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. október 25.

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. október 25.

Jost Márk Benedek
hallgató

Kivonat

Különböző hálózatok, legyen az egy szociális média felhasználói hálója, vagy egy ország úthálózata, de akár egy NYÁK-terven a különböző komponensek vezetékezése során alapvető problémák az úgynevezett útkeresési problémák (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 egy heurisztikus algoritmus segítségével, az úgynevezett Hangyakolónia Optimalizáció (ACO) elvével egy heurisztikus, vagyis valószínűségi megoldás találása polinomidőben. Az ACO nagymértékben párhuzamosítható, melyet azzal tudtam kihasználni, hogy videokártyán, szálak 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

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 [2]

Az utazóügynök probléma (Travelling Salesman Problem, továbbiakban: TSP) egy fajta optimalizációs probléma, mely során egy járműnek 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.

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

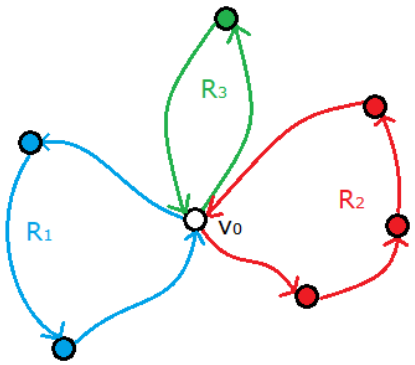
az SMD beültetőgép bejárása áll. A gép feladata, hogy egy adott nyomtatott áramköri terv alapján lepakolja az alkatrészeket a hordozó lapkára. Az iparban fontos a sebesség, ugyanis ha felére csökkentjük a beültetési időt, akkor akár duplaannyi terméket gyárthatunk le azonos idő alatt. Egy szerelőlemezre alkatrészek százai kerülhetnek, ami nagyon sokféleképpen rendezhető sorba. Természetes igényünk rövid idő alatt gyors útvonalat találni a beültetőfej számára. A TSP-re KÉPET. ábrán látható egy látványosabb, vizuális szemléltető példa.

2.2. Jármű útvonaltervezési problémák [2]

A jármű útvonaltervezési probléma (Vehicle Routing Problem, továbbiakban: VRP) tekinthető a TSP általánosításának. 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 2.1. ábrán látható egy vizuális szemléltető példa.



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

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),

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

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 $T_{min}(v_i) \leqslant t(v_i) \leqslant T_{max}(v_i)$

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

Az eljárás a nevéből adódóan a hangyák (Formicidae) természetben is megfigyelhető élelemkeresési módszerén alapszik. Az első hangyák véletlenszerű (random) útvonalakon haladva keresik az élelemhez vezető utat, majd a visszaúton feromonnal jelölik meg az útjukat. Ezt szagolva a többi hangya későbbmár megfontoltabb döntést hozhat a bejárást

illetően. A hátrahagyott feromon erőssége idővel gyengül (modellezhető exponenciális lecsengéssel), ezzel biztosítva, hogy a nem optimális útvonalak (az élelemhez vezet, de már van nála rövidebb) maguktól elhaljanak. Látható, hogy olyan él, ami sok ideig nem kap feromon utánpótlást, exponenciális jelleggel lecseng.

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

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

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

ahol N_i^k az algoritmus k-adik lépésében az i-edik csúcsból elérhető szomszédos csúcsok halmaza. Az α és β a feromonok és élsúlyok figyelembevételét szabályozza. Például minél nagyobb valahol a feromon, annál inkább akarunk oda továbbmenni, illetve minél messzebb van egy adott pont, annál inkább el akarjuk kerülni. Saját megvalósításom esetén gyorsítás céljából elhanyagoltam ezen ponton az élhosszak figyelembe vételét, ezért $\beta = 0$. Továbbá az egyszerűség kedvéért legyen $\alpha = 1$. Ennek az az előnye, hogy a 2.2. egyenlet a következővé egyszerűsödik:

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

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

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

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

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

2.3. Alternatív megoldások

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

2.3.1. „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 ez 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 csak pl. $n = 48$ db csúcsunk van (mert esetleg 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, 5GHz-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 egy szerelőlemezre akár alkatrészek százait pakolhatja fel, ilyen módszerrel lehetetlen lenne megmondani, hogy milyen sorrendben haladjon. Látható tehát, hogy ha véges időn belül meg akarjuk oldani a problémát, ravaszabbnak kell lennünk.

