

# **Implementace vybraných evolučních algoritmů v OpenCL**

Implementation of chosen evolutionary algorithms using OpenCL

Jakub Janošík



Univerzita Tomáše Bati ve Zlíně  
Fakulta aplikované informatiky  
akademický rok: 2011/2012

## ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Jakub JANOŠTÍK**  
Osobní číslo: **A09640**  
Studijní program: **B 3902 Inženýrská informatika**  
Studijní obor: **Informační a řídicí technologie**

Téma práce: **Implementace vybraných evolučních algoritmů v OpenCL**

Zásady pro vypracování:

1. Zpracujte literární rešerši na dané téma.
2. Popište algoritmy SOMA, PSO a DE.
3. Navrhněte vlastní implementaci algoritmů v OpenCL.
4. Analyzujte výkon navržených implementací.
5. Demonstrujte výsledky a formulujte závěr.

Rozsah bakalářské práce:

Rozsah příloh:

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

1. ZELINKA, Ivan, Zuzana OPLATKOVÁ, Miloš ŠEDA, Pavel OŠMERA a František VČELÁŘ. Evoluční výpočetní techniky: principy a aplikace. 1. české vyd. Praha: BEN, 2009, 534 s. ISBN 978-052-1880-688.
2. PRESS, William H., Saul A. TEUKOLSKY, William T. VETTERLING a Brian P. FLANNERY. Numerical recipes: the art of scientific computing. 3rd ed. Cambridge: Cambridge University Press, 2007, 1235 s. ISBN 978-052-1880-688.
3. GOVE, Darryl. Programování aplikací pro vícejádrové procesory. Vyd. 1. Brno: Computer Press, 2011, 416 s. ISBN 978-802-5134-870.
4. MUNSI, Aaftab. OpenCL Programming Guide. Upper Saddle River, NJ: Addison-Wesley Professional, 2011. ISBN 978-0321749642.
5. KRAMPL, Jakub. Implementace vybraných evolučních algoritmů v prostředí .NET. Zlín, 2011. Bakalářská práce. Univerzita Tomáše Bati ve Zlíně.

Vedoucí bakalářské práce:

**Ing. Erik Král**

Ústav počítačových a komunikačních systémů

Datum zadání bakalářské práce:

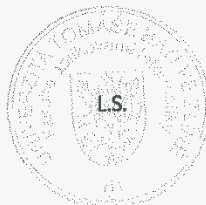
**24. února 2012**

Termín odevzdání bakalářské práce:

**8. června 2012**

Ve Zlíně dne 24. února 2012

prof. Ing. Vladimír Vašek, CSc.  
*děkan*



prof. Ing. Vladimír Vašek, CSc.  
*ředitel ústavu*

## **ABSTRAKT**

Práce obsahuje literární rešerši na téma evoluční algoritmy. Zabývá se konkrétně algoritmy Diferenciální evoluce (DE), Rojení částic (PSO) a Samo-Organizující se Migrační Algoritmus (SOMA). Dále teoretická část obsahuje přehled OpenCL standardu. V praktické části je popsána paralelní implementace algoritmů pomocí OpenCL a srovnání výsledků, vůči sériovým výpočtům implementovaných v jazyce C++.

Klíčová slova: evoluční algoritmy, optimalizace, DE, PSO, SOMA, OpenCL

## **ABSTRACT**

Thesis includes research on the topic of evolutionary algorithms. Specifically it deals with the Differential evolution (DE), the Particle swarm optimization (PSO) and the Self-Organizing Migration Algorithm (SOMA). Theoretical part also contains overview of the OpenCL standard. In the practical part is described parallel implementation of algorithms using OpenCL and comparison with serial calculations implemented using C++ language.

Keywords: evolutionary algorithms, optimization, DE, PSO, SOMA, OpenCL

Rád bych poděkoval panu Ing. Eriku Královi, vedoucímu mé bakalářské práce, za postřehy při tvorbě algoritmů a za čas, který si na mě vyhradil.

**Prohlašuji, že**

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

**Prohlašuji,**

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně

.....

podpis diplomanta

## OBSAH

<b>ÚVOD.....</b>	<b>10</b>
<b>I TEORETICKÁ ČÁST.....</b>	<b>11</b>
<b>1 EVOLUČNÍ ALGORITMY .....</b>	<b>12</b>
1.1 ÚVOD DO EVOLUČNÍCH ALGORITMŮ.....	12
1.2 ZÁKLADNÍ POJMY .....	12
1.2.1 Populace .....	12
1.2.2 Účelová funkce.....	12
1.2.3 Dimenze .....	12
1.2.4 Obecný popis ukončovacích kritérií.....	13
1.2.4.1 Referenční kritéria .....	13
1.2.4.2 Kritéria vyčerpání prostředků .....	13
1.2.4.3 Kritéria založená na zlepšení .....	13
1.2.4.4 Kritéria založená na pohybu .....	13
1.2.4.5 Kritéria založená na distribuci .....	14
<b>2 VYBRANÉ EA.....</b>	<b>15</b>
2.1 SOMA .....	15
2.1.1 Princip .....	15
2.1.2 Parametry .....	17
2.1.2.1 Mass .....	17
2.1.2.2 Step .....	17
2.1.2.3 Perturbace (PRT) .....	17
2.1.3 Strategie.....	17
2.1.3.1 Všichni k jednomu (All To one) .....	17
2.1.3.2 Všichni ke všem (All To All) .....	18
2.1.3.3 Adaptivně všichni ke všem (All To All Adaptive) .....	18
2.1.3.4 Všichni k jednomu náhodně (All To One Rand). .....	18
2.2 DE 18	
2.2.1 Princip DE .....	19
2.2.2 Parametry .....	20
2.2.2.1 Práh křížení (CR) .....	20
2.2.2.2 Mutační konstanta (F) .....	20
2.2.3 Mutace.....	20
2.2.4 Křížení.....	20
2.2.5 Varianty diferenciální evoluce .....	21
2.3 PSO.....	21
2.3.1 Princip PSO.....	22
2.3.2 Parametry .....	24
2.3.2.1 Faktory učení (c1, c2) .....	24
2.3.2.2 Setrvačnost - $w_{start}$ , $w_{end}$ .....	24
2.3.2.3 Omezující faktor - $\chi$ .....	24
<b>3 PARALELNÍ IMPLEMENTACE – POPULAČNÍ MODEL Y .....</b>	<b>26</b>
3.1 FARMÁŘSKÝ MODEL.....	26
3.2 DIFÚZNÍ MODEL.....	27
3.2.1 Lokální selekce.....	27

3.3	MIGRAČNÍ MODEL .....	29
<b>4</b>	<b>OPEN CL .....</b>	<b>31</b>
4.1	ÚVOD DO OPENCL .....	31
4.2	PLATFORMNÍ MODEL .....	32
4.3	OPENCL OBJEKTY .....	32
4.3.1	Compute devices .....	32
4.3.1.1	Compute unit.....	32
4.3.1.2	Processing element .....	32
4.3.2	Paměťové objekty .....	33
4.3.2.1	Datové pole .....	33
4.3.2.2	Images .....	33
4.3.3	Executable objekty .....	33
4.3.3.1	Compute kernel.....	33
4.3.3.2	Compute program .....	33
4.3.4	Work units .....	33
4.3.4.1	work-item.....	33
4.3.4.2	Work-group.....	34
4.3.4.3	Work-item identifikátory .....	34
4.3.5	Prostory adres .....	34
4.4	EXECUTION MODEL .....	35
4.4.1	Datově paralelní model (Data-Parallel programming model).....	36
4.4.2	Úlohově paralelní model (Task-Parallel programming model) .....	36
4.4.3	Synchronizace .....	36
4.5	HARDWARE .....	37
4.5.1	Struktura.....	37
4.5.1.1	Thread processing cluster .....	37
4.5.1.2	Procesory, multiprocesory .....	38
<b>II</b>	<b>PRAKTICKÁ ČÁST .....</b>	<b>40</b>
<b>5</b>	<b>UŽIVATELSKÁ DOKUMENTACE .....</b>	<b>41</b>
5.1	METODY.....	41
5.2	DEFAULTNÍ HODNOTY .....	42
5.3	ZACHYCENÍ CHYB .....	42
<b>6</b>	<b>PROGRAMÁTORSKÁ DOKUMENTACE.....</b>	<b>43</b>
6.1	OPENCL API.....	43
6.1.1	Inicializace .....	43
6.1.2	Alokace paměti.....	45
6.1.3	Tvorba kernelu .....	46
6.1.4	Spuštění kernelu .....	48
6.1.5	Ukončení .....	49
6.2	OPENCL KERNEL .....	50
6.2.1	Organizace dat.....	50
6.2.2	Společné funkce .....	51
6.2.2.1	Generátor pseudonáhodných čísel .....	51
6.2.3	Společné proměnné .....	51
6.2.4	SOMA_kernel .....	52
6.2.4.1	Popis proměnných.....	52



6.2.4.2	Popis kódu.....	52
6.2.5	Popis PSO_kernel.....	53
6.2.5.1	Popis proměnných.....	53
6.2.5.2	Popis kódu.....	54
6.2.6	Popis DE_kernel.....	55
6.2.6.1	Popis proměnných.....	55
6.2.6.2	Popis kódu.....	56
<b>7</b>	<b>SROVNÁNÍ VÝKONU .....</b>	<b>58</b>
7.1	POPIS TESTOVACÍCH FUNKCÍ .....	58
7.1.1	První de Jongova funkce .....	58
7.1.2	Rastriginova funkce .....	58
7.1.3	Griewangkova funkce .....	59
7.1.4	Rosenbrokovo sedlo .....	60
7.2	PARAMETRY MĚŘENÍ .....	61
7.3	NAMĚŘENÉ HODNOTY .....	62
7.3.1	PSO .....	62
7.3.2	SOMA .....	64
7.3.3	DE .....	66
7.4	ZHODNOCENÍ NAMĚŘENÝCH DAT .....	68
7.4.1	Závislost na testovací funkci .....	68
7.4.2	Závislost na velikosti populace .....	69
7.4.3	Srovnání výkonu CPU a GPU .....	69
	<b>ZÁVĚR .....</b>	<b>70</b>
	<b>ZÁVĚR V ANGLIČTINĚ.....</b>	<b>71</b>
	<b>SEZNAM POUŽITÉ LITERATURY.....</b>	<b>72</b>
	<b>SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK.....</b>	<b>74</b>
	<b>SEZNAM OBRÁZKŮ .....</b>	<b>75</b>
	<b>SEZNAM TABULEK.....</b>	<b>76</b>
	<b>SEZNAM PŘÍLOH.....</b>	<b>77</b>

## ÚVOD

Evoluční algoritmy, založené na principech přirozeného výběru, patří k velmi výkonným metodám globální optimalizace. Jsou využity v mnoha inženýrských aplikacích. Obecně jsou evoluční algoritmy schopny najít přijatelné řešení optimalizovaného problému v rozumném čase. Nicméně, při nasazení evolučních algoritmů na komplexní a složité problémy, čas nutný k hledání vhodného řešení, může být nepříjemně dlouhý. Jedním z řešení, které se za poslední dobu objevilo, je paralelizace evolučních algoritmů.

Cílem práce je paralelně implementovat vybrané evoluční algoritmy pomocí standardu OpenCL. Tedy přesunout výpočet paralelních evolučních algoritmů na grafickou kartu, ze které se v posledních letech stává velmi výkonný více-jádrový procesor určený pro paralelní výpočty.

V teoretické části práce je zpracována literární rešerše na téma evolučních algoritmů. Konkrétně se zabývá diferenciální evolucí, rojem částic a samo-organizujícím se migračním algoritmem. Popisuje parametry i možné strategie, které řídí činnost optimalizačního procesu.

V další kapitole jsou v práci popsány možné strategie pro paralelizaci evolučních algoritmů. V této kapitole jsou popsány tři hlavní modely určené pro paralelizaci evolučních algoritmů

V rámci teoretické části této práce je také popsán standard OpenCL. Je zde zmíněna hlavní struktura OpenCL a základy pravidel pro práci s tímto standardem.

V praktické části se zaměřuji na popis, mnou vytvořené, paralelní implementace evolučních algoritmů a následné srovnání výkonu s výpočty prováděnými na procesoru.

## **I. TEORETICKÁ ČÁST**

# 1 EVOLUČNÍ ALGORITMY

## 1.1 Úvod do evolučních algoritmů

Evoluční algoritmy (EA) jsou založeny na teorii evoluce formulované Charlesem Darwinem. Špatné vlastnosti jsou eliminovány z populace, protože se objevují u jedinců, kteří nepřežijí proces selekce. [1]

EA se řadí mezi stochastické optimalizační metody, které se velmi hodí k řešení obtížných problémů, u kterých máme málo informací o prohledávaném prostoru. Tyto algoritmy udržují populaci jedinců, kde každý člen populace je možným řešením praktického problému, který je specifickým způsobem zakódován.

Členové populace jsou iterativně měněni použitím přírodou inspirovaných operátorů, jako mutace, rekombinace, selekce. [2]

## 1.2 Základní pojmy

### 1.2.1 Populace

Velikost populace je důležitým parametrem, protože určuje, jestli EA dokáže najít kvalitní řešení v rozumném čase. Pokud je populace příliš malá, může mít algoritmus problém s identifikací dobrého řešení. Na druhou stranu, příliš velká populace plýtvá výpočetními zdroji. [1]

### 1.2.2 Účelová funkce

Účelovou funkcí, v rámci této práce, rozumíme funkci, kdy skrz její optimalizaci hledáme optimální hodnoty jejích argumentů. Jinými slovy účelovou funkci chápeme jako  $N$  rozměrnou hyperplochu, v jejíž rámci hledáme minimum, či maximum.[6]

### 1.2.3 Dimenze

Dimenze určuje počet parametrů, které algoritmus vzhledem k účelové funkci optimalizuje. [3]

### 1.2.4 Obecný popis ukončovacích kritérií

#### 1.2.4.1 Referenční kritéria

V reálných problémech není optimum většinou známo, proto jsou tyto kritéria použitelná pouze pro testovací funkce.

- RefCrit - Algoritmus končí, když určité procento populace dosáhne konvergence v minimu. [4]

#### 1.2.4.2 Kritéria vyčerpání prostředků

Tyto ukončovací kritéria berou ohled na omezené výpočetní zdroje a jsou silně závislé na použité optimalizované funkci.

- Maximální počet iterací - Ukončovací parametr, který udává maximální počet generací vytvořených za běhu evolučního algoritmu. Parametr definuje uživatel dle obtížnosti problému. [5] [4]

#### 1.2.4.3 Kritéria založená na zlepšení

Pokud populace dosahuje pouze malého zlepšení za určený čas. Optimalizace by se měla zastavit.

- ImpBest - Zlepšení nejlepší hodnoty účelové funkce je pod hranicí  $t$  pro počet generací  $g$ .
- ImpAv - Zlepšení průměrné hodnoty účelové funkce je pod hranicí  $t$  pro počet generací  $g$ .
- NoAcc - Během počtu generací  $g$  nebyla přijata nová pozice pro člena populace, nebo se neobjevilo zlepšení v populaci. [4]

#### 1.2.4.4 Kritéria založená na pohybu

Namísto zlepšení se kontroluje pozice jedinců.

- MovObj - Pohyb v populaci, s ohledem na průměrnou hodnotu účelové funkce, je pod hranicí  $t$  pro počet generací  $g$ .
- MovPar - Pohyb v populaci, s ohledem na pozici, je pod hranicí  $t$  pro počet generací  $g$ . [4]

#### *1.2.4.5 Kritéria založená na distribuci*

Obecně můžeme tvrdit, že je dosaženo konvergence, pokud jsou jedinci blízko u sebe. Protože není známo optimum, je testována vzdálenost k nejlepšímu jedinci.

- MaxDist - Maximální vzdálenost každého jedince od nejlepšího jedince v populaci je pod hranicí  $m$ .
- MaxDistQuick - Vzdálenost od nejlepšího jedince je kontrolována pro procento  $p$  nejlepších jedinců.
- MinDiv - Udává maximální přípustný rozdíl mezi hodnotou účelové funkce nejlepšího a nejhoršího jedince populace. Pokud uživatel nechce tuto podmínku využít, pak ji volí jako zápornou. [4][6]

## **2 VYBRANÉ EA**

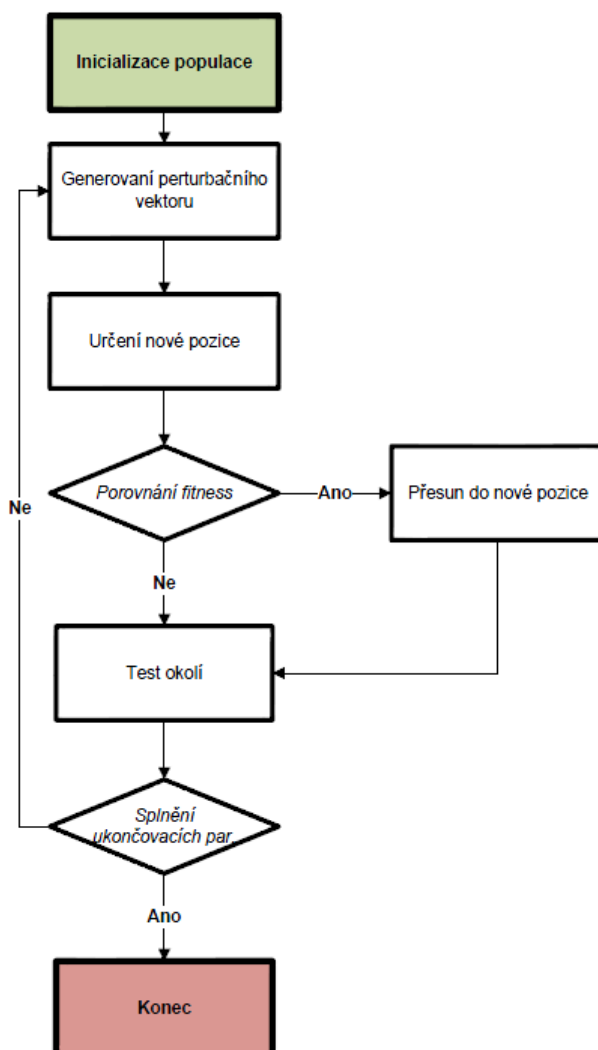
### **2.1 SOMA**

Samo-organizující se migrační algoritmus (SOMA) vznikl v roce 1999 a byl inspirován soutěživě-kooperativním chováním inteligentních jedinců, řešících společný problém. [5]

SOMA, která pracuje nad populací jedinců, je založena na samo-organizujícím se chování jedinců v „sociálním prostředí“. Může být také klasifikována jako evoluční algoritmus, i když se během běhu algoritmu netvoří žádní potomci. Pouze se, během migračního kola, mění pozice jedinců v prohledávaném prostoru.[5]

#### **2.1.1 Princip**

V této kapitole popíši princip algoritmu SOMA, za použití strategie All to One, protože se jedná o strategii, kterou jsem se rozhodl implementovat ve svém programu. Podrobný popis ostatních strategií je dobře zdokumentovaný a lze jej nalézt například v publikacích [5], či [6].



