# Možnosti paralelizace vybraných evolučních algoritmů na GPU

Bc. Martin Šůstek

Diplomová práce
2024

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2023/2024

# ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

| | |
|---|---|
| Jméno a příjmení: | **Bc. Martin Šůstek** |
| Osobní číslo: | **A21180** |
| Studijní program: | **N0613A140022 Informační technologie** |
| Specializace: | **Softwarové inženýrství** |
| Forma studia: | **Kombinovaná** |
| Téma práce: | **Možnosti paralelizace vybraných evolučních algoritmů na GPU** |
| Téma práce anglicky: | **GPU Parallelization of Selected Evolution Algorithms** |

## Zásady pro vypracování

1. Seznamte se se základními principy evolučních algoritmů.
2. Vhodným způsobem vyberte několik algoritmů a uveďte detailně jejich strukturu.
3. Prostudujte možnosti paralelizace algoritmů s využitím CUDA a OpenACC.
4. Identifikujte kritické části evolučních algoritmů s možností paralelizace.
5. Navrhněte způsob paralelizace těchto částí.
6. Proveďte testovací implementaci s využitím paralelizace a ověřte výsledky.

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

Seznam doporučené literatury:

1. ZELINKA, Ivan. Evoluční výpočetní techniky : principy a aplikace. 1. vyd. B.m.: BEN – technická literatura, 2009. ISBN 978-80-7300-218-3.
2. CHANDRASEKARAN, Sunita a Guido JUCKELAND. OpenACC for Programmers: Concepts and Strategies. 1st edition. B.m.: Addison-Wesley Professional, 2017.
3. SUDHOLT, Dirk. Parallel Evolutionary Algorithms. In: Janusz KACPRZYK a Witold PEDRYCZ, ed. Springer Handbook of Computational Intelligence [online]. Berlin, Heidelberg: Springer, 2015 [vid. 2022-12-01], Springer Handbooks, s. 929–959. ISBN 978-3-662-43505-2. Dostupné z: doi:10.1007/978-3-662-43505-2_46.
4. CHENG, John. Professional CUDA C programming. B.m.: John Wiley and Sons, Inc., 2014. ISBN 978-1-118-73932-7.
5. DAVENDRA, Donald a Ivan ZELINKA, ed. Self-Organizing Migrating Algorithm: Methodology and Implementation. 1st ed. 2016 edition. B.m.: Springer, 2016.
6. KACPRZYK, Janusz a Witold PEDRYCZ, ed. Springer handbook of computational intelligence: with 115 tables [online]. Berlin Heidelberg: Springer, 2015. Springer handbooks. ISBN 978-3-662-43504-5. Dostupné z: doi:10.1007/978-3-662-43505-2.

Vedoucí diplomové práce: **Ing. Peter Janků, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **5. listopadu 2023**
Termín odevzdání diplomové práce: **13. května 2024**

**doc. Ing. Jiří Vojtěšek, Ph.D.** v.r.
děkan

**prof. Mgr. Roman Jašek, Ph.D., DBA** v.r.
ředitel ústavu

Ve Zlíně dne 5. ledna 2024

**Prohlašuji, že**

- beru na vědomí, že odevzdáním diplomové 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 diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji diplomovou 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 – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové 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 diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové 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 diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 5.5.2024

Martin Šůstek v.r.
.......................................
podpis studenta

## ABSTRAKT

Cílem diplomové práce je vytvoření paralelních verzí několika vybraných evolučních algoritmů na GPU pomocí knihoven CUDA a OpenACC a vyhodnocení jejich výkonu. Práce je rozdělena do dvou částí – teoretické a praktické. V teoretické části se zabývá vybranými evolučními algoritmy a popisuje jejich strukturu. Dále se rozebírá možnosti paralelizace pomocí CUDA a OpenACC. V praktické části se věnuje vytvoření 3 verzí vybraných algoritmů – sekvenční, CUDA a OpenACC, a srovnání jejich výkonu. Práce hodnotí, za jakých podmínek se vyplatí provádět paralelizaci a jaké omezení a benefity přináší použité knihovny.

Klíčová slova: evoluční algoritmus, paralelizace, GPU, CUDA, OpenACC

## ABSTRACT

The aim of this master’s thesis is the creation of parallel versions of several evolutionary algorithms using the CUDA and OpenACC libraries and evaluation of their performance. The thesis is separated into two parts – theoretical and practical. In the theoretical part, it talks about the chosen evolutionary algorithms and describes their structure. Next it talks about the parallelization options using CUDA and OpenACC. In the practical part, it talks about the creation of 3 version of the selected algorithms - sequential, CUDA and OpenACC, and comparing their performance. The thesis evaluates, under what configuration it is beneficial to do such parallelization and what pros and cons to selected libraries entail.

Keywords: evolutionary algorithms, parallelization, GPU, CUDA, OpenACC

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

## OBSAH

**ÚVOD ... 9**
**I TEORETICKÁ ČÁST ... 10**
**1 ÚVOD DO EVOLUČNÍCH ALGORITMŮ ... 11**
1.1 Optimalizační problém ... 11
1.2 Základní pojmy evolučních algoritmů ... 11
1.3 Principy evolučních algoritmů ... 12
1.4 Společné rysy ... 13
1.5 Omezení prohledávaného prostoru ... 13
1.5.1 Oříznutí ... 14
1.5.2 Náhodná metoda ... 14
1.5.3 Odraz ... 14
1.5.4 Periodická ... 14
**2 VYBRANÉ EVOLUČNÍ ALGORITMY ... 15**
2.1 Genetický algoritmus ... 15
2.1.1 Výběr rodičů ... 16
2.1.1.1 Výběr turnajem ... 16
2.1.1.2 Výběr proporcionální ruletou ... 17
2.1.1.3 Výběr řazenou ruletou ... 17
2.1.2 Elitismus ... 18
2.1.3 Křížení a mutace ... 19
2.1.3.1 Parametry křížení a mutace ... 20
2.2 Diferenciální evoluce ... 21
2.2.1 Inicializace ... 22
2.2.2 Mutace ... 22
2.2.3 Křížení ... 23
2.2.3.1 Jednotné křížení ... 23
2.2.3.2 Exponenciální křížení ... 24
2.2.4 Výběr ... 24
2.3 SOMA ... 24
2.3.1 Mutace ... 25
2.3.2 Migrace ... 25
2.3.3 Parametry ... 26
2.3.3.1 PathLength ... 26
2.3.3.2 Step ... 27
2.3.3.3 PRT ... 27
2.3.4 Migrační strategie ... 27
2.3.4.1 AllToOne ... 27
2.3.4.2 AllToAll ... 27
2.3.4.3 AllToAll Adaptive ... 27
**3 VLASTNOSTI GPU ... 28**
3.1 Architektura GPU ... 28
3.2 Nevýhody práce s GPU ... 29
3.2.1 Paměť ... 29
3.2.2 Divergence warpu ... 29

**4 KNIHOVNY PRO PRÁCI S GPU ..... 30**
4.1 CUDA ..... 30
4.1.1 Výpočetní kernely ..... 30
4.1.2 Hierarchie CUDA vláken ..... 30
4.1.3 Paměťový model ..... 31
4.1.3.1 Globální paměť ..... 31
4.1.3.2 Sdílená paměť ..... 31
4.1.4 Streamy ..... 32
4.2 OPENACC ..... 32
4.2.1 Exekuční model ..... 32
4.2.2 Paralelní direktivy ..... 33
4.2.3 Kompilace OpenACC ..... 34
**II PRAKTICKÁ ČÁST ..... 35**
**5 SOUČASNÝ VÝZKUM ..... 36**
**6 IDENTIFIKACE KRITICKÝCH ČÁSTÍ ..... 37**
6.1 GENETICKÝ ALGORITMUS ..... 37
6.2 DIFERENCIÁLNÍ EVOLUCE ..... 38
6.3 SOMA ..... 39
6.4 VÍCE POPULAČNÍ ZPŮSOBY PARALELIZACE ..... 39
6.4.1 Několik sekvenčních spuštění ..... 39
6.4.2 Několik populací s komunikací ..... 40
**7 NÁVRH PARALELIZACE ..... 41**
7.1 SPOLEČNÉ PRVKY ..... 41
7.1.1 Generování náhodných čísel ..... 41
7.1.2 Práce s pamětí ..... 42
7.2 GENETICKÝ ALGORITMUS ..... 43
7.3 DIFERENCIÁLNÍ EVOLUCE ..... 43
7.4 SOMA ..... 44
**8 SROVNÁNÍ IMPLEMENTACÍ ..... 46**
8.1 HARDWAROVÁ KONFIGURACE ..... 46
8.2 KONFIGURACE ALGORITMŮ ..... 46
8.2.1 Genetický algoritmus ..... 48
8.2.2 Diferenciální evoluce ..... 48
8.2.3 SOMA ..... 48
8.3 GENETICKÝ ALGORITMUS ..... 49
8.3.1 Dejong ..... 49
8.3.2 Schwefel ..... 53
8.4 DIFERENCIÁLNÍ EVOLUCE ..... 57
8.4.1 Dejong ..... 57
8.4.2 Schwefel ..... 61
8.5 SOMA ..... 65
8.5.1 Dejong ..... 65
8.5.2 Schwefel ..... 69

8.6 VLIV OBTÍŽNOSTI ÚČELOVÉ FUNKCE ........ 72
8.7 PAMĚŤOVÉ NÁROKY ........ 75
8.8 ZHODNOCENÍ ........ 77
**ZÁVĚR ........ 79**
**SEZNAM POUŽITÉ LITERATURY ........ 80**
**SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK ........ 84**
**SEZNAM OBRÁZKŮ ........ 85**
**SEZNAM PŘÍLOH ........ 88**

# ÚVOD

V současné době dochází ve světě k obrovskému nárustu využití umělé inteligence pro každodenní činnosti. Algoritmy, které se za těmito, často až magicky vypadajícími nástroji skrývají, jsou výpočetně velmi náročné a pro své výpočty často využívají obrovské množství grafických karet, které paralelně zpracovávají obrovské množství dat.

V této práci se budu zaměřovat na evoluční algoritmy, které také spadají do kategorie umělé inteligence a je tedy zajímavé, jakou paralelizaci nabízejí tyto algoritmy a zda se dá paralelizace efektivně provést na grafické kartě.

Cílem práce je vytvořit paralelní implementaci několika evolučních algoritmů. Nejprve bude třeba vybrat algoritmy a vytvořit jejich sekvenční implementaci a na základě analýzy těchto implementací navrhnout způsob paralelní implementace. K paralelní implementaci mi poslouží knihovny CUDA, podporující paralelizaci na grafických kartách značky NVIDIA, a knihovna OpenACC, která umožnuje paralelizaci na různých akcelerátorech.

Výkon vytvořených paralelních algoritmů poté srovnám se sekvenčním algoritmem a vyhodnotím vhodnost použitých knihoven.

# I. TEORETICKÁ ČÁST

# 1 ÚVOD DO EVOLUČNÍCH ALGORITMŮ

Evoluční algoritmy, dále jen EA, jsou podtřídou evolučních výpočtů a patří do oblasti obecných stochastických vyhledávacích algoritmů. Jedná se o metaheuristické optimalizační algoritmy inspirované přírodními procesy a fungují na principu populací. Metaheuristiky jsou vysoko úrovňové procesy, jejichž účelem je vytvořit, najít nebo vybrat nízko úrovňové procesy, které provádějí lokální vyhledávání. EA jsou použitelné v mnoha optimalizačních problémech s omezenými výpočetními možnostmi a omezenými nebo neperfektními informacemi.[1]

Cílem EA je nalézt co nejlepší řešení optimalizačního problému dostatečně rychle.

## 1.1 Optimalizační problém

Optimalizační problém je jakákoliv situace, u které můžeme chtít hledat nejlepší řešení. Například nalezení nejkratší cesty z bodu A do bodu B. Optimalizační problém je popsán účelovou funkcí, která určuje vhodnost jednotlivých řešení. Účelová funkce představuje matematický model optimalizovaného problému. V příkladě nejkratší cesty je účelovou funkcí funkce pro výpočet délky cesty. Vstupem do této funkce je kombinace parametrů řešeného problému a výstupem je hodnota udávající vhodnost řešení. Dalšími příklady účelové funkce mohou být výpočet počtu předmětů v batohu nebo model pro identifikaci předmětu v obraze.

## 1.2 Základní pojmy evolučních algoritmů

**Jedinec**

Reprezentuje jedno konkrétní nebo také kandidátní řešení daného problému. Na základě řešeného problému a vybraného algoritmu může mít různé podoby, jako například binární řetězec, vektor reálných čísel nebo hodnoty zapsané ve stromové struktuře.

**Dimenze**

Dimenze označuje jeden konkrétní parametr, který má vliv na výslednou hodnotu účelové funkce a je třeba jej optimalizovat. Počet dimenzí tak udává počet optimalizovaných parametrů.

**Populace**

Populace je množina jedinců, tedy množina řešení daného problému, která se v průběhu algoritmu mění. Její stav v jedné iteraci algoritmu označujeme jako **generaci**. Populace může být znázorněna jako matice M*N, kde N představuje počet jedinců a M je počet dimenzí

| | $J_1$ | $J_2$ | $J_3$ | ... | ... | $J_N$ |
|---|---|---|---|---|---|---|
| $D_1$ | 8,01 | 2,25 | -45 | ... | ... | 0,89 |
| $D_2$ | 7,45 | 54,2 | 44 | ... | ... | 15,8 |
| $D_3$ | 9,62 | 33 | 15,8 | ... | ... | -88,1 |
| ... | ... | ... | ... | ... | ... | ... |
| $D_M$ | 8,77 | -78 | 5 | ... | ... | 500 |

Obrázek 1 Populace

**Rodič**

Jedinec z n-té generace, který byl algoritmem vybrán pro vytvoření dalšího jedince v následující generaci.

**Potomek**

Jedinec z generace n+1. Byl vytvořen na základě vlastností svých rodičů.

## 1.3 Principy evolučních algoritmů

EA jsou inspirovány procesy přírodní evoluce, jako jsou reprodukce, mutace, křížení a selekce. Jeho cílem je nalezení dostatečně dobrého řešení v dostatečně rychlém čase. Na počátku je náhodně vytvořena množina kandidátních řešení. Náhodnost této volby zvyšuje kvalitu výsledků algoritmu. Tato populace tvoří první generaci běhu algoritmu. Pro každého jedince z populace je vypočtena jeho kvalita, tedy jeho hodnota účelové funkce. Následuje volba jedinců pro vytvoření další generace. Pro tvorbu další generace se vybírají jedinci s nejlepší vhodností. Toto mohou být jedinci s nejvyšší hodnotou kvality (nebo nejnižší, dle řešeného problému) a označují se jako rodiče. Z rodičů se poté vytvářejí potomci, kteří patří do nové generace. S novou generací se opět provede stejný proces výpočtu jejich kvality, výběru rodičů a vytvoření nové generace. Algoritmus takto obměňuje generaci za generaci a každá z generací je kvalitnější než ta předcházející, a běží tak dlouho dokud nenalezne dostatečně dobré řešení nebo nenarazí na nějakou z ukončujících podmínek. [1] Mezi ukončující podmínky patří například vypršení přiděleného času nebo maximální počet ohodnocení účelové funkce.

## 1.4 Společné rysy

Evoluční algoritmy mají několik společných rysů:

1. Jednoduchost – Obvykle jsou tyto algoritmy jednoduché na implementaci.
2. Hybridnost – Jsou velmi flexibilní v tom, s jakými čísly pracují. Pro reprezentaci jedince lze využívat a kombinovat čísla různých typů jako *integer*, *real* nebo specifické množiny čísel, jako například { -5, 8, 12, 0, 156}.
3. Používání dekadických čísel – Obvykle není třeba převádět jedince do binárního kódu. Z důvodu omezení velikosti binárního čísla totiž při převodu dochází ke zkreslení daného čísla. Druhým problémem binárních čísel, je jejich nepravidelná Hammingova vzdálenost za sebou jdoucích čísel. Při mutaci čísla 8, binárně zapsaného 01000, v jedné z dimenzí, tak může dojít k mutaci na číslo 0, binárně zapsaného 00000, ale zároveň také na číslo 9, binárně zapsaného 110000. Takovéto skokové změny jsou pak pro chování algoritmu nežádoucí.
4. Rychlost – Díky své jednoduchosti dokáží najít požadované řešení velmi rychle.
5. Vícenásobné řešení – Výsledkem EA je nejlepší jedinec, ale poslední generace, vždy obsahuje další dobré jedince. Jsou-li tedy z této populace vybrání například 3 jedinci, představují tito jedinci 3 různá řešení s odstupňovanou kvalitou. Pokud řešený problém obsahuje více globálních extrému, lze očekávat, že je EA nalezne.

[2]

## 1.5 Omezení prohledávaného prostoru

Téměř každý parametr, který má algoritmus optimalizovat, je nějak ohraničen. Tyto omezení často existují jako následek nějakého praktického požadavku (například objem lahve musí být pozitivní číslo). Chování evolučních algoritmů může vést ke kandidátnímu řešení, které v kontextu problému nedává smysl. V takových případech se evoluční algoritmy spoléhají na různé mechanismy, které se s takovýmto řešením vypořádají. Některé metody mohou být použity pro jakýkoli algoritmus, jiné jsou optimalizovány pro specifický algoritmus.

Za běhu algoritmu musí být tedy u každého kandidátního řešení zkontrolováno, zda se nachází v rámci omezení prohledávaného prostoru. Pokud tomu tak není, a jeden nebo více z parametrů se nacházejí mimo tyto hranice, je třeba toto řešení upravit. Jakým způsobem

řešení upravíme může mít vliv na výkon algoritmu a je třeba provést opatrný výběr použité metody. Mezi nejběžnější metody úprav patří následující.[3]

### 1.5.1 Oříznutí

V této metodě, nesmí jedinec překročit omezení v žádné z dimenzí.

$$x'_{i,k} = \begin{cases} b_k^u, & pokud\ x_{i,k} > b_k^u \\ b_k^l, & pokud\ x_{i,k} < b_k^l \\ x_{i,k} & jinak. \end{cases} \tag{1}$$

Kde $x_{i,k}$ je pozice i-tého jedince v k-té dimenzi před kontrolou hranic, $x'_{i,k}$ je nová, upravená pozice po kontrole omezení a $b_k^u$ a $b_k^l$ jsou horní a dolní omezení v dané dimenzi.

### 1.5.2 Náhodná metoda

Pokud některá z dimenze jednotlivce poruší dané omezení, je nová hodnota vygenerována jako nové číslo mezi dolním $b_k^l$ a horním $b_k^u$ omezením.

$$x'_{i,k} = \begin{cases} U(b_k^l, b_k^u), & pokud\ x_{i,k} > b_k^u\ nebo\ x_{i,k} < b_k^l \\ x_{i,k,} & jinak. \end{cases} \tag{2}$$

### 1.5.3 Odraz

Metoda odrazu napodobňuje jednoduché zrcadlo. Jedincova dimenze, která překročila omezení, je odražena zpět do prohledávaného prostoru.

$$x'_{i,k} = \begin{cases} b_k^u - (x_{i,k} - b_k^u), & pokud\ x_{i,k} > b_k^u \\ b_k^l + (b_k^l - x_{i,k}), & pokud\ x_{i,k} < b_k^l \\ x_{i,k} & jinak. \end{cases} \tag{3}$$

### 1.5.4 Periodická

Tato metoda využívá nekonečně velkého prostoru řešení. Tento nekonečně velký prosto je docílen mapováním jedince zpět do prohledávaného prostoru pomocí funkce modulo.

$$x'_{i,k} = b_k^l + \left(x_{i,k} mod(b_k^u - b_k^l)\right). \tag{4}$$

[3]

# 2 VYBRANÉ EVOLUČNÍ ALGORITMY

Vzhledem k tomu, že optimalizační problémy jsou rozmanité a mají různá úskalí, může být na každý problém vhodný jiný algoritmus se specifickými parametry. Tomuto poznatku se v oblasti optimalizace říká No Free Lunch Theorem. [4]

Pro svoji práci jsem si tedy vybral paralelizaci 3 algoritmů: Genetický algoritmus, Diferenciální evoluci a algoritmus SOMA.

Genetické algoritmy patří mezi jedny z nejstarších a stále využívaných algoritmů[5], diferenciální evoluce a její variace jsou jedny z nejvýkonnějších a nejpopulárnějších algoritmů [6] a SOMA se od předchozích dvou liší svou strukturou, kde se více podobá hejnovým algoritmům, a byla vytvořena na UTB[7]. Dále každý z vybraných algoritmů ve své struktuře, nabízí jinou úroveň paralelizace, což může mít zajímavý vliv na závěrečné zhodnocení.

## 2.1 Genetický algoritmus

Genetický algoritmus, dále GA, je optimalizační algoritmus inspirovaný principem přirozeného výběru. Je založen na populacích a využívá princip zákona silnějšího. Nové populace jsou tvořeny opakovaným použitím genetických operátorů, jako křížení a mutace, na jedince v populaci. Jedinec je obvykle reprezentován jako binární vektor, označovaný jako chromozom, a jednotlivé parametry jsou označovány jako geny. Pseudokód genetického algoritmu vypadá následovně [8]:

**GENETICKÝ ALGORITMUS**

***input:***

*Velikost populace N,*
*maximální počet iterací MAX,*
*pravděpodobnost křížení $P_c$,*
*pravděpodobnost mutace $P_m$.*

***output:***

*Globální nejlepší nalezené řešení, $J_{bst}$.*

***begin***

*Vygeneruj počáteční populaci chromozomů velikosti N, $J_n = (i = 1,2,\ldots,N)$.*
*Nastav počet iterací na $t = 0$.*
*Spočítej vhodnost každého chromozomu.*
***while (t < MAX)***
*Vyber rodičovský pár chromozomů na základě jejich vhodnosti.*
*Aplikuj na vybraný pár operaci křížení s pravděpodobností $P_c$.*
*Na nového potomka aplikuj operaci mutace s pravděpodobností $P_m$.*
*Nahraď předchozí populaci nově generovanou populací.*
*Zvyš hodnotu počítadla t o 1.*
***end while***
*Vrať řešení s nejlepší vhodností $J_{bst}$.*

***end***

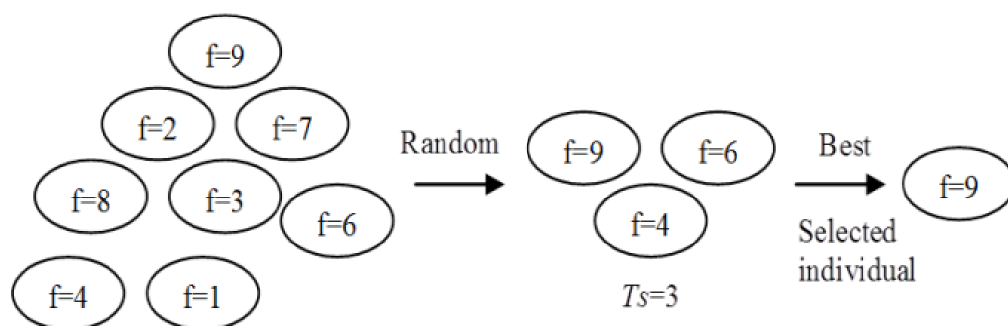
[8]

### 2.1.1 Výběr rodičů

Výběr rodičů určuje, kteří jedinci se budou podílet na tvorbě další generace jedinců a kolik jedinců může každý rodič vytvořit. Hlavním principem výběru je, že čím lepší je jedinec, tím větší má šanci na to stát se rodičem. Špatní jedinci by však neměli být jenom tak zahazováni a měli by mít nějakou šanci být vybrání, jelikož mohou nést užitečnou genetickou informaci. Aby byl zachován dobrý poměr mezi prohledáváním prostoru a zaměřováním se na slibná místa, je tak třeba nastavit správně strategii výběru dobrých a špatných rodičů.[9]

#### *2.1.1.1 Výběr turnajem*

Výběr turnajem je pravděpodobně nejpopulárnější způsob výběru rodičů, a to díky své efektivitě a jednoduchosti implementace. V této metodě je náhodně vybráno *n* jedinců z populace a vybraní jedinci proti sobě soutěží. Jedinec s nejvyšší hodnotou vhodnosti vítězí a je zařazen do dalšího kola turnaje. Kolik jedinců soupeří v každém turnaji, je označováno jako velikost turnaje, a je často nastavována na 2. Výběr turnajem má několik výhod, jako malou časovou složitost, obzvláště pak, pokud je implementace paralelní, nízká pravděpodobnost dominance jednoho jedince a žádné požadavky na škálování vhodnosti jedince nebo řazení jedinců.




