Srovnání paralelní implementace algoritmu PSO v OpenCL, CUDA a C++ AMP

A comparison of the parallel implementation of the algorithm PSO in OpenCL, CUDA and C + + AMP

Bc. Roman Hala

Diplomová práce
2013

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2012/2013

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Roman Hala**
Osobní číslo:	**A11398**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Forma studia:	**prezenční**
Téma práce:	**Srovnání paralelní implementace algoritmu PSO v OpenCL, CUDA a C++ AMP**

Zásady pro vypracování:

1. **Zpracujte literární rešerši na téma Particle Swarm Optimization (PSO), OpenCL, CUDA a C++ Accelerated Massive Parallelism (C++ AMP).**
2. **Navrhněte implementaci PSO pro OpenCL, CUDA a C++ AMP.**
3. **Vytvořte testovací aplikaci.**
4. **Srovnejte výkon jednotlivých implementací.**
5. **Demonstrujte výsledky a formulujte závěr.**

Rozsah diplomové práce:

Rozsah příloh:

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

Seznam odborné literatury:

1. **BRATTON, D. Defining a Standard for Particle Swarm Optimization. In: 2007 IEEE Swarm Intelligence Symposium: Honolulu, HI, 1-5 April 2007. Piscataway, NJ: IEEE, 2007, s. 120-127. DOI: 1-4244-0708-7.**
2. **ZELINKA, Ivan. Umělá inteligence v problémech globální optimalizace. 1. vyd. Praha: BEN - technická literatura, 2002, 189 s. ISBN 80-730-0069-5.**
3. **GREGORY, Kate. C AMP: accelerated massive parallelism with Microsoft Visual C. United States of America: O'Reilly Media, 2012, 326 pages. ISBN 9780735668195.**
4. **MUNSHI, Aaftab. OpenCL programming guide. 1. vyd. Upper Saddle River, NJ: Addison-Wesley, 2012, xliv, 603 p. ISBN 03-217-4964-2.**
5. **SANDERS, Jason. CUDA by example: an introduction to general-purpose GPU programming. 1st print. Upper Saddle River: Addison-Wesley, 2010, xix, 290 s. ISBN 978-0-13-138768-3.**

Vedoucí diplomové práce: **Ing. Erik Král**
Ústav počítačových a komunikačních systémů

Datum zadání diplomové práce: **22. února 2013**

Termín odevzdání diplomové práce: **22. května 2013**

Ve Zlíně dne 22. února 2013

prof. Ing. Vladimír Vašek, CSc.
děkan

L.S.

doc. Mgr. Roman Jašek, Ph.D.
ředitel ústavu

ABSTRAKT

Cílem práce je srovnání CUDA, OpenCL a C++ AMP implementace evolučního algoritmu PSO. Teoretická část práce obsahuje úvod k paralelnímu programování, popis koncepce CUDA, OpenCL a C++ AMP, zabývá se základy evolučních algoritmů, z nichž podrobně rozebírá algoritmus PSO a vybrané testovací funkce. V praktické části se práce zaobírá popisem implementací algoritmu PSO pomocí standardů CUDA, OpenCL a C++ AMP a jejich srovnáním z hlediska náročnosti implementace, délky zdrojového kódu a doby výpočtu.

Klíčová slova: CUDA, OpenCL, NVIDIA, C++ AMP, C, C++, paralelní programování, PSO, evoluční algoritmy, optimalizace, testovací funkce

ABSTRACT

The aim of this thesis is the comparison of CUDA, OpenCL and C++ AMP implementation of the evolutionary algorithm PSO. The theoretical part provides an introduction to parallel programming, a description of the concept of CUDA, OpenCL and C + + AMP, deals with the fundamentals of evolutionary algorithms, of which the PSO algorithm and selected test functions are described in detail. Practical part of the thesis deals with the description of the algorithm PSO's implementations using standards CUDA, OpenCL and C + + AMP and compares their performance in terms of complexity of the implementation, the length of the source code and calculation time.

Keywords: CUDA, OpenCL, NVIDIA, C + + AMP, C, C + +, parallel programming, PSO, evolutionary algorithms, optimization, test functions

Chtěl bych poděkovat svému vedoucímu Ing. Eriku Královi, za jeho cenné rady a odbornou pomoc při psaní diplomové práce

Prohlašuji, že

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

Prohlašuji,

- že jsem na 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 …….

podpis diplomanta

OBSAH

ÚVOD

Dnešní GPU mají obrovské počty výpočetních jader a začaly se využívat nejen pro zpracování obrazu, ale také pro náročné paralelní výpočty.

Pro práci s výpočetními jádry GPU se používají standardy, jako jsou např. C++ AMP (C++ AMP (C++ Accelerated Massive Parallelism), OpenCL (Open Computing Language), nebo CUDA (Compute Unified Device Architecture). Ty podporují více programovacích jazyků, takže záleží na programátorovi, který z nich využije.

Cílem této práce je napsat paralelní implementaci evolučního algoritmu PSO (Particle Swarm Optimization) v rámci C++ AMP, OpenCL, CUDA a srovnat složitost implementace, délky kódu, přenositelnosti kódu a délky výpočetního času.

Teoretická část popisuje problematiku paralelních výpočtů a uvádí čtenáře do problematiky jednotlivých standardů. Dále popisuje základní princip evolučních algoritmů, z nichž podrobně rozebírá algoritmus PSO, který je řešen v praktické části. Na závěr teoretické části je popsáno i několik testovacích funkcí, které se využívají k testování optimalizačních algoritmů.

Praktická část se pak zabývá implementací standardní verze algoritmu PSO v jednotlivých standardech a jejich vzájemným srovnáním.

I. TEORETICKÁ ČÁST

1 PARALELNÍ VÝPOČTY

Paralelní výpočty se využívají pro ušetření výpočetního času stráveného nad řešením nějakého výpočetního problému. Ideální by bylo, kdyby se čas ušetřený paralelizováním zvětšoval lineárně. Tím pádem by se výpočetní čas rovnal při použití pěti výpočetních jednotek na jeden proces pětině času, který potřebuje jedna výpočetní jednotka. V praxi ale samozřejmě takových výsledků nelze téměř dosáhnout, protože program brzdí přirozeně sekvenční části programu, které nelze spustit paralelně.

Maximální možné paralelní zrychlení řeší Amdahlův zákon (1)

$$S(P) = \frac{1}{\frac{1-\alpha}{P}+\alpha}, \tag{1}$$

kde α je procentuálně vyjádřená část procesu, kterou je možné paralelizovat, (1-α) je část, kterou nelze paralelizovat a P je počet výpočetních jednotek.

V případě, kdy se změnou počtu procesorů změní i velikost úlohy, řeší rozšíření Amdahlova zákona tzv. Gustavsonův zákon (2)

$$S(P) = P - \alpha \cdot (P - 1), \tag{2}$$

kde α je procentuálně vyjádřená část procesu, kterou není možné paralelizovat a P je počet výpočetních jednotek.

Dále může také výpočet zbrzdit vzájemná komunikace a synchronizace výpočetních jednotek, které při dosažení větší časové ztráty, než zabere vlastní výpočet, nazýváme paralelní zpomalení.

K paralelismu, souběžnému běhu několika procesů zároveň, lze přistupovat různě. Program je schopen běžet na více jádrovém procesoru (nebo grafické kartě), na víceprocesorovém počítači, na několika počítačích zároveň a jejich kombinaci.

Při použití grafické karty hovoříme o tzv. heterogenním programování, protože program využívá dvě různá zařízení procesor (host) a grafickou kartu (kernel). [1]

1.1 OpenCL

OpenCL je standart pro programování heterogenních počítačových systémů a je prvním frameworkem, který byl navrhnut pro tyto systémy. První verze vyšla v prosinci roku 2008.

Pomocí OpenCL je možné napsat jeden program, který lze spustit na širokém spektru systémů – od mobilních telefonů, přes notebooky, až po uzly v obrovských super počítačích. Toto je jeden z důvodů, proč je OpenCL tak důležitý a má potenciál transformovat softwarový průmysl. Právě to je ale také zdrojem kritiky OpenCL. Přenositelnost je založena na explicitním popisu platformy, jejím kontextu a rozdělení práce mezi jednotlivá zařízení.[2][3]

1.1.1 Koncepce OpenCL

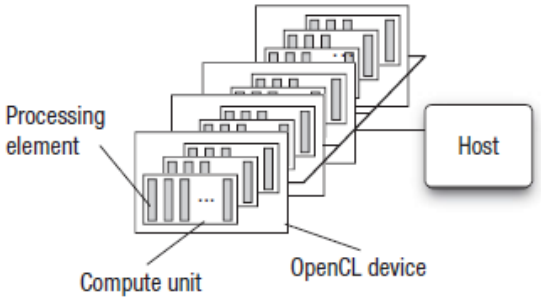
OpenCL lze aplikovat na širokou škálu aplikací. Postup implementace pro každou z nich je různý. V každém případě se ale pro použití heterogenní platformy musí použít následující postup:

1. Zjištění hardwarové konfigurace heterogenního systému.
2. Analýza architektury těchto prvků, aby software mohl být přizpůsoben specifickým rysům daného hardwaru.
3. Vytvoření bloků instrukcí (kernelů), které na platformě poběží.
4. Nastavení paměťových objektů, které jsou zapojené do výpočtu.
5. Spuštění kernelů ve správném pořadí a na správném prvku systému.
6. Shromáždění finálních výsledků.

Tyto kroky jsou prováděny pomocí API uvnitř OpenCL a programovacího prostředí pro kernely. Ty můžeme rozdělit pomocí 4 modelů (Platform model, Execution model, Memory model a Programming model). [2][3]

1.1.1.1 Platform model

Platform model popisuje vysokoúrovňovou reprezentaci všech heterogenních platforem používaných v OpenCL.



Obr. 1: Ukázka platform modelu OpenCL [2]

Platforma má vždy jen jednoho hosta, který externě komunikuje s programem OpenCL pomocí I/O nebo s uživatelem. Host je spojen s jedním nebo více OpenCL device, na kterém se spouští kernely. Dále se OpenCL zařízení dělí na compute units. Ty se dělí na processing elements. [2][3]

1.1.1.2 Execution model

OpenCL aplikace se skládá ze dvou částí: jednoho host programu a soustavy jednoho nebo více kernelů.

Host program běží na hostovacím zařízení (procesor). Detaily toho, jak má host program pracovat, nejsou definovány. Důležitá je jen jeho spolupráce s objekty v OpenCL.

Kernely, které se spouští na OpenCL device, dělají to, pro co byl OpenCL program navržen. Většinou jsou to jednoduché funkce, které transformují vstupní paměťové objekty do výstupních paměťových objektů. OpenCL má dva druhy kernelů:

- OpenCL kernel: Funkce napsány v jazyce OpenCL C pomocí ukazatelů na funkce, jsou zkompilovány pomocí OpenCL kompilátoru.
- Native kernel: Funkce vytvořené mimo OpenCL a spojeny s OpenCL pomocí ukazatelů na funkce. Tyto funkce mohou být např. definovány ve zdrojovém kódu host programu nebo mohou být ve speciální knihovně.

Execution model definuje spouštění těchto kernelů. [2][3]

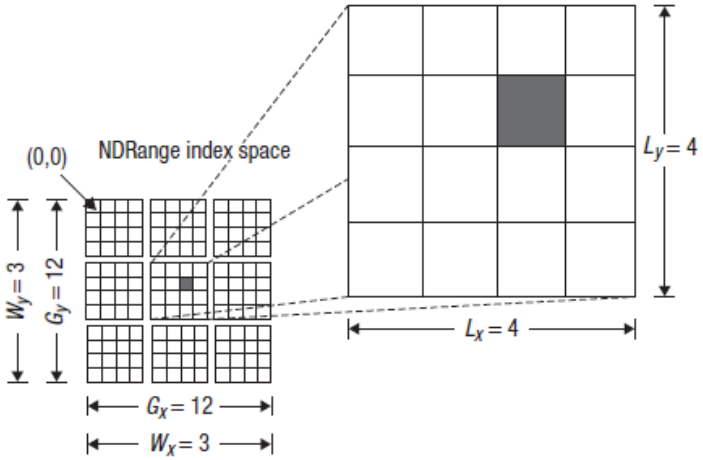
Spouštění kernelu

Kernely jsou definovány v host programu. Ten obsahuje příkaz, který potvrdí spuštění kernelu na OpenCL device. Při spuštění tohoto příkazu si OpenCL runtime systém vytvoří indexovací pole. Pro každý tento index je spuštěna jedna instance kernelu. Tyto instance se nazývají work-item a jsou identifikovány globálně pomocí svého indexu. Work-itemy

jsou uspořádány do work-groups. Mezi work-groups je rozdělen celý indexovací prostor. Work-groups mají unikátní ID, které mají stejnou dimenzionalitu, jako indexovací pole, které používaji work-items. Work-items mají přiřazeny unikátní local ID v rámci work-group, takže každý work item je indentifikován buď pomocí jeho global ID, nebo pomocí kombinace local ID a work-group ID.

Work-items v rámci jedné work-group souběžně zpracovávají prvky jedné compute unit. V implementaci můžeme serializovat spouštění kernelů. Dokonce se může serializovat spouštění work-groups do jednoho zavolání kernelu.

Indexovací prostor N-dimenzionálního rozsahu hodnot se nazývá NDRange. Globální a lokální ID každého work-itemu je N-dimenzionální n-tice.



Obr. 2: Ukázka dvourozměrného NDRange [2]

Na Obr. 2 je vidět orientace ve dvourozměrném NDRange. Zvýrazněný čtverec (work-item) má globální index (6, 5) a spadá pod work-group s ID (1, 1) s lokálním indexem (2, 1). [2][3]

Context

V OpenCL dělají výpočetní práci OpenCL devices. Nicméně host hraje v aplikaci OpenCL také velkou roli. V hostu jsou definovány všechny kernely. Host definuje context pro kernely. Host definuje NDRange a fronty, které řídí detaily o tom, jak a kde se kernely spouští. Všechny tyto důležité informace jsou obsaženy v OpenCL API.

První úkol hosta je definovat context pro OpenCL aplikaci. Context je prostředí, ve kterém jsou definovány a posléze spouštěny kernely, a obsahuje k tomu tyto prostředky:

- Devices: Souhrn OpenCL devices použitých hostem.
- Kernels: Funkce OpenCL, které běží na OpenCL devices.

- Program objects: Zdrojové kódy a spustitelné soubory, které implementují kernely.
- Memory objects: Množina paměťových objektů, které jsou viditelné pro OpenCL devices a obsahují proměnné, se kterými mohou pracovat jednotlivé instance kernelu. [2][3]

Command-Queues

Vzájemnou komunikaci mezi hostem a OpenCL devices zajišťuje command-queue pomocí příkazů napsaných v hostu. Tyto příkazy čekají v command queue, dokud se nespustí na OpenCL device. Command-queue může obsahovat tři typy příkazů:

- Kernel Execution commands: Spouští kernely na processing elements ,které jsou na OpenCL device.
- Memory commands: Zajišťují přenos dat mezi hostem a různými paměťovými objekty, přesunují data mezi paměťovými objekty nebo zajišťují mapování paměťových objektů z adresového prostoru hosta.
- Synchronization commands: Synchronizuji spouštění příkazů. [2][3]

1.1.1.3 Memory model

Memory model se stará o definování memory objects. Tyto memory objects se dělí na buffer objects a image objects.

Buffer objects jsou jen souvislé bloky vyrovnávací paměti, které jsou k dispozici kernelům. Programátor může do těchto bloků mapovat data a přistupovat k nim pomocí ukazatelů.