Obr. 1 Vývojový diagram algoritmu SOMA.

- Generování perturbačního vektoru - Vygeneruje se vektor náhodných čísel z intervalu  $\langle 0;1 \rangle$ , který nazýváme perturbační vektor (PRTVector). Pokud je prvek vektoru menší, než hodnota parametru PRT, vloží se na jeho místo 1. V opačném případě 0.
- Určení nové pozice - Pro výpočet nové pozice se využije následujícího vztahu:

$$x_{id} = x_{start,id} + (x_{L,d} - x_{start,id}) * t * PRTVector_d \quad (1)$$

, kde  $\vec{x}$  je vektor populace,  $i$  je index částice,  $d$  je dimenze, start značí startovní pozici,  $L$  značí Leadera a  $t$  je vzdálenost, kterou jedinec doposud urazil.

$$t \in \langle 0, Step, PathLength \rangle \quad (2)$$

- Porovnání fitness - Jedinec otestuje kvalitu nové pozice. Pokud je lepší tak si pozici zapamatuje.



- Test okolí - Jedinec putuje po krocích o velikosti *Step*, do vzdálenosti *PathLength*. Vrací se do pozice s nejlepší hodnotou účelové funkce.[5][6]

### 2.1.2 Parametry

Tab. 1 Seznam parametrů a optimálních hodnot pro algoritmus SOMA.

Parametr	Doporučený rozsah
Velikost populace	>10
Mass	1,1 - 5
Step	0,11 - Mass
PRT	0 - 1

#### 2.1.2.1 Mass

Určuje vzdálenost od vedoucího jedince, na které se daný jedinec zastaví. Hodnota  $< 1$  vede k degeneraci, protože se jedinec zastaví před vedoucím jedincem. Hodnota  $= 1$ , znamená, že se jedinec zastaví na pozici vedoucího jedince. [5][6]

#### 2.1.2.2 Step

Tento parametr udává velikost kroku, s jakým částice budou prohledávat hyperplochu. Nižší hodnota vede k podrobnějšímu prohledání plochy za cenu vyšší výpočetní náročnosti. [5][6]

#### 2.1.2.3 Perturbace (PRT)

Parametr, dle kterého se tvoří tzv. perturbační vektor. Ten ovlivňuje míru, s jakou se bude jedinec pohybovat k vedoucímu jedinci. S vyšší hodnotou PRT vzrůstá konvergence k lokálním extrémům. Optimální hodnota se pohybuje okolo 0,1. [5][6]

### 2.1.3 Strategie

#### 2.1.3.1 Všichni k jednomu (All To one)

V populaci existuje Leader. Tím rozumějme jedince s nejvyšší hodnotou účelové funkce. Zbytek populace poté migrují směrem k němu. [5][6]

### **2.1.3.2 Všichni ke všem (All To All)**

V této strategii se nenachází Leader. Populace migruje ke všem členům stejným způsobem jako v předchozí strategii, ale na konci migračního kola se jedinec vrací na pozici s nejlepší hodnotou účelové funkce. [5][6]

Tato strategie prohledává větší prostor řešení a má větší pravděpodobnost nalezení globálního extrému za cenu vyšší výpočetní náročnosti. [5][6]

### **2.1.3.3 Adaptivně všichni ke všem (All To All Adaptive)**

Jedná se o modifikaci předchozí strategie, kdy jedinec nemění svou pozici až na konci migračního kola, ale po každém přesunu porovnává hodnotu účelové funkce a přesouvá se na lepší pozici. Z této pozice opět migruje k ostatním členům populace[5][6]

### **2.1.3.4 Všichni k jednomu náhodně (All To One Rand).**

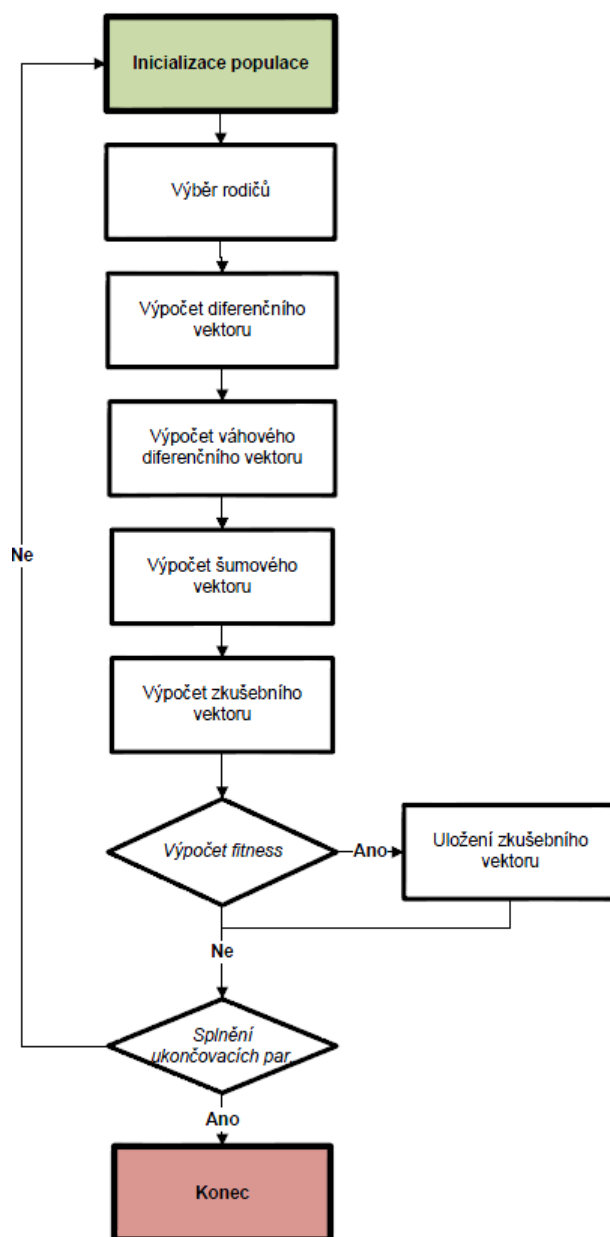
Modifikace strategie všichni k jednomu. Leader není volen na základě hodnoty účelové funkce, ale je volen náhodně. [5][6]

## **2.2 DE**

Diferenciální evoluce (DE) je velmi jednoduchý, ale zároveň velmi silný stochastický nástroj pro optimalizaci. Od svého počátku prokázal, že je efektivní a robustní technikou pro optimalizaci funkcí a byl použit k řešení mnoha vědeckých i inženýrských problémů.

DE byla poprvé navržena Kennethem V. Pricem a R. Stornem v roce 1995, když se snažili vyřešit problém aproximace Čebyševými polynomy. DE vychází z algoritmu genetického žíhání, vyvinutého Kennethem V. Pricem. Potíže s pomalou konvergencí genetického žíhání a problémy, při určování efektivních kontrolních parametrů, vedli V. Price k modifikaci genetického žíhání. Při změně boolean operací s binárními hodnotami na aritmetické operace s reálnými hodnotami, diskutovali operátor diferenciální mutace, který se později ukázal jako klíčový, při vzniku DE.[7]

### 2.2.1 Princip DE



Obr. 2 Vývojový diagram algoritmu DE.

- Výběr rodičů - Jedná se o náhodný výběr tří jedinců, kteří se budou, spolu s aktivním jedincem, podílet na tvorbě nového potomka.
- Výpočet diferenčního vektoru - Diferenční vektor získáme výpočtem rozdílu, mezi druhým a třetím jedincem
- Výpočet váhového diferenčního vektoru - Tento vektor získáme mutováním diferenčního vektoru. Neboli vynásobením mutační konstantou  $F$ .
- Výpočet šumového vektoru - Jedná se o součet předchozího vektoru s třetím vybraným jedincem.

- Výpočet zkušebního vektoru - Tento vektor dostaneme křížením šumového vektoru s aktivním jedincem. Výpočet se liší dle použité metody křížení.
- Výpočet fitness - Porovnání zkušebního vektoru s aktivním jedincem. Do další iterace vstupuje lepší z obou.

### 2.2.2 Parametry

Tab. 2 Seznam parametrů a optimálních hodnot pro algoritmus DE.

Parametr	Doporučený rozsah
Velikost populace	> 4
F	0 - 2
CR	0 - 1

#### 2.2.2.1 Práh křížení (CR)

Určuje míru s jakou se nový jedinec bude "podobat" rodiči. Při hodnotě  $CR = 0$  získáváme kopii aktivního jedince, a naopak při  $CR = 1$  je nový jedinec tvořen pouze šumovým vektorem.[5][6]

#### 2.2.2.2 Mutační konstanta (F)

Určuje počet rodičů podílejících se na tvorbě potomka a míru jejich působení. [6]

### 2.2.3 Mutace

Diferenciální mutace vyžaduje k tvorbě potomka čtyř rodičů. Pro každého jedince jsou z populace vybráni tři partneři, pomocí nichž se vytváří šumový vektor (viz. 2.2.4 Varianty diferenciální evoluce). Šumový vektor chápeme jako mutaci kombinace rodičů.[5] [6]

### 2.2.4 Křížení

Na rozdíl od genetických algoritmů, v diferenciální evoluci nastává křížení až po dokončení mutace. A to tak, že se kříží šumový vektor s rodičem, který se nepodílel na jeho tvorbě. Nově vzniklý jedinec se nazývá zkušební vektor. Křížení probíhá za pomoci prahu křížení (CR), který udává s jakou pravděpodobností, příslušný parametr zkušebního jedince, bude patřit šumovému vektoru, či čtvrtému rodiči.[5] [6]

### 2.2.5 Varianty diferenciální evoluce

Jedná se o různé způsoby tvorby šumového vektoru. Značení je uváděno ve tvaru DE/metoda výběru rodiče/počet perturbujících vektorů/metoda křížení. [6]

Šumový vektor můžeme získat dle jednoho z následujících vztahů

- DE/best/1/exp

$$v = x_{best,j}^G + F \cdot (x_{r2,j}^G - x_{r3,j}^G) \quad (3)$$

- DE/rand/1/exp

$$v = x_{r1,j}^G + F \cdot (x_{r2,j}^G - x_{r3,j}^G) \quad (4)$$

- DE/rand-to-best/1/exp

$$v = x_{r1,j}^G + \lambda \cdot (x_{best,j}^G - x_{i,j}^G) + F \cdot (x_{r1,j}^G - x_{r2,j}^G) \quad (5)$$

- DE/best/2/exp

$$v = x_{best,j}^G + F \cdot (x_{r1,j}^G + x_{r2,j}^G - x_{r3,j}^G - x_{r4,j}^G) \quad (6)$$

- DE/rand/2/exp

$$v = x_{r5,j}^G + F \cdot (x_{r1,j}^G + x_{r2,j}^G - x_{r3,j}^G - x_{r4,j}^G) \quad (7)$$

- DE/best/1/bin

$$v = x_{best,j}^G + F \cdot (x_{r2,j}^G - x_{r3,j}^G) \quad (8)$$

- DE/rand/1/bin

$$v = x_{r1,j}^G + F \cdot (x_{r2,j}^G - x_{r3,j}^G) \quad (9)$$

- DE/rand-to-best/1/bin

$$v = x_{r1,j}^G + \lambda \cdot (x_{best,j}^G - x_{i,j}^G) + F \cdot (x_{r1,j}^G - x_{r2,j}^G) \quad (10)$$

- DE/best/2/bin

$$v = x_{best,j}^G + F \cdot (x_{r1,j}^G + x_{r2,j}^G - x_{r3,j}^G - x_{r4,j}^G) \quad (11)$$

- DE/rand/2/bin

$$v = x_{r5,j}^G + F \cdot (x_{r1,j}^G + x_{r2,j}^G - x_{r3,j}^G - x_{r4,j}^G) \quad (12)$$

## 2.3 PSO

Optimalizační metoda particle swarm (PSO) je stochastická, na populaci založená, technika, kterou vyvinuli Dr. Eberhart a Dr. Kennedy v roce 1995. Při tvorbě PSO se inspirovali chováním hejna ptáků, či ryb. [8]

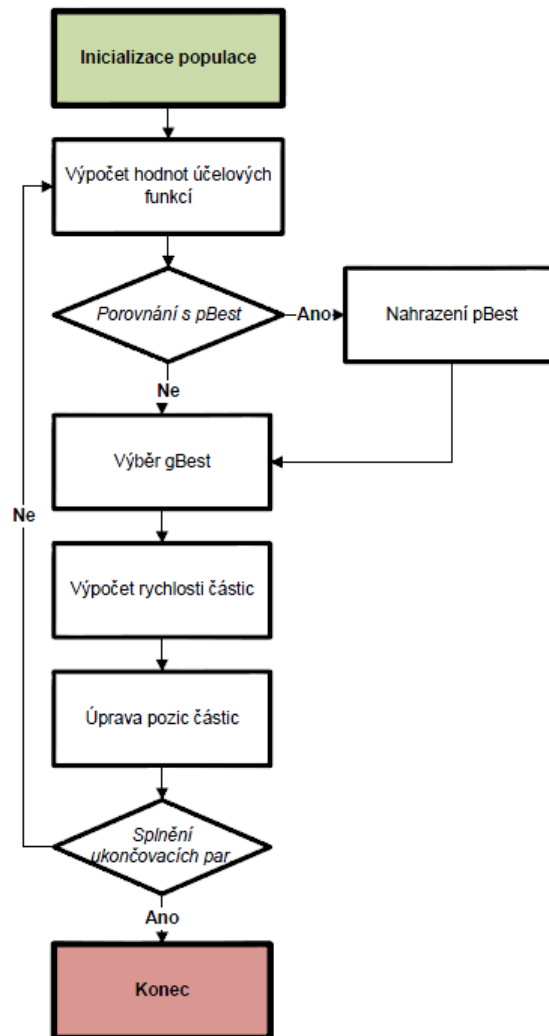
PSO sdílí mnoho podobností s technikami evolučních výpočtů, jako třeba genetickými algoritmy (GA). Systém je inicializován s populací náhodných řešení a hledá optimální řešení pomocí vývoje generací. Nicméně, na rozdíl od GA, PSO nepoužívá evoluční techniky jako křížení, či mutace. V PSO potencionální řešení, zvané částice, „proletí“ prostorem možných řešení, sledováním nejlepší částice. [5]

### 2.3.1 Princip PSO

Každá částice si uchovává souřadnice, které jsou spojeny s nejlepším řešením, kterého prozatím dosáhla. Tato hodnota se nazývá *pBest*. Další řešení, které si tato metoda uchovává, je nejlepší řešení, kterého dosáhla kterákoliv částice v okolí. Toto řešení se nazývá *lBest*. Pokud částice považuje za okolní částice celou populaci, pak je tato hodnota globálním maximem, nazývaným *gBest*. [8]

Koncept PSO se skládá ze změny rychlosti každé částice ve směru její *pBest* a *lBest*. Akcelerace je oceněna náhodným číslem generovaným zvlášť pro směry *pBest* a *lBest*. [8]

Průběh algoritmu PSO je zobrazen na obrázku níže.



Obr. 3 Vývojový diagram algoritmu PSO.

- Porovnání s pBest - Algoritmus porovná uloženou hodnotu pBest s vypočtenými hodnotami účelových funkcí. Pokud najde lepší řešení, nahradí uloženou hodnotu pBest, novou hodnotou.
- Výběr gBest - Nové hodnoty pBest se porovnají s dosavadním gBest. Pokud jsou lepší, tak jej nahradí.
- Výpočet rychlosti částic - Po nalezení dvou nejlepších hodnot, algoritmus vypočítá hodnotu rychlosti, pro každou částici, dle následujícího vzorce

$$v_{id} = v_{id} \cdot w + c1 \cdot rand \cdot (pbest_{id} - x_{id}) + c2 \cdot rand \cdot (gbest_d - x_{id}) \quad (13)$$

,kde  $\vec{v}$  je vektor rychlosti,  $i$  je index částice,  $d$  je dimenze,  $rand$  je náhodné číslo v intervalu  $<0;1>$  a  $\vec{x}$  označuje vektor populace. [9]

- Úprava pozic částic - Jakmile algoritmus získá hodnotu rychlosti, aplikují na částici podle vzorce:

$$x_{id} = x_{id} + v_{id} \quad (14)$$

,kde  $\vec{x}$  je vektor populace. [8]

### 2.3.2 Parametry

Tab. 3 Seznam parametrů a optimálních hodnot pro algoritmus PSO.

