FPA ALGORITMUS IMPLEMENTÁLÁSA MASSZÍVAN PÁRHUZAMOS ARCHITEKTÚRÁRA

IMPLEMENTATION OF FPA ALGORITHM ON MASSIVLY PARALLEL ARCHITECTURE

Nagy Szilárd[*], *Dr. Jármai Károly*[**]

ABSTRACT

Evolutionary algorithms are powerful tools for solving non-linear, multidimensional optimization problems. Solving large-scale problems is often time consuming. Evolution of GPUs (Graphics Processing Unit) in recent years allows them to be used for general purpose calculations. In this paper the implementation of the FPA (Flower Pollination Algorithm) algorithm on GPU and the results are presented.

1. BEVEZETÉS

A mesterséges intelligencia heurisztikus ágához tartozó evolúciós algoritmusok kutatása az elmúlt években nagy hangsúlyt kapott. Ez nem meglepő, hiszen jól használhatók nem lineáris, sokváltozós, bonyolult keresési, optimálási feladatok megoldására.

$$\min f(\bar{x}) \qquad \bar{x} = [x_1 \quad x_2 \quad \cdots \quad x_j \quad \cdots \quad x_D] \qquad (1)$$

ahol D a probléma változóinak száma, és minden x_j rendelkezik egyenként egy alsó és felső határral.

Olyan esetekben is képesek eredményesek lenni, amikor a tradicionális gradiens alapú módszerek nehézkesen, vagy egyáltalán nem alkalmazhatók. További nagy előnyük, hogy képesek a célfüggvényt feketedobozként kezelni. Nem kell ismerni a függvény konkrét belső felépítését elég csak a bemeneteket és az azokra adott választ.

A rengeteg típusú evolúciós algoritmus létezik. Mindegyik közös jellemzője, hogy az alapelvét valamilyen a természetben is lejátszódó folyamat ihlette és az alábbi általános lépésekből áll [1]:

1. legyen $p^{(0)}$ a kezdeti populáció,
2. ha a megállási kritérium teljesül, add vissza a $p^{(G)}$ populációt,
3. egyébként bővítsd a $p^{(G)}$ populációt új egyedekkel,
4. szelekcióval állítsd elő a $p^{(G)}$ populációból az új $p^{(G+1)}$ populációt,
5. ismételd a 2. ponttól.

A harmadik pontban a bővítés történhet többek között például mutációval. A negyedik lépés feladata, hogy a populáció méretet adott n méreten tartsa.

Iterációs eljárásokról lévén szó, elképzelhető, hogy egy-egy optimalizálási feladat megoldása költséges, idő igényes lesz. Egyrészt köszönhető a célfüggvény többszöri kiszámításának. Másrészt pedig az algoritmus belsejében lezajló mutáció és szelekció műveletek műveletigényének. Jelen cikk az utóbbi kiküszöbölésére kínál alternatívát, a párhuzamosítás alkalmazásával.

2. EREDETI VIRÁG BEPORZÁSI ALGORITMUS

A virág beporzási algoritmust (flower pollination algorithm; röviden: FPA), mint ahogyan a neve is mutatja a növények beporzási folyamata inspirálta. Fiatal algoritmusról van szó. Először Xin-She Yang javasolta [2] 2012-ben.

A természetben a növények reprodukciós folyamatára a különböző beporzási módszerek a működnek. Az adott növényre jellemző virágpor, pollen átkerül egy másik növényre, rovarok, madarak, denevérek, egyéb állatok vagy a szél segítségével. Van néhány növény, ami ettől eltérő speciális beporzási módszert választott.

Általánosságban megfogalmazható négy domináns szabály, melyek alapján modellezhető a folyamat, és az algoritmus matematikai alapját képezik:

- **globális-beporzás (**kereszt-beporzás) során a pollen átkerül az egyik egyedről egy másik egyedre. A beporzók mozgása modellezhető Lévy eloszlást követő véletlen számmal.
- **helyi-beporzásnál** a pollen ugyanabból a virágból, vagy ugyanazon növény másik virágából származik.
- adott fajból származó pollen csak az ugyanabba fajba tartozó növényt tudja beporozni. Az FPA vonatkozásában ez azt jelenti, hogy a beporzás csak akkor történik meg, ha utána a meglévőnél jobb eredmény jön létre.
- helyi- és globális-beporzás bekövetkezésének valószínűségét egy $P \in [0; 1]$ normál eloszlású véletlen valós szám fejezi ki.

