)	1058 (+12,9%)
30 rep			
20480 ant	32,93	1098,2 (+17,2%)	1047 (+11,7%)

5.9. táblázat. 26 csúcsú gráfon futtatott TSP v2

DANTZIG42 : 42 csúcs, minimális út : 699, átlagos út : 3110,5			
	Futásidő (s)	Végeredmény átlag	Végeredmény min.
10 rep			
1024 ant	26,19	1601,3 (+129%)	1018 (+46%)
2048 ant	26,61	1488,4 (+113%)	1319 (+89%)
4096 ant	27,02	1286,7 (+84%)	1168 (+67%)
8192 ant	27,57	1118,3 (+60%)	1004 (+44%)
16384 ant	31,2	1014,7 (+45%)	909 (+30%)
20480 ant	33,31	964,9 (+38%)	822 (+17,6%)
30 rep			
20480 ant	99,83	972,8 (+39,2%)	886 (+26,75%)

5.10. táblázat. 42 csúcsú gráfon futtatott TSP v2

ATT48 : 48 csúcs, minimális út : 33523, átlagos út : 157686,9			
	Futásidő (s)	Végeredmény átlag	Végeredmény min.
10 rep			
1024 ant	38,12	84660,7 (+153%)	67541 (+102%)
2048 ant	38,63	72522,8 (+116%)	64969 (+94%)
4096 ant	39,14	62808,6 (+87%)	53395 (+59%)
8192 ant	39,69	59604,8 (+78%)	52514 (+57%)
16384 ant	41,56	48618,2 (+45%)	46227 (+38%)
20480 ant	41,93	49879,9 (+48,8%)	45836 (+36,7%)
30 rep			
20480 ant	124,72	47562,9 (+41,9%)	44144 (+31,7%)

5.11. táblázat. 48 csúcsú gráfon futtatott TSP v2

5.2.3. VRP

A VRP-hez eleinte nem értettem, hogy miért nem találtam külön adathalmazt. Később megértettem, hogy mivel nincs megadva egyéb feltétel, (általában) teljesen mindegy, hogy hány járművet használhat az algoritmus, a legjobban akkor fog járni, ha az összes állomást ugyanazon járművel járja végig. A VRP-hez is készült külön kódimplementáció. Ez azért jó, mert amikor különböző feltételeket szabunk meg, elég annyit tenni, hogy az adott feltételt kiértékelő függvényt megírva meghívjuk az eredményt értékelő (antRouteLength) programrészben.

5.2.4. CVRP

Az első olyan algoritmus, amely feltételes útvonaltervezést igényel, a Kapacitásos járműútvonal-tervezési feladat. Adathalmazt a következő helyről szedtem: [9]. Az iterációk az előzőeknél látottakkal megegyezőek: 1 rep-en belül 20 random hangyát követ 500 tudatos hangya.

Az egyes mérések eredményei a 5.12 - 5.15. táblázatokból olvashatóak ki.

17-city problem (Groetschel): 17 csúcs, minimális út: 2685			
	Futásidő (s)	Végeredmény átlag	Végeredmény min.
10 rep			
1024 ant	2,96	3158,7 (+17,6%)	2726 (+1,5%)
16384 ant	3,44	2790,4 (+3,9%)	2711 (+1%)
20480 ant	3,83	2800,5 (+4,3%)	2685
30 rep			
1024 ant	6,05	2975,2 (+10,8%)	2685
16384 ant	10,64	2846,2 (+6%)	2727 (+1,5%)
20480 ant	13,80	2758,5 (+2,7%)	2685
50 rep			
1024 ant	9,97	2927,5 (+9%)	2685
16384 ant	17,64	2737,25 (+1,9%)	2685
20480 ant	18,51	2724,9 (+1,5%)	2685