2.3.2. Held-Karp algoritmus

A Held-Karp algoritmust M. Held és R. Karp urak alkották meg 1962-ben [6]. Azért beszédes ez az évszám, mert ekkoriban kezdett magának teret hódítani a számítástudomány, és ekkor készültek el az első számítógépek. Módszerük, hogy egyesével, rendezetten szúrnak be csúcsokat egy egyre növekvő ponthalmazba. Hasonlít a beillesztéses rendezésre, csak kicsit komplexebb. Kihasználja, a korai számítógépek képesek voltak már rekurzív programok futtatására.

Jelöljük a csúcsokat $V = (v_1, v_2, \dots v_n)$ -nel, v_1 önkényesen kijelölhető kezdőpontnak. Legyen S halmaz a csúcsok valamely 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-ekkel.

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 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.

2.3.2.1. 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. Hátránya azonban, hogy rekurziót alkalmaz, ami gyakran kedvezőtlen programvégrehajtás szempontjából (sok rekurzív függvényhívás megterheli a stacket). Egyéb észrevétel, hogy ha különböző 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.

3. fejezet

Technológiai háttér

Ebben a fejezetben szeretném ismertetni az általam 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 bemutatom a CUDA platformon fejlesztéshez elengedhetetlen lépéseket. A fogalmak összegyűjtéséhez nagymértékben ihletett nyújtott többek között Király Gábor munkája hasonló témakörben. [7]

