

Паралелизација PSO алгоритма коришћењем *CUDA* архитектуре

Јован Руменић 69/2017
Стефан Исаиловић 29/2016

25. септембар 2021.

Садржај

1	Увод	3
1.1	Дефиниција паралелизације	3
1.2	Оптимизација ројем честица (ПСО)	3
1.3	<i>CUDA</i> архитектура	5
1.3.1	Програмирање у <i>CUDA</i> -и	5
1.4	<i>CUDA</i> и <i>PSO</i>	7
1.5	Опис проблема	7
2	Паралелизација <i>PSO</i> алгоритма помоћу <i>CUDA</i>	7
2.1	<i>SyncPSO</i>	8
2.2	<i>RingPSO</i>	9
3	Резултати поређења	10
3.1	Поређење по итерацијама	10
3.2	Поређење по броју честица	12
4	Закључак	13
5	Литература	14

1 Увод

Коришћење паралелне обраде у рачунарству је присутно скоро са самим настанком истог, а његова популарност и данас наставља да расте. Један од разлога је боља искоришћеност процесора са више језгара, као и графичких картица. Овај рад приказује паралелизацију ПСО алгорита коришћењем *CUDA* архитектуре.

1.1 Дефиниција паралелизације

Паралелна обрада (паралелизација) је облик рачунања у којем се многи прорачуни спроводе истовремено и независно. Идеја је да се већи проблеми поделе на мање, независне проблеме (израчунавања), а затим решавају паралелно. У основи паралелизације су нити. Нити представљају задатке који се истовремено извршавају у оквиру једног процеса. Инструкције које сачињавају једну нит се извршавају у исто време са инструкцијама које сачињавају друге нити (ако у рачунару постоји више процесора) или је та истовременост само привидна (ако у рачунару постоји само један процесор). Као што више процеса раде у исто време на једном рачунару, тако се нити могу замислити као више послова у оквиру једног процеса. Све нити једног процеса имају исти именски простор, користе заједничку меморију и табеле отворених датотека. На тај начин остварујемо знатно већу ефикасност од распоређивања послова у више системских процеса.

1.2 Оптимизација ројем честица (ПСО)

Оптимизација ројем честица је оптимизациони алгоритам заснован на понашању појединачних јединки унутар одређене групе (на пример, јата птица или роја инсеката). Уколико се, вођено инстинктом, јато птица упутује у одређеном смеру у потрази за храном, очекивање је да ће читаво јато следити управо ону птицу која је пронашла извор хране. Међутим, и свака птица понаособ може бити вођена сопственим инстинктом и тиме на тренутак, у потрази за храном напустити јато. Тада се вероватно може десити да, уколико пронађе бољи извор хране, читаво јато управо крене да следи ту птицу.

ПСО припада групи алгорита који се заснивају на интелигенцији роја (*swarm intelligence*). Алгоритам ради над скупом јединки, који се назива ројем. Елементи овог скупа се називају честицама. Честице се на унапред дефинисан начин крећу по простору претраге. Њихово кретање се усмерава имајући у виду њихову тренутну позицију, њихову до сада најбољу позицију, као и до сада најбољу позицију читавог роја. Под најбољом позицијом читавог роја се подразумева до сада најбоља позиција, узимајући у обзир сва његова решења. Процес се понавља док не буде задовољен критеријум заустављања, а у свакој итерацији се ажурира најбоља вредност решења за сваку честицу, као и за рој у целини.

Оптимизација ројем честица се може приказати следећим псеудокодом:

За сва решења (честице) i из скупа решења (роја) X :

Изабрати координате вектора x_i униформно из интервала (l, u)

$p_i = x_i$

Ако је $f(x_i) < f(g)$ тада $g = x_i$

Означити координате вектора v_i да су једнаки нули или униформно из интервала $(-|u - l|, |u - l|)$

Док није испуњен критеријум заустављања:

За све честице и из роја X :

