VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

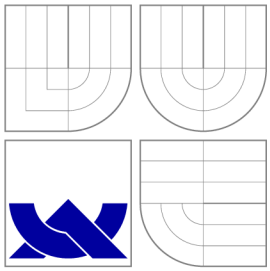
AKCELERACE ČÁSTICOVÝCH ROJŮ PSO POMOCÍ GPU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

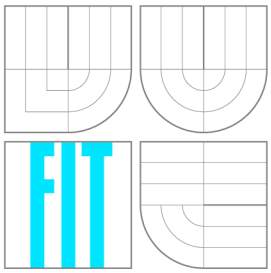
AUTOR PRÁCE
AUTHOR

Bc. DRAHOSLAV ZÁŇ

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE ČÁSTICOVÝCH ROJŮ PSO POMOCÍ GPU

PARTICLE SWARM OPTIMIZATION ON GPUS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. DRAHOSLAV ZÁŇ
AUTHOR

VEDOUCÍ PRÁCE — Ing. JIŘÍ JAROŠ, Ph.D.
SUPERVISOR

BRNO 2013

Abstrakt

Tato práce sa zabývá populačně založenou stochastickou optimalizační technikou PSO (Particle Swarm Optimization) a její akcelerací. Jedná se o jednoduchou, ale velmi efektivní techniku, určenou k řešení složitých multidimenzionálních problémů, která nachází uplatnění v široké oblasti aplikací. Cílem práce je vytvořit paralelní implementaci tohoto algoritmu s důrazem na co nejvyšší zrychlení výpočtu. K tomuto účelu byla zvolena grafická karta (GPU), která v dnešních dobách poskytuje cenově dostupný, masivní výpočetní výkon. Za účelem vyhodnocení přínosu akcelerace s využitím GPU byly vytvořeny a porovnávány dvě aplikace řešící problém odvozený od známého NP-těžkého problému Knapsack. Akcelerovaná aplikace na GPU vykazuje až 5-násobné průměrné a téměř 10-násobné maximální zrychlení výpočtu oproti optimalizované aplikaci pro vícejádrový procesor, ze které vycházela.

Abstract

This thesis deals with a population based stochastic optimization technique PSO (Particle Swarm Optimization) and its acceleration. This simple, but very effective technique is designed for solving difficult multidimensional problems in a wide range of applications. The aim of this work is to develop a parallel implementation of this algorithm with an emphasis on acceleration of finding a solution. For this purpose, a graphics card (GPU) providing massive performance was chosen. To evaluate the benefits of the proposed implementation, a CPU and GPU implementation were created for solving a problem derived from the known NP-hard Knapsack problem. The GPU application shows 5 times average and almost 10 times the maximum speedup of computation compared to an optimized CPU application, which it is based on.

Klíčová slova

PSO, HPC, CPU, GPU, GPGPU, CUDA, Knapsack, MKP, optimalizace, roj, částice

Keywords

PSO, HPC, CPU, GPU, GPGPU, CUDA, Knapsack, MKP, optimalization, swarm, particle

Citace

Drahoslav Záň: Akcelerace částicových rojů PSO pomocí GPU, diplomová práce, Brno, FIT VUT v Brně, 2013

Akcelerace částicových rojů PSO pomocí GPU

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jiřího Jaroše, Ph.D. a uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Drahoslav Záň
17. května 2013

Poděkování

Děkuji svému vedoucímu Ing. Jiřímu Jarošovi, Ph.D. za cenné rady a pomoc při řešení této práce.

Obsah

1. Úvod

Napriek zvyšujúcemu sa výkonu dnešných počítačov existujú stále problémy, ktorých riešenie je extrémne výpočtovo alebo pamäťovo náročné. Na riešenie takto zložitých problémov už „bežné“ postupy využívajúce numerické metódy, nestačia. Existujú však heuristické metódy, ktoré tieto problémy dokážu riešiť.

Jednou z týchto metód je aj optimalizačná technika využívajúca časticové roje (*Particle Swarm Optimization* – PSO). Táto technika vznikla v snahe napodobňovať správanie vypozorované v prírode, konkrétne bola inšpirovaná sociálnym správaním kŕdľa vtákov. Schopnosť tejto techniky riešiť problémy je založená na zdieľaní informácie v rámci spoločenstva. To potom vykazuje vyššiu mieru inteligencie ako jedinci, z ktorých je toto spoločenstvo zložené.

Algoritmus PSO je veľmi jednoduchý a efektívny, ale čas strávený hľadaním optimálneho riešenia rastie s dimenziou riešeného problému. Vzhľadom k tomu, že častice sú na sebe nezávislé, je možné tento algoritmus s výhodou paralelizovať a dosiahnuť tak výrazne vyššieho výkonu.

Táto práca sa zaoberá algoritmom PSO a jeho akceleráciou využitím grafickej karty. Cieľom tejto práce je vytvoriť paralelnú implementáciu aplikácie, ktorá využíva PSO k riešeniu zvoleného problému. Na demonštrovanie prínosu akceleráciou bol zvolený *Multidimensional Knapsack Problem* (MKP), ktorý je zobecnením známeho NP-ťažkého problému Knapsack. Problém MKP je možné chápať ako problém alokácie zdrojov. Alokovaním zdroja je navýšený profit, ale takisto dôjde k odčerpaniu obnosu z vyhradeného rozpočtu. Problémom je maximalizácia zisku v limitovanom rozpočte. Tento problém je nie len dobrým výkonnostným testom, ale aj nástrojom na riešenie reálnych problémov, ktoré sú naň redukovateľné.

Grafická karta prešla dlhým vývojom od špecializovaného adaptéru určeného k syntéze obrazu až ku koprocesoru reprezentujúceho obecnú výpočtovú platformu. Behom tejto doby dramaticky narástol jej výpočtový výkon a takisto sa zväčšila veľkosť dostupnej pamäte. K zvoleniu si tejto práce ma motivovalo jednak zoznámenie sa s tvorbou aplikácií zameraných na dosiahnutie vysokého výkonu s využitím tejto platformy, ale aj celkové zdokonalenie sa v tvorbe efektívneho kódu.

2. Optimalizačná technika využívajúca časticové roje PSO

Optimalizácia pomocou časticových rojov (*Particle Swarm Optimization* – PSO) je populačne založená stochastická technika inšpirovaná sociálnym správaním kŕdľa vtákov, prípadne húfu rýb. Jej úlohou je riešenie zložitých problémov veľkej dimenzie, ktorých riešenie je pomocou numerických metód príliš náročné. Stochastická povaha tohto algoritmu však nezaručuje nájdenie optimálneho riešenia. Pre PSO ale existujú rôzne stratégie, ktorých úlohou je „vyhýbať sa nástrahám“ v podobe lokálnych optím.

PSO je podobné evolučným výpočtovým technikám ako napr. Genetické Algoritmy (GA), v ktorých je systém inicializovaný populáciou náhodných riešení a v procese aktualizácie generácií hľadá riešenie. Na rozdiel od GA, PSO nemá evolučné operátory kríženia a mutácie. V algoritme PSO figurujú častice reprezentujúce potenciálne riešenia. V procese hľadania riešenia tieto častice nasledujú v priestore riešeného problému súčasné optimálne častice [22].

2.1 História

Technika PSO bola pôvodne navrhnutá a prvýkrát predstavená Eberhartom a Kennedym v roku 1995. Postupne sa vyvinula od sociálneho simulátora až do optimalizátora.

Systém častíc prezentovaný Eberhartom a Kennedym však nebol úplne neznámy. Už v roku 1983 použil Reevs jednoduchý časticový systém na modelovanie dynamických objektov z oblasti počítačovej grafiky. Tieto objekty boli obtiažne reprezentované pomocou polygónov alebo plôch. Patria medzi ne napríklad oheň, dym, voda alebo oblaky. V týchto systémoch, takisto ako v PSO, figurujú častice, ktoré sú na sebe nezávislé a ich pohyb je riadený množinou pravidiel. Neskôr v roku 1987 uskutočnil Reynolds simuláciu podobnú tej, z ktorej vzišiel algoritmus PSO a to simuláciu kolektívneho správania kŕdľa vtákov. Podobnú simuláciu takisto uskutočnili Heppner a Grenander v roku 1990 [8].

Reynolds a Heppner vo svojich simuláciách zistili, že lokálne procesy (častokrát modelované celulárnymi automatmi) môžu byť podstatou nepredikovateľnej dynamiky sociálneho správania vtákov. Obidva ich modely sa silne spoliehali na individuálne vzdialenosti medzi jedincami spoločenstva. Synchronizácia správania kŕdľa bola teda považovaná za funkciu vtáčieho úsilia udržať si optimálnu vzdialenosť medzi sebou a ich susedmi [20]. Pre vznik algoritmu PSO bola taktiež zásadná hypotéza z oblasti sociobiológie – zdieľanie informácií medzi príslušníkmi sociálneho spoločenstva predstavuje evolučnú výhodu.

Obrázok 2.1: Pôvod PSO (kŕdeľ vtákov) [11].

2.2 Podstata a vývoj pôvodného algoritmu

Jedným z motívov na vývoj algoritmu PSO bolo modelovanie sociálneho správania ľudí, ktoré je odlišné od správania kŕdľa vtákov alebo húfu rýb. Najdôležitejším rozdielom je jeho abstrakcia. Ľudia okrem fyzického pohybu uplatňujú takisto poznanie a skúsenosti, zdieľajú názory a presvedčenia.

Táto abstrakcia teda umožňuje pozerať na pohyb ľudí ako na zaujatie určitého stanoviska a nie ako na fyzický pohyb, v ktorom by mohlo dôjsť ku kolízii. Koncept *zmeny* v sociálnom spoločenstve ľudí je teda analogický ku vtáčiemu alebo rybiemu *pohybu*. [20].

Vývoj algoritmu začal ako simulácia zjednodušeného sociálneho životného prostredia, v ktorom figurovali bezkolízne vtáky (preberali teda určitú mieru spomínanej abstrakcie) nazývané agenti. Nebolo teda nutné zaoberať sa prípadom, v ktorom by sa dvaja jedinci zrazili. Účelom vyvinutého algoritmu bolo simulovať nepredikovateľnú choreografiu kŕdľa vtákov.