5.12. táblázat. gr17: 17 csúcsú gráfon futtatott CVRP

21-city problem (Groetschel): 21 csúcs, minimális út: 3704			
	Futásidő (s)	Végeredmény átlag	Végeredmény min.
10 rep			
1024 ant	2,96	5026,3 (+35,7%)	4389 (+18,5%)
16384 ant	4,75	4217 (+13,9%)	4068 (+9,8%)
20480 ant	5,72	4103,3 (+10,8%)	3903 (+5,4%)
30 rep			
1024 ant	8,31	4770,5 (+28,8%)	4001 (+8%)
16384 ant	12,17	4105,4 (+10,8%)	3864 (+4,3%)
20480 ant	16,61	4133 (+11,6%)	3881 (+4,8%)
50 rep			
1024 ant	19,76	4467,5 (+20,6%)	3758 (+1,5%)
16384 ant	22,73	4259,3 (+15%)	4090 (+10,4%)
20480 ant	28,03	4211,6 (+13,7%)	4053 (+9,4%)

5.13. táblázat. gr21: 21 csúcsú gráfon futtatott CVRP

5.2.5. CVRPTW

Most már az időablakok adta bonyolítást is bevesszük a feltételek közé. Olyan adathalmazzal dolgoztam [15], ami egyszerre írja elő a járművek kapacitását, illetve a célpontok készen állási és határidejét. A program tesztelése során egy korábban nem tapasztalt jelenséget észleltem: **gyakran egyáltalán nem talált megoldást a program**.

Én ezt két okra vezetem vissza. Egyik, hogy az ide tartozó adathalmaz jóval nagyobb méretű gráfokat tartalmazott (akár 100 csúcs), mint a korábbi vizsgált adatok. Másik lehetséges ok, hogy sok feltétel mellett kevésbé hatékony a Hangyakolónia Optimalizáció. Míg a sima VRP esetén ha adott egy optimális megoldás, akkor abban kis sorrendcserék még ugyanúgy közeli eredményre vezetnek. A CVRPTW-nél fokozottan igaz, hogy kis módosítások teljesen rossz megoldást tudnak kreálni egy előző rövid végeredményből. A CVRP is ki volt téve bizonyos fokig a problémának, de ott ez látszólag nem különösebben befolyásolta az eredményeket.

Az adathalmaz sok gráfot tartalmaz, ebből én kettőt választottam: C101 és C201. Mindkettő 101 csúcsot tartalmaz, viszont az algoritmus futtatható az első 25 vagy első 50 pontra is. A C101-ben a különböző feltételek csak néhány klienst engednek járművenként

Augerat et al: 33 csúcs, minimális út: 742			
	Futásidő (s)	Végeredmény átlag	Végeredmény min.
10 rep			
1024 ant	5,62	1571,62 (+111,8%)	1283,05 (+72,9%)
16384 ant	8,61	1496,24 (+101,6%)	1342,16 (+80,7%)
20480 ant	13,01	1470,99 (+98,3%)	1310,18 (+76,6%)
30 rep			
1024 ant	22,74	1529,4 (+106%)	1351,81 (+82%)
16384 ant	23,68	1375,43 (+85,3%)	1209,97 (+63%)
20480 ant	35,24	1415,17 (+90,7%)	1182,57 (+59,4%)
50 rep			
1024 ant	24,91	1434,34 (+93,9%)	1230 (+65,8%)
16384 ant	41,26	1421,19 (+91,5%)	1361,51 (+83,5%)
20480 ant	61,01	1402,18 (+89%)	1092,92 (+47,3%)