Изабрати бројеве r_g и r_p униформно из интервала $(0, 1)$

$v_i = c_v v_i + c_p r_p (p_i - x_i) + c_g r_g (g - x_i)$

$x_i = x_i + v_i$

Ако је $f(x_i) < f(x_i)$ тада $p_i = x_i$

Ако је $f(x_i) < f(g)$ тада $g = x_i$

Решење је вектор g , а вредност решења број $f(g)$

Алгоритмом је приказан псеудокод основне методе оптимизације ројем честица. Нека је са X означен рој и нека су свакој честици из скупа X додељени вектори $x_i \in R^n$ и $v_i \in R^n, i \in X$, који представљају њене векторе позиције и брзине. Додатно, n -димензионим вектором p_i означена је тренутна најбоља позиција честице $i \in X$, а n -димензионим вектором g тренутна глобална најбоља позиција. При иницијализацији честице i , вредности координата вектора x_i се бирају униформно из скупа (l, u) , а вредности координата вектора v_i постављају на нуле или бирају униформно из скупа $(-|u - l|, |u - l|)$. Овде је са l и u означено доње и горње ограничење претраживачког простора. При иницијализацији алгоритма се додељују почетне вредности векторима p_i и g .

У свакој итерацији се, за сваку честицу i , врши ажурирање вредности вектора брзине са

$$v_i = c_v v_i + c_p r_p (p_i - x_i) + c_g r_g (g - x_i)$$

а затим и вектора тренутне позиције са $x_i = x_i + v_i$. Параметри r_g и r_p се у свакој итерацији бирају униформно из интервала $(0, 1)$, док су c_v , c_p и c_g унапред дефинисани параметри који се могу евентуално и мењати током алгоритма. У свакој итерацији се ажурира и вредност вектора p_i и g . Коначно, одговарајуће решење је садржано у вектору g . Описани процес се понавља све док није испуњен критеријум заустављања.

Постоји више начина да се *PSO* приступ прилагоди дискретним проблемима. Први је пресликавање дискретног претраживачког простора у континуални, примене алгоритма на описан начин, а затим пресликавање назад у дискретан простор. Други начин представља другачије представљање вектора брзине и позиције, као и модификовање оператора. У основној верзији алгоритма вектор позиције је n -торка реалних бројева, а коришћени оператори су стандардни аритметички. Међутим, координате вектора позиције могу бити и бинарни бројеви, а оператори могу бити дефинисани као нпр. оператори над скуповима, итд.

1.3 *CUDA* архитектура

CUDA представља платформу која омогућава паралелно израчунавање на графичким картицама. Развијена је од стране *NVIDIA*-е 2006. године, као прво комерцијално решење које омогућава да се израчунавања опште намене извршавају на графичкој картици. *CUDA* заједно са *NVIDIA* графичким картицама су постали опште прихваћени у многим областима које захтевају прецизна израчунавања децималних бројева. Само нека од њих су:

- Финансије
- Моделовање климе, времена и океана
- Истраживање података
- Машинско учење и *DeepLearning*
- Архитектура, инжењерство и грађевина
- Мултимедија
- Научна истраживања

Званични *CUDA Toolkit* се може преузети са *NVIDIA* веб стране. Сам *Toolkit* поседује библиотеке за машинско учење (*cuDNN*, *TensorRT*), математичке и библиотеке за линеарну алгебру (*cuBLAS*, *cuRand*, *CUDA Math Library*), као и библиотеке за процесирање сигнала, слика и видеа (*cuFFT*, *NVIDIA Performance Primitives*).

1.3.1 Програмирање у *CUDA*-и

У пројекту је коришћен *CUDA* у комбинацији са *C/C++* програмским језиком. Основне концепте оваквог програмирања представљају:

1. *Host* - *CPU* и његова меморија (*host memory*).
2. *Device* - *GPU* и његова меморија (*device memory*).

Ток извршавања програма који користи *CUDA*-у може се описати на следећи начин:

1. Копирају се улазни подаци из *CPU* меморије у *GPU* меморију.
2. Учитава се *GPU* програм и извршава, кеширају се подаци на чипу ради перформанси.
3. Копирају се резултати из *GPU* меморије у *CPU* меморију.

Неке од битних функција *CUDA API* су:

- *cudaMalloc(void** devPtr, size_t size)* - Алоцира меморију величине *size* на *GPU* и преусмерава показивач *devPtr* на ту меморију.