Obrázek 2 Výběr jedinců turnajem [9]

Na Obrázek 2 je znázorněn princip turnajového výběru, kde velikost turnaje $T_s$ je nastavena na 3. To znamená, že v každém turnaji spolu soupeří 3 chromozomy, a pouze ten s nejlepší vhodností je vybrán pro další kroky algoritmu. U turnajového výběru způsobují větší turnaje větší ztrátu různorodosti populace.[9]

#### *2.1.1.2 Výběr proporcionální ruletou*

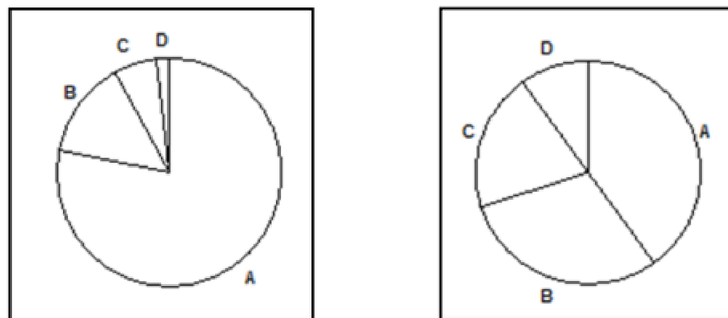
Ve výběru proporcionální ruletou jsou jednotlivci vybíráni s pravděpodobností, která je přímo úměrná jejich vhodnosti, tedy každý jedinec, má přiřazenu proporcionální část rulety. Pravděpodobnost výběru jedince, může být tedy chápána jako zatočení rulety, kde políčko každého jedince je proporcionálně velké jeho vhodnosti. Jedinci s nejvyšší vhodností pak mají největší políčka, a tedy šanci být vybráni. Obvod rulety je dán součtem všech vhodností jedinců. Když je tato ruleta roztočena, kolo se zastaví a šipka bude ukazovat na jedno z políček, s největší pravděpodobností na jedno z nejširších. Všechny políčka však mají šanci vyhrát. Kolo je takto roztáčeno pokaždé, když je potřeba vybrat nového jedince, a jsou tak častěji vybíráni jedinci s většími políčky než jedinci s menšími políčky. Označme $f_1, f_2, \dots, f_n$ hodnoty kovhodnosti jedinců *1,2,...,n*. Pak pravděpodobnost $P_i$, vybrání jedince $i$, je definována jako:

$$P_i = \frac{f_i}{\sum_{j=1}^{n} f_j}. \tag{5}$$

Výhodou výběru proporcionální ruletou je, že nezahazuje žádné jednotlivce z populace a dává všem šanci být vybráni, čímž zachovává různorodost populace. Má však také několik výrazných nevýhod. Výjimečně vhodní jednotlivci vytvoří na začátku prohledávání sklon, který může způsobit předčasnou konvergenci jedinců, a tak následné snížení různorodosti populace. Dalším problémem je, že se tento přístup těžko používá pro minimalizační problémy, kde musí být pro ruletu účelová funkce přetransformována do maximalizační podoby. [9]

#### *2.1.1.3 Výběr řazenou ruletou*

Výběr řazenou ruletou je metoda, kde pravděpodobnost výběru chromozomu, je určena na základě pořadí jeho vhodnosti vůči celé populaci. Výběr řazenou ruletou nejprve seřadí jednotlivce na základě jejich vhodnosti a poté spočítá pravděpodobnosti, podle pořadí, a nikoliv podle vhodnosti. Výběr řazenou ruletou tak může udržet konstantní směřování průběhu algoritmu vytvořením jednotného škálování a není ovlivněno výjimečně vhodnými jedinci.



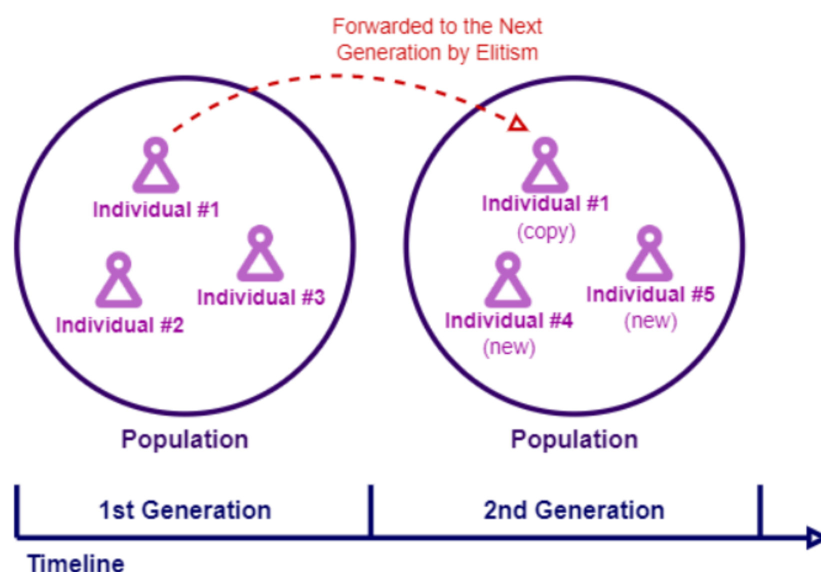

Obrázek 3 Srovnání pravděpodobností u výběru proporcionální ruletou a výběru řazenou ruletou [9]

### 2.1.2 Elitismus

Elitismus je koncept, který vyžaduje splnění alespoň jedné z následujících podmínek:

- Zahrnutí nejlepších jedinců generace do následující generace.
- Několikanásobní využití nejlepších jedinců z generace v křížení s jinými náhodnými jedinci při tvorbě nové generace.

Hlavním důvodem pro elitismus v GA je zachování odkazu na slibná místa v prohledávaném prostotu mezi generacemi. Elitismus tedy zajišťuje, že nejlepší jedinci, nelezení za běhu algoritmu, jsou součástí poslední generace, ze které se bere výsledek samotného algoritmu.[10]




Obrázek 4 Elitismus [10]

Elitismus ovšem také GA oslabuje. Jelikož zachovává již známe jedince do dalších generací, omezuje místo v nové generaci pro nalezení nových jedinců, kteří by přinesli lepší výsledek. Zmenšuje tedy různorodost populace a snižuje schopnost algoritmu prohledávat celý prostor.

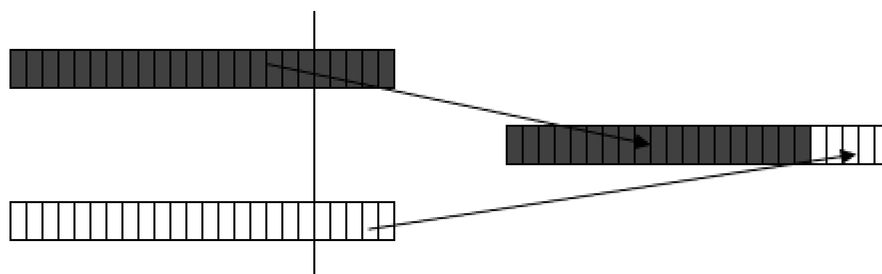
Z tohoto důvodu se tato skupina elitních jedinců udržuje malá. Běžně využívaná velikost je nastavována na:

- 10 % jedinců z generace tvoří skupinu elitních jedinců
- 50 % nejlepších jedinců z elitní skupiny je zahrnuto do další generace
- 50 % nejhorších jedinců z elitní skupiny je použito pro křížení s ne-elitními jedinci

Jaké nastavení elitismu se v praxi používá je však velmi individuální a velmi záleží na použitém algoritmu a řešeném problému.[10]

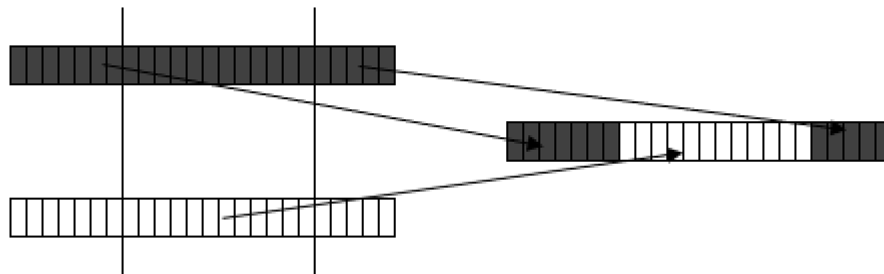
### 2.1.3 Křížení a mutace

Křížení je proces používaný pro reprodukci. Dochází při něm k prohození chromozómů rodičů v určitém bodě. Nový potomek poté obsahuje část chromozomů od každého rodiče. Existuje několik způsobů křížení: jednobodové, dvoubodové nebo vícebodové. Jednobodové křížení dělí chromozómy na dvě části v určitém bodě.



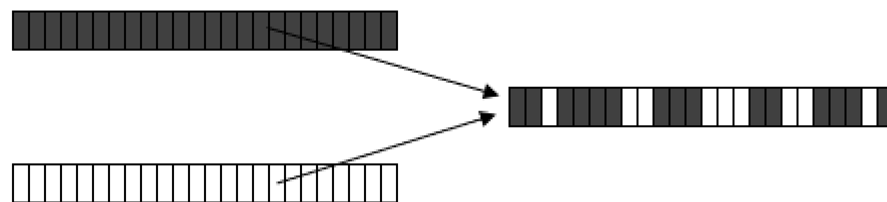
Obrázek 5 Jednobodové křížení

Při dvoubodovém křížení dochází k výměně chromozomů mezi dvěma náhodně vybranými body.



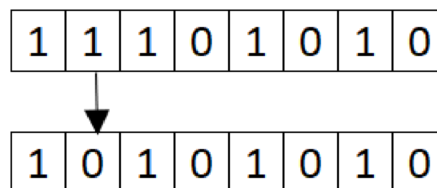
Obrázek 6 Dvoubodové křížení

Při vícenásobném křížení se náhodně určují geny, které se předávají novému potomkovi.



Obrázek 7 Vícenásobné křížení

Jakmile je dokončeno křížení, dochází v algoritmu k mutaci. Jedná se o jednoduché převrácení hodnoty některých genů. O které geny se jedná je většinou vybráno náhodně. Opět existuje několik druhů od jednobodové až po vícebodovou.[2]




Obrázek 8 Jednobodová mutace

#### *2.1.3.1 Parametry křížení a mutace*

Křížení a mutace se nedějí vždy. Provádějí se na základě vstupních parametrů pravděpodobnosti křížení a mutace. Aby potomci nebyli přímými kopiemi svých rodičů, neměla by pravděpodobnost křížení být nulová. Pokud by pravděpodobnost byla nulová, změnu populace by pak zajišťoval pouze výběr rodičů, kteří se účastní produkce potomků a potomkem by se stal jeden z rodičů. Pokud by pravděpodobnost byla 100 %, znamenalo by to, že všichni potomci by byli tvořeni křížením a v populaci by nezůstal žádný původní jedinec. Předpokladem pro úspěch křížení, je vzájemné předávání dobré části chromozomu a následné společné nalezení lepšího řešení.

Pravděpodobnost mutace funguje stejně jako u křížení. Pokud je pravděpodobnost nastavena na 100 %, je celý potomek změněn na svůj přesný opak. V opačném případě není u jedince zmutováno nic. Mutace funguje jako prevence před uváznutím v lokálním extrému. Její pravděpodobnost by neměla být příliš vysoká. Potom by se algoritmus více podobal náhodnému prohledávání. [2]

## 2.2 Diferenciální evoluce

Diferenciální evoluce, dále jen DE, je jedním z nejpopulárnějších optimalizačních algoritmů pro řešení komplexních optimalizačních problémů. DE vytváří nové jedince matematickými operacemi nad jedinci za určitých podmínek. Jednotliví jedinci budou nahrazeni, pokud bude nalezeno lepší řešení. DE je považován za robustní a jednoduchý algoritmus, jelikož je jeho proces vyhledávání řízen několika specifickými parametry jako faktor škálování a četnost křížení. Podobně jako jiné EA, produkuje DE nové potomky pomocí mutace, křížení a selekce, kde k výkonu algoritmu nejvíce přispívají mutace a křížení.

Základní kostra algoritmu DE se skládá ze 4 částí, a to inicializace, mutace, křížení a selekce. Inicializace je jednorázový proces, zatímco ostatní tři části jsou opakovány v rámci procesu prohledávání.[11]

---

**DIFERENCIÁLNÍ EVOLUCE**

---

***input:***
    *Velikost populace, N,*
    *maximální počet iterací, MAX,*
    *mutační strategie, MS.*
***output:***
    *Globální nejlepší nalezené řešení, $J_{bst.}$*

---

***begin***
    *Náhodně vygeneruj počáteční populaci velikosti N.*
    *Nastav počet iterací na k = 0.*
    *Spočítej vhodnost každého jedince.*
    ***while (k < MAX)***
        *Vytvoř mutační vektory pomocí strategie MS.*
        *Vytvoř testovací vektory kombinací rodičovských a mutačních vektorů.*
        *Spočítej vhodnost testovacích vektorů.*
        *Vyber vítězný vektor jako potomka do nové generace.*
        *Zvyš hodnotu počítadla k o 1.*
    ***end while***
    *Vrať nejlepší nalezené řešení $J_{bst.}$*
***end***

---

[12]

### 2.2.1 Inicializace

Inicializace je počáteční fáze, která se odehrává v DE při hledání globálního optima nacházejícího se v *D*-rozměrném prostoru. Prvotní populace pro vícedimenzionální optimalizační problém se skládá z $n$ vektorů parametrů s reálnými hodnotami, které chceme optimalizovat, kde $n$ vyjadřuje velikost populace. Během *k*-té iterace, je každý $i$-tý jedinec reprezentován jako *D*-dimenzionální vektor:

$$X_i^k = (X_{i,1}, X_{i,2}, \dots, X_{i,D}), \tag{6}$$

kde $i = 1,2, \dots, NP$. Prvotní jedinci mohou být generováni na základě vrchního a horního omezení prohledávaného prostoru reprezentovaných následovně:

$$X_{min} = (X_{min,1}, X_{min,2}, \dots, X_{min,D}), \tag{7}$$

$$X_{max} = (X_{max,1}, X_{max,2}, \dots, X_{max,D}). \tag{8}$$

Každý *j*-tý parametr jedince, je tedy inicializován jako náhodná hodnota mezi horní hranicí $X_{max,j}$ a dolní hranicí $X_{min,j}$:

$$X_{i,j}^{(0)} = X_{min,j} + rand_{i,j}[0,1](X_{max,j} - X_{min,j}), \tag{9}$$

kde $rand_{i,j}[0,1]$ je náhodná veličina s normálním rozdělením, která vrací hodnoty mezi 0 a 1.[11]

### 2.2.2 Mutace

V kontextu evolučních algoritmů je mutace náhodný proces změny vybraných parametrů. V DE je mutant, označený jako $Y_i^t$, vytvořen pomocí mutačního procesu na základě vybraného rodičovského vektoru $X_i^t$. Obecně je mutační strategie používaná v DE zapisována jako „*DE/*/n*“, kde *n* označuje počet použitých diferenčních vektorů a * reprezentuje rodičovský vektor použitý v procesu mutace. Vyhledávací mechanismy pěti běžně používaných strategií v DE jsou následující:

DE/rand/1:

$$Y_i^t = X_{r_1}^t + F(X_{r_2}^t - X_{r_3}^t), \tag{10}$$

DE/rand/2:

$$Y_i^t = X_{r_1}^t + F\left(X_{r_2}^t - X_{r_3}^t\right) + F\left(X_{r_4}^t - X_{r_5}^t\right), \tag{11}$$

DE/best/1:

$$Y_i^t = X_{best}^t + F\left(X_{r_1}^t - X_{r_2}^t\right), \tag{12}$$

DE/best/2:

$$Y_i^t = X_{best}^t + F\left(X_{r_1}^t - X_{r_2}^t\right) + F\left(X_{r_3}^t - X_{r_4}^t\right), \tag{13}$$

DE/current-to-best/1:

$$Y_i^t = X_i^t + F(X_{best}^t - X_i^t) + F\left(X_{r_1}^t - X_{r_2}^t\right). \tag{14}$$

Kde $r_1$ je populační index jedince vybraného jako výchozí vektor, $r_2, r_3, r_4$ a $r_5$ jsou populační indexy náhodných jedinců vybraných pro vytvoření mutačního vektoru, kde $r_1, r_2, r_3, r_4, r_5 \in [1, NP]$ a $r_1 \neq r_2 \neq r_3 \neq r_4 \neq r_5 \neq i$. $X_{best}^t$ značí, že jako rodičovský vektor byl vybrán nejlepší jedinec z dané generace a $F$ je míra škálování určená pro kontrolu vlivu mutace a nabývá hodnot $[0,1]$. Výběr vhodné hodnoty $F$ je kritický pro dosažení správné rovnováhy mezi prohledáváním prostoru a prohledáváním slibných míst a předejití tak předčasné konvergenci jedinců nebo naopak příliš pomalé konvergenci.[11]

### 2.2.3 Křížení

V této fázi dochází ke křížení rodičovského vektoru s mutačním na základě pravděpodobnosti a výsledkem je testovací vektor neboli potomek. Tento proces umožnuje potomkovi zdědit vlastnosti rodičovského vektoru. Dva populární způsoby jsou jednotné křížení a exponenciální křížení.

#### *2.2.3.1 Jednotné křížení*

Jednotné křížení se řídí mírou křížení $CR$, která nabývá hodnoty $[0,1]$. Testovací vektor vytvořen jednotným křížením pak má podobu:

$$Z_i^t = \begin{cases} X_{i,j}^t \; pokud \; rand_{i,j} \leq CR \; nebo \; j = k \\ Y_{i,j}^t \; jinak. \end{cases} \tag{15}$$

Kde $rand_{i,j}$ je náhodné číslo mezi $[0,1]$ a $k \in \{1,2,\ldots,D\}$ je náhodně vybraný index dimenze, který zajišťuje, že alespoň jeden parametr testovacího vektoru $Z_i^t$ je zděděn z mutanta $Y_{i,j}^t$.

#### 2.2.3.2 *Exponenciální křížení*

Pro exponenciální křížení je náhodně vybráno celé číslo $n \in \{1,2,\dots,D\}$, označující počáteční index dimenze cílového vektoru pro provedení křížení s mutantem. Další parametr $L \in \{1,2,\dots,D\}$ označuje počet parametrů, které mají být zděděny z mutačního vektoru pro vytvoření testovacího vektoru. S použitím $n$ a $L$ je pak možné testovací řešení vytvořit jako:

$$Z_i^t = \begin{cases} X_{i,j}^t \; pokud \; j = \langle n \rangle_D, \langle n+1 \rangle_D, \dots, \langle n+L-1 \rangle_D \\ Y_{i,j}^t \; jinak, \end{cases} \tag{16}$$

kde $\langle . \rangle_D$ označuje funkci modulo $D$. Exponenciální křížení dosahuje lepších výsledků na určitém druhu optimalizačních problémů, jako například problémů, u kterých existují vztahy mezi sousedícími parametry.[11]

### 2.2.4 Výběr

Proces výběru určuje, zda se do další generace dostane rodič, nebo testovací vektor neboli potomek. Jakmile je vytvořena nová populace, proces mutace, křížení a výběru se opakuje dokola, dokud není splněna ukončující podmínka. Existují dva způsoby výběru, a to lokální a globální. Proces výběru lze matematicky zapsat jako:

$$X_i^{t+1} = \begin{cases} Z_i^t \; pokud \; f(Z_i^t) \leq f(X_i^t) \\ X_i^t \; jinak, \end{cases} \tag{17}$$

kde $f$ označuje účelovou funkci použitou pro určení vhodnosti vybraného řešení. Pokud testovací řešení získá lepší hodnocení vhodnosti, dojde k nahrazení současného cílového vektoru $X_i^t$ vektorem $Z_i^t$ v další generaci.[11]

## 2.3 SOMA

SOMA, z anglického Self-organizing migrating algorithm, je algoritmus založený na samo organizačním chování skupin a jedinců v „sociálním prostředí“. Může být také klasifikován jako evoluční algoritmus, přestože nevytváří žádné nové populace, a to na základě totožnosti výstupů z generací s klasickými EA. Mezi generacemi se mění pouze pozice jedinců pomocí takzvané migrace. V každé migraci jedinci mění svoji pozici, například na základě pozice nejlepšího jedince, označovaného jako leader. Jakým způsobem se budou k leaderovi pohybovat pak určuje takzvaný *PRTVector*.[7][13]

**SOMA - SELF-ORGANIZING MIGRATING ALGORITHM**

***input:***

*Velikost populace, N*
*maximální počet iterací, MAX.*
***output:***
*Globální nejlepší nalezené řešení, L.*

***begin***
*Náhodně vygeneruj počáteční populaci velikosti N.*
*Nastav počet iterací na k = 0.*
***while (k < MAX)***
*Vyber z populace nejvhodnějšího jedince a označ leaderem L.*
***for** i = 0; i < pop; i++ **do***
*Vygeneruj PRTVector.*
*Proveď migrační kroky směrem k L.*
*Ulož si nejlepší nalezenou pozici.*
***end while***
*Vrať nejlepší nalezené řešení L.*
***end***

[13]

### 2.3.1 Mutace

SOMA pro dosažení mutace používá parametr *prt*. Tento parametr je používám pro vytvoření chybového vektoru (*PRTVector*) a to následovně:

$$PRTVector_j = \begin{cases} 1\ pokud\ rnd_j\ < prt \\ 0\ jinak. \end{cases} \tag{18}$$

| $rnd_j$ | PRTVector |
|---|---|
| 0.231 | 1 |
| 0.456 | 0 |
| 0.671 | 0 |
| 0.119 | 1 |

Obrázek 9 Příklad chybového vektoru pro 4 parametry a *prt* = 0,3 [7]

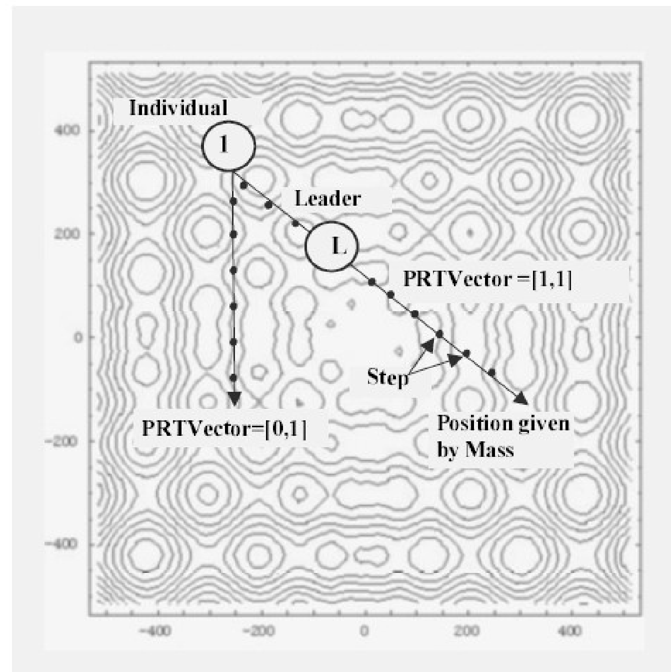
Zajímavostí tohoto přístupu je, že tento vektor se generuje před započetím migrace jedince, tedy před „křížením“. *PRTVector* definuje pohyb jedince v *n*-dimenzionálním prostoru.