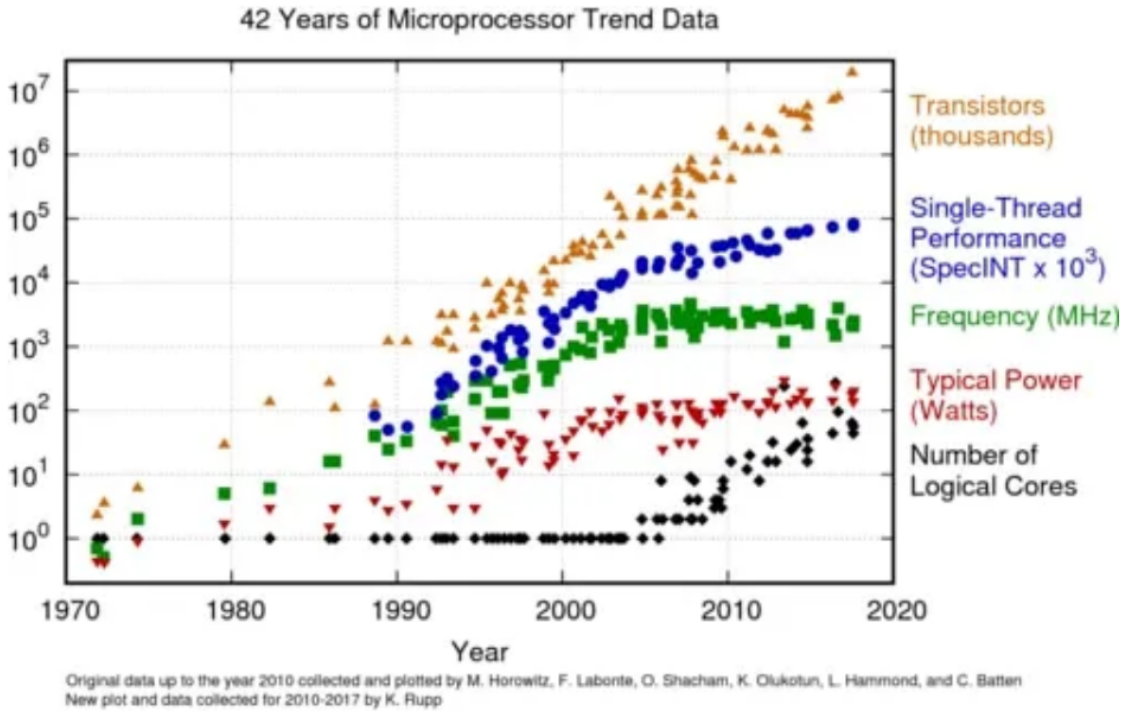
3.1. GPGPU

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. [7] 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 is alkalmas általánosabb megközelítésekre is, többek között bizonyos utasítások kifejezetten jól 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 értelemszerűen frekventáltabb utasításvégrehajtást biztosított, és evidens volt, hogy a nagyobb frekvencia nagyobb számítási erővel 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 frekvenciájának több, mint az ezerszerese. Napjainkban változás látható a fejlesztési trendekben, ugyanis az órajelnövelést a processzorok disszipációja erősen 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 éppen ezért új, alternatív megoldásokat kerestek a számítási teljesítmény növelésére. Legjobb ötlet-

nek a feladatok párhuzamosítása bizonyult. Napjainkban a kutatásoknak két nagy témája van. Egyik a kvantumszámítógépek témaköre, amit dolgozatomban nem részletezek. Másik aktívan vizsgált lehetőség a párhuzamosítás több szálon. Már a CPU-k fejlesztésénél is megfigyelhető, hogy inkább a minél több processzormag telepítése az iparági trend.



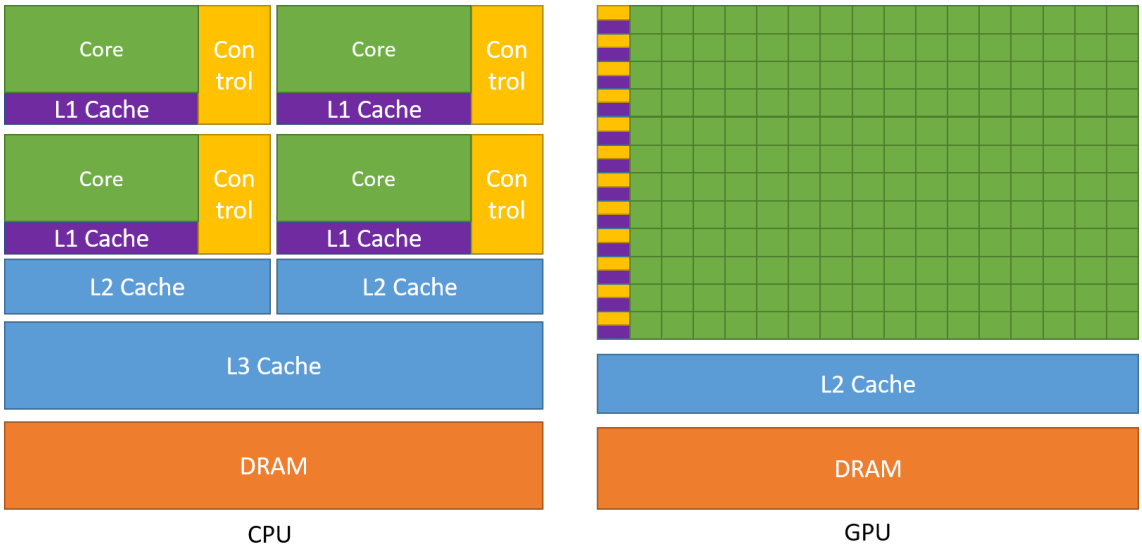
3.1. ábra. Látható, hogy kb. 2010-re befejeződött a CPU-k órajelfrekvencia-növekedése, helyette egyre nőni kezdtek a magok számai. [1]