Parametr	Doporučený rozsah
Velikost populace	20 - 40
c1,c2	2,05
$w_{start}$	0,9
$w_{end}$	0,4
$\chi$	0,72984

#### 2.3.2.1 Faktory učení ( $c1$ , $c2$ )

Jedná se o faktory ovlivňující pohyb částic. Hodnota  $c1$  přitahuje částice směrem k  $pBest$ ,  $c2$  naopak přitahuje částice směrem k  $gBest$ . [8]

Většinou jsou rovny hodnotě 2. Nicméně různé hodnoty jsou použity, při řešení různých problémů. [8]

#### 2.3.2.2 Setrvačnost - $w_{start}$ , $w_{end}$

Hraje funkci vah, které určují poměr mezi globálním a lokálním hledáním. Malá setrvačnost je rychlá, ale často nenajde globální optimum. Proto hraje roli hlavně pro hledání lokálních extrémů. Zato velká setrvačnost zaručuje prohledání větší plochy za cenu rychlosti algoritmu.[9]

$$w = w_{start} - \frac{(w_{start} - w_{end}) \cdot Iterace}{MaxPočetIterací} \quad (15)$$

#### 2.3.2.3 Omezující faktor - $\chi$

Alternativní metoda k balancování globálního a lokálního hledání. Parametr  $\chi$  vychází z učících faktorů, dle následujícího vzorce:



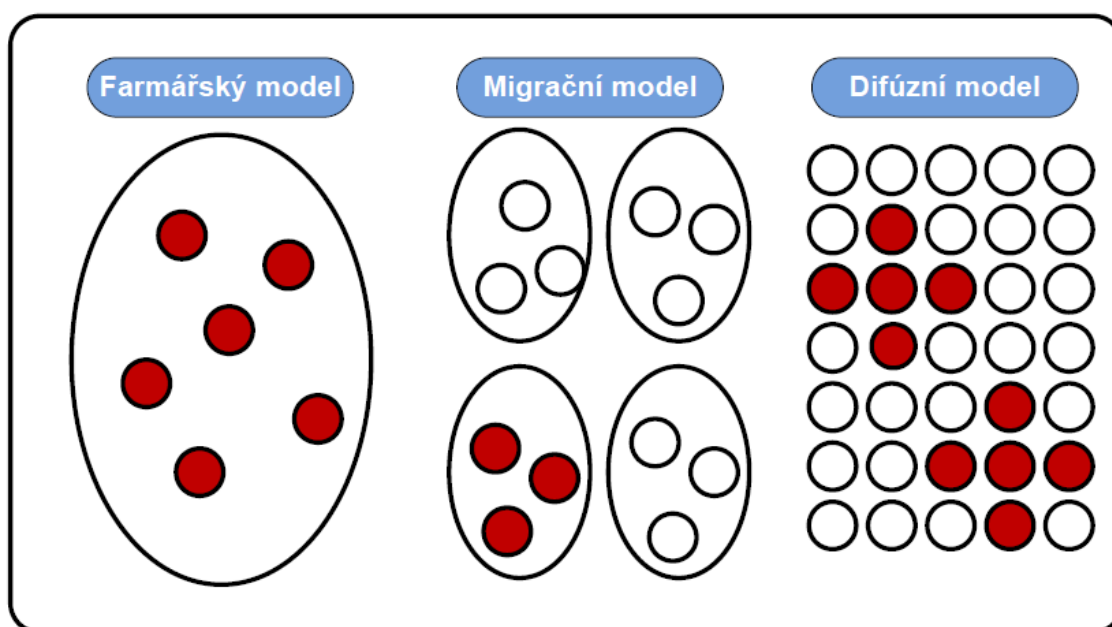
$$\chi = \frac{2}{|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}|}, \varphi = c_1 + c_2 \quad (16)$$

Bylo zjištěno, že pro  $\varphi < 4$  není garantována konvergence, protože roj "krouží" okolo nejlepšího řešení ve spirále. [9]

### 3 PARALELNÍ IMPLEMENTACE – POPULAČNÍ MODELY

Populační modely mohou být od sebe rozlišeny, buď rozsahem výběrových strategií rodičů, nebo definicí množiny jedinců, se kterou pracujeme. Definujeme tři populační modely.

- Farmářský model - Ve farmářském modelu se vybírá z celé populace. To znamená, že každý dva (nebo více) jedinci, mohou být vybráni pro tvorbu potomků.
- Difúzní model - Difúzní model omezuje výběr rodičů do lokálních sousedství.
- Migrační model - Migrační model omezuje výběr rodičů do částí populace izolovaných od sebe, zvaných sub-populace. Uvnitř sub-populace je výběr neomezený.



Obr. 4 Ukázka různých paralelních implementací.

#### 3.1 Farmářský model

Farmářský model nerozděluje populaci. Místo toho využívá zděděný paralelismus evolučních algoritmů (populace jedinců). Farmářský model koresponduje s klasickými evolučními algoritmy. [10]

Výpočty, kde je potřebná celá populace, jako výpočet hodnot účelové funkce a selekce jedinců, jsou provedeny na hlavním procesoru (host master). Všechny ostatní výpočty, které jsou prováděny na jednom, nebo dvou jedincích, mohou být distribuovány mezi množství pomocných pracovních procesů. Tyto pomocné procesy provádějí

rekombinaci, mutaci a ohodnocení účelové funkce odděleně. Toto je známo jako synchronní master-slave struktura. [5] [11]

Pro většinu problému je nejvíce časově náročné ohodnocení účelové funkce. V tomto případě je celý evoluční algoritmus prováděn na hlavním procesoru a pouze ohodnocení účelové funkce je distribuováno mezi pomocné procesy. Může být dosaženo skoro lineárního zrychlení (za podmínky, že vyhodnocení účelové funkce je časově náročnější, než komunikace mezi hlavním procesorem a zařízeními provádějícími pomocné procesy). [5] [10]

Globální model je jednoduchou cestou (vlastní pro každý evoluční algoritmus) jak zrychlit dlouhé výpočetní časy. [10]

Tato distribuce výpočtů hodnoty účelové funkce může být využita pro ostatní populační modely. [10]

### **3.2 Difúzní model**

Difúzní model manipuluje s každý jedincem zvlášť a vybírá mu partnera v lokálním sousedství, pomocí lokální selekce. Proto se objevuje difúze informace skrz populaci. Během hledání se vyvinou virtuální ostrovy. [10] [11]

#### **3.2.1 Lokální selekce**

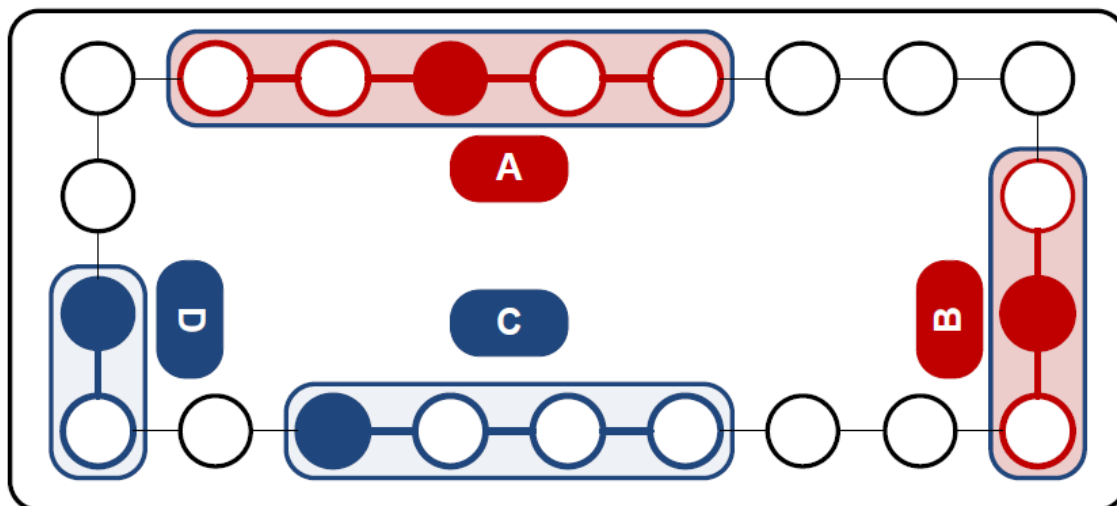
V lokální selekci každý jedinec zůstává uvnitř uzavřeného prostředí, zvaného lokální sousedství. Jedinci na sebe vzájemně působí pouze uvnitř tohoto regionu. Sousedství je definovaná struktura, ve které je populace distribuována. Sousedství můžeme chápat jako skupinu potencionálních partnerů. [10]

Prvním krokem je selekce první poloviny populace náhodným a uniformním způsobem (nebo za pomoci jiných selekčních algoritmů, jako stochastic universal sampling, nebo truncation selection). Nyní se definuje lokální sousedství pro každého zvoleného jedince. Uvnitř tohoto sousedství je zvolen partner pro reprodukci (náhodně a uniformně, nebo podle ohodnocení účelové funkce). [10]

Struktura sousedství může být:

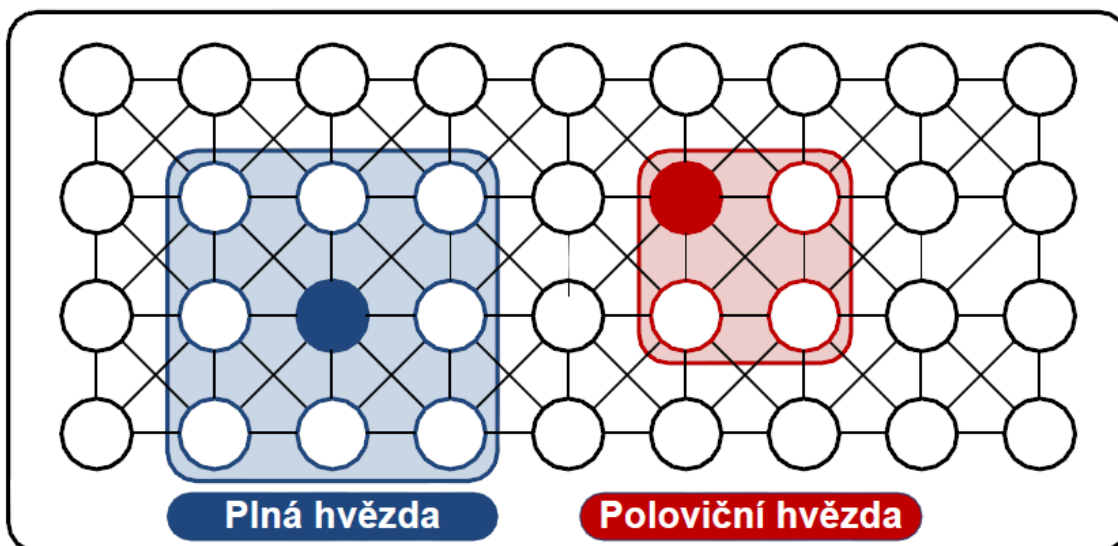
- lineární
  - plný kruh, poloviční kruh
- dvojdimenzionální

- plný kříž, poloviční kříž
- plná hvězda, poloviční hvězda
- třídímní a více komplexní s kombinací výše zmíněných struktur



Obr. 5 Ukázka lineární struktury sousedství. A - plný kruh (vzdálenost = 1,2), B - plný kruh (vzdálenost = 1), C - poloviční kruh (vzdálenost = 1,2,3), D - poloviční kruh (vzdálenost = 1). [10]

Vzdálenost mezi možnými sousedy spolu se zvolenou strukturou definuje velikost sousedství. [10]



Obr. 6 Ukázka 2D struktury sousedství. [10]

Mezi jedinci populace existuje "izolace vzdáleností". Čím menší sousedství, tím větší je vzdálenost izolace. Nicméně, díky překrývajícím se sousedstvím, dochází k větší variaci populace. Toto zajišťuje výměnu informace mezi všemi jedinci populace. [10]

Velikost sousedství určuje rychlost přenosu informace mezi jedinci populace, čímž určuje míru různorodosti a variability populace. Často je požadována větší variabilita, čímž se předchází problémům jako předčasná konvergence k lokálnímu minimu. [10]

### 3.3 Migrační model

Migrační model (migrační model) dělí populaci do vícenásobných sub-populací. Tyto sub-populace se vyvíjí nezávisle na sobě, po dobu několika generací (doba izolace). Po skončení doby izolace je několik jedinců distribuováno mezi sub-populacemi (migrace). Počet distribuovaných jedinců (migrační poměr), metoda selekce využitá pro výběr jedinců k migraci a systém migrace určuje kolik genetické různorodosti se může objevit v sub-populaci a množství informace vyměněné mezi sub-populacemi. [5] [10]

Paralelní implementace migračního modelu ukázala, nejenom snížení výpočetního času, ale také, že, v porovnání s jednopopulačním algoritmem, je potřeba menší množství ohodnocených účelových funkcí. Proto i jednoprocessorový počítač, implementující regionální model v serializované formě (pseudo-paralelní), poskytuje lepší výsledky (Algoritmus najde řešení rychleji, nebo častěji). [10]

Selekce jedinců pro migraci může probíhat:

- rovnoměrně a náhodně.
- na základě ohodnocení účelové funkce (pro migraci se využijí nejsilnější jedinci).

Existuje mnoho možností pro vytvoření struktury migrace jedinců mezi sub-populacemi. Například migrace může probíhat:

- mezi všemi sub-populacemi (kompletní neomezená síťová topologie).
- jako kruhová topologie.
- jako sousedská topologie.

Nejčastější obecná migrační strategie je neomezená migrace (kompletní síťová topologie). V tomto případě, jedinci mohou migrovat z jedné populace do další. Pro každou sub-populaci, množina potencionální imigrantů je vytvořena z jiné sub-populace. Z této množiny jsou posléze náhodně vybráni jedinci, určeni k migraci. [10]

Nejzákladnější migrační strukturou je kruhová topologie. Zde, jsou jedinci přesouváni mezi přímo sousedícími populacemi.

Podobnou strategií ke kruhové topologii je sousedská migrace. Jako v kruhové topologii je migrace uskutečněna mezi nejbližšími sousedy. Nicméně, migrace se může objevit v obou směrech. Pro každou sub-populaci jsou možní imigranti vybráni, podle požadované selekční metody, z přilehlé sub-populace, a je provedena finální selekce jedinců. [10][11]

## 4 OPEN CL

OpenCL je standardizovaný Framework sloužící k programování paralelních problémů na počítačích složených z kombinace jednoho, či více CPU, GPU a jiných procesorů. OpenCL, které bylo poprvé vydáno v prosinci 2008 s prvními produkty podporujícími OpenCL v listopadu 2009, je relativně nová technologie. S pomocí OpenCL můžeme psát programy, které poběží na široké řadě systémů. Od telefonů, přes laptopy, až po uzly v superpočítačích. Žádný jiný standart pro paralelní programování nemá takovou kompatibilitu a přenositelnost jako OpenCL.[12]

### 4.1 Úvod do OpenCL

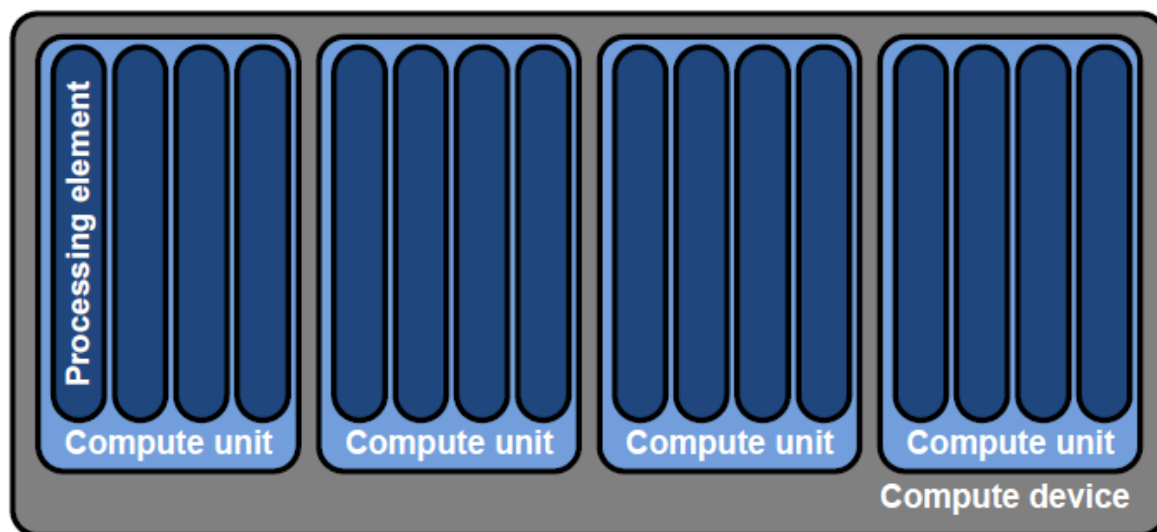
Probíhající pohyb směrem k paralelním procesorům, je hnán schopnostmi a omezeními moderní polovodičové výroby. Paralelní výpočty nabízejí zvýšení výkonu za rozumnou cenu spotřeby energie. Nicméně, aby budoucí programy získaly všechny výhody, které přichází s paralelními procesy, musí být napsány škálovatelným způsobem. Psaní škálovatelných programů bylo, po mnoho let, obtížné a dnes se stává povinným pro všechny výpočetně náročné programy. [13] [12]

Přesun k paralelismu komplikovala různorodost a heterogenita různých paralelních architektur, které jsou dnes dostupné. Tradiční procesory se stali vícejádrovými procesory s dvěma až osmi jádry. Grafické karty, které byly vždy velmi paralelní, schopné provádět stovky paralelních výpočtů najednou, se dnes stávají velmi snadno programovatelné. A to natolik, že je vhodné, bavit se o GPU jako o vícejádrových procesorech, namísto akcelérátorech pro specifické úlohy. Všechna různorodost se projevila v množství neslučitelných nástrojů a programových modelů potřebných pro programování těchto architektur, což vedlo k mnoha problémům při psaní programů pro různorodé platformy.