### 2.3.2 Migrace

V standardních EA křížení vytváří nové jedince na základě informací z přechozí generace. Geometricky vzato dochází k výběru nové pozice v *N* dimenzionálním prostoru. V SOMA však dochází ke generování nových pozic pomocí migrace. Můžeme nad nimi uvažovat, jako nad novými jedinci získanými speciální operací křížení. Toto křížení určuje chování SOMA. Pohyb jednotlivce je tedy dán takto:

$$x_{i,j}^{MLnew} = x_{i,j,start}^{ML} + \left(x_{L,j}^{ML} - x_{i,j,start}^{ML}\right)t * PRTVector_j, \quad (19)$$

kde $t \in \{0, po\ Step\ do\ PathLength\}$ a $ML$ je aktuální migrační smyčka.




Obrázek 10 *PRTVector* a jeho vliv na pohyb jedince [7]

Každý jedince tedy provádí tolik kroků velikosti $Step$, kolik se vejde do maximální délky cesty $PathLength$ , ve které dimenzi se pohybuje určuje dříve vypočtený *PRTVector*. Po dokončení těchto kroků se každý jedince přesune do nejlepší nalezené pozice, na kterou narazil během provádění kroků a ta se tak stává výchozím bodem pro migraci jedince v následující generaci. Nevznikne tedy nový jedinec, ale dojde k posunu již existujícího.

### 2.3.3 Parametry

SOMA používá pro svůj chod několik parametrů ovlivňujících kvalitu výsledku.

#### *2.3.3.1 PathLength*

$PathLength \in \langle 1.1; 3 \rangle$

Tento parametr definuje, jak daleko za *Leaderem* se jedinec zastaví. Délka 1, znamená, že se zastaví na *Leaderovi*. Při nastavení délky 2, skončí jedinec ve stejné vzdálenosti od leadera jako začal, ale na druhé straně *Leadera*.

#### *2.3.3.2 Step*

$$Step \in \langle 0.11; PathLength \rangle$$

Velikost kroku určuje podrobnost, s jakou je prostor prohledáván. V případě „jednoduchých“ prostorů, lze použít větší délku kroku, pro zajištění urychlení vyhledávání. Pokud však nejsou známi informace o prohledávaném prostoru, je doporučeno používat zmíněné hodnoty.

#### *2.3.3.3 PRT*

$$prt \in \langle 0,1 \rangle$$

Určuje pravděpodobnost, s jakou bude jedinec dělat kroky přímo k *Leaderovi*. Se zvyšováním hodnoty dochází ke zrychlení konvergence jedinců.

[13]

### 2.3.4 Migrační strategie

Existuje několik strategií, které jsou používány při migraci jedinců.

#### *2.3.4.1 AllToOne*

Výše popsaná strategie, kde se všichni jedinci pohybují k *Leaderovi* a *Leader* se v rámci migrace nepohybuje.

#### *2.3.4.2 AllToAll*

V této strategii neexistuje *Leader*. Všichni jedinci se hýbou ke všem jedincům. Tato strategie je časově více náročná ale zato nachází výsledky rychleji než strategie AllToOne.

#### *2.3.4.3 AllToAll Adaptive*

Rozdíl proti obyčejné AllToAll strategii je v tom, že jedinci nezačínají migrace pokaždé ze stejného místa, ale začínají z nejlepšího nalezeného místa během přechozích migrací.

[7][13]

# 3 VLASTNOSTI GPU

Sekvenční algoritmy jsou obvykle psány jako posloupnost instrukcí, které jsou prováděny jedna po druhé na procesoru a je možné provádět pouze jednu instrukci najednou. Paralelizace algoritmu znamená rozdělení algoritmu na menší části, které mohou být řešeny najednou. Každá taková část má svoji posloupnost instrukcí, které se pro každou část provádí najednou jiným procesorem.[14]

Jakou paralelizaci pak podporují systémy popisuje Flynnova taxonomie, která je rozděluje do čtyřech hlavních kategorií: [15]

**SISD**

Single instruction, single data stream. Procesor dokáže najednou zpracovat pouze jedny data pomocí jedné instrukce. Zapadají sem takzvané jednoprocesorové systémy, někdy nazývané jako skalární.

**SIMD**

Single instruction, multiple data stream. Procesor dokáže najednou pomocí jedné instrukce zpracovat několik streamů dat. Podkategorií SIMD je takzvaný SIMT (Single instruction, multiple threads), který zpracování dat rozděluje do více vláken, každé s vlastními registry. Právě do SIMT zapadají moderní GPU.

**MISD**

Multiple instruction, single data stream. Několik procesorů, každý se svou instrukcí, pracují nad jedněmi daty. Toto není velmi využívaný přístup, jak příklad lze však uvést využití firmou Microsoft v pozadí vyhledávače Bing. [16]

**MIMD**

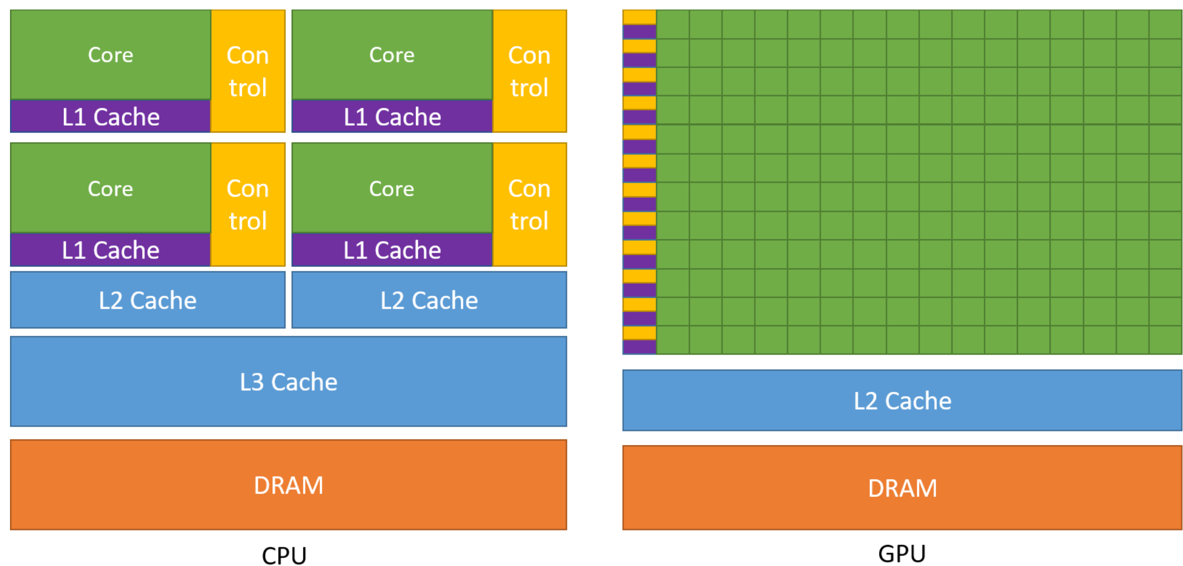
Multiple instruction, multiple data streams. Několik procesorů, každý se svou instrukcí, pracují nad několika streamy dat. Zapadají sem například moderní více jádrová CPU.

[15]

## 3.1 Architektura GPU

GPU i CPU obsahují výpočetní jednotky, které dokážou provádět instrukce. Každé je však navrženo za jinými účely. CPU je navrženo pro rychlé zpracování sekvenčních instrukcí, tedy vláken, na několika desítkách jader, kdežto GPU je navrženo pro paralelní

zpracování tisíců těchto vláken. Jednotlivá vlákna jsou sice pomalejší, ale celkově GPU dokáže zpracovat mnohem větší množství dat. [17]




Obrázek 11 CPU a GPU architektura [17]

GPU tedy obsahuje velké množství výpočetních jader, které jsou rozděleny do jednotlivých Streamovacích multiprocesorů (SM). Hlavní úkolem SM je paralelní vykonávání bloků vláken. Na jednom SM se může nacházet až 8 bloků vláken, každý jiné velikosti. SM tyto bloky zpracovává pomocí takzvaných warpů, což jsou skupiny 32 vláken, kterým warp scheduler daného SM přiřazuje instrukce k provedení formou SIMT. [18]

## 3.2 Nevýhody práce s GPU

### 3.2.1 Paměť

Algoritmus běžící na CPU, jak sekvenční, tak paralelní, dokáže pracovat s daty uloženými v paměti RAM. GPU však do této paměti přístup nemá, ale disponuje vlastní pamětí, do které je třeba před zahájením výpočtů potřebná data přesunout a po dokončení výpočtů nová data přesunout zpět.

### 3.2.2 Divergence warpu

Warp je skupina 32 vláken která provádějí stejnou instrukci. V situaci, kdy se však jedno z vláken má nacházet v jiné větvi kódu, než ostatní vlákna dochází k takzvané divergenci warpu, která způsobuje snížení efektivity paralelizace algoritmu.

# 4 KNIHOVNY PRO PRÁCI S GPU

## 4.1 CUDA

CUDA je platforma pro paralelní výpočty a programové rozhraní, které umožňuje softwaru využívat GPU značky NVIDIA pro obecné výpočty, tedy umožňuje takzvaný general-purpose computing on GPUs. CUDA API je softwarová vrstva umožňující přístup k virtuálním instrukcím GPU a prvkům umožňujícím paralelizaci pomocí výpočetních kernelů.

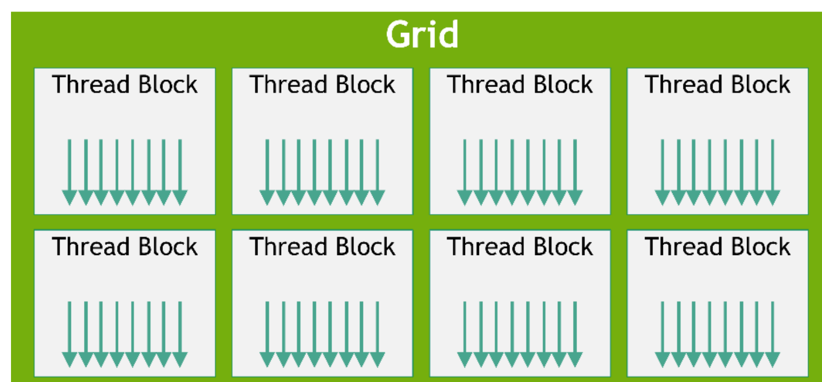
### 4.1.1 Výpočetní kernely

Kernely jsou funkce, které jsou při volání paralelně vykonány N CUDA vlákny na několika SM, oproti běžné funkci, která je vykonána jedenkrát. Tyto funkce jsou definovány speciálním klíčovým slovem __*global*__ a při jejich volání je nutné definovat počet CUDA vláken, které mají funkci provést, pomocí syntaxe <<<…>>>.

### 4.1.2 Hierarchie CUDA vláken

Každé CUDA vlákno má přístup k vektorové proměnné *threadIdx*, díky které může identifikovat svou pozici uvnitř bloku vláken, nazývaného *thread block*. Proměnná *threadIdx* je tří dimenzionální vektor a umožňuje tedy vytvořit až tří dimenzionální *thread block*.

Počet vláken v bloku je z hardwarových důvodů omezen a na současných GPU je tento limit 1024 vláken. Každý kernel však může být proveden několika stejně velkými bloky a výsledný počet využívaných vláken je pak násobkem velikosti bloku a počtu bloků.
Bloky jsou organizovány do 1 až 3dimenzionální mřížky a počet bloků v mřížce je obvykle určen velikostí zpracovávaného problému, tak aby celkový počet vláken dosáhl velikosti problému.

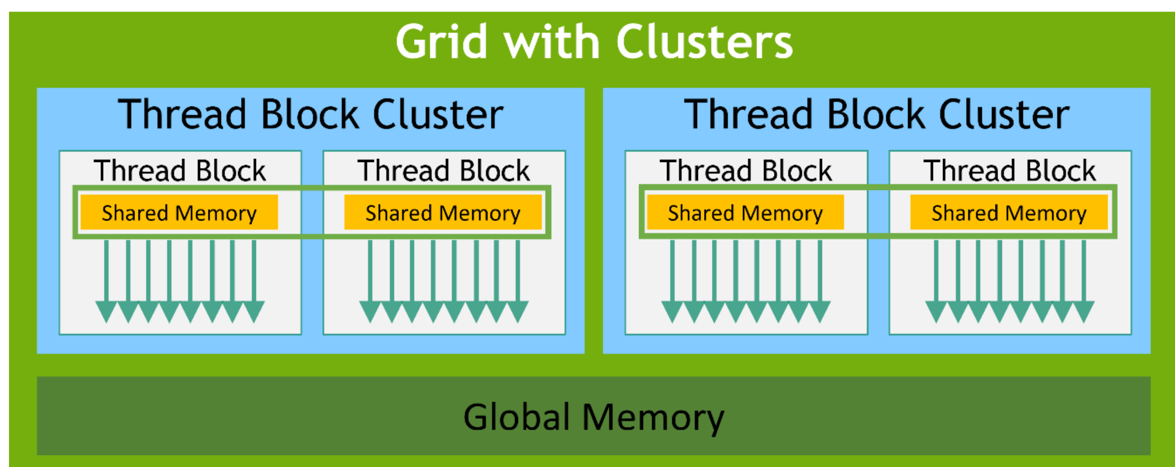



Obrázek 12 Hierarchie vláken v mřížce [17]

Synchronizace vláken v jednom bloku, pro zajištění koordinovaného přístupu do paměti, je možná voláním funkce *__syncthreads()*, která funguje jako bariéra, kde musí všechna vlákna počkat, než je kterékoli z nich puštěno dále.

### 4.1.3 Paměťový model

CUDA vlákna mohou přistupovat pouze k paměti nacházející se na GPU. Moderní GPU poskytují různé paměti s různými časovými dopady na rychlost běhu programu, různými velikosti a různými způsoby čtení a zápisu.




Obrázek 13 Paměťový model [17]

#### *4.1.3.1 Globální paměť*

Největší a nejpomalejší paměť, přístupná z každého vlákna. Pro přístup k této paměti z vlákna na CPU, existuje speciální funkce *cudaMemcpy*, která umožňuje přenášet data mezi pamětí CPU a GPU. Pro přístup z GPU je využíván pointer, vytvořený například pomocí funkce *cudaMalloc*. Data v této paměti zůstávají po celou dobu běhu programu, který paměť alokoval.

#### *4.1.3.2 Sdílená paměť*

Každé vlákno v bloku má přístup pouze ke sdílené paměti odpovídajícího bloku. V současné době je výjimkou z tohoto pravidla nejnovější GPU NVIDIA H100[19], která představila koncept klusterů. V rámci těchto klusterů je pak možné přistupovat ke sdílené paměti ostatních bloků vláken ve stejném klusteru. Dá se očekávat, že toto chování bude v budoucnu normou a nikoliv výjimkou.

Sdílená paměť je rychlejší než globální paměť a často se proto používá jako softwarová cache pro data, která se mají během výpočtu využívat několikrát. Životnost dat v této paměti je pak rovna životnosti odpovídajícího bloku vláken a je proto třeba na počátku běhu každého kernelu tuto paměť znovu naplnit.

[17]

### 4.1.4 Streamy

Streamy umožňují paralelní spouštění několika nezávislých operací. Mohou být využity například pro paralelizaci přenosu dat mezi CPU a GPU s během kernelu (tento kernel však musí využívat jiná data, než která jsou paralelně přenášena), jelikož v závislosti na velikosti dat může přenos trvat dlouhou dobu a tento čas pak může být využit k provádění výpočtů, namísto čekání na dokončení přenosu dat. Další možností je spuštění několika paralelních kernelů.[20]

## 4.2 OPENACC

OpenACC je vysokoúrovňový programovací model pro paralelní výpočty na CPU, GPU a jiných akcelerátorech, vyvíjený organizací The OpenACC Organization.

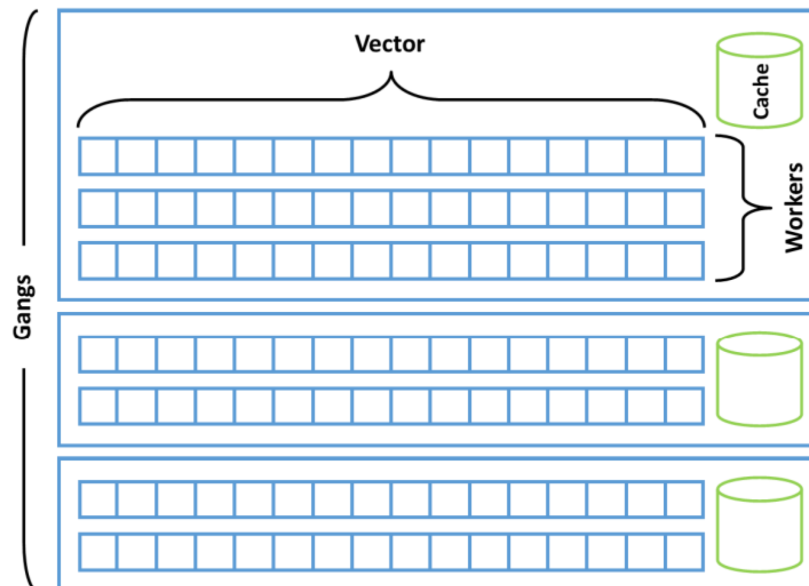
OpenACC API používá direktivy pro specifikaci smyček a regionů kódu, které mají být vykonány na akcelerátoru.[21] Velkou výhodou využívání direktiv je možnost minimálních úprav do sériového kódu pro zprovoznění paralelní verze. V důsledku pak při odstranění OpenACC direktiv zůstává sériová verze plně funkční.

### 4.2.1 Exekuční model

Exekuční model OpenACC je rozdělen do tří úrovní: gang, worker a vector. Při spuštění paralelního regionu, dochází ke spuštění jednoho nebo více gangů, každého s jedním nebo více workery, kde každý worker může mít jeden nebo více vektorových pruhů podporujících vektorové operace.

Zpočátku jeden worker každého gangu provádí stejný kód, dokud nenarazí na smyčku označenou pro paralelizaci, kde dojde k rozdělení jednotlivých iterací. Pokud je smyčka označena pro paralelizaci na úrovni gangu, jsou iterace rozděleny mezi gangy. Pokud je namísto toho smyčka označena pro paralelizaci na úrovni workerů, dojde k rozdělení iterací

smyčky mezi jednotlivé workery gangu. Vyskytuje-li se označení pro paralelizaci jak na úrovni gangu, tak workerů, jsou iterace rozděleny mezi všechny workery všech gangů. Smyčka označená pro paralelizaci na úrovni vektorů se obdobně rozdělí mezi jednotlivé pruhy využívající vektorové a SIMD instrukce.




Obrázek 14 Exekuční model OpenACC [22]

### 4.2.2 Paralelní direktivy

**#pragma acc parallel**

Zahájí paralelní provádění kódu na akcelerátoru. Vytvoří jeden nebo více gangů, které paralelně provedou následující region kódu. Jeden worker z každého gangu redundantně provádí kód, dokud nenarazí na konstrukt *loop*.

**#pragma acc loop**

Poskytuje kompilátoru informace o tom, jak se zachovat k následující smyčce. Specifikuje například jakou úroveň paralelizace použít. Dá se také specifikovat že smyčka provádí paralelní redukci pomocí slova *reduction*. Redukce podporuje několik redukčních operátorů, jako například +, min a max.

**#pragma acc kernel**

Definuje region kódu, který má být kompilátorem přeložen do sekvence kernelů, které budou spuštěny na akcelerátoru. Spoléhá se na schopnosti kompilátoru rozeznat které

smyčky jsou bezpečné pro paralelizaci a jejich následnou akceleraci. Každý kompilátor tak může daný region paralelizovat jiným způsobem. [21]

### 4.2.3 Kompilace OpenACC

OpenACC při kompilaci interpretuje direktivy ve zdrojovém kódu a ověřuje korektnost kódu. Při tomto vytváří programátorovy výstup obsahující mnoho užitečných informací o jeho závěrech. Výstup může typicky obsahovat nalezené smyčky, proměnné omezující paralelizaci a proměnné které jsou třeba kopírovat na GPU.

```
[ 12%] Building C object CMakeFiles/acc_gen.dir/main.c.o
CreateNewPopulation:
     38, Generating present(newPopulation[:],population[:],newFitness[:],fitness[:])
         Generating implicit firstprivate(bestIdx,dimension)
         Accelerator serial kernel generated
         Generating NVIDIA GPU code
     44, Generating present(newPopulation[:],identifiers[:],newFitness[:],population[:],randomCounters[:],threadData[:],fitness[:],processData[:])
         Generating implicit firstprivate(i,popSize)
         Generating NVIDIA GPU code
         48, #pragma acc loop gang /* blockIdx.x */
     48, Generating implicit firstprivate(dimension,seed,problem,boundary,crossProb,generation,mutateProb)
    124, Generating copy(newBestIdx) [if not already present]
         Generating copyin(bestValue) [if not already present]
         Generating present(newFitness[:])
         Generating implicit firstprivate(popSize,i)
         Generating NVIDIA GPU code
        126, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             Generating reduction(min:bestValue)
    131, Generating implicit firstprivate(popSize,i)
         Generating NVIDIA GPU code
        133, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
main:
    205, Generating create(newPopulation[:size]) [if not already present]
         Generating copyout(bestValue) [if not already present]
         Generating copyin(population[:size]) [if not already present]
         Generating create(newFitness[:popSize],identifiers[:popSize]) [if not already present]
         Generating copyin(fitness[:popSize]) [if not already present]
         Generating create(threadData[:popSize],randomCounters[:popSize]) [if not already present]
         Generating copyin(processData[:1]) [if not already present]
         Generating create(bestIdx) [if not already present]
    239, Generating copyout(bestValue) [if not already present]
         Generating present(fitness[:])
         Accelerator serial kernel generated
         Generating NVIDIA GPU code
[ 25%] Linking C executable acc_gen
[100%] Built target acc_gen
```

Obrázek 15 Ukázka OpenACC výstupu

Tento výstup tak může programátorovi posloužit nejen jako kontrola, že OpenACC interpretuje direktivy tak jak je zamýšlel, ale také jako velmi dobrý nástroj na upozornění, kde existuje potenciál na další paralelizaci nebo kde se nachází logická chyba v programu a usnadnit tak následující vývoj.

# II. PRAKTICKÁ ČÁST

# 5 SOUČASNÝ VÝZKUM

Obdobnému tématu se v posledních letech věnovalo několik prací. V roce 2014 se skupina SINGH Satvir, KAUR Jaspreet a SINHA Rashmi Sharan ve své práci A Comprehensive Survey on Various Evolutionary Algorithms on GPU zabývala paralelizací několika evolučních algoritmů, a to pomocí knihovny CUDA. V této práci ukázali, že paralelizace všech jimi vybraných algoritmů poskytuje zrychlení v řádu několika desítek oproti sekvenčnímu algoritmu. [23]

EA na GPU se zabývá také práce Samarth Gupta a Gary Tan: A Scalable Parallel Implementation of Evolutionary Algorithms for Multi-Objective Optimization on GPUs, kde autoři implementují paralelní verzi evolučního algoritmu NSGA-II a podle testované konfigurace dosahují 3-15násobné zrychlení. [24]

V roce 2017 SYBERFELDT Anna a EKBLOM Tom se v práci A Comparative Evaluation of the GPU vs The CPU for Parallelization of Evolutionary Algorithms Through Multiple Independent Runs věnovali paralelizaci EA, jak na CPU, tak na GPU a srovnání jejich výsledků. Pro paralelizaci využívaly přístup několika sekvenčních spuštění, popsaný v 6.4.1, a ukázali, že GPU nemusí být vždy výkonnější, než CPU a poskytují několik rad, jak vybrat mezi CPU a GPU pro paralelizaci daných problémů.[25]

Jenda z novějších publikací je práce Patrika Valkoviče: GPU Parallelization of Evolutionary Algorithmsm, která se zabývá využitím programovacího jazyka Python k vytvoření knihovny pro podporu různých EA. Pro paralelizaci je využívá knihovnu PyTorch, která na pozadí běh na GPU paralelizuje pomocí CUDy. Vzniklou knihovnu otestoval a vyhodnotil zrychlení oproti sekvenčnímu algoritmu, které se projevilo obzvláště při větších populacích.[26]

# 6 IDENTIFIKACE KRITICKÝCH ČÁSTÍ

Všechny vybrané algoritmy mají společnou vlastnost, že v rámci jedné generace jsou jedinci datově navzájem nezávislí. Výpočty, které zajišťují vytváření populace nové generace tedy mohou být paralelizovány.