* *PhD hallgató, Miskolci Egyetem, vegynsz@uni-miskolc.hu*
** *egyetemi tanár, Miskolci Egyetem, jarmai@uni-miskolc.hu*

$p^{(0)}$ *populáció inicializálása véletlenszerűen*
$\bar{g}^*$ *legjobb megoldás kiválasztása* $p^{(0)}$*-ból*
$P \in [0;1]$ *valószínűség meghatározása*
ciklus míg $G <$ *maximum genráció*
  ciklus $i = 1:n$ *(összes egyedre)*
    ha $rand < P$
      L lévy eloszlású véletlen szám generálása (3)
      globális beporzás (2) alapján
    egyébként
      ϵ *véletlen szám generálása*
      helyi beporzás (4) alapján
    ha vége
    új függvényérték meghatározás
    ha a kapott eredmény jobb, új egyed megtartása
  ciklus vége
ciklus vége

1. ábra Szekvenciális FPA algoritmus

A kereszt-beporzás matematikai formája:

$$\bar{x}_i^{(G+1)} = \bar{x}_i^{(G)} + L\left(\bar{g}^* - \bar{x}_i^{(G)}\right) \tag{2}$$

ahol $\bar{g}^*$ a G generációig megtalált globális minimum, L pedig a Lévy-szám, ami közelíthető az alábbi formulával [2][3]:

$$L \approx \frac{\lambda \Gamma(\lambda) \sin\left(\frac{\pi\lambda}{2}\right)}{\pi} \frac{1}{s^{1+\lambda}} \tag{3}$$

ahol λ egy konstans (ajánlott értéke: $\lambda = 1{,}5$), $\Gamma(\lambda)$ gamma eloszlási függvény, s pedig $s > 0$ véletlen lépés.

A helyi-beporzást pedig a differenciál evolúcióból [4] jól ismert mutációs formulával lehet leírni:

$$\bar{x}_i^{(G+1)} = \bar{x}_i^{(G)} + \epsilon\left(\bar{x}_{r_1}^{(G)} - \bar{x}_{r_2}^{(G)}\right) \quad r_1 \neq r_2 \neq i \tag{4}$$

ahol $\epsilon \in [0;1] \cup \mathbb{R}$ normál eloszlású véletlen szám, r_1 és r_2 véletlen egész számok.

Az előzőekben ismertetett szekvenciális FPA algoritmus pszeudokódját az 1. ábra szemlélteti.

3. FPA ALGORITMUS IMPLEMENTÁCIÓJA CUDA ALKALMAZÁSÁVAL.

3.1. GPU programozás és C-CUDA koncepciója

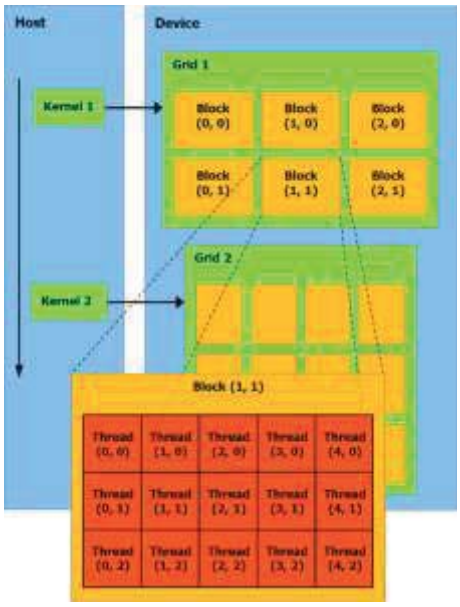
Az elmúlt években a grafikus kártyák rohamos fejlődése új igényt fogalmazott meg velük szemben, az általános célú felhasználást: GPGPU. A kérdés az, hogy a GPU-k mért tesznek lehetővé gyorsabb számításokat, mint a CPU. A válasz az eltérő architektúrában rejlik. Míg a CPU általános célú műveletekre (be- kimeneti eszközök elérése, adatfeldolgozás stb.) van optimalizálva, addig a GPU egyetlen meghatározott feladatra. Ezért nem minden feladat implementálható hatékonyan az utóbbi eszközön.