OpenCL je odpovědí na tento problém. Vývojáři pracující s OpenCL mohou využít jednotný soubor nástrojů a jazyk zaměřený na všechny, v současnosti vyráběné paralelní procesory. To je provedeno pomocí abstraktního modelu, který vnímá všechny architektury stejným způsobem, stejně tak jako použití execution modelu, podporujícího datový a úlohový paralelismus napříč heterogenními platformami.[12]

## 4.2 Platformní model

OpenCL vnímá heterogenní zařízení pomocí abstraktního hierarchického modelu. V tomto modelu, hostující systém koordinuje spouštění úloh, přenos dat do a z více compute devices. Každé compute device se skládá z pole compute units, které se zase skládají z množství processing elements. Jedna ze silných stránek OpenCL je, že tento model, nspecifikuje, které hardwarové zařízení hraje roli compute device. [12]



Obr. 7 Ukázka struktury platformního modelu. [14]

## 4.3 OpenCL objekty

### 4.3.1 Compute devices

Jedná se o jakýkoliv procesor, který se může podílet na výpočtech datově-paralelních programů. V rámci OpenCL můžeme, do compute devices, zařadit všechna dostupná CPU, GPU, nebo jejich kombinace. V tomto případě, nazýváme množinu všech compute devices, device group. Uvnitř této skupiny jsou si všechna zařízení rovna, což odlišuje OpenCL od CUDY. OpenCL nadále dokáže rozlišovat vícenásobné skupiny v rámci jednoho systému. [12]

#### 4.3.1.1 Compute unit

Součástí compute device. Můžeme chápat jako jednotlivá jádra procesoru.[14]

#### 4.3.1.2 Processing element

Jedná se o části jádra, které dělají samotné výpočty. [14]



### 4.3.2 Paměťové objekty

#### 4.3.2.1 Datové pole

Zápis a použití polí v OpenCL se neliší od standardního C99. K jednotlivým elementům pole přistupujeme přes ukazatel. Rozdíl mezi grafickou kartou a procesorem, při použití polí, je ten, že grafická karta při zápisu, či čtení, nevyužívá vyrovnávací mezipaměť. [14]

#### 4.3.2.2 Images

OpenCL rozlišuje dva typy obrazů. 2D a 3D obrazy. Na rozdíl od pole, jsou data uložena ve velmi optimalizované, nelineární formě, kdy jednotlivé elementy nejsou přímo přístupné, přes ukazatele. Při čtení dat, se, na rozdíl od pole, využívá texturové vyrovnávací mezipaměti. [14]

### 4.3.3 Executable objekty

#### 4.3.3.1 Compute kernel

Nejmenší součást executable objektů, která je v podstatě datově-paralelní funkce, která se spouští přímo na CPU, či GPU. Kernelem v podstatě rozumíme work item, což je součást programu, která má přístup k jednomu prvku paměti. [14]

Model využívající kernel je založen na hierarchické abstrakci výpočtů, které jsou prováděny. OpenCL kernely jsou spouštěny nad indexovým prostorem, který může být jedno, dvou, či trojrozměrný. [15] [17]

#### 4.3.3.2 Compute program

Kolekce kernelů a funkcí. Jedná se o strukturu, která zjednodušuje a zpřehledňuje práci programátorovi. [15]

### 4.3.4 Work units

#### 4.3.4.1 work-item

Nejmenší jednotka, provádějící výpočet. Můžeme ji chápat jako vlákno. Každá instance spouštěného kernelu je typicky provedena v jednom vláknu, neboli v jedné work-item. Počet work-items, v OpenCL, nazýváme globální velikost. Rozdíl mezi GPU a CPU v rámci vláken je ten, že GPU zpracovává vlákna hardwarově a můžeme pracovat s tisíci vláken. [15]

Globální velikost, určuje také rozsah problému, v následující části dokumentu popisovaný jako NDrange. [15]

#### 4.3.4.2 *Work-group*

Skupina jednotlivých work-items. Slouží k rozdělení problému do menších úseků. Důležitým prvkem v optimalizaci programu. Velikost pracovní skupiny, v OpenCL, nazýváme lokální velikost. Na grafické kartě musí být lokální velikost dělitelem velikosti globální. Oproti tomu, na CPU, je lokální velikost pokaždé rovna jedné. [15]

#### 4.3.4.3 *Work-item identifikátory*

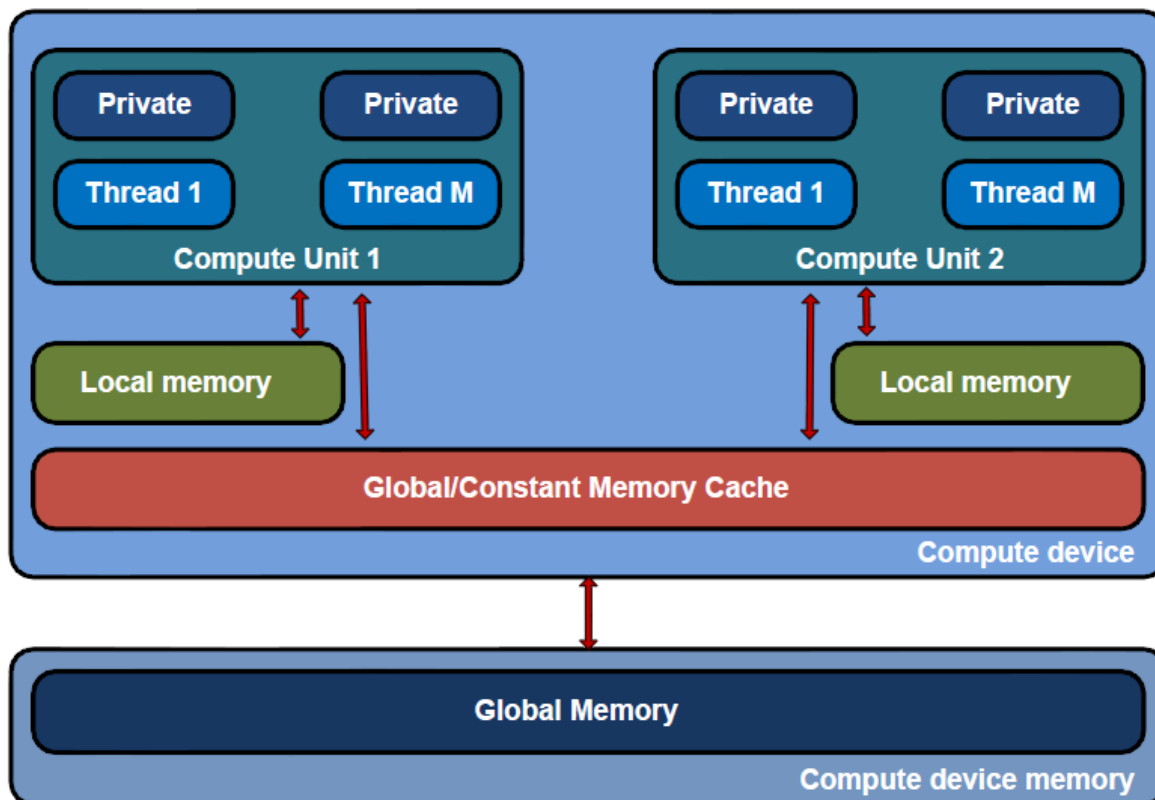
Každá work-item si je vědoma, na které části problému právě pracuje. Stejně tak může být, v rámci kernelu, pomocí zabudovaných funkcí, každá work-item identifikována. A to i v rámci jednotlivých work-groups. [15]

#### 4.3.5 **Prostory adres**

Každé compute device má globální paměť, která je největším paměťovým prostorem dostupným zařízení. Většinou DRAM umístěná mimo chip. K dispozici je také konstantní paměťový prostor limitované velikosti pouze pro čtení, který umožňuje efektivní opětovné použití neměnných parametrů výpočtu. Každá výpočetní jednotka na zařízení má místní paměť, většinou umístěnou přímo na procesoru, která má díky tomu mnohem větší šířku pásma a nižší latenci než globální paměť. Místní paměť může být čtena/změněna, kteroukoliv work-item z work-group a díky tomu umožňuje komunikaci mezi work-groups. Nakonec, připojena ke každému výpočetnímu prvku, se nachází privátní paměť, která není typicky využita programátory, ale slouží k uložení dat, která se nevešla do registrů výpočetního prvku. [15]

- Privátní (private) - Specifická pro work-item. Není viditelná pro ostatní work-items.
- Lokální (local) - Specifická pro work-group. Pouze přístupná v rámci dané skupiny.
- Konstantní (constant) - Oblast pro hostujícím systémem alokované a inicializované objekty, které jsou neměnné za běhu kernelu.
- Globální (global) - Přístupná pro všechny běžící work-items, stejně tak přístupná pro hostující systém.
- Paměť procesoru - Oblast pro datové struktury aplikace a programové data.

- PCIe paměť - Část procesorové paměti přístupné z hlavního programu a z výpočetních zařízení GPU. Změna této paměti vyžaduje synchronizaci mezi GPU a CPU. [15][16]



Obr. 8 Ukázka struktury paměťového prostoru. [14]

#### 4.4 Execution Model

OpenCL obsahuje flexibilní execution model, který zahrnuje, jak úlohový, tak datový paralelismus. Přesun dat mezi hostujícím a výpočetním zařízením, stejně tak jako OpenCL úlohy, jsou koordinovány pomocí fronty příkazů. Fronty příkazů nabízejí obecný způsob specifikování vztahů mezi úlohami, a tím zajišťují, že se úlohy vykonávají v pořadí, které uspokojuje přirozené závislosti ve výpočtech. Běžící program OpenCL může provádět úlohy paralelně, pokud jsou jejich závislosti uspokojeny, což nabízí univerzální úlohově paralelní execution model. Úlohy samotné mohou být tvořeny datově paralelními kernely, které aplikují jednu funkci, paralelně, na pole datových prvků. [15] [17]

#### 4.4.1 Datově paralelní model (Data-Parallel programming model)

V modelu datového paralelismu je výpočet definován jako sekvence instrukcí vykonána na více prvcích části paměti. Tyto prvky jsou součástí indexového prostoru, který určuje, jak se daný výpočet mapuje na work-item. V OpenCL není podmínkou.

Hierarchie toho modelu se dá rozdělit do dvou podskupin.

- Explicitní – Vývojář definuje počet work-items, které se paralelně vykonají, stejně jako rozdělí work-items do specifických work-groups.
- Implicitní – Vývojář definuje počet work-items, které se paralelně vykonají a OpenCL se postará o rozdělení work-items do specifických work-groups. [16]

#### 4.4.2 Úlohově paralelní model (Task-Parallel programming model)

Instance kernelu se vykonává nezávisle na jakémkoliv indexovém prostoru. Toto je ekvivalentní vykonávání kernelu na výpočetním zařízení s work-groups a NDRange obsahujícími jednotlivé work-items. Paralelismus je vyjádřený použitím vektoru datových typů implementovaný zařízením, vložením více úlohových problémů a/nebo původních kernelů do fronty. [16]

#### 4.4.3 Synchronizace

Dvě oblasti synchronizace v OpenCL jsou work-items v rámci jedné work-group a fronta příkazů v rámci jednoho kontextu.

Work-group bariéry umožňují synchronizaci work-items ve work-group. Každá work-item ve work-group musí prvně spustit bariéru než je jakákoliv work-item puštěna za bariéru. Buď všechny nebo žádná work-item ve work-group se musí setkat s bariérou. Momentálně není v rámci OpenCL povolena globální synchronizace.

Máme k dispozici dva způsoby synchronizace mezi příkazy ve frontě příkazů.

- Bariéra fronty příkazů - Vynucuje řazení v rámci jedné fronty. Všechny výsledné změny paměti jsou přístupné následujícím příkazům ve frontě.
- Události (Events) - Vynucuje řazení mezi frontami nebo v rámci fronty. Zařazené příkazy v OpenCL vracejí událost identifikující příkaz, stejně tak jako paměťový objekt, který se příkazem mění. Toto zajišťuje, že následující příkazy, které čekají na danou událost, vidí upravený paměťový objekt, než se spustí. [16]

## 4.5 Hardware

Díky poptávce trhu po high-definition 3D grafice se programovatelné GPU (Graphic Processor Unit) vyvinuly do paralelních, více vláknových, více jádrových procesorů s obrovskou výpočetní silou a velkou pamětí. [12]

GPU nyní nabízí mnohem rychlejší floating-point výpočty, než je možné na CPU, a mimo práci s grafikou jsou vhodné pro řešení všeobecných problémů, které je možné vyjádřit jako datově paralelní problém. [17]

Následující dělení se týká grafických karet od společnosti NVIDIA, jež jsem měl k dispozici. Struktura hardwaru ostatních společností je lehce dohledatelná ve specifikacích konkrétních zařízení.

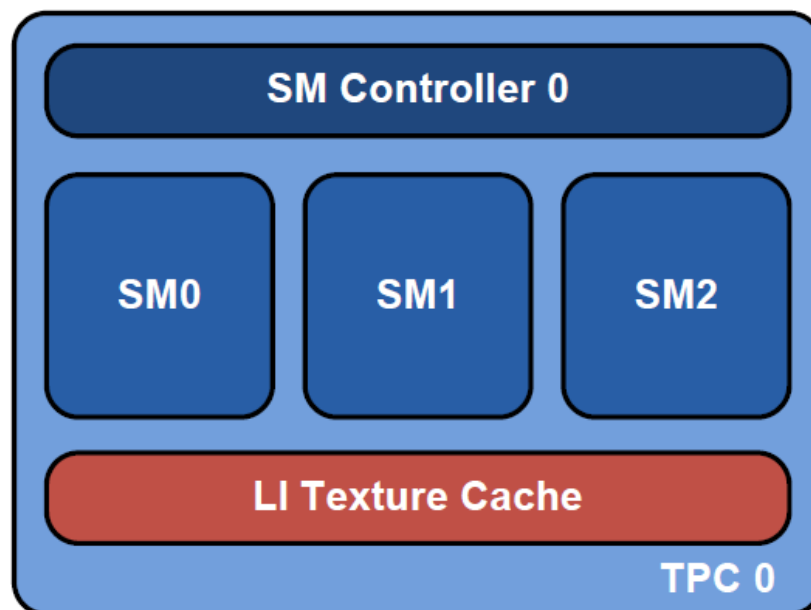
### 4.5.1 Struktura

- Thread Processing Clusters (TPC)
- Streaming Multiprocessors (SM)
- Streaming Processors (SP)
- Special Function Unit
- Double precision unit
- Local memory

[17]

#### 4.5.1.1 *Thread processing cluster*

TPC typické grafické karty obsahuje SM kontrolér, který obsluhuje vícenásobné zabudované SM, které sdílí malou společnou texturovou vyrovnávací paměť.

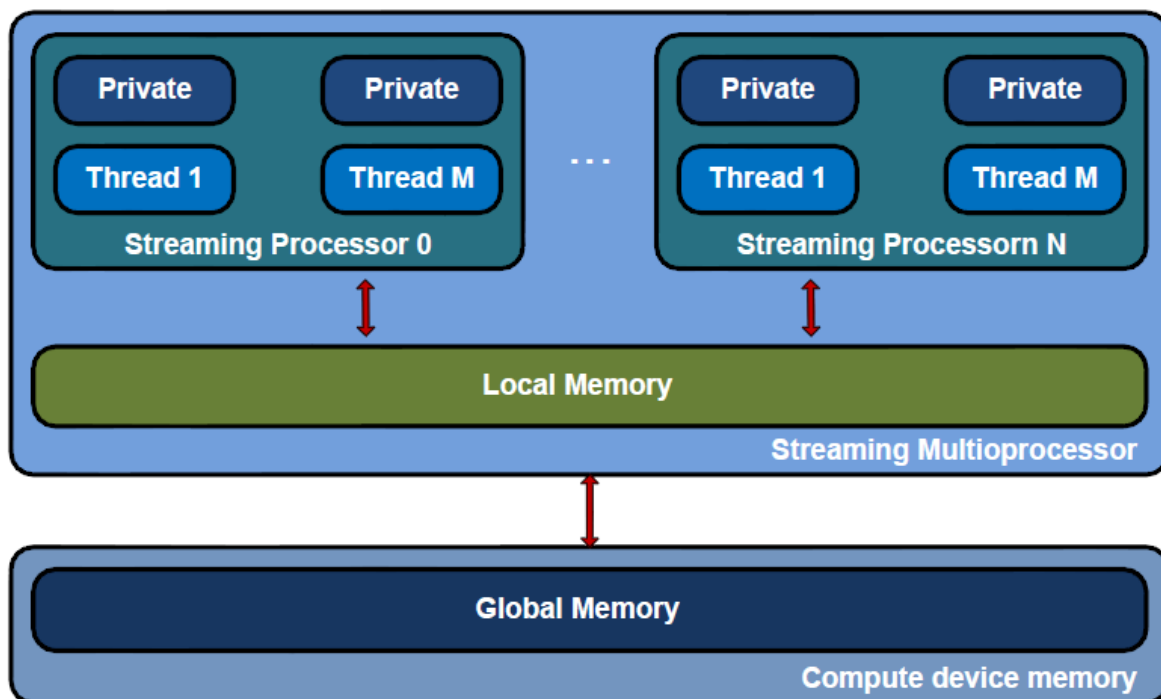


Obr. 9 Ukázka struktury TPC. [14]

#### 4.5.1.2 Procesory, multiprocesory

Všechny základní procesory jsou na moderních grafických kartách uspořádány do větších multi-procesorů. Tyto multi-procesory musí vykonávat stejnou instrukci na jiných datech (SIMD model, Single Instruction Multiple Data). [14]

Každý multiprocesor má přístup ke sdílené lokální paměti, která slouží ke sdílení dat mezi jednotlivými procesory.