Dalším prvkem, který je společný pro všechny algoritmy, je výpočet hodnoty účelové funkce. Tato funkce je obvykle vypočítávána jako suma parciálních výpočtů nad každým prvkem vstupního pole. Tento výpočet lze paralelizovat pomocí paralelní redukce. Některé matematické operace využívané pro výpočet účelové funkce však mohou být časově mnohem náročnější než jiné (například výpočet funkce sinus je časově náročnější, než obyčejný součet dvou čísel), což má vliv na výsledný efekt paralelizace.

## 6.1 Genetický algoritmus

Genetický algoritmus je svou strukturou velice jednoduchý. Kromě iterace jedinců v něm nedochází k žádným smyčkám, které by se daly paralelizovat.

Při pohledu na profil sekvenčního programu na (Obrázek 16) je vidět funkci *CreateNewPopulation*, která se stará o vytvoření nových jedinců v rámci generace. V této funkci stráví program asi 91 % času. Celkem program stráví asi 86 % času výpočtem účelové funkce (Dejong 1 v tomto příkladě) a samotné kroky algoritmu jako mutace (*Mutate)* a křížení (*CrossParents*) společně nezabírají ani 5 % času. Minimální část z těchto 86 %, asi 0,002 %, času výpočtu účelové funkce je vykonáno mimo funkci *CreateNewPopulation*, počátečním generování populace. U GA je tedy vhodné zaměřit se hlavně na paralelizaci výpočtu účelové funkce.

| Incl. | Self | Distance | Calling | Callee |
|---|---|---|---|---|
| 91.39 | 0.21 | 1 | 49 | CreateNewPopulation (seq_gen) |
| 86.16 | 0.09 | 1-2 (2) | 49 951 | CalculateFitness (seq_gen) |
| 86.07 | 86.07 | 2-3 (3) | 49 951 | Dejong_Fitness (seq_gen) |
| 7.32 | 0.24 | 3-5 (3) | 1 420 834 | 0x00000000001091f0 |
| 7.08 | 0.61 | 4-6 (4) | 1 420 834 | rand (libc.so.6: rand.c) |
| 6.88 | 0.00 | 1 | 1 000 | NewIndividual (seq_gen) |
| 6.88 | 1.72 | 2 | 1 000 | New_Dejong (seq_gen) |
| 6.47 | 2.57 | 5-7 (5) | 1 420 834 | random (libc.so.6: random.c) |
| 4.04 | 0.21 | 2 | 48 951 | CrossParents (seq_gen) |
| 3.89 | 3.89 | 6-8 (6) | 1 420 834 | random_r (libc.so.6: random_r.c) |
| 3.38 | 0.01 | 3 | 73 404 | 0x0000000000109190 |
| 3.37 | 3.37 | 4 | 73 332 | __memcpy_avx_unaligned_erms ( |
| 2.29 | 0.07 | 2 | 48 951 | RandomTournament (seq_gen) |
| 2.22 | 0.43 | 3 | 97 902 | SelectRandomWinner (seq_gen) |
| 1.79 | 0.28 | 4 | 293 706 | RandomPopIdx (seq_gen) |
| 0.62 | 0.10 | 3 | 100 313 | SmallRandom (seq_gen) |
| 0.41 | 0.08 | 2 | 48 951 | Mutate (seq_gen) |
| 0.16 | 0.03 | 3 | 26 815 | RandomChromozome (seq_gen) |

Obrázek 16 Profil GA

## 6.2 Diferenciální evoluce

Algoritmus DE provádí několik smyček. První při provádění mutace, druhou při křížení. Obě tyto smyčky jsou bezpečně paralelizovatelné.

Na Obrázek 17 je vidět profil sekvenční DE. Opět je zde funkce *CreateNewPopulation*, která vytváří nové jedince, ale u DE již většinu času zabírají právě dvě zmíněné smyčky křížení – 54 %, a mutace – 33 %. Také je zde vidět, velký vliv generování náhodných čísel, jelikož se v DE náhodná čísla generují v rámci smyčky křížení.

| Incl. | Self | Distance | Calling | Callee |
|---|---|---|---|---|
| 98.83 | 0.06 | 1 | 49 | CreateNewPopulation (seq_diff) |
| 53.70 | 12.59 | 2 | 49 000 | Cross (seq_diff) |
| 41.03 | 6.85 | 3 | 48 951 000 | SmallRandom (seq_diff) |
| 35.05 | 1.17 | 3-4 (4) | 50 196 314 | 0x00000000001091f0 |
| 33.88 | 2.93 | 4-5 (5) | 50 196 314 | rand (libc.so.6: rand.c) |
| 33.18 | 33.06 | 2 | 49 000 | Mutate (seq_diff) |
| 30.95 | 12.29 | 5-6 (6) | 50 196 314 | random (libc.so.6: random.c) |
| 18.66 | 18.66 | 6-7 (7) | 50 196 314 | random_r (libc.so.6: random_r.c) |
| 11.68 | 0.01 | 1-2 (2) | 50 000 | CalculateFitness (seq_diff) |
| 11.67 | 11.67 | 2-3 (3) | 50 000 | Dejong_Fitness (seq_diff) |
| 0.93 | 0.00 | 1 | 1 000 | NewIndividual (seq_diff) |
| 0.93 | 0.23 | 2 | 1 000 | New_Dejong (seq_diff) |
| 0.45 | 0.00 | 2 | 49 000 | 0x0000000000109190 |
| 0.45 | 0.45 | 3 | 49 000 | __memcpy_avx_unaligned_erms ( |
| 0.16 | 0.03 | 3 | 196 314 | RandomPopIdx (seq_diff) |
| 0.04 | 0.01 | 3 | 49 000 | RandomChromozome (seq_diff) |

Obrázek 17 Profil DE

## 6.3 SOMA

U algoritmu SOMA jsou vhodné pro paralelizaci smyčky generování PRT vektoru a migrace jedinců. Při migraci musí každý jedinec provést několik kroků a každý tento krok znamená přepočítávání někdy i všech dimenzí. Ve výsledku se tedy jedná o dvojitou vnořenou smyčku.

Z profilu SOMA algoritmu je zřejmé, že proces migrace zabírá téměř veškerý čas běhu algoritmu a je proto vhodným kandidátem na paralelizaci. Smyčka generování PRT vektoru oproti tomu zabírá mnohem méně času.

| Incl. | Self | Distance | Calling | Callee |
|---|---|---|---|---|
| 99.76 | 0.05 | 1 | 49 | RunGeneration (seq_soma) |
| 90.34 | 61.44 | 2 | 4 802 | Migrate (seq_soma) |
| 28.94 | 0.30 | 1-3 (3) | 62 526 | CalculateFitness (seq_soma) |
| 28.65 | 28.65 | 2-4 (4) | 62 526 | Dejong_Fitness (seq_soma) |
| 9.38 | 1.54 | 2 | 4 802 | GeneratePRT (seq_soma) |
| 7.84 | 1.31 | 3 | 480 200 | SmallRandom (seq_soma) |
| 6.67 | 0.22 | 3-4 (4) | 490 200 | 0x00000000001091f0 |
| 6.44 | 0.56 | 4-5 (5) | 490 200 | rand (libc.so.6: rand.c) |
| 5.89 | 2.34 | 5-6 (6) | 490 200 | random (libc.so.6: random.c) |
| 3.55 | 3.55 | 6-7 (7) | 490 200 | random_r (libc.so.6: random_r.c) |
| 0.18 | 0.00 | 1 | 100 | NewIndividual (seq_soma) |
| 0.18 | 0.05 | 2 | 100 | New_Dejong (seq_soma) |

Obrázek 18 Profil SOMA

## 6.4 Více populační způsoby paralelizace

Toto jsou způsoby, které se nezaměřují na paralelizaci samotné struktury algoritmu, ale zaměřují se na více paralelních běhů algoritmů s různými populacemi.

### 6.4.1 Několik sekvenčních spuštění

Tento přístup paralelně spouští několik sekvenčních algoritmů. Každý sekvenční algoritmus je nezávislý a pracuje se svou populací. Parametry jednotlivých spuštění mohou být rozdílné a mohou tak mít více flexibilní prohledávání prostoru. V momentě, kdy všechny sekvenční algoritmy naleznou svá globální minima, dojde k výběru finálního minima z těchto minim, které je použito jako výsledek celého algoritmu.

### 6.4.2 Několik populací s komunikací

Podobně jako u 6.4.1 zde existuje několik populací. Za běhu si však tyto populace občas vyměňují své nejlepší jedince. Výsledkem je poté opět minimum ze všech populací.

# 7 NÁVRH PARALELIZACE

## 7.1 Společné prvky

### 7.1.1 Generování náhodných čísel

Evoluční algoritmy jsou stochastické a pro svoji funkčnost vyžadují využití náhodných čísel. CUDA náhodná čísla podporuje pomocí knihovny cuRAND[27]. Tato knihovna umožňuje generování náhodných čísel přímo na GPU. Pro každé vlákno je třeba pouze inicializovat a udržovat stavovou proměnnou.

```
1. int tid = threadIdx.x + blockIdx.x * blockDim.x;
2. curand_init(seed, tid, 0, &state[tid]);
```

Kterou poté každé vlákno využívá při generování náhodných čísel.

```
1. unsigned int randomNumber = curand(&state[i]);
```

OpenACC žádnou podporu náhodných čísel na akcelerátoru nenabízí. Jedna z možností, jak tento problém vyřešit je předpočítání náhodných čísel na CPU a následné zkopírování těchto čísel na akcelerátor, kde jsou poté náhodná čísla po jednom čtena. Tento přístup je vhodný při malém množství takto generovaných čísel, kdy sekvenční generování a následná kopie nezabere příliš času. U většího množství je však už samotné sekvenční generování náhodných čísel příliš pomalé, jak šlo vidět na Obrázek 17. Dalším problémem tohoto přístupu je nutnost znalosti, kolik náhodných čísel je třeba vygenerovat. Například u diferenciální evoluce je tento požadavek nedosažitelný. U DE dochází k výběru náhodných indexů, které se od sebe musí lišit, a nelze předem určit, kolik náhodných čísel bude pro tento výběr potřeba.

Dalším přístupem je využití algoritmu pro generování náhodných čísel. Pro generování náhodných čísel v OpenACC využívám tento přístup a konkrétně se jedná o DES PNRG generátor[28]. Obdobně jakou u cuRAND, využívá každé vlákno stavovou proměnnou, navíc je však využívána společná stavová proměnná.

```
1. identifiers[idx] = idx + seed;
2. create_identifier(&identifiers[idx]);
3. initialize_individual(processData, threadData + idx, identifiers[idx]);
```

Při generování náhodných čísel je také potřeba udržovat počítadlo, které určuje kolikáté číslo v řadě má být vygenerováno.

```
1. unsigned long randomNumber;
2. make_prn(processData, threadData + idx, randomCounters[idx], &randomNumber);
```

### 7.1.2 Práce s pamětí

Důležitým aspektem práce s GPU je správa paměti. Kopírování dat na a z GPU zabírá hodně času a pokud by se k němu nepřistupovalo správně, může zabírat většinu času běhu programu a výrazně tak zhoršit jeho výkon. Z vybraných algoritmů potřebují GA a SOMA vědět, který jedinec byl v předchozí generaci nejlepší, aby mohli začít počítat novou generaci. DE nepotřebuje vědět o předchozí generaci nic. Díky tomuto, není třeba mezi generacemi provádět mnoho kopírování dat z GPU a je možné data ponechávat po celou dobu běhu programu aktuální pouze na GPU a u GA a SOMA si kopírovat jen index nejlepšího jedince.

CUDA data mezi jednotlivými kernely ponechává v paměti. OpenACC však mezi paralelními regiony data automaticky neuchovává. Všechny paralelní regiony, mezi kterými se mají data sdílet musí být proto součástí regionu obaleného pomocí *#pragma acc data* se specifikací jaké data se mají kopírovat mezi CPU a GPU a jak. Příkladem:

```
1. #pragma acc data copyin(population[:size],fitness[:popSize],processData[:1])\
2.          create(newPopulation[:size],newFitness[:popSize],identifiers[:popSize]\
3.          ,threadData[:popSize],bestIdx,randomCounters[:popSize])
4.          copyout(bestValue)
5. {
6.        ...
7. }
```

Každý paralelní region poté specifikuje, jaké data jsou již na GPU přítomná:

```
1. #pragma acc parallel loop present(population, fitness, newFitness, newPopulation,\
2.        processData,threadData,identifiers,randomCounters)
```

Aby algoritmus pracoval se správnými daty stačí mezi generacemi zaměnit pointery na pole se starou populací a pole s novou populací. Obdobně s poli hodnot účelových funkcí.

```
1. double * tmp;
2. tmp = population;
3. population = newPopulation;
4. newPopulation = tmp;
5.
6. tmp = newFitness;
7. newFitness = fitness;
8. fitness = tmp;
```

U sekvenčního algoritmu můžeme využívat stále stejnou paměť pro stejné výpočty u každého jedince a do dalších proměnných si ukládat pouze pár nejnutnějších informací. Většinou tak sekvenční programy nevyžadují velké množství paměti. Abych však mohl provádět výpočty na všech jedincích paralelně, je třeba mít u některých algoritmů připraveno mnohonásobně větší množství paměti tak, aby mohlo každé vlákno ukládat své výpočty do jemu odpovídající paměti a nedocházelo k současným přístupům k jedné paměti.

## 7.2 Genetický algoritmus

Pro paralelizaci tohoto algoritmu využiji pouze výše zmíněnou paralelizaci jedinců a výpočtu účelové funkce. Tímto způsobem bude navíc dobře vidět, jaký vliv může mít jen takováto jednoduchá paralelizace.

Před každou další generací, je třeba vybrat nejlepšího jedince pro zachování elitismu. V sekvenční implementaci se toto dá dokázat jednoduše tím, že se v rámci smyčky jedinců udržuje proměnná, která si pamatuje index jedince, který měl v rámci generace nejlepší hodnotu účelové funkce.

```
1. NewFitness[i]=CalculateFitness(&NewPopulation[i*Dimensions],Dimensions,Problem);
2. if (NewFitness[i] < NewFitness[NewBestIdx])
3. {
4.   NewBestIdx = i;
5. }
```

V paralelní implementaci se toto stává problematickým a pokud by se implementace nezměnila, docházelo by k paralelnímu přístupu k jedné proměnné a algoritmus by se nechoval korektně.

V CUDA tento problém řeším pomocí knihovny CUB[29] a její funkce *cub::DeviceReduce::ArgMin()*. Tato funkce vezme pole prvků a na GPU najde minimální prvek v poli a jeho index.

```
1. cub::DeviceReduce::ArgMin(g_helperArray, helperArrayBytes, g_fitness, g_bestIndividual,
popSize);
```

V OpenACC je situace lehce komplikovanější. V rámci jedné smyčky, lze pomocí redukce najít pouze minimální prvek. Je proto potřeba následovně v rámci druhé smyčky a atomických operací najít index minimálního prvku.

```
 1. #pragma acc parallel loop reduction(min: bestValue)
 2. for(int i = 0; i < popSize; i++) {
 3.   if(newFitness[i] < bestValue)
 4.   {
 5.       bestValue = newFitness[i];
 6.   }
 7. }
 8. #pragma acc parallel loop
 9. for(int i = 0; i < popSize; i++) {
10.       if(newFitness[i] == bestValue){
11.                 #pragma acc atomic write
12.                 newBestIdx = i;
13.       }
14. }
```

## 7.3 Diferenciální evoluce

Paralelizace částí uvedených v 6.2, vyžaduje v CUDA značné úpravy. Pro mutaci každého jedince je první potřeba získat indexy jeho cílových vektorů. Toho dosahuji tak, že před kernelem mutace, který spouští $velikost\ populace * počet\ dimenzí$ vláken, spustím

kernel velikosti $velikost\ populace$, který pro každého jedince získá potřebné indexy. Každý index je pak využíván kernelem mutace pro $počet\ dimenzí$ vláken, které společně počítají jednoho jedince. Pro kernel křížení je potřeba opět provést obdobný kernel pro určení mutačního indexu a indexu pro elitismus. Vzniklé testovací vektory jsou pak dalším kernelem srovnány s rodičovským vektorem a je provedeno rozhodnutí o tom, který vektor se dostane do další generace. Po doběhnutí všech generací je nejlepší jedinec vybrán opět pomocí funkce *cub::DeviceReduce::ArgMin()*.

Oproti tomu jsou úpravy pro OpenACC minimální. Každou smyčku stačí obalit odpovídajícím *#pragma acc* konstruktem a po doběhnutí všech generací vybrat nejlepšího jedince způsobem popsaným v 7.2.

## 7.4 SOMA

Migrace jedinců u SOMA je vnořená smyčka. V sekvenčním algoritmus je možné lehce si pamatovat, který z kroků migrace je pro daného jedince ten nejlepší. V paralelní verzi je toto opět problematické a je třeba algoritmus rozdělit na více částí, které provádějí konkrétní operace. Jednou z těchto operací je provedení migračních kroků a zapamatování si, kde kroky dopadly. Další operací je pak jakou měla v daném bodě účelová funkce hodnotu.

V CUDA se dá vnořená smyčka migrace řešit pomocí dynamického paralelizmu. Tento koncept umožňuje spouštění vnořených kernelů. V mé implementaci tohoto využívám spuštěním kernelu o velikosti $velikost\ populace$, který v sobě následně spouští další kernel, který pro daného jedince paralelně provede všechny migrační kroky. Pro očekávanou funkcionalitu paralelního výpočtu migrací paralelně běžících jedinců je však potřeba vnořený kernel spouštět v speciálním streamu *cudaStreamPerThread*, čímž se každý vnořený kernel spustí ve vlastním streamu a dojde k opravdovému paralelnímu běhu. Jediným omezením tohoto přístupu je fyzická kapacita samotného GPU. Každé GPU má totiž limit kolik kernelů může běžet v najednou a také kolik kernelů může čekat ve frontě. Při přesažení limitu fronty dojde k chybě a kernely které se do fronty nevlezou se jednoduše neprovedou. Proto tyto kernely spouštím po dávkách a to tak, že rodičovský kernel spustím pouze s tolika vlákny, kolik kernelů je GPU schopno provádět najednou.

```
1. void __global__ RunMigrations(...)
2. {
3.       ...
4.     G_Migrate<<<blockCount,blockSize,0,cudaStreamPerThread>>>(...);
5. }
6.
7. for(int i = 0; i <= popSize;i+= gpuLimit)
8. {
9.       RunMigrations<<<1,gpuLimit>>>(...);
10. }
```

Další kroky, jako výpočet hodnot účelových funkcí v daných krocích a výběr nejlepšího kroku jsou provedeny obdobným způsobem.

Při implementaci pomocí OpenACC jsou úpravy opět výrazně menší. Všechny smyčky je pouze třeba obalit odpovídajícími *#pragma acc* konstrukty. Fáze výběru nejlepšího kroku je přesunut až po dokončení migrací a krok výběru leadera následuje hned poté.

# 8 SROVNÁNÍ IMPLEMENTACÍ

Výše popsané paralelní implementace jsem vytvořil v jazyce C++. Implementace využívají společné prvky a nastavení konkrétní konfigurace lze dosáhnout pomocí parametrů příkazové řádky. Implementace je navržena tak, aby se dala jednoduše rozšířit o další algoritmy a účelové funkce. Celá implementace je přiložena jako Příloha PI.

V této kapitole budu testovat mé implementace a popíšu jejich výkonnost. Čas běhu algoritmu měřím přímo v kódu a to tak, že vynechávám první generování populace, které probíhá u všech implementací stejně a čas ukončím po vrácení nejlepčšího jedince poslední generace. V CUDA a OpenACC se do času počítají také veškeré režie spojené s alokací, kopírováním a uvolnováním paměti, stejně tak se počítá i inicializace stavů pro generování náhodných čísel.

## 8.1 Hardwarová konfigurace

Testy provádím na stolní PC v konfiguraci:

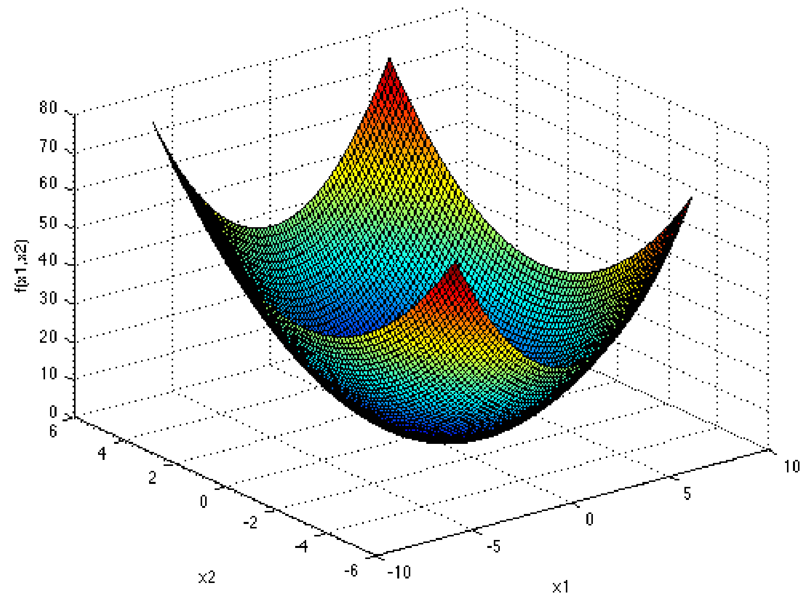
- Operační systém Ubuntu 22.04.4 LTS
- Procesor AMD Ryzen 7 2700X
- GPU NVIDIA GeForce RTX 2070 s 8 GB paměti
- OpenACC kompilátor z NVIDIA HPC SDK 23.11
- CUDA verze 12.3

## 8.2 Konfigurace algoritmů

Všechny algoritmy jsou testovány s počtem generací nastaveným na 50. Pro velikost populace použiju hodnoty od 10 do 5000, stejně tak pro počet dimenzí. Každou konfiguraci spouštím 10krát a algoritmy srovnávám na základě průměru délky těchto běhů.

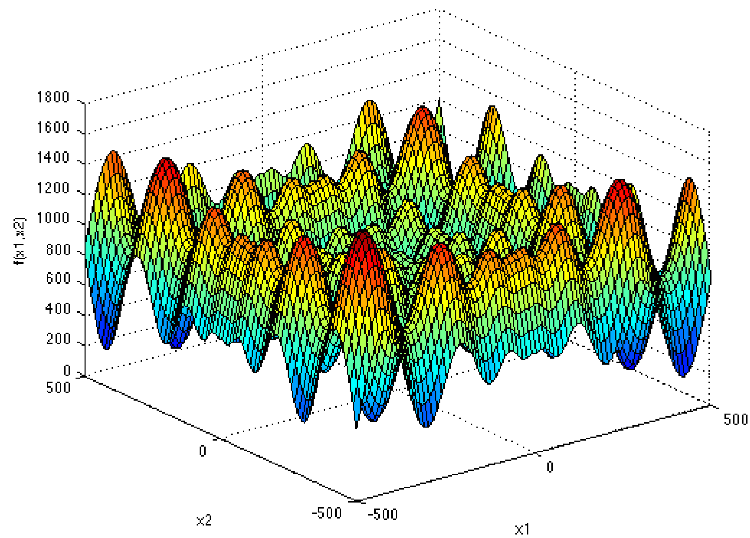
Pro pozorování vlivu rychlosti výpočtu účelové funkce, na celkovou rychlost výpočtu, provádím testování na dvou účelových funkcích:

- De Jong’s 1st function: $f(\boldsymbol{x}) = \sum_{i=1}^{d} x_i^2$, $-5,12 < x_i < 5,12$.