- `cudaFree(void* devPtr)` - Ослобађа меморију на *GPU* на коју показује показивач `devPtr`, а који је претходно алоциран помоћу `cudaMalloc`.
- `cudaMallocPitch(void** devPtr, size_t* pitch, size_t width, size_t height)` - Алоцира блок меморије величине *width* \times *height* на *GPU* и преусмерава показивач `devPtr` на ту меморију.
- `cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind kind)` - Копира меморију показивача `src` у меморију показивача `dst` величине *count*. Ако се као аргумент *kind* наведе `cudaMemcpyDeviceToHost` тада се меморија копира са *GPU* на *CPU*, а ако се наведе `cudaMemcpyHostToDevice` онда се меморија копира са *CPU* на *GPU*.
- `__syncthreads()` - Користи се за синхронизацију нити на *GPU*

Да би нека функција могла да се извршава на *GPU*, потпис функције мора да се означи са `__device__`. Ако желимо да се нека од функција извршава и на *CPU* и на *GPU* можемо је означити са `__device__ __host__` макроима. Извршавање *GPU* дела програма започиње позивањем кернел функције, која се означава са `__global__` и нема повратну вредност. Позив кернел функције се врши из хост дела програма коришћењем:

```
kernelFunctionName <<<numberOfBlocks, numberOfThreadsPerBlock>>> (args)
```

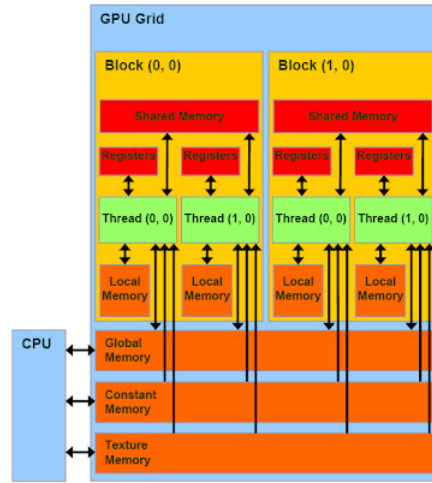
Може се дефинисати више кернел функција, које могу да се позивају из хост дела програма са различитим бројем блокова и нити, као и да се иста функција позива више пута.

Нити које су покренуте од стране кернела су распоређене у наведени број блокова. Сваки блок поседује своју дељену (*shared*) меморију, коју могу да алоцирају и користе само нити из истог блока (слика 1). Дељена меморија се означава са макроом `__shared__` испред имена променљиве и обично се прави на почетку кернел функције. Поред тога што нити поседују ову меморију, оне такође и поседују свој јединствени "*id*", којем се може приступити са `threadIdx.x`. Овај ид представља редни број нити у њеном блоку, који јој је додељен при креирању, и налази се у њеном регистру.

Поред овог ид-а, свака нит поседује и `blockIdx.x`, који означава у којем се блоку налази та нит, као и `blockDim.x` који представља величину блока. Глобални *id* нити се може наћи простим рачуном користећи редни број нити у блоку, редни број блока и величину блока:

```
globalID = blockIdx.x * blockDim.x + threadIdx.x
```

Ово може бити изразито корисно када обраду неког низа желимо да одрадимо тако да свака нит обрађује један елемент низа. У том случају елемент који нека нит обрађује је заправо елемент на позицији `globalID` у том низу.



Slika 1: CUDA niti

1.4 CUDA и PSO

Ако се мало боље анализира *PSO* алгоритам може се приметити да је сам алгоритам скоро па потпуно могуће паралелизовати. Једина зависност која постоји између процеса су информације које морају да се деле између честица. То може бити Глобална најбоља позиција или низ најбољих позиција свих честица (у зависности од имплементације).

1.5 Опис проблема

Сада, када смо објаснили све појмове везане за паралелизацију *PSO* алгоритма, дефинисаћемо детаљније проблем којим ће се овај рад бавити.

Датe су функције из $f \in R^n$, $n \in \{1, 2, 3\}$.

Потребно је одредити минимум датe функције f .

У наставку рада ће бити детаљно описана два приступа паралелизацији *PSO* алгоритма, као и њихове предности и мане. Такође, биће урађено поређење та два приступа међусобно, као и поређење са стандардним *PSO* алгоритмом имплементираним на *CPU*.