Obr. 10 Ukázka přístupů k paměti pro procesory a multiprocesory. [14]

## **II. PRAKTICKÁ ČÁST**



## 5 UŽIVATELSKÁ DOKUMENTACE

Jednotlivé algoritmy jsou implementovány jako třídy v jazyce c++. Pro implementaci tříd do projektu jsou potřeba tři soubory. Hlavičkový soubor, soubor s kódem a kernel. Názvy jednotlivých souborů jsou uvedeny v tabulce níže.

Tab. 4 Seznam souborů tvořících paralelní implementaci vybraných EA.

Algoritmus	Hlavičkový soubor	Kód	Kernel
<b>PSO</b>	PSO_CL.h	PSO_CL.cpp	PSO_CL.cl
<b>SOMA</b>	SOMA_CL.h	SOMA_CL.cpp	SOMA_CL.cl
<b>DE</b>	DE_CL.h	DE_CL.cpp	DE_CL.cl

### 5.1 Metody

Tab. 5 Seznam a popis metod jednotlivých tříd.

Definice metody	Funkce metody
set_pop_size(int value)	Nastavení velikosti populace.
set_dim(int value)	Nastavení dimenze problému.
set_max_iterations(int value)	Nastavení maximálního počtu iterací.
set_min_div(float value)	Nastavení hodnoty minimal difference.
return_error_code(void)	Metoda vracející kód chyby.
cout_error_message(void)	Metoda vypisující na konsoli popis chyby.
return_results(void)	Metoda vracející ukazatel na pole s nejlepším členem populace.
<b>PSO</b>	
run_PSO_cl(void)	Spuštění algoritmu PSO.
set_c1(float value)	Nastavení parametru c1.
set_c2(float value)	Nastavení parametru c2.
<b>SOMA</b>	
run_SOMA_cl(void)	Spuštění algoritmu SOMA.
set_path_length(float value)	Nastavení parametru path_length.
set_step(float value)	Nastavení parametru step.
set_PRT(float value)	Nastavení parametru PRT.
<b>DE</b>	
run_DE_cl(void)	Spuštění algoritmu DE.
set_F(float value)	Nastavení parametru F.
set_CR(float value)	Nastavení parametru CR.

## 5.2 Defaultní hodnoty

Tab. 6 Seznam parametrů a defaultních hodnot pro vybrané algoritmy.

Parametr		Hodnota	
pop_size		10	
dimension		3	
max_iterations		2000	
min_div		-1.0	
PSO		SOMA	
c1,c2	2.05	Path_Length	5.0
		Step	0.11
		PRT	0.9
		DE	
		F	0.3
		CR	0.8

## 5.3 Zachycení chyb

V případě výskytu chyby, lze chybu identifikovat pomocí metod *return\_error\_code* a *cout\_error\_message*. První z nich vrací id chyby, kterou lze posléze dohledat v openCL specifikaci.

V případě, že má uživatel k dispozici konsoli, může využít druhou metodu, která mu sdělí, ve které části kódu se stala chyba. A také vypíše OpenCL specifikaci chyby a její kód.

## 6 PROGRAMÁTORSKÁ DOKUMENTACE

Programová část OpenCL je rozdělena do dvou celků. První část je vlastní program, spuštěný na OpenCL zařízení (ve většině případů GPU). Tento program je složený z kernelů a dalších pomocných funkcí, které vykonávají samotné výpočty. Druhá část je program spouštěný na hostujícím systému (ve většině případů CPU). Tento program umožňuje správu OpenCL prostředí.

### 6.1 OpenCL API

Jak již bylo řečeno výše, jedná se o program spouštěný na procesoru. Jeho činnost můžeme rozdělit do následujících částí:

- Inicializace
- Alokace paměti
- Tvorba kernelu
- Spuštění kernelu
- Ukončení

#### 6.1.1 Inicializace

V této části zvolíme zařízení a vytvoříme kontext, ve kterém poběží výpočty.

- **Platform**

Platformou rozumíme hostující systém spolu s množinou zařízení, spravovaných OpenCL frameworkem, který umožňuje aplikaci sdílet zdroje a spouštět kernely v rámci platformy. Platformy jsou reprezentovány objektem *cl\_platform*, který může být inicializován pomocí následující funkce:

```
cl_int oclGetPlatformID(cl_platform_id *platforms)
```

- **Device**

Zařízení jsou reprezentovány pomocí *cl\_device* objektů. Tyto objekty inicializujeme pomocí následující funkce:

```
cl_device_type device_type,    // Bitové pole identifikující typ.  
//Pro GPU se vloží hodnota CL_DEVICE_TYPE_GPU  
cl_uint num_entries,         // Počet zařízení  
cl_device_id *devices,       // Ukazatel na objekt cl_device  
cl_uint *num_devices)        // Počet zařízení, které se shodují s  
device_type
```

- **Context**

Kontext definuje cele OpenCL prostředí. Zahrnuje OpenCL kernely, zařízení, správu paměti, úkolové fronty, apod. Kontexty jsou v OpenCL referencovány pomocí *cl\_kontext* objektů, které musí být inicializovány pomocí následující funkce:

```
cl_context clCreateContext  
(const cl_context_properties *properties, // vlastnosti  
    cl_uint num_devices, // počet zařízení  
    const cl_device_id *devices, // Ukazatel na cl_device  
//objekt  
    void (*pfn_notify)(const char *errinfo, const void  
*private_info, size_t cb, void *user_data),  
    void *user_data,  
    cl_int *errcode_ret) // Výsledek vracející chybový kód
```

- **Command-queue**

OpenCL úkolová fronta, je objekt, do kterého se vkládají příkazy, které mají být spuštěny na OpenCL zařízení. Úkolová fronta je vytvořena pro konkrétní zařízení, v konkrétním kontextu. Použití vícenásobné fronty, umožňuje aplikaci zařadit vícenásobné nezávislé příkazy, bez použití synchronizace.

```
cl_command_queue clCreateCommandQueue (cl_context context,  
    cl_device_id device,  
    cl_command_queue_properties properties,  
    cl_int *errcode_ret)
```

Konkrétní ukázka inicializace:

```
cl_int error = 0;
cl_platform_id platform;
cl_context context;
cl_command_queue queue;
cl_device_id device;

//Inicializace platformy
error = oclGetPlatformID(&platform);
if( error != CL_SUCCESS )
{
    cout<<"Error occured while getting platform id."<<endl;
    exit(error);
}

//Inicializace zařízení
error = clGetDeviceIDs(platform,CL_DEVICE_TYPE_GPU,1,&device,NULL);
if( error != CL_SUCCESS )
{
    cout<<"Error occured while getting device id."<<endl;
    exit(error);
}

//Inicializace kontextu
context = clCreateContext(0,1,&device,NULL,NULL,&error);
if( error != CL_SUCCESS )
{
    cout<<"Error occured while creating context."<<endl;
    exit(error);
}

//Inicializace fronty
queue = clCreateCommandQueue(context, device, 0, &error);
if( error != CL_SUCCESS )
{
    cout<<"Error occured while creating command-queue."<<endl;
    exit(error);
}
```

### 6.1.2 Alokace paměti

K reprezentaci alokované paměti na zařízení využíváme datový typ *cl\_mem*. K alokaci využijeme následující funkce:

```
cl_mem clCreateBuffer (cl_context context,    // Kontext, ve kterém bude
//paměť alokována
                    cl_mem_flags flags,
                    size_t size,    // Velikost v bytech
                    void *host_ptr,
                    cl_int *errcode_ret)
```

, kde *flags* je bitová proměnná dosahující těchto hodnot:

- CL\_MEM\_READ\_WRITE
- CL\_MEM\_WRITE\_ONLY
- CL\_MEM\_READ\_ONLY
- CL\_MEM\_USE\_HOST\_PTR
- CL\_MEM\_ALLOC\_HOST\_PTR
- CL\_MEM\_COPY\_HOST\_PTR

Konkrétní ukázka alokace paměti:

```
//Pole obsahující populaci
float * population_host = new float[data_size];

//Velikost potřebná pro uložení populace
const int mem_size_population = sizeof(float)*data_size;

//Buffer pro populaci
cl_mem res_population_device = clCreateBuffer(context,
CL_MEM_READ_WRITE, mem_size_population, res_population_host, &error);
if( error != CL_SUCCESS )
{
    cout<<"Error occured while creating buffers."<<endl;
    exit(error);
}

//Vynucení zápisu bufferu
error = clEnqueueWriteBuffer(queue, res_population_device, CL_TRUE, 0,
mem_size_population, (void*)res_population_host, 0, NULL, NULL);
if( error != CL_SUCCESS )
{
    cout<<"Error occured while forcing buffer write."<<endl;
    exit(error);
}
```

### 6.1.3 Tvorba kernelu

V tomto kroku se vytváří kernel a OpenCL program. Program tvoří množina kernelů a funkcí, volaných kernelem. Při tvorbě programu, musíme specifikovat, které soubory tvoří program a ty pak zkompileovat. Všechny kernely jsou kompilovány za běhu programu.

K vytvoření programu nám slouží následující funkce:

```
cl_program clCreateProgramWithSource (cl_context context,
                                     cl_uint count,    // počet souborů
                                     const char **strings, // pole znaků tvořících soubor
                                     const size_t *lengths, // pole specifikující délky
                                     //souborů
                                     cl_int *errcode_ret) // vrácený chybový kód
```

Po vytvoření programu, musí dojít k jeho kompilaci:

```
cl_int clBuildProgram (cl_program program,
                      cl_uint num_devices,
                      const cl_device_id *device_list,
                      const char *options,    // Vlastnosti kompilátoru
                      void (*pfn_notify) (cl_program, void *user_data),
                      void *user_data)
```

Pro zobrazení logu kompilátoru (varování, chyby,...), můžeme použít následující:

```
cl_int clGetProgramBuildInfo (cl_program program,
                              cl_device_id device,
                              cl_program_build_info param_name,    // Parametr, který
//chceme znát
                              size_t param_value_size,
                              void *param_value,    // Odpověď
                              size_t *param_value_size_ret)
```

Nakonec musíme získat vstupní body do programu. Toho dosáhneme vytvořením *cl\_kernel* objektů:

```
cl_kernel clCreateKernel (cl_program program,    // Program s kernelem
                          const char *kernel_name,    // Název kernelu
                          cl_int *errcode_ret)
```

Konkrétní ukázka tvorby kernelu:

```
size_t src_size = 0;
const char* path = shrFindFilePath(cSourceFile, NULL);
const char* source = oclLoadProgSource(path, "", &src_size);
cl_program program = clCreateProgramWithSource(context, 1, &source,
&src_size, &error);

//Kompilace
error = clBuildProgram(program, 1, &device, NULL, NULL, NULL);
if( error != CL_SUCCESS )
{
    cout<<"Error occured while compiling.";
    exit(error);
}

//Vytvoř Log
char* build_log;
size_t log_size;

//První volání pro zjištění velikosti Logu
clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, 0, NULL,
&log_size);
build_log = new char[log_size+1];

//Druhé volání pro získání Logu
clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, log_size,
build_log, NULL);
build_log[log_size] = '\0';
cout<<build_log<<endl;
delete[] build_log;

//Extrahování kernelu
cl_kernel PSO_kernel = clCreateKernel(program, "PSO_GPU",&error);
if( error != CL_SUCCESS )
{
    cout<<"Error occured while creating kernel."<<endl;
    exit(error);
}
```

### 6.1.4 Spuštění kernelu

Před spuštěním kernelu, musíme nastavit jeho argumenty:

```
cl_int  clSetKernelArg (cl_kernel kernel,    // Kernel
                        cl_uint arg_index,    // Argument
                        size_t arg_size,      // Velikost argumentu
                        const void *arg_value) // Hodnota
```

Tuto funkci voláme pro každý argument. Po dokončení tohoto kroku, můžeme zavolat kernel pomocí:



```

cl_int  clEnqueueNDRangeKernel (cl_command_queue command_queue,
                                cl_kernel kernel,
                                cl_uint  work_dim,      // Rozměr problému
                                const size_t *global_work_offset,
                                const size_t *global_work_size,  // Počet
//work-items
                                const size_t *local_work_size,    // počet
//work-items ve work-group
                                cl_uint num_events_in_wait_list,
                                const cl_event *event_wait_list,
                                cl_event *event)

```

Konkrétní ukázka spuštění kernelu:

```

//Nastavení argumentů kernelu
error = clSetKernelArg(PSO_kernel, 0, sizeof(cl_mem),
&population_device);
if( error != CL_SUCCESS )
{
    cout<<"Error occured while enquing parameters.";
    exit(error);
}

//Spuštění kernelu
const size_t local_ws = local_work_size;
const size_t global_ws = shrRoundUp(local_ws, pop_size_host);
error = clEnqueueNDRangeKernel(queue, PSO_kernel, 1, NULL, &global_ws,
&local_ws, 0, NULL, NULL);
if( error != CL_SUCCESS )
{
    cout<<"Error occured while launching kernel.";
    system("PAUSE");
    exit(error);
}

```

### 6.1.5 Ukončení

Analogicky k zápisu do paměti, paměť přečteme, abychom získali výsledná data.

```

cl_int  clEnqueueReadBuffer (cl_command_queue command_queue,
                             cl_mem buffer,    // Identifikace bufferu
                             cl_bool blocking_read,  // Blokující/neblokující
//čtení
                             size_t offset,    // Offset od začátku
                             size_t cb,      // Velikost dat, které se budou číst
// (v bytech)
                             void *ptr,      // Ukazatel na paměť počítače
                             cl_uint num_events_in_wait_list,
                             const cl_event *event_wait_list,
                             cl_event *event)

```

Po dokončení výpočtů uvolníme paměť pomocí *clRelease*.

### Konkrétní ukázka ukončení

```
//Čtení výsledků
float* results = new float[data_size];
clEnqueueReadBuffer(queue, res_population_device, CL_TRUE, 0,
mem_size_population, results, 0, NULL, NULL);

//Uvolnění paměti
delete[] population_host;

clReleaseKernel(PSO_kernel);
clReleaseCommandQueue(queue);
clReleaseContext(context);
clReleaseMemObject(population_device);
```

## 6.2 OpenCL Kernel

Vzhledem k danému problému, jsem se rozhodl nevyužívat lokální paměti. Hlavním důvodem, který mě vedl k tomuto rozhodnutí, byl fakt, že globální paměť je natolik rychlá, že nám plně postačuje a urychlení programu pomocí lokální paměti, by bylo naprosto zanedbatelné.

Všechny algoritmy tedy pracují s populací uloženou v globální paměti a některými konstantami uloženými v paměti privátní. Konkrétní případy budou upřesněny níže.

Paralelní implementace je provedena za pomoci farmářského modelu. Inicializace populace je provedena na CPU a na GPU je každému procesu přidělen jeden člen populace. Vlákna je tedy tolik, kolik je členů populace.

### 6.2.1 Organizace dat

Pro populaci platí, že je uložena v jednorozměrném poli. Předpokládejme, že dimenze problému je  $dim$  a velikost populace je  $pop\_size$ . Pole *population* o velikosti  $pop\_size * dim$  je použito pro uložení všech pozic celé populace. Člen populace  $Y$  s indexem  $(i, d)$ , kde  $i$  značí index člena populace a  $d$  dimenzi, ke které přistupujeme, koresponduje s prvkem pole *population* o indexu  $(i * dim + d)$ . Tato metoda mapování adres může být formulována následovně:

$$Y(i, d) = population(i * dim + d) \quad (21)$$

, kde  $Y(i, d)$  značí data  $d$ -té dimenze,  $i$ -té částice.

Pro zpřehlednění kódu je zavedena proměnná `particle_id`, která je rovna součinu rozměru problému s identifikátorem dané `work_item` a slouží jako identifikátor částice. Získáme ho pomocí následujícího kódu:

```
//index částice  
int particle_id = get_global_id(0)*dim;
```

, kde metoda `get_global_id(0)` vrací id aktuální work-item.

## 6.2.2 Společné funkce

### 6.2.2.1 Generátor pseudonáhodných čísel

Protože grafická karta neobsahuje dobré zdroje náhodnosti, OpenCL nepodporuje žádný implementovaný generátor náhodných, či pseudonáhodných, čísel. Momentálně existují dvě možnosti, jak použít pseudonáhodná čísla při běhu programu. První možností je vygenerovat pole pseudonáhodných čísel, které se předá jako parametr funkce kernelu. Druhou možností je napsat vlastní funkci na generování pseudonáhodných čísel. Tato funkce ovšem bude opět potřebovat náhodné čísla, generované procesorem, ke své inicializaci.

## 6.2.3 Společné proměnné

### Globální

- `population` ( $\vec{x}$ ) - Pole uchovávající pozice všech jedinců.
- `variables` - Pole, který přenáší parametry nutné pro běh algoritmu. Liší se pro každý algoritmus. Definováno v API jednotlivých algoritmů.
- `boundaries` - Pole, o velikosti  $2 * \text{pop\_size}$ , obsahující omezení populace. Levá část vektoru obsahuje omezení shora, pravá část obsahuje omezení zespodu. K pravé části se přistupuje pomocí offsetu o hodnotě `dim`.
- `fitness` - Pole fitness hodnot.