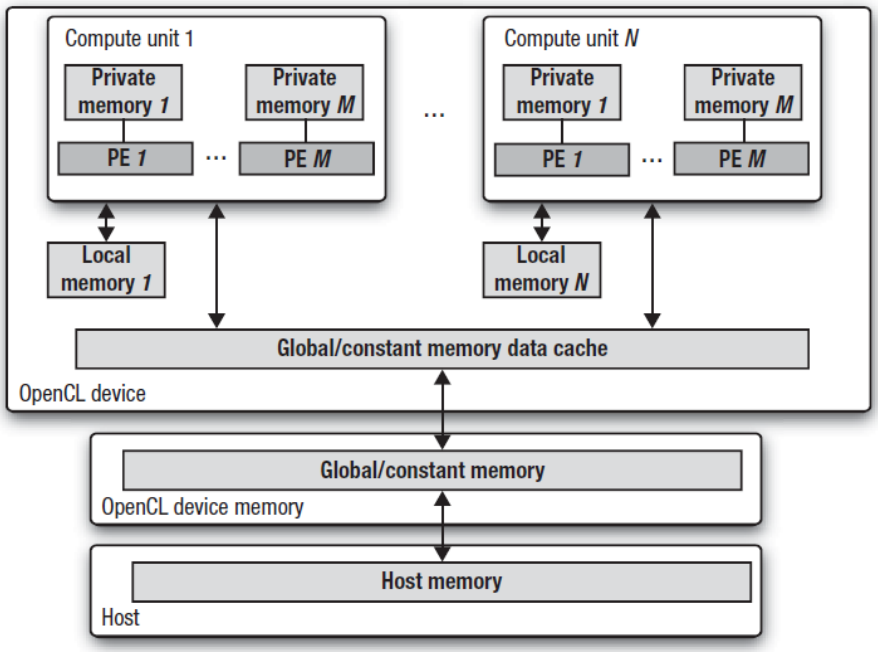
Image objects jsou určeny výhradně pro ukládání obrazů. Ukládací formát těchto obrazů může být optimalizován pro potřeby konkrétního OpenCL device. Framework OpenCL má pro práci s obrazy specifické funkce. Jinak než pomocí těchto funkcí nelze k obsahu image objects přistupovat.

OpenCL umožňuje programátorům specifikovat podobjekty paměťových objektů, které lze zpracovávat pomocí command-queue.

OpenCL memory model se dělí na pět různých částí:

- Host memory: Tato část paměti je přístupná pouze hostovi.
- Global memory: Umožňuje přístup všem work-items ve všech work-groups ke čtení a zápisu.

- Constant memory: Zůstává neměnná během spuštěného výpočtu na kernelu. Host přiděluje a inicializuje memory objects, které jsou uloženy v constant memory. Work-items mají v této paměti právo pouze na čtení.
- Local memory: Je omezena jen pro work-group. Může být použita pro alokaci proměnných, které jsou sdílené všemi work-items ve work-group. Implementace se provádí buď jako dedikovaná část paměti na OpenCL device, nebo se může namapovat část globální paměti.
- Private memory: Tato část paměti je přístupná pouze pro work-item a není viditelná ostatním work-itemům.



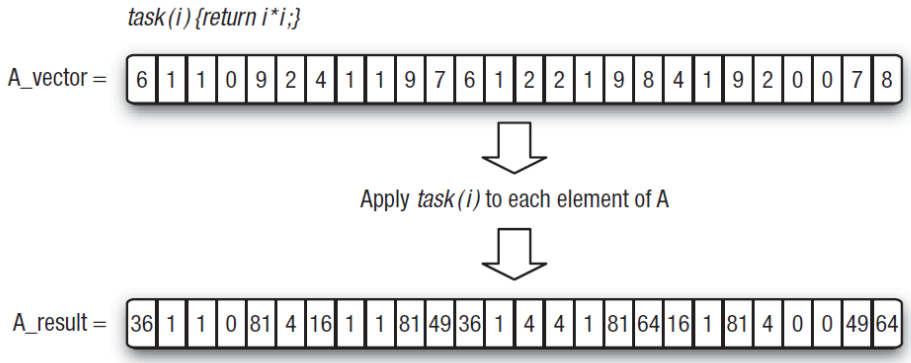
Obr. 3: Ukázka memory modelu v OpenCL [2]

Na Obr. 3 je vidět, že paměť hosta a paměť OpenCL device jsou na sobě nezávislé, nicméně spolu potřebují spolupracovat pomocí kopírování dat a mapování jednotlivých částí memory objectu. Při explicitním kopírování dat zařadí host do fronty příkazy pro přenos dat mezi memory objectem a pamětí hosta. Tyto příkazy mohou být blokující a neblokující. Blokující se liší od neblokujících tím, že se spustí teprve tehdy, až je bezpečné paměť znova použít. Mapování umožňuje memory objektům využít paměť hosta. Tyto příkazy jsou taktéž blokující a neblokující. [2][3]

1.1.1.4 Programming models

Programming model je flexibilnější než přesně definovaný Execution model. Každý programátor může stejný algoritmus napsat jiným způsobem. OpenCL má dva druhy programming models: task parallelism a data parallelism. Může být použita i jejich kombinace. [2][3]

Data-parallel programming model



Obr. 4: Jednoduchá ukázka data-parallel-programming modelu [2]

Problémy, které spadají do data-parallel programming modelu jsou zaměřeny na datové struktury, jejichž prvky lze měnit současně. Prakticky to funguje tak, že jedna sekvence instrukcí je aplikována na všechny prvky datové struktury. [2]

Task-parallel programming model

OpenCL execution model je primárně určený pro datové paralelizování, nicméně také podporuje širokou škálu algoritmů pro paralelizování jednotlivých úkolů. OpenCL považuje za úkol kernel, který se spustí jako jeden work-item bez ohledu na NDRange použitý jinými kernely v aplikaci. Tento programovací model se používá, když chce programátor paralelně spustit více různých kernelů. [2]

1.2 CUDA

Standard CUDA představený firmou NVIDIA je univerzální architektura pro paralelní výpočty na zařízeních NVIDIA. Zahrnuje instrukční sadu Instruction Set Architecture (ISA) a paralelní výpočetní jednotku GPU. K programování se využívá různých jazyků a API jako C, C++, C#, Fortran, Java, Python, OpenCL nebo DirectX Compute a jiných. CUDA

Podporuje Heterogenní výpočty, kdy spolu pracují jak CPU, tak GPU. GPU, která podporují CUDA mají stovky výpočetních jader, na nichž můžou běžet až tisíce vláken. [4]

Systémové požadavky CUDA:

Podporované operační systémy:

- Operační systém Microsoft Windows XP a novější, Linux, Mac OS X.

Podporovaný hardware:

- Grafická karta s podporou CUDA.

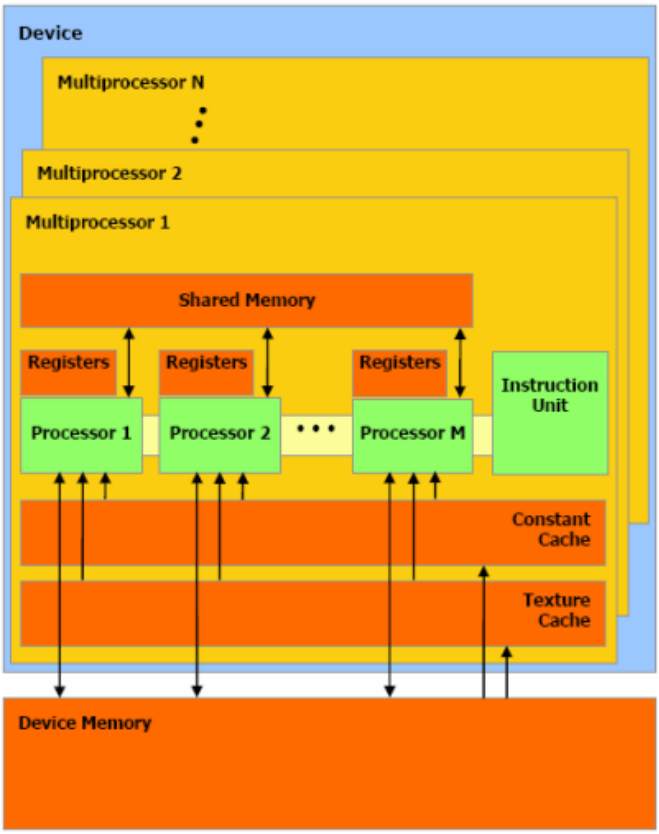
Podporované softwarové vybavení:

- Ovladač zařízení,
- CUDA software (dostupný na http://www.nvidia.com/cuda),
- Microsoft Visual Studio 2005 a novější. [5]

1.2.1 Koncepce CUDA

1.2.1.1 Platform model

Stejně jako OpenCL má platforma vždy jen jednoho hosta, který externě komunikuje s kernelem (kernely) pomocí I/O nebo s uživatelem. Host je spojen k jednomu nebo více CUDA device, na kterém se spouští kernely. CUDA device se skládá z globální Device memory a Multiprocesorů, které mají uvnitř sebe ještě další menší a rychlejší paměťové moduly, které využívají Procesory uvnitř něj. [4]



Obr. 5: Ukázka platform modelu CUDA [6]

1.2.1.2 Memory model

Paměťový model je rozdělený do několika částí:

Grid:

- Constant memory: Umožňuje jen čtení, je pomalá, ale má 8kb cache.
- Global memory: Ke čtení i zápisu, je pomalá a nemá cache. Aby byla rychlá, potřebuje sekvenční a 16bytové čtení a zápis.

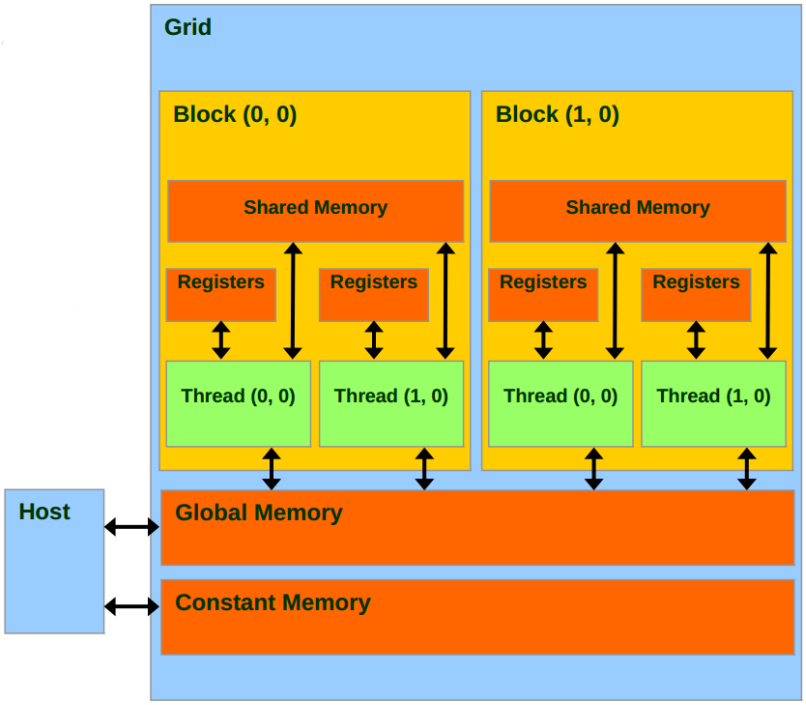
Block:

- Shared memory: Ke čtení i zápisu, je rychlá, využívá se pro výměnu dat mezi vlákny v blocku.

Thread:

- Local memory: Ke čtení i zápisu, je pomalá a nemá cache, ale má automaticky sloučené čtení a zápis.
- Registers: Ke čtení i zápisu, nejrychlejší, omezena jen pro jedno vlákno.

Vlákna mají omezený přístup do některých paměťových částí – Různé vlákna si navzájem nevidí do svých registrů a jednotlivé bloky si navzájem nevidí do své Shared memory.



Obr. 6: Ukázka memory modelu CUDA [7]

Kromě zmíněných existuje i Texture memory, která navíc nabízí jiné režimy adresování a filtrování dat pro některé specifické datové formáty. Tato paměť je pouze ke čtení. [7]

1.2.1.3 Execution a Programming model

Kernely

CUDA dovoluje programátorovi používat funkce jazyku C, nazvané kernels, které se Nkrát paralelně spouštějí na N rozdílných CUDA threads.

Kernel je definován pomocí specifikátoru declarace __global__ a počtu CUDA threads, které spustí kernel tolikrát, kolikrát je volán pomocí execution cofiguration, např. Function<<1, N>>(A, B, C). Každý thread, na kterém probíhá kernel má unikátní thread ID, který lze zjistit pomocí vestavěné proměnné threadIdx. [4]

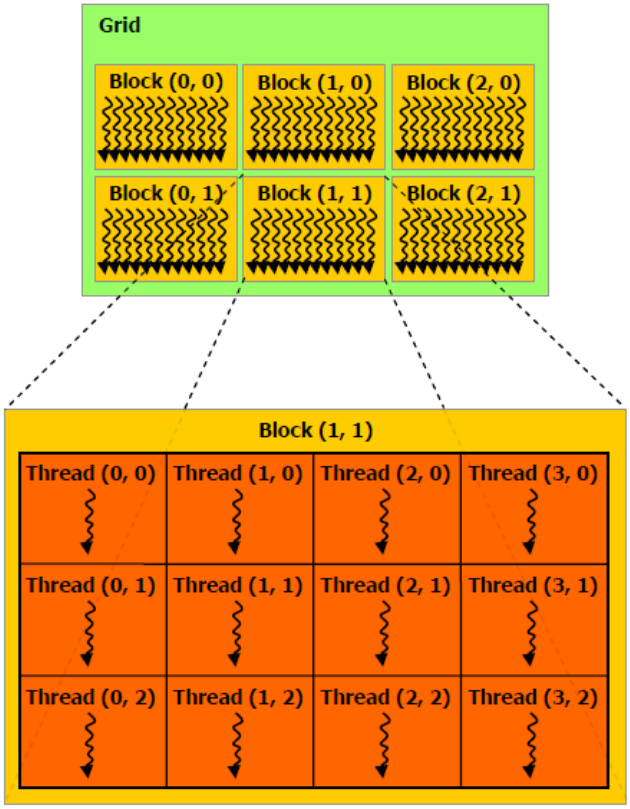
Hierarchie threads

ThreadIdx je navrhnut jako tříprvkový vektor, který identifikuje vlákna pomocí jednodimenzionálního, dvoudimenzionálního a třídimenzionálního thread index, které tvoří jednodimenzionální, dvoudimenzionální a třídimenzionální threadblock. Díky tomuto získáme možnost dělat výpočty v prvcích, jako jsou např. vektory a matice.

Index threadu a jeho thread ID vztahující se ke každému z nich jsou pro jednodimenzionální blok stejné, pro dvoudimenzionální blok o velikosti (D_x, D_y) má threadID ($x + yD_x$) thread index (x, y). Pro třidimenzionální blok o velikosti (D_x, D_y, D_z) má threadID ($x + y + zD_x,D_y$) thread index (x, y, z).

Počet threads per block je omezen, protože všechny threads per block jsou umístěny na stejném výpočetním jádře, a tak musí sdílet omezené paměťové zdroje toho jádra.

Nicméně kernel může být spuštěn na několika threads per block se stejnými rozměry, takže výsledný počet threads je roven počtu threads per block vynásobeným počtem blocks.



Obr. 7: Ukázka threads per block v grid [4]

Na Obr. 7 je vidět uspořádání blocks jsou uspořádány do jednodimenzionálního, dvoudimenzionálního nebo třídimenzionálního grid, který určuje počet thread.

Počet threads blocks v grid je obvykle určen velikostí dat, která se mají zpracovat, nebo počtem procesorových jader systému.

Počet threads per block a počet blocks per grid se určuje pomocí int nebo pomocí dim3.

Každý blok v grid může být identifikován pomocí jednodimenzionálního, dvoudimenzionálního nebo třídimenzionálního indexu přístupného v kernelu pomocí vestavěné proměnné blockIdx. Rozměr thread blocku je přístupný v kernelu pomocí vestavěné proměnné blockDim.

Grid se vytváří s dostatkem blocků, aby každý prvek matice měl jedno vlákno.

Thread blocks se musí spouštět nezávisle. Musí být možné je spouštět v jakémkoliv pořadí, paralelně i sériově. Tato nezávislost umožňuje tread blocks, aby byly spouštěny na jakémkoliv počtu výpočetních jader, a programátorovi umožňuje napsat kód, který se rozdělí podle počtu jader.