2 Паралелизација *PSO* алгоритма помоћу *CUDA*

Не постоји јединствени начин да се *PSO* алгоритам паралелизује. У овом раду ће бити описана два приступа паралелизацији овог алгоритма коришћењем *CUDA*-е:

- Синхронизовани *PSO* (*SyncPSO*)
- *RingPSO*

2.1 *SyncPSO*

Имплементација овог решења се заснива на чинењуи да свака честица може да ради своја рачунања и праћење тренутних најбољих позиција, независно од других честица. Самим тим, свакој честици можемо доделити једну нит која ће се извршавати на *GPU*. Остаје само да се реши проблем чувања глобалних вредности и очување интегритета критичне секције кроз итерације. То може да се реши тако што ће се глобалне вредности као што су најбоље решење/позиција, које се користе од стране свих честица приликом рачунања, чувати у *shared*(дељеној) меморији. Да би се избегли проблеми критичне секције приликом ажурирања глобалних променљивих, за то може да се задужи само једна честица-нит (обично то може да буде баш прва честица), док остале чекају да ажурирање буде готово па тек онда да наставе своје израчунавање. Пример алгоритма је дат псеудо кодом 1.

Algorithm 1 Sync PSO

```
1: if threadIdx.x == 0 then
2:   <initialize global memory>
3: end if
4: __syncthreads()
5:
6: <inicijalize particle data>
7: __syncthreads()
8:
9: for (i=0; i < iterations; ++i) do
10:  <Update particles>
11:  <Evaluate and update fitness for particle>
12:  __syncthreads()
13:
14:  if threadIdx.x == 0 then
15:    <update global values>
16:  end if
17:  __syncthreads()
18: end for
19: if threadIdx.x == 0 then
20:  <store global data in global (CPU) memory>
21: end if
```

Listing 1: SyncPSO kernel

2.2 RingPSO

За разлику од *SyncPSO* алгоритма који се у целости извршава на једном *kernelu*, *RingPSO* дели целокупни алгоритам у више *kernel* позива, који представљају појединачне акције. Идеја је да и даље свака нит представља једну честицу, а да се позиви за ажурирање глобалних вредности обављају позивањем само једне нити [2](#).

Algorithm 2 Ring PSO

InitializeData()	▷ kernel poziv sa jednom niti
2: UpdateParticleBestValues()	▷ kernel poziv sa n niti
UpdateBestGlobalValues()	▷ kernel poziv sa jednom niti
4: for (i=0; i < iterations; ++i) do	
UpdateParticleData()	▷ kernel poziv sa n niti
6: UpdateParticleBestValues()	▷ kernel poziv sa n niti
UpdateBestGlobalValues()	▷ kernel poziv sa jednom niti
8: end for	

Listing 2: RingPSO host pseudokod

У овој конкретној имплементацији алгоритам је раздвојен на три различита кернел позива:

- *IntializeData()* - функција која иницијализује почетне позиције и брзине честице, позива се са n нити
- *UpdateParticleData()* - функција која врши ажурирање брзине и позиције честице, позива се са n нити
- *UpdateParticleBestValues()*- функција која врши евалуацију и ажурирање најбоље позиције/фитнеса честице, позива се са n нити
- *UpdateBestGlobalValues()* - функција која врши ажурирање глобално најбољих вредности позиције/фитнеса, позива се над једном нити

3 Резултати поређења

Да би смо упоредили различите имплементације *PSO* алгорита, и на тај начин уочили предности и мане међу њима, урадили смо неколико тестова брзине за различити број димензија проблема, итерација и броја честица. Сва тестирања вршићемо над три имплементације *PSO* алгорита:

- *Standard PSO* - ПСО имплементиран у *Python* програмском језику, без коришћења нити и *CUDA*-е
- *SyncPSO* - ПСО алгоритам имплементиран помоћу само једног једног кернела
- *RingPSO* - ПСО алгоритам имплементиран помоћу више од једног кернела