### Privátní

- `dim` - Proměnná určující dimenzi problému.
- `pop_size` - Proměnná určující velikost populace.
- `iteration` - Čítač iterací.

- max\_iterations - Ukončující kritérium.

## 6.2.4 SOMA\_kernel

### 6.2.4.1 Popis proměnných

#### Globální

- PRTVector - Pole uchovávající hodnoty PRTVectoru.
- leader\_position ( $x_L$ ) - Pozice nejlepšího jedince.
- temp\_individual - Prostor pro dočasné uložení jedince během výpočtů.
- index\_sort - Pomocné pole indexů, použité při paralelní redukci. Obsahuje celočíselné hodnoty 0 - pop\_size. Inicializováno před každou iterací.

#### Privátní

- path\_length - Maximální vzdálenost jakou částice urazí.
- step - Krok, se kterým se částice pohybuje.
- prt - Parametr sloužící pro uložení hodnoty perturbace.

### 6.2.4.2 Popis kódu

```
inicializuj všechny proměnné

//sub-procesy ve smyčce jsou prováděny paralelně
for iteration to max_iteration do
    spočítej fitness hodnoty všech částic
    najdi Leadera
    if index work_item se nerovná index Leadera then
        posuň populaci směrem k Leaderovi
    end if
end for
```

- Nalezení Leadera

V této části algoritmu narážíme na jeden z problémů paralelní implementace. Při hledání nejmenšího prvku v poli, nemůžeme, kvůli paralelnímu přístupu do paměti, využít klasických metod. V našem případě jsem problém vyřešil pomocí paralelní redukce, která může vypadat následovně:

```

for(int offset = pop_size / 2; offset > 0; offset = offset / 2)
{
    if (g_id < offset)
    {
        int left_index = index_sort[g_id];
        int right_index = index_sort[g_id + offset];
        index_sort[g_id] = ( fitness[left_index] < fitness[right_index]
) ? left_index : right_index;
    }
}

```

Při paralelní redukci se využívá poloviny vláken, která porovnají levou polovinu pole fitness s pravou a v poli index\_sort uloží, do levé poloviny, indexy lepších prvků. V dalším kroku se levá polovina opět rozdělí a proces se opakuje, dokud se do pole index\_sort, na pozici s indexem 0, nepřesune index nejlepšího prvku.

- Posun populace

Aplikujeme vztah (1) na populaci. Pokud se jedinec posune do výhodnější pozice, tak si pozici zapamatuje. Kód vypadá následovně:

```

for( float t = 0; t < path_length ; t+=step )
{
    for(int d = 0; d < dim; d++)
    {
        temp_individual[d] = population[particle_id + d];
        temp_individual[d] += (index_sort[0]*dim
- population[particle_id + d])*PRTVector[particle_id + d]*t;
    }

    //Pokud je nová pozice zlepšení, tak ji ulož
    float temp = cost_function(temp_space, dim, t_fce);
    if( temp < fitness[g_id] )
    {
        fitness[g_id] = temp;
        for(int d = 0; d < dim; d++)
        {
            population[particle_id + d] = temp_individual[d];
        }
    }
}

```

## 6.2.5 Popis PSO\_kernel

### 6.2.5.1 Popis proměnných

### Globální

- pBest\_position - Pole nejlepších dosažených pozic, každého jedince.
- pBest\_fitness - Pole nejlepších dosažených fitness, každého jedince.
- velocity - Pole rychlostí.
- gBest\_position - Pole uchovávající pozici dosavadního globálního minima.
- index\_sort - Pomocné pole indexů, použité při paralelní redukci. Obsahuje celočíselné hodnoty 0 - pop\_size. Inicializováno před každou iterací.

### Privátní

- c1,c2 - Proměnná uchovávající učící faktory.
- gBest - Fitness hodnota dosavadního globálního minima.
- w - Setrvačnost.

#### 6.2.5.2 Popis kódu

```
inicializuj všechny proměnné

//sub-procesy ve smyčce jsou prováděny paralelně
for iteration to max_iteration do
    spočítej fitness hodnoty všech částic
    aktualizuj pBest všech částic
    aktualizuj gBest
    aktualizuj rychlost a pozici všech částic
end for
```

- Aktualizace pBest

Po dokončení výpočtu fitness, by se nové hodnoty měly porovnat s dosavadními *pBest*, protože částice mohla, v předchozím kroku, najít nové lokální minimum. Aktualizace *pBest* se provede následovně:

```
namapuj work-items na N částic jedna ku jedné

//proved' synchronně operace na částicích i (i = 1,...,N)
if fitness(i) je lepší než pBest_fitness(i) then
    pBest_fitness(i) = fitness(i)
    for each dimension d do
        Ulož pozici population(i*N + d) do pBest_position(i*N + d)
    end for
end if
```

, kde fitness je vektor hodnot fitness, pBest je vektor nejlepších dosavadních fitness jednotlivých členů a pBest\_position je vektor pozic pBest o velikosti  $N * d$ .

V samotném kernelu bude kód vypadat následovně:

```
//porovnej fitness s pBest
if( fitness[particle_id] < pBest_fitness[particle_id] )
{
    pBest_fitness[particle_id] = fitness[particle_id];
    for(int d = 0; d<dim; d++) pBest_positon[particle_id + d] =
population[particle_id + d];
}
```

- Aktualizace gBest

Zde při porovnávání s gBest dochází k problému s paralelním přístupem do paměti. Tato část programu je řešena stejným způsobem, jako hledání Leadera v algoritmu SOMA.

- Aktualizace rychlosti a pozice

Po dokončení aktualizace hodnot *pBest* a *gBest*. Je nutné aktualizovat rychlost a pozici částice. Rychlost je aktualizována paralelně. Tento proces probíhá pro každou dimenzi.

```
for(int d = 0; d<dim;d++)
{
    //vypočítej novou rychlost
    velocity[particle_id + d] *= w;
    velocity[particle_id + d] += c1*random_01(&r)*
(pBest_positon[particle_id + d] - population[particle_id + d]);
    velocity[particle_id + d] += c1*random_01(&r)*
(gBest_position[d] - population[particle_id + d]);

    //aktualizuj pozici
    population[particle_id + d] += velocity[particle_id + d];
}
```

### 6.2.6 Popis DE\_kernel

DE jako jediný z algoritmů nevyužívá nejlepšího jedince pro výpočet nových pozic. Díky této vlastnosti není nutné využívat paralelní redukce, jak tomu bylo v předchozích dvou případech, a paralelní implementace se příliš neliší od implementace sériové.

#### 6.2.6.1 Popis proměnných

### Globální

- temp\_space - Pole sloužící jako prostor pro dočasné výpočty.

### Privátní

- F - Mutační konstanta.
- CR - Práh křížení.
- p1,p2,p3 - Indexy tří rodičů podílejících se na tvorbě nového člena populace.

#### 6.2.6.2 Popis kódu

```
inicializuj všechny proměnné

//sub-procesy ve smyčce jsou prováděny paralelně
for iteration to max_iteration do
    spočítej fitness hodnoty všech částic
    vyber rodiče
    vypočítej nového člena populace
    porovnej s původní pozicí
end for
```

- Výpočet nového člena populace

Výpočet probíhá v poli temp\_space. Kvůli náhodnému a nerovnoměrnému přístupu do pole population dochází k "deparalelizaci" kódu. Jedná se ovšem o natolik nenáročné výpočty, že se neprojeví na konečném výkonu.

```
for(int d = 0; d < dim; d++ )
{
    //diferenční vektor
    temp_space[particle_id + d] = population[p1*dim + d] -
    population[p2*dim + d];
    //mutace
    temp_space[particle_id + d] *= F;
    //šumový vektor
    temp_space[particle_id + d] += population[p3*dim + d];
    //zkušební vektor
    if( random_01(&r) > CR ) //random_01(&) vrací float v rozsahu 0-1
    {
        temp_space[particle_id + d] = population[particle_id + d];
    }
}
```



- Porovnání s původní pozicí

Zjistíme fitness hodnotu nového člena populace. V případě lepší hodnoty si pozici uložíme.

```
float temp = cost_function(temp_space, dim, t_fce);
if( temp < fitness[g_id] )
{
    fitness[g_id] = temp;
    for(int d = 0; d < dim; d++)    population[particle_id + d] =
temp_space[particle_id + d];
}
```

## 7 SROVNÁNÍ VÝKONU

### 7.1 Popis testovacích funkcí

Pro testování algoritmů, jsem použil množinu testovacích funkcí, která se skládá z funkcí s různými vlastnostmi. Tyto funkce se liší složitostí, použitím goniometrických funkcí a mocnin.

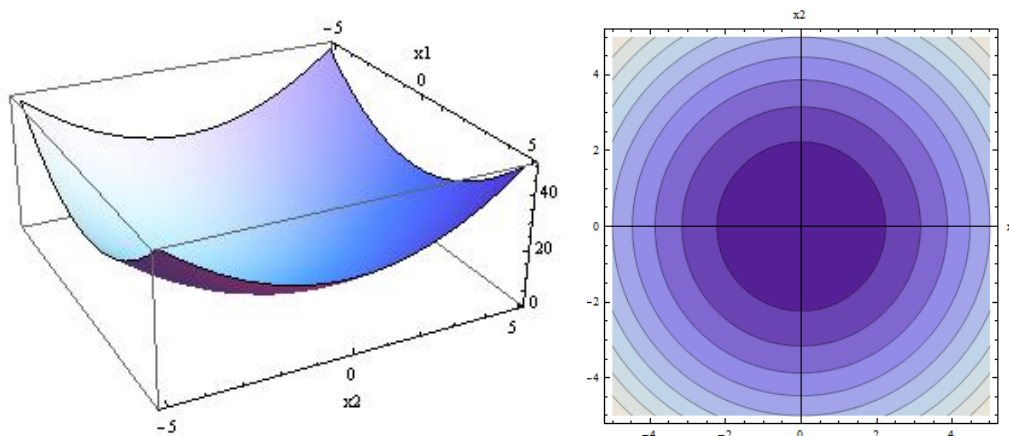
Funkce popisují pomocí analytického zápisu, jejího vzhledu v trojrozměrném prostředí a pomocí vrstevnicového grafu. Pro zobrazení jsem využil programu Wolfram Mathematica.

#### 7.1.1 První de Jongova funkce

- Analytický popis

$$\sum_{i=1}^D x_i^2 \quad (17)$$

- Grafické zobrazení



Obr. 11 Grafické zobrazení první de Jongovy funkce.

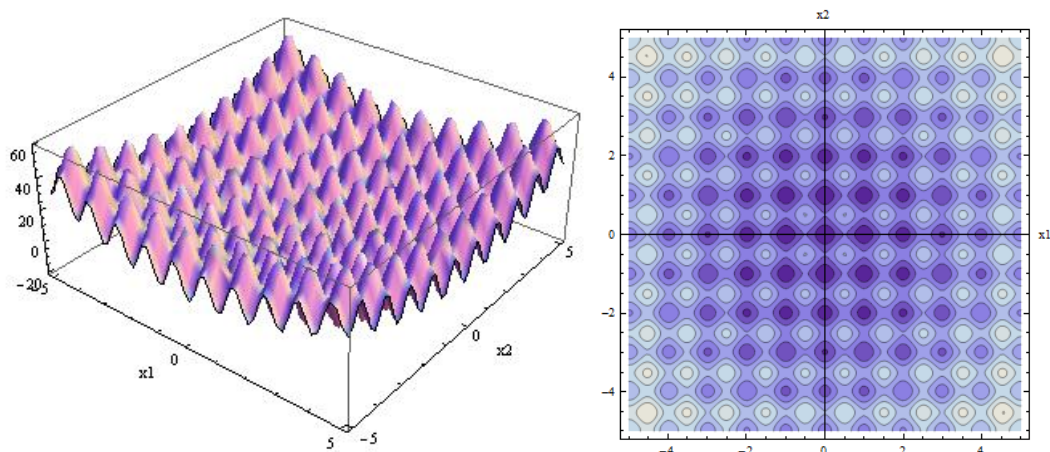
- Globální minimum v  $E_n$ 
  - Pozice -  $(x_1, x_2, \dots, x_n) = (0, 0, \dots, 0)$
  - Hodnota -  $y = 0$

#### 7.1.2 Rastriginova funkce

- Analytický popis

$$\sum_{i=1}^D [x_i^2 - 10 \cos(2\pi x_i) + 10] \quad (18)$$

- Grafické zobrazení



Obr. 12 Grafické zobrazení Rastriginovy funkce

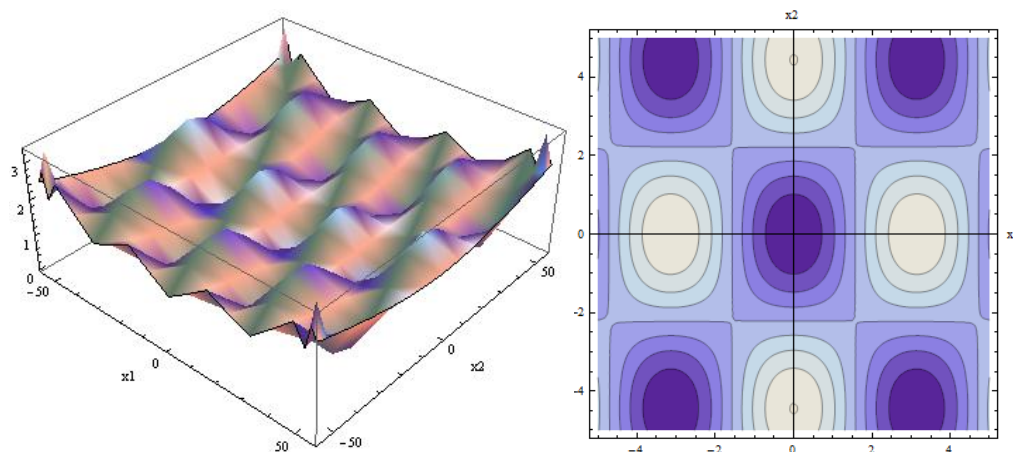
- Globální minimum v  $E_n$ 
  - Pozice -  $(x_1, x_2, \dots, x_n) = (0, 0, \dots, 0)$
  - Hodnota -  $y = 0$

### 7.1.3 Griewangkova funkce

- Analytický popis

$$1 + \sum_{i=1}^D \frac{x_i^2}{4000} - \prod_{i=1}^D \cos\left(\frac{x_i}{\sqrt{i}}\right) \quad (19)$$

- Grafické zobrazení



Obr. 13 Grafické zobrazení Griewankovy funkce.

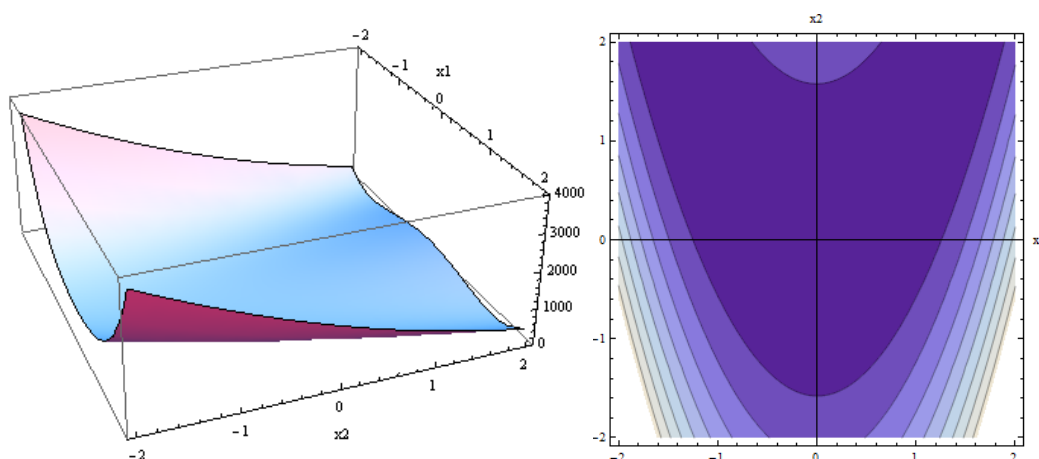
- Globální minimum v  $E_n$ 
  - Pozice -  $(x_1, x_2, \dots, x_n) = (0, 0, \dots, 0)$
  - Hodnota -  $y = 0 \times n = 0$

#### 7.1.4 Rosenbrokovo sedlo

- Analytický popis

$$\sum_{i=1}^{D-1} (100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2) \quad (20)$$

- Grafické zobrazení



Obr. 14 Grafické zobrazení Rosenbrokova sedla.

- Globální minimum v  $E_n$

- Pozice -  $(x_1, x_2, \dots, x_n) = (0, 0, \dots, 0)$
- Hodnota -  $y = 0 \times n = 0$

## 7.2 Parametry měření

Porovnání výkonu mezi GPU a PCU je založeno na čtyřech klasických testovacích funkcích f1, f2, f3 a f4, které jsou popsány výše. Optimální řešení všech funkcí se nachází v nule.

Měření probíhalo na následující sestavě:

- CPU: AMD Phenom II X4 3.4GHz
- RAM: 6.0GB
- GPU: GeForce GTX560 Ti
- OS: Windows 7 Ultimate

V rámci měření definujeme *zrychlení* jako hodnotu, určující kolikrát je, algoritmus spuštěný na grafické kartě, rychlejší, než algoritmus spuštěný na procesoru. *Zrychlení* můžeme vyjádřit následovně:

$$\gamma = \frac{T_{CPU}}{T_{GPU}} \quad (21)$$

, kde  $\gamma$  je *zrychlení*,  $T_{CPU}$  a  $T_{GPU}$ , je doba, kterou procesor, nebo grafická karta, potřebuje k optimalizaci problému, během specifického počtu iterací.