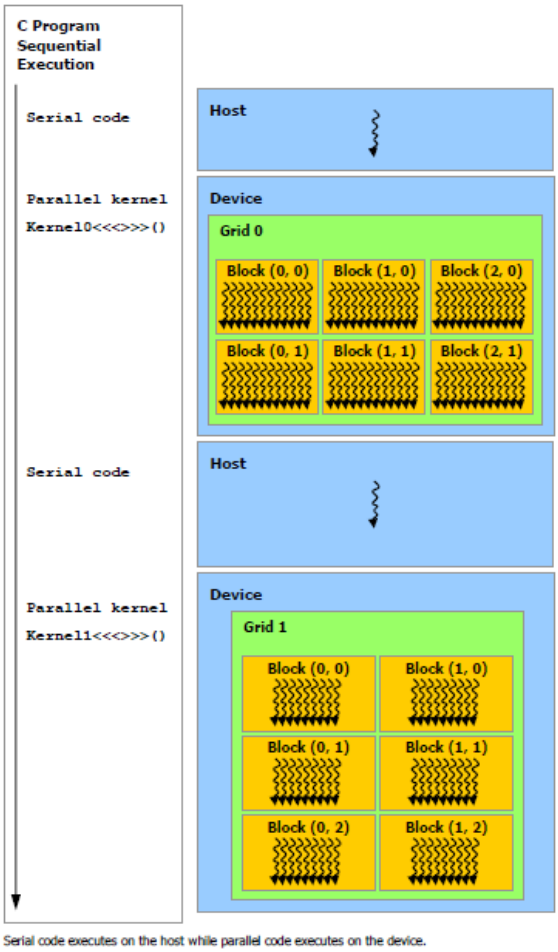
Threads v blocku mohou spolupracovat pomocí sdílení shared memory a synchronizování jejich spouštění ke koordinaci paměťových přístupů. Synchronizační bod se volá pomocí vnitřní funkce __syncthreads(), která se chová jako závora, na níž se musí všechny threads zastavit a počkat na ostatní, než můžou zase pokračovat.

Pro efektivní spolupráci musí mít shared memory nízkou latenci a __syncthread() nesmí být náročná. [4]

1.2.2 Heterogenní programování na CUDA zařízení

Programovací model CUDA předpokládá, že se CUDA threads spustí na fyzicky odděleném zařízení device, které pracuje jako coprocesor programu host. Kernel běží na GPU a host běží na CPU.

Dále se předpokládá, že host i device budou používat oddělené paměťové oddíly DRAM host memory a device memory. Proto si program sám spravuje globální, konstantní a texturovou paměť viditelnou pro kernely volané pomocí CUDA runtime. To zahrnuje alokaci, dealokaci a přesun dat mezi host memory a device memory. [4]



Obr. 8: Ukázka heterogenního programování pomocí CUDA [4]

1.2.3 Výpočetní možnosti CUDA devices

Výpočetní možnosti device se dají zjistit pomocí čísel major revision number a minor revision number.

Devices se stejným major revision number mají stejnou architekturu výpočetních jader. Major revision number je 3 pro architekturu Kepler, 2 pro Fermi architektruru a 1 pro zařízení s architekturou Tesla.

Minor revision number odpovídá výkonnostnímu vylepšení architektury. [4]

1.3 C++ AMP

C++ AMP (Accelerated Massive Parallelism) je rozšíření C++, které zrychluje provedení C++ kódu využitím datově paralelního hardwaru, obvykle GPU. Programovací model C++ AMP zahrnuje podporu vícerozměrných polí, indexování, paměťového přenosu

a rozdělování threadů do bloků. Taktéž zahrnuje knihovnu matematických funkcí. Lze také řídit přesun dat z CPU na GPU a zpět. [8]

Systémové požadavky C++ AMP

Podporované operační systémy:

- Windows 7 + Service pack 1, Windows 8, Windows Server 2008 R2 (x64) a Windows Server 2012 (x64).

Podporovaný hardware:

- Grafická karta s podporou DirectX 11,
- Procesor s frekvencí alespoň 1,6 GHz,
- 1GB RAM pro 32 bitový operační systém a 2GB RAM pro 64 bitový operační systém,
- 10 GB volného místa na pevném disku pro instalaci Visual Studia 2012.

Požadované softwarové vybavení:

- Visual Studio 2012 (Professional nebo Ultimate),
- Direct X 11. [8]

1.3.1 Koncepce C++ AMP

1.3.1.1 Platform model

Platform model pro C++ AMP využívá pro paralelní zrychlení grafické karty s podporou DirectX 11. Tím pádem není omezena pouze na grafické čipy firmy NVIDIA, ale podporuje i AMD. Struktura platform modelu vychází z modelu OpenCL viz Obr. 1. DirectX 11 zařízení lze ale také simulovat jako Microsoft DirectX REF device, WARP (zrychlení pomocí instrukcí SSE) nebo přímo procesorem. [8]

1.3.1.2 Memory model

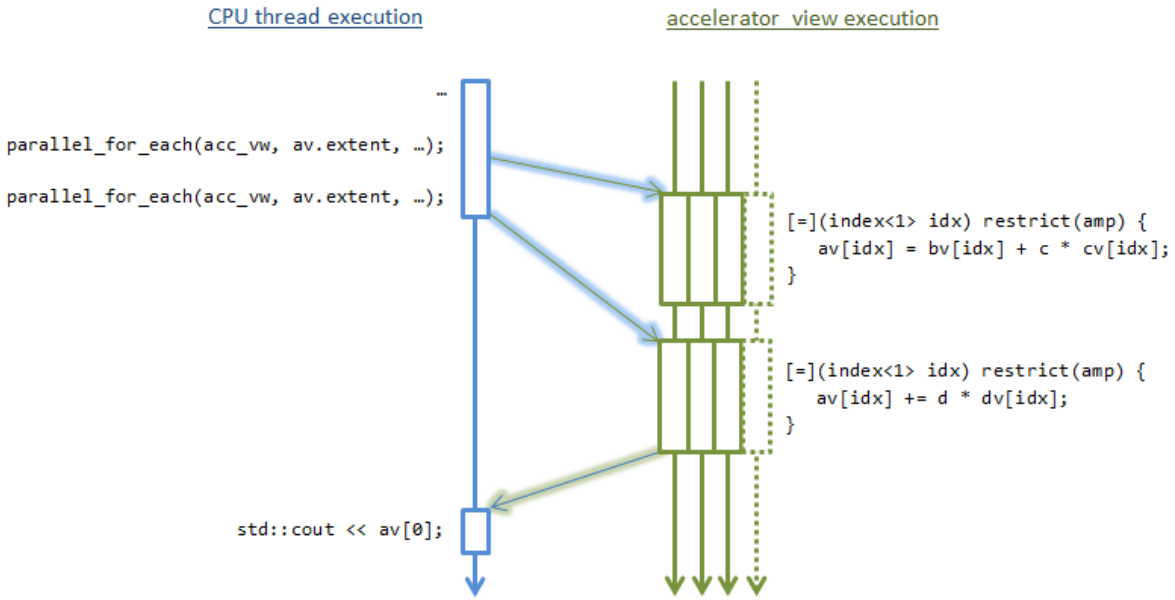
parallel_for_each může využít buď globální, nebo lokální paměti. Při použití lokální paměti je třeba při definici proměnné přidat na začátek příznak tile_static a poté teprve typ. Lokální paměť je rychlejší, ale je omezená, proto se musí dobře zvážit, co se do ní vyplatí uložit. [8]

1.3.1.3 Execution model

C++ AMP nevyžaduje pro heterogenní výpočty zvláštní kernely jako OpenCL nebo CUDA. Stačí použít metody tohoto rozšíření v jakémkoliv zdrojovém kódu, do kterého naimportujeme hlavičku <amp.h>. Pak už k paralelizaci stačí použít místo normálních C++ proměnných a cyklů použít speciální C++ AMP. [8]

1.3.1.4 Programming model

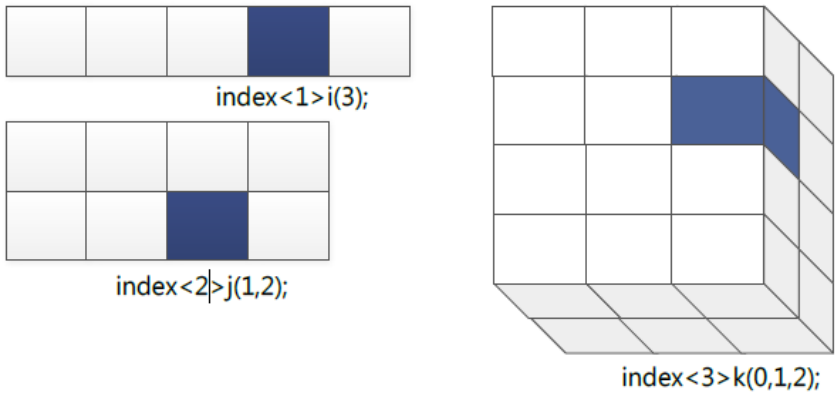
Programovací model, který je zaměřený na heterogenní programování. V host programu, který běží na procesoru se vytvoří paralelní for cyklus parallel_for_each, který funguje jako kernel v OpenCL, nebo CUDA.



Obr. 9: Ukázka paralelizování cyklu for pomocí funkce parallel_for_each [9]

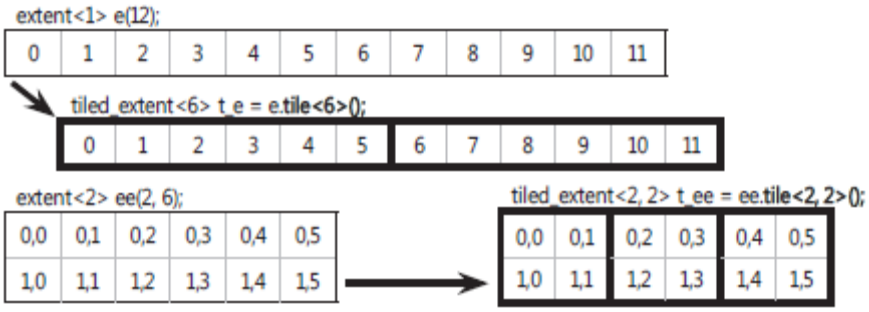
Při použití funkcí, které jsou mimo parallel_for_each je nutné za deklaraci připsat restrict(amp) pro paralelizování.

Každý prvek pole má přiřazeno svoje ID pomocí proměnné index. Index může nabývat jednoho až tří rozměrů. [8]



Obr. 10: Ukázka jedno, dvou a tří dimenzionálního indexování [8]

Při využití lokální paměti se indexování změní a místo objektu index se použije tiled_index, který má stejné rozměry.



Obr. 11: Ukázka jedno a dvourozměrného indexování pomocí tiled_extent [10]

2 TDR

2.1 Definice

TDR neboli Timeout Detection and Recovery GPU je mechanismus sloužící k ošetření při přetížení grafického čipu náročnými grafickými operacemi nebo při pozastavení zobrazování uživateli.

Pokud nastane jedna z těchto situací, operační systém Microsoft Windows tento mechanismus použije pro restartování grafického ovladače pokaždé, když nějaká aplikace odstřihne všechny prostředky GPU od zbytku systému na delší dobu, než je povoleno. Při restartování grafického ovladače obrazovka displeje problikne, uživateli se zobrazí zpráva ve tvaru „Grafický ovladač přestal odpovídat a úspěšně obnoven“ a GPU se uvolní pro ostatní aplikace. [11]

2.2 Nastavení TDR v registrech

Nastavení TDR v registrech se nachází pod HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\GraphicDrivers.

TdrLevel (REG_DWORD): první krok obnovy.

- TdrLevelOff (0): Vypnutá detekce uváznutí.
- TdrLevelBugcheck (1): Neřeší obnovu funkce GPU, může ale např. kontrolovat bugy.
- TdrLevelRecoverVGA (2): Obnova grafického zobrazení (není implementováno).
- TdrLevelRecover (3): Obnova při uváznutí na dobu delší než je povoleno.

TdrDelay (REG_DWORD): Určuje maximální délku uváznutí GPU v sekundách.

TdrDdiDelay (REG_DWORD): Určuje maximální dobu, po kterou operační systém dovolí vlákna opustit ovladač. Po uplynutí této doby operační systém zkontroluje systém kódem VIDEO_TDR_FAILURE (0x116).

TdrTestMode (REG_DWORD): Využití pro vnitřní testování.

TdrDebugMode (REG_DWORD): Chování související s debugováním mechanismu TDR.

- TDR_DEBUG_MODE_OFF (0): Systém před resetováním grafického ovladače spustí kernel debuger pro zjištění důvodu uváznutí.

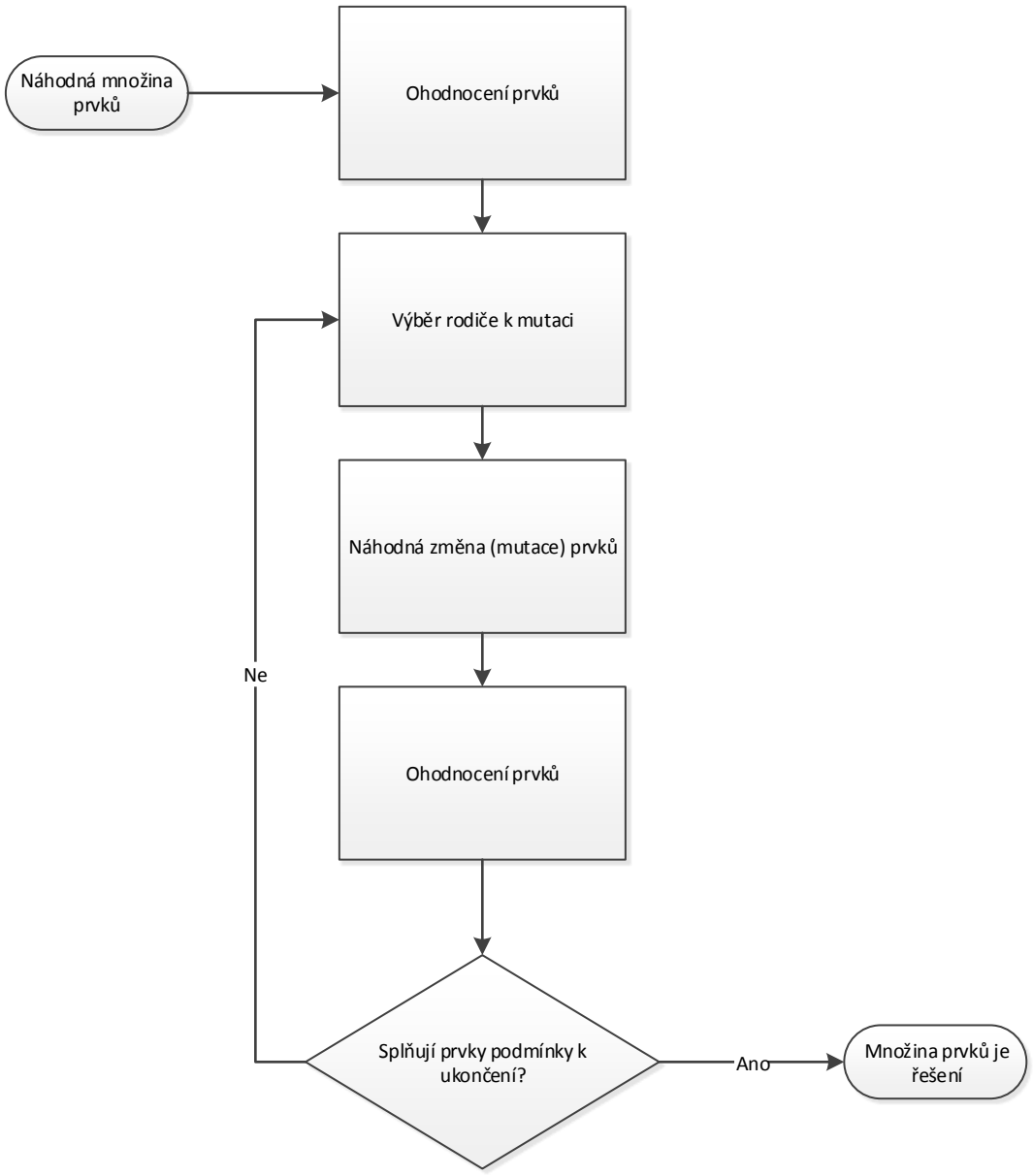
- TDR_DEBUG_MODE_IGNORE_TIMEOUT (1): Systém ignoruje maximální dobu uváznutí.
- TDR_DEBUG_MODE_NO_PROMPT (2): Systém restartuje grafický ovladač bez vstupu do debugování kernelu.
- TDR_DEBUG_MODE_RECOVER_UNCONDITIONAL (3): Grafický ovladač se restartuje, i když nejsou splněny všechny podmínky uváznutí.