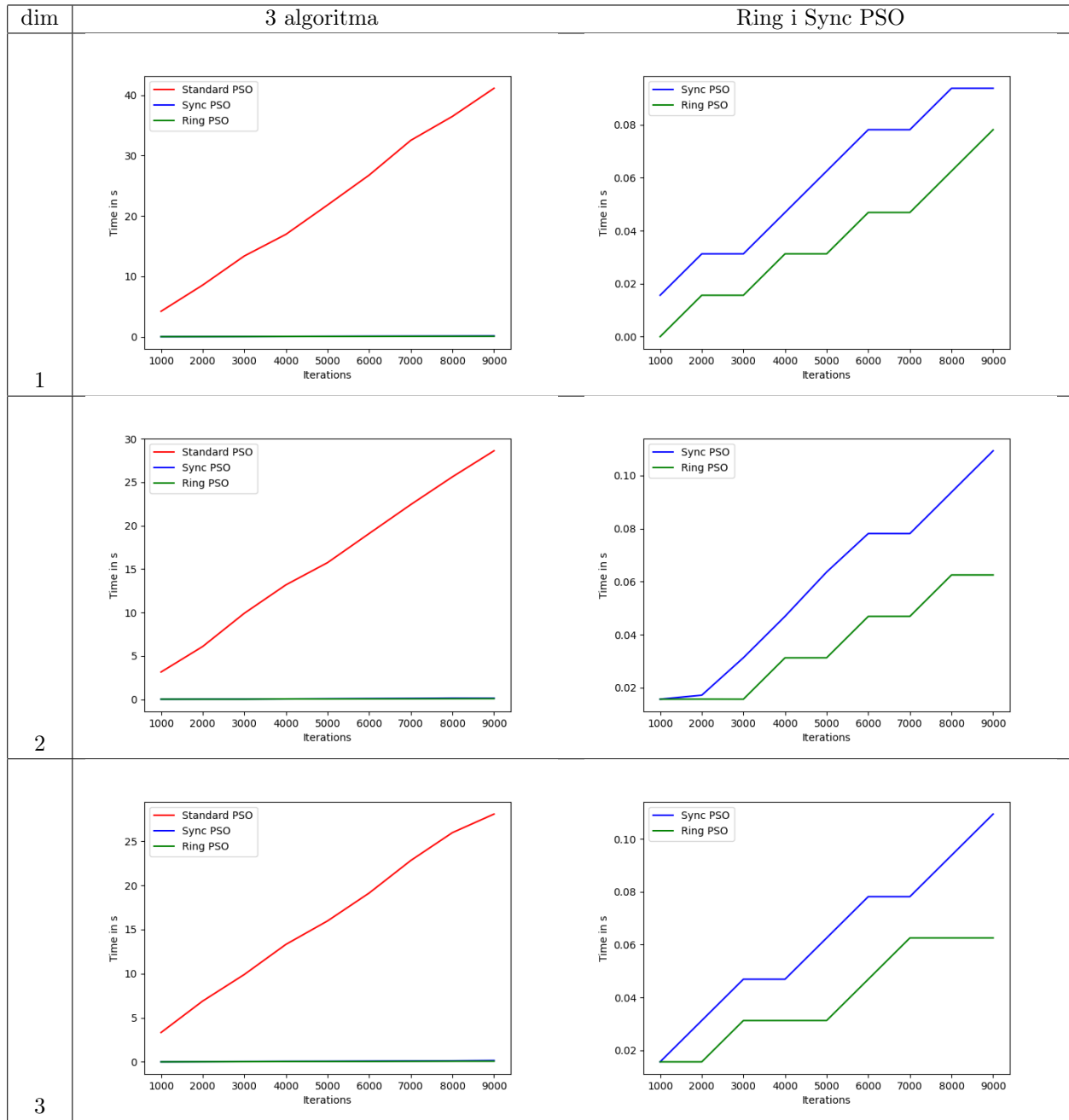
Функције над којима су вршена тестирања алгорита су:

- $f(x) = x^2 - 5x + 6$
- $f(x, y) = x^2 + y^2 + 2$
- $f(x, y, z) = x^2 + y^2 + z^2 + 3$

3.1 Поређење по итерацијама

Да би смо извршили коректно поређење односа итерације - времена извршавања, фиксирани смо број честица на 100. Алгоритми се покрећу у петљи, а број итерација се креће од 1000 до 10000, повећавајући број итерација у сваком проласку петље за 1000.

Са графика се може приметити да стандардни ПСО ради незанемарљиво спорије од *Sync* и *Ring* ПСО. Такође, када поредимо ова два алгорита који користе *CUDA*-а архитектуру, можемо приметити да *RingPSO* ради за нијансу брже од *SyncPSO*.