Habár történtek erőfeszítések API-k kifejlesztésére még mindig komoly kihívásokat jelentő feladat maradt GPU-k programozása. NVida kifejlesztette a CUDA környezetet [5]. Lehetővé teszi a C/C++ programozási nyelvek minimális kiegészítésével, a kódok direktben GPU-ra történő fordítását.

Két futtatásra alkalmas eszköz került nevesítésre: a gazdagép (host), és az eszköz (device, GPU). A hívható függvények pedig három csoportba sorolhatók:

- **__global__** kulcsszóval azonosítható függvények. Csak a gazda eszköz tudja meghívni és az eszközön futnak. Visszatérési értékük kizárólag *void* lehet.
- **__device__** kulcsszóval azonosítható függvények. Eszköz tudja meghívni őket és az eszközön futnak. Bármilyen típusú lehet a visszatérési értékük.
- normál C/C++ függvények. Gazda eszköz tudja meghívni és a gazda eszközön futnak. Bármilyen típusú lehet a visszatérési értékük.

A függvény típusokon kívül, még a CUDA a saját struktúrájában (2. ábra) bevezeti a rács, a blokk, és a szállak fogalmát.



2. ábra CUDA architektúra Forrás: [6]

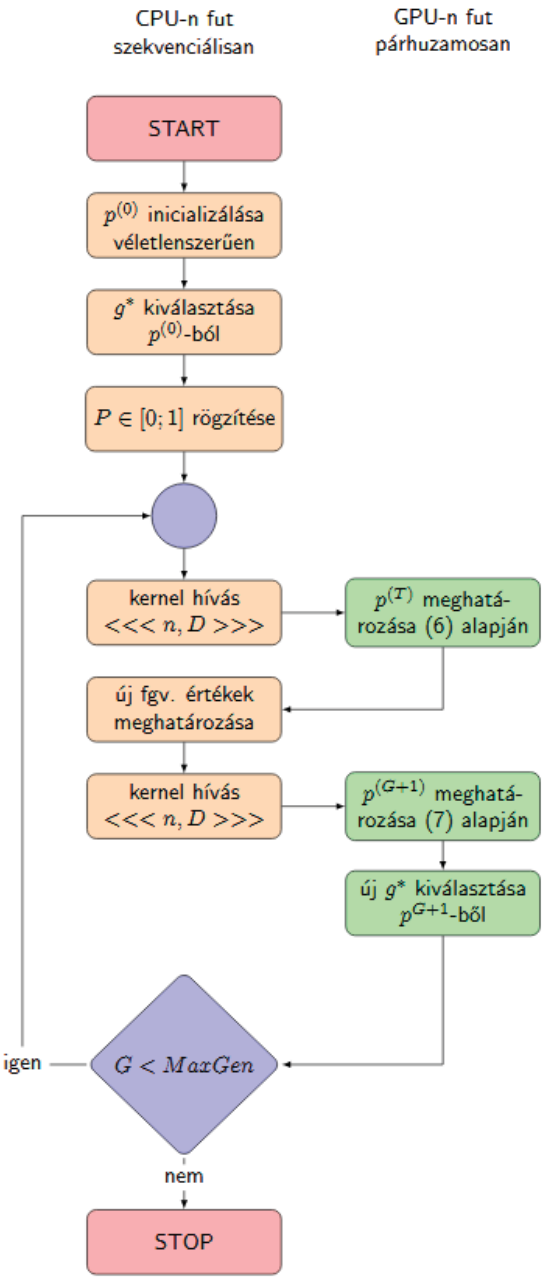
Minden egyes kernel hívás egy rácsot eredményez, ami több blokkból áll. Egy rácson belül képzeletben a blokkok egy kétdimenziós elrendezéssel jellemezhetők. A blokkok tovább bonthatók szálakra, amik elrendezése képzeletben háromdimenziós. Egy blokkban minden szál ugyanazt a feleadatot hajtja végre, csak más adaton. Tipikus példája SIMD (single instruction multiple data) típusú párhuzamosításnak [7]. Ugyanazon blokkba tartozó szálak szinkronizálhatók és képesek egymással kommunikálni egy gyors elérésű néhány Kbyte nagyságú osztott memória területen keresztül. Egy

eszközön párhuzamosan, aszinkron módon futhat több különböző feladat is. Ekkor több egymástól független rács jön létre.