A párhuzamosításra azonban sokkal alkalmasabb a grafikus segédprocesszor, a GPU. Amíg a CPU feladata, hogy műveletek egy adott szekvenciáját, és annak minden utasítását a lehető leggyorsabban hajtsa végre, addig a GPU célja minél több szál (akár több ezer) párhuzamos futtatása. A videokártyák előnye akkor válik láthatóvá, ha ugyanazt az utasítást több, nagy adattömbön kell végrehajtani. Ez az úgynevezett SIMD megközelítés (Single Instruction Multiple Data). [7] Az **??**. ábra szemlélteti, hogy a GPU-n arányában több tranzisztor van adatfeldolgozásra rendelve, cserébe a gyorsítótárazás és a folyamatvezérlés kisebb hangsúlyt kapott.

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

3.2. CUDA

Többféle keretrendszer is megvalósítja a GPGPU szabta alapelveket. Én a CUDA (Compute Unified Device Architecture) rendszert választottam. A CUDA egy, az NVIDIA által fejlesztett párhuzamos számítási platform és API, amely szoftveres támogatást nyújt az ezzel kompatibilis grafikus feldolgozóegységek általános célú programozására [7].



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

A programozás C vagy C++ nyelven történhet, melyhez minimális nyelvi kiegészítéseket tettek például a szálkezelés egyszerűbb használatához. A CUDA programozás elsajátítása érdekében elérhető egy rendkívül kiterjedt dokumentáció a gyártó weboldalán, melyet jelenleg is folyamatosan frissítenek. [4]

3.2.1. Programozási modell

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

Az, hogy a kernelt egy adott híváskor hány CUDA szálon szeretnénk futtatni, az új nyelvi elemként megjelenő «< · · · »> végrehajtási konfiguráció szintaxissal specifikálható. Sajnos a Visual Studio még szintaxishibaként kezeli [verziószám], ezért a programozó érdemes, hogy odafigyeljen, milyen IntelliSense hibaüzeneket vesz figyelembe. Minden, a

kernelt végrehajtó szál egy egyedi thread azonosítót kap, mely a beépített threadIdx változón keresztül érhető el a kernelből.

Példa: A hivatalos dokumentáció az alábbi példát adja kernel definícióra. A kódrészlet az N méretű A és B vektorok összeadását végzi és az eredményt a C vektorban tárolja:

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

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

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. [3]
Hatékony párhuzamosított programnak három alappillére van: [7]

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

Foglalkozzunk jelen esetben a 3. ponttal, a szinkronizáció kérdésével. Szinkronizáció szükséges ahhoz, hogy elágazó programunk biztonságos, fenntartható és moduláris legyen. A CUDA 9 bevezette az ún. Kooperatív csoport nevű absztrakciót (angolul Cooperative Groups), amely erős támogatást nyújt a kerneleknek ahhoz, hogy dinamikusan alakíthassanak ki csoportokat a szálakon belül.

Korábban a CUDA API egy nagyon egyszerű, ámbár 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 szálas 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.

Leegyszerűsítve, a Cooperative Groups egy API Support Package, ami szálak csoportosítására és szinkronizálását segíti CUDA programokban. Az Package nagy része az összes

CUDA 9-el kompatibilis GPU-ra működik, azaz Kepler és későbbi architektúrákra (Compute Capability 3.0+).