TdrLimitCount (REG_DWORD): Výchozí počet TDR bez restartování systému (Windows Vista SP1 a novější).

TdrLimitTime (REG_DWORD): Defaultní nastavení doby pro přípustný počet TDR, aniž by se musel restartovat systém (Windows Vista SP1 a novější). [11]

3 EVOLUČNÍ ALGORITMY

Evoluční algoritmy jsou optimalizační výpočetní metody, které vycházejí z principů Darwinovy a Mendelovy teorie evoluce. Podle této teorie se jednotlivé druhy vyvíjejí z rodiče na potomka pomocí mutací. Potomci, kteří nesplňují podmínky pro přežití, tak zanikají po generacích a uvolní místo pro lepší potomky, ze kterých se v další generaci vytvoří další potomci. Tímto se hledají nejlepší možní jedinci. Tato teorie se aplikuje na optimalizaci funkcí (hledání extrémů). [12]



Obr. 12: Vývojový diagram optimalizace pomocí evolučního algoritmu

3.1 PSO (Particle Swarm Optimization)

PSO je stochastický optimalizační algoritmus vyvinutý Russellem Eberhartem a Jamesem Kennedym v roce 1995. Při tvorbě algoritmu se inspirovali chováním rybích a ptačích hejn. [13]

3.1.1 Princip algoritmu

Za hejno algoritmus považuje několik bodů náhodně rozmístěných na optimalizační funkci. Pomocí těchto bodů hledá algoritmus nejvyšší bod. Hejno sice neví, jestli je nějaký bod v extrému, ale ví, který bod má po každé iteraci největší hodnotu funkce, a ten bod ostatní body následují. [12]

3.1.2 Parametry algoritmu PSO

Dimenze

Je prostor, ve kterém se optimalizační problém řeší. Rozměry dimenze určuje počet argumentů v účelové funkci. [12]

Rozsah

Je velikost prostoru, ve kterém se můžeme v dané dimenzi pohybovat. Každá dimenze může mít jiné parametry. [12]

Počet částic

Je to počet jedinců roje, kteří budou hledat extrém účelové funkce. Tento počet se většinou volí mezi 20 – 40 v poměru přesnosti a složitosti výpočtu. [12]

Učící faktory c1, c2

Mají vliv na posun částic. Můžou nastat tři situace – posun k nejlepšímu výsledku, k nejlepšímu výsledku populace nebo se částice drží původního směru. *c1* směruje částici k nejlepšímu výsledku a *c2* k nejlepšímu výsledku populace. Oba tyto faktory jsou ještě doplněny o náhodný prvek, a to o vynásobení náhodným číslem v rozmezí od 0 do 1. Hodnota *c1* a *c2* bývá nejčastěji nastavena na hodnotu 2, ale běžně se používá interval od 0 do 4. Některé implementace PSO algoritmu nahrazují násobené *c1* a *c2* fixními parametry, ale v tom případě není do výpočtu rychlosti zaveden žádný prvek náhody. [12]

Vmax

Určuje rychlost jedinců roje. Malá rychlost způsobuje přesné prohledání malého prostoru, ale moc velká zase může způsobit, že jedinci budou překračovat povolený rozsah pohybu a že se pak bude muset vygenerovat nová náhodná pozice jedince. Z PSO se poté stane náhodný prohledávací algoritmus, který nemá s evolucí nic společného. [12]

V rovnici (3) se provádí výpočet rychlosti pomocí předchozí rychlosti.

$$v_{i+1} = v_i + c1 \cdot random(0,1) \cdot (pBest - x_i) + c2 \cdot random(0,1) \cdot (gBest - x_i) \quad (3)$$

Setrvačnost w

Ovlivňuje rychlost částice, zpravidla s každou iterací rychlost částice zpomaluje, aby se nejdříve prohledávaly velké části funkce a postupně se tento prostor snižoval a zaměřil jen na okolí nejlepší částice. Setrvačnost bývá nastavena buď na začátku algoritmu na určitou hodnotu a je postupně lineárně snižována, nebo může být předem definována jak počáteční, tak koncová hodnota, a výsledná setrvačnost je před každou iterací přepočítána. [12]

$$w = w_{start} - \frac{(w_{start} - w_{end}) \cdot \text{číslo aktuální iterace}}{\text{počet iterací}} \quad (4)$$

V rovnici (5) se provádí výpočet rychlosti pomocí setrvačnosti a předchozí rychlosti.

$$v_{i+1} = w \cdot v_i + c1 \cdot random(0,1) \cdot (pBest - x_i) + c2 \cdot random(0,1) \cdot (gBest - x_i) \quad (5)$$

Constriction faktor χ

Používá se místo setrvačnosti. Hodnota bývá nastavena na hodnotu 0,729. [12]

$$\chi = \frac{2}{\left|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}\right|} \quad (6)$$

V rovnici (7) se provádí výpočet rychlosti pomocí constriction faktoru a předchozí rychlosti.

$$v_{i+1} = \chi \cdot (v_i + c1 \cdot random(0,1) \cdot (pBest - x_i) + c2 \cdot random(0,1) \cdot (pBest - x_i)) \quad (7)$$

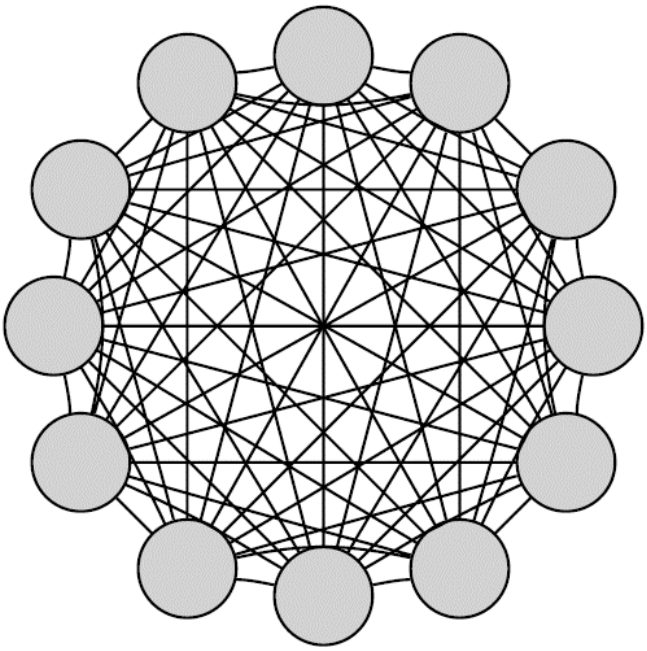
Výpočet pozice nové pozice částice

$$x_{i+1} = x_i + v_{i+1} \quad (8)$$

3.1.3 Varianty PSO

gbest (global topology)

Varianta s globální topologií vybírá po přepočtu fitness nejlepšího jedince z celé populace částic. Tato varianta je vhodná pro jednodušší funkce s jedním maximem, protože vlivem orientace všech částic za jedním jedincem má tendenci algoritmus předčasně konvergovat k lokálním maximům. [12]

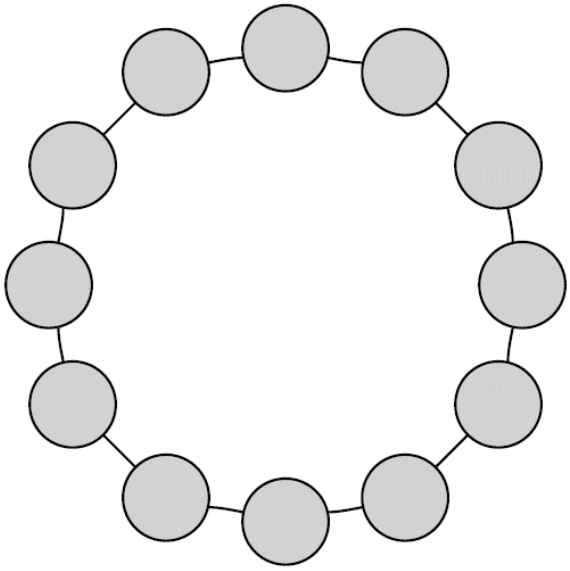


Obr. 13: Ukázka komunikace částic v gbest topologii [14]

lbest (local topology)

Varianta s lokální topologií vybírá po přepočtu fitness více nejlepších jedinců. Všechny částice jsou rozděleny do skupin neboli sousedství, které mohou být geografické a sociální. Geografické skupiny rozdělují body podle oblasti, ve které se nachází. Sociální skupiny využívají různých topologií. Nejčastější sociální topologií je kruh, v němž určuje sousedství „pořadové“ číslo jedince. Je to varianta, která omezuje riziko konvergování do lokálních extrémů.

Při použití sousedství vznikne nový parametr, který určuje počet prvků ve skupině. Tento parametr bývá nastavený v rozmezí 3-5, ale není to nutné, a algoritmus změna tohoto parametru nijak zásadně neovlivní. [12]

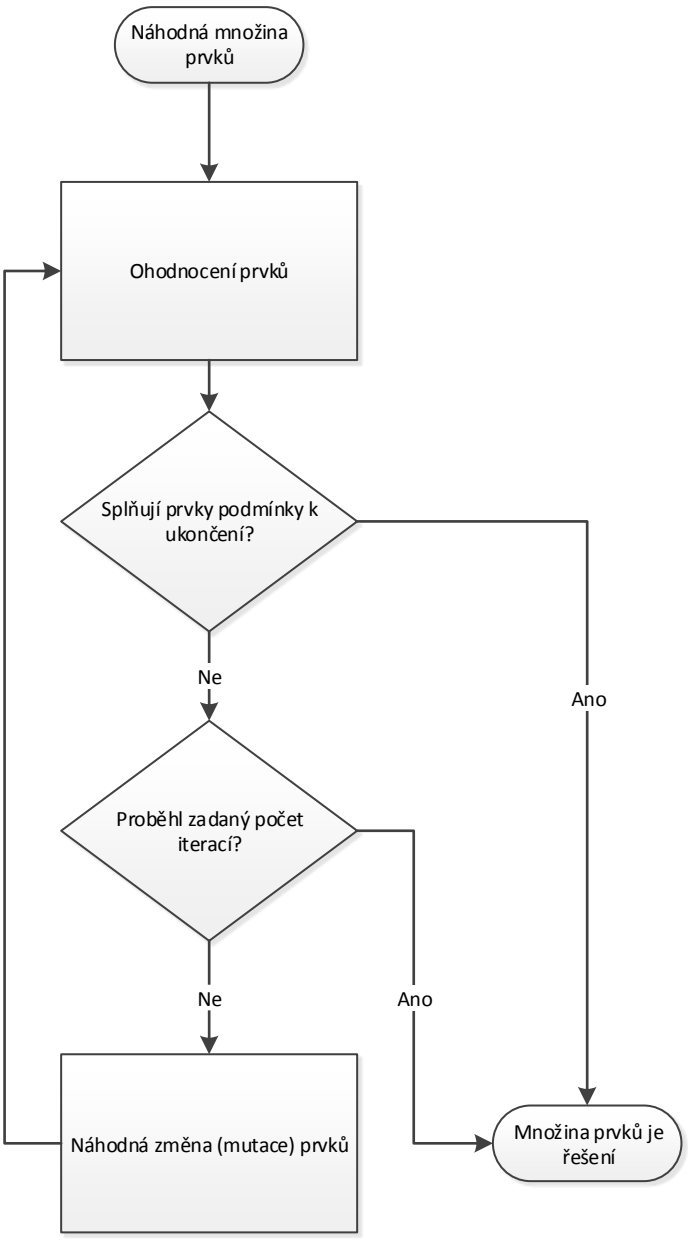


Obr. 14: Ukázka komunikace částic v lbest topologii [14]

3.1.4 Postup optimalizace

Nejprve se algoritmus musí inicializovat, to znamená, že se vygenerují náhodné body, přidělí se jim náhodné rychlosti, spočítají se hodnoty funkce a vybere se nejlepší jedinec v populaci, jenž je zároveň i nejlepším jedincem celkově.

Poté se pro počet zadaných iterací provede pro všechny částice přepočet hodnoty funkce, vybere se nejlepší bod z populace, porovná se s nejlepším bodem celkově, porovnají se ukončovací kritéria (pokud jsou splněna, tak se algoritmus ukončí), přepočítá se rychlost částice, nová poloha částice.



Obr. 15: Vývojový diagram znázorňující postup optimalizace algoritmu PSO

4 TESTOVACÍ FUNKCE

Kvalita optimalizačních algoritmů se hodnotí pomocí tzv. testovacích funkcí. U těchto funkcí jsou předem známa správná řešení. Testovací funkce jsou rozděleny do několika tříd:

a) **unimodální, konvexní, multidimenzionální**

   Tato třída obsahuje funkce snadno optimalizovatelné, ale i funkce se špatnou a pomalou konvergencí ke globálnímu extrému.

b) **multimodální, dvourozměrné s malým počtem lokálních extrémů**

   Tato třída je složitostí někde mezi třídami *a)* a *c).*

c) **multimodální, dvourozměrné s velkým počtem lokálních extrémů**

   Tato třída je vhodná pro testování inteligentních algoritmů. Problémem pro použití těchto funkcí je, že v praxi je většina optimalizačních problému více, než dvourozměrná.

d) **multimodální, multidimenzionální, s velkým počtem lokálních extrémů**

   Tato třída je vhodná pro testování inteligentních algoritmů.