3.2 Поређење по броју честица

Сада фиксирамо број итерација на 10000. Алгоритми се покрећу у петљи, а број честица се креће од 1 до 800 са кораком од 100. Након првог тестирања ова три алгоритма (слика 2) са димензијом проблема величине један, видимо да је стандардни ПСО доста спорији од остала два алгоритма. Његова брзина само још више опада са порастом димензија, па те графике нећемо укључити у ово поређење, јер не представљају значајан аспект овог тестирања. Поређење *Sync* и *Ring* алгоритма можете погледати у следећој табели 3.2.

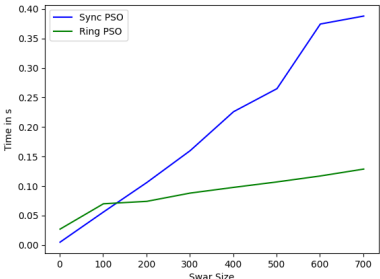
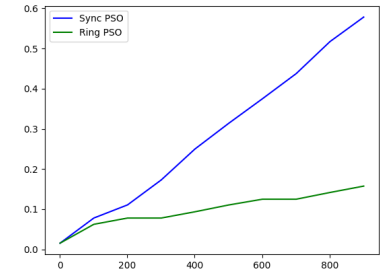
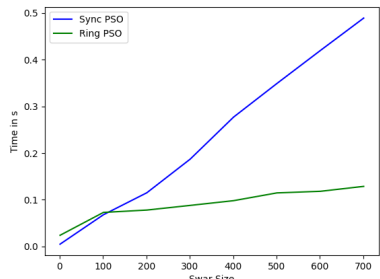
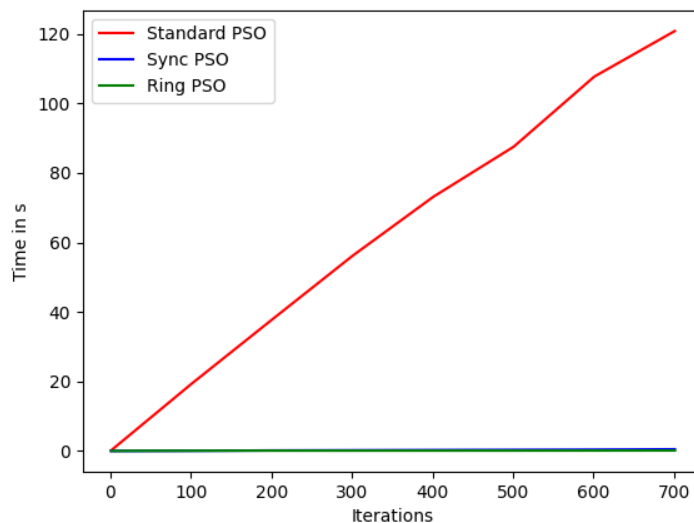
dim	Ring i Sync PSO																											
1	 <table><caption>Approximate data for dimension 1</caption><thead><tr><th>Swar Size</th><th>Sync PSO (s)</th><th>Ring PSO (s)</th></tr></thead><tbody><tr><td>0</td><td>0.00</td><td>0.02</td></tr><tr><td>100</td><td>0.05</td><td>0.05</td></tr><tr><td>200</td><td>0.10</td><td>0.06</td></tr><tr><td>300</td><td>0.15</td><td>0.08</td></tr><tr><td>400</td><td>0.22</td><td>0.10</td></tr><tr><td>500</td><td>0.28</td><td>0.11</td></tr><tr><td>600</td><td>0.38</td><td>0.12</td></tr><tr><td>700</td><td>0.40</td><td>0.13</td></tr></tbody></table>	Swar Size	Sync PSO (s)	Ring PSO (s)	0	0.00	0.02	100	0.05	0.05	200	0.10	0.06	300	0.15	0.08	400	0.22	0.10	500	0.28	0.11	600	0.38	0.12	700	0.40	0.13
Swar Size	Sync PSO (s)	Ring PSO (s)																										
0	0.00	0.02																										
100	0.05	0.05																										
200	0.10	0.06																										
300	0.15	0.08																										
400	0.22	0.10																										
500	0.28	0.11																										
600	0.38	0.12																										
700	0.40	0.13																										
2	 <table><caption>Approximate data for dimension 2</caption><thead><tr><th>Swar Size</th><th>Sync PSO (s)</th><th>Ring PSO (s)</th></tr></thead><tbody><tr><td>0</td><td>0.00</td><td>0.00</td></tr><tr><td>100</td><td>0.08</td><td>0.05</td></tr><tr><td>200</td><td>0.12</td><td>0.07</td></tr><tr><td>300</td><td>0.20</td><td>0.08</td></tr><tr><td>400</td><td>0.28</td><td>0.10</td></tr><tr><td>500</td><td>0.38</td><td>0.12</td></tr><tr><td>600</td><td>0.45</td><td>0.13</td></tr><tr><td>700</td><td>0.58</td><td>0.15</td></tr></tbody></table>	Swar Size	Sync PSO (s)	Ring PSO (s)	0	0.00	0.00	100	0.08	0.05	200	0.12	0.07	300	0.20	0.08	400	0.28	0.10	500	0.38	0.12	600	0.45	0.13	700	0.58	0.15
Swar Size	Sync PSO (s)	Ring PSO (s)																										
0	0.00	0.00																										
100	0.08	0.05																										
200	0.12	0.07																										
300	0.20	0.08																										
400	0.28	0.10																										
500	0.38	0.12																										
600	0.45	0.13																										
700	0.58	0.15																										
3	 <table><caption>Approximate data for dimension 3</caption><thead><tr><th>Swar Size</th><th>Sync PSO (s)</th><th>Ring PSO (s)</th></tr></thead><tbody><tr><td>0</td><td>0.00</td><td>0.02</td></tr><tr><td>100</td><td>0.08</td><td>0.05</td></tr><tr><td>200</td><td>0.12</td><td>0.07</td></tr><tr><td>300</td><td>0.20</td><td>0.08</td></tr><tr><td>400</td><td>0.28</td><td>0.10</td></tr><tr><td>500</td><td>0.38</td><td>0.12</td></tr><tr><td>600</td><td>0.45</td><td>0.13</td></tr><tr><td>700</td><td>0.50</td><td>0.14</td></tr></tbody></table>	Swar Size	Sync PSO (s)	Ring PSO (s)	0	0.00	0.02	100	0.08	0.05	200	0.12	0.07	300	0.20	0.08	400	0.28	0.10	500	0.38	0.12	600	0.45	0.13	700	0.50	0.14
Swar Size	Sync PSO (s)	Ring PSO (s)																										
0	0.00	0.02																										
100	0.08	0.05																										
200	0.12	0.07																										
300	0.20	0.08																										
400	0.28	0.10																										
500	0.38	0.12																										
600	0.45	0.13																										
700	0.50	0.14																										