A CUDA a C/C++ nyelv kiegészítésén kívül még tartalmaz az eszköz egyéb funkcióit elérő függvény könyvtárakat is.

3.2. FPA algoritmus párhuzamos implementációja

A populáció egyedei legyenek egy mátrixban tárolva. Az így kapott mátrix minden egyes sora feleljen meg egy-egy egyednek.



3. ábra Párhuzamos FPA algoritmus

$$p^{(G)} = \begin{bmatrix} \bar{x}_1^{(G)} \\ \bar{x}_2^{(G)} \\ \vdots \\ \bar{x}_i^{(G)} \\ \vdots \\ \bar{x}_n^{(G)} \end{bmatrix} = \begin{bmatrix} x_{1,1}^{(G)} & x_{1,2}^{(G)} & \cdots & x_{1,D}^{(G)} \\ x_{2,1}^{(G)} & x_{2,2}^{(G)} & \cdots & x_{2,D}^{(G)} \\ \vdots & \vdots & \ddots & \vdots \\ x_{i,1}^{(G)} & x_{i,2}^{(G)} & \cdots & x_{i,D}^{(G)} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1}^{(G)} & x_{n,2}^{(G)} & \cdots & x_{n,D}^{(G)} \end{bmatrix} \tag{5}$$

A beporzással kapott új egyedek legyenek tárolva egy ideiglenes $p^{(T)}$ mátrixban. Könnyen belátható, hogy ekkor a mátrix minden eleme egymástól függetlenül számolható. Hatékonyan párhuzamosítható. A szekvenciálisan $O(n \cdot D)$ művelet igényű beporzási lépés elméletileg $O(1)$ művelet igényű lesz.

$$p_{i,j}^{(T)} = \begin{cases} x_{i,j}^{(G)} + L\left(g^* - x_{i,j}^{(G)}\right) & \text{ha } rand < P \\ x_{i,j}^{(G)} + \epsilon\left(x_{a,j}^{(G)} - x_{b,j}^{(G)}\right) & \text{egyébként} \end{cases} \tag{6}$$

A szelekció is hasonló módon párhuzamosítható. Elméletileg $O(n \cdot D)$ művlet igény itt is $O(1)$-re csökken.

$$p_{i,j}^{(G+1)} = \begin{cases} p_{i,j}^{(T)} & \text{ha } f\left(p_i^{(T)}\right) < f\left(p_i^{(G)}\right) \\ p_{i,j}^{(G)} & \text{egyébként} \end{cases} \tag{7}$$

A párhuzamos FPA-t a *3. ábra* részletezi. A kódok egyrésze szekvenciálisan fut a CPU-n, a számításigényes műveletek pedig a GPU-n. Két művelet között a CPU-nak van egy szinkronizációs feladata is, vagyis megvárja míg egy lépés teljesen befejeződik és csak utána hívja meg a következő lépést. Megtartva a fekete doboz elvet a függvény kiértékelés megvalósítása a felhasználóra van bízva. Jelen esetben a feldolgozása maradt szekvenciális.

4. SZIMULÁCIÓ

Szimulációhoz a

$$f(x) = \sum_{j=1}^{D} \left(x_j\right)^2 \tag{8}$$

egyszerű, könnyen optimalizálható, de annál kevésbé hatékonyan párhuzamosítható függvényt választottuk.

Az alkalmazott számítógép főbb jellemzői:

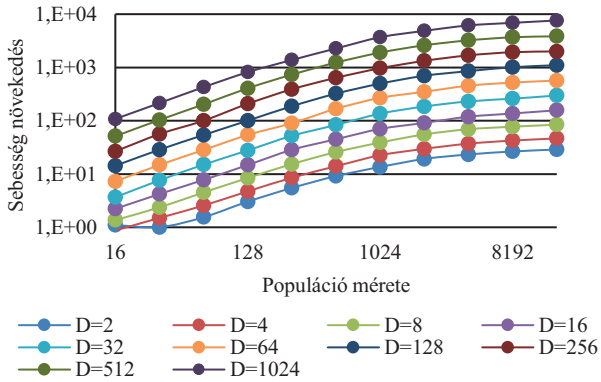
- processzor: Intel i5-3470 3,2GHz
- videokártya: Nvidia Geforce 650
- memória: 8 Gbyte
- operációs rendszer: Linux Mint 18.3