Obrázek 19 De Jong's 1st function ve třech dimenzích [30]

- Schwefel function: $f(\boldsymbol{x}) = 418.9829d - \sum_{i=1}^{d} x_i sin\left(\sqrt{|x_i|}\right), -500 < x_i < 500.$




Obrázek 20 Schwefel function ve třech dimenzích [31]

Kde $\boldsymbol{x}$ je vstupní vektor a $d$ je dimenze řešeného problému.

Kompletní datový soubor, ze kterého jsou průměry počítány je k nahlédnutí v Příloze P II.

### 8.2.1 Genetický algoritmus

Pravděpodobnost křížení $P_c = 0.5$.

Pravděpodobnost mutace $P_m = 0.05$.

Výběr rodičů: Turnajem velikosti 3.

### 8.2.2 Diferenciální evoluce

Mutační strategie DE/rand/1 s mírou škálování $F = 0.8$.

Jednotné křížení s mírou křížení $CR = 0.5$.

### 8.2.3 SOMA

Migrační strategie AllToOne.

Délka cesty $PathLength = 1.5$.

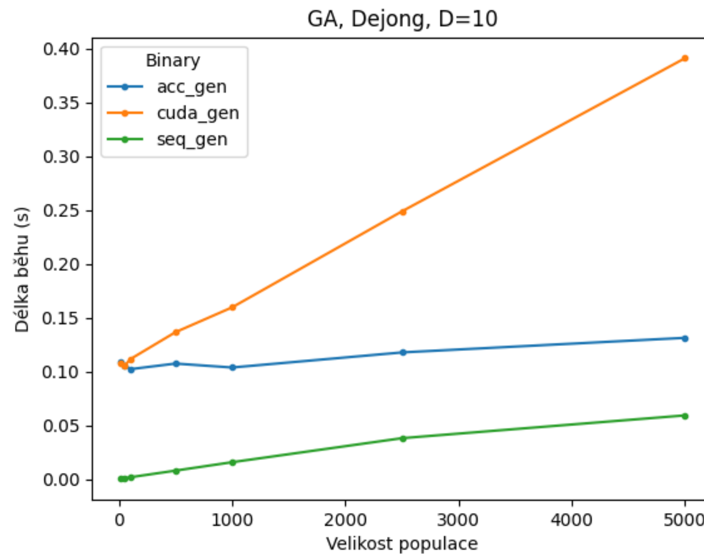
Velikost kroku $Step = 0.49$.

Směrový parametr $PRT = 0.3$.

Při tomto nastavení dělají jedinci při migraci pouze 3 kroky. Toto nastavení nahrává sekvenčnímu algoritmu, jelikož s poklesem počtu kroků klesá množství dostupné paralelizace.

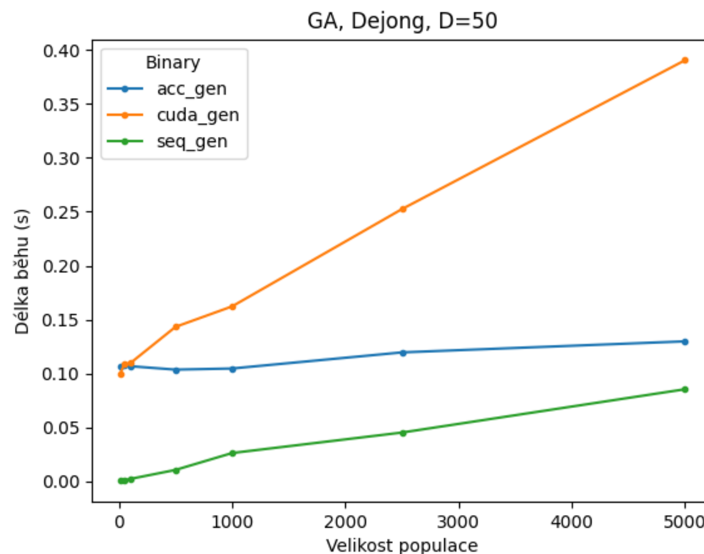
## 8.3 Genetický algoritmus

### 8.3.1 Dejong



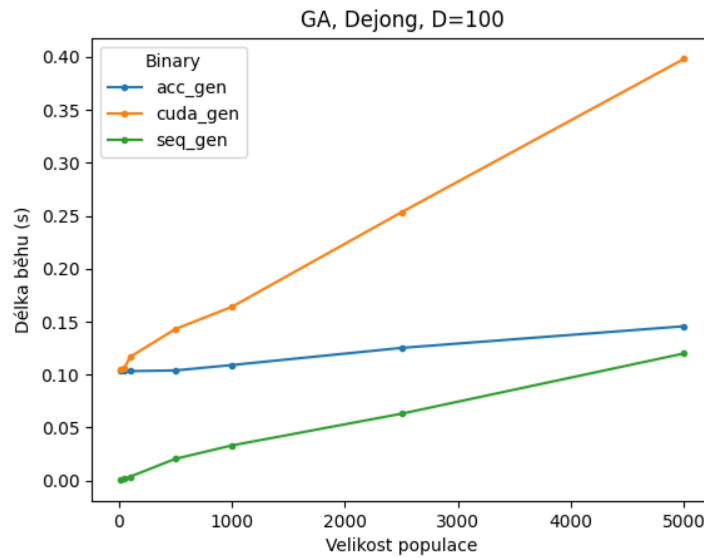

Obrázek 21 GA, Dejong 1st, dimenze = 10

Při takto nízké dimenzi řešeného problému sekvenční algoritmus funguje velmi dobře. Implementace v OpenACC běží asi 2x pomaleji a CUDA verze je ve srovnání asi 8x pomalejší.

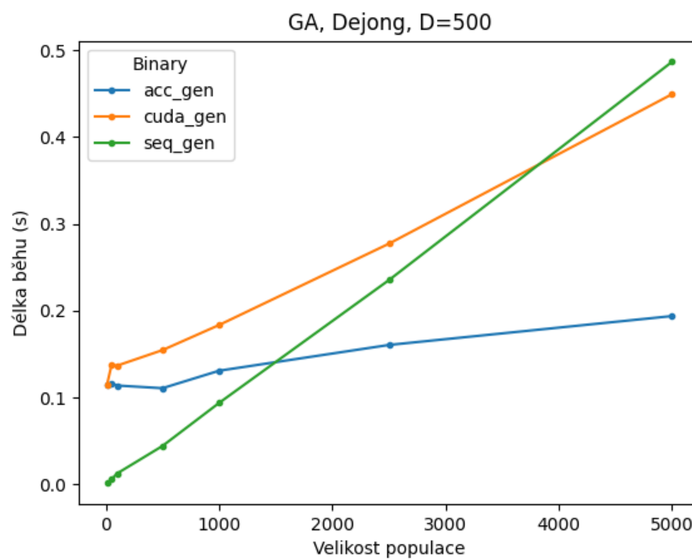



Obrázek 22 GA, Dejong 1st, dimenze = 50

Sekvenční algoritmus stále funguje velmi dobře. Se zvyšující populací se však jeho rychlost pomalu přibližuje k OpenACC. CUDA zde stále zaostává.

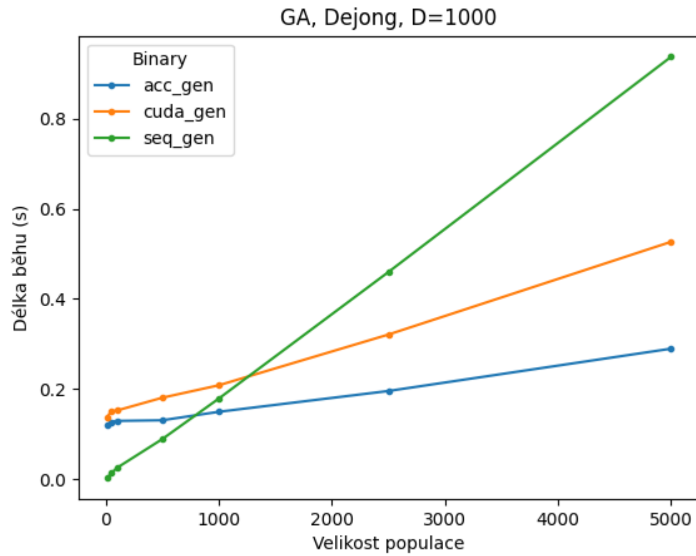



Obrázek 23 GA, Dejong 1st, dimenze = 100



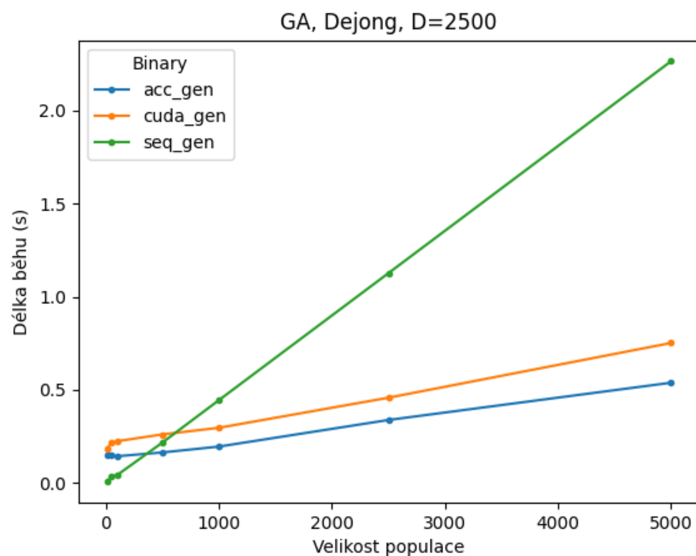

Obrázek 24 GA, Dejong 1st, dimenze = 500

Sekvenční algoritmus je stále nejrychlejší při menší populaci. Při populaci větší jak 1500 již však zaostává za OpenACC a při populaci zhruba 4000 již zaostává i za CUDA implementací.



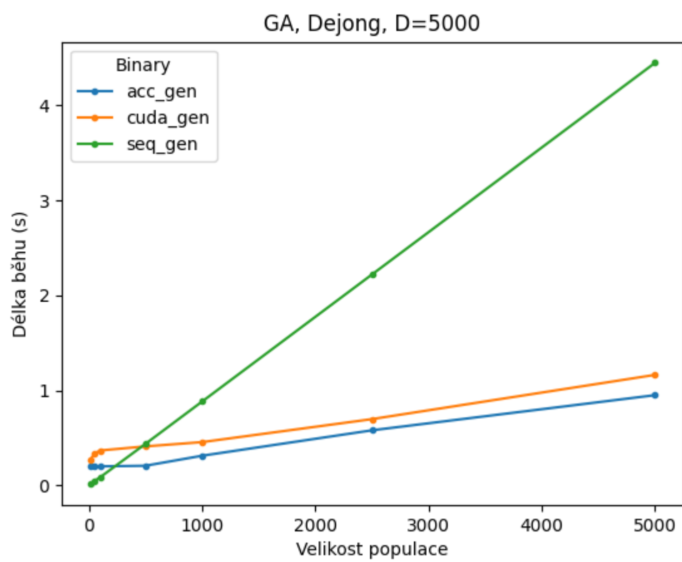

Obrázek 25 GA, Dejong 1st, dimenze = 1000

Sekvenční algoritmus zaostává již od populace 1000 jedinců za OpenACC a zhruba od 1200 i za CUDA. Při populaci 5000 je sekvenční algoritmus až 4x pomalejší než OpenACC.



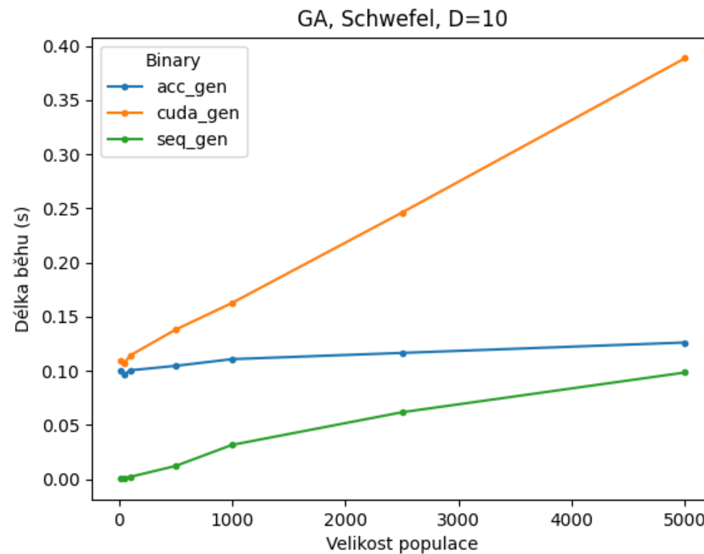

Obrázek 26 GA, Dejong 1st, dimenze = 2500

Sekvenční algoritmus zaostává již při populaci asi 800 ze oběma paralelními implementacemi asi 4x.



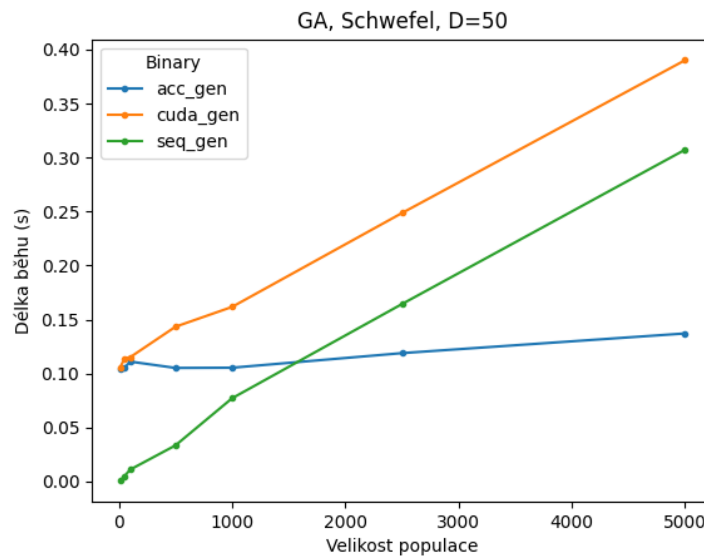

Obrázek 27 GA, Dejong 1st, dimenze = 5000

### 8.3.2 Schwefel



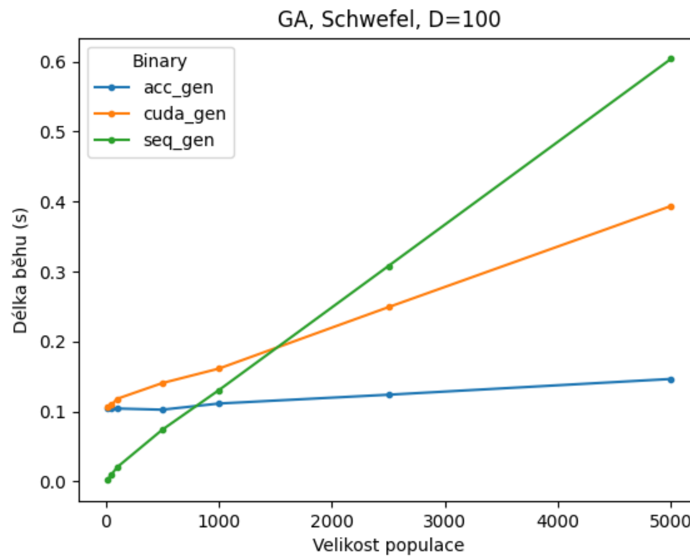

Obrázek 28 GA, Schwefel, dimenze = 10

Při takto nízké dimenzi je sekvenční algoritmus opět nejrychlejší. S vyšší populací se však rychlosti OpenACC přibližuje mnohem rychleji než na Obrázek 21. CUDA je opět značně pomalejší.



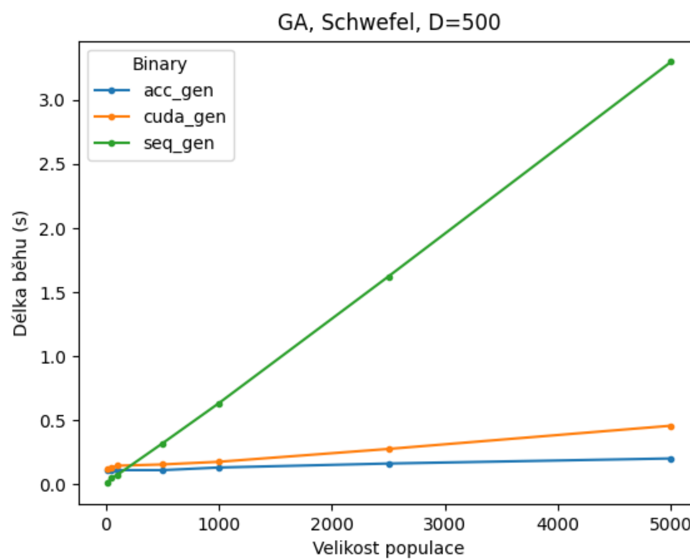

Obrázek 29 GA, Schwefel, dimenze = 50

OpenACC je zde rychlejší již při populaci asi 1500 jedinců. CUDA a sekvenční algoritmus běží více méně s konstantní rozestupem přes celou škálu populací.



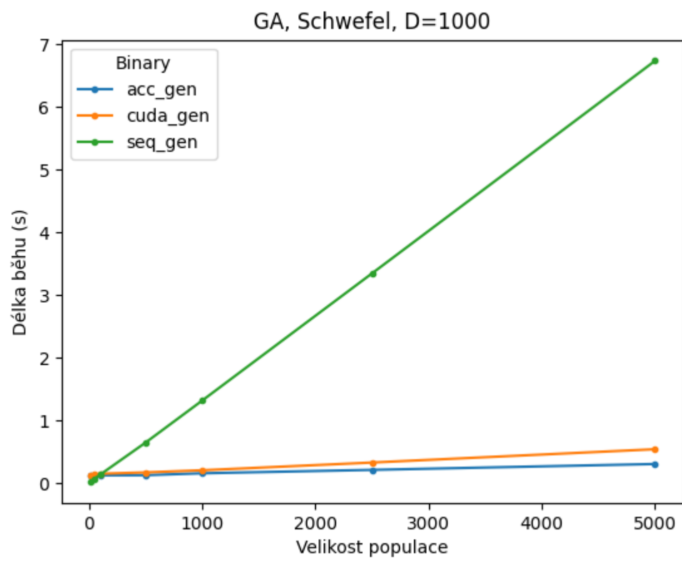

Obrázek 30 GA, Schwefel, dimenze = 100

OpenACC je než sekvenční algoritmus rychlejší již při asi 900 jedincích, CUDA pak při asi 1500 jedincích.

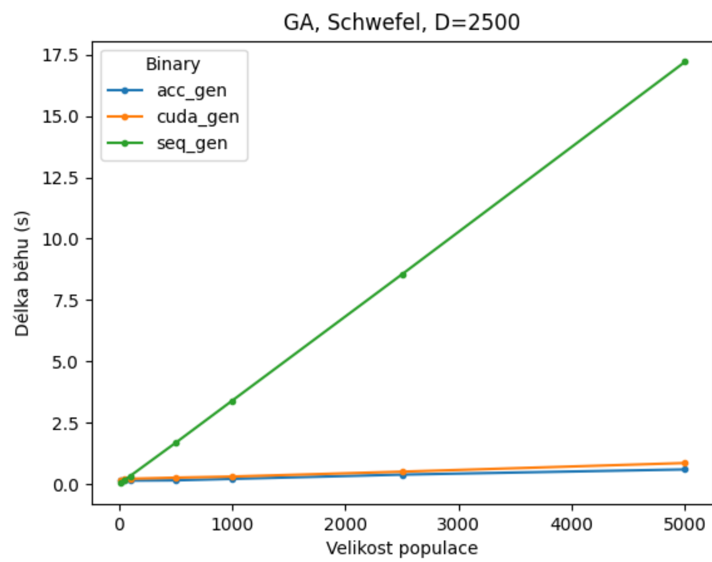



Obrázek 31 GA, Schwefel, dimenze = 500

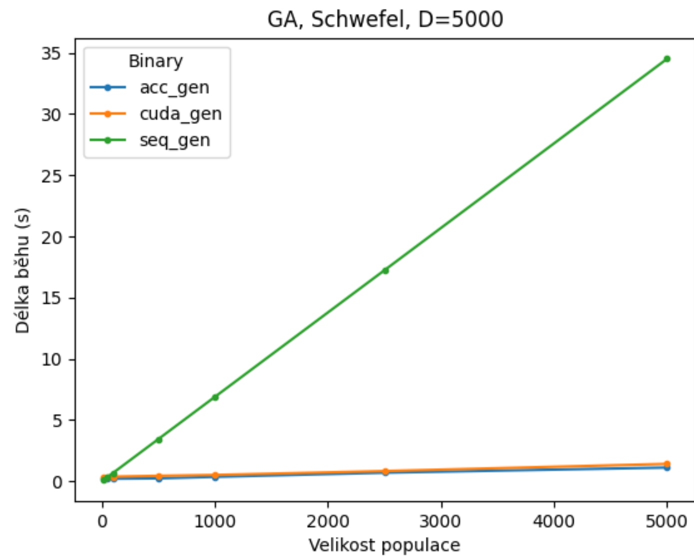
Oba paralelní algoritmy jsou rychlejší již při populaci asi 150 jedinců. OpenACC pak běží asi 2x rychleji než CUDA.




Obrázek 32 GA, Schwefel, dimenze = 1000




Obrázek 33 GA, Schwefel, dimenze = 2500

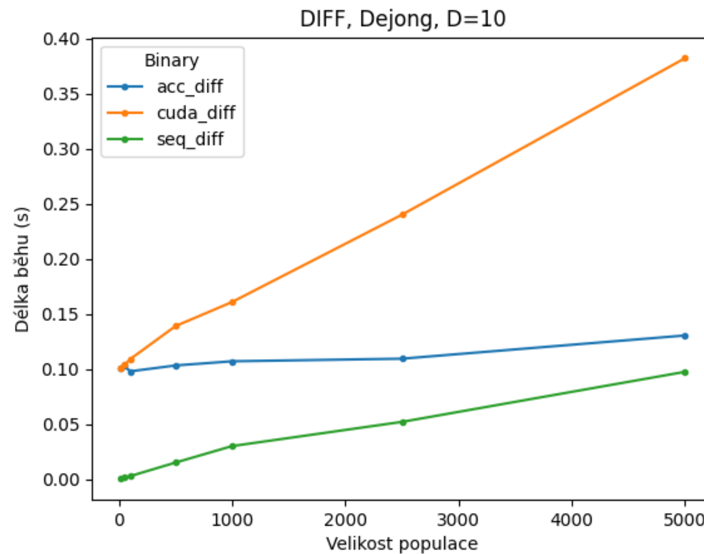



Obrázek 34 GA, Schwefel, dimenze = 5000

Se zvyšující se dimenzí nabírají paralelní implementace náskok a při dimenzi 5000 se dostávají až na 35násobné zrychlení.