Všechny tyto testovací funkce jsou spojité. [15]

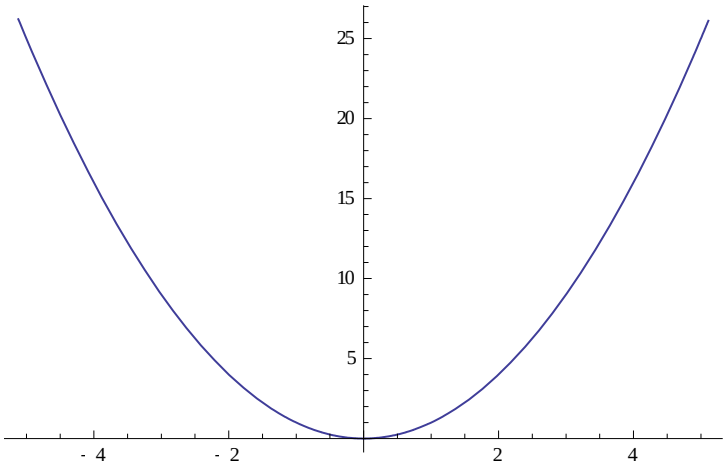
4.1 De Jongovy testovacífunkce

Tyto funkce jsou jedny z nejjednodušších. Spojité, konvexní a unimodální. Celkem jsou 4 verze: [15][16]

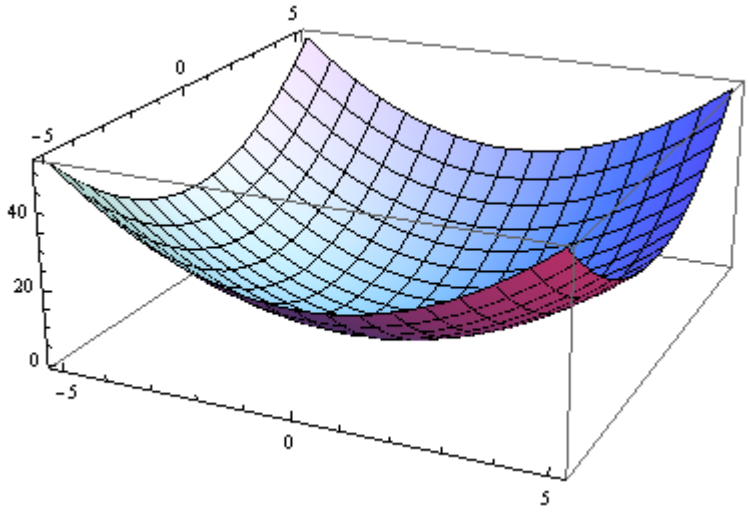
4.1.1 1. De Jongova funkce

Popis

Je to velmi jednoduchá funkce tvaru paraboloidu. Pokud má testovaný algoritmus problém najít extrém na této funkci, tak není kvalitní. [15][16]



Obr. 16: Grafické 2D znázornění 1. De Jongovy funkce



Obr. 17: Grafické 3D znázornění 1. De Jongovy funkce

Matematický vzorec

$$f(x) = \sum_{i=1}^{N} x_i^2 \tag{8}$$

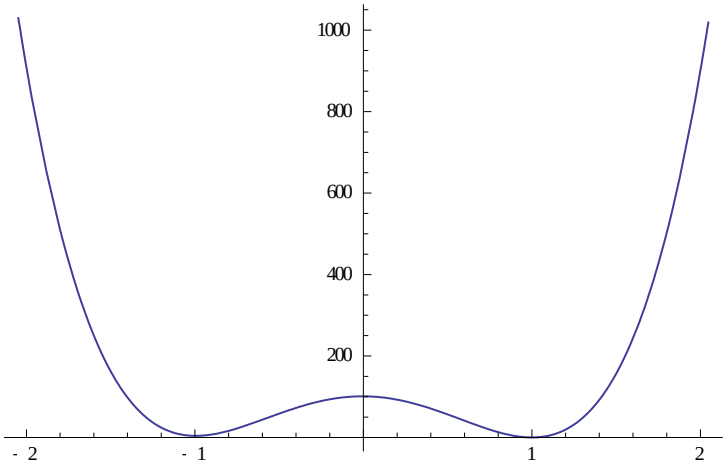
Globální minimum

Globální minimum 1. De Jongovy funkce v n-dimenzionálním prostoru se nachází na pozici $(x_1, x_2, x_3, \dots x_n) = 0$ a má hodnotu $(y = 0 \ \times n = 0)$. [12]

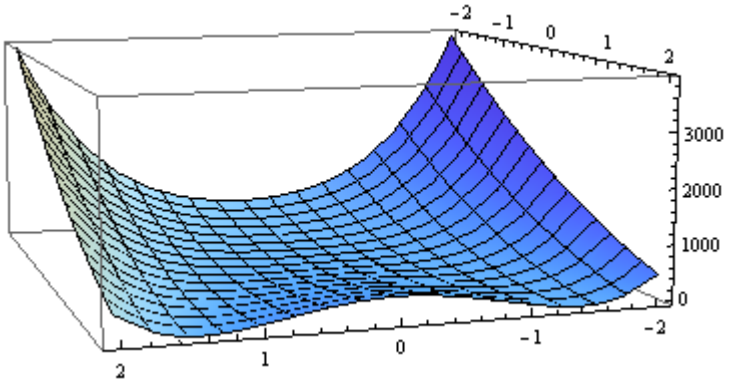
4.1.2 2. De Jongova funkce

Popis

2. De Jongova funkce, neboli Rosenbrockovo sedlo je jednomodální neseparabilní funkce. Nalezení globálního minima, které leží na dně sedla je zde složitější kvůli malému klesání, díky němuž může algoritmus ukončit hledání extrému dřív, než se k němu dostane. [15][16]



Obr. 18: Grafické 2D znázornění 2. De Jongovy funkce



Obr. 19: Grafické 3D znázornění 2. De Jongovy funkce

Matematický vzorec

$$f(x) = \sum_{i=1}^{N-1} (100(x_i^2 - x_{i+1})^2 + (1 - x_i^2)) \tag{9}$$

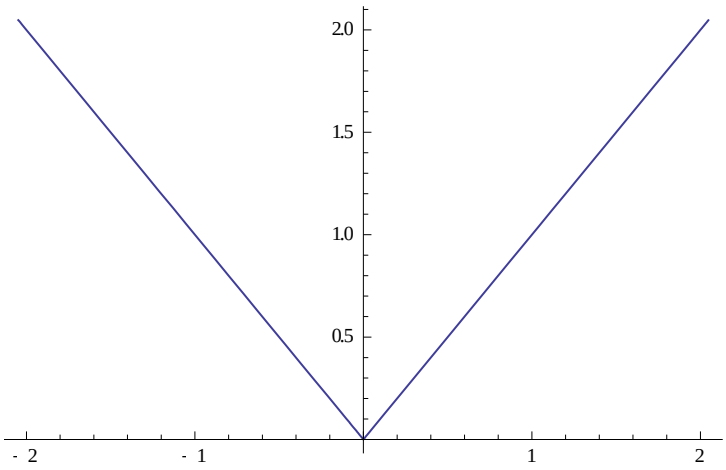
Globální minimum

Globální minimum 2. De Jongovy funkce v n-dimenzionálním prostoru se nachází na pozici $(x_1, x_2, x_3, \dots x_n) = (1,1,1, \dots 1)$ a má hodnotu $(y = 0 \ \times n = 0)$. [12]

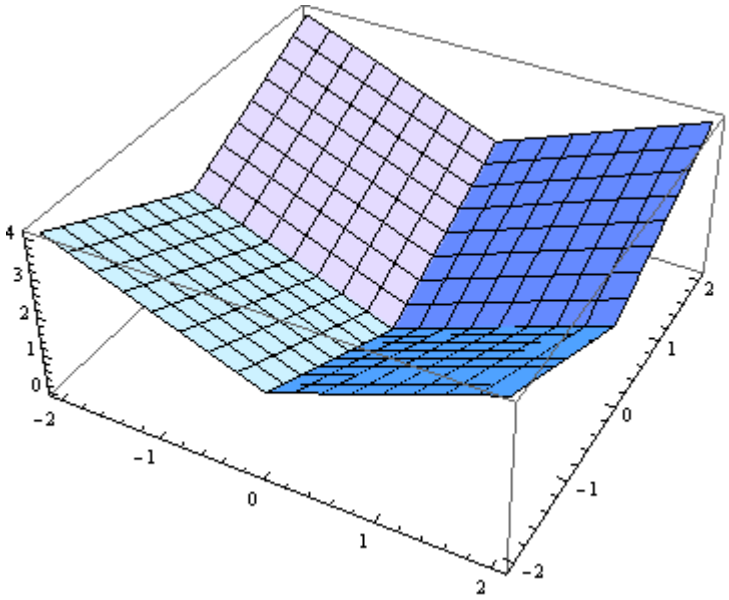
4.1.3 3. De Jongova funkce

Popis

3. De Jongova funkce je jednoduchá funkce, obdobně jako 1. De Jongova funkce, jen nemá tvar paraboloidu. Pokud má testovaný algoritmus problém najít extrém na této funkci, tak není kvalitní. [15][16]



Obr. 20: Grafické 2D znázornění 3. De Jongovy funkce



Obr. 21: Grafické 3D znázornění 3. De Jongovy funkce

Matematický vzorec

$$f(x) = \sum_{i=1}^{N} |x_i| \tag{10}$$

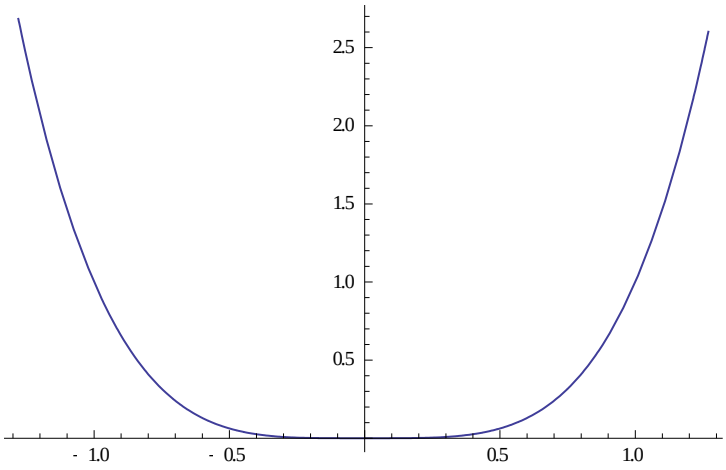
Globální minimum

Globální minimum 3. De Jongovy funkce v n-dimenzionálním prostoru se nachází na pozici $(x_1, x_2, x_3, \dots x_n) = 0$ a má hodnotu $(y = 0 \ \times n = 0)$. [12]

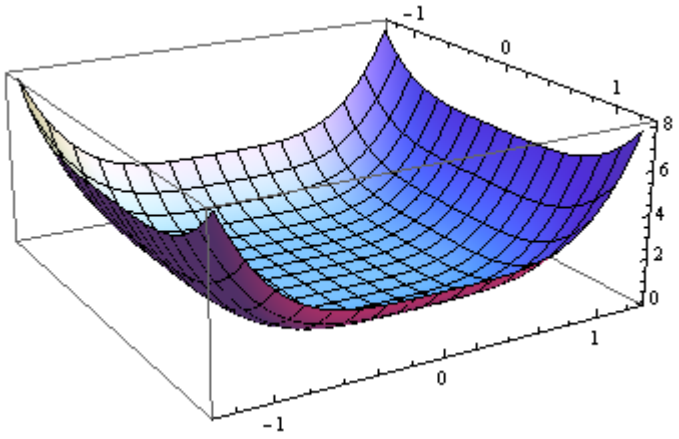
4.1.4 4. De Jongova funkce

Popis

4. De Jongova funkce je tvaru paraboloidu, ale má velmi malé klesání ke globálnímu minimu, takže algoritmus může ukončit hledání extrému dřív, než se k němu dostane. [15][16]



Obr. 22: Grafické 2D znázornění 4. De Jongovy funkce



Obr. 23: Grafické 3D znázornění 4. De Jongovy funkce

Matematický vzorec

$$f(x) = \sum_{i=1}^{N} i x_i^4 \tag{11}$$

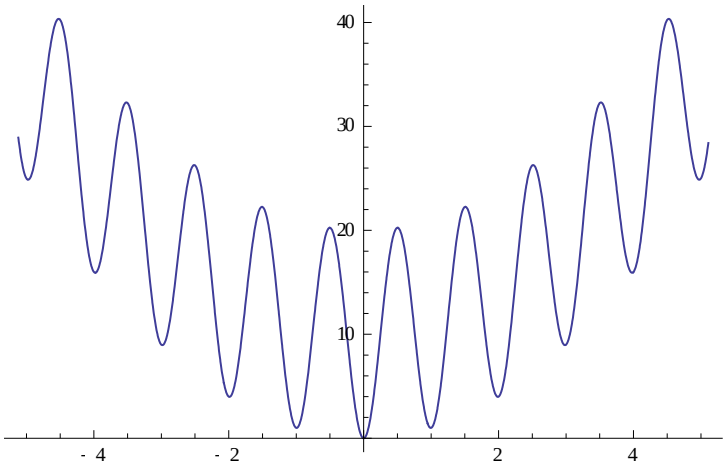
Globální minimum

Globální minimum 4. De Jongovy funkce v n-dimenzionálním prostoru se nachází na pozici $(x_1, x_2, x_3, \dots x_n) = 0$ a má hodnotu $(y = 0 \ \times n = 0)$. [12]

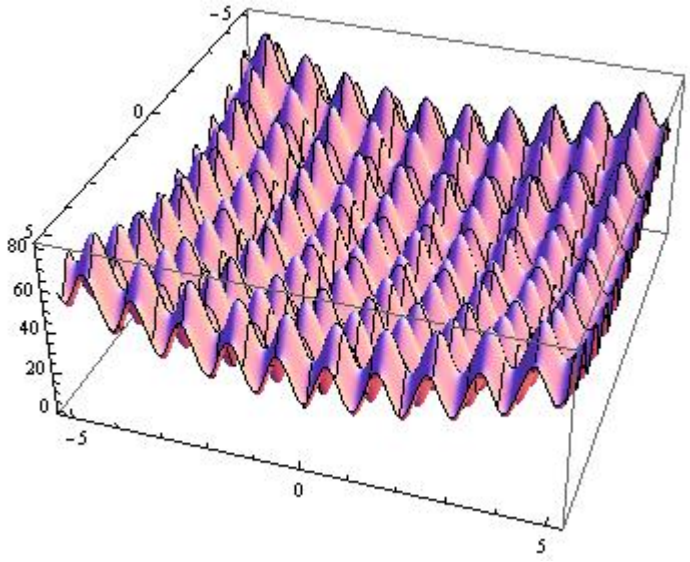
4.1.5 Rastriginova funkce

Popis

Rastriginova funkce je multimodální a separabilní. Je založena na De Jongově funkci s přidáním funkce kosinus, aby se vytvořila hojná lokální minima. Pro optimalizační algoritmy je nalezení globálního minima této funkce složitý úkol. [15][16]



Obr. 24: Grafické 2D znázornění Rastriginovy funkce



Obr. 25: Grafické 3D znázornění Rastriginovy funkce

Matematický vzorec

$$f(x) = 10N + \sum_{i=1}^{N} [x_i^2 + 10\cos(2\pi x_i)] \tag{12}$$

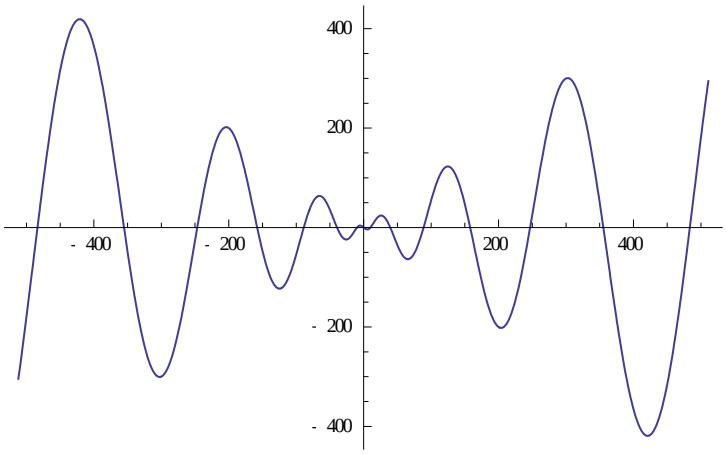
Globální minimum

Globální minimum Rastriginovy funkce v n-dimenzionálním prostoru se nachází na pozici $(x_1, x_2, x_3, \dots x_n) = 0$ a má hodnotu $(y = -200 \times n)$. [12]

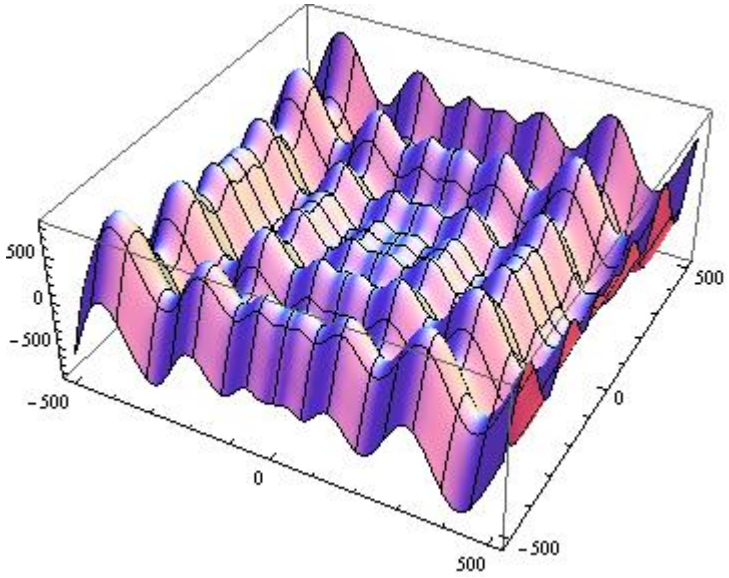
4.1.6 Schwefelova funkce

Popis

Schwefelova funkce je multimodální. Je záludná v tom, že globální minimum je vzdálené od lokálního minima, a proto mají optimalizační algoritmy tendenci konvergovat špatným směrem. [15][16]