Egy iterációs lépéshez szükséges idő két részre bontható

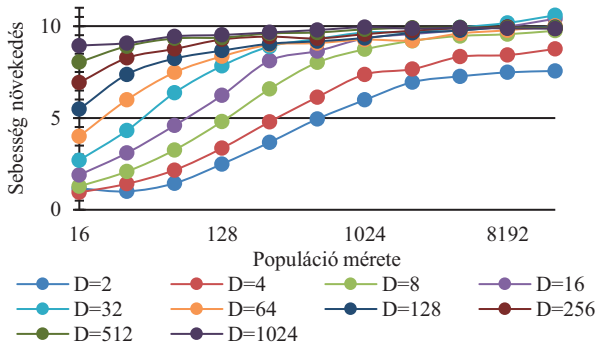
$$t_{it} = t_{alg} + t_{fgv} \quad (9)$$

az új függvényérték kiszámításához szükséges t_{fgv} időre és maga az algoritmushoz szükséges számítások t_{alg} idejére.

Szimuláció során a t_{it} és t_{alg} időknél elérhető sebesség növekedés lett vizsgálva. Mind a teljesen szekvenciális FPA algoritmusnál, mind a párhuzamosítottnál az új függvényérték kiszámítását a CPU végzi. Kapott eredményeket a 4. ábra és 5. ábra szemlélteti.



4. ábra FPA algoritmus (t_{alg}) sebesség növekedése különböző változó szám és populáció esetén



5. ábra Egy iterációs lépés (t_{it}) sebesség növekedése különböző változó szám és populáció esetén

5. ÖSSZEFOGLALÁS

Komoly sebesség növekedés érhető el párhuzamosítással. Csak az FPA algoritmus mutációs és szelekciós lépését vizsgálva (4. ábra) megközelítőleg 7000-szeres sebesség növekedés is elérhető. Minél több változóból áll az optimálandó feladat és minél nagyobb a populáció mérete annál szembetűnőbb a sebesség növekedés. Míg a használt hardware korlátait el nem érjük addig a várható gyorsulás közelítőleg exponenciálisan nő a probléma nagyságával.

A teljes iterációs lépésre fókuszálva, ahol a függvény kiértékelés maradt szekvenciális, nem ennyire szembetűnőek az eredmények. A sebesség növekedés egy felső korláthoz tart, és a számítási idő nagy százalékát már a célfüggvény meghatározása teszi ki. Jelen esetben ez körülbelül 10-szeres. Messzemenő következtetés nem vonható le, mert a sebesség növekedés az optimálási probléma bonyolultságától is függ.

Az eredmények további kutatásra adnak okot, ahol már a teljes iterációs lépés párhuzamosításra kerül, így az optimálást még gyorsabbá és hatékonyabbá téve.

6. KÖSZÖNETNYILVÁNÍTÁS

A cikkben ismertetett kutató munka az EFOP-3.6.1-16-2016-00011 jelű „Fiatalodó és Megújuló Egyetem – Innovatív Tudásváros – a Miskolci Egyetem intelligens szakosodást szolgáló intézményi fejlesztése” projekt részeként – a Széchenyi 2020 keretében – az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.

7. IRODALOM

[1] SIMON, D.: *Evolutionary Optimization Algorithms*, John Wiley & Sons, New Jersey, 2013.

[2] YANG X. S.: Flower Pollination Algorithm for Global Optimization, *Unconventional Computation and Natural Computation*, Vol. 7445, (2012), pp 240-249, doi.org/10.1007/978-3-642-32894-7_27

[3] YANG X. S.: *Nature-Inspired Metaheuristic Algorithms Second Edition*, Luniver Press, Frome, 2010.

[4] STORN R., PRICE K.: Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continous Spaces, *Journa of Global Optimization*, Vol. 11, (1997), pp 341-359, doi.org/10.1023/A:1008202821328

[5] NVidia CUDA C programing guide v10.0.130 https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2018.

[6] FÁBIÓ F, RENATO A. K.: A co-evolutionary differential evolution algorithm for solving min–max optimization problems implemented on GPU using C-CUDA, *Expert Systems with Applications*, Vol. 39, (2012), pp 10324-10333

[7] JASON S, EDWARD K.: *CUDA by Example*, Addison-Wesley, Boston, 2010.