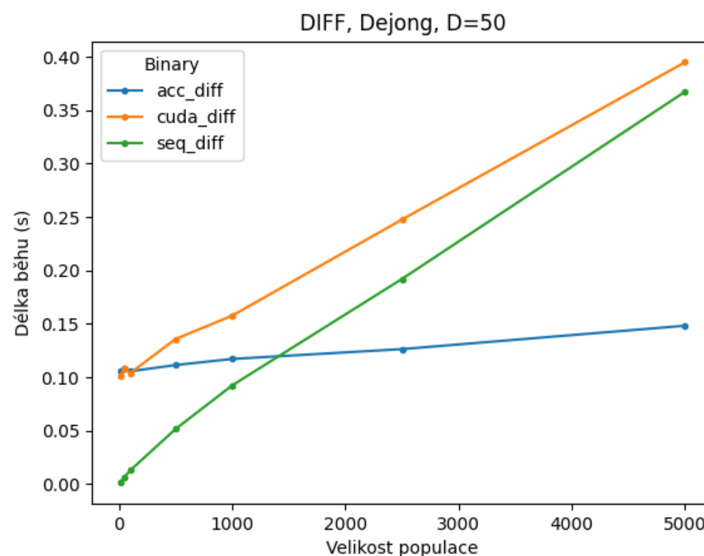
## 8.4 Diferenciální evoluce

### 8.4.1 Dejong



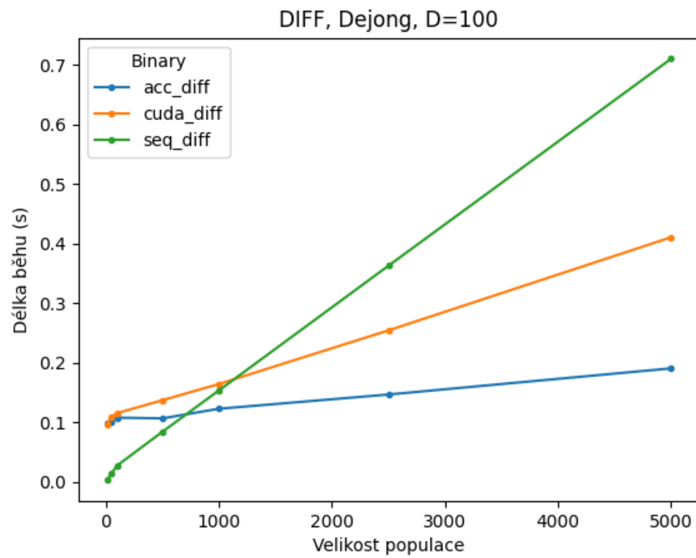

Obrázek 35 Diferenciální evoluce, Dejong 1st, dimenze = 10

Sekvenční algoritmus běží zdánlivě rychleji. S rostoucí populací však rychlost velmi klesá a OpenACC je při populaci 5000 jedinců pouze lehce pomalejší. CUDA implementace výrazně zaostává za oběma implementacemi.



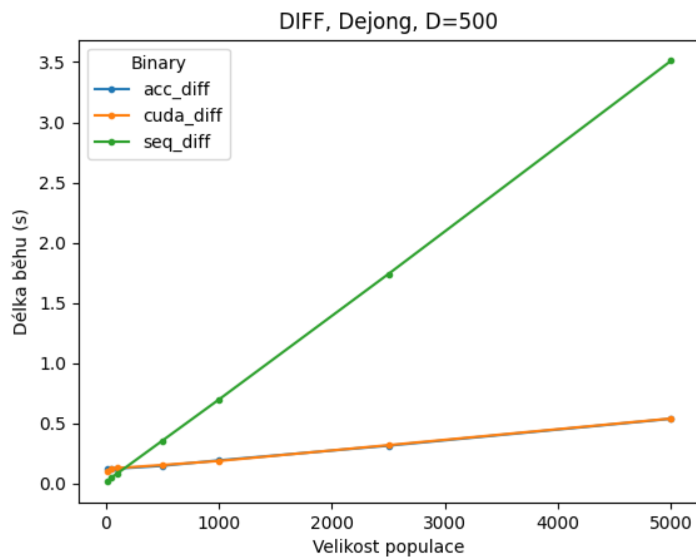

Obrázek 36 Diferenciální evoluce, Dejong 1st, dimenze = 50

CUDA stále zaostává za ostatními algoritmy. OpenACC však sekvenční algoritmus předbíhá již při asi 1300 jedincích a při 5000 je již více jak 2x rychlejší.



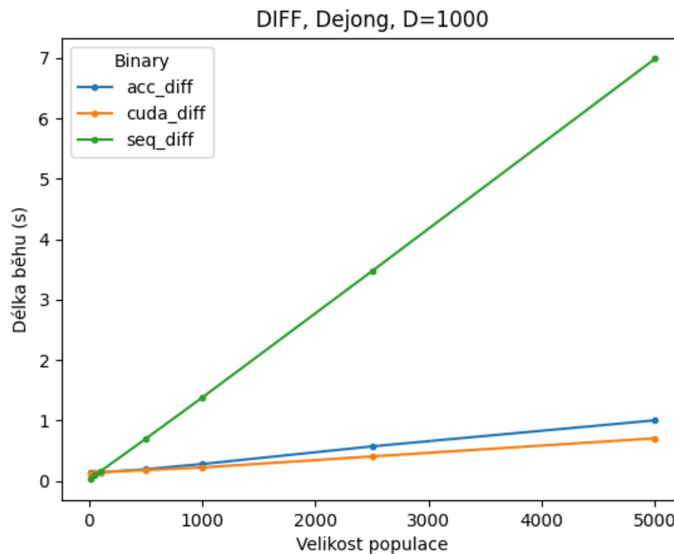

Obrázek 37 Diferenciální evoluce, Dejong 1st, dimenze = 100

CUDA sekvenční algoritmus předbíhá při asi 1100 jedincích a OpenACC při 800.



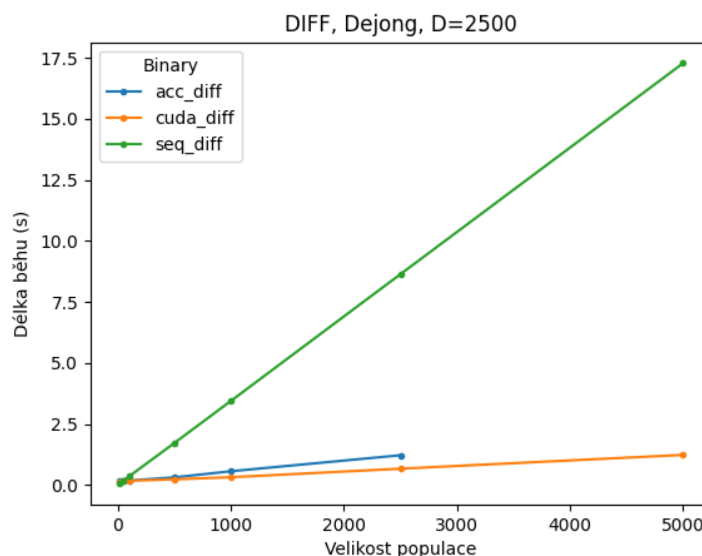

Obrázek 38 Diferenciální evoluce, Dejong 1st, dimenze = 500

Zde CUDA a OpenACC poráží sekvenční algoritmus již při 100 jedincích a poté běží téměř totožnou rychlostí.



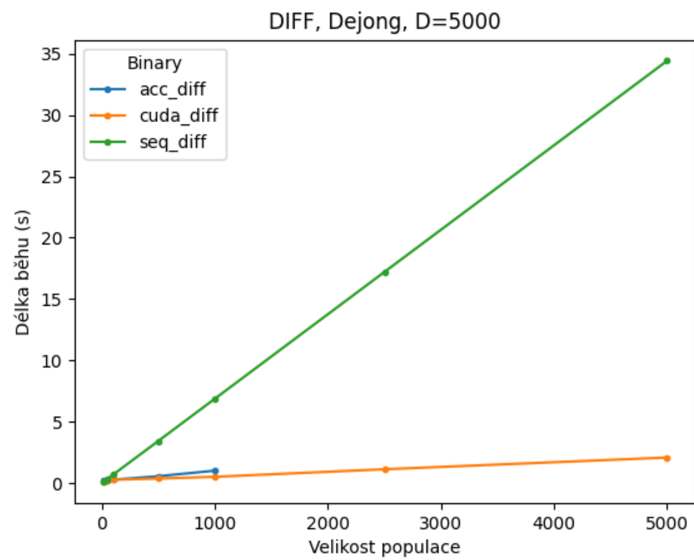

Obrázek 39 Diferenciální evoluce, Dejong 1st, dimenze = 1000

CUDA neběží nejrychleji od nejmenších populací, ale postupně nabírá na rychlosti a při populaci 5000 je asi o 20 % rychlejší než OpenACC a asi 7x rychlejší než sekvenční algoritmus.




Obrázek 40 Diferenciální evoluce, Dejong 1st, dimenze = 2500

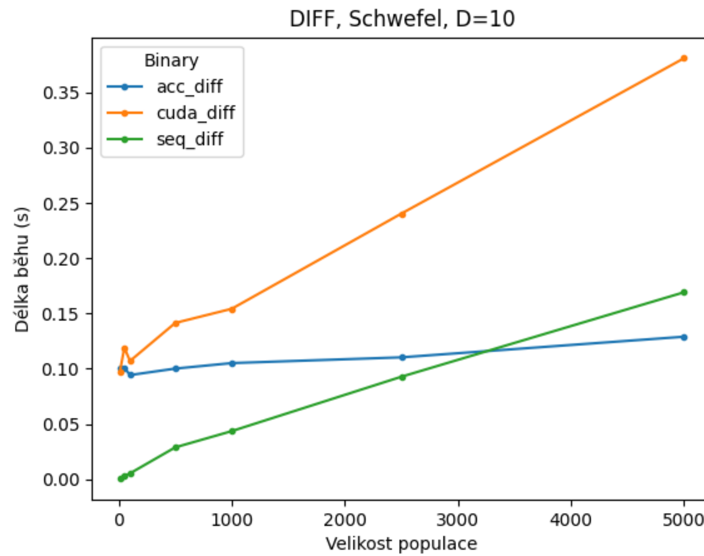
OpenACC začíná narážet na paměťovou náročnost implementace. Z toho důvodu při vyšších populacích a dimenzích dochází na GPU paměť a algoritmus tak v dané konfiguraci nemůže běžet.




Obrázek 41 Diferenciální evoluce, Dejong 1st, dimenze = 5000

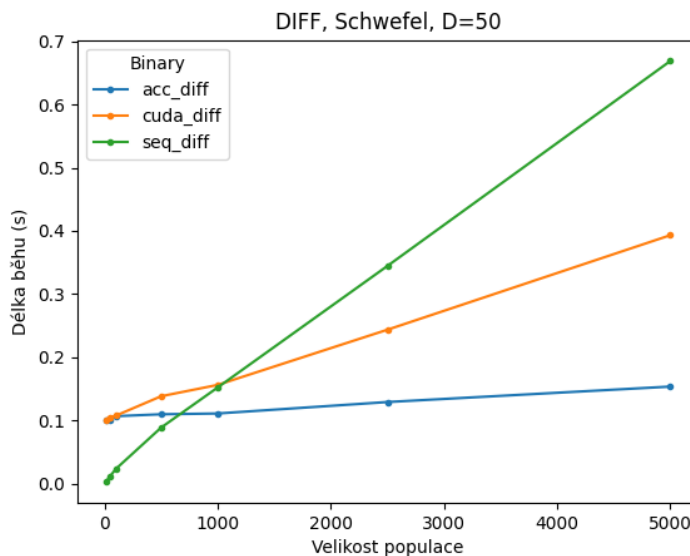
OpenACC opět dochází paměť. CUDA dosahuje asi 16násobného zrychlení oproti sekvenčnímu algoritmu.

### 8.4.2 Schwefel



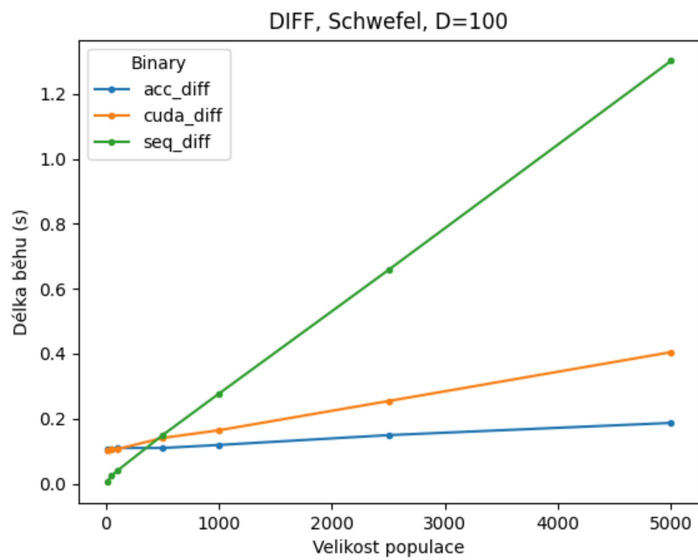

Obrázek 42 Diferenciální evoluce, Schwefel, dimenze = 10

OpenACC je již při 10 dimenzích s populací asi 3200 jedinců rychlejší než sekvenční algoritmus. CUDA běží nejpomaleji.

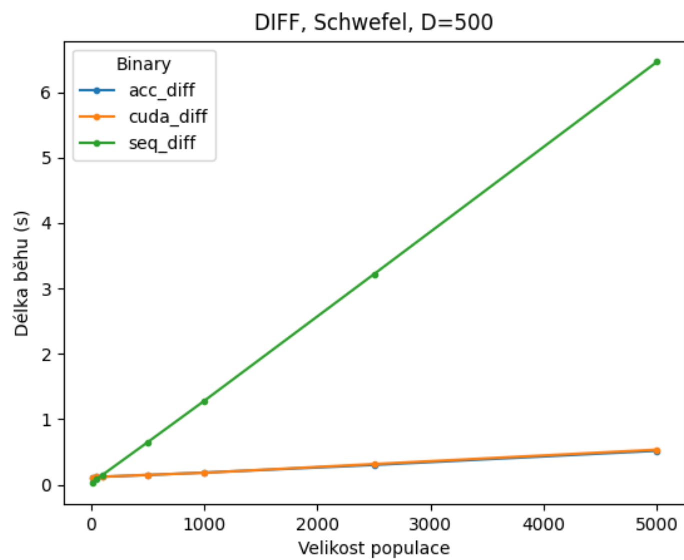



Obrázek 43 Diferenciální evoluce, Schwefel, dimenze = 50

Oba paralelní algoritmy běží rychleji než sekvenční verze od 800 a 1100 jedinců. OpenACC běží rychleji než CUDA.

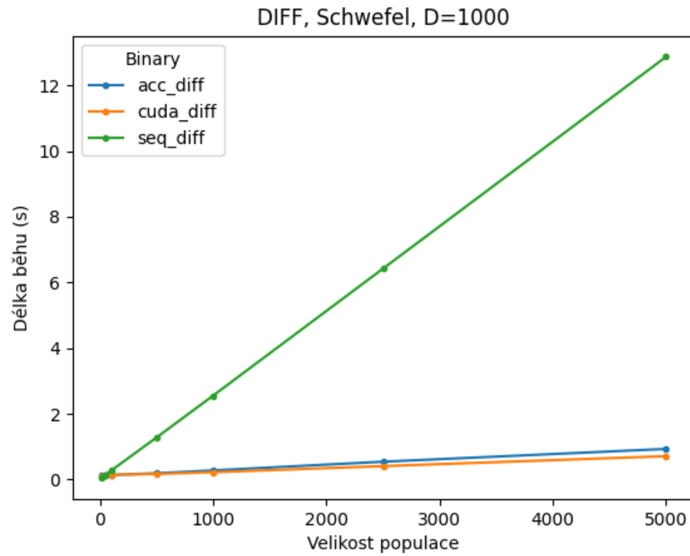



Obrázek 44 Diferenciální evoluce, Schwefel, dimenze = 100



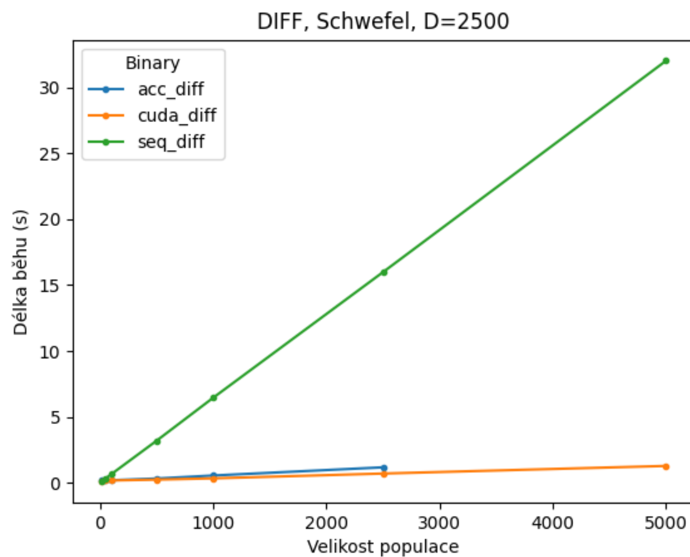

Obrázek 45 Diferenciální evoluce, Schwefel, dimenze = 500

CUDA a OpenACC běží téměř totožnou rychlostí.



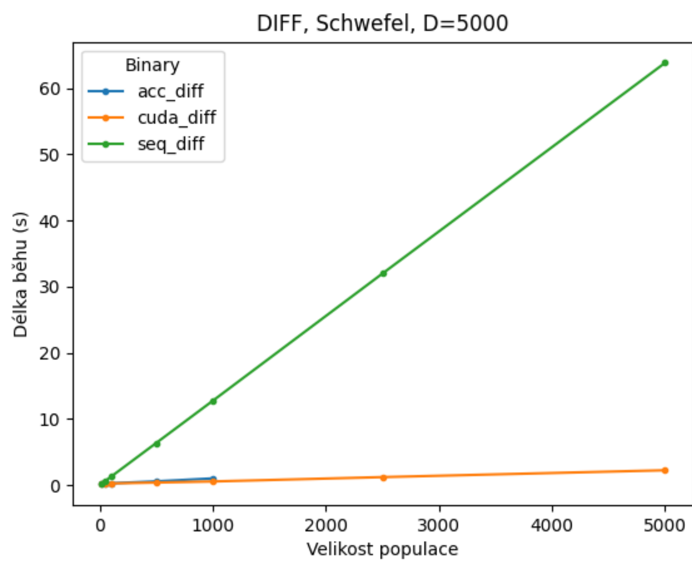

Obrázek 46 Diferenciální evoluce, Schwefel, dimenze = 1000

CUDA začíná běžet rychleji než OpenACC.




Obrázek 47 Diferenciální evoluce, Schwefel, dimenze = 2500

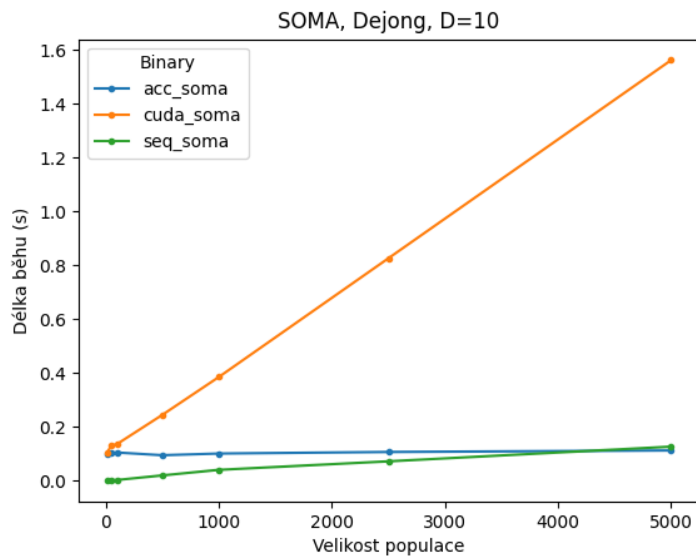
OpenACC dochází paměť na GPU a CUDA běží mnohonásobně rychleji než sekvenční algoritmus.




Obrázek 48 Diferenciální evoluce, Schwefel, dimenze = 5000

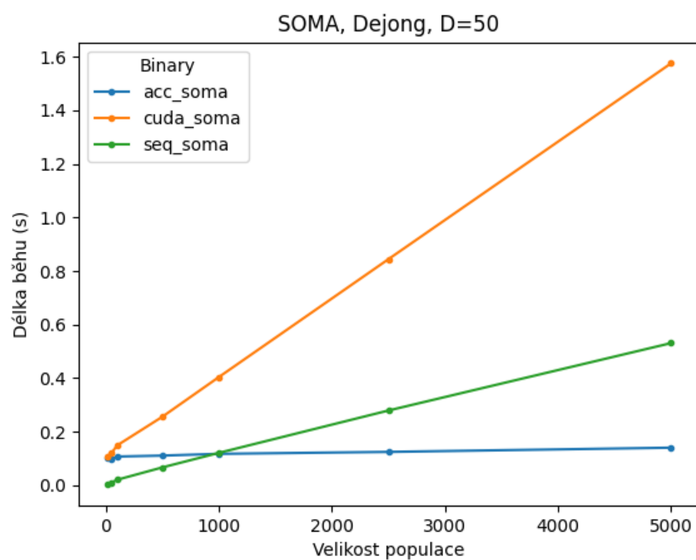
## 8.5 SOMA

### 8.5.1 Dejong



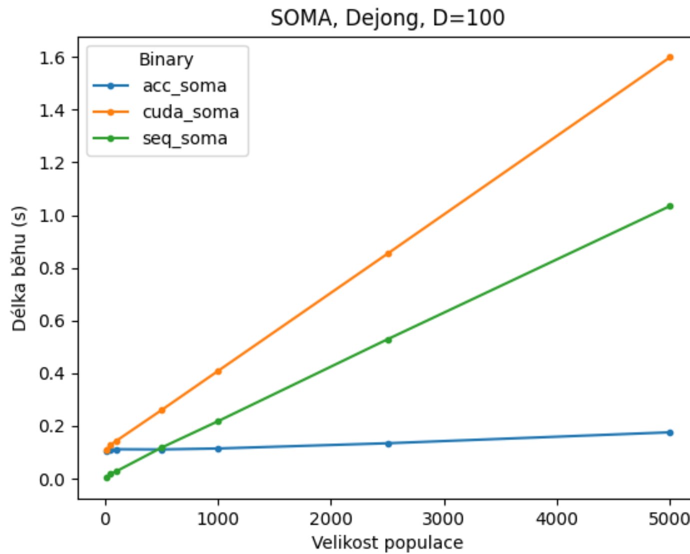

Obrázek 49 SOMA, Dejong 1st, dimenze = 10

CUDA je již od malých populací značně pomalejší. Sekvenční algoritmus při malé populaci běží rychleji, při populaci 5000 je nejrychlejší OpenACC.



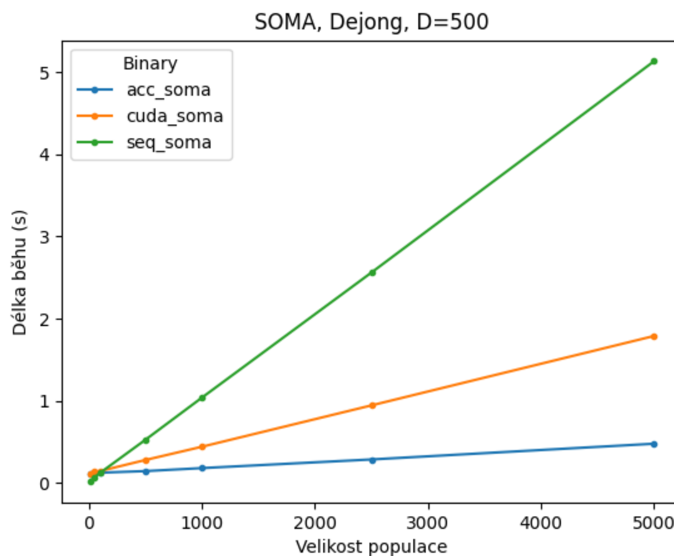

Obrázek 50 SOMA, Dejong 1st, dimenze = 50

OpenACC je rychlejší než sekvenční algoritmus již při 1000 jedincích. CUDA stále výrazně zaostává.



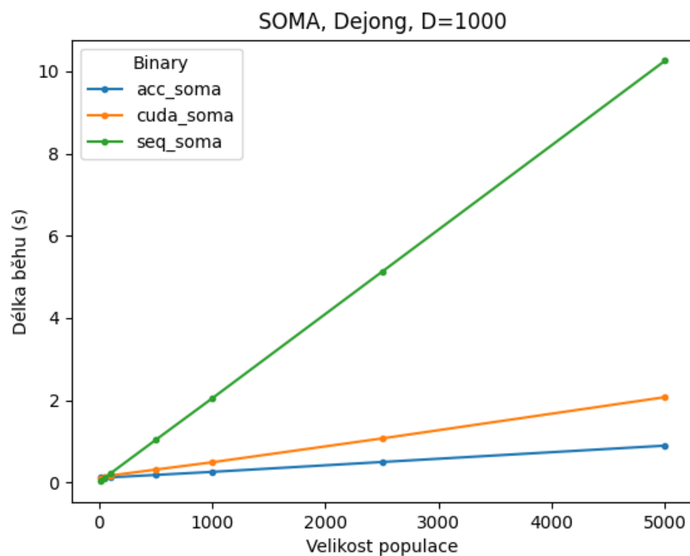

Obrázek 51 SOMA, Dejong 1st, dimenze = 100

CUDA se přibližuje rychlosti sekvenčního algoritmu, ale je stále pomalejší.