Tab. 7 Funkce použité pro měření výkonu implementace a definované hranice.

Značení	Název funkce	Hranice
<b>f1</b>	První de Jongova funkce	$(-10,10)^D$
<b>f2</b>	Rastriginova funkce	$(-10,10)^D$
<b>f3</b>	Griewangkova funkce	$(-10,10)^D$
<b>f4</b>	Rozenbrokovo sedlo	$(-10,10)^D$

Algoritmy byly měřeny pro populaci postupně se zvětšující ze 400 členů na 2800 členů. Populace je vytvořena s dimenzí 100 a prochází 2000 iteracemi.

Pro takto definovaný problém byl měřen výpočetní čas jednotlivých algoritmů, postupně zpracovávajících testovací funkce f1, f2, f3, f4. Hodnoty v následujících tabulkách udávají průměrné hodnoty z 15 měření.

### 7.3 Naměřené hodnoty

#### 7.3.1 PSO

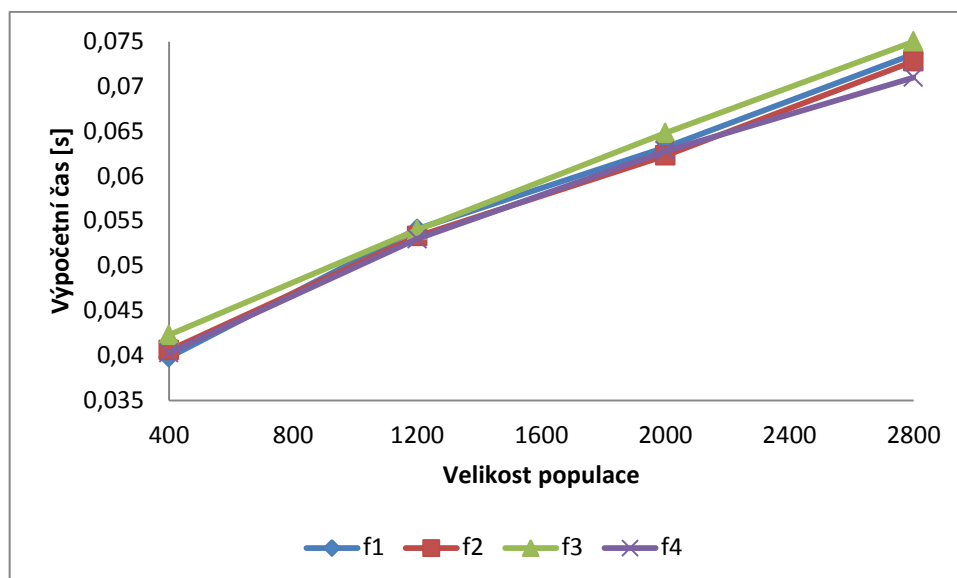
Tab. 8 Hodnoty parametrů algoritmu PSO použité pro měření.

Parametr	Nastavená hodnota
<b>c1,c2</b>	2,05
<b>W<sub>start</sub></b>	0,9
<b>W<sub>end</sub></b>	0,4

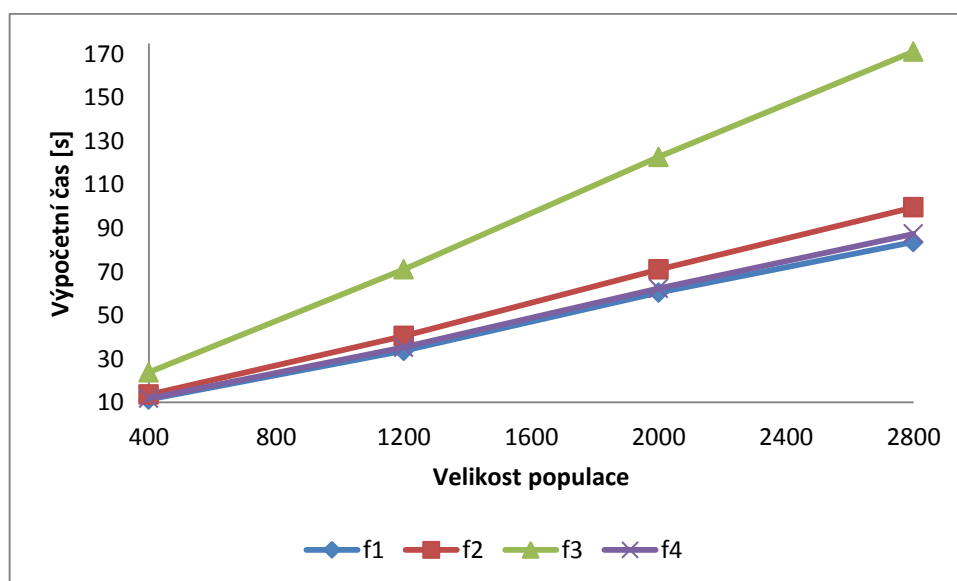
- Naměřená data

Tab. 9 Naměřené výpočetní časy algoritmu PSO pro GPU a CPU.

Velikost populace	Čas[s]							
	GPU				CPU			
	f1	f2	f3	f4	f1	f2	f3	f4
<b>400</b>	0,0398	0,0406	0,0423	0,0403	11,29	13,558	23,757	11,755
<b>1200</b>	0,0541	0,0533	0,054	0,053	33,659	40,508	71,116	35,243
<b>2000</b>	0,0632	0,0623	0,0648	0,0628	60,408	71,116	122,809	62,297
<b>2800</b>	0,0736	0,0728	0,075	0,071	83,666	99,596	171,18	87,422



Obr. 15 Závislost výpočetní doby algoritmu PSO na velikosti populace pro GPU.

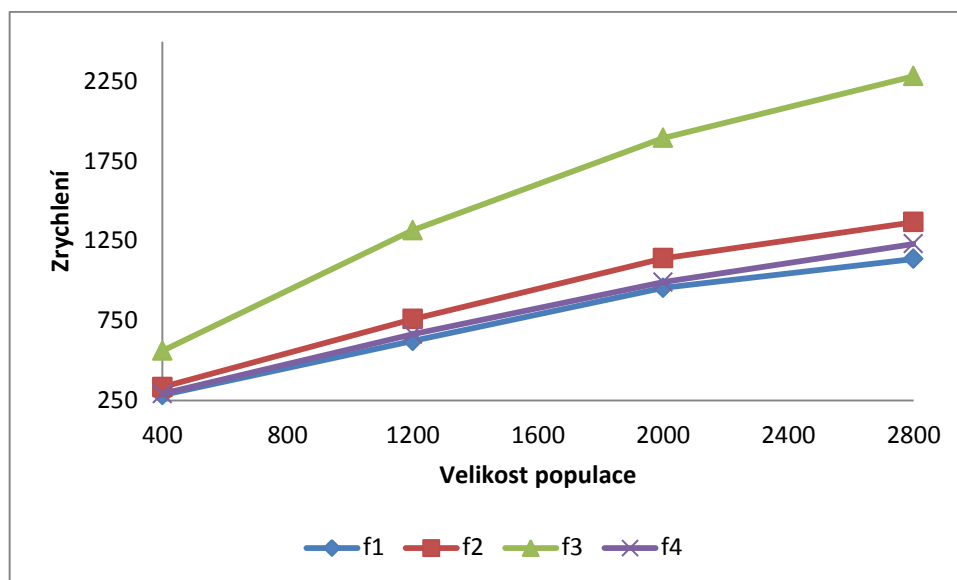


Obr. 16 Závislost výpočetní doby algoritmu PSO na velikosti populace pro CPU.

- Naměřené zrychlení

Tab. 10 Naměřené hodnoty zrychlení algoritmu PSO.

Velikost populace	Zrychlení			
	f1	f2	f3	f4
400	283,6683	333,9409	561,6312	291,6873
1200	622,1627	760	1316,963	664,9623
2000	955,8228	1141,509	1895,201	991,9904
2800	1136,766	1368,077	2282,4	1231,296



Obr. 17 Závislost hodnoty zrychlení algoritmu PSO na velikosti populace.

### 7.3.2 SOMA

Tab. 11 Hodnoty parametrů algoritmu SOMA použité pro měření.

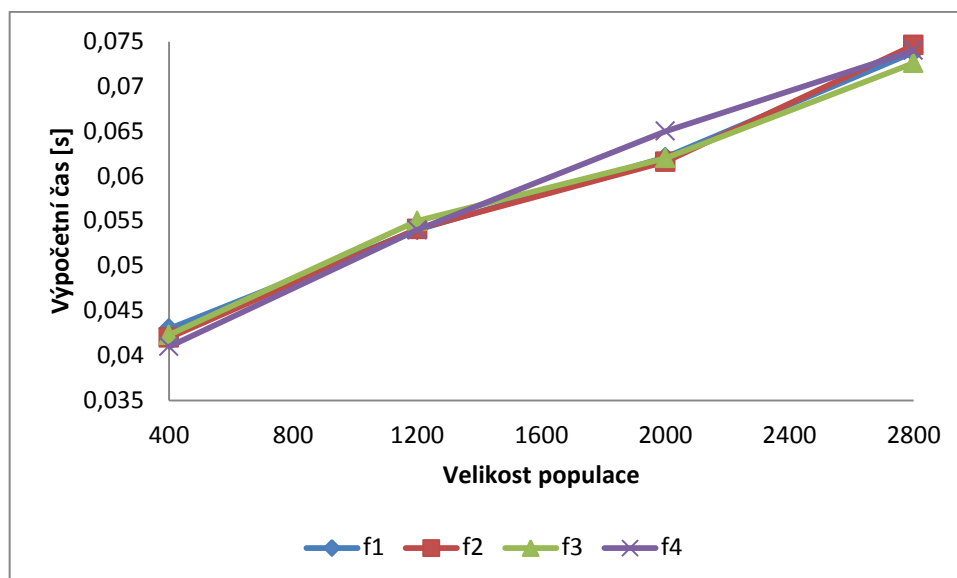
Parametr	Nastavená hodnota
Mass	2,0
Step	0,11
PRT	0,1

- Naměřená data

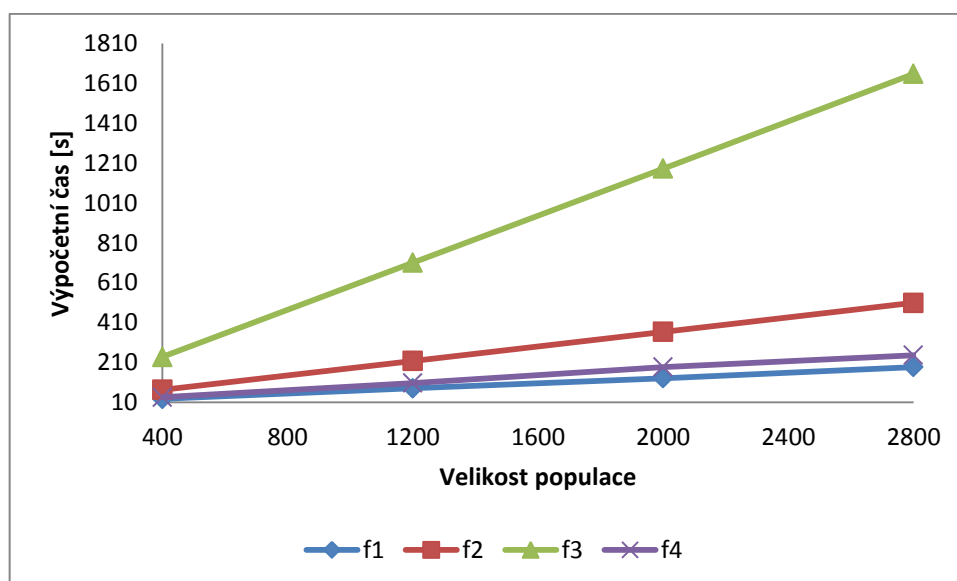
Tab. 12 Naměřené výpočetní časy algoritmu SOMA pro GPU a CPU.

Velikost populace	Čas[s]							
	GPU				CPU			
	f1	f2	f3	f4	f1	f2	f3	f4
400	0,043	0,042	0,0423	0,041	26,62	72,452	236,945	35,421
1200	0,054	0,0541	0,055	0,054	79,609	217,472	710,137	106,105
2000	0,0621	0,0616	0,062	0,065	130,814	363,073	1181,92	186,566
2800	0,0738	0,0746	0,0726	0,074	186,395	508,237	1656,83	246,243





Obr. 18 Závislost výpočetní doby algoritmu SOMA na velikosti populace pro GPU.

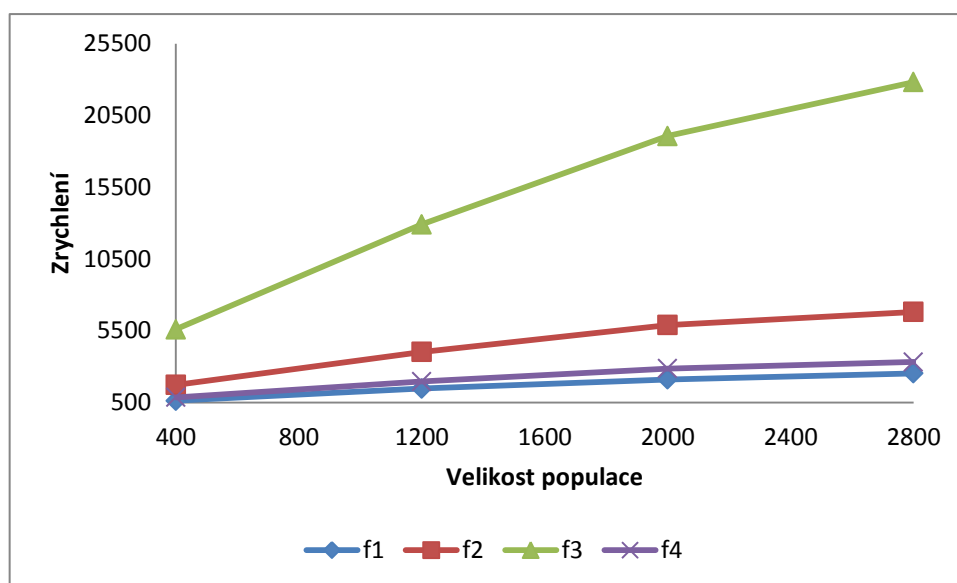


Obr. 19 Závislost výpočetní doby algoritmu SOMA na velikosti populace pro CPU.

- Naměřené zrychlení

Tab. 13 Naměřené hodnoty zrychlení algoritmu SOMA.

Velikost populace	Zrychlení			
	f1	f2	f3	f4
400	619,0698	1725,048	5601,537	863,9268
1200	1474,241	4019,815	12911,58	1964,907
2000	2106,506	5894,042	19063,23	2870,246
2800	2525,678	6812,828	22821,35	3327,608



Obr. 20 Závislost hodnoty zrychlení algoritmu SOMA na velikosti populace.

### 7.3.3 DE

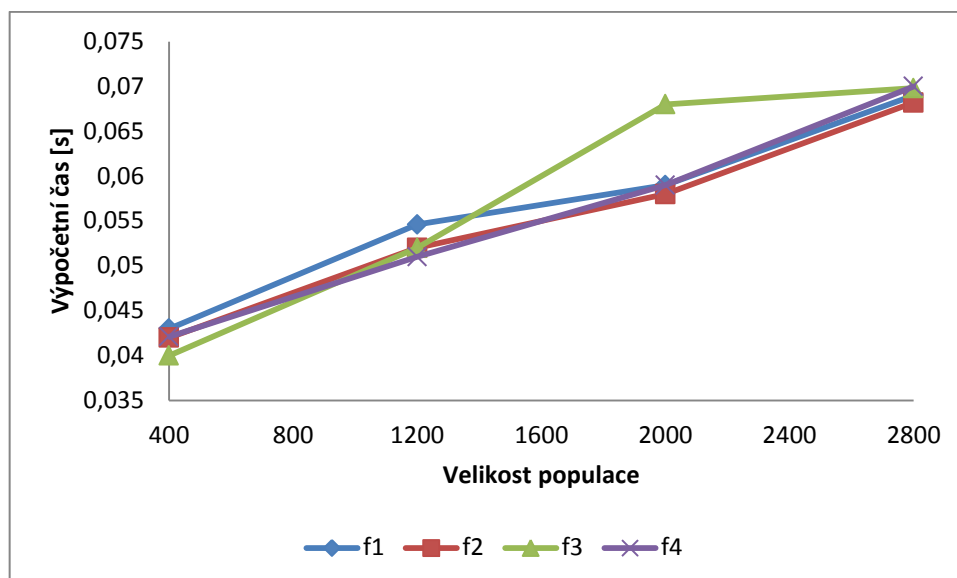
Tab. 14 Hodnoty parametrů algoritmu DE použité pro měření.