Ahhoz, hogy használhassuk, 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 vagy mindig prefixként kiírjuk, hogy ”cooperative_groups::”, vagy betöltjük a névteret a ”using” direktívával. Én a munkám során az utóbbi 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 terjeszti ki az összes, később tárgyalandó csoport objektum. Alapvető függvények rájuk:
- Megkaphatjuk a csoport méretét, azaz a benne futó szálak számát a size() metódussal

```
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 lehetőségeket, 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 odáig. Az összes thread csoport fajta támogatja a szinkronizálást, azonban mindegyik kicsit másképp.
A g csoporthoz tartozó szálakat a kollektív sync() metódussal, vagy a g-re a cooperative_groups::synchronize() függvényt meghívva szinkronizálhatjuk. Ezek ún. barrier szinkronizációt hajtanak 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 legfontosabb koncepcióját. A szálak egy-, kettő-, vagy háromdimenziós logikai egységbe szervezhetők, amit blokknak 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. 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 thread blokkok szinkronizálása nagyon hasonló a korábban említett ___syncthreads() metódushoz. A következő kódok mind ugyanolyan hatást érnek el: (feltéve, ha a blokk összes szála elér oda)

```
___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 finomítva párhuzamos algoritmusainkat. A cooperative_groups::tiled_partition() függvény felosztja az adott thread blokkot több részegységre. Itt egy példa, amely a teljes thread blokkot 32 szálú részegységekre bontja. A 32 gyakori felosztás, ugyanis a warpok (a CUDA szoftvermodell egyik alapfogalma) 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) 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, ugyanis így nem teszünk hibás következtetéseket elágazó függvényhívások miatt. Az alábbi egy elkerülendő példa szinkronizáció használatakor.

```
__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.

```
// 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:

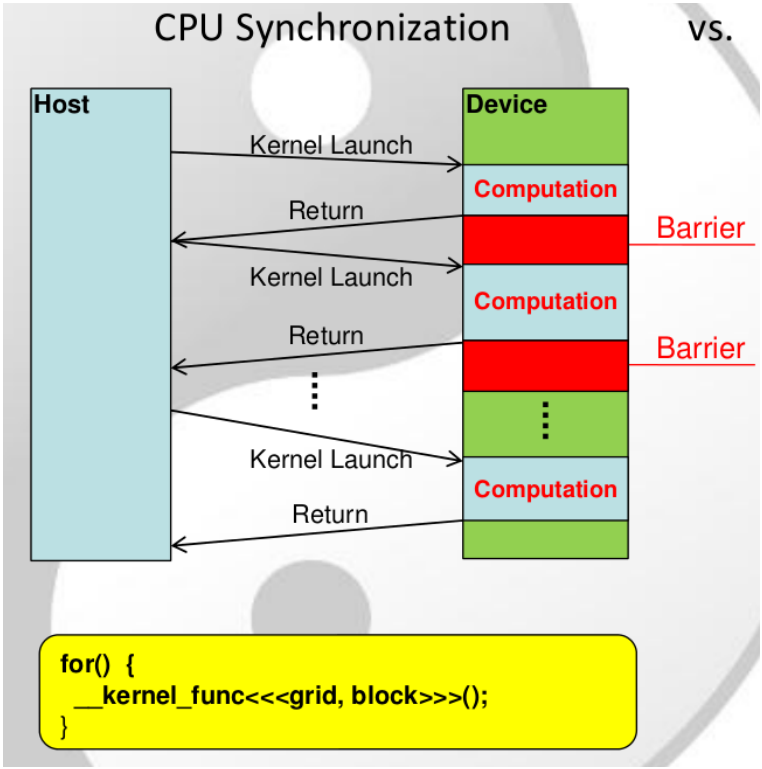
```
grid_group grid = this_grid();
```

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 bokkokon 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 3.3. ábra szemlélteti a módszer lényegét. [5]