Obr. 26: Grafické 2D znázornění Schwefelovy funkce



Obr. 27: Grafické 3D znázornění Schwefelovy funkce

Matematický vzorec

$$f(x) = \sum_{i=1}^{N} -x_i \sin(\sqrt{|x_i|}) \tag{13}$$

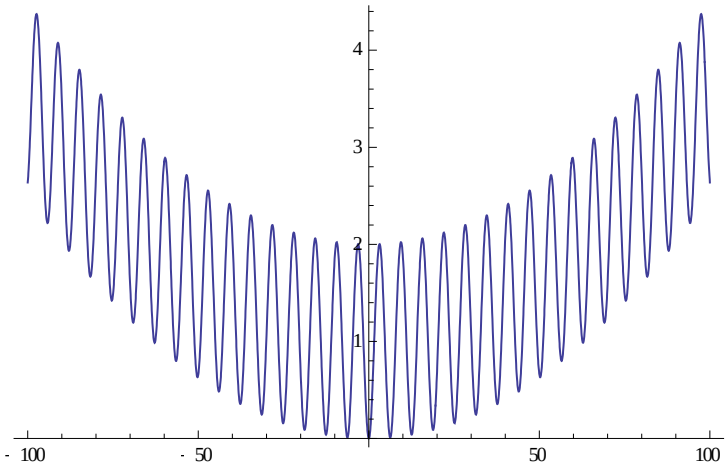
Globální minimum

Globální minimum Schwefelovy funkce v n-dimenzionálním prostoru se nachází na pozici $(x_1, x_2, x_3, \dots x_n) = (420{,}969; 420{,}969; 420{,}969; \dots 420{,}969;)$ a má hodnotu $(y = -418{,}983 \times n)$. [12]

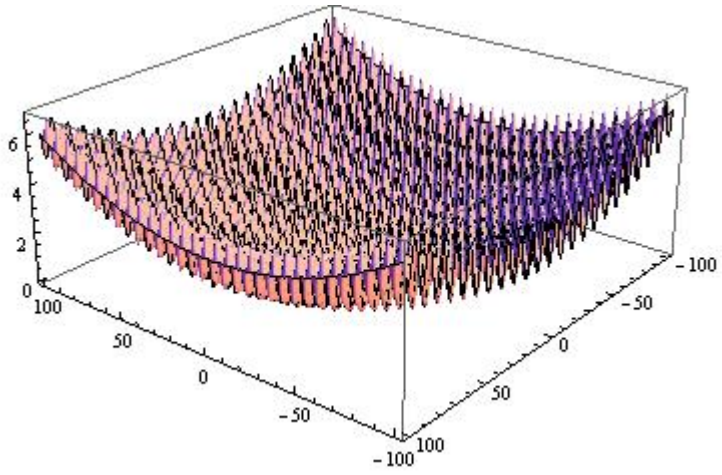
4.1.7 Griewangkova funkce

Popis

Griewangkova funkce je multimodální a nalezení jejího globálního minima patří mezi složité úlohy. [15][16]



Obr. 28: Grafické 2D znázornění Griewangkovy funkce



Obr. 29: Grafické 3D znázornění Griewangkovy funkce

Matematický vzorec

$$f(x) = \sum_{i=1}^{N} \frac{x_i^2}{4000} - \prod_{i=1}^{N} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1 \tag{14}$$

Globální minimum

Globální minimum Griewangkovy funkce v n-dimenzionálním prostoru se nachází na pozici $(x_1, x_2, x_3, \dots x_n) = 0$ a má hodnotu $(y = 0 \ \times n = 0)$. [12]

II. PRAKTICKÁ ČÁST

5 SDK

5.1 OpenCL

Pro řešení problému paralelní implementace algoritmu PSO pomocí OpenCL bylo zvoleno OpenCL SDK od firmy Intel z důvodu automatické integrace do MS Visual Studia a přislíbené možnosti debugingu. Byla využita aktuální verze Intel® SDK for OpenCL* Applications 2013 pro 64 bitové systémy dostupná ze zdroje [17].

5.2 CUDA

K vývoji CUDA zařízení byla použita nejnovější verze CUDA toolkitu verze 5 pro 64 bitové systémy. Byla zvolena verze s nástrojem pro integraci do MS Visual Studia Nsight™ Visual Studio Edition 3.0 Release Candidate 2 Installer Bundled with CUDA Toolkit dostupná pouze po registraci ze zdroje [18]. Díky nástroji Nsight lze pohodlně debugovat i na GPU.

5.3 C++ AMP

C++ AMP jako jediné nepotřebuje doinstalovat žádné SDK, pro paralelizaci kódu pomocí C++ AMP stačí pouze do zdrojového kódu importovat knihovnu *amp.h*. Jediná podmínka pro funkčnost je splnění systémových požadavků.

6 IMPLEMENTACE ALGORITMU STANDART PSO

Pro srovnání OpenCL, CUDA a C++ AMP bylo použito PSO s lbest topologií.

6.1 Inicializace proměnných

Na začátku hlavní funkce *main* byly inicializovány všechny proměnné, které jsou při výpočtu potřeba.

6.1.1 OpenCL

V OpenCL byla použita proměnná pole *cl_float*, kterou používá přímo OpenCL. Pro jednoduchost předávání parametrů paralelní funkci bylo vytvořeno pomocné pole *parameters*.

```
1  int Iter = 300000;            //počet iterací
2  int N = 50;                   //počet jedinců
3  int D = 30;                   //počet dimenzí
4  float min = -100;             //spodní hranice prostoru, ve kterém
   se částice mohou pohybovat
5  float max = 100;              //horní hranice prostoru, ve kterém
   se částice mohou pohybovat
6  float c1 = 2.05;              //parametr pro mutaci
7  float c2 = 2.05;              //parametr pro mutaci
8  float lambda = c1 + c2;       //parametr pro mutaci
9  float chi = 2 / ( abs(2 - lambda - sqrt(lambda * lambda - 4 *
   lambda)) );                                  //parametr pro mutaci
10 cl_float* X = new cl_float[ N * D ];      //souřadnice jedinců
11 cl_float* P = new cl_float[ N * D ];      //nejlepší souřadnice
   jedinců
12 cl_float* V = new cl_float[ N * D ];      //rychlosti jedinců
13 cl_float* f = new cl_float[ N ];          //fitness jedinců
14 cl_float parameters[] = {Iter, N, D, c1, c2, chi, min, max}
   //pomocné pole pro předání parametrů výpočtu
```

6.1.2 CUDA

Stejně jako v OpenCL implementaci bylo pro jednoduchost předávání parametrů paralelní funkci vytvořeno pomocné pole *parameters*.

```
1  int Iter = 300000;
2  int N = 50;
3  int D = 30;
4  float min = -100;
5  float max = 100;
```

```
 6 float c1 = 2.05;
 7 float c2 = 2.05;
 8 float lambda = c1 + c2;
 9 float chi = 2 / ( abs(2 - lambda - sqrt(lambda * lambda - 4 *
   lambda)) );
10 float* X = new float[N * D];
11 float* P = new float[N * D];
12 float* V = new float[N * D];
13 float* f = new float[N];
14 float parameters[] = {Iter, N, D, c1, c2, chi, min, max};
```

6.1.3 C++ AMP

V C++ AMP byly mimo funkci *main* definovány dvě statické proměnné, které sloužily pro definování velikosti pole alokovaného z local memory GPU.

```
1 static const int TileSize = 50;           //stejná velikost jako je
  počet částic v roji
2 static const int TileDimension = 30;      //stejná velikost jako je
  dimenzí
```

Ve funkci *main* pak byla tyto proměnné použity taktéž.

```
 3 int Iter = 300000;
 4 const int N = TileSize;
 5 const int D = TileDimension;
 6 float min = -100;
 7 float max = 100;
 8 float c1 = 2.05;
 9 float c2 = 2.05;
10 float lambda = c1 + c2;
11 float chi = 2 / ( abs(2 - lambda - sqrt(lambda * lambda - 4 *
   lambda)) );
12 float* Xh = new float[N * D];
13 float* Ph = new float[N * D];
14 float* Vh = new float[N * D];
15 float* fh = new float[N];
```

6.2 Inicializace algoritmu

Inicializace algoritmu byla vytvořena pomocí roje náhodných čísel v rozmezí zadaných parametrů.

```
1  for(int i = 0; i < N; ++i)
2  {
3     f[i] = FLT_MAX;
4     for(int d = 0; d < D; ++d)
5     {
6        int id = (d + i * D);
7        X[id] = min + ( (max - min) * (float)rand()/(float)RAND_MAX);
8        V[id] = 0;
9     }
10 }
```

Rychlost jedinců v inicializaci byla nastavena na hodnotu nula a fitness všech jedinců na hodnotu *FLT_MAX*.

Inicializace byla provedena pro OpenCL, CUDU i C++ AMP stejně.

6.3 Přechod do paralelní části programu

6.3.1 OpenCL

6.3.1.1 Zpracování chybových stavů

Pro zpracování chybových stavů při přechodu do paralelní části programu byla implementována funkce volající se při každém kroku, který by mohl způsobit chybu při přechodu do kernelu.

Ukázka volání funkce:

```
1 checkErr(err, "Context::Context()");
```

Funkce s výpisem chyby:

```
2 inline void checkErr(cl_int err, const char * name)
3 {
4    if (err != CL_SUCCESS)
5    {
6        std::cerr << "ERROR: " << name << " (" << err << ")" <<
  std::endl;
7        exit(EXIT_FAILURE);
8    }
9 }
```

Pokud nastala jakákoliv chyba, bylo do konzole vypsáno, ve kterém kroku se chyba objevila a číslo chyby.

6.3.1.2 Volba výpočetního zařízení

Před volbou zařízení byly vypsány všechny platformy, z nichž byla zvolena první dostupná platforma, což na mém notebooku odpovídá platformě NVIDIA.

```
1  cl::vector< cl::Platform > platformList;
2  cl::Platform::get(&platformList);

3  checkErr(platformList.size() != 0 ? CL_SUCCESS : -1,
   "cl::Platform::get");

4  std::string platformVendor;
5  for(int i = 0; i < platformList.size(); ++i)
6  {
7      platformList[i].getInfo((cl_platform_info)CL_PLATFORM_VENDOR,
   &platformVendor);
8     std::cout << i << " Platform is by: " << platformVendor << "\n";
9  }

10 cl::Platform selectedPlatform = platformList[0];

11 cl_context_properties cprops[3] =
12 {
13     CL_CONTEXT_PLATFORM,
14   (cl_context_properties)(selectedPlatform)(),
15   0
16 };
```

Po zvolení platformy bylo vybráno také výpočetní zařízení. Všechna zařízení se nejprve vypsala do konzole i s jejich vlastnostmi, a poté bylo vybráno první zařízení, což odpovídalo zařízení NVIDIA GT555M.

```
1  cl::vector<cl::Device> deviceList;
2  deviceList = context.getInfo<CL_CONTEXT_DEVICES>();
3  checkErr(deviceList.size() > 0 ? CL_SUCCESS : -1, "deviceList.size()
   > 0");
4  std::string deviceName;
5  cl_ulong localSize;
6  cl_uint maxWorkItemDimensions;
7  size_t localMaxWorkGroupSize[3];
8  for(int i = 0; i < deviceList.size(); ++i)
9  {
```

```
10    deviceList[i].getInfo((cl_device_info)CL_DEVICE_NAME,
   &deviceName);
11 deviceList[i].getInfo((cl_device_info)CL_DEVICE_LOCAL_MEM_SIZE,
   &localSize);
12 deviceList[i].getInfo((cl_device_info)CL_DEVICE_MAX_WORK_ITEM_DIMENS
   IONS, &maxWorkItemDimensions);
13 deviceList[i].getInfo((cl_device_info)CL_DEVICE_MAX_WORK_GROUP_SIZE,
   localMaxWorkGroupSize);
14 std::cout << i << " Device name is: " << deviceName << " local size:
   " << localSize;
15 std::cout << " maxWorkItemDimensions: " << maxWorkItemDimensions;
16 std::cout << " max work group size: " << localMaxWorkGroupSize[0] <<
   " " << localMaxWorkGroupSize[1] << " "<<localMaxWorkGroupSize[2]  <<
   "\n";
17 }
18 cl::Device device = deviceList[0];
```

6.3.1.3 Alokace paměti a kopie dat do GPU pro pole s parametry hejna

Pro každé pole byl vytvořen *buffer*, který alokoval paměť, nastavil práva zápisu a čtení a uložil do ní pole z host programu.

Ukázka *bufferu* pro pole souřadnic *X*:

```
1 cl::Buffer XCL(context, CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR,
  sizeof(cl_float) * N * D, X, &err);
```

6.3.1.4 Spuštění kernelu

Název souboru *Kernel.cl* byl uložen do proměnné typu *string* z knihovny *std* s názvem *filename* a bylo mu přiřazeno debugovací nastavení.

```
1 std::string filename = "Kernel.cl";
2 std::string oclDebugOptions = "-g -s \"" + filename + "\"";
3 oclDebugOptions = "";
```

Dále se program pokusil otevřít soubor. V případě neúspěchu vypsal chybovou hlášku. Pokud chyba nenastala, tak se pokusil načíst pomocí *ifstream* kernel do proměnné typu *string* s názvem *prog*.

```
4 std::ifstream file(filename);
5 checkErr(file.is_open() ? CL_SUCCESS:-1, "Kernel.cl");
6 std::string prog( std::istreambuf_iterator<char>(file),
  (std::istreambuf_iterator<char>() ) );
```

Po načtení kernelu byl vytvořen objekt *program* z knihovny *cl*.

```
7 cl::Program::Sources source(1, std::make_pair(prog.c_str(),
  prog.length()+1));
```

```
8 cl::Program program(context, source);
9 program.build(deviceList, oclDebugOptions.c_str() );
```

Po té se načetl kernel s názvem *„PSO_OpencCL_Kernel“*

```
10 cl::Kernel kernel(program, "PSO_OpenCL_Kernel", &err);
```

Kernelu musely být nastaveny argumenty (hodnoty), se kterými se bude volat.

Ukázka argumentu s bufferem XCL pro pole X.:

```
11 kernel.setArg(0, XCL);
```

Ukázka argumentu pro nastavení lokální proměnné pro pole X:

```
12 int groupSize = N;
13 kernel.setArg<cl::LocalSpaceArg>(5, cl::__local(D * groupSize *
   sizeof(cl_float)));
```

Dále byl vytvořen objekt *queue* pro příkazy.

```
14 cl::CommandQueue queue(context, device, 0, &err);
```

Před spuštěním kernelu byl vytvořen objekt *event*, který byl volán při spuštění.

```
15 cl::Event event;
```

A následně byl spuštěn samotný kernel. Paralelní rozdělení vláken bylo ponecháno na OpenCL.

```
16 queue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange(N),
   cl::NullRange, NULL, &event);
```

6.3.2 CUDA

6.3.2.1 Zpracování chybových stavů

Stejně jako u OpenCL sloužila pro zpracování chybových stavů při přechodu do paralelní části programu funkce, která byla volána při každém kroku, jenž by mohl způsobit chybu při přechodu do kernelu.

Ukázka volání funkce:

```
1 checkErr(err, "cudaMalloc failed!");
```

Funkce s výpisem chyby:

```
1 inline void checkErr(cudaError_t err, const char * name)
2 {
3     if (err != cudaSuccess)
4     {
5         std::cerr << "ERROR: " << name << " (" << err << ") - " <<
  cudaGetErrorString(err) << std::endl;
```

```
6        getchar();
7        exit(EXIT_FAILURE);
8    }
9 }
```

Při jakékoliv chybě bylo vypsáno, v jakém kroku se chyba objevila a číslo této chyby.

6.3.2.2 Volba výpočetního zařízení

Protože v testovacím notebooku bylo s CUDA kompatibilní pouze jedno zařízení. Z toho důvodu bylo zvoleno první detekované CUDA zařízení bez výpisu ostatních zařízení.

```
1 cudaSetDevice(0);
```

6.3.2.3 Alokace paměti a kopie dat do GPU pro pole s parametry hejna

Pro každé pole musela být alokována paměť pomocí *cudaMalloc*, a pak se pomocí *cudaMemcpy* musela nakopírovat do paměti data.

