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 7.

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 7.

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.

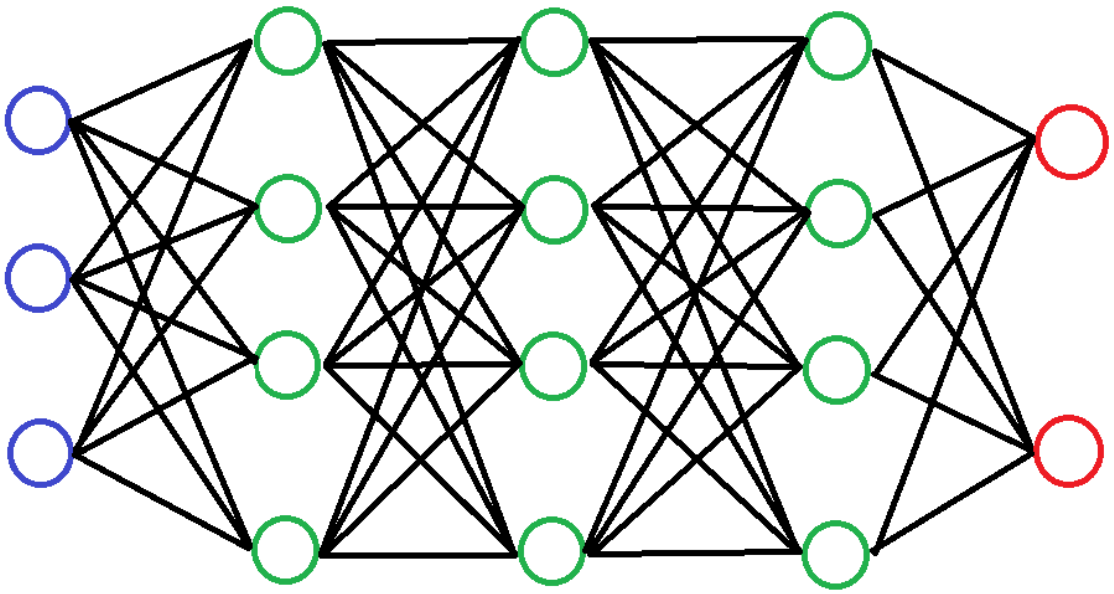
Abstract

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

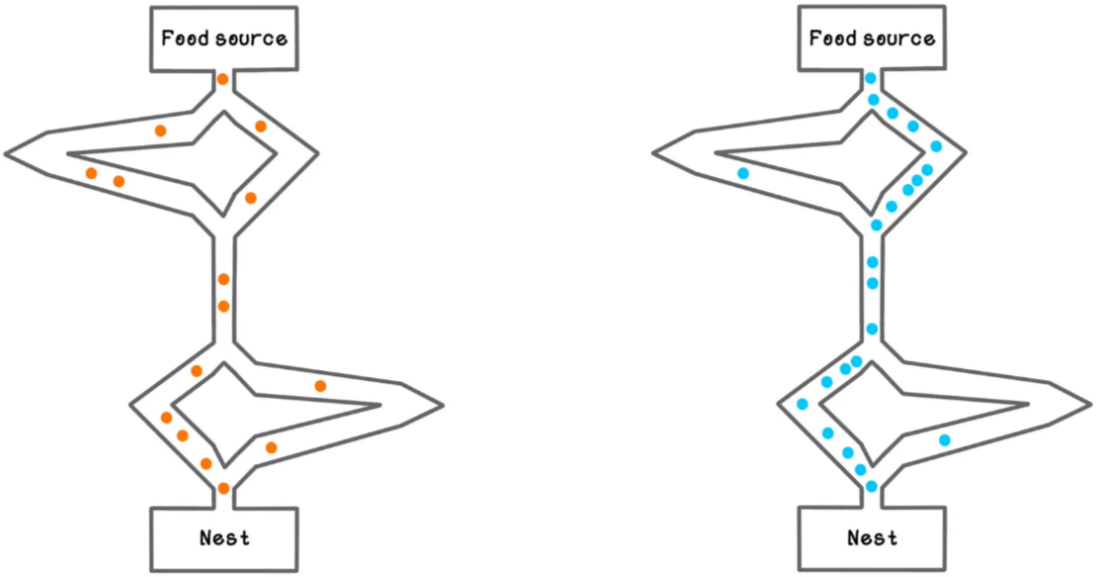
Bevezetés

A konzulensem, Dr. Szegletes Luca már régóta gyűjt videokártyás kódimplementációkat különböző NP-nehéz problémákra. Amikor megbeszéltük, hogy mi lenne a dolgom, 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 dukálnak: például ha egy bútor szállítmányozási cég akarja kitalálni, hogy az aznapi 20 klienséhez milyen sorrendben juttassa el az árukat, már azt is bele kell kalkulálnia, hogy hogyan fognak a termékek beleférni egy kamionba (véges kapacitás problémája). Másik példa egy nemzetközi körutazás: a cég vezérigazgatója be szeretné járni egy kampány keretében a cég különböző leányvállalatait, melyek a mai világ 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 szeretné magát fogadtatni a helyi vezetőkkel (a kliens nem korlátlanul áll rendelkezésre, igazítani kell a fogadó és vendég beosztását). Még számtalan hasonló szituáció képzelhető el, ezek, amiket említettem csak a legtipikusabbak. Ezen problémákat a számítástudomány már évtizedek óta számon tartja, hátha lehet előrébb jutni velük.