Keďže agenti sa pohybovali priestorom, mali okamžitú polohu a rýchlosť. Spôsob aktualizácie týchto veličín v čase (iteráciách) predstavuje algoritmus PSO. Nasledujúce podkapitoly sa venujú myšlienkovému procesu autorov algoritmu pri jeho vývine.

Porovnávanie rýchlosti s najbližším susedom a šialenstvo

Populácia agentov bola náhodne inicializovaná v 2D priestore tvaru torusu s pozíciou na mriežke pixelov a rýchlosťou v obidvoch súradniciach. S každou iteráciou bola pre každého agenta prepočítaná jeho rýchlosť na základe jemu najbližšieho agenta.

Toto vytvorilo synchronizáciu, avšak kŕdeľ po krátkom čase prestal meniť smer pohybu. Autori z tohto dôvodu zaviedli stochastickú premennú nazývanú šialenstvo, ktorá pridala do systému dostatok variácie.

Vektor kukuričného poľa

Neskôr sa autori inšpirovali myšlienkou Heppnera, ktorý do svojich simulácií pridal hniezdo, čo v nich vytvorilo dynamickú silu. Vtáky sa v simuláciách pohybovali okolo hniezda (pixel na obrazovke) až kým tam „nepristáli“. Toto eliminovalo potrebu stochastickej zložky šialenstva, ale nepostihlo reálne správanie vtákov, keďže tie pristanú na mieste, ktoré vyhovuje ich okamžitým potrebám, napr. tam, kde sa nachádza potrava.

V ďalšej variante simulácie bol zavedený dvoj-dimenzionálny „vektor kukuričného poľa“. Každý z agentov bol naprogramovaný, aby ohodnocoval svoju pozíciu na základe rovnice [20]:

$$Eval = \sqrt{(presentx - 100)^2} + \sqrt{(presenty - 100)^2} \tag{2.1}$$

V pozícii (100, 100) bola hodnota rovnice nulová. Táto pozícia predstavovala simulované kukuričné pole.

Každý agent si „zapamätal“ najlepšiu hodnotu a odpovedajúcu pozíciu, ktorá viedla k tejto hodnote. Táto hodnota sa nazývala $pbest[]$ a k nej odpovedajúce pozície $pbestx[]$ a $pbesty[]$ (jedná sa o polia o veľkosti počtu agentov). Každý z agentov si pri pohybe priestorom upravoval svoju rýchlosť na základe svojej aktuálnej a najlepšej polohy tak, aby sa od svojej najlepšej polohy príliš nevzdialil. Ak bol vpravo od svojej $pbestx$, tak sa od jeho x-ovej zložky rýchlosti (vx) odpočítala náhodná hodnota váhovaná parametrom systému (funkcia $rand()$ v tejto a nasledujúcich rovniciach predstavuje generátor náhodných čísiel s uniformným rozložením v intervale $\langle 0, 1 \rangle$) [20]:

$$vx[] = vx[] - rand() * p\_increment \tag{2.2}$$

Ak bola vľavo, váhovaná náhodná hodnota bola k $vx[]$ pripočítaná. Úprava y-ovej zložky rýchlosti ($vy[]$) na základe $pbesty$ bola analogická.

Každý agent mal ďalej znalosť globálne najlepšej pozície v systéme. Toto bolo dosiahnuté zavedením hodnoty $gbest$, ktorá bola indexom agenta s najlepšou hodnotou, do poľa agentov. Rýchlosť ($vx[]$ a $vy[]$) pre každého agenta bola nasledovne upravená ($g\_increment$ je parametrom systému) [20]:

$$presentx[] > pbestx[gbest] \Rightarrow vx[] = vx[] - rand() * g\_increment \tag{2.3}$$
$$presentx[] < pbestx[gbest] \Rightarrow vx[] = vx[] + rand() * g\_increment \tag{2.4}$$
$$presenty[] > pbesty[gbest] \Rightarrow vy[] = vy[] - rand() * g\_increment \tag{2.5}$$
$$presenty[] < pbesty[gbest] \Rightarrow vy[] = vy[] + rand() * g\_increment \tag{2.6}$$

Sledovaním simulácie bolo vypozorované, že ak boli parametre systému $p\_increment$ a $g\_increment$ nastavené na príliš vysoké hodnoty, kŕdeľ bol silne vtiahnutý do kukuričného poľa. Ak boli nastavené na nízke hodnoty, kŕdeľ realisticky krúžil okolo cieľa, až nakoniec pristál.

Eliminácia pomocných premenných

Po tom, čo bolo jasné, že zvolená paradigma dokáže optimalizovať jednoduché dvoj-dimenzionálne lineárne funkcie, vznikla snaha o redukciu nepotrebných premenných a zachovanie len nutných častí dôležitých na splnenie účelu.

Nutnými časťami boli premenné *pbest* a *gbest*. Premenná *pbest* predstavovala autobiografickú pamäť (každý jedinec si zapamätal svoju vlastnú prežitú skúsenosť) a s ňou asociovaná úprava rýchlosti predstavovala mieru „nostalgie“ jedinca (každý jedinec mal tendenciu vracať sa na miesto, kde bol v minulosti najspokojnejší). Na druhej strane *gbest* predstavuje publikovanú znalosť alebo štandard, ktorý sa každý jedinec snaží dosiahnuť. Príliš veľký pomer *p_increment* ku *g_increment* viedol k relatívnej izolácii jedinca v roji a naopak pri väčšom *g_increment* sa jedinci predčasne upli k lokálnemu optimu. Rovnaké hodnoty *p_increment* a *g_increment* viedli k najefektívnejšiemu prehľadávaniu priestoru problému [20].

Prehľadávanie multidimenzionálneho priestoru

Keďže jedným z motívov vývoja algoritmu bola simulácia sociálneho správania ľudí (multidimenzionálny bezkolízny priestor), došlo k nahradeniu jedno-dimenzionálnych polí $presentx[]$, $pbestx[]$, $vx[]$ (podobne pre y-ovú súradnicu) maticami $N \times D$, kde D je počet dimenzií a N je počet agentov. Matice nadobudli tvar $presentx[][]$, kde prvou zložkou je agent a druhou dimenzia.

Akcelerácia vzdialenosťou

Napriek tomu, že algoritmus už fungoval dobre, mal ešte niektoré estetické nedokonalosti. Úprava rýchlosti bola založená na teste nerovnosti: ak $presentx > bestx \Rightarrow$ pridaj, ak $presentx < bestx \Rightarrow$ uber. Tento test bol nahradený postupnou úpravou rýchlosti na základe rozdielu aktuálnej polohy od najlepšej polohy (pre každého agenta a dimenziu z matice rýchlosti). [20]:

$$\begin{aligned} vx[][] = {} & vx[][] + \\ & rand() * p\_increment * (pbestx[][] - presentx[][]) + \\ & rand() * g\_increment * (gbestx[][] - presentx[][]) \end{aligned} \tag{2.7}$$

Finálna zjednodušená verzia

V poslednej verzii algoritmu boli odstránené *p_increment* a *g_increment* parametre roja, pretože neexistoval spôsob, ako správne určiť ich hodnotu. Stochastický faktor rovnice aktualizácie rýchlosti bol vynásobený číslom 2, aby sa dosiahlo priemernej hodnoty 1 (agenti „preletia“ svoj cieľ približne v polovici prípadov). Rovnica mala teda nakoniec tvar [20]:

$$\begin{aligned} vx[][] = {} & vx[][] + \\ & 2 * rand() * (pbestx[][] - presentx[][]) + \\ & 2 * rand() * (pbestx[][gbest] - presentx[][]) \end{aligned} \tag{2.8}$$

Pokusy o ďalšie vylepšenia zjednodušenej verzie

Autori algoritmu skúšali ďalšie experimenty so zjednodušenou verziou, ktoré však už neprniesli žiadne zlepšenie. Jednalo sa napr. o zlúčenie pozícií *pbest* a *gbest* do jednej pozície medzi nimi. Táto verzia však konvergovala aj k neoptimálnym pozíciám.

Ďalším pokusom autorov bolo vytvorenie dvoch typov agentov – „prieskumníkov“ a „osadníkov“. Prieskumníci na aktualizáciu rýchlosti používali test nerovnosti, osadníci rozdiel od najlepšej pozície. Hypotézou bolo, že prieskumníci sa dostanú výrazne ďalej za „známy“ región, kým osadníci budú tento región podrobne preskúmavať. Ani táto verzia však nepriniesla žiadne zlepšenie oproti zjednodušenej verzii.

Odstránenie zotrvačnosti $vx[][]$ z rovnice 2.8 sa ukázalo tiež ako nevhodné, pretože nebolo efektívne v hľadaní globálne optimálnych pozícií.

2.3 Základný algoritmus

Modernizovaním pôvodného algoritmu sa do systému opäť zaviedli parametre. Voľba týchto parametrov je väčšinou empirická, avšak existujú publikácie [36] na tému ich vhodnej voľby. Každá častica v roji je reprezentovaná vektorom (polohou $\vec{x}_k^i$) v hyperpriestore problému a pohybuje sa rýchlosťou ($\vec{v}_k^i$), ktorá je aktualizovaná rovnicou:

$$\vec{v}_{k+1}^i = w\vec{v}_k^i + \varphi_1\delta_1\left(\vec{p}_k^i - \vec{x}_k^i\right) + \varphi_2\delta_2\left(\vec{p}_k^g - \vec{x}_k^i\right) \tag{2.9}$$

Vektor $\vec{v}_{k+1}^i$ značí rýchlosť častice i v nasledujúcej iterácii $k+1$. Parameter w je váhou zotrvačnosti častice, jeho úlohou je pridávať ďalšiu variabilitu do systému častíc. Tento parameter môže figurovať ako konštanta, prípadne môže byť v každej iterácii volený stochasticky. J. C. Bansal a ďalší [5] prezentujú niektoré stratégie voľby tohto parametra. Ďalšími parametrami systému sú φ_1 (sociálny) a φ_2 (kognitívny) parameter. Vektor $\vec{p}_k^i$ značí polohu s najlepším ohodnotením pre časticu i. Častica g je časticou s najlepším ohodnotením v celom systéme. Symboly $\delta_{1,2}$ predstavujú uniformne distribuované náhodné čísla v intervale $\langle 0, 1\rangle$. Poloha častice je vypočítaná na základe rovnice:

$$\vec{x}_{k+1}^i = \vec{x}_k^i + \vec{v}_{k+1}^i \tag{2.10}$$

Proces hľadania optimálneho riešenia je iteratívny s fixným počtom iterácií, alebo pokým nie je splnené zvolené kritérium minimálnej chyby riešenia (napr. počet iterácií, v ktorých nedôjde k nájdeniu lepšieho riešenia). Celý algoritmus PSO je možné zachytiť pseudokódom 1.

2.4 Diskrétna varianta algoritmu

V základnom algoritme je poloha každej častice vyjadrená vektorom reálnych čísiel, tj. je určená na riešenie spojitých úloh. V diskrétnej binárnej verzii KBPSO [21] je poloha častice reprezentovaná binárnym vektorom $(x_{i1}, x_{i2}, \ldots, x_{ij}, \ldots, x_{in}) = \vec{x}^{\,i}$, pri probléme veľkosti n-dimenzií. Element v_{ij} z rýchlosti častice $(v_{i1}, v_{i2}, \ldots, v_{ij}, \ldots, v_{in}) = \vec{v}^{\,i}$ vyjadruje

Pseudokód 1 PSO

Initialization
    Initialize all particles with random positions and velocities

Optimization
    For $k = 0$ to number of iterations
        For each particle i
            Evaluate fitness function $f(\vec{x}_k^i)$
            Update $\vec{p}_k^i$ if $\vec{x}_k^i$ has better fitness
        End
        Update $\vec{p}_k^g$ according to best fitness found
        For each particle i
            Update velocity $\vec{v}_{k+1}^i$ according to equation 2.9
            Update position $\vec{x}_{k+1}^i$ according to equation 2.10
        End
        Stop if minimum error criteria has been attained
    End

Report result

pravdepodobnosť, že poloha častice i nadobudne na pozícii j hodnotu 1. Rovnica rýchlosti častice je rovnaká ako v základnej variante 2.9, pričom rýchlosť je obmedzená do intervalu $[0, 1]$ pomocou esovitej transformačnej funkcie:

$$S(v_{ij}) = \frac{1}{1 + e^{-v_{ij}}} \tag{2.11}$$

Častica potom aktualizuje svoju polohu prostredníctvom nasledujúcej rozhodovacej funkcie (symbol $\delta \in \langle 0, 1 \rangle$ má rovnaký význam ako δ_1 v predošlej definícii).

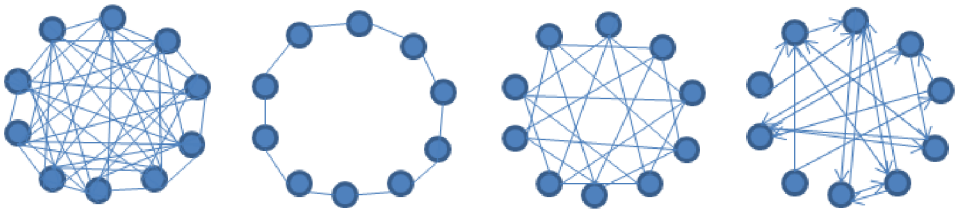
$$x_{ij} = \begin{cases} 1 & \delta \leq S(v_{ij}) \\ 0 & \text{inak} \end{cases} \tag{2.12}$$

2.5 Ďalšie varianty

Existuje niekoľko ďalších variant algoritmu PSO. Vznikli zobecňovaním základnej spojitej verzie, prípadne jeho kombináciou s inými metódami. V tejto sekcii sú stručne popísané len niektoré z nich. Ich podrobnejší popis, ako aj popis ďalších variánt, je možné nájsť v článkoch [38] venovaným tejto problematike.

Celočíselná

Diskrétna varianta PSO v ktorej sa optimálne riešenie zaokrúhľuje z reálnych na najbližšie celočíselné hodnoty. Reálne hodnoty riešenia sú získané pomocou základného algoritmu PSO.



(a) Plne prepojená (b) Prstencová (c) von Neumann (d) Náhodná

Obrázok 2.2: Najznámejšie topológie PSO [18].

Hybridná

V tejto variante je algoritmus PSO rozšírený o evolučné výpočtové techniky. Dôvodom tohto rozšírenia bolo zlepšenie rozmanitosti roja a to buď prevenciou prílišného priblíženia častíc k sebe, alebo adaptáciou parametrov algoritmu.

Medzi najznámejšie hybridné algoritmy patrí kombinácia genetických algoritmov a PSO (GA-PSO) [35], evolučné PSO (EPSO) [27] a kombinácia diferenciálnej evolúcie a PSO (DEPSO) [41].

Adaptívna

Snaha o vyladenie parametrov alogritmu PSO pomocou pridania náhodnej zložky k váhe zotrvačnosti. Aplikáciou Fuzzy logiky využitím pomocného PSO k hľadaniu optimálnych parametrov pre primárne PSO, Q-learning alebo adaptívna kritika.

Ďalšou z metód je rozdelenie roja na kmene na základe ich podobnosti. Každý kmeň je zoskupený okolo dominantnej častice. V každej iterácii je pre každý kmeň identifikovaná dominantná častica a jej riešenie je prehlásené za najlepšie riešenie v rámci kmeňa. Týmto je možné nájsť niekoľko lokálnych optím, z ktorých je po skončení všetkých iterácií vybrané globálne optimum.

2.6 Topológie časticového roja

V štandardnom (globálnom) algoritme PSO je každá častica každej inej častici susedom, tj. spoločenstvo má plne prepojenú topológiu. V tejto topológii je častica s globálne najlepším kandidátnym riešením (fitness) propagovaná do celého roja a tým ovplyvňuje smer rýchlosti všetkých častíc smerom k nájdenému optimu. Výhodou tejto topológie je rýchla konvergencia, ale môže zo sebou niesť riziko nenájdenia globálneho optima, ak súčasne nájdené nie je v blízkosti toho globálneho.

Z dôvodov lepšieho preskúmavania priestoru problému (za účelom vyhnutia sa lokálnym optimám) bolo vyvinuté lokálne PSO, v ktorom sú častice zoskupované do formácií podľa určitej stratégie. V tejto variante má každá formácia vlastného lídra (časticu s najlepším fitness v rámci formácie), ktorý ovplyvňuje len častice v rámci tej formácie, do ktorej patrí. Cenou za vyššiu šancu výhnúť sa lokálnym optimám je v lokálnom PSO pomalejšia konvergencia. Najznámejšie topológie sú zobrazené na obr. 2.2.

3. Vysoko výkonné počítanie

Vysoko výkonné počítanie (*High Performance Computing* – HPC) rieši problémy, ktorých výpočet je na bežných PC príliš pomalý, alebo sú tieto problémy príliš veľké.

Klasickým prístupom v HPC k zrýchleniu výpočtu je vytvoriť cluster počítačov s jedno alebo viacjadrovými procesormi (príkladom môže byť taktiež superpočítač), medzi ktorými je výpočet distribuovaný. V posledných rokoch sa k HPC čoraz viac pridáva technika GPGPU (*General-Purpose computation on GPU*) [28], ktorá využíva okrem CPU ešte aj GPU k obecným vedecko-technickým výpočtom. V dnešnej dobe je GPU cenovo dostupným „superpočítačom“, ktorý oproti bežným viacjadrovým CPU predstavuje masívny paralelný výpočtový výkon. Tento výkon môže dosahovať až 10 násobku výkonu viacjadrového CPU. Aj keď sa jedná len o teoretický dosiahnuteľný výkon, ktorý je v reálnych aplikáciách pracujúcich s dátami využitý na cca 10% [23], je tento prínos nesporný.

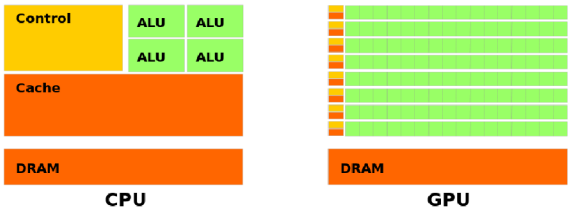
V programoch pre HPC je zásadný spomínaný paralelizmus ako aj maximálna optimalizácia na cieľovú platformu. V tejto kapitole rozoberiem a popíšem architektúru grafických kariet a viacjadrových procesorov, ako aj ich prístup k riešeniu paralelizmu.

3.1 CPU vs. GPU

Na obr. 3.1 je možné vidieť základný rozdiel v architektúrach CPU a GPU. Viacjadrové CPU pozostáva z niekoľkých jadier (štandardne 2 až 4) a je optimalizované na paralelizmus na úrovni úloh, tj. paralelné spracovanie rôznych funkcií v rámci jedného alebo rôznych programov. Na druhej strane GPU dosahuje obrovský výkon za použitia stoviek jadier a implementuje model SIMD (*Single Instruction Multiple Data*), v ktorom jedna funkcia paralelne spracúva množinu vstupných dát.

GPU dosahuje obrovského výkonu, ale množina aplikácií, ktorá je schopná tento výkon využiť je špecifická. Drastické zníženie výkonu prichádza s frekventovanou výmenou dát medzi hlavnou pamäťou a pamäťou na GPU. Vetvenie programu môže spôsobiť divergenciu vlákien, ktorá serializuje ich vykonávanie. Takisto je obmedzená synchronizácia prístupu k pamäti.

Aj keď GPU nie je možné využiť samostatne (CPU musí minimálne preniesť dáta na GPU a zahájiť výpočet) je výhodné kombinovať zároveň CPU a GPU tak, aby na GPU boli akcelerované len výpočtovo náročné, paralelizovateľné časti aplikácie a CPU vykonávalo sekvenčnú časť (ideálne počas výpočtu zahájenom na GPU).



Obrázok 3.1: Porovnanie architektúr CPU a GPU [29]. CPU disponuje väčšou cache pamäťou oproti GPU, to ale zas prevyšuje väčším počtom jadier.

Nástroje podporujúce GPGPU

GPGPU sa v posledných rokoch rapídne vyvinulo, takže dnes existuje niekoľko prístupov programovania GPU. Takisto nastupuje trend ich štandardizácie. V začiatkoch GPGPU tieto prístupy zahrňovali využitie špecializovaných jazykov na programovanie shaderov ako napr. *Cg*, *HLSL* alebo *GLSL*.

Tieto jazyky však neboli na účely GPGPU vytvorené a preto začali neskôr vznikať toolkity ako napr. *CUDA*, *ATI Stream* alebo *OpenCL*, ktoré sú určené priamo na tieto účely. V súčasnosti je CUDA [30] od spoločnosti NVIDIA najvyvinutejším takýmto nástrojom, ale umožňuje programovať len vybrané grafické karty a iba od tejto spoločnosti, nekladie si teda za cieľ hardwérovú multiplatformnosť.

Závislosť na výpočtovej platforme sa snaží natívne odstrániť až OpenCL [15], ktoré je abstraktnejšie ale zatiaľ nedosahuje také kvality ako CUDA. Medzi ďalšie nástroje, ktoré zaisťujú multiplatformnosť akcelerátoru (nie len GPU) patrí OpenACC [32], v ktorom sa pomocou direktív kompilátoru špecifikujú cykly a regióny kódu, ktoré budú vykonávané na dostupnom akcelerátore (podobný prístup ako v OpenMP [3]).

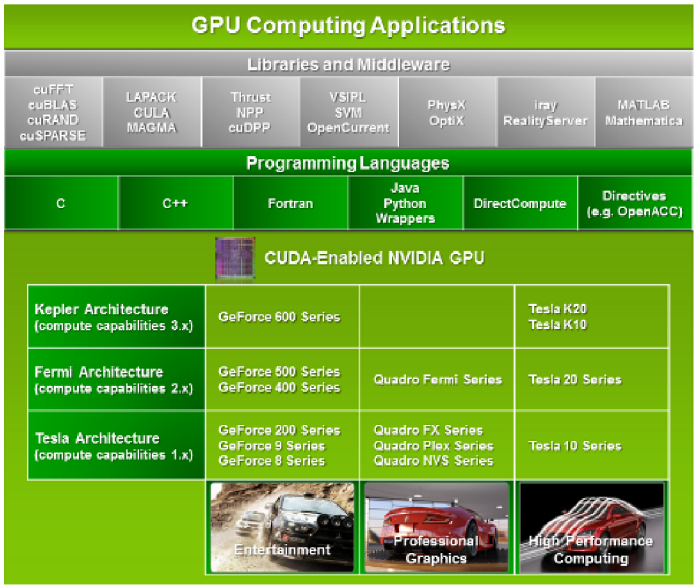
3.2 CUDA Toolkit

CUDA je paralelná výpočtová platforma a programovací model implementovaný vybranými grafickými kartami od spoločnosti NVIDIA. Poskytuje knižnice, direktívy kompilátoru a rozšírenia niektorých programovacích jazykov vrátane C/C++ (obr. 3.2).

Kernel

Základným konceptom v CUDA je tzv. kernel. Ide o funkciu, ktorá je po zavolaní spustená N-krát paralelne s využitím N rôznych CUDA vlákien. Nižšie je zobrazená ukážka kernelu, ktorý spočíta súčet dvoch vektorov veľkosti N [29]:

```
1 __global__ void VecAdd(float *A, float *B, float *C)
2 {
3         int i = threadIdx.x;
4         C[i] = A[i] + B[i];
5 }
6
```



Obrázok 3.2: CUDA Toolkit [29]. Ukážka zobrazuje architektúry súčasných GPU ako aj oblasť ich využitia, podporu programovacích jazykov a nástrojov, knižnice ktoré sú súčasťou Toolkitu.

```
7  int main()
8  {
9          ...
10         VecAdd<<<1, N>>>(A, B, C);
11         ...
12 }
```

Hierarchia vlákien

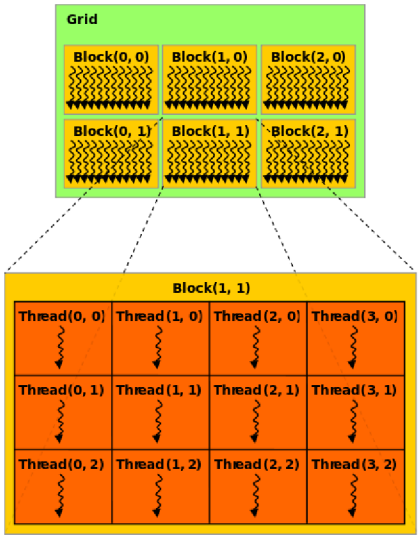
Vo vyššie uvedenom kóde je threadIdx 3-zložkový vektor (x, y, z), ktorý sa používa na identifikáciu vlákna v rámci bloku. Tým, že blok môže byť až 3-dimenzionálny, je poskytnutý prirodzený prostriedok k vyvolaniu výpočtu nad vektorom, maticou alebo kvádrom dát.

Počet vlákien v rámci bloku nie je neobmedzený. V súčasných grafických kartách môže blok vlákien obsahovať až 1024 vlákien. Kernel však môže byť spustený s viacerým počtom blokov, takže celkový počet vlákien je rovný násobku počtu blokov s počtom vlákien v bloku.

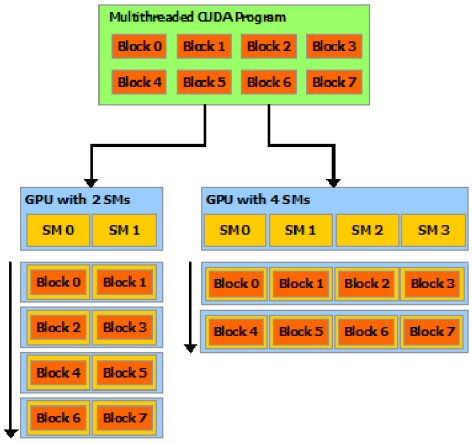
Bloky sú organizované v jedno, dvoj alebo troj-dimenzionálnom gride, tak ako je zobrazené na obr. 3.3. Bloky vlákien sú vykonávané nezávisle. Ich vykonávanie v ľubovolnom poradí, paralelne alebo sériovo, umožňuje škálovať kód s množstvom dostupných jadier (obr. 3.4).

Vlákna v rámci bloku môžu kooperovať. Môžu využiť zdieľanú pamäť, ktorá sa vyznačuje nízkou latenciou a je v blízkosti každého procesoru (podobná L1 cache). Takisto môžu synchronizovať svoj beh využitím funkcie *__syncthreads*(), ktorá sa správa ako bariéra[1].

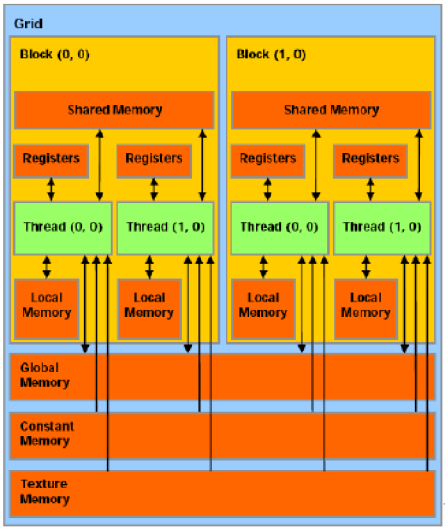
[1] bariéra – miesto na ktorom sa všetky vlákna pred ďalším pokračovaním počkajú.



Obrázok 3.3: Organizácia vlákien [29]. Vlákna sú usporiadané v bloku, ktorý je zas súčasťou gridu. Počet blokov v gride je zvyčajne udávaný veľkosťou spracovávaných dát. Počet vlákien v bloku je v súčasnosti obmedzený na 512 dimenzií pre architektúry Tesla a 1024 dimenzií pre architektúru Fermi a Kepler.



Obrázok 3.4: Plánovanie výpočtu [29]. Ukážka zobrazuje škálovateľnosť výpočtu s množstvom dostupných SM (*Streaming Multiprocessor*). Program je rozdelený do blokov vlákien, vykonávaných nezávisle. Použitím GPU s väčším počtom SM je možné skrátiť dobu výpočtu.



Obrázok 3.5: Hierarchia pamätí [4]. CUDA poskytuje niekoľko pamäťových priestorov. Prístup do pamätí v rámci bloku je rýchlejší než prístup do globálnej pamäte, avšak rýchlu zdieľanú pamäť môžu využiť len vlákna v rámci bloku do ktorého patria.

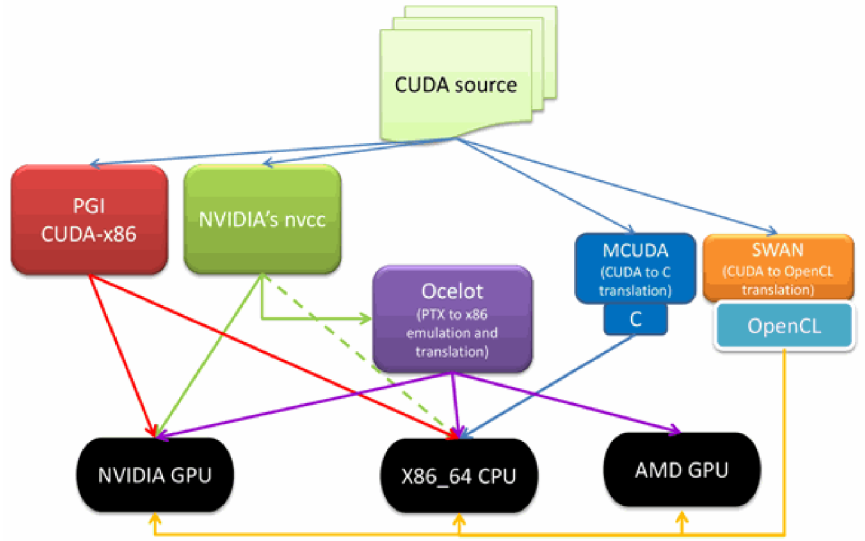
Synchronizácia vlákien medzi blokmi je možná len ukončením (a následným znovuspustením) kernelu, čo je relatívne náročná operácia.

Hierarchia pamätí

CUDA poskytuje vláknam niekoľko pamäťových priestorov. Na obr. 3.5 je možné vidieť, že každé vlákno disponuje lokálnou pamäťou a vlastnou sadou registrov. Zdieľaná pamäť je dostupná všetkým vláknam v rámci bloku a nakoniec posledné tri zobrazené pamäťe sú dostupné všetkým vláknam vo všetkých blokoch. Prvé tri pamäťové priestory, ktoré sú k vláknam najbližšie, majú síce menšiu kapacitu ako ostatné pamäte, ale z pohľadu prístupu k nim sú najrýchlejšie a preto by mali byť čo najviac využívané, pokiaľ je to možné. Globálna, textúrovacia pamäť a pamäť pre konštanty sú perzistentné v rámci celej aplikácie (medzi kernelmi) a sú optimalizované na rôzne použitie. Pamäť konštánt a textúrovacia pamäť sú však na rozdiel od ostatných iba na čítanie. Pri textúrovacej pamäti je možné takisto využiť rôzne adresovacie módy ako aj filtrovanie dát, avšak tieto funkcie sú pre techniku GPGPU relatívne nepodstatné.

Rozšírenia CUDA

Okrem základného rozšíreného jazyka C (CUDA C) je možné použiť CUDA platformu už aj v skriptovacích jazykoch (Python, Perl, Haskell, atď), čím je možné zrýchliť vývoj aplikácie na GPU. Tiež nástroje určené na vedecko-technické výpočty, ako napr. MATLAB alebo Mathematica, umožnujú využiť CUDA akcelerované funkcie (napr. výpočet fourierovej transformácie) veľmi jednoducho a bez absolútnej znalosti tejto platformy. Takisto existujú kompilátory (obr. 3.6), ktoré sprístupňujú CUDA programovací model rôznym



Obrázok 3.6: Riešenie multiplatformnosti [10]. Zobrazené nástroje umožňujú zdrojový kód CUDA programu prekladať na rôzne architektúry. Týmto je možné využiť paradigmu vysoko výkonného počítania zahrňujúcu CPU a GPU, popísanú v CUDA, bez obmedzenia sa na konkrétnu HW platformu.

platformám a vytvárajú tak z neho všestranný prostriedok využiteľný k tvorbe rôznych aplikácií nielen na GPU.

4. Problém MKP

Na ohodnotenie kvality a výkonu implementácie bol zvolený *Multidimensional Knapsack Problem* [26, 40] (MKP), tiež nazývaný *Multiple* alebo *Multiconstraint*, ktorý je zobecnením známeho NP-ťažkého *0-1 Knapsack* [26] (KP) problému.

4.1 Multidimensional Knapsack Problem

MKP môže byť považovaný tiež za problém alokácie zdrojov. Je daná množina n položiek (objektov) $J = \{1, \ldots, n\}$ a množina m zdrojov $I = \{1, \ldots, m\}$. Každý zdroj $i \in I$ má vyhradený rozpočet (batoh s limitovanou váhou) M_i. Každý objekt $j \in J$ má profit p_j a spotrebuje množstvo (má váhu) w_{ij} zdroja i. Problémom je maximalizácia zisku v limitovanom rozpočte. Definícia MKP je teda nasledovná [40]:

$$\text{Maximalizácia} \qquad f(\vec{x}) = \sum_{j=1}^{n} p_j x_j \tag{4.1}$$

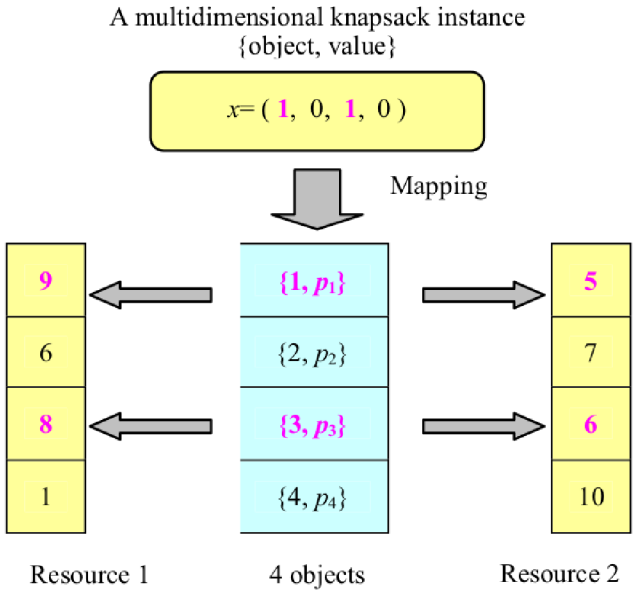
$$\text{S ohľadom na} \qquad f(\vec{x}) = \forall_{i \in I} \sum_{j=1}^{n} w_{ij} x_j \leq M_i, \;\; x_j \in \{0, 1\} \tag{4.2}$$

$$\text{Pri dodržaní} \qquad p_j > 0,\, w_{ij} \geq 0,\, M_i \geq 0 \tag{4.3}$$

Podmienka 4.2 je tiež nazývaná *Knapsack podmienka*, takže MKP problém je tiež nazývaný m-dimenzionálny Knapsack problém [40]. Ak je $m = 1$, problém sa stáva klasickým 1-dimenzionálnym KP problémom.

Vo vyššie uvedených rovniciach figuruje vektor $\vec{x} = (x_1, x_2, \ldots, x_j, \ldots, x_n)$, ktorého každá zložka nadobúda hodnotu 0 alebo 1. V prípade $x_j = 1$ je zabraté množstvo w_{ij} zdroja i. V opačnom prípade ak $x_j = 0$ zdroj nebude alokovaný. Na obr. 4.1 možno vidieť proces alokácie zdrojov v jednoduchej inštancii MKP problému so 4 položkami a 2 zdrojmi. Binárny vektor nesie informáciu o výbere položiek batohu. Tieto položky (1 až 4) nesú profit p_1 až p_4 a každá z nich navyšuje celkovú cenu takéhoto riešenia. Ako možné riešenie (nie nutne optimálne) s profitom $p_1 + p_3$, by bola ukážka akceptovaná jedine v prípade, že prvý zdroj má vyhradený rozpočet minimálne 17 jednotiek a druhý minimálne 11.

V procese automatizovaného hľadania optimálneho riešenia problému (napr. pomocou evolučných techník), môže dôjsť pri alokácii zdrojov k prekročeniu rozpočtu. Existuje niekoľko metód, ktoré ohodnocujú kvalitu (fitness) takto získaného riešenia, pričom zohľadňujú stav prekročenia rozpočtu. Medzi ne patria metódy ako *penalizačná funkcia* [31], *greed algoritmus* [7] alebo *surrogate duality metóda* [34].



Obrázok 4.1: Príklad alokácie zdrojov [40]. Kandidátne riešenie x uvažovanej inštancie MKP problému vyberá objekty ktoré budú do batohu vložené, čím odčerpajú hodnoty z vyhradených rozpočtov, ale vytvoria zisk $p_1 + p_3$.

4.2 Penalizačná funkcia

Úlohou tejto funkcie je ohodnocovať kvalitu (fitness) navrhnutého riešenia MKP problému. Táto funkcia zohľadňuje a penalizuje prípadné prekročenie kapacity batohu (rozpočtu). Existuje niekoľko penalizačných funkcií [31], pričom uvediem jednu. Penalizačná funkcia má nasledujúci tvar:

$$f(\vec{x}) = \sum_{j=1}^{n} p_j x_j - \sum_{i=1}^{m} P \; poslin \left(\sum_{j=1}^{n} w_{ij} x_j - M_i \right) \tag{4.4}$$

Funkcia mapuje binárny vektor predstavujúci potencionálne riešenie problému na jeho kvalitu. Funkcia sčíta profity všetkých vybraných položiek a od ich súčtu odčíta penalizáciu. Táto penalizácia rastie s počtom batohov u ktorých došlo k prekročeniu vyhradeného rozpočtu. Úlohou funkcie *poslin* je určit mieru prekročenia rozpočtu. Je určená nasledujúcou rovnicou:

$$poslin(x) = \begin{cases} 0 & x < 0 \\ x & \text{inak} \end{cases} \tag{4.5}$$

Miera s akou došlo k prekročeniu rozpočtu, funkcia *poslin* získava ako rozdiel ceny potencionálneho riešenia od vyhradeného rozpočtu. Riešenie v ktorom dôjde k prekročeniu rozpočtu hoci len v 1 batohu a to len o 1 jednotku, nemôže byť akceptované. Z tohto dôvodu v rovnici figuruje ešte aj parameter P, ktorý posúva mieru prekročenia do vyšších rádov. Táto konštanta by mala byť vhodne zvolená, nemala by byť príliš malá, ale ani moc veľká, pretože môže výrazne ovplyvniť rýchlosť konvergencie a kvalitu nájdeného riešenia.

Je dobré ju voliť v závislosti na konkrétnej veľkosti problému. Jedna z možných techník je uvedená v článku [37].

4.3 Využitie

Veľké množstvo reálnych problémov je možné redukovať na problém MKP. Napr. prerozdeľovanie investičného kapitálu [24] alebo voľba projektu [33]:

> „Allocating funds to independent R&D projects is a problem of practical importance for many firms. We formulate the problem as a 0-1 integer programming problem with the objective of selecting projects that will maximize the anticipated dollar contract volume, yet not exceed cost budgets.“

5. Implementácia na CPU

V tejto kapitole rozoberiem a popíšem spôsob implementácie optimalizačnej techniky pomocou časticových rojov PSO prezentovanej v kapitole 2. Implementácia je v jazyku C++ s využitím nástroja OpenMP pomocou ktorého je vyriešená paralelizácia. V implementácii nie sú využité žiadne platformne závislé rozhrania ani knižnice, tým je zaistená multiplatformnosť výsledného riešenia. Pri vytváraní programu boli ďalej využité bezplatné a voľne dostupné nástroje, použité na kompiláciu, meranie výkonu a profiláciu (podkapitola 5.6).

Napriek tomu, že cieľom práce je vytvoriť paralelnú aplikáciu akcelerovanú na GPU, je vhodné najprv vytvoriť túto aplikáciu na CPU. Tento prístup je zvolený z pragmatických dôvodov. Návrh aplikácie prezentovaný v tejto kapitole bude prevzatý a použitý pri tvorbe jej akcelerovanej podoby, v ktorej sa už nebude nutné zaoberať algoritmom PSO, ale tvorbou samotného akcelerátora. Taktiež vďaka tomu, že vytvorená aplikácia na CPU bude plne optimalizovaná, bude možné objektívne zhodnotiť prínos jej akceleráciou.

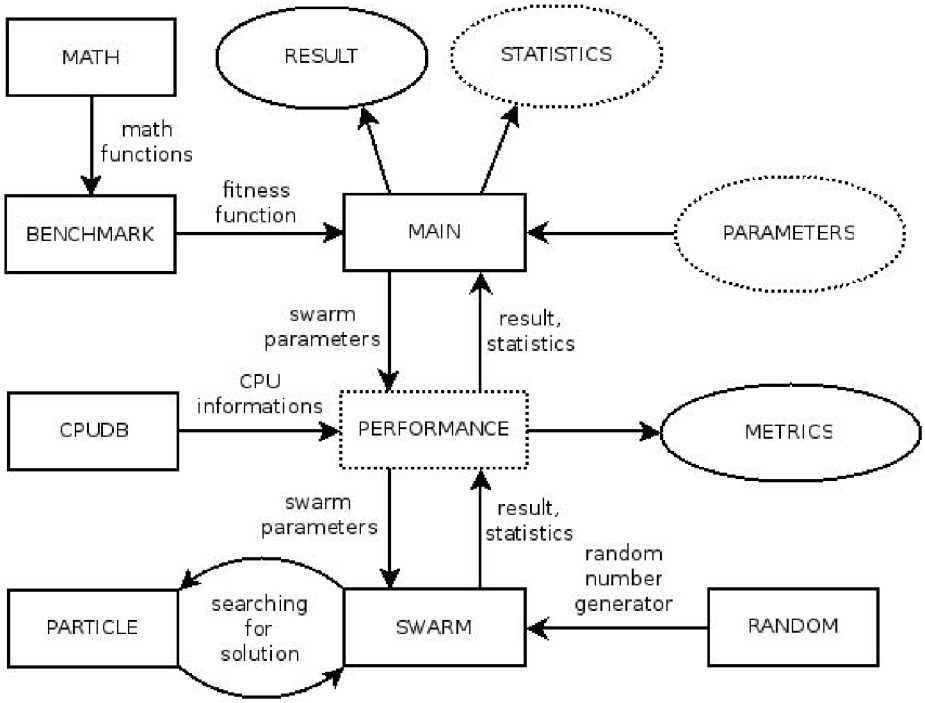
V priebehu času došlo k zmene návrhu architektúry programu. Program bol v prvopočiatkoch určený len na riešenie funkcií, ktoré nepracujú s dátami. Príkladom takýchto funkcií sú klasické funkcie na ohodnocovanie výkonu implementácie ako napr. *Weierstrass*, *Sumcan*, *Ackley* a mnohé ďalšie. V nasledujúcej podkapitole popíšem návrh pôvodného programu, pretože je základom pre program určený na riešenie zvolenej úlohy popísanej v kapitole 4.

5.1 Návrh a dekompozícia pôvodného programu

Architektúra programu je objektová, pričom je dekomponovaná do niekoľkých modulov a to konkrétne **particle**, **swarm**, **benchmark**, **main**, **math**, **random**, **performance** a **cpudb**.

Prvé štyri moduly sú z pohľadu účelu programu hlavné. Ďalšie tri sú len pomocné a ich úlohou je zabezpečiť jednotné rozhranie pri použití viacerých alternatív. Posledný modul je databázou informácií o vybraných procesoroch od výrobcu. Tieto informácie sú použité pri meraní dosiahnutého výkonu implementácie.

Architektúra programu, súčasne s popisom komunikácie medzi jej modulmi, je zobrazená na obr. 5.1. Moduly a pseudomoduly (vstupy a výstupy) zobrazené prerušovane nemusia byť využité.



Obrázok 5.1: Architektúra pôvodného programu.

Modul particle

V tomto module je definovaná častica roja ako entita, ktorá v sebe nesie údaje o svojej aktuálnej **polohe**, **rýchlosti** a svojho zatiaľ najlepšie dosiahnutého **riešenia** (poloha s najlepším ohodnotením).

Táto entita je modelovaná ako trieda. Aby bolo zaistené zapúzdrenie, je prístup ku každému jej atribútu pomocou metódy.

Modul swarm

Tento modul obsahuje triedu, ktorá predstavuje roj častíc (z modulu *particle*). Jej atribútmi sú **množina častíc** ($\mathbb{M}$), **fitness funkcia** (f) určená k ohodnocovaniu kvality kandidátneho riešenia každej častice, **maximálna rýchlosť častíc** ($\vec{V}$) v absolútnej hodnote pre každú dimenziu, **globálne najlepšie riešenie** (**g**) ako referencia na časticu, ktorá toto riešenie dosiahla.

Kvalitu výsledného riešenia, tj. presnosť a/alebo rýchlosť konvergencie, je možné ďalej ovplyvňovať pomocou parametrov roja. Týmito parametrami sú **váha zotrvačnosti** častice (W), **poznávací** (C) a **sociálny** (S) parameter.

Hlavné metódy tejto triedy sú **init()**, **update()** a **solve()**, ktoré riešia inicializáciu, aktualizáciu roja a spustenie samotného algoritmu (hľadania riešenia). Tento modul predstavuje hlavnú časť celej aplikácie.

Parametre metódy init() vymedzujú podpriestor z celkového priestoru riešenej funkcie, v ktorom sa predpokladá, že sa nachádza riešenie. Častice sú následné inicializované tak, aby sa nachádzali v tomto podpriestore. Častice však v procese riešenia môžu tento podpriestor opustiť za účelom hľadania lepšieho riešenia (ak sa toto nachádza mimo zadaný podpriestor).

Metóda solve() má 2 parametry. Prvým je maximálny počet iterácií na nájdenie riešenia (e). Druhým je kritérium zastavenia riešenia (s), ktoré sa uplatní ak je splnená podmienka $s > 0$. Riešenie je zastavené, ak nedôjde v s iteráciach k zmene globálne najlepšieho riešenia (roj pravdepodobne skonvergoval).

V algoritmoch operátor $\cdot$ označuje prístup k atribútom častíc – poloha ($\vec{p}$), rýchlosť ($\vec{v}$), kandidátne riešenie ($\vec{b}$) a vhodnosť tohto riešenia (ohodnotené fitness funkciou) (b). V uvedených algoritmoch sa predpokladá, že vo všetkých operáciách nad vektormi, sú tieto vektory rovnakej dimenzie. Vektor nad číslicou znamená vektor týchto číslic potrebnej dimenzie.

Funkcia $\delta(\vec{u}, \vec{v})$, $\vec{u} < \vec{v}$ je funkciou s vnútorným stavom (z matematického pohľadu sa nejedná o funkciu), ktorá každým volaním vracia vektor iných náhodných čísiel $\vec{x}$ s uniformným rozložením tak, že platí $\vec{u} \leq \vec{x} < \vec{v}$, ďalej funkcia δ_n, pre ľubovolné n je ekvivalentná funkcii $\delta\left(\vec{0}, \vec{1}\right)$.

$$\begin{aligned} \vec{u} &= (u_0,\ u_1,\ \ldots,\ u_n) \\ \vec{v} &= (v_0,\ v_1,\ \ldots,\ v_n) \end{aligned}$$

$$\vec{u} < \vec{v} \Leftrightarrow \forall i \in \{0, 1, \ldots, n\}, u_i < v_i \tag{5.1}$$
$$\vec{u} = \vec{v} \Leftrightarrow \forall i \in \{0, 1, \ldots, n\}, u_i = v_i \tag{5.2}$$
$$\vec{u} \leq \vec{v} \Leftrightarrow \vec{u} < \vec{v} \vee \vec{u} = \vec{v} \tag{5.3}$$

Algoritmus 2 Inicializácia časticového roja

Require: Existing swarm $\mathbb{M}$
Ensure: Initialized swarm, velocity clamping vector $\vec{V}$ set

```
1: procedure INIT(m⃗, n⃗)                                   ▷ m⃗ < n⃗
2:     V⃗ ← m⃗ − n⃗
3:     parallel
4:         for ∀p: p ∈ M
5:             p·p⃗ ← δ(m⃗, n⃗)
6:             p·v⃗ ← δ(0⃗, V⃗)
7:         end for
8:     end parallel
9: end procedure
```

Modul benchmark

V tomto module sú implementované **fitness** funkcie zo sady testov (podkapitola 5.4). Každá fitness funkcia mapuje kandidátne riešenie (polohu častice) na vhodnosť (kvalitu) takéhoto riešenia a to tak, že čím je riešenie lepšie, tým je táto hodnota väčšia.

Každá fitness funkcia v tomto module „preklápa“ svoj výsledok tak, aby bola dodržaná požiadavka na maximálnu hodnotu vhodnosti (fitness) riešenia a to na základe toho, či ide o hľadanie globálneho maxima, minima alebo riešenia rovnice. Okrem preklopenia výsledku môže takisto dochádzať k jeho normalizácii, penalizácii alebo rankingu.

Ak ide o hľadanie maxima, hodnota sa nepreklápa preto, že najlepšie riešenie je automaticky to najväčšie. Ak však ide o hľadanie minima, výsledok je preklopený zo záporných

Algoritmus 3 Aktualizácia časticového roja

Require: Initialized or previously updated swarm $\mathbb{M}$, vector $\vec{V}$, parameters W, C, S
Ensure: Updated swarm

procedure UPDATE
  parallel
    for $\forall$**p**: **p**, **g** $\in \mathbb{M}$ ▷ **g** is particle with best solution
      ▷ operator $\diamond$ denote elementwise multiplication
      $\vec{v}_a \leftarrow W\ \mathbf{p}{\cdot}\vec{v} + C\ \delta_1 \diamond \left(\mathbf{p}{\cdot}\vec{b} - \mathbf{p}{\cdot}\vec{p}\right) + S\ \delta_2 \diamond \left(\mathbf{g}{\cdot}\vec{b} - \mathbf{p}{\cdot}\vec{p}\right)$
      $\vec{v}_b \leftarrow$ clamped $\vec{v}_a$ according to $\vec{V}$ ▷ ensure $-\vec{V} \leq \vec{v}_b \leq \vec{V}$
      $\mathbf{p}{\cdot}\vec{p} \leftarrow \mathbf{p}{\cdot}\vec{p} + \vec{v}_b$
      $\mathbf{p}{\cdot}\vec{v} \leftarrow \vec{v}_b$
    end for
  end parallel
end procedure

Algoritmus 4 Hľadanie optima

Require: Initialized swarm $\mathbb{M}$, fitness function f
Ensure: Possible optimal result

function SOLVE(e, s) ▷ $e > 0 \wedge s \geq 0$
  if $s = 0$ **then**
    $s \leftarrow e$
  end if
  $m \leftarrow 0$
  while $e > 0 \vee m < s$ **do**
    parallel
      for $\forall$**p**: **p**, **g** $\in \mathbb{M}$ ▷ **g** is particle with best solution
        $v \leftarrow f\ (\mathbf{p}{\cdot}\vec{p})$
        if $v > \mathbf{p}{\cdot}b$ **then**
          $\mathbf{p}{\cdot}b \leftarrow v$
        end if
        critical
          if $\mathbf{p}{\cdot}b > \mathbf{g}{\cdot}b$ **then**
            g $\leftarrow$ **p**
            $m \leftarrow 0$
          end if
        end critical
      end for
    end parallel
    update swarm according to algorithm 3
    $e \leftarrow e - 1$
    $m \leftarrow m + 1$
  end while
  return $\mathbf{g}{\cdot}\vec{b}$
end function

do kladných hodnôt. Pri hľadaní riešenia rovnice je najlepší výsledok ten, ktorý je najbližší k 0, výsledok je teda preklopený len pre kladné hodnoty.

Modul main

Tento modul vykonáva parametrizáciu, inicializáciu a spustenie samotného riešenia. Parametre pre PSO dodáva užívateľ, alebo sú použité východzie. Po ukončení výpočtu tento modul zaisťuje prezentáciu výsledku.

Komunikácia s modulom *swarm* prebieha nepriamo cez modul *performance*, ktorý spustí výpočet a prípadne zmeria metriku, ktorá bola požadovaná. Táto metrika sa vzťahuje len na výpočet samotného riešenia (metóda solve()), pretože sa jedná o najdôležitejšie miesto úlohy. Nemeria sa teda čas strávený inicializáciou ani prípravnými krokmi. Na meranie metrík je využitá knižnica PAPI (podkapitola 5.6).

5.2 Implementácia

Implementácia prebiehala v etapách a vyvíjala sa na základe poznatkov uvedených v podkapitole 5.3. V prvej etape som vytvoril sekvenčnú implementáciu podľa návrhu z podkapitoly 5.1 a jej správnosť som overil na sade testov (podkapitola 5.4). Po overení správnosti implementácie som túto prehlásil za základnú, ktorá bude slúžiť ako referenčná vzhľadom k ďalším optimalizovaným verziám.

V základnej verzii a v nasledujúcich optimalizovaných verziách som na reprezentáciu reálnych čísiel zvolil dátový typ **float**. Jeho presnosť je síce oproti typu **double** menšia, ale pri riešení úloh pomocou algoritmu PSO postačuje. Výhoda použitia dátového typu float oproti typu double podľa [1] spočíva v rýchlejšom výpočte niektorých operácií nad týmto typom. Takisto pri použití vektorových registrov XMM a YMM (novšia a dnes kompilátormi preferovanejšia metóda pri generovaní kódu, ak ju procesor podporuje), je umožnený paralelný výpočet nad vektorom dát jednou inštrukciou. Výhoda je takisto v menšej veľkosti tohto typu.

Paralelizácia

V nasledujúcej verzii (Verzia PAR) som paralelizoval sekcie kódu vyznačené (*paralell*) v algoritmoch z predchádzajúcej podkapitoly na inicializáciu, aktualizáciu a hľadanie riešenia. Vo všetkých oblastiach sa jedná o paralelizáciu cyklu, ktorý iteruje všetkými časticami roja. Keďže častice nemajú medzi sebou závislosti, paralelizácia týchto oblasti je triviálna (*embarrassingly parallel problem*). K samotnej paralelizácii sekcií som využil pragmu z OpenMP. Východzí (možno zmeniť pomocou parametru) počet vlákien som nastavil rovnaký ako počet dostupných procesorov (jadier) v systéme.

```
1 #pragma omp parallel
2 {
3    ...
4 #pragma omp for schedule (static)
```

```
5    for(size_t p = 0; p < swarm.size(); ++p)
6    { // parallel section
7       ...
8    }
9    ...
10 }
```

Pragma na riadku 1 vytvorí vlákna, ktoré budú vykonávať paralelnú oblasť pod ňou. Na riadku 4 sa nachádza pragma, ktorá zase rozdelí iterácie (častice) medzi vytvorené vlákna.

Vzhľadom k tomu, že spracovanie každej častice vyžaduje rovnaké množstvo práce, je zvolený statický plán, ktorý má najmenšiu réžiu. Porcia častíc (počet susediacich iterácií – *chunk*) pridelená každému vláknu je v tomto pláne rovnaká alebo takmer rovnaká [6].

V algoritme 4 určeného na aktualizáciu častíc sa nachádza kritická (*critical*) sekcia, ktorá sa nemôže vykonávať paralelne. Avšak je malá pravdepodobnosť, že sa tu stretne viac vlákien, pretože vstup do tejto oblasti je podmienený nájdením lepšieho riešenia ako je súčasné najlepšie.

```
1  if(swarm[p].best < swarm[g].best)
2  { // parallel section
3  #pragma omp critical
4     if(swarm[p].best < swarm[g].best)
5     { // critical section
6        ...
7     }
8  }
```

V OpenMP sa táto oblasť ošetrí dvakrát rovnakou podmienkou. Prvýkrát všetkými vláknami v paralelnej oblasti. Tie, ktoré splnia podmienku (ich častica má lepšie riešenie ako zatiaľ najlepšie nájdené), vstúpia do oblasti pred kritickou sekciou. Keďže sa v tejto sekcii môže nachádzať viac vlákien, je možné, že budú čakať. Preto sa platnosť podmienky overuje druhýkrát po vstupe do kritickej sekcie.

Vnorenie metód a náhrada generátora pseudonáhodných čísiel

Z profilácie paralelnej implementácie na obr. 5.2 vidno zbytočne strávený čas volaním metód z modulu *particle*. Tieto metódy sú veľmi krátke a je vhodné ich obsah začleniť priamo do kódu, namiesto volania metódy. Toto bolo dosiahnuté vnorením definície metód do hlavičkového súboru a predradením kľúčového slova **inline** pred ich definíciu.

```
1  class Particle
2  {
3     float best;
4     ...
5  public:
6     inline float getBest() const { return best; }
7     ...
8  };
```

Z profilácie ďalej vidno, že sa nejaký čas riešenia strávi v generátore pseudonáhodných čísiel. V základnej verzii bol použitý generátor Mersenne-Twister z knižnice *Boost*[1]. Tento generátor má veľmi dobré štatistické vlastnosti, avšak prístup k nemu je serializovaný a teda nie vhodný pri paralelnej implementácii.

Jedným riešením by bolo použiť reentrantnú variantu lineárneho kongruentného generátora *rand_r()* zo štandardnej knižnice jazyka C. Následkom toho by ale došlo k zhoršeniu štatistických vlastností takto generovaných pseudonáhodných čísiel.

Ďalším a vhodnejším riešením je použiť voľne dostupný generátor z knižnice *Random123*[2], ktorý je podľa autorov, kvôli svojmu výkonu, nízkej pamäťovej náročnosti, veľmi dobrej paralelizácii a vektorizácii, výhodný na použitie v paralelných aplikáciách. Takisto generuje pseudonáhodné čísla s veľkou periódou a výrazne kvalitnejšími vlastnosťami oproti generátoru zo štandardnej knižnice jazyka C. Túto variantu som nakoniec zvolil.

Rozbalenie cyklov

Z profilácie predošlej verzie (Verzia VME) na obr. 5.2 vidno, naďalej pomerne veľké množstvo času stráveného réžiou algoritmu PSO a to konkrétne algoritmu 3 na aktualizáciu častíc.

Táto réžia by mala byť čo najmenšia, preto som sa v tejto verzii zameral na rozbalenie cyklov v tomto algoritme. Rozbalenie cyklov má svoje výhody aj nevýhody. Hlavnou výhodou je zníženie počtu iterácií a tým aj zníženie počtu testovaní podmienky konca cyklu. Hlavnou nevýhodou však je, že rozbalený cyklus zaberá viac priestoru v inštrukčnej cache [1].

Rozbalenie cyklov dokáže v rámci optimalizácií robiť kompilátor, ale len pre cykly s dopredu známym počtom iterácií, nie obecne. Niektoré kompilátory (napr. od firiem IBM® alebo INTEL®) podporujú konštrukciu, ktorá umožňuje vynútiť automatické rozbalenie cyklu.

```
#pragma unroll(F)
for(size_t n = 0; n < N; ++n)
{
   ...
}
```

Nepovinný parameter F údáva faktor s akým počtom iterácií sa má cyklus rozbaliť. Tento funguje aj s dopredu neznámym počtom iterácií N. Avšak v tomto prípade dochádza k vygenerovaniu kódu, ktorého účelom je zabezpečiť vykonanie všetkých iterácií napriek tomu, že je zaistená podmienka $0 \equiv N \mod F$.

V algoritme 3 som použil ručné rozbalenie cyklu, pretože sa jedná o cyklus pomerne veľkého bloku. Na dielčie bloky som potom použil cykly so známym počtom iterácií, ktoré kompilátor dokáže automaticky rozbaliť. Použil som aj pragmu spomínanú vyššie, ktorá toto dokáže vynútiť. Kompilátormi, ktoré ju nepodporujú, je ignorovaná.

[1] Boost je sada knižníc jazyka C++ použiteľných v širokom spektre aplikácií. Je súčasťou nového štandardu jazyka C++ (C++11). Dostupná na `http://www.boost.org`

[2] Knižnica generátorov pseudonáhodných čísiel založená na princípe (*counter-based*), v ktorom sa N-té pseudonáhodné číslo získáva bezstavovou transformačnou funkciou aplikovanou na číslo N, na rozdiel od konvenčného prístupu využívajúceho N-tú iteráciu stavovej transformačnej funkcie. Dostupná na `http://www.deshawresearch.com/resources_random123.html`

```
for(size_t p = 0; p < swarm.size(); p += 4)
{
  ...
#pragma unroll
  for(size_t n = 0; n < 4; ++n)  // loop unrolled by compiler
    swarm[p + n] ...
  ...
}
```

Pri rozbalení cyklov po štvoriciach je nutné zabezpečiť, aby bol celkový počet častíc deliteľný 4. Odchýlka maximálne 3 častíc od počtu častíc zadaného užívateľom je zanedbateľná. Nasledujúca konštrukcia zabezpečí, že počet častíc v roji bude zaokrúhlený smerom nahor, len ak je to nutné (*padding*).

```
size_t Swarm::adjustParticleCount(size_t n)
{
  return (n % 4) ? n + (4 - (n % 4)) : n;
}
```

Optimalizácia fitness funkcie

Z profilácie poslednej verzie (ale aj zo všetkých predošlých) na obr. 5.2 vidno, že posledným najdôležitejším miestom a miestom s najväčším priestorom na optimalizáciu je práve **fitness** funkcia.

K jej optimalizácii som zvolil podobnú techniku ako v predošlej verzii, tj. rozbalenie cyklu po štvoriciach tentokrát v algoritme 4. A to kvôli možnosti upraviť **fitness** funkciu tak, aby ohodnocovala 4 častice naraz. Touto technikou je umožnené kompilátoru v procese optimalizácie vektorizovať niektoré cykly a bloky kódu v tejto funkcii. Nižšie je ukážka fitness funkcie *Weierstrass* ohodnocujúca kvalitu riešenia pri hľadaní globálneho maxima.

```
void Weierstrass::fit(const Particle::arg_t * x[4], float result[4])
{
  const size_t K = 20;

  float v[4] = { .0f };
  float t = 0.f;

  for(size_t k = 0; k <= K; ++k)
  {
    float a = pow(0.5f, k);
    float b = 2.f * M_DOUBLE2FLOAT(M_PI) * pow(3.f, k);

    for(size_t i = 0; i < dims; ++i)
    {
      v[0] += a * fastCos(b * (( *x[0] )[i] + 0.5f));
      v[1] += a * fastCos(b * (( *x[1] )[i] + 0.5f));
      v[2] += a * fastCos(b * (( *x[2] )[i] + 0.5f));
      v[3] += a * fastCos(b * (( *x[3] )[i] + 0.5f));
```

```
19      }
20
21      t += a * fastCos(b * 0.5f);
22    }
23
24    for(size_t n = 0; n < 4; ++n)  // vectorized loop
25      result[n] = FIT_MAX(v[n] - dims * t);
26  }
```

Kvôli ďalšiemu zrýchleniu výpočtu som využil knižnicu matematických funkcií *VICephes*[3], implementovanú s využitím konštrukcií, ktoré sú kompilátorom ľahko rozpoznateľné pri vektorizácii blokov kódu. V predchádzajúcej ukážke je možné vidieť funkciu *fastCos*, z modulu **math**, ktorá môže zastupovať práve funkciu na výpočet kosínusu z knižnice *VICephes*. Pri použití kompilátoru od firmy INTEL® sa na miesto spomínanej matematickej knižnice používa natívna *INTEL® Math Kernel Library*, ktorá je maximálne vyladená pre tento kompilátor.

Úkažka volania fitness funkcie je nasledovná:

```
1  for(size_t p = 0; p < swarm.size(); p += 4)
2  {
3    float fval[4];
4    fit(swarm[p + 0].position(), swarm[p + 1].position(),
5        swarm[p + 2].position(), swarm[p + 3].position(),
6        fval);
7    ...
8  }
```

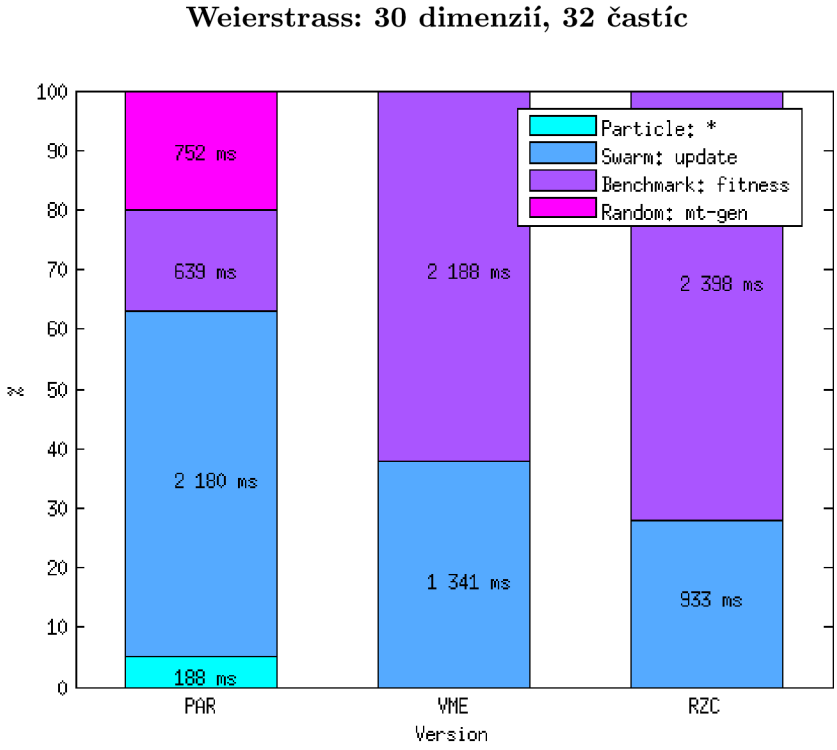
5.3 Výkonnostné metriky a profilácia implementácií

Implementácie verzií, okrem referenčnej a poslednej, boli podrobené profilácii kvôli zisteniu času stráveného v každej funkcii z celkového času behu. Profiláciou prešiel kód vygenerovaný kompilátorom **GCC** za využitia **gprof** profileru (podkapitola 5.6). Spriemerovaný výsledok profilácie z niekoľkých behov je pre jednotlivé verzie zobrazený na obr. 5.2.

Tabuľka 5.1: Zmeny v profilovaných verziách.

Verzia	Zmeny oproti predchádzajúcej verzii
REF	Sekvenčná implementácia
PAR	Paralelizácia algoritmu PSO
VME	Vnorenie metód a náhrada generátora náhodných čísiel
RZC	Rozbalenie cyklov
OPT	Vektorizácia fitness funkcie a využitie optimalizovanej matematickej knižnice

[3]Dostupná na `https://twiki.cern.ch/twiki/bin/view/LCG/VICephes`



Obrázok 5.2: Profilácia implementácie. Na ose x sú zľava smerom vpravo zobrazené verzie s vyšším stupňom optimalizácie. Smerom k optimalizovanejšej verzií je možné vidieť postupné znižovanie réžie algoritmu PSO. Z výkonového hľadiska verzia VME priniesla zlepšenie oproti verzií PAR (obr. 5.3), verzia RZC však už tak výrazne výkon nenavyšila.

Zo všetkých profilácii je zrejmé, že najviac času je stráveného vo **fitness** funkcii. Táto funkcia je výpočetne najnáročnejším miestom celej aplikácie, preto je jej akcelerácia z pohľadu zrýchlenia celej aplikácie najdôležitejšia.

Na zistenie maximálneho možného zrýchlenia akého ide teoreticky dosiahnuť v tejto funkcii (nekonečné zrýchlenie, tj. úplne odstránenie tohto miesta), je využitý Amdahlov zákon 5.4.

$$\xi(P) = \lim_{S\to\infty} \frac{1}{(1-P) + \frac{P}{S}} = \frac{1}{(1-P)} \tag{5.4}$$

Dosadením počtu percent času stráveného vo fitness funkcii, získaného z profilácie na obr. 5.2 do vyššie uvedeného vzťahu, je možné získať maximálne možné zrýchlenie pre Verziu 3, ktorá bola profilovaná.

$$\xi(0.81) \doteq \mathbf{5.26}$$

Akceleráciou fitness funkcie je teda teoreticky možné dosiahnuť **5**-násobného zrýchlenia výpočtu.

Porovnanie výkonu všetkých verzií implementácii je zachytené v tab. 5.2. Porovnanie výkonu verzií je tiež zobrazené na obr. 5.3, v ktorom je zachytená závislosť ohodnotenia počtu častíc za sekundu na veľkosti problému (v dimenziách). Priemerná odchýlka od správneho riešenia je zachytená v tab. 5.3.

Tabuľka 5.2: Porovnanie verzií implementácií.