Ukázka *bufferu* pro pole souřadnic X:

```
1 cudaMalloc((void**)&X_d, parameters[1] * parameters[2] *
  sizeof(float));
2 cudaMemcpy(X_d, X, parameters[1] * parameters[2] * sizeof(float),
  cudaMemcpyHostToDevice);
```

6.3.2.4 Spuštění kernelu

Spuštění kernelu bylo doprovázeno také rozdělením threadů do bloků. Testovací GPU mělo 144 CUDA jader, a proto byla vytvořena tato podmínka: pokud by bylo vláken méně než 144, tak by se nastavil počet, který je potřeba. Pokud by ale bylo vláken více, tak by se tato vlákna musela dělit do bloků po 144 vláknech.

```
3 int threads = 144;
4 if((int)parameters[1]<144)
5 {
6    threads = parameters[1];
7 }
8
9 int grid = ceil(parameters[1]/144);
10 pso_kernel<<< grid, threads, parameters[1] * parameters[2] *
   sizeof(float)>>>(X_d, P_d, V_d, f_d, parameters_d);
```

6.4 PARALELNÍ ČÁST PROGRAMU

6.4.1 Generování náhodných čísel

Protože grafické karty neumožnují generovat náhodné čísla, bylo potřeba použít jiné řešení. V této práci byl problém řešen metodou, která pomocí již vygenerovaných čísel (seed) generuje další náhodná čísla. Byl využit generátor pseudonáhodných čísel ze zdroje [19].

Implementace této funkce byla provedena na OpenCL, CUDA i C++ AMP stejně.

Inicializace generátoru:

```
1 unsigned long seeds = (unsigned long)X(local_id,local_id); //using
  random numbers in population as seeds
2 random_state r = {0,0,0};
3 seed_random(r, seeds);
```

Struktura a funkce generátoru:

```
4 struct random_state
5 {
6     unsigned long a;
7     unsigned long b;
8     unsigned long c;
9 }
10 ;
```

Výpočet náhodného čísla:

```
11 unsigned long random(random_state& r)
12 {
13     unsigned long old = r.b;
14     r.b = r.a * 1103515245 + 12345;
15     r.a = (~old ^ (r.b >> 3)) - r.c++;
16     return r.b;
17 }
```

Přepočet na hodnotu mezi 0 a 1:

```
18 float random_01(random_state& r)
19 {
20     return (random(r) & 4294967295) / 4294967295.0f;
21 }
```

Seed:

```
22 void seed_random(random_state& r, unsigned long seed)
```

```
23 {
24   r.a = seed;
25   r.b = 0;
26   r.c = 362436;
27 }
```

6.4.2 Testovací funkce

Volba testovacích funkcí byla řešena pomocí příkazu *switch* a samotný výpočet pomocí *for* cyklu. Testovací funkce vyhodnotila vzorec pro všechny dimenze a vrátila výslednou hodnotu.

```
 1 float test_function(float* pos, int D, int function, float min,
   float max)
 2 {
 3    float return_value = 0;
 4
 5    for(int d = 0; d < D; ++d)
 6    {
 7       if( (parameters[d] < min) || (parameters[d] >max) )
 8    {
 9       return FLT_MAX;
10    }
11  }
12
13  switch(function)
14  {
15  //1st de jong
16  case 1:
17     {
18     for(int d = 0; d < D; d++)
19     {
20        return_value = return_value + pos[d] * pos[d];
21     }
22     break;
23     }
24  //2nd de jong
25  case 2:
26     {
27     for(int d = 0; d < D; d++)
28     {
```

```
29        return_value = return_value + (100*(((pos[d]*pos[d])-
   pos[d+1])*((pos[d]*pos[d])-pos[d+1]))+((1-pos[d])*(1-pos[d])));
30        }
31        break;
32        }
33   //3rd de jong
34   case 3:
35        {
36        for(int d = 0; d < D; d++)
37        {
38           return_value = return_value + fabs(pos[d]);
39        }
40        break;
41        }
42   default:
43        {
44        return_value = 999;
45        break;
46        }
47     }
48 return return_value;
49 }
```

6.4.3 Výpočetní část

Struktura:

V kernelu bylo využito všech proměnných, které byly uloženy do globální paměti grafického akcelerátoru již v host programu, a několik dalších pomocných proměnných.

Při výpočtu byla využita i lokální paměť GPU, protože je rychlejší než globální paměť. Z důvodu omezené velikosti lokální paměti byla do této paměti uložena pouze data pole X, do něhož se během výpočtu zapisovalo nejčastěji.

Pseudokód:

```
50 pro zadaný počet iterací
   pro každou částici
     spočítej fitness
     pokud je fitness menší než nejmenší dosud nalezený pro danou
     částici, tak jej aktualizuj
     aktualizuj rychlost a pozici všech částic
```

6.4.3.1 OpenCL

Proměnné kernelu:

```
1  __global float* X;             //pole X v globální paměti GPU
2  __global float* P;             //pole P v globální paměti GPU
3  __global float* V;             //pole V v globální paměti GPU
4  __constant float* parameters;  //pole parameters v globální paměti
   GPU
5  __global float* f;             //pole f v globální paměti GPU
6  __local float* lX;             //pole X v lokální paměti GPU

7  int Iter = (int)parameters[0];
8  int N = (int)parameters[1];
9  int D = (int)parameters[2];
10 float c1 = parameters[3];
11 float c2 = parameters[4];
12 float chi = parameters[5];

13 int global_id = get_global_id(0);    //index prvku
14 int local_id = get_local_id(0);      //index prvku


15 __local float* x = lX + local_id * D;  //ukazatel na aktuální pozici
   počítaného jedince v poli lX
16 __global float* p = P + global_id * D; //ukazatel na aktuální pozici
   počítaného jedince v poli P
17 __global float* v = V + global_id * D; //ukazatel na aktuální pozici
   počítaného jedince v poli V
18 __global float* pg = p;              //ukazatel na aktuální pozici
   počítaného jedince v poli p

19 int iB = global_id - 1;              //předchozí jedinec
20 int iF = global_id + 1;              //následující jedinec
21 float r1 = random_01(&r);            //random hodnota mezi 0-1
22 float r2 = random_01(&r);            //random hodnota mezi 0-1

23 float p1 = c1 * r1 * (p[d] -  x[d]); //parametr mutace
24 float p2 = c2 * r2 * (pg[d] - x[d]); //parametr mutace
```

Nahrazení stávající nejlepší hodnoty bodu novou:

Pokud výsledek testovací funkce splnil podmínku, že je menší než stávající hodnota, tak se pro všechny dimenze přepsaly pozice.

```
1 if(fitness < f[global_id])
2 {
3    f[global_id] = fitness;
4    for(int d = 0; d < D; d++)
5    {
6       p[d] = x[d];
7    }
8 }
```

Kontrola pozice počítaného prvku:

Protože byl paralelizován výpočet každého prvku zvlášť, musela být při průchodu prvků pole *X* kontrolována pozice předchozího a následujícího prvku. Pokud se program nacházel na prvním prvku v poli, tak se nastavil předchozí na poslední prvek a následující na druhý prvek v poli. Pokud se nacházel na posledním prvku v poli, předchozí nastavil na předposlední a následující na první prvek v poli.

```
1 if(global_id == 0)
2 {
3    iB = N - 1; iF = 1;
4 } else if( global_id == (N - 1) )
5 {
6    iB = N - 2;
7    iF = 0;
8 }
```

Přenastavení ukazatele na nejlepší výsledek prvku:

Pokud bylo *fitness* předchozího prvku menší než aktuálního, tak ukazatel odkazoval na adresu předchozího prvku a pokud bylo *fitness* následujícího prvku menší než aktuálního, tak ukazatel odkazoval na adresu následujícího prvku.

```
1 if(f[iB] < f[global_id])
2 {
3    pg = P + iB * D;
4 } else if(f[iF] < f[global_id])
5 {
6    pg = P + iF * D;
7 }
```

Aktualizace pozice rychlosti prvku:

Pro každý prvek se podle vztahů PSO aktualizuje pozice a rychlost. Využívá se k tomu náhodných čísel, předchozích výsledků a parametru χ (*chi*).

```
1 for(int d = 0; d < D; d++)
2 {
3    float r1 = random_01(&r);
4    float r2 = random_01(&r);

5    float p1 = c1 * r1 * (p[d] -  x[d]);
6    float p2 = c2 * r2 * (pg[d] - x[d]);
7    v[d] = chi * ( v[d] + p1 + p2 );
8    x[d] = x[d] + v[d];
9 }
```

6.4.3.2 CUDA

Proměnné kernelu:

Ukázka kódu:

```
1 float *X = 0;
2 float *P = 0;
3 float *V = 0;
 float *f = 0;
4 float *parameters = 0;
5 extern __shared__ float sX[];

6 int Iter = (int)parameters[0];
7 int N = (int)parameters[1];
8 int D = (int)parameters[2];
9 float c1 = parameters[3];
10 float c2 = parameters[4];
11 float chi = parameters[5];

12 int global_id = threadIdx.x;

13 float* x = sX + global_id * D;
14 float* p = P + global_id * D;
15 float* v = V + global_id * D;
16 float* pg = p;

17 int iB = global_id - 1;
18 int iF = global_id + 1;
19 float r1 = random_01(&r);
20 float r2 = random_01(&r);
```

```
21 float p1 = c1 * r1 * (p[d] -  x[d]);
22 float p2 = c2 * r2 * (pg[d] - x[d]);
```

Kromě proměnných byl zbytek výpočtu řešen stejným kódem jako u OpenCL.

6.4.3.3 C++ AMP

Proměnné kernelu:

Ukázka kódu:

```
 1 array_view<float,2> X(N * D, Xh);
 2 array_view<float,2> P(N * D, Ph);
 3 array_view<float,2> V(N * D, Vh);
 4 array_view<float> f(N, fh);
 5 tile_static float Xt[N][D];

 6 int local_id = t_idx.local[0];

 7 float* x = Xt[local_id];

 8 int iB = local_id - 1;
 9 int iF = local_id + 1;

10 float igb = local_id;

11 float r1 = random_01(&r);
12 float r2 = random_01(&r);

13 float p1 = c1 * r1 * (p[d] -  x[d]);
14 float p2 = c2 * r2 * (pg[d] - x[d]);
```

Zbytek výpočtu byl řešen stejně jako u OpenCL nebo CUDA s tím rozdílem, že C++ AMP umožňovalo využít jednorozměrného pole jako dvourozměrného pole. Díky tomu odpadla nutnost použít ukazatele s aktuální pozicí a jejich přepočet.

Nahrazení stávající nejlepší hodnoty bodu novou:

Ukázka kódu:

```
1 if(fitness < f[local_id])
2 {
```

```
3     f(local_id) = fitness;
4     for(int d = 0; d < D; ++d)
5     {
6        P(local_id, d) = Xt[local_id][d];
7     }
8  }
```

Kontrola pozice počítaného prvku:

Ukázka kódu:

```
1  int iB = local_id - 1; // Predchozi (Backward)
2  int iF = local_id + 1; // Nasledujici (Forward)

3  if(local_id == 0)
4  {
5    iB = N - 1;
6    iF = 1;
7  } else if(local_id == (N - 1) )
8  {
9    iB = N - 2;
10   iF = 0;
11 }
```

Přenastavení ukazatele na nejlepší výsledek prvku:

Ukázka kódu:

```
1  if(f(iB) < f(local_id))
2  {
3     igb = iB;
4  } else if(f(iF) < f(local_id))
5  {
6     igb = iF;
7  }
```

Aktualizace pozice rychlosti prvku:

Ukázka kódu:

```
1  for(int d = 0; d < D; ++d)
2  {
3     float r1 = random_01(r);
4     float r2 = random_01(r);
```

```
5     float p1 = c1 * r1 * (P(local_id,d) -  Xt[local_id][d]);
6     float p2 = c2 * r2 * (P(igb, d) - Xt[local_id][d]);
7     V(local_id,d) = chi * (V(local_id,d) + p1 + p2);
8     Xt[local_id][d] = Xt[local_id][d] + V(local_id,d);
9  }
```

6.5 Měření času výpočtu

Pro měření délky doby výpočtu byla použita funkce *clock()* a měřena doba, po kterou byl spuštěn kernel.

Ukázka měření a výpisu doby výpočtu do konzole:

```
1 diff = clock() - start;


2 výpočet


3 int msec = diff * 1000 / CLOCKS_PER_SEC;
4 std::cout << "Time taken " << msec/1000 << " seconds and " <<
  msec%1000  << " milliseconds\n" <<std::endl;
```

Přesnost funkce *clock()* byla cca 15ms, proto byl algoritmus spuštěn 30x pro každou testovací funkci a výsledky zprůměrovány.

6.6 Načtení dat z kernelu do host programu

6.6.1 OpenCL

```
1 queue.enqueueReadBuffer(fCL,CL_TRUE,0, sizeof(cl_float) * N, f);
```

6.6.2 CUDA

```
1 cudaMemcpy(f, f_d, parameters[1] * sizeof(float),
  cudaMemcpyDeviceToHost);
```

6.6.3 C++ AMP

C++ AMP nevyžadovalo žádné speciální funkce pro získání dat z grafického adaptéru, protože dokáže pracovat v host programu s paralelními proměnnými.

6.7 Uvolnění paměti po skončení výpočtu

Po ukončení výpočtu bylo potřeba uvolnit alokovanou paměť, aby nevznikaly tzv. memory leaky.

```
1 delete[] X;
2 delete[] P;
3 delete[] V;
4 delete[] f;
```

6.7.1 CUDA

U CUDA bylo nutné uvolnit paměť na GPU i v host programu:

Ukázka uvolnění paměti na GPU:

```
1 cudaFree(X_d);
2 cudaFree(P_d);
3 cudaFree(V_d);
4 cudaFree(f_d);
5 cudaFree(parameters_d);
```

Ukázka uvolnění paměti v host programu:

```
6 delete[] X;
7 delete[] P;
8 delete[] V;
9 delete[] f;
```

7 DETEKCE TDR

Během testování se vyskytl problém s bezdůvodnými pády aplikace vytvořené pomocí C++ AMP. Při hledání řešení byl nalezen mechanismus TDR, který tyto pády mohl způsobit.

Pro detekci TDR byla implementována metoda ze zdroje [20].

```
1 try
2 {
3    compute_kernel(data, results);
4    return true;
5 }
6 catch(concurrency::accelerator_view_removed& ex)
7 {
8    cout << "TDR exception received: " << ex.what();
9    cout << "Error code:" << std::hex << ex.get_error_code();
10   cout << "Removed reason:" << std::hex
11        << ex.get_view_removed_reason();
12   return false;
13 }
```

Při pádu byla vypsána chybová hláška *887a0005*, kterou způsobuje právě TDR, takže se potvrdil důvod pádů aplikace.

Pro tento problém byla nalezena tři řešení. Buď lze TDR ošetřit dvěma způsoby přímo ve vlastní aplikaci, nebo v operačním systému změnou registrů grafického ovladače.