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

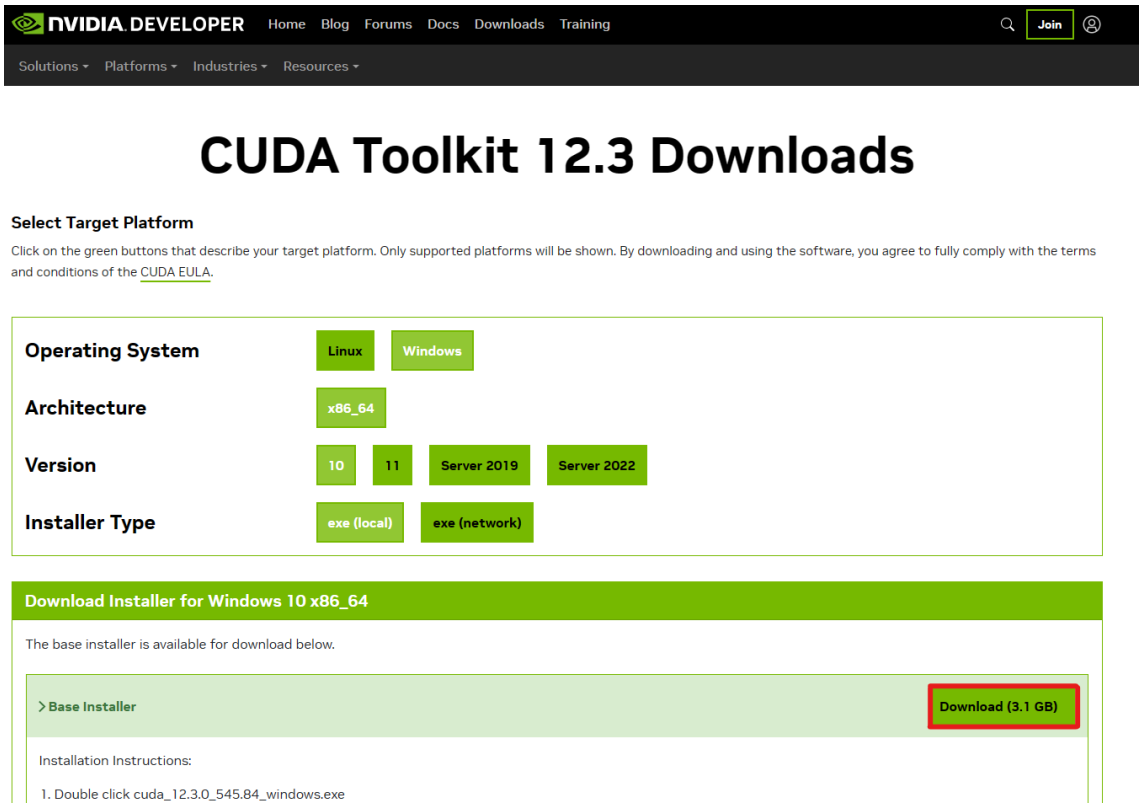
4. fejezet

asd

4.0.1. CUDA használata Visual Studio alatt

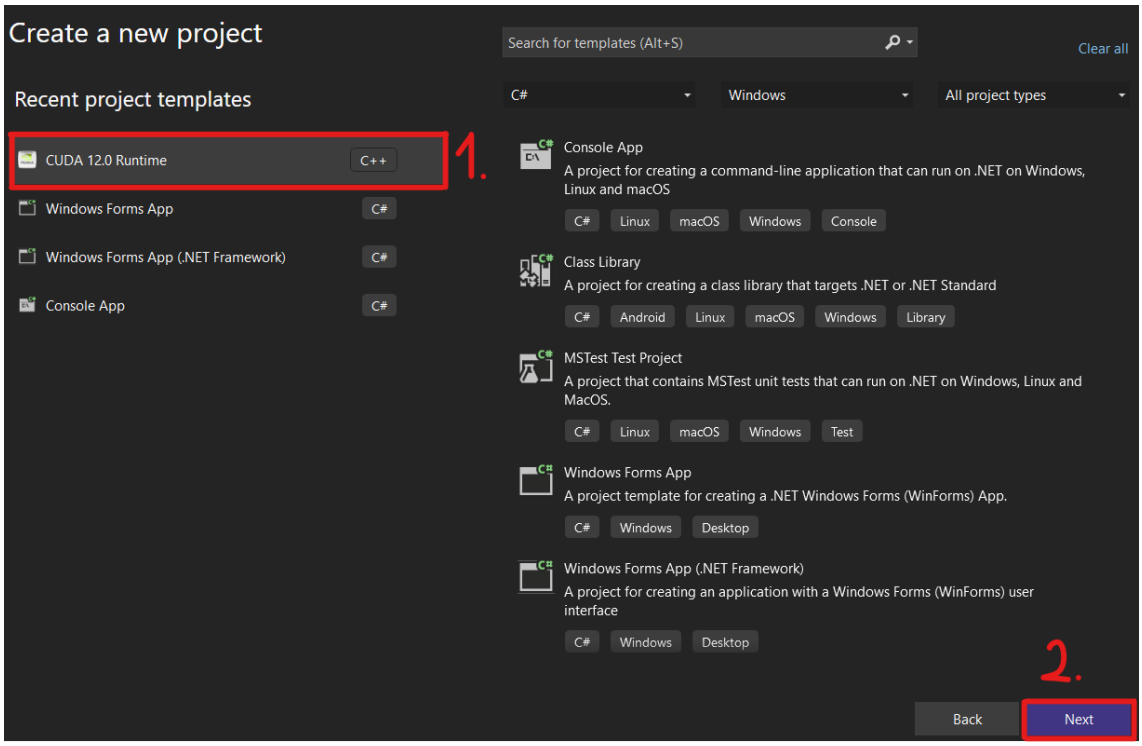
Szeretnék röviden leírást nyújtani az első CUDA nyelven megírt program létrehozásához.

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.



4.1. 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.



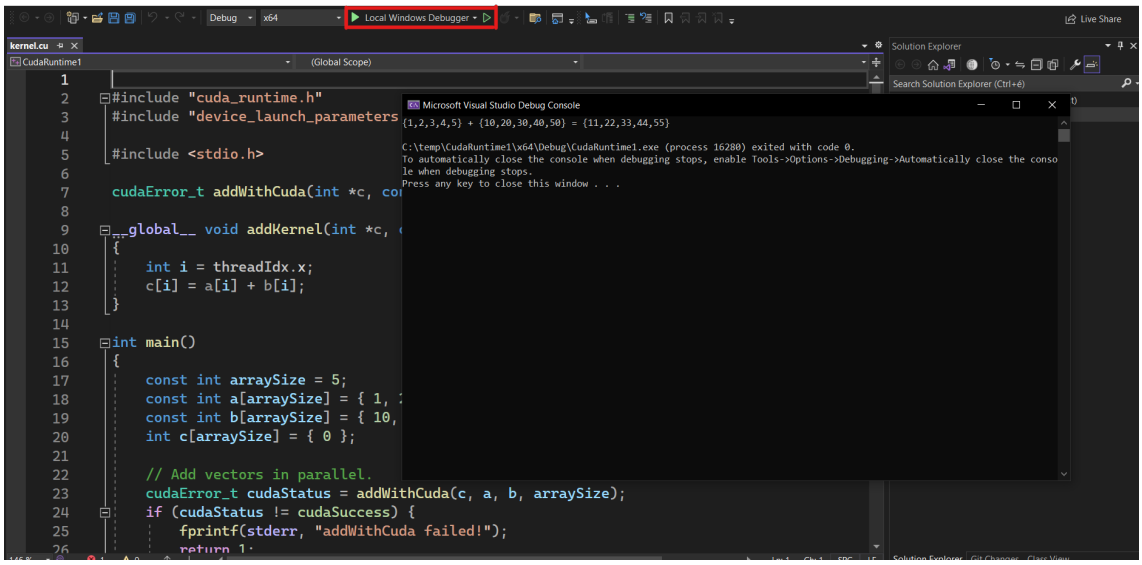
4.2. ábra. 2. lépés: Új CUDA projekt létrehozása

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

A kernel függvény nagyon egyszerű, mindössze 2 sorból áll: [4]

```
__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.



4.3. ábra. 3. lépés: Példakód futtatása

5. fejezet

Implementáció

Hardver ismertetése
- Kódról duma
- Táblázatosan a mérések

6. fejezet

Eredmények

6.1. Mérési eredmények

táblázatos formában

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

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

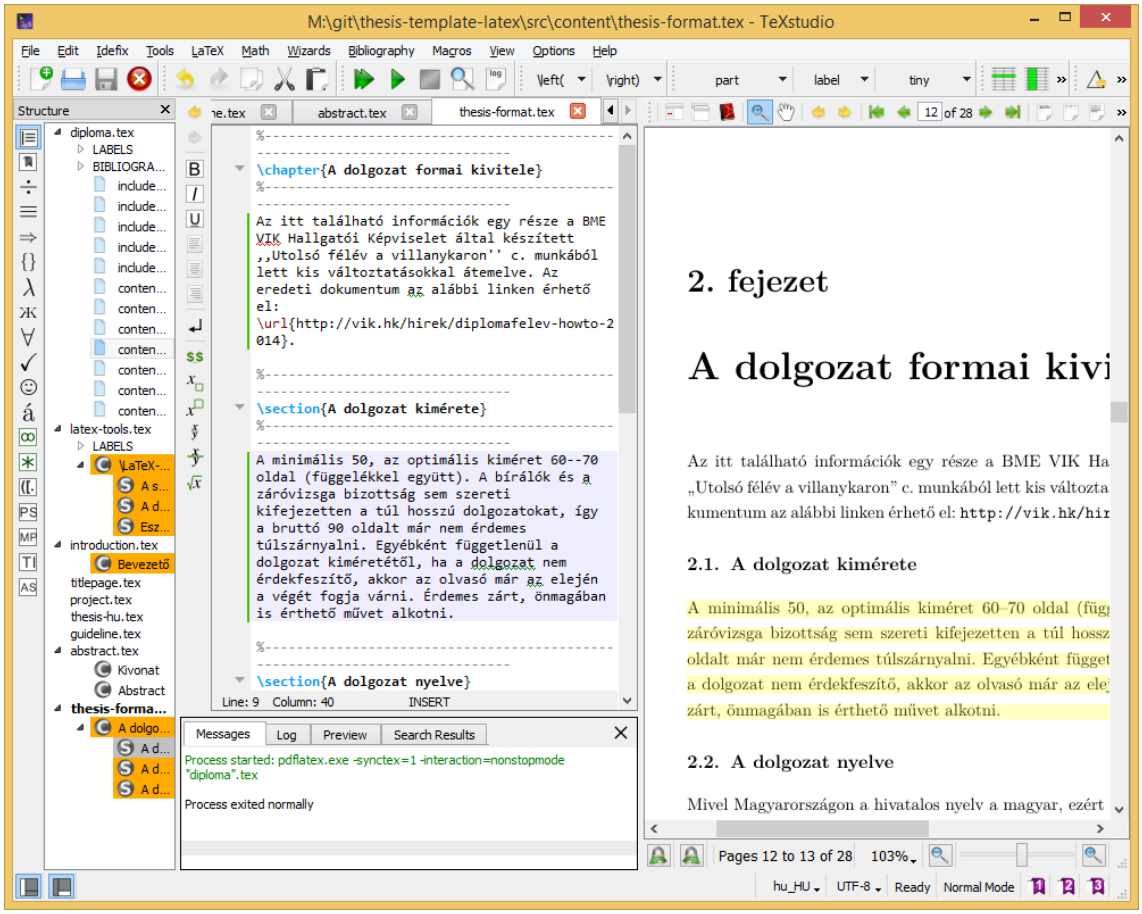
[5] Prof. Wu Feng: To gpu synchronize or not gpu synchronize? https://www.nvidia.com/content/GTC/documents/SC09_Feng.pdf.

[6] 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.

[7] 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

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