5.14. táblázat. a33: 33 csúcsú gráfon futtatott CVRP

Rinaldi,Yarrow/Araque: 48 csúcs, minimális út: 40002			
	Futásidő (s)	Végeredmény átlag	Végeredmény min.
10 rep			
1024 ant	44,28	89736,85 (+124%)	56813,06 (+42%)
16384 ant	44,86	83208,23 (+108%)	57162,13 (+43%)
20480 ant	48,89	89271,67 (+123%)	56966,07 (+42%)
30 rep			
1024 ant	76,93	81507,64 (+104%)	67745 (+69,4%)
16384 ant	133,62	74530,89 (+86,3%)	58940,59 (+47,3%)
20480 ant	146,36	76197,09 (+90,5%)	53783,10 (+34,5%)
50 rep			
1024 ant	223,39	75265,07 (+88,2%)	58007,69 (+45%)
16384 ant	223,5	69196,02 (+73%)	54812,73 (+37%)
20480 ant	215,03	64510,92 (+61,3%)	56817,40 (+42%)

5.15. táblázat. att48: 48 csúcsú gráfon futtatott CVRP

Fisher problem 11: 72 csúcs, minimális út: 237			
	Futásidő (s)	Végeredmény átlag	Végeredmény min.
10 rep			
16384 ant	110,84	887,96 (+275%)	645,34 (+172%)
20480 ant	112,34	923,91 (+290%)	874,96 (+269%)
30 rep			
16384 ant	292,38	764,31 (+223%)	551,81 (+133%)
20480 ant	346,49	706,20 (+198%)	458,31 (+93%)
50 rep			
16384 ant	478,71	732,96 (+209%)	574,85 (+143%)
20480 ant	572,36	828,31 (+250%)	492,35 (+108%)

5.16. táblázat. f72: 72 csúcsú gráfon futtatott CVRP

(megközelítőleg 5-10), míg a C201-ben szabadabbak a feltételek. Mindkét csúcshalmazt kipróbáltam 25, 50, és 100 csúcsra.

Új oszlopként megjelent a ”Hiba (%)”, mely arra utal, hogy az algoritmus a programfuttatások hány %-ában nem talált egyáltalán megoldást. Az eredmények lejegyzésénél már a 10 iterációs eseteket elhagytam, ugyanis az összes gráfra 90-100%-os hibával dolgoztak, vagyis egyáltalán nem tudtak eredményre jutni. Hasonló okokból kifolyólag csak a számítógépemen futtatható maximális, 20 blokkos, vagyis 20480 szálú eseteket néztem.

C101.25: 2 csúcs, minimális út: 191,3				
	Futásidő (s)	Végeredmény átlag	Végeredmény min.	Hiba(%)
30 rep				
1024 ant				
16384 ant				
20480 ant	39,31	441,68 (+131%)	336,19 (+76%)	70%
50 rep				
1024 ant				
16384 ant				
20480 ant	71,45	357,19 (+87%)	346,44 (+81%)	60%
70 rep				
20480 ant	94,93	410,70 (+115%)	391,42 (+105%)	60%

5.17. táblázat. A szigorúbb időablakokkal bíró adathalmaz első 26 csúcsára vett CVRPTW mérés (a nevében a 25 arra utal, hogy 25 kliens van, és egy raktár)

C101.50: 50 csúcs, minimális út: 362.4				
	Futásidő (s)	Végeredmény átlag	Végeredmény min.	Hiba(%)
30 rep				
1024 ant				
16384 ant				
20480 ant				100%
50 rep				
1024 ant				
16384 ant				
20480 ant				100%
70 rep				
20480 ant				100%

5.18. táblázat. A szigorúbb időablakokkal bíró adathalmaz első 51 csúcsára vett CVRPTW mérés

5.3. Eredmények értékelése, továbbfejlesztési lehetőségek

Sok mérést végeztem, amiből szeretnék levonni több konklúziót.