7.1 Vyřazení TDR

7.1.1 Ošetření aplikace C++ AMP

7.1.1.1 Vytváření nových accelerator_view

Ve zdroji [20] bylo nalezeno řešení, které při TDR vytvoří nový *accelerator_view*, realokuje paměť a zavolá výpočetní algoritmus znova. Pokud nenastane chyba při vytváření *accelerator_view*, bude výpočet probíhat tak dlouho, dokud neskončí počet iterací.

```
1 bool compute(vector<int>& data, vector<int>& results)
2 {
3    accelerator device = accelerator();
4    try
5    {
6       compute_kernel(data, results, device);
```

```
 7    }
 8    catch(concurrency::accelerator_view_removed& ex)
 9    {
10       cout << "TDR exception received: " << ex.what() << endl;
11       cout << "Error code: " << std::hex << ex.get_error_code();
12       try
13       {
14          cout << "Retrying with immediate queuing_mode...";
15          compute_kernel(data, results, device,
   queuing_mode::queuing_mode_immediate);
16       }
17       catch(concurrency::accelerator_view_removed& ex)
18       {
19          cout << "TDR exception received: " << ex.what() << endl;
20          cout << "Error code:" << std::hex << ex.get_error_code() <<
   endl;
21          cout << "Removed reason: " << std::hex <<
   ex.get_view_removed_reason();
22          cout << "Aborting." << endl;
23          return false;
24       }
25    }
26    return true;
27 }
```

7.1.1.2 Vytvoření accelerator:view bez ošetření TDR

Ve zdroji [21] je popsáno, jak vytvořit *accelerator_view*, který má vypnutý TDR.

```
1 unsigned int createDeviceFlags =
  D3D11_CREATE_DEVICE_DISABLE_GPU_TIMEOUT;
2 ID3D11Device *pDevice;
3 ID3D11DeviceContext *pContext;
4 D3D_FEATURE_LEVEL featureLevel;
5 HRESULT hr = D3D11CreateDevice(pAdapter,
                                  D3D_DRIVER_TYPE_UNKNOWN,
                                  NULL,
                                  createDeviceFlags,
                                  NULL,
                                  0,
                                  D3D11_SDK_VERSION,
                                  &pDevice,
                                  &featureLevel,
```

```
                                    &pContext);
 6 if (FAILED(hr) ||
       ((featureLevel != D3D_FEATURE_LEVEL_11_1) &&
       (featureLevel != D3D_FEATURE_LEVEL_11_0)))
 7 {
 8     fprintf(stderr, "Failed to create Direct3D 11 device\n");
 9     return hr;
10 }

11     accelerator_view noTimeoutAcclView =
12     concurrency::direct3d::create_accelerator_view(pDevice);
```

7.1.2 Operační systém

TDR nemusí být ošetřeno jen v aplikacích, ale přímo v systémových registrech lze změnit nebo úplně vypnout TDR.

K upravení registrů je možné využít nástroje *regedit*, kde se dá TDR různě nastavit nebo vypnout. V této práci bylo zvoleno zpoždění doby aktivace TDR pomocí TdrDelay (REG_DWORD) v registrech na cestě HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\GraphicDrivers. Byl zvolen interval 60 sekund, který byl dostatečný pro použitou hardwarovou konfiguraci.

8 VÝSLEDKY SROVNÁNÍ

8.1 Parametry výpočtu

Zvoleno dle doporučení ze zdroje [14].

- Počet jedinců (N): 50
- Počet dimenzí (D): 30
- Počet Iterací (Iter): 300000
- Učící faktor (c1): 2,05
- Učící faktor (c2): 2,05
- Konstanta φ (lambda): 4,1
- Contriction factor (chi): 0,729843855
- Výpočetní hranice:
    - 1. De Jongova testovací funkce
        - Minimum (min): -100
        - Maximum (max): 100
    - 2. De Jongova testovací funkce
        - Minimum (min): -2,048
        - Maximum (max): 2,047
    - 3. De Jongova testovací funkce
        - Minimum (min): -2,048
        - Maximum (max): 2,047

8.2 Srovnání doby výpočtu

Tab. 1: Srovnání dob výpočtů z 30 měření na různých testovacích funkcích

	1. De Jongova testovací funkce [s]	2. De Jongova testovací funkce [s]	3. De Jongova testovací funkce [s]
OpenCL	35,397461 ± 0,002012	35,309330 ± 0,249751	32,489170 ± 0,248673
CUDA	23,368834 ± 0,172174	24,955963 ± 0,263675	23,384132 ± 0,129781
C++ AMP	43,630531 ± 0,036639	43,171963 ± 0,438750	43,599632 ± 0,030419

Z výsledků v Tab. 1 vyplývá, že CUDA bylo jednoznačně nejrychlejší. Obdobných výsledků jako CUDA dosáhlo i OpenCL. C++ AMP hodně zaostalo.

8.3 Ověření správnosti výsledků

Pro všechny testované funkce byl nalezen extrém funkce v bodě s hodnotou 0,000000 bez žádných odchylek. Lze tedy tvrdit, že extrémy byly nalezeny zcela přesně.

8.4 Srovnání standardů OpenCL, CUDA a C++ AMP

Tab. 2: Srovnání standardů OpenCL, CUDA a C++ AMP

	Rychlost výpočtu	Přenositelnost kódu	Intuitivnost implementace
OpenCL	★★	★★★	★
CUDA	★★★	★★	★★
C++ AMP	★	★	★★★

V Tab. 2 je znázorněno srovnání standardů OpenCL, CUDA a C++ AMP z různých úhlů pohledu – rychlosti výpočtu, přenositelnosti kódu mezi různými platformami a intuitivnosti implementace, která souvisí i s kvalitou dokumentace k danému standardu. Jako hodnoty hodnocení jsou využity symboly ★, kdy jeden symbol značí nejhorší hodnocení a tři symboly nejlepší.

ZÁVĚR

Hlavním cílem této práce bylo vytvořit a srovnat implementace evolučního algoritmu PSO v CUDA, OpenCL a C++ AMP. K řešení tohoto problému bylo využito verze PSO s lbest topologií a nastavení parametrů algoritmu dle doporučení ze zdroje [14].

Teoretická část práce se zaobírá popisem koncepce třech standardů pro heterogenní programování kompatibilní s grafickou kartou NVIDIA GT555M, představením evolučních algoritmů, detailním popisem algoritmu PSO a několika testovacích funkcí.

V praktické části jsou popsány jednotlivé implementace a výsledky jejich srovnání. Všechny implementace byly navrhnuty stejným způsobem, aby byla zajištěna možnost testování doby výpočtu.

Celý vývoj i následné srovnání jednotlivých implementací probíhalo na notebooku s procesorem Intel® Core™ i3-2330M Processor (3M Cache, 2.20 GHz) a grafickou kartou NVIDIA GeForce GT 555M.

Na srovnání CUDA, OpenCL a C++ AMP se dá pohlížet z více pohledů, nelze jednoznačně určit, který z těchto standardů je nejlepší. Z pohledu na dobu výpočtu vyhrává jednoznačně CUDA. OpenCL je na druhém místě a nejpomalejší je C++ AMP. Když ale bude potřeba mít program na více platformách, tak je nejlepší OpenCL, protože podporuje širokou škálu výpočetních zařízení a operačních systémů. CUDA podporuje jen grafické čipy s podporou CUDA od firmy NVIDIA, ale podporuje operační systémy Linux, OSX a Windows. C++ AMP je omezen na grafické čipy s podporou DirectX 11 a operační systém Windows 7 nebo Windows Server 2008 R2 a novější. Délka kódu a složitost implementace je také důležitým faktorem, který rozhoduje o tom, jaký standard zvolit. Nejjednodušší a nejkratší implementace je určitě C++ AMP – obdoba implementace C++. Dlouhý kód, ale spoustu materiálů, které napomáhají k zjednodušení implementace má CUDA. Implementace v OpenCL nebyla o moc delší než v CUDA, ale implementace byla o dost náročnější.

Cíle práce byly splněny. Z pohledu na dobu výpočtu zvítězil standard CUDA, nejlepší přenositelnost má OpenCL a nejsnazší implementace je možná pomocí C++ AMP. Veškeré zdrojové kódy jsou dostupné na přiloženém CD.

ZÁVĚR V ANGLIČTINĚ

The aim of this work was to develop and compare evolutionary algorithm PSO implementation in CUDA, OpenCL and C + + AMP. For solving this problem was used version of PSO with lbest topology and algorithm parameters were set according to the recommendation of resource [14].

The theoretical part of the thesis deals with the description of the concept of three standards for heterogeneous programming compatible with the NVIDIA GT555M graphics card, the introduction to evolutionary algorithms, detailed description of the PSO algorithm and several test functions.

The practical part describes all implementations and compares them among themselves. All implementations were designed in the same way as to ensure the possibility of testing the time of calculation.

The entire development and subsequent comparison of the implementation was carried out on a laptop with an Intel ® Core ™ i3-2330 Processor (3M Cache, 2.20 GHz) and NVIDIA GeForce GT 555m.

The comparison of CUDA, OpenCL and C + + AMP can be viewed from multiple perspectives, it's impossimble to say which of these standards is the best. From the perspective of the computing time CUDA clearly wins. OpenCL is the second slowest and slowest is the C + + AMP. But when you will need to have a program on multiple platforms, so it is best to OpenCL, it supports a wide range of computing devices and operating systems. CUDA supports only graphics chips with support for CUDA from NVIDIA, but supports multiple operational systems - Linux, OSX and Windows. C++ AMP is limited to GPUs with support for DirectX 11 and Windows 7 or Windows Server 2008 R2 and later. Code length and complexity of implementation is also an important factor that decides which standard to choose. The easiest and shortest implementation is definitely C + + AMP - similar to C + + implementation. Code length, but a lot of materials that help to simplify the implementation of CUDA. Implementation in OpenCL was not much longer than in CUDA, but the implementation was much more difficult.

The objectives of the thesis were fulfilled. From the perspective of the computing time standard CUDA is winner, OpenCL has the best portability and using C + + AMP has the easiest implementation is. All source codes are available on the enclosed CD.

SEZNAM POUŽITÉ LITERATURY

[1] Paralelní počítání (1): Paralelní počítače a jejich základní principy. *České vysoké učení technické v Praze* [online]. 2012 [cit. 2013-05-09]. Dostupné z: http://popular.fbmi.cvut.cz/it/Stranky/Paralelni-pocitani-1-paralelni-pocitace-a-jejich-zakladni-principy.aspx

[2] *OpenCL Programming Guide*. Upper Saddle River: Addison-Wesley, c2012, xliv, 603 s. ISBN 978-0-321-74964-2.

[3] *Heterogeneous computing with OpenCL*. Amsterdam: Morgan Kaufmann, 2012, xvi, 277 s. parallel programming/computer architecture. ISBN 978-0-12-387766-6.

[4] NVIDIA. *NVIDIA CUDA C Programming Guide*. Version 4.2. California, 2012.

[5] NVIDIA. *NVIDIA CUDA GETTING STARTED GUIDE FOR MICROSOFT WINDOWS*. California, 2012 [cit. 2013-05-09]. Dostupné z: http://developer.download.nvidia.com/compute/cuda/5_0/rel/docs/CUDA_Getting_Started_Guide_For_Microsoft_Windows.pdf

[6] NVIDIA CUDA Introduction - Page 3. *Beyond3D* [online]. 2007 [cit. 2013-05-09]. Dostupné z: http://www.beyond3d.com/content/articles/12/3

[7] INSTITUTE OF COMPUTATIONAL SCIENCE OF THE COMPUTER SCIENCE DEPARTMENT OF ETH ZURICH. *CUDA Memory Architecture*. Švýcarsko, 2011 [cit. 2013-05-09]. Dostupné z: http://www.cvg.ethz.ch/teaching/2011spring/gpgpu/cuda_memory.pdf

[8] GREGORY, Kate. *C AMP: accelerated massive parallelism with Microsoft Visual C*. California: O'Reilly Media, Inc., 2012, xxiv, 326 pages. ISBN 978-073-5664-739.

[9] Parallel Programming in Native Code: accelerator_view::create_marker in C++ AMP. MICROSOFT CORPORATION. *Microsoft Developer Network* [online]. 2012 [cit. 2013-05-09]. Dostupné z: http://blogs.msdn.com/b/nativeconcurrency/archive/2012/02/28/accelerator-view-create-marker-in-c-amp.aspx

[10] Introduction to Tiling in C++ AMP. MICROSOFT CORPORATION. *Microsoft Developer Network* [online]. 2012 [cit. 2013-05-09]. Dostupné z: http://msdn.microsoft.com/en-us/magazine/hh882447.aspx

[11] Timeout Detection and Recovery of GPUs through. MICROSOFT CORPORATION. *Microsoft Developer Network* [online]. 2009 [cit. 2013-05-09]. Dostupné z: http://msdn.microsoft.com/en-us/windows/hardware/gg487368

[12] ZELINKA, Ivan. *Aplikace umělé inteligence*. Vyd. 1. Zlín: Univerzita Tomáše Bati ve Zlíně, 2010, 151 s. ISBN 978-80-7318-898-6.

[13] HU, Xiaohui. *Particle Swarm Optimization* [online]. 2006 [cit. 2013-01-24]. Dostupné z: http://www.swarmintelligence.org/

[14] Proceedings of the 2007 IEEE Swarm Intelligence Symposium (SIS 2007): Defining a Standard for Particle Swarm Optimization. In: BRATTON, Daniel a James KENNEDY. *Max planck institut informatik* [online]. 2007 [cit. 2013-03-14]. Dostupné z: https://svn-d1.mpi-inf.mpg.de/AG1/MultiCoreLab/papers/BrattonKennedy07%20-%20Defining%20a%20Standard%20for%20PSO.pdf

[15] MOLGA, Marcin a Czesław SMUTNICKI. Test functions for optimization needs. In: *Optimisation Techniques in Bioinformatics and Systems Biology* [online]. 2005 [cit. 2013-05-09]. Dostupné z: http://www.bioinformaticslaboratory.nl/twikidata/pub/Education/NBICResearchSchool/Optimization/VanKampen/BackgroundInformation/TestFunctions-Optimization.pdf

[16] TVRDÍK, Josef. OSTRAVSKÁ UNIVERZITA V OSTRAVĚ. *Stochastické algoritmy pro globální optimalizaci*. 1. vyd. Ostrava, 2010 [cit. 2013-05-09]. Dostupné z: http://www1.osu.cz/~tvrdik/down/files/STAGO_10.pdf

[17] Intel SDK for OpenCL* Applications 2013. INTEL CORPORATION. *Intel Software* [online]. 2013 [cit. 2013-05-13]. Dostupné z: http://software.intel.com/en-us/vcsource/tools/opencl-sdk

[18] Nsight Visual Studio Edition Downloads. NVIDIA CORPORATION. *NVIDIA DEVELOPER ZONE* [online]. 2012 [cit. 2013-05-13]. Dostupné z: https://developer.nvidia.com/rdp/nsight-visual-studio-edition-early-access

[19] HENDERSON, Ian. Pseudorandom Number Generation. In: *EECS Instructional and Electronics Support* [online]. 2012 [cit. 2012-02-11]. Dostupné z: http://inst.cs.berkeley.edu/~ianh

[20] Parallel Programming in Native Code: Handling TDRs in C++ AMP. MICROSOFT CORPORATION. *Microsoft Developer Network* [online]. 2012 [cit. 2013-05-09]. Dostupné z: http://blogs.msdn.com/b/nativeconcurrency/archive/2012/03/07/handling-tdrs-in-c-amp.aspx

[21] AGARWAL, Amit. Disabling TDR on Windows 8 for your C++ AMP algorithms. MICROSOFT CORPORATION. *Parallel Programming in Native Code* [online]. 2012 [cit. 2013-05-11]. Dostupné z: http://blogs.msdn.com/b/nativeconcurrency/archive/2012/03/01/disabling-tdr-on-windows-8-for-your-c-amp-algorithms.aspx

[22] SCARPINO, Matthew. *OpenCL in action: how to accelerate graphics and computation*. Shelter Island: Manning, 2012, xxii, 434 s. ISBN 978-1-617290-17-6.

[23] OpenCL Reference Pages. *Khronos* [online]. 2013 [cit. 2013-05-08]. Dostupné z: http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/

[24] KRAMPL, Jakub. *Implementace vybraných evolučních algoritmů v prostředí .NET*. Zlín, 2011. Bakalářská práce. Univerzita Tomáše Bati ve Zlíně.

[25] JANOŠTÍK, Jakub. *Implementace vybraných evolučních algoritmů v OpenCL*. Zlín, 2012. Bakalářská práce. Univerzita Tomáše Bati ve Zlíně.

[26] PRATA, Stephen. *Mistrovství v C++*. 2. aktualiz. vyd. Brno: Computer Press, 2004, xxvi, 1006 s. ISBN 80-251-0098-7.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

API	Application Programming Interface
C++ AMP	C++ Accelerated Massive Parallelism
CPU	Computer Processor Unit
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
I/O	Input/Output
ID	Identifikační člen
OpenCL	Open Computing Language
PSO	Particle Swarm Optimization
TDR	Timeout Detection and Recover

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH