Párhuzamos programozás projekt

Dns szekvenciálás lehetséges megoldása párhuzamosított mintaillesztési algoritmusokkal

# Projekthez beadott ismertető

Az általam választott alkalmazás, egy DNS szekvenciálási felhasználásra fejlesztett, karakterlánc mintaillesztő alkalmazás, melynek célja a Knuth-Morris-Pratt, a Boyer-Moore és a Rabin-Karp algoritmusok párhuzamosítása.   
A projekt célja, hogy a DNS szekvencia kettős láncolt lista ábrázolásának minél gyorsabb feldolgozása, illetve adott minta megtalálása a szekvenciákban.  
Egy DNS szekvencia közel 3 milliárd vagyis 3\*10^9 adatot tartalmaz, melyek a heterociklusos bázis párok (A adenin, guanin G, citozin C, timin T) listájával ábrázolható , melyek a következő páronként jelennek meg közel arányos eloszlásban: A-T, G-C.

Az alkalmazás az alábbi algoritmusokat fogja tartalmazni:

* Egy általam fejlesztett GPU algoritmus, mely legenerálja a szekvenciát
* Knuth-Morris-Pratt (KMP) algoritmus CPU-ra megírt verziója.
* Boyer-Moore (BM) algoritmus CPU-ra megírt verziója.
* Rabin-Karp (RK) algoritmus CPU-ra megírt verziója.
* Knuth-Morris-Pratt algoritmus GPU-ra megírt verziója.
* Rabin-Karp algoritmus GPU-ra megírt verziója.

A legtöbb számítási időt a CPU-ra megírt algoritmusok fogják elhasználni tekintve, hogy mindegyik algoritmus -kis heurisztikával is- de lineáris elven működnek, illetve a KMP és BM algoritmusok előkészítési algoritmusa magas lépésszámú lesz A láncolt vektor feldolgozása és a mintaillesztés((KMP,BM)) a vektor összeadáshoz hasonló módszerrel jól párhuzamosítható. A RB algoritmus jelen alkalmazási elv szerint mentesül a Horner elrendezés algoritmus lefuttatása alól, mivel teljes ABC helyett csak 4 karakterünk van, tehát szintén könnyen párhuzamosítható.

# A megvalósított projekt

A projekt kiválasztásakor, nem vettem figyelembe, hogy mekkora időigény szükséges az algoritmusok helyes párhuzamosításához, és a párhuzamosítás implementálásához. Emiatt nem készült el a projekt határidőre, és nem lett volna kész a pót határidőre sem. Alábecsültem a feladat bonyolultságát.

Ebből az okból kifolyólag redukáltam a projektet, hogy az elkészülhessen a póthatáridőre. Az így megvalósított projekt leírása a következő. Elkészült a párhuzamosított DNS szekvencia generátor, mely 3 kernel metódust tartalmaz. (részletezés a későbbiekben) Illetőleg elkészült a 3 választott algoritmus implementálása processzorra.

A projekt ugyan nem készült el, de a generate függvény teljes mértékben kimeríti a kurzuson tanultak alkalmazását, így alkalmas az értékelés meghatározására.

# Implementáció

## Szerkezeti vázlat

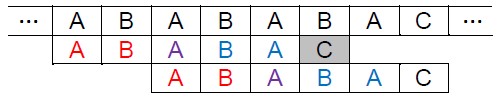
A megvalósított alkalmazás főbb elemei a következők:

* generate.h
  + A DNS szekvencia generálás párhuzamosított algoritmusát tartalmazza
* CPU\_search.h
  + Tartalmazza a 3 mintaillesztési algoritmus szekvenciális változatát
    - KMP
    - BM
    - RK
* GPU\_search.h
  + Tartalmazza a 3 mintaillesztési algoritmusból kettőnek a párhuzamosítására tett próbálkozásaimat.
    - KMP
    - BM ( nem tartalmazza)
    - RK
* DNA\_seq.cu
  + Az alkalmazás main metódusát tartalmazza.

## Az algoritmusok ismertetése

KMP algoritmus

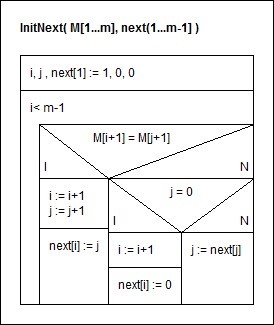
A Knuth-Morris-Pratt eljárásnak a Brute-Force (hasonlítsuk össze, toljunk egyet, stb..) módszerrel szemben az az előnye, hogy egyes esetekben, ha a mintában vannak ismétlődő elemek, akkor egy tolásnál akár több karakternyit is ugorhatunk.



. ábra KMP algoritmus több karakter tolás esetén

Az ugrás megállapítását a következőképp tesszük: Az eddig megvizsgált egyező mintarész elején (prefix) és végén (suffix) olyan kartersorozatot keresünk, melyek megegyeznek. Ha találunk ilyet, akkor a mintát annyival tolhatjuk, hogy az elején lévő része ráilleszkedjen a végen levőre.

Azt, hogy ez egyes esetekben mekkorát tolhatunk nem kell minden elromlás alkalmával vizsgálni. Ha a mintára ¨önmagával lefuttatjuk az algoritmus egy módosított változatát (3. ´ábra), kitölthetünk egy tömböt, mely alapján a tolásokat végezni fogjuk.



. ábra: KMP tolásokat szabályzó tömb kitöltésé

Az algoritmus (ld 2. ábra):

Két indexet *i* ´es *j* futtatunk a szövegen illetve a mintán.

Ha az *i* + 1-edik ´es *j* + 1-edik karakterek megegyeznek, akkor léptetjük mind a kettőt.

Ha nem egyeznek meg, akkor:

Ha a minta első elemét vizsgáltuk, akkor egyet tolunk a mintán, magyarul a minta indexe marad az első betűn, ´es a szövegben lévő indexet növeljük eggyel (*i* = *i* + 1)

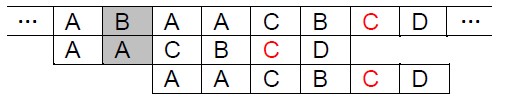
Ha nem a minta első elemét vizsgáltuk, akkor annyit tolunk, amennyit szabad. Ez azt jelenti, hogy csak a mintán lévő indexet helyezzük egy kisebb helyre (*j* = *next*[*j*])

Addig megyünk, míg vagy a minta, vagy a szöveg végére nem ´érünk. Ha a minta végére ´értünk, akkor megtaláltuk a mintát a szövegben, ha a szöveg végére ´értünk, akkor pedig nem.

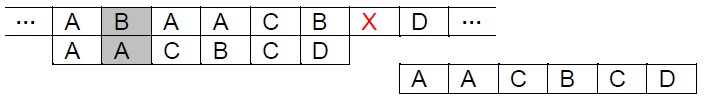
BM algoritmus (Quick Search)

Míg a KMP algoritmus az elromlás helye előtti rész alapján döntött a tolásról, addig a QS a minta utáni karakter alapján. Tehát elromlás esetén:

• Ha a minta utáni karakter benne van a mintában, akkor jobbról az első előfordulására illesztjük



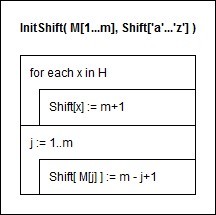
. ábraQS - eltolás ha a minta utáni karakter benne van a mintában • Ha a minta utáni karakter nincs benne a mintában, akkor a mintát ezen karakter után illesztjük.



. ábra: QS - eltolás ha a minta utáni karakter nincs benne a mintában

Az eltolás kiszámítását megint elő lehet segíteni egy tömbbel, most azonban, mivel nem a minta az

´érdekes, ´es nem tudjuk pontosan mely karakterek szerepelnek a szövegben, így a tömbbe az egész abc-t fel kell vennünk (7. ´ábra)



. ábra: QS - Az eltolást elősegítő tömb (Shift[0a0...0z0]) konstruálása

Két indexet *k* ´es *j* futtatunk a szövegen illetve a mintán.

Ha a szöveg *k* + *j*-edik eleme megegyezik a minta *j*-edik karakterével, akkor léptetjük *j*-t (mivel a szövegben *k* + *j*-edik elemet nézzük, ´így elég *j*-t növelni).

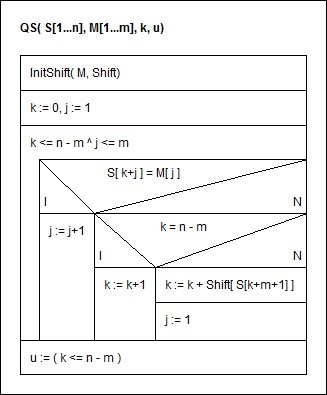
Ha nem egyeznek meg, akkor:

Ha a minta már a szöveg végén van (*k* = *n*−*m*), akkor csak növeljük *k*-t eggyel, ami hamissá teszi a ciklus feltételt.

Ha még nem vagyunk a szöveg végén *k*-t toljuk annyival, amennyivel lehet (ezt az előre beállított *Shift* tömb határozza meg). Es a´ *j*-t visszaállítjuk 1-re.

Addig megyünk, míg vagy a minta végére ´érünk *j*-vel, vagy a mintát tovább toltuk a szöveg végénél.

Előbbi esetben egyezést találtunk, míg az utóbbiban nem.



. ábra QS

RK algoritmus

A Rabin-Karp algoritmus lényege, hogy minden betűhöz az ´abécéből egy számjegyet rendelünk, ´es a keresést számok összehasonlításával végezzük. Világos, hogy ehhez egy ´ábécé méretnek megfelelő számrendszerre lesz szükségünk. A szövegből mindig a minta hosszával egyező részeket szelünk ki, ´es ezeket hasonlítjuk ¨össze.

Példa:

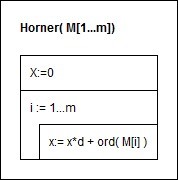
Minta: BBAC → 1102

Szöveg: DACABBAC → 30201102, amiből a következő számokat ´állítjuk elő: 3020, 0201, 2011, 0110, 1102

A fent látható szeletek lesznek az *si*-k.

Az algoritmus működéséhez azonban számos apró ¨ötletet alkalmazunk:

A minta számokká alakítását Horner-módszer segítségével végezzük.



. ábra: RK - Horner-módszer

Az *ord*() függvény az egyes betűknek megfelelő számot adja vissza. A *d* a számrendszer alapszáma.

A szöveg mintával megegyező hosszú´ szeletéinek (*si*) előállítása: *s*0-t a Horner-módszerrel ki tudjuk számolnia. Ezek után *si*+1 a következőképp számolandó: *si*+1 = (*si* − *ord*(*S*[*i*]) · *dm*−1) · *d* + *ord*(*S*[*i* + 1]) Felmerülhet a kérdés, hogy az ilyen magas alapszámú´ számrendszerek nem okoznak-e gondot az ábrázolásnál? A kérdés jogos. Vegyük a következő ´életszerű˝ példát: 4 bájton ´ábrázoljuk a számainkat (232). Az abc legyen 32 elemű˝ (*d* = 32), a minta 8 hosszú´ (*m* = 8). Ekkor a *dm*−1 kiszámítása: 327 = (25)7 = 235 , ami már nem ´ábrázolható 4 bájton. Ennek kiküszöbölésére vezessünk be egy nagy *p* prímet, melyre *d* · *p* még ´ábrázolható. Es a´ műveleteket számoljuk mod *p*. Ekkor természetesen a kongruencia miatt lesz olyan eset, amikor az algoritmus egyezést mutat, mikor valójában nincs. Ez nem okoz gondot, mivel ilyen esetben karakterenkénti egyezést vizsgálva ezt a problémát kezelni tudjuk. (Fordított eset nem fordul elő tehát nem lesz olyan eset, mikor karakterenkénti egyezés van, de numerikus nincs). [Ha *p* kellően nagy, a jelenség nagyon ritkán fordul elő.]

A mod *p* számítás egy másik problémát is felvet. Ugyanis a kivonás alkalmával negatív számokat is kaphatunk.

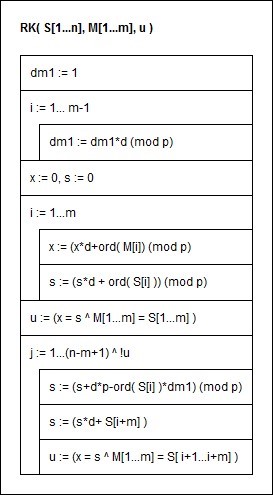
Például: Legyen *p* = 7, ekkor, ha *ord*(*S*[*i*]) = 9, akkor előző számítás után *si* = 2*...*, de ebből *ord*(*S*[*i*]) · *dm*−1 = 9 · 103 = 9000-et vonunk ki negatív számot kapunk. Megoldásként *si*+1-et két lepésben számoljuk:

*s* := (*si* + *d* · *p* − *ord*(*S*[*i*]) · *dm*−1) mod *p*

*si*+1 := (*s* · *d* + *ord*(*S*[*i* + 1])) mod *p*

A fentiek alapján az algoritmus a következő (ld. 8. ´ábra)

* Kiszámoljuk *dm*−1-et (*dm*1)
* Egy iterációban meghatározzuk Horner-módszerrel a minta számait (*x*) ´es *s*0-t
* Ellenőrizzük, hogy egyeznek-e
* Addig számolgatjuk *si* ´érteket míg a minta nem egyezik *si*-vel, vagy a minta a szöveg végére nem ért.

8. ábra: RK

## generate.h

A generate.h fejállomány 3 párhuzamosított metódust tartalmaz, melyek teljes mértékben a saját általam kitalált és lefejlesztett algoritmusokat tartalmazzák. Továbba egy szekvenciális kiirató metódust, a generálást megvalósító generate függvényt, és a szokásos órán használt cuda error ellenőrző metódust.

### generate\_kernel

A metódus paraméterül várja a két még üres szekvenciát, azok méretét, n illetve egy curand generált unsigned int tömböt mely a genomok eloszlását fogja megadni. A genom tömb tartalmazza a 4 karaktert amit egy genom tartalmazhat. Az első szekvenciába kerülnek a random kapott genom tömb elemek, a második szekvenciába kerülnek a random kapott elem ellentétei egy switch-case segítségével. Ezek után a metódus egy random helyen elrejti a mutációt, a paraméteról kapott random tömb alapján ami nem mellesleg az implementáció szerint Poisson eloszlású.

### HIDE(KERNEL)

A hide metódus végzi a pattern elrejtését a szekvenciák valamelyikében, ehhez paraméterül várja két random számot, hogy melyik szekvenciában és melyik karakternél(hanyadik elemnél) kezdje el. innentől a metódus azonos elven működik mint a generate függvény, vagyis switch – case-ek segítségével módosítja egyszerre a két tömböt.

### PRINT\_DNA\_KERNEL

Egy egyszerű párhuzamosított tömb kiirató metódus, melyhez a lépéseket stride-ok segítségével implementáltam.

### Generate metódus

A generate metódus végzi a memória foglalásokat a GPU részére, a hide kernelhez szükséges random számok, illetve a curand poisson tömb generálását. A szükséges memória másolások utén futtatja a 3 fent említett metódus futtatását és szinkronizálását.

## GPU\_search.h

Mint ahogy a „A megvalósított projekt” fejezetben már leírtam, nem sikerült a keresések párhuzamosítása, a header fájl fordul, de a keresés nem talál eredményt a szekvenciában. Az algoritmusok párhuzamosításához két a github-on fellelhető implementációt vettem alapul, de átírásuk nem érte el a kellő sikert. A KMP algoritmushoz, a <https://github.com/Fang-Haoshu/KMP-on-CUDA> oldalon található fájlokat kértem segítségül. Míg az RK algoritmushoz a <https://github.com/mstftrn/rabin-karp-for-CUDA> repót használtam fel.

## CPU\_SEARCH

A CPU\_seearch fejállomány tartalmazza a CPU keresésekhez szükséges algoritmusokat, melyek megfelelnek az algoritmusok előzőekben megadott leírásának. A gyors haladás érdekében, és mert a Tanár úr is javasolta, a CPU algoritmusokat, már kész implementációk alapján készítettem.

A KMP algoritmust  
<https://www.techiedelight.com/implementation-kmp-algorithm-c-cpp-java/>,  
A QS algoritmust  
<https://www.geeksforgeeks.org/boyer-moore-algorithm-good-suffix-heuristic/>  
Az RK algoritmust  
<https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/>  
alapján implementáltam a feladathoz.

# Fordítás és futtatás

## Fordtás

A programot a -lcurand kapcsolóval kell fordítani a generate.h állományban található curand random generátor miatt. A fordítás teljes parancsa: nvcc DNA\_seq.cu -lcurand -o dna

## Futtatás

Innentől a szokásos módon futtatható az állomány, viszont az átláthatóság érdekében érdemes egy txt fájlba futtatni, pontosabban: ./dna > result.txt

## Megjegyzés

A DNA\_GPU.cu nevű fájllal futtathatóak az algoritmusok futtatására tett próbálkozásaim.  
A CPU\_search.h és a GPU\_search.h együtt nem include-olható, ezért került külön fájlba.  
A helyes parancs: nvcc DNA\_GPU.cu -lcurand -o dnagpu.  
A futtatás az előzőekkel megegyező.  
A szekvenciák mérete növelhető, de a gyors futtatás végett, nem ajánlott(A CPU keresések nagy méretű tömbön lassúak.) Továbbá az elrejtést hozzá kell igazítani a tömbhöz:

int j = (i-1)%PATTERN\_SIZE; // if the sequence size is 15000 i+2