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

A mai világban a mesterséges intelligencia forradalmasította a számítástechnikát a számítási kapacitások soha eddig nem látott bővülése miatt. Az NVIDIA egy olyan cég ,mely hardvergyártóként, konkrétan videokártyák gyártásával és értékesítésével kezdte. Az elmúlt években komoly szerkezetváltozáson megy 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 program írása során a kódírónak teljes körű elképzelése kell legyen arról, hogy hogyan fog eljutni az eredményhez.



1.2. ábra. A hangyák sajátos módon optimalizálják a táplálékszerzést: a Hangyakolónia Optimalizáció segítségével [12]

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 épnek 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
- at algoritmus adott ideig (pl. 2 óra) fut, a végő 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. Szemléleti a 1.2. ábra

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, **feromon**t bocsát ki magából. A többi hangya megérzi a szagot, és a nyomába megy. 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, ezért ők visszafelé már nem eresztenek anyagot. A szél előbb-utóbb elfújja az úton hagyott feromont, ezért nem megy oda több hangya.

1.1. Fejezetek

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

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.

2.1. Utazóügynök probléma

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

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

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

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

elérhető utak halmaza (Edge),

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

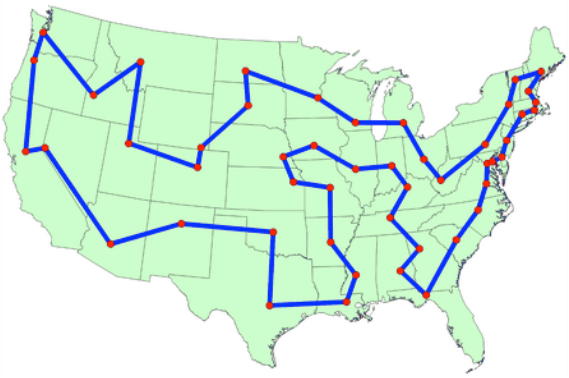
költségfüggvény (Distance).

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

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

bejárás, hogy $\forall v_i \in V, v_i \in R$ 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 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



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

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. Jármű útvonaltervezési problémák

A jármű útvonaltervezési 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

Feltételezem, 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 elérhető járművek maximális száma.

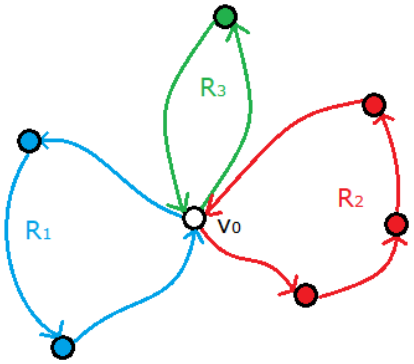
$$V = (v_0, v_1, \dots v_n)$$

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

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

elérhető utak halmaza (Edge),

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



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

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 algoritmus [2]

A 2.1. és 2.2. fejezetekben ismertetett problémákra az optimális megoldás megtalálása NP-nehéz feladat, tahát 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 biztosítani. 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 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. Saját megvalósításom esetében elhanyagoltam ezen ponton az élhosszak figyelembe vételét, ezért $\beta = 0$. Továbbá az egyszerűség kedvéért legyen $\alpha = 1$. Ennek az lesz az előnye, hogy a 2.2. egyenlet 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. Összegezzük, hogy az egyes iterációk végén micsoda történik egy él feromonjával:

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

Az útkeresés közben mindig fel kell jegyezni a legjobb addigi megtalált utat. Az ACO algoritmus egyik előnye, hogy több, hasonlóan jó alternatív utat is képes megtalálni. Ez például térképes útvonaltervezésnél lehet hasznos. 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 látni fogjuk, hogy a helyzet bonyolódhat. UTALÁS

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 kiindulási gondolatot.

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. **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 csakcsak 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, márpedig szeretnénk, akkor ravaszabbnak kell lennünk.

2.4.2. Held - Karp algoritmus

A Held-Karp algoritmust M. Held és R. Karp urak alkották meg 1962-ben [11]. 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 kezdőpontnak. 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 $\neq v_1$ csúcsba végződő legrövidebb út hossza. Az u-ból a v-be mutató él költsége d(u,v). Lépésenként kiszámítjuk a g(S,e) értékeket kezdte 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 visszatérünk a UTALÁS 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™ által erre kifejlesztett párhuzamos számítási platformot, a CUDA keretrendszert. Részletezem a megértéshez szükséges fontosabb fogalmakat, valamint bemutatom a CUDA platformon fejlesztéshez elengedhetetlen lépéseket. A fogalmak összegyűjtéséhez nagymértékben ihletett nyújtott többek között Király Gábor diplomamunkája, mely hasonló témakörrel foglalkozik. [13]