5.3.1. A két Utazóügynök probléma (TSP) verzió összehasonlítása

Az első megvalósított algoritmusommal a TSP, vagyis az Utazóügynök problémát oldottam meg, segítségül hívva a már korábban látott Hangyakolónia optimalizációt. Az első verzió kellő tapasztalat hiányában készült, ezért több szempontból is problémásnak bizonyult. Ilyenek a hosszú futásidő és a korlátozott bővíthetőség. Látva a profilozó mérések eredményeit, olyan változtatásokat tudtam eszközölni, mint az adattípus cseréje *float*-ra,

C101.100: 100 csúcs, minimális út: 827.3				
	Futásidő (s)	Végeredmény átlag	Végeredmény min.	Hiba(%)
30 rep				
1024 ant				
16384 ant				
20480 ant				
50 rep				
1024 ant				
16384 ant				
20480 ant				

5.19. táblázat. [CVRPTW] A szigorúbb időablakokkal bíró adathalmaz teljes egészére vett mérés

C201.25: 25 csúcs, minimális út: 214.7				
	Futásidő (s)	Végeredmény átlag	Végeredmény min.	Hiba(%)
30 rep				
1024 ant				
16384 ant				
20480 ant	41,61	590,41 (+175%)	432 (+101%)	70%
50 rep				
1024 ant		477,76 (+123%)	452,63 (+111%)	70%
16384 ant				
20480 ant	66,97	398,36 (+86%)	322 (+50%)	10%
70 rep				
20480 ant	96,92	469,16 (+119%)	396,80 (+85%)	60%

5.20. táblázat. A tágabb időablakokkal bíró adathalmaz első 26 csúcsára vett mérés

C201.50: 50 csúcs, minimális út: 360.2				
	Futásidő (s)	Végeredmény átlag	Végeredmény min.	Hiba(%)
30 rep				
1024 ant				
16384 ant				
20480 ant				100%
50 rep				
1024 ant				
16384 ant				
20480 ant				100%
70 rep				
20480 ant				%

5.21. táblázat

valamint a kernelen belüli szigorú pointer szerinti paraméterátadás. A módosításoknak meglett az eredménye: hasonló végeredmények mellett **a futásidők 3-4-szer rövidebbek tudtak lenni**. Az összehasonlítás a 5.3. ábrán látható.

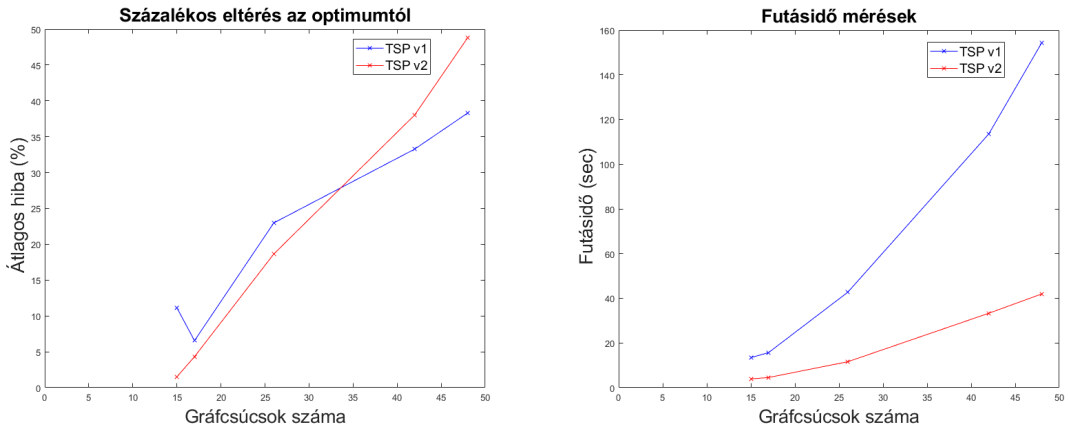
Mindkét implementációról elmondható, hogy jócskán felülmúlta a ”Brute force” megoldást. Míg nyers összehasonlításokkal egy 48 csúcsú gráf összes lehetséges csúcssorrendjének kiértékelése évmilliókba telt volna, az ACO képes volt erre 2 perc alatt úgy, hogy az

C201.100: 100 csúcs, minimális út: 589.1				
	Futásidő (s)	Végeredmény átlag	Végeredmény min.	Hiba(%)
30 rep 1024 ant 16384 ant 20480 ant				
50 rep 1024 ant 16384 ant 20480 ant				

5.22. táblázat

Gráf neve: x csúcs, minimális út: x				
	Futásidő (s)	Végeredmény átlag	Végeredmény min.	Hiba(%)
30 rep 1024 ant 16384 ant 20480 ant				
50 rep 1024 ant 16384 ant 20480 ant				

5.23. táblázat



5.3. ábra. A két TSP verziót ugyanazon tesztadatokon futtattam. 20480 thread és 10 iteráció mellett Matlabban ábrázoltam a kapott eredményeket. Látható, hogy a TSP v2 hasonló eredmények mellett 3-4x gyorsabb futásra volt képes

optimális megoldástól átlagosan mindössze 41,2%-kal hosszabb utat talált, legjobb esetben csak 31,6%-kal.

5.3.2. A különböző Járműútvonal-tervezési problémák

szép grafikonok plusz rizsa: több szál = lassabb, de jobb eredmény

A futásidőkben gyakran nem fokozatos az emelkedés például a threadszám növelésével. Ez azért is lehet, mert időben nem egyszerre futtattam a teszteket, más volt a gép állapota, nem ugyanolyan programok futottak a háttérben stb.

5.3.3. Kritériumfeltételek problémája a Hangyakolónia algoritmussal

Ahogy már korábban a 2.4.2.1. részben utaltam rá, a Hangyakolónia algoritmus számára is kihívással jár az, amikor a keresett útvonal különböző feltételeknek maradéktalanul meg kell feleljen. Dolgozatomban kétféle feltételrendszert vizsgáltam: a járművek korlátozott kapacitását, illetve a felhasználók megszabott időbeosztását. A CVRP esetében Most összefoglalnám, hogy miért érhettek el a feltételes esetben végzett méréseim jelentősen rosszabb eredményeket.

Az ACO úgy működik, hogy eleinte a gyengébb megoldásokból generál egyre jobbakat. Feltételezi, hogy találunk megoldást, amit javítani tudunk. A kritériumfeltételek olyanok, hogy vagy teljesülnek, vagy nem. Nehéz úgy értékelni, hogy az első csak kicsit hibázott, a következő közepesen stb. Ha rossz, akkor nem engedhetjük a megoldáshalmazba.

Most azt nézném meg, hogy ha véletlenszerű csúcssorrendeket generálok úgy, ahogyan ezt a hangyák is teszik az ACO-ban, akkor mekkora annak a valószínűsége, hogy találunk egy, a kritériumnak megfelelő megoldást.

Ha egy esemény bekövetkezési valószínűsége p, akkor annak a valószínűsége, hogy n-szer ismételve az eseményt legalább 1 alkalommal bekövetkezik, binomiális valószínűségi eloszlás szerint:

$$P(x > 0) = 1 - (1 - p)^n \tag{5.1}$$

5.3.4. Kapacitásfeltétel

Kezdjük a kapacitáskritériummal, mert az az egyszerűbb. Az első 5 adathalmazban minden csúcs igénye 1, vagyis azt kell néztünk, hogy az egyes járművek hány csúcsot járnak be. A kezdőcsúcsot nem kell beleszámolni, mert annak nincs igénye.

Probléma: Az (n-1) db csúcsot véletlenszerűen szétosztjuk k db jármű között. Mennyi annak a valószínűsége, hogy egyik járműhöz sem osztanak több, mint C db-ot? Első példa (gr17): 16 állomás, 3 jármű, a járművek kapacitása 6. $p_{gr17} =?$

Csak 6-6-4 vagy 6-5-5 felosztásban lehetnek a csúcsok. Az összes eset 3^{16}, mert minden csúcsra eldöntjük, hogy melyik járműhöz osztjuk be. Összesen $\binom{16}{6} * \binom{10}{6} * 3 + \binom{16}{6} * \binom{10}{5} * 3$ esetben nem lesz több, mint 6 egyik jármű kapacitása sem.

$$p_{gr17} = 3 * \frac{\binom{16}{6} * \binom{10}{6} + \binom{16}{6} * \binom{10}{5}}{3^{16}} \approx 0,258 \tag{5.2}$$

Nézzük meg az att48-ra is ugyanezt: $(n-1) = 47$ állomás, k=3 jármű, a járművek kapacitása C=16. $p_{att48} =?$

Ebben a problémában csak 16-16-15 felosztásban lehetnek a csúcsok. Az előbbi példához hasonlóan gondolkozva a valószínűség:

$$p_{gr17} = \frac{3 * \binom{47}{16} * \binom{31}{16}}{3^{47}} \approx 0,051 \tag{5.3}$$

Ha 20480 szálon 10 iterációt végzek, és 1 iteráción belül 20-szor generálok egymás után random sorrendet, akkor összesen n=4096000kísérletet végzek. A 5.1. egyenletbe behelyettesítve annak a valószínűsége, hogy találok megoldást:

$$P_{gr17} = 1 - (1 - 0,258)^{4096000} = 1 \tag{5.4}$$

$$P_{att48} = 1 - (1 - 0,051)^{4096000} = 1 \tag{5.5}$$

Ezek alapján nem meglepő, hogy mindig találtam megoldást a CVRP tesztelése során.

5.3.5. Időablakok feltétele

Most jön annak a vizsgálata, hogy miért nem találtam megoldást olyan sok esetben a CVRPTW mérésekor.

Köszönetnyilvánítás

Szeretném kifejezni hálás köszönetemet konzulensemnek, Dr. Szegletes Lucának, aki már a Témalaboratórium tárgy óta segítséget nyújtott a munkámhoz. Továbbá szeretném megköszönni családomnak, barátaimnak, hogy észrevételeikkel támogattak abban, hogy dolgozatom helyesírási és fogalmazási hibáktól mentesebb lehessen.

Irodalomjegyzék

[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, 2022.

[3] Claudemir Woche V. C.: Modeling and solving the traveling salesman problem with python and pyomo. URL http://www.opl.ufc.br/en/post/tsp/.

[4] Cooperative groups: Flexible cuda thread programming. https://developer.nvidia.com/blog/cooperative-groups/. Accessed: 2017-10-04.

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

[6] Cuda c++ programming guide - cooperative groups thread block tile. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#thread-block-tile-group-cg.

[7] Cuda c++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 2020-04-27.

[8] cudalaunchcooperativekernel documentation. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EXECUTION.html#group__CUDART__EXECUTION_1g504b94170f83285c71031be6d5d15f73.

[9] Cvrplib - capacitated vehicle routing problem library. http://vrp.galgos.inf.puc-rio.br/index.php/en/.

[10] Datasets fot traveling salesperson problem - florida state university. https://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html.

[11] Prof. Wu Feng: To gpu synchronize or not gpu synchronize? https://www.nvidia.com/content/GTC/documents/SC09_Feng.pdf.

[12] Michael Held – Richard M. Karp: A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10. évf. (1962) 1. sz., 196–210. p. URL https://doi.org/10.1137/0110015.

[13] Rishal Hurbans: Ant colony optimization for beginners. URL https://rishalhurbans.medium.com/ant-colony-optimization-for-beginners-841d2be60cc6.