Tabela 1: Sync i Ring PSO - iteration 10000



Slika 2:

4 Закључак

Моћ графичких картица и њихових процесора расте из године у годину, и представља моћно оружје за оптимизацију, ако се њихова могућност паралелизације искорити на прави начин. *CUDA* архитектура је веома моћна технологија, која се и поред рачунарске интелигенције може искорисити у многим областима. Коришћење *CUDA* са *C/C++* програмским језиком, додатно нам пружа могућност и контролу оптимизације, због саме *low-level* природе тих програмских језика. У нашем примеру, иако је у питању само тражење минимума функција до три аргумента, резултати који су постигнути том паралелизацијом, говоре у прилог томе.

У наредном периоду планирано је да се убаци и интервал на којем ће алгоритми тражити минимум функција, као и да се дода подршка за решавање неких реалних проблема помоћу *PSO* алгоритма имплементираног у *CUDI*.

5 Литература

- Mussi L., Daolio F., Cagnoni S. - Evaluation of parallel particle swarm optimization algorithms within the CUDA™ architecture, (2011) Information Sciences, 181 (20) , pp. 4642-4657.
- Bali O., Elloumi W., Abraham A., Alimi A.M. - ACO-PSO Optimization for Solving TSP Problem with GPU Acceleration (2017)
- Cuda Toolkit Documentation - <https://docs.nvidia.com/cuda/>
- Денис Аличић. Оптимизација ројем честица.
Доступно на [http : //poincare.matf.bg.ac.rs/ denis_alicic/ri/8.html](http://poincare.matf.bg.ac.rs/~denis_alicic/ri/8.html), 2020.