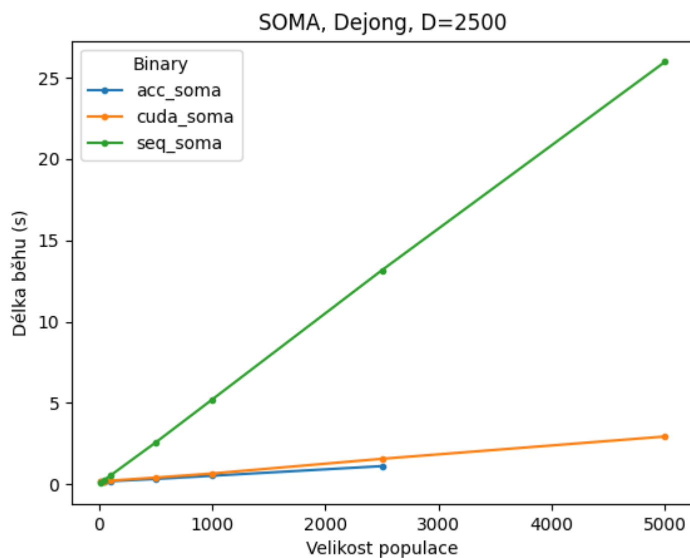



Obrázek 52 SOMA, Dejong 1st, dimenze = 500

CUDA je rychlejší než sekvenční algoritmus již od 200 jedinců, ale na OpenACC je stále více jak 2x rychlejší.

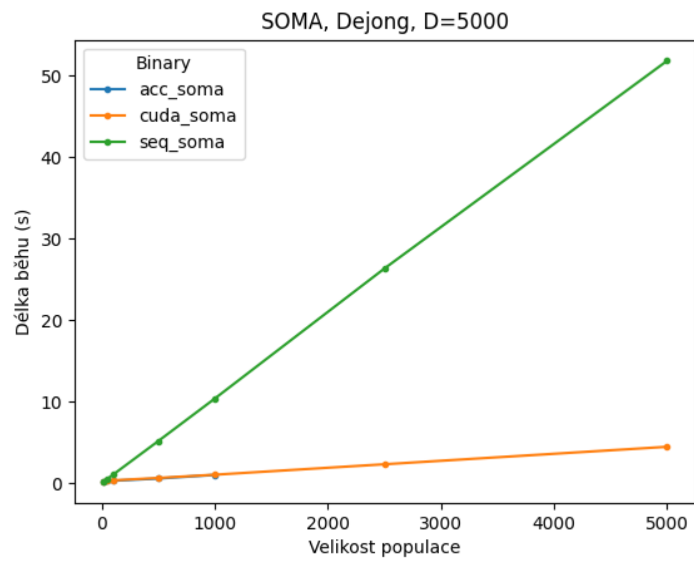



Obrázek 53 SOMA, Dejong 1st, dimenze = 1000



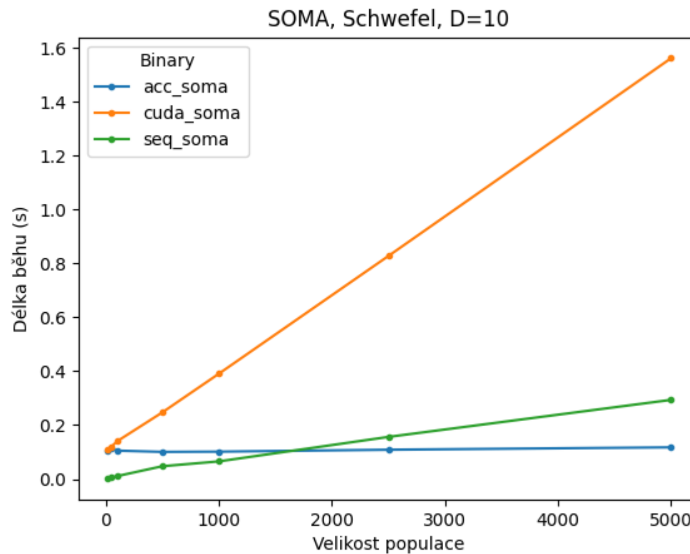

Obrázek 54 SOMA, Dejong 1st, dimenze = 2500

OpenACC běží rychleji než CUDA, ale při rostoucí populaci dojde k nedostatku paměti na GPU. CUDA je asi 8x rychlejší než sekvenční algoritmus.



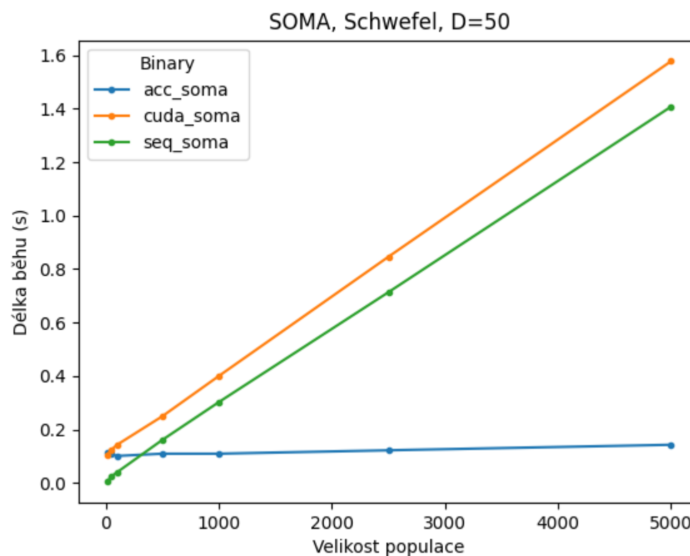

Obrázek 55 SOMA, Dejong 1st, dimenze = 5000

### 8.5.2 Schwefel



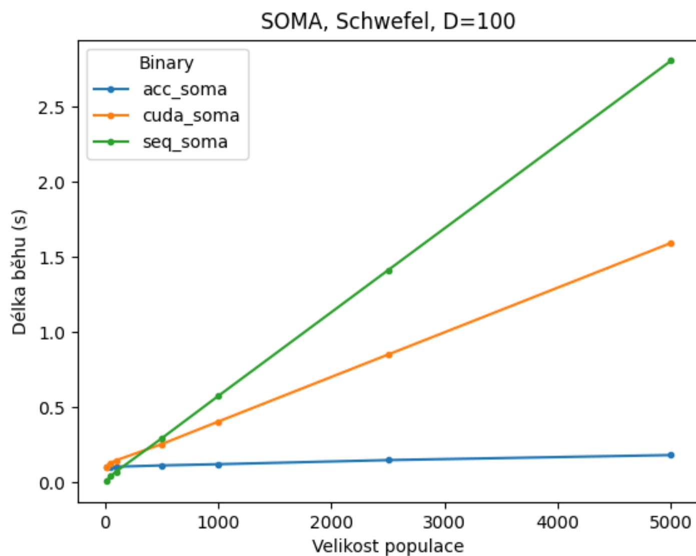

Obrázek 56 SOMA, Schwefel, dimenze = 10

OpenACC je rychlejší než sekvenční algoritmus od asi 1500 jedinců. CUDA běží mnohonásobně pomaleji.



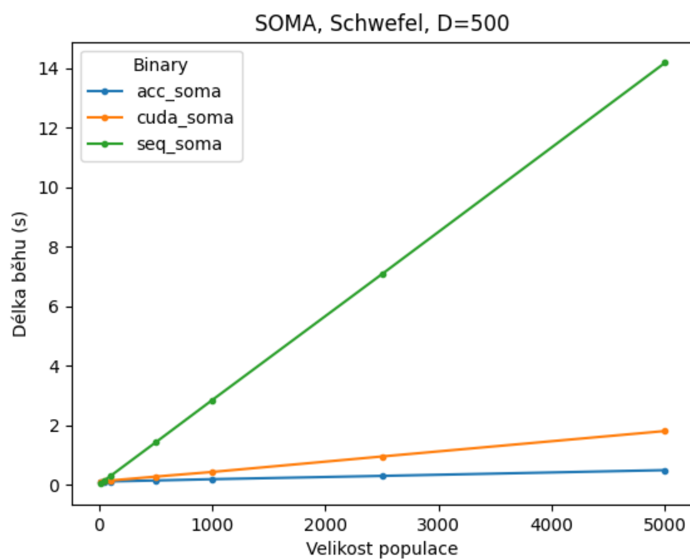

Obrázek 57 SOMA, Schwefel, dimenze = 50

CUDA a s ekvenční algoritmus se rychlostí velice podobají, ale OpenACC je zdaleka nejrychlejší.



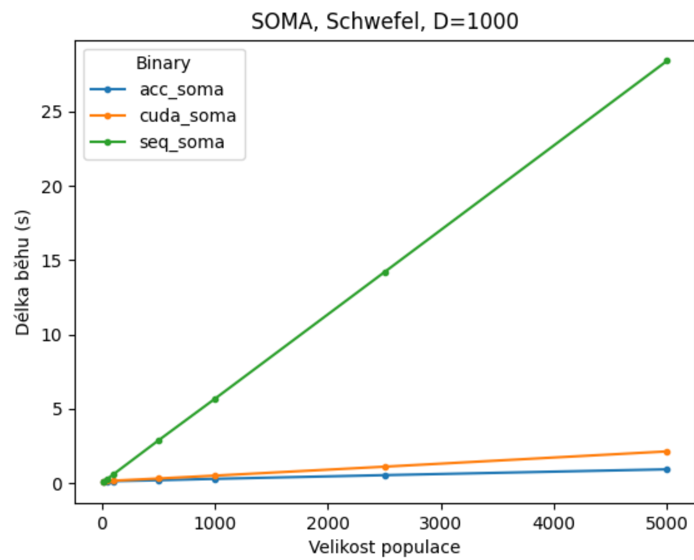

Obrázek 58 SOMA, Schwefel, dimenze = 100

CUDA poráží sekvenční algoritmus od asi 400 jedinců.

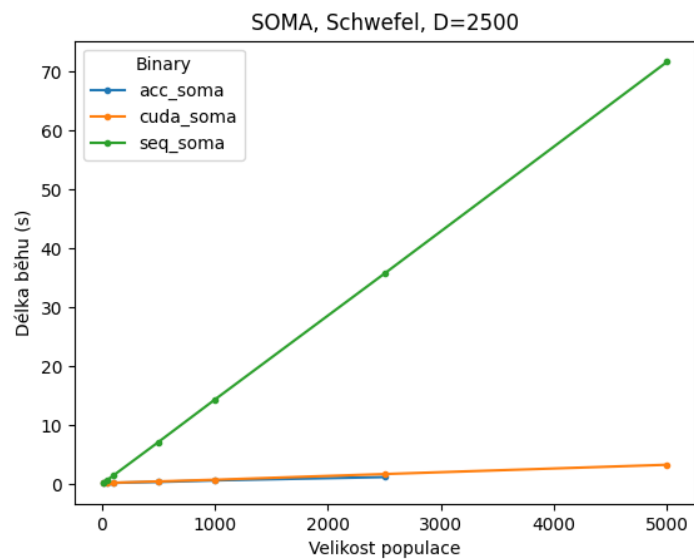



Obrázek 59 SOMA, Schwefel, dimenze = 500

Sekvenční algoritmus zaostává za paralelními verzemi. OpenACC je více jak 3x rychlejší než CUDA.

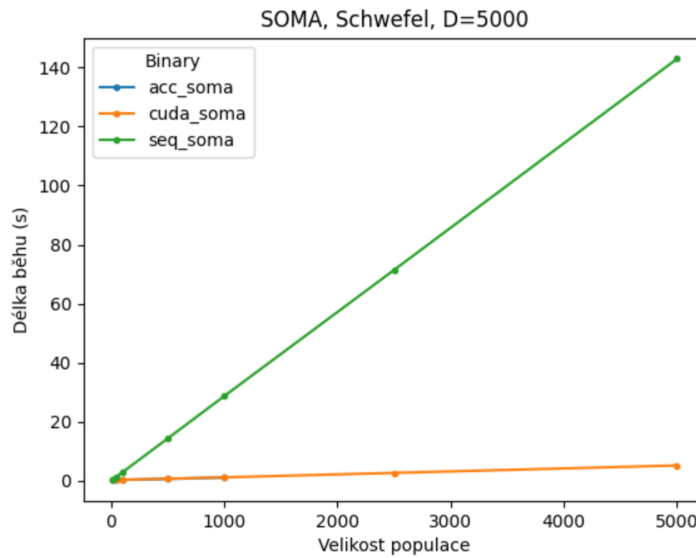



Obrázek 60 SOMA, Schwefel, dimenze = 1000




Obrázek 61 SOMA, Schwefel, dimenze = 2500

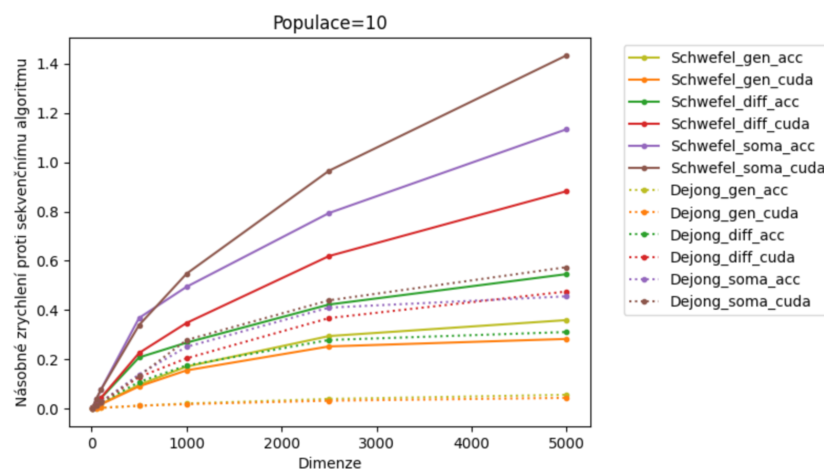
OpenACC dochází paměť, přestože je při populaci 2500 jedinců asi o 40 % rychlejší.




Obrázek 62 SOMA, Schwefel, dimenze = 5000

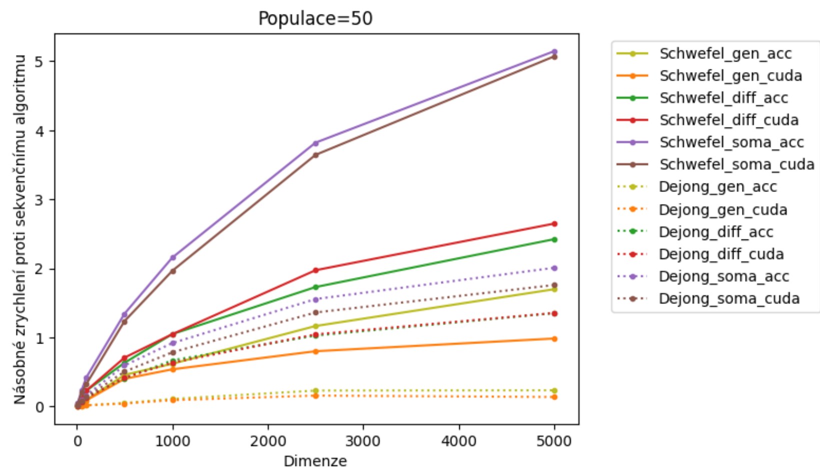
## 8.6 Vliv obtížnosti účelové funkce

Následují grafy ukazují zrychlení paralelních verzí v poměru k sekvenční verzi. Tečkovanou čarou jsou značeny zrychlení při použití účelové funkce Dejong. Přerušovanou čarou jsou vyznačeny zrychlení při použití účelové funkce Schwefel. Funkce Schwefel je výpočetně značně obtížnější než funkce Dejong.

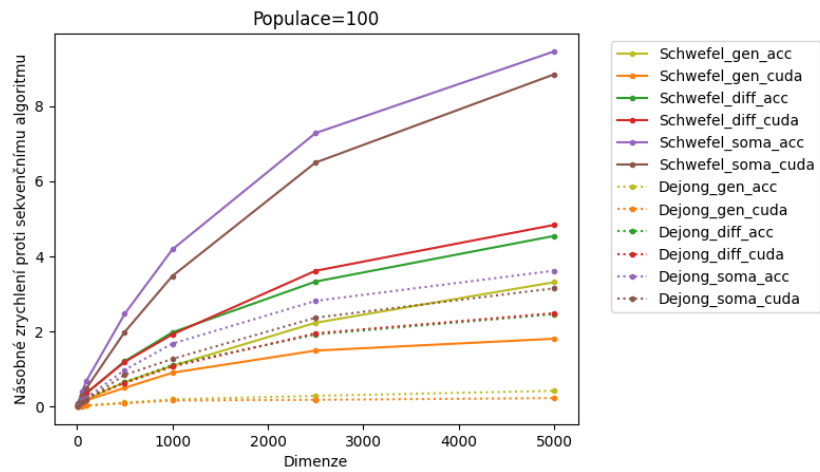



Obrázek 63 Zrychlení při populaci 10

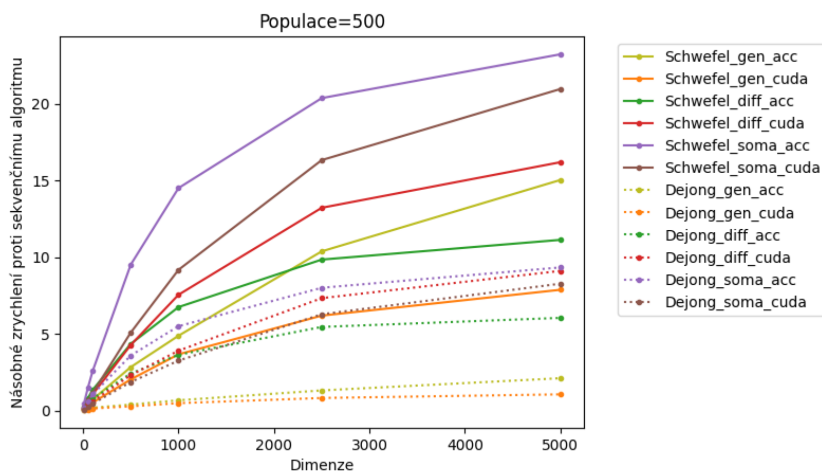
Již při takto malé populaci má paralelizace větší vliv na funkci Schwefel a dosahuje většího zrychlení. V tomto případě jsou však stále implementace ve většině případů pomalejší.



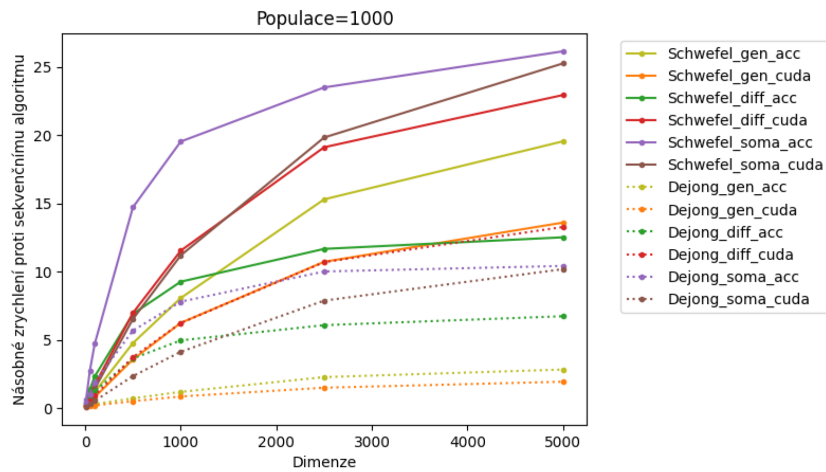

Obrázek 64 Zrychlení při populaci 50



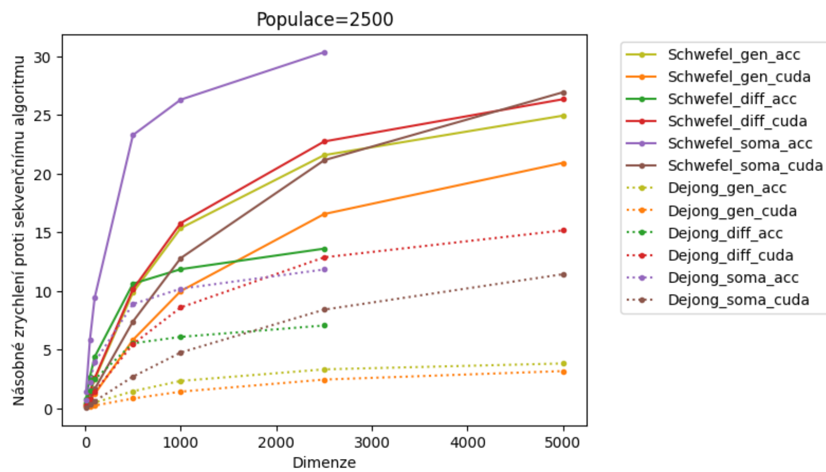

Obrázek 65 Zrychlení při populaci 100



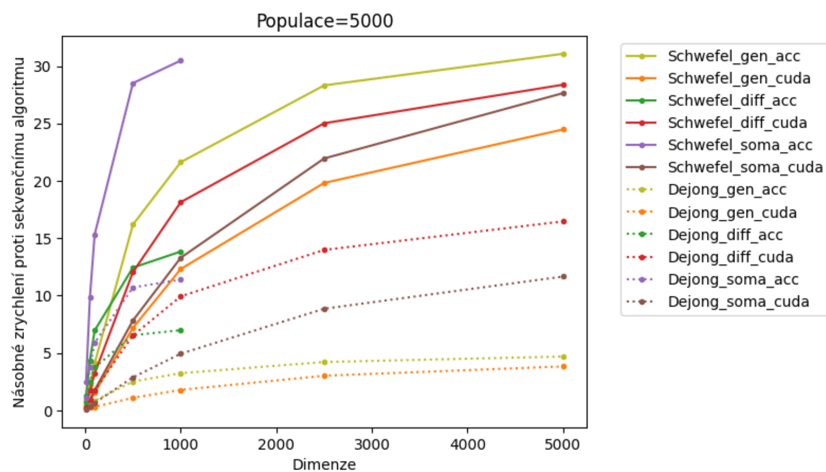

Obrázek 66 Zrychlení při populaci 500




Obrázek 67 Zrychlení při populaci 1000



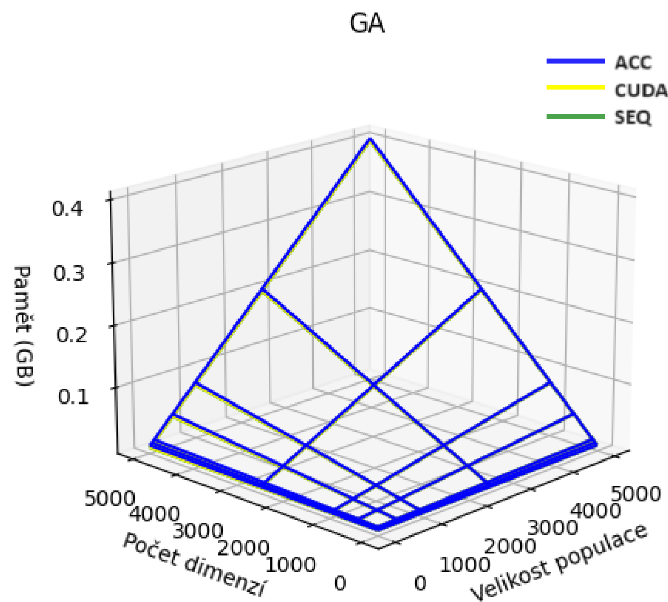

Obrázek 68 Zrychlení při populaci 2500




Obrázek 69 Zrychlení při populaci 5000

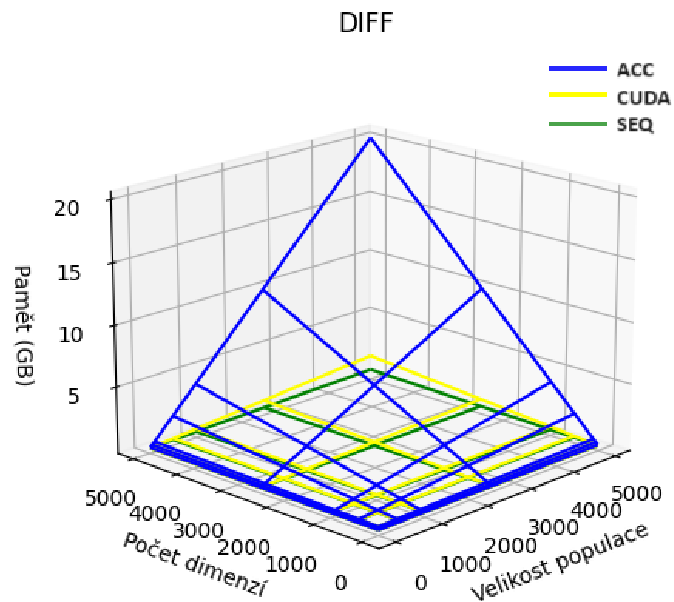
## 8.7 Paměťové nároky

Kompletní datový soubor obsahující paměťové nároky spouštěných konfigurací, ze kterého vycházejí následující grafy, je k nahlédnutí v Příloze PIII.