Parametr	Nastavená hodnota
<b>F</b>	0,3
<b>CR</b>	0,8

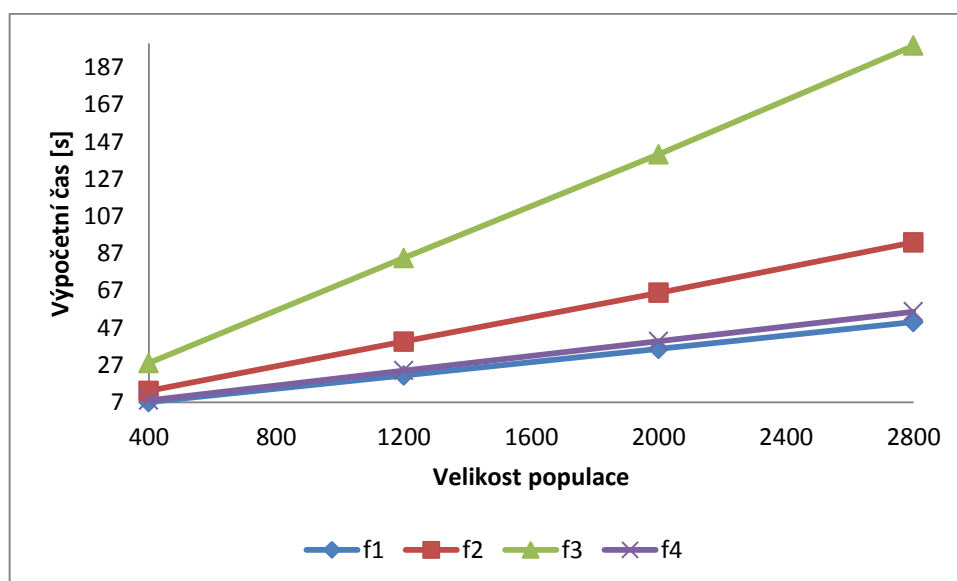
- Naměřená data

Tab. 15 Naměřené výpočetní časy algoritmu DE pro GPU a CPU.

Velikost populace	Čas[s]							
	GPU				CPU			
	f1	f2	f3	f4	f1	f2	f3	f4
<b>400</b>	0,043	0,042	0,04	0,0421	7,14	13,155	28,119	8,007
<b>1200</b>	0,0546	0,052	0,052	0,051	21,355	39,629	84,526	24,042
<b>2000</b>	0,059	0,058	0,068	0,059	35,643	65,856	140,349	39,838
<b>2800</b>	0,069	0,0682	0,0698	0,07	50,123	92,982	198,734	55,723



Obr. 21 Závislost výpočetní doby algoritmu DE na velikosti populace pro GPU.

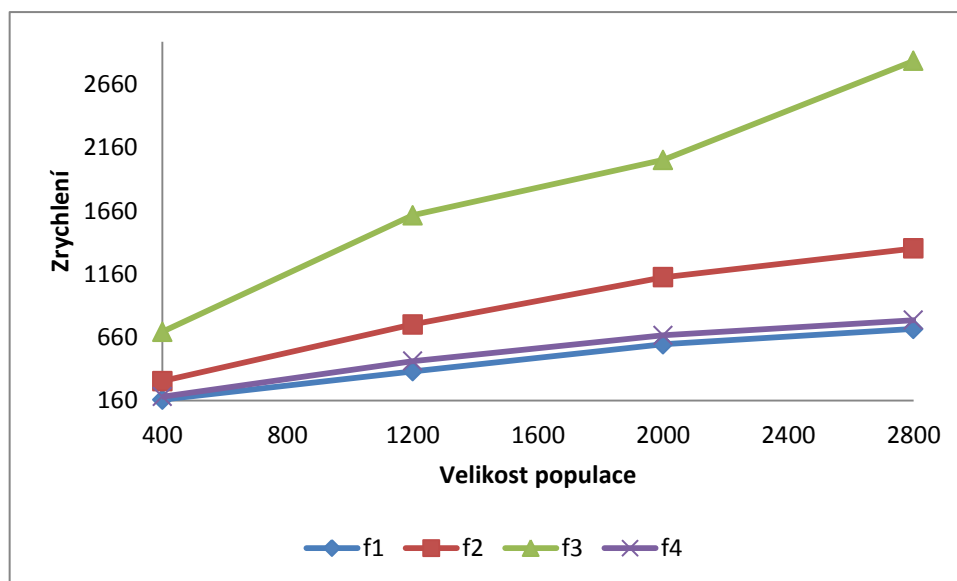


Obr. 22 Závislost výpočetní doby algoritmu DE na velikosti populace pro CPU.

- Naměřené zrychlení

Tab. 16 Naměřené hodnoty zrychlení algoritmu DE.

Velikost populace	Zrychlení			
	f1	f2	f3	f4
400	166,0465	313,2143	702,975	190,19
1200	391,1172	762,0962	1625,5	471,4118
2000	604,1186	1135,448	2063,956	675,2203
2800	726,4203	1363,372	2847,192	796,0429



Obr. 23 Závislost hodnoty zrychlení algoritmu DE na velikosti populace.

## 7.4 Zhodnocení naměřených dat

### 7.4.1 Závislost na testovací funkci

- GPU

Z grafů lze vidět, že výpočet na grafické kartě je nezávislý na použité testovací funkci. Grafická karta zpracovává testovací funkce s natolik velkou rychlostí, že pro námi definované měření se složitost testovací funkce neprojeví.

Zde se projevuje síla grafických karet při řešení komplexních aritmetických funkcí.

- CPU

Na rozdíl od grafické karty, CPU nedokáže řešit komplexní aritmetické funkce s takovou rychlostí, a z grafů je patrná velká závislost na testovací funkci. Závislost na testovací funkci je nejvíce patrná na algoritmu DE, kde rozdíl mezi funkcemi, pro maximální populaci, činí až 150s.

Při odečítání výsledků algoritmu SOMA, musíme brát ohled na fakt, že se účelová funkce volá přibližně 20x častěji, než u ostatních algoritmů. Proto není příliš vhodné srovnávat výkon algoritmu SOMA, pouze na základě doby, za kterou dokončí určený počet iterací.

### 7.4.2 Závislost na velikosti populace

- GPU

Z měření vyplývá, že za použití paralelního farmářského modelu, s rostoucí populací lineárně stoupá doba výpočtu.

- CPU

Opět je zde viditelná závislost na testovací funkci. Nejstrmější stoupání vidíme pro funkci f3.

### 7.4.3 Srovnání výkonu CPU a GPU

Z naměřených výsledků lze vyčíst, že zrychlení výpočtů na GPU, roste, oproti výpočtům na CPU, se složitostí testovací funkce a s rostoucí populací. Toto zrychlení se pohybuje řádově ve stovkách, pro jednodušší problémy, až tisících, pro složitější problémy. Lze očekávat, že od určité hranice se zrychlení ustálí, a stane se konstantní. Bohužel k omezeným zdrojům CPU a velmi výkonné grafické kartě, se mi tuto hranici nepodařilo nalézt.

## ZÁVĚR

V rámci této práce se mi podařilo implementovat vybrané evoluční algoritmy pomocí standardu OpenCL. Vytvořené třídy lze jednoduše zahrnout do OpenCL projektu a využít pro další testování. V budoucnu by bylo vhodné, pro algoritmy SOMA a DE, doplnit více strategií a do všech algoritmů zahrnout testování více ukončovacích kritérií.

Měření výkonu aplikací, které bylo důležitým bodem praktické části práce, silně ovlivnila nevyvážená sestava, kterou jsem měl k dispozici a také fakt, že jsem nemohl aplikace testovat na více grafických kartách. Ale, i když přihlédneme k hardwarovým vlastnostem systému, můžeme bezpečně vyvodit několik závěrů. Aplikace napsané pomocí OpenCL, dosáhly zrychlení řádově v tisících. Nejvíce z tohoto faktu mohou čerpat vysoce náročné reálné aplikace, jako systémy rozpoznávání obličeje, či rozpoznávání otisků prstů. Dále mohu říct, že se grafické karty dostaly do stádia, kdy je nevýhodné je porovnávat OpenCL aplikace s jednoprocessorovými aplikacemi. Rozdíl mezi výpočetními časy je natolik markantní, že pokud nastavíme podmínky měření výhodné pro procesor, grafické karty budou pracovat natolik rychle, že bude nemožné přesně měřit výpočetní čas. A naopak pokud nastavíme experiment tak, abychom plně využili potenciálu, který grafická karta představuje, čas, který procesor stráví s výpočtem, bude nepříjemně velký.

Pokud bych měl někdy znovu porovnávat výkon aplikace běžící na GPU oproti aplikaci běžící na CPU, bylo by lepší, abych porovnával dvě rovnocenné paralelní implementace. Pomocí OpenCL na GPU a například pomocí rozhraní OpenMP na vícejádrovém procesoru.

I přes všechna negativa si ale algoritmy v testech vedly překvapivě dobře a díky svým vlastnostem mohou být aplikovány na širokou škálu praktických optimalizačních problémů.

## ZÁVĚR V ANGLIČTINĚ

In this thesis, I managed to implement chosen evolutionary algorithms using OpenCL standard. Created classes are easily implemented into the OpenCL project and used for further testing. In future, it would be adequate to supply more strategies for SOMA and DE algorithms and include more stopping criterion testing for all algorithms.

Measurement of the performance of applications, which was an important point of the practical part of a thesis, was strongly influenced by an unbalanced system, that was available to me and the fact, that I couldn't test on multiple graphics cards. But, even if we account hardware properties of the system, we can safely draw some conclusions. Applications achieved speedup in order of thousands. From this fact can mostly gain high dimensional real word applications such as face recognition or finger print recognition. Also I think I can say that graphics cards reached state, when it is disadvantageous to compare OpenCL applications to single processor applications. The difference between compute times is so striking, that if we set conditions of experiment advantageous for the processor, graphics cards will run so fast, it would be impossible to measure runtime precisely. On the other hand, if set conditions of an experiment in a way, that fully uses the potential, which graphics cards represent, time, which a processor spends on computing, would be unacceptably big.

If I am ever again to compare a performance of the application run on GPU against the application run on CPU, it would be better to compare to equal parallel implementations. Using the OpenCL on a GPU and for example using the OpenMP interface on the multi core processor.

Despite of all negatives, all algorithms performed exceptionally well and thanks to their properties, can be applied to wide scale of real word optimization problems.

**SEZNAM POUŽITÉ LITERATURY**

- [1] ČERVENKA, Miroslav. *Distributed Evolutionary Algorithms*. Zlín, 2006. Doctoral Thesis. Tomas Bata University in Zlín. Vedoucí práce doc. Ing. Ivan Zelinka, Ph. D.
- [2] KUMAR TATI, Kiran. *General purpose evolutionary algorithm testbed*. Missouri, 2009. Thesis. Faculty of the Graduate School at University of Missouri. Vedoucí práce Dr. Tina Smilkstein.
- [3] KRAMPL, Jakub. *Implementace vybraných evolučních algoritmů v prostředí .NET*. Zlín, 2011. Bakalářská práce. Universita Tomáše Baťi ve Zlíně. Vedoucí práce Ing. Erik Král.
- [4] ZIELINSKI, Karin, Dagmar PETERS a Rainer LAUR. UNIVERSITY OF BREMEN. *Stopping Criteria for Single-Objective Optimization*. Bremen, Germany, 2009.
- [5] *Evoluční výpočetní techniky: principy a aplikace*. 1. české vyd. Praha: BEN, 2009, 534 s. ISBN 978-80-7300-218-3.
- [6] ZELINKA, Ivan. *Umělá inteligence v problémech globální optimalizace*. 1. vyd. Praha: BEN - technická literatura, 2002, 189 s. ISBN 80-730-0069-5.
- [7] QING, Anyong. *Differential evolution fundamentals and applications in electrical engineering: Fundamentals and Applications in Electrical Engineering*. Singapore: J. Wiley, 2009. ISBN 04-708-2393-3.
- [8] HU, Xiaohui. Particle Swarm Optimization: Tutorial. [online]. 2006 [cit. 2012-05-27]. Dostupné z: <http://www.swarmintelligence.org/>
- [9] KENNEDY, James a Daniel BRATTON. *2007 IEEE Swarm Intelligence Symposium: Honolulu, HI, 1-5 April 2007*. Piscataway, NJ: IEEE, c2007, ix, 374 p. ISBN 14-244-0708-7.
- [10] Evolutionary Algorithms: Overview, Methods and Operators. In: *Genetic and Evolutionary Algorithm Toolbox for Matlab* [online]. 2006 [cit. 2012-05-29]. Dostupné z: [geatbx.com](http://geatbx.com)
- [11] NEDJAH, Nadia, Enrique ALBA a Luiza de MACEDO MOURELLE. *Parallel evolutionary computations*. New York: Springer-Verlag, c2006, xxii, 200 p. ISBN 35-403-2837-8.



- [12] *OpenCL programming guide*. Upper Saddle River: Addison-Wesley, c2012, xliv, 603 s. ISBN 978-0-321-74964-2.
- [13] GOVE, Darryl. *Programování aplikací pro vícejádrové procesory*. Vyd. 1. Brno: Computer Press, 2011, 416 s. ISBN 978-80-251-3487-0.
- [14] OpenCL Tutorial. In: *MacResearch* [online]. 2009 [cit. 2012-05-29]. Dostupné z: <http://www.macresearch.org/opengl>
- [15] OpenCL™ and the AMD APP SDK. In: *AMD Developer Central* [online]. 2011 [cit. 2012-05-29]. Dostupné z: <http://developer.amd.com/documentation/articles/pages/OpenCL-and-the-AMD-APP-SDK.aspx>
- [16] ATI Stream Computing: OpenCL Programming guide. In: *AMD developer Central* [online]. 2010 [cit. 2012-05-29]. Dostupné z: [http://developer.amd.com/gpu\\_assets/ATI\\_Stream\\_SDK\\_OpenCL\\_Programming\\_Guide.pdf](http://developer.amd.com/gpu_assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf)
- [17] Khronos Group: OpenCL. *Open Standards for Media Authoring and Acceleration* [online]. 2012 [cit. 2012-05-29]. Dostupné z: <http://www.khronos.org/opengl/>
- [18] *Numerical recipes: the art of scientific computing*. 3rd ed. Cambridge: Cambridge University Press, 2007, xxi, 1235 s. ISBN 978-0-521-88068-8.

**SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK**

EA	Evoluční algoritmy.
SOMA	Self-Organizing Migrating Algorithm.
DE	Differential Evolutin.
PSO	Particle Swarm Optimalization
GA	Genetic Algorithms
CPU	Central Processing Unit
GPU	Graphics Processing Unit
API	Application Programming Interface
TPC	Thread Processing Cluster
SM	Streaming Multiprocessor
SP	Streaming Processor
SIMD	Single Instruction Multiple Data

**SEZNAM OBRÁZKŮ**

Obr. 1 Vývojový diagram algoritmu SOMA. ....	16
Obr. 2 Vývojový diagram algoritmu DE. ....	19
Obr. 3 Vývojový diagram algoritmu PSO. ....	23
Obr. 4 Ukázka různých paralelních implementací. ....	26
Obr. 5 Ukázka lineární struktury sousedství. [10] .....	28
Obr. 6 Ukázka 2D struktury sousedství. [10] .....	28
Obr. 7 Ukázka struktury platformního modelu. [14] .....	32
Obr. 8 Ukázka struktury paměťového prostoru. [14] .....	35
Obr. 9 Ukázka struktury TPC. [14] .....	38
Obr. 10 Ukázka přístupů k paměti pro procesory a multiprocesory. [14] .....	39
Obr. 11 Grafické zobrazení první de Jongovy funkce. ....	58
Obr. 12 Grafické zobrazení Rastriginovy funkce .....	59
Obr. 13 Grafické zobrazení Griewangkovy funkce. ....	60
Obr. 14 Grafické zobrazení Rosenbrokova sedla. ....	60
Obr. 15 Závislost výpočetní doby algoritmu PSO na velikosti populace pro GPU. ....	63
Obr. 16 Závislost výpočetní doby algoritmu PSO na velikosti populace pro CPU. ....	63
Obr. 17 Závislost hodnoty zrychlení algoritmu PSO na velikosti populace. ....	64
Obr. 18 Závislost výpočetní doby algoritmu SOMA na velikosti populace pro GPU. ....	65
Obr. 19 Závislost výpočetní doby algoritmu SOMA na velikosti populace pro CPU. ....	65
Obr. 20 Závislost hodnoty zrychlení algoritmu SOMA na velikosti populace. ....	66
Obr. 21 Závislost výpočetní doby algoritmu DE na velikosti populace pro GPU. ....	67
Obr. 22 Závislost výpočetní doby algoritmu DE na velikosti populace pro CPU. ....	67
Obr. 23 Závislost hodnoty zrychlení algoritmu DE na velikosti populace. ....	68

**SEZNAM TABULEK**

Tab. 1 Seznam parametrů a optimálních hodnot pro algoritmus SOMA. ....	17
Tab. 2 Seznam parametrů a optimálních hodnot pro algoritmus DE.....	20
Tab. 3 Seznam parametrů a optimálních hodnot pro algoritmus PSO.....	24
Tab. 4 Seznam souborů tvořících paralelní implementaci vybraných EA. ....	41
Tab. 5 Seznam a popis metod jednotlivých tříd.....	41
Tab. 6 Seznam parametrů a defaultních hodnot pro vybrané algoritmy. ....	42
Tab. 7 Funkce použité pro měření výkonu implementace a definované hranice. ....	61
Tab. 8 Hodnoty parametrů algoritmu PSO použité pro měření. ....	62
Tab. 9 Naměřené výpočetní časy algoritmu PSO pro GPU a CPU. ....	62
Tab. 10 Naměřené hodnoty zrychlení algoritmu PSO. ....	63
Tab. 11 Hodnoty parametrů algoritmu SOMA použité pro měření.....	64
Tab. 12 Naměřené výpočetní časy algoritmu SOMA pro GPU a CPU.....	64
Tab. 13 Naměřené hodnoty zrychlení algoritmu SOMA.....	65
Tab. 14 Hodnoty parametrů algoritmu DE použité pro měření. ....	66
Tab. 15 Naměřené výpočetní časy algoritmu DE pro GPU a CPU. ....	66
Tab. 16 Naměřené hodnoty zrychlení algoritmu DE.....	67

## SEZNAM PŘÍLOH

P I CD s bakalářskou prací a zdrojovými kódy.