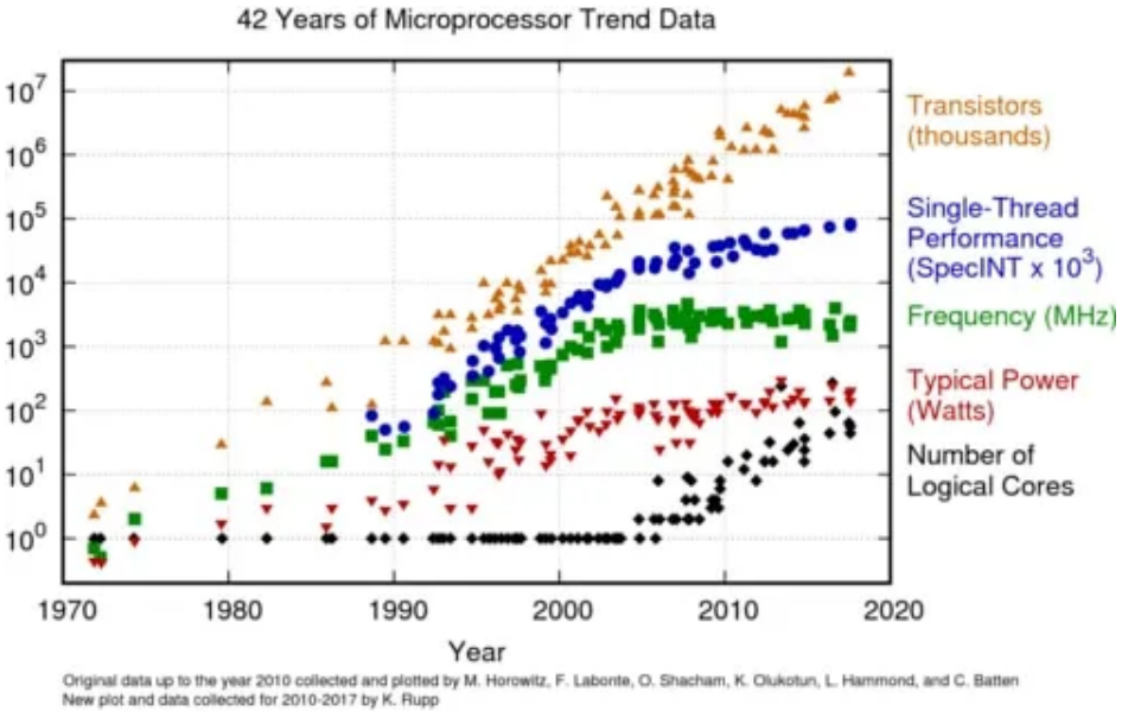
3.1. GPGPU

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. [13] 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 utasításvégrehajtást 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 ugorni velük sem lehetséges. A számítógépgyártók mai napig új, alternatív megoldásokat keresnek a számítási teljesítmény növelésére. Nap-

jainkban 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.



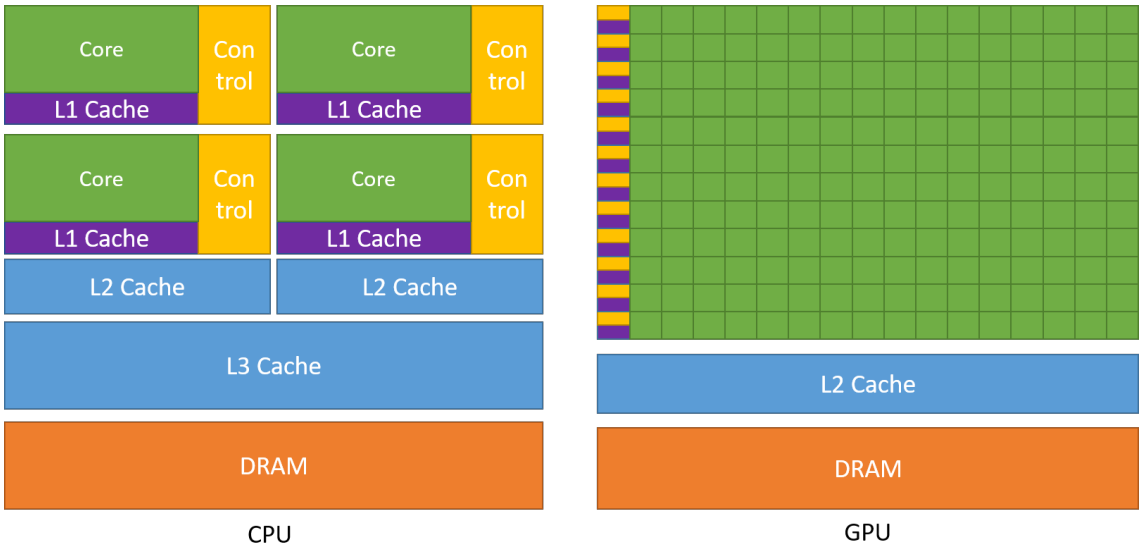
3.1. ábra. Látható, hogy kb. 2010-re befejeződött a CPU-k órajelfrekvencia-növekedése, helyette egyre nőni kezdtek a magok számai. [1] POCSÉK KÉP, LECSERÉLNI

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). [13] 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 a CUDA (Compute Unified Device Architecture) rendszerét használtam. A CUDA egy, az NVIDIA által fejlesztett párhuzamos számítási platform és API (felhasználói interfész), amely



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

szoftveres támogatást nyújt az ezzel kompatibilis grafikus feldolgozóegységek általános célú programozására [13].

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

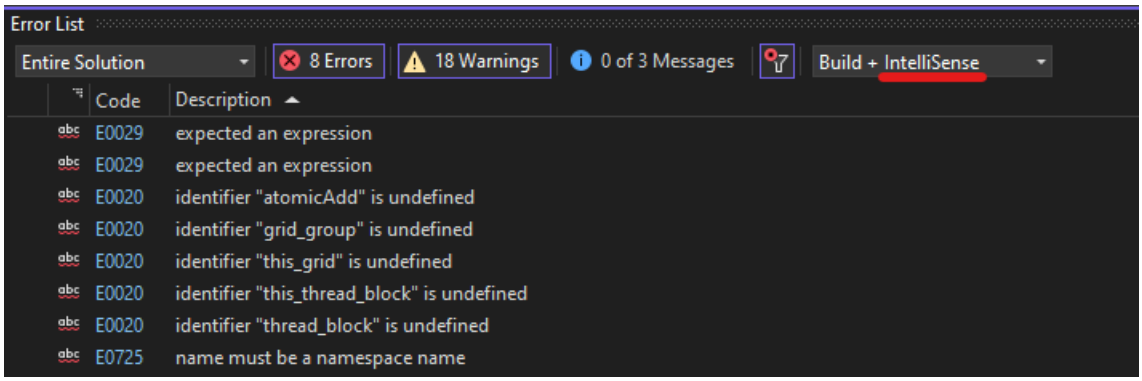
A továbbiakban összefoglalom a legfontosabb fogalmakat úgy, hogy ismertetem, azok hogyan lettek megvalósítva C++ programozási nyelven.

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, ekkor tudja tudni a szoftverkörnyezet fordítóprogramja, a compiler, hogy mostantól GPU kódként kell értelmezze a programot.

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 írta 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.

Az, hogy a kernelt egy adott híváskor hány CUDA szálon szeretnénk futtatni, az új nyelvi elemként megjelenő «< · · · »> végrehajtási konfiguráció szintaxissal tudjuk specifikálni. Sajnos a Visual Studio még szintaxishibaként kezeli verziószám?, ezért a programozónak érdemes odafigyelni, hogy milyen IntelliSense hibaüzeneket vesz figyelembe. A 3.3. ábrán például olyan fogalmakra jelez a fejlesztői környezet, amelyek a keretrendszer adta bővítményekben vannak definiálva. Minden, a kernelt végrehajtó szál egy egyedi thread azonosítót kap, mely a beépített threadIdx változón keresztül érhető el a kernelből.



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 [4].

Hatékony párhuzamosított programnak három alappillére van: [13]

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, hogyha 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 - 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. 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();

  // Each iteration halves the number of active threads
  // Each thread adds its partial sum[i] to 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. [13] Egy thread blokk példánya az alábbi módon inicializálható:

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

Azon threadek, amelyek ugyanazon beépített CUDA blockIdx értékkel rendelkeznek, ugyanazon thread blokkba tartoznak. A blokkok szinkronizálása nagyon hasonló a korábban emített ___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:

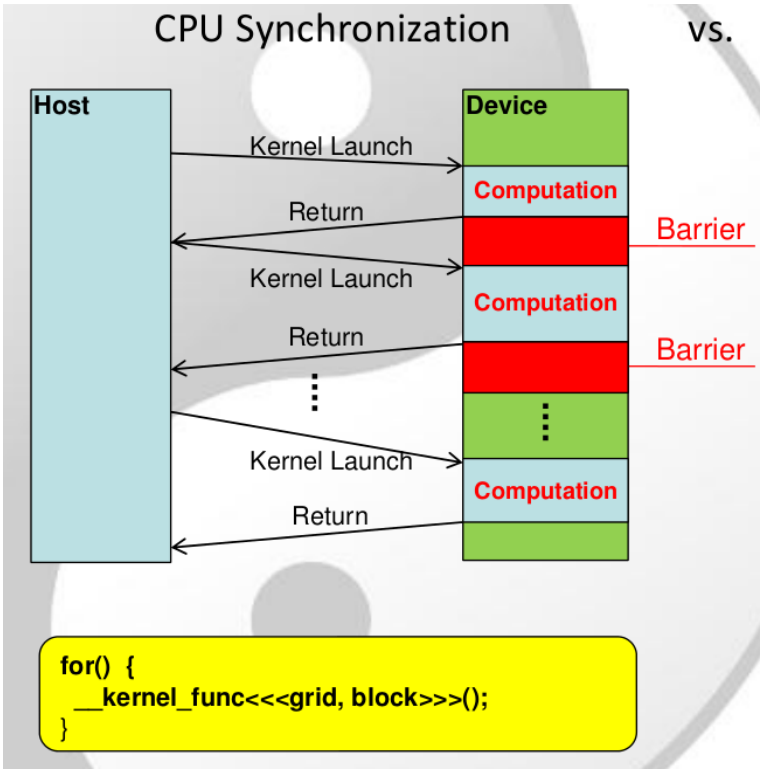
```
grid_group grid = this_grid();
```

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 modell natív támogatást csupán thread blokkokon belüli összehangolásra nyújtott. A régi gyakorlat az volt, hogy amennyiben grid szinten akartunk szinkronizációt alkalmazni, a kernelt felbontottuk több kisebb alkernelre, majd azon pontokon, ahol grid szintű szinkronizációra vágytunk, ott befejeztük az adott alkernelt, és hívtuk az újat. Ezen módszerre sokan "CPU Szinkronizáció" vagy "Implicit Szinkronizáció" néven hivatkoznak. Az 3.4. ábra szemlélteti a módszer lényegét. [10]



3.4. ábra. A kooperatív csoportok megjelenése előtt az egyetlen járható út teljes GPU szinkronizálásra. Látható, hogy gyakorlatilag minden programozási alapelvnek ellentmond. UTALÁS

A módszer több szempontból problémás. Egyrészt egy kernel hívása sok erőforrást igényel, ezért lehet, hogy sok kernelhívás mellett lassabb lesz a program, mintha egy szálon CPU-n futtattuk volna. Másrészt ha az eredeti függvényben ciklus iterációnként 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();
```

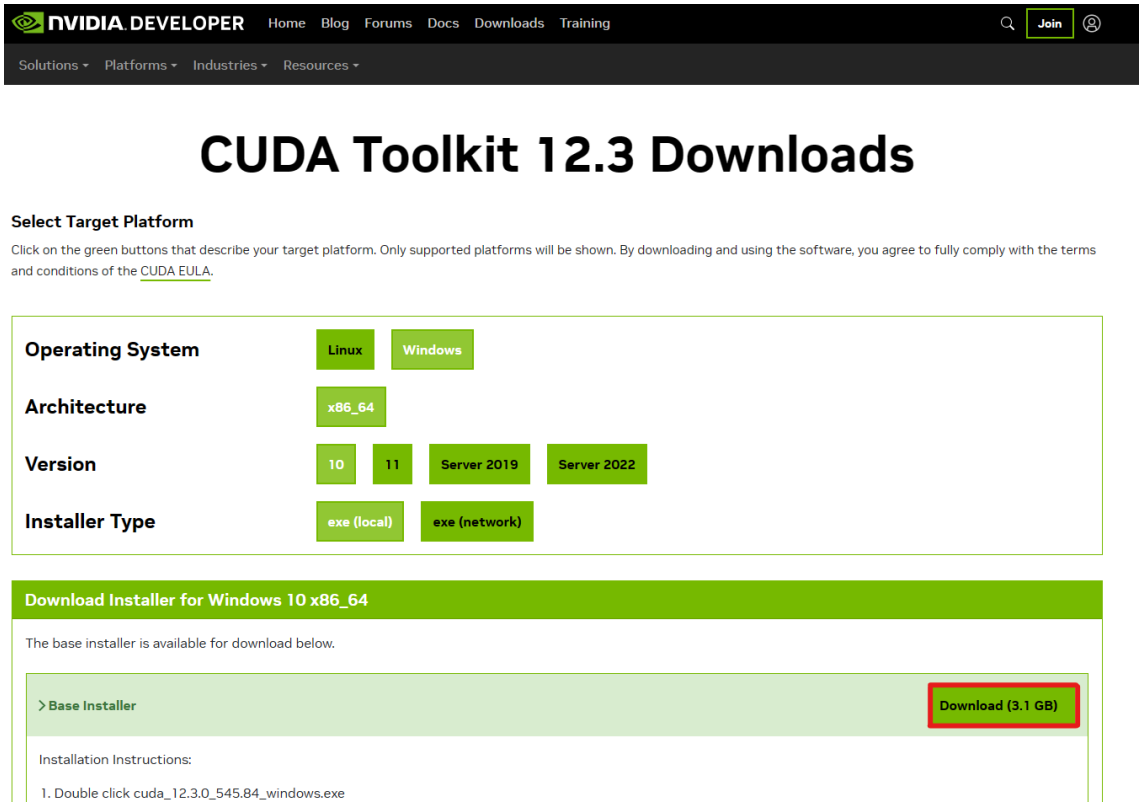
A főbb különbséget ott tapasztaljuk, amikor a kernel hívására kerül sor. A szokásos < < <...> > > konfigurációs szintaktika helyett a CUDA runtime API hívást kell végrehajtani (vagy annak driver megfelelőjét) [8]

HIÁNYZIK A STORY FELE :(

3.2.4. CUDA használata Visual Studio alatt

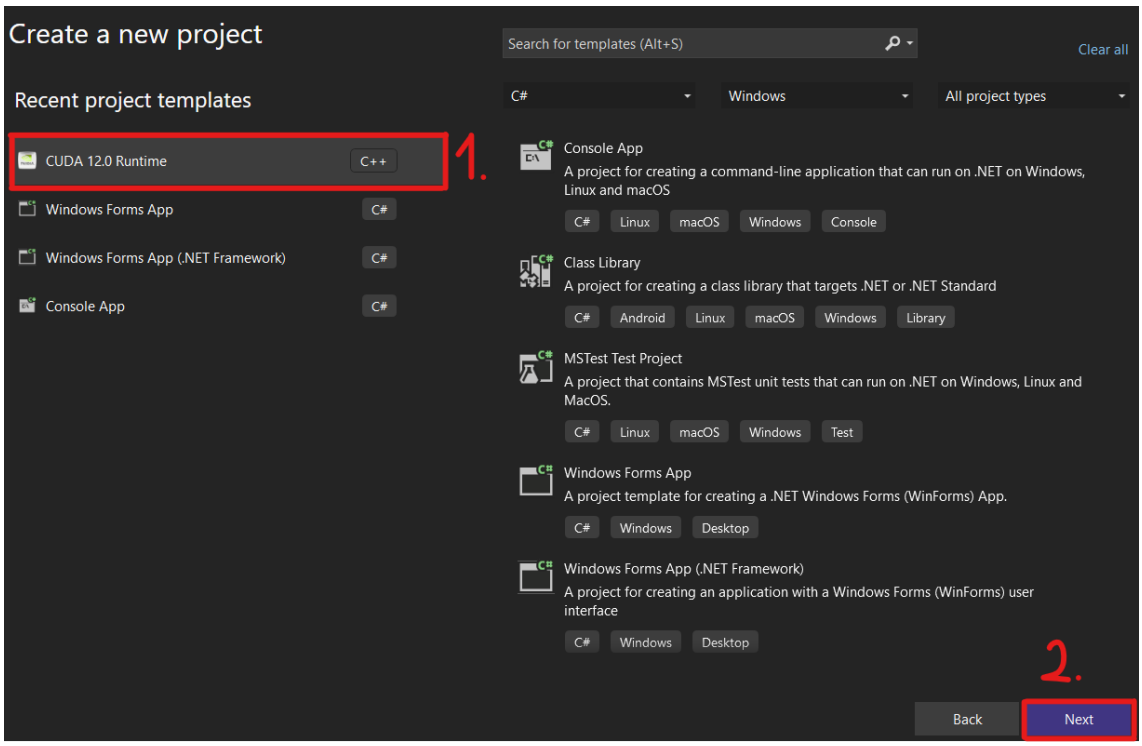
Szeretnék röviden leírást nyújtani az első CUDA nyelven megírt program létrehozásához.

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.



3.5. ábra. 1. lépés: A CUDA letöltése

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.



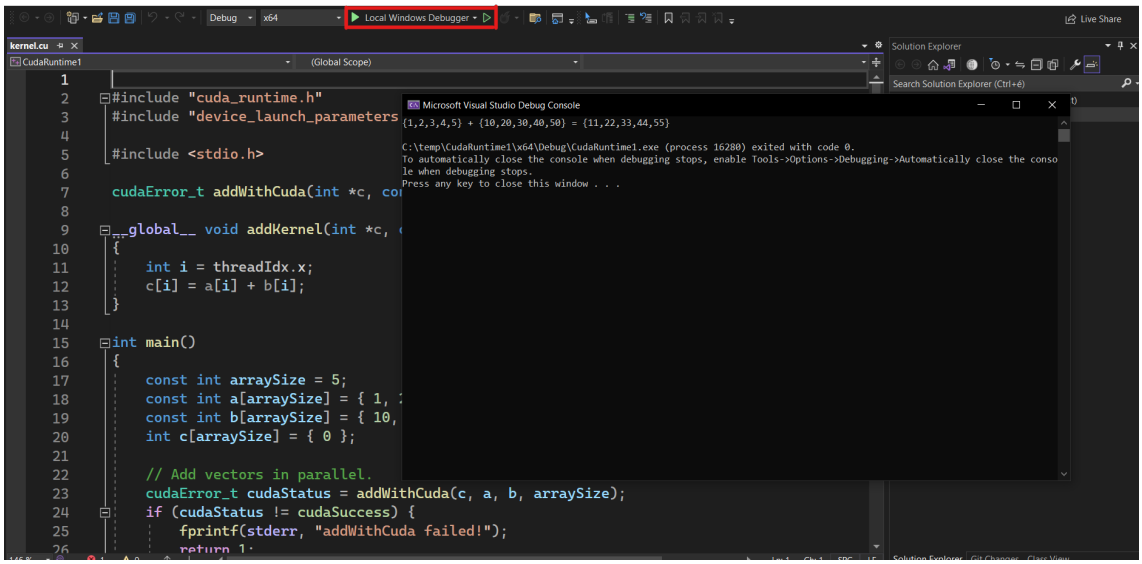
3.6. ábra. 2. lépés: Új CUDA projekt létrehozása

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.



3.7. ábra. 3. lépés: Példakód futtatása

4. fejezet

Implementáció

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

4.1. Hangyakolónia algoritmus

A munkám során a felsorolt algoritmusok mindegyikét a hangyakolónia optimalizációval oldottam meg.

4.1.1. Adatstruktúrák

Az útvonaltervezési problémák gráfokon futnak, ezért elsődleges feladatom annak eldöntése volt, hogy milyen formában tároljam a gráfokat. Egyik legalapvetőbb megadási mód a szomszédossági mátrixával történő reprezentáció. Ennek előnye, hogy az egyes csúcsokhoz tartozó élek könnyen, direkt lekérdezhetőek, ezért a rajtuk végzett műveletek gyorsabbak lehetnek. Hátránya azonban, hogy minél kevesebb él van a gráfban, annál pazarlóbbá válik ez a megadás. Később tekintünk egyéb megadási módokat is, de most azokkal nem foglalkozok. Az adattípusok megválasztásában eltérőek az egyes megvalósítások: az első TSP verzióban például minden lebegőpontos számot *double* típusú változóban tárolok, míg a többi algoritmusnál *float* adatípust 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 egyszeres pontossággal.

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.

A TSP során egy NxN-es mátrix által adott feromongráfon egyértelműen végrehajtható mohó algoritmus, vagyis olyan döntéshozatal, 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. Az inputgráf egyszerű

BEFEJEZETLEN SZÖVEGRÉSZ

4.1.3. A rulettkerék algoritmus [2]

Az algoritmusaimban újra és újra feltűnik a random számok kezelése. 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 is 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 is úgynevezett pszeudorandomszám-generátor elvét követi. Ez tökéletesen meghatározható egy kezdeti érték alapján, 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. 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. Éppen ezért érdemes lehet többször (például 10-szer) egymás után végrehajtani az algoritmust, és megvizsgálni az egyes eredményeket. Ilyenkor végső eredménynek célszerű a legrövidebb megoldást venni. Sajnos a tesztelések során azt tapasztaltam, hogy a GPU főleg amikor több szálon dolgozik, mint ami fizikailag elérhető CUDA core formájában, gyakran hibázik, és nem vagy rosszul ír át memóriaterületeket. Azon memóriaterület, mely tárolja azt aktuális minimális szekvenciát, rendkívül ki van téve

Hardver ismertetése

5. fejezet

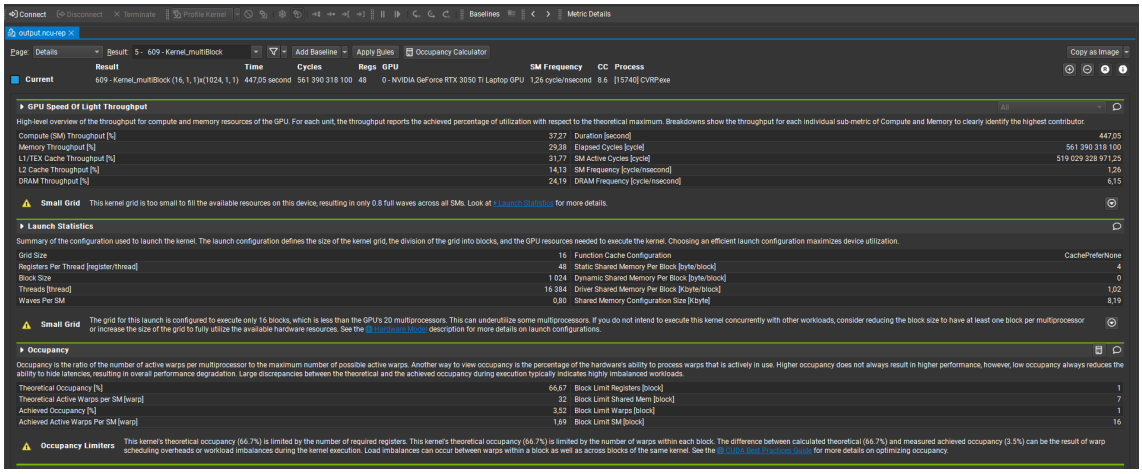
Eredmények

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 5.1. ábrán ábrázoltam.



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

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

5.2. Mérési eredmények

IDE VALAMI BEVEZETÉS

5.2.1. TSP első verzió

Teszteléshez szükségem volt ismert eredményű adathalmazokra. A Floridai Állami Egyetem weboldalán [9] 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).

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

5.1. táblázat. Elsőnek egy kicsi, 4 állomásból (és a 0. kiindulási pontból) 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	291
2048 ant	1,18	365	327
4096 ant	1,20	359,7	332
10 rep			
1024 ant	11,39	350,4	291
2048 ant	11,47	328	295
4096 ant	11,59	336,8	291

5.2. táblázat

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

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

5.3. táblázat

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

5.4. táblázat

DANTZIG42 : 42 csúcs, minimális út : 699, átlagos út : 3110,5			
	Futásidő (s)	Végeredmény átlag	Végeredmény min.
10 rep			
1024 ant	115,41	1735,7	1554
2048 ant	113,94	1420	1252
4096 ant		1263,8	1106
8192 ant		1082,6	928
16384 ant	114,34	987,7	906

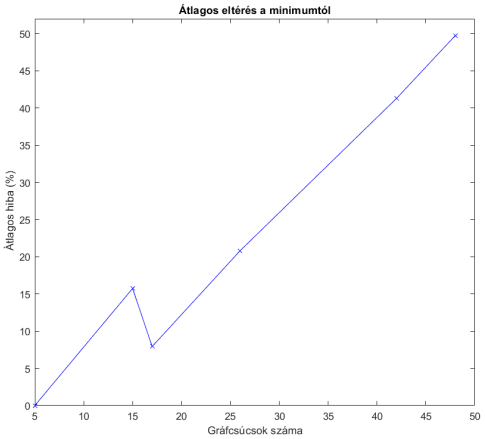
5.5. táblázat

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

5.6. táblázat

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 [9] helyről származó csomag.



5.2. ábra. Matlabbal ábrázoltam a gráfcsúcsok számának függvényében az elméleti minimumhoz képesti átlagos hibát: TSP első verzió

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

5.7. táblázat

GR : 17 csúcs, minimális út : 2085			
	Futásidő (s)	Végeredmény átlag	Végeredmény min.
10 rep			
1024 ant	3,54	2246	2094
2048 ant	3,69	2237,7	2207
4096 ant	3,69	2196,5	2142
8192 ant	3,89	2211,5	2170
16384 ant	4,37	2179,2	2129
20480 ant			

5.8. táblázat

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.

FRI26 : 26 csúcs, minimális út : 937, átlagos út : 2693			
	Futásidő (s)	Végeredmény átlag	Végeredmény min.
10 rep			
1024 ant	9,00	1393	1347
2048 ant	9,23	1363,1	1288
4096 ant	9,43	1215,3	1163
8192 ant	9,72	1136,9	1055
16384 ant	10,82	1104,9	1007
20480 ant			

5.9. táblázat

DANTZIG42 : 42 csúcs, minimális út : 699, átlagos út : 3110,5			
	Futásidő (s)	Végeredmény átlag	Végeredmény min.
10 rep			
1024 ant	26,19	1601,3	1018
2048 ant	26,61	1488,4	1319
4096 ant	27,02	1286,7	1168
8192 ant	27,57	1118,3	1004
16384 ant		1014,7	909
20480 ant			

5.10. táblázat

ATT48 : 48 csúcs, minimális út : 33523, átlagos út : 157686,9			
	Futásidő (s)	Végeredmény átlag	Végeredmény min.
10 rep			
1024 ant	38,12	84660,7	67541
2048 ant	38,63	72522,8	64969
4096 ant	39,14	62808,6	53395
8192 ant	39,69	59604,8	52514
16384 ant	41,56	48618,2	46227
20480 ant	124,72	49879,9	45836

5.11. táblázat

	Futásidő (s)	Végeredmény átlag	Végeredmény min.
10 rep			
1024 ant			
2048 ant			
4096 ant			
8192 ant			
16384 ant			
20480 ant			

5.12. táblázat

5.2.4. CVRP

Az első olyan algoritmus, amely feltételes útvonaltervezést igényel, a Kapacitásos jármű útvonaltervezési feladat. Adathalmazt a HIÁNYZIK helyről szedtem. Az iterációk az elő-

zőeknél látottakkal megegyezőek: 1 rep-en belül 20 random hangyát követ 500 tudatos hangya.

	Futásidő (s)	Végeredmény átlag	Végeredmény min.
10 rep			
1024 ant			
2048 ant			
4096 ant			
8192 ant			
16384 ant			
20480 ant			

5.13. táblázat

	Futásidő (s)	Végeredmény átlag	Végeredmény min.
10 rep			
1024 ant			
2048 ant			
4096 ant			
8192 ant			
16384 ant			
20480 ant			

5.14. táblázat

5.3. Eredmények értékelése

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

Köszönetnyilvánítás

Szeretném kifejezni hálás köszönetemet konzulensemnek, Dr. Szegletes Lucának, aki már a Témalaboratórium tárgy óta nyújtott segítséget a munkámban. Továbbá szeretném megköszönni a barátnőmnek, hogy egyszer, de professzionális odafigyeléssel készségesen megnyomta nekem az Alt gombot profilozó mérés közben.

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

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

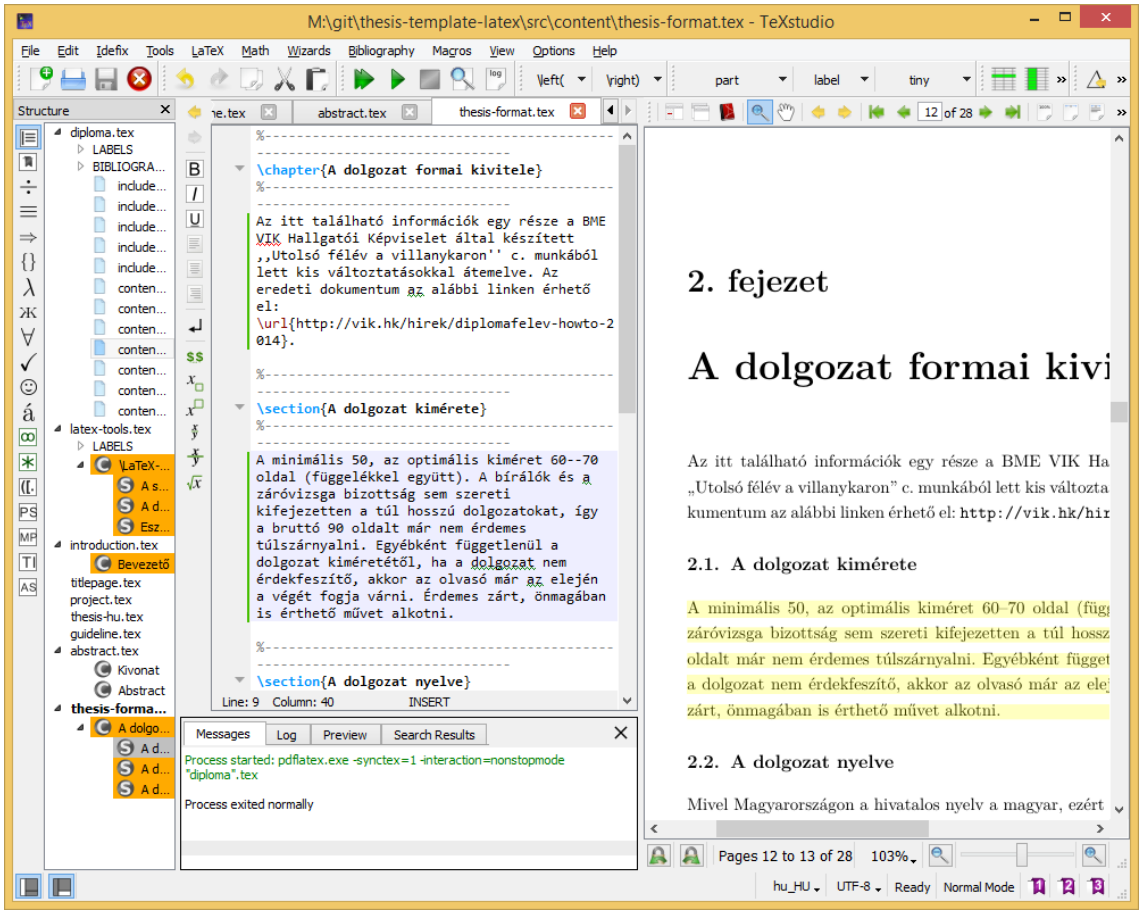
[11] Michael Held – Richard M. Karp: A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10. évf. (1962) 1. sz., 196–210. p. URL https://doi.org/10.1137/0110015.

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

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

Függelék

F.1. A TeXstudio felülete



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

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

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