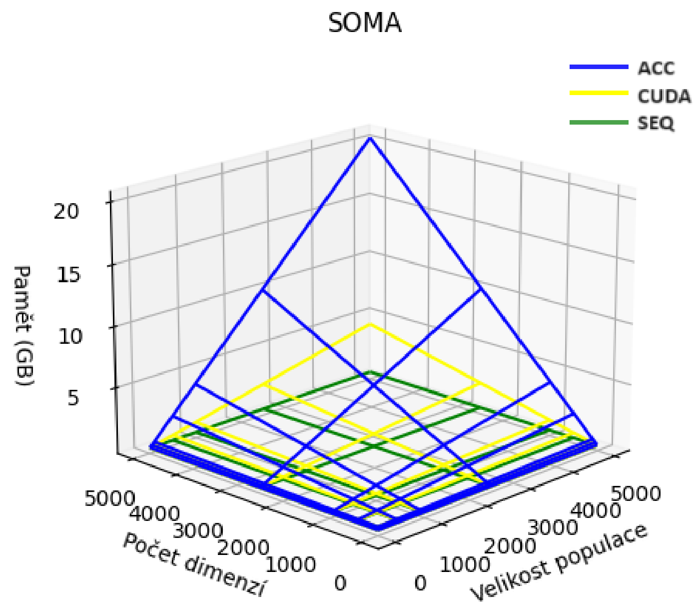

Obrázek 70 Paměťové požadavky GA

Implementace GA má pro všechny tři algoritmy velmi podobnou paměťovou náročnost. Rozdílem jsou pouze identifikátory a jiné proměnné pro generování náhodných čísel, a pár pomocných proměnných u CUDA implementace. Jelikož se paralelizují pouze jedinci a výpočet účelové funkce, nejsou třeba žádná pomocná pole pro udržování mezivýpočtů.




Obrázek 71 Paměťové požadavky DE

DE využívá náhodná čísla v rámci smyčky křížení, což v paralelní implementaci vyžaduje $velikost\ populace * dimenze$ vláken generování náhodných čísel. V CUDA je pro každé takové vlákno potřeba pouze 48 b paměti. V OpenACC je to však 792 bytů. Kombinace mnoha vláken a velké paměťové náročnosti generování náhodných čísel způsobí u OpenACC výše viditelný prudký růst paměťové náročnosti implementace. CUDA implementace navíc využívá pomocná pole pro přenesení indexů cílových vektorů mezi kernely.




Obrázek 72 Paměťové požadavky SOMA

SOMA, podobně jako DE, generuje náhodná čísla ve vnořené smyčce při generování PRT vektoru. OpenACC kvůli tomu opět trpí na velké paměťové požadavky při generování náhodných čísel. Oproti sekvenční implementaci vyžadují paralelní implementace rozšíření polí využívaných pro ukládání kroků v rámci migrace. Sekvenčnímu algoritmu stačí jedno pole velikosti $dimenze$ a ukládání si nejlepších hodnot do pár proměnných. Paralelní implementace potřebují pro každého jedince pole o velikosti $počet\ kroků * dimenze$. Dohromady tedy až $velikost\ populace * počet\ kroků * dimenze$. Pro ukládání mezivýsledků účelové funkce je přidáno ještě jedno pole o velikosti $velikost\ populace * \ počet\ kroků$ Stejně tak je třeba rozšířit pole pro ukládání PRT vektoru z velikosti $dimenze$ na velikost $velikost\ populace * dimenze$.

## 8.8 Zhodnocení

Paralelizace se ukázala u všech tří zvolených algoritmů, jako velmi vhodný nástroj pro urychlení jejich běhů. Ve většině případů umožnila mnohonásobné zrychlení, obzvláště pak při problémech s větším počtem dimenzí. Jediné, kdy paralelizace zaostávala, byly problémy s malým počtem dimenzí, nebo pokud byla nastavena velice malá velikost populace. V tomto případě je dobré myslet na celkový čas běhu, který je ve všech implementacích v rámci sekund a paralelizace tedy ani nemá velký prostor k urychlení. Nastavení malé

populace také může mít vliv na kvalitu nalezeného řešení a je tak při rozhodování o nastavení algoritmu brát potaz i na toto.

Paralelizace účelové funkce se také projevila jako vhodný nástroj. Již od malé populace dokázala urychlit běh programu a implementace paralelizace byly v obou použitých knihovnách jednoduché.

OpenACC implementace obecně běžela ze všech nejrychleji. Za CUDA implementací zaostávala pouze u DE při středně velkých a velkých dimenzích. Toto považuji za velmi dobré, vzhledem k relativní jednoduchosti práce s OpenACC ve srovnání s CUD-ou, která vyžaduje více méně kompletní přepis algoritmu, a se sekvenční verzí si kód zachovává velmi malou podobnost.

Bohužel se u OpenACC ukázala problematika velké paměťové náročnosti, při generování náhodných čísel, které jsou ovšem základem evolučních algoritmů a značně tak omezuje, na jaké problémy se dá dobře využít. Vyžadování 20 GB GPU paměti je pro běžný PC téměř nemožné a spouštění je tak omezeno na profesionální GPU a serverová řešení, která ale stojí mnohem více. Pokud by tedy někdo potřeboval řešit vetší problémy, u kterých jsou generovány náhodná čísla hlouběji než na první úrovni smyčky, nezbyde, než využít CUDU a provést náročnější implementaci zvoleného algoritmu.

Přestože CUDA implementace celkově nepřekonala OpenACC, je její rychlost stále velmi dobrá a neběží o tolik pomaleji, aby se nad ní nedalo uvažovat.

Zajímavým řešením s náhodnými čísli by mohla pak být například implementace, která kombinuje oba přístupy a vybírá si CUDU nebo OpenACC na základě zadaných parametrů algoritmu.

# ZÁVĚR

Tato práce byla zaměřena na poukázání možností využití GPU paralelizace evolučních algoritmů. Vybral jsem proto několik, běžně využívaných algoritmů, a to Genetický algoritmus, Diferenciální evoluci a algoritmus SOMA. Tyto algoritmy jsem nastudoval, popsal a vytvořil sekvenční implementaci. Na základě sekvenční implementace jsem identifikoval kritická místa těchto algoritmů vhodná pro paralelizaci a navrhl jejich paralelizaci.

Poté jsem nastudoval knihovny CUDA o OpenACC, které jsou moderními knihovnami pro paralelizaci na GPU, a popsal jejich hlavní prvky, které mohu dále využít pro vytvoření paralelních implementací zvolených algoritmů.

Na základě vytvořeného návrhu paralelizace a znalostí konstruktů knihoven jsem vytvořil paralelní verze všech 3 algoritmů. Knihovna OpenACC vyžadovala mnohem menší úpravy kódu, ale sama nepodporuje generování náhodných čísel, což ve výsledku značně ovlivnilo její schopnosti. CUDA oproti tomu náhodná čísla podporuje, ovšem struktura kódu je úplně jiná než u sekvenčního algoritmus a byla tedy značně náročnější na implementaci.

Vytvořené paralelní implementace jsem otestoval a vyhodnotil, že paralelizace není vhondá pouze pro problémy s malou dimenzí a při malém nastavení velikosti populace algoritmu. Při problémech středních a větších rozměrů poskytují paralelní implementace mnohonásobné zrychlení, za cenu paměťových požadavků.

# SEZNAM POUŽITÉ LITERATURY

[1] VIKHAR, Pradnya A. Evolutionary algorithms: A critical review and its future prospects. Online. 2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC). 2016, s. 261-265. ISBN 978-1-5090-0467-6. Dostupné z: https://doi.org/10.1109/ICGTSPICC.2016.7955308. [cit. 2023-03-02].

[2] ZELINKA, Ivan. Evoluční výpočetní techniky: principy a aplikace. Praha: BEN - technická literatura, 2009. ISBN 978-80-7300-218-3.

[3] KADAVY, Tomas; PLUHACEK, Michal; SENKERIK, Roman a VIKTORIN, Adam. Boundary Strategies for Self-organizing Migrating Algorithm Analyzed Using CEC'17 Benchmark. Swarm, Evolutionary, and Memetic Computing and Fuzzy and Neural Computing. Communications in Computer and Information Science. 2020, s. 58-69. ISBN 978-3-030-37837-0. Dostupné z: https://doi.org/10.1007/978-3-030-37838-7_6.

[4] ADAM, Stavros P.; ALEXANDROPOULOS, Stamatios-Aggelos N.; PARDALOS, Panos M. a VRAHATIS, Michael N. No Free Lunch Theorem: A Review. Online. In: DEMETRIOU, Ioannis C. a PARDALOS, Panos M. (ed.). Approximation and Optimization. Springer Optimization and Its Applications. Cham: Springer International Publishing, 2019, s. 57-82. ISBN 978-3-030-12766-4. Dostupné z: https://doi.org/10.1007/978-3-030-12767-1_5. [cit. 2024-04-11].

[5] SLOWIK, Adam a KWASNICKA, Halina. Evolutionary algorithms and their applications to engineering problems. Online. Neural Computing and Applications. 2020, roč. 32, č. 16, s. 12363-12379. ISSN 0941-0643. Dostupné z: https://doi.org/10.1007/s00521-020-04832-8. [cit. 2023-03-21].

[6] BILAL; PANT, Millie; ZAHEER, Hira; GARCIA-HERNANDEZ, Laura a ABRAHAM, Ajith. Differential Evolution: A review of more than two decades of research. Online. Engineering Applications of Artificial Intelligence. 2020, roč. 90. ISSN 09521976. Dostupné z: https://doi.org/10.1016/j.engappai.2020.103479. [cit. 2024-04-02].

[7] ZELINKA, Ivan, 2004. SOMA — Self-Organizing Migrating Algorithm. In: ONWUBOLU, Godfrey C. a B. V. BABU. New Optimization Techniques in Engineering [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, s. 167-217 [cit.

2023-04-05]. Studies in Fuzziness and Soft Computing. ISBN 978-3-642-05767-0. Dostupné z: doi:10.1007/978-3-540-39930-8_7

[8] KATOCH, Sourabh; CHAUHAN, Sumit Singh a KUMAR, Vijay. A review on genetic algorithm: past, present, and future. Online. Multimedia Tools and Applications. 2021, roč. 80, č. 5, s. 8091-8126. ISSN 1380-7501. Dostupné z: https://doi.org/10.1007/s11042-020-10139-6. [cit. 2023-04-02].

[9] RAZALI, Noraini Mohd a GERAGHTY, John. Genetic Algorithm Performance with Different Selection Strategies in Solving TSP. Online. Proceedings of the World Congress on Engineering 2011 Vol II. ISSN 2078-0966. Dostupné z: https://www.iaeng.org/publication/WCE2011/WCE2011_pp1134-1139.pdf. [cit. 2024-04-15].

[10] Elitism in Evolutionary Algorithms. Online. 2022. Dostupné z: https://www.baeldung.com/cs/elitism-in-evolutionary-algorithms. [cit. 2023-03-11].

[11] AHMAD, Mohamad Faiz et al., 2022. Differential evolution: A recent review based on state-of-the-art works. Alexandria Engineering Journal [online]. 61(5), 3831-3872 [cit. 2023-04-02]. ISSN 11100168. Dostupné z: doi:10.1016/j.aej.2021.09.013

[12] LEON, Miguel a Ning XIONG, 2014. Investigation of Mutation Strategies in Differential Evolution for Solving Global Optimization Problems. In: RUTKOWSKI, Leszek et al., ed. Artificial Intelligence and Soft Computing [online]. Cham: Springer International Publishing, s. 372-383 [cit. 2023-04-02]. Lecture Notes in Computer Science. ISBN 978-3-319-07172-5. Dostupné z: doi:10.1007/978-3-319-07173-2_32

[13] SKANDEROVA, Lenka, 2023. Self-organizing migrating algorithm: review, improvements and comparison. Artificial Intelligence Review [online]. 56(1), 101-172 [cit. 2023-04-05]. ISSN 0269-2821. Dostupné z: doi:10.1007/s10462-022-10167-8

[14] Introduction to Parallel Computing Tutorial. Online. Dostupné z: https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial. [cit. 2024-04-27].

[15] FLYNN, Michael. Flynn's Taxonomy. Online. Encyclopedia of Parallel Computing. 2011, s. 689-697. ISBN 978-0-387-09765-7. Dostupné z: https://doi.org/10.1007/978-0-387-09766-4_2. [cit. 2024-04-27].

[16] BAAGUIGUI, Ramzi. SISD, SIMD, MISD, and MIMD: Easily Explained!. Online. 2023. Dostupné z: https://medium.com/@ramzi.baaguigui1/sisd-simd-misd-and-mimd-easily-explained-c668d5382be6. [cit. 2024-04-27].

[17] NVIDIA. CUDA C++ Programming Guide. Online. 2007, 2024-03-07. Dostupné z: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. [cit. 2024-03-19].

[18] CUDA – Streaming Multiprocessors. Online. 2020. Dostupné z: http://thebeard-sage.com/cuda-streaming-multiprocessors/. [cit. 2024-04-27].

[19] NVIDIA. CUDA GPUs - Compute Capability | NVIDIA Developer. Online. 2024. Dostupné z: https://developer.nvidia.com/cuda-gpus. [cit. 2024-03-26].

[20] MAO, Lei. CUDA Stream - Lei Mao's Log Book. Online. 2020. Dostupné z: https://leimao.github.io/blog/CUDA-Stream/. [cit. 2024-03-26].

[21] THE OPENACC ORGANIZATION. Specification | OpenACC. Online. 2011. Dostupné z: https://www.openacc.org/specification. [cit. 2024-03-27].

[22] OpenACC Programming and Best Practices Guide. Online. 2021. Dostupné z: https://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0_0.pdf. [cit. 2024-04-02].

[23] SINGH, Satvir; KAUR, Jaspreet a SINHA, Rashmi Sharan. A Comprehensive Survey on Various Evolutionary Algorithms on GPU. 2014.

[24] GUPTA, Samarth a TAN, Gary. A scalable parallel implementation of evolutionary algorithms for multi-objective optimization on GPUs. Online. 2015 IEEE Congress on Evolutionary Computation (CEC). 2015, s. 1567-1574. ISBN 978-1-4799-7492-4. Dostupné z: https://doi.org/10.1109/CEC.2015.7257074. [cit. 2024-04-30].

[25] SYBERFELDT, Anna a EKBLOM, Tom. A Comparative Evaluation of the GPU vs The CPU for Parallelization of Evolutionary Algorithms Through Multiple Independent Runs. Nternational Journal of Computer Science & Information Technology. 2017, roč. 9, č. 3. ISSN 0975-3826.

[26] VALKOVIČ, Patrik. GPU Parallelization of Evolutionary Algorithms. Diplomová práce, vedoucí Pilát, Martin. Praha: Univerzita Karlova, Matematicko-fyzikální fakulta, Katedra teoretické informatiky a matematické logiky, 2021.

[27] NVIDIA. CuRAND :: CUDA Toolkit Documentation. Online. 2024. Dostupné z: https://docs.nvidia.com/cuda/curand/index.html. [cit. 2024-04-07].

[28] CARLSSON, Johan. Pseudo Random Number Generation by Lightweight Threads. Online. In: OPENACC. OpenACC. 2011. Dostupné z: https://www.openacc.org/blog/pseudo-random-number-generation-lightweight-threads. [cit. 2024-04-07].

[29] NVIDIA. NVIDIA/cccl: CUDA C++ Core Libraries. Online. 2024. Dostupné z: https://github.com/NVIDIA/cccl. [cit. 2024-04-07].

[30] Sphere function. Online. Dostupné z: https://www.sfu.ca/~ssurjano/spheref.html. [cit. 2024-04-30].

[31] Schwefel function. Online. Dostupné z: https://www.sfu.ca/~ssurjano/schwef.html. [cit. 2024-04-30].

# SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

| | |
|---|---|
| GPU | Graphics processing unit |
| CUDA | Compute Unified Device Architecture |
| GPGPU | General-purpose computing on GPUs |
| GA | Genetický algoritmus |
| DE | Diferenciální evoluce |
| SOMA | Self-organizing migrating algorithm |
| CPU | Central processing unit |
| API | Application program interface |
| SIMD | Single instruction, Multiple data |
| SM | Streamovací multiprocesor |

# SEZNAM OBRÁZKŮ

Obrázek 1 Populace ....12
Obrázek 2 Výběr jedinců turnajem [9] ....16
Obrázek 3 Srovnání pravděpodobností u výběru proporcionální ruletou a výběru řazenou ruletou [9] ....18
Obrázek 4 Elitismus [10] ....18
Obrázek 5 Jednobodové křížení ....19
Obrázek 6 Dvoubodové křížení ....20
Obrázek 7 Vícenásobné křížení ....20
Obrázek 8 Jednobodová mutace ....20
Obrázek 9 Příklad chybového vektoru pro 4 parametry a *prt* = 0,3 [7] ....25
Obrázek 10 *PRTVector* a jeho vliv na pohyb jedince [7] ....26
Obrázek 11 CPU a GPU architektura [17] ....29
Obrázek 12 Hierarchie vláken v mřížce [17] ....30
Obrázek 13 Paměťový model [17] ....31
Obrázek 14 Exekuční model OpenACC [22] ....33
Obrázek 15 Ukázka OpenACC výstupu ....34
Obrázek 16 Profil GA ....38
Obrázek 17 Profil DE ....38
Obrázek 18 Profil SOMA ....39
Obrázek 19 De Jong’s 1st function ve třech dimenzích [30] ....47
Obrázek 20 Schwefel function ve třech dimenzích [31] ....47
Obrázek 21 GA, Dejong 1st, dimenze = 10 ....49
Obrázek 22 GA, Dejong 1st, dimenze = 50 ....49
Obrázek 23 GA, Dejong 1st, dimenze = 100 ....50
Obrázek 24 GA, Dejong 1st, dimenze = 500 ....50
Obrázek 25 GA, Dejong 1st, dimenze = 1000 ....51
Obrázek 26 GA, Dejong 1st, dimenze = 2500 ....51
Obrázek 27 GA, Dejong 1st, dimenze = 5000 ....52
Obrázek 28 GA, Schwefel, dimenze = 10 ....53
Obrázek 29 GA, Schwefel, dimenze = 50 ....53
Obrázek 30 GA, Schwefel, dimenze = 100 ....54
Obrázek 31 GA, Schwefel, dimenze = 500 ....54

Obrázek 32 GA, Schwefel, dimenze = 1000 ....................55
Obrázek 33 GA, Schwefel, dimenze = 2500 ....................55
Obrázek 34 GA, Schwefel, dimenze = 5000 ....................56
Obrázek 35 Diferenciální evoluce, Dejong 1st, dimenze = 10 ....................57
Obrázek 36 Diferenciální evoluce, Dejong 1st, dimenze = 50 ....................57
Obrázek 37 Diferenciální evoluce, Dejong 1st, dimenze = 100 ....................58
Obrázek 38 Diferenciální evoluce, Dejong 1st, dimenze = 500 ....................58
Obrázek 39 Diferenciální evoluce, Dejong 1st, dimenze = 1000 ....................59
Obrázek 40 Diferenciální evoluce, Dejong 1st, dimenze = 2500 ....................59
Obrázek 41 Diferenciální evoluce, Dejong 1st, dimenze = 5000 ....................60
Obrázek 42 Diferenciální evoluce, Schwefel, dimenze = 10....................61
Obrázek 43 Diferenciální evoluce, Schwefel, dimenze = 50....................61
Obrázek 44 Diferenciální evoluce, Schwefel, dimenze = 100....................62
Obrázek 45 Diferenciální evoluce, Schwefel, dimenze = 500....................62
Obrázek 46 Diferenciální evoluce, Schwefel, dimenze = 1000....................63
Obrázek 47 Diferenciální evoluce, Schwefel, dimenze = 2500....................63
Obrázek 48 Diferenciální evoluce, Schwefel, dimenze = 5000....................64
Obrázek 49 SOMA, Dejong 1st, dimenze = 10 ....................65
Obrázek 50 SOMA, Dejong 1st, dimenze = 50 ....................65
Obrázek 51 SOMA, Dejong 1st, dimenze = 100 ....................66
Obrázek 52 SOMA, Dejong 1st, dimenze = 500 ....................66
Obrázek 53 SOMA, Dejong 1st, dimenze = 1000 ....................67
Obrázek 54 SOMA, Dejong 1st, dimenze = 2500 ....................67
Obrázek 55 SOMA, Dejong 1st, dimenze = 5000 ....................68
Obrázek 56 SOMA, Schwefel, dimenze = 10....................69
Obrázek 57 SOMA, Schwefel, dimenze = 50....................69
Obrázek 58 SOMA, Schwefel, dimenze = 100....................70
Obrázek 59 SOMA, Schwefel, dimenze = 500....................70
Obrázek 60 SOMA, Schwefel, dimenze = 1000....................71
Obrázek 61 SOMA, Schwefel, dimenze = 2500....................71
Obrázek 62 SOMA, Schwefel, dimenze = 5000....................72
Obrázek 63 Zrychlení při populaci 10 ....................72
Obrázek 64 Zrychlení při populaci 50 ....................73

Obrázek 65 Zrychlení při populaci 100 .......... 73
Obrázek 66 Zrychlení při populaci 500 .......... 73
Obrázek 67 Zrychlení při populaci 1000 .......... 74
Obrázek 68 Zrychlení při populaci 2500 .......... 74
Obrázek 69 Zrychlení při populaci 5000 .......... 74
Obrázek 70 Paměťové požadavky GA .......... 75
Obrázek 71 Paměťové požadavky DE .......... 76
Obrázek 72 Paměťové požadavky SOMA .......... 77

# SEZNAM PŘÍLOH

Příloha P I: Zdrojové kódy implementovaných algoritmů

Příloha PII: Výsledky testů

Příloha PIII: Paměťové požadavky

# PŘÍLOHA P I: ZDROJOVÉ KÓDY IMPLEMENTOVANÝCH ALGORITMŮ

Zdrojové kódy se nacházejí v přiloženém souboru *sources.zip*.

## PŘÍLOHA PII: VÝSLEDKY TESTŮ

Výsledky testů a použité konfigurace se nacházejí v přiloženém souboru *results.csv.*

## PŘÍLOHA PIII: PAMĚŤOVÉ POŽADAVKY

Paměťové požadavky použitých konfigurací se nacházejí v přiložením souboru *memory.csv.*