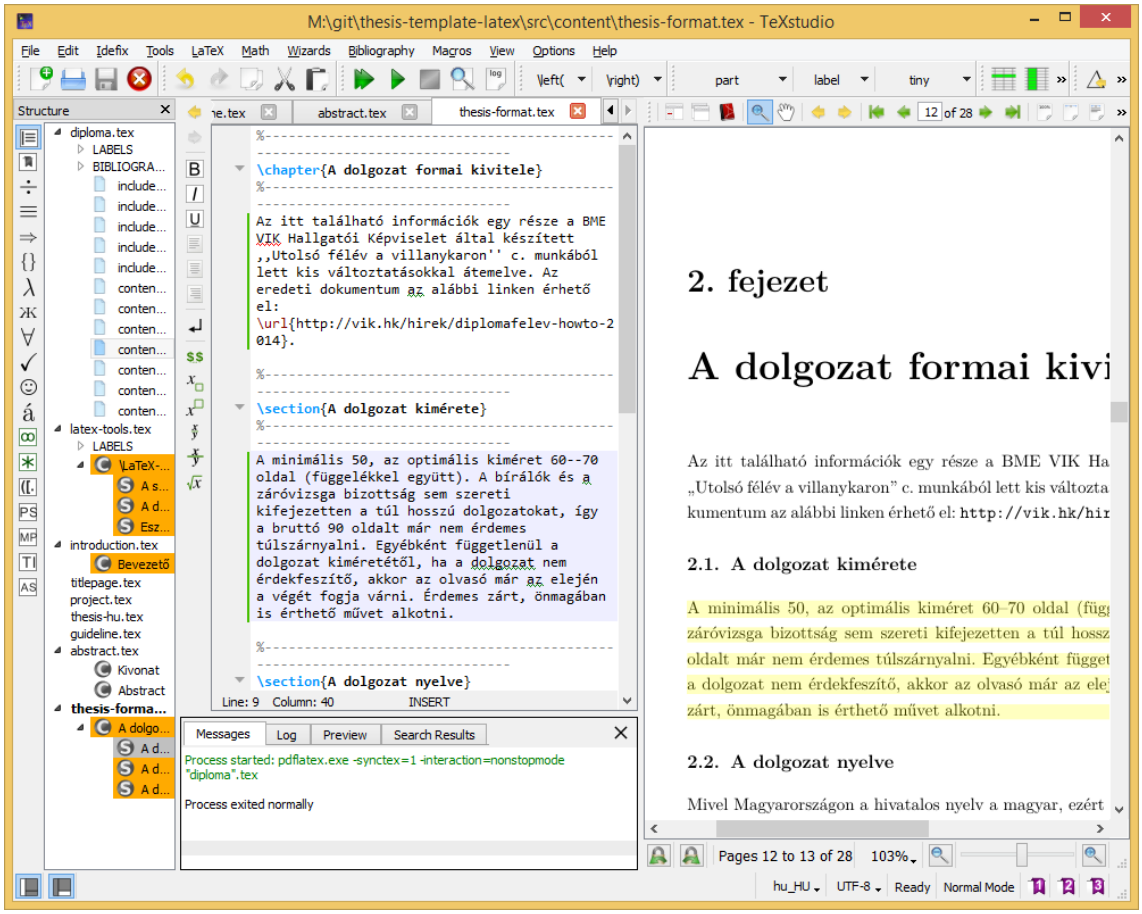
[14] Támogatott windows operációs rendszerek listája. https://learn.microsoft.com/en-us/microsoft-365/security/defender-vulnerability-management/tvm-supported-os?view=o365-worldwide.

[15] Vrptw benchmark problems. http://web.cba.neu.edu/~msolomon/problems.htm.

[16] Király Zoltán: Kvantummechanikai számítások optimalizálása, 2022.

Függelék

F.1. A TeXstudio felülete



F.1.1. ábra. A TeXstudio LaTeX-szerkesztő.

Ehelyett a github elérhetőséget

F.2. Válasz az „Élet, a világmindenség, meg minden” kérdésére

A Pitagorasz-tételből levezetve

$$c^2 = a^2 + b^2 = 42. \tag{F.2.1}$$

A Faraday-indukciós törvényből levezetve

$$\operatorname{rot} E = -\frac{dB}{dt} \qquad \longrightarrow \qquad U_i = \oint_{\mathbf{L}} \mathbf{Edl} = -\frac{d}{dt} \int_{A} \mathbf{Bda} = 42. \tag{F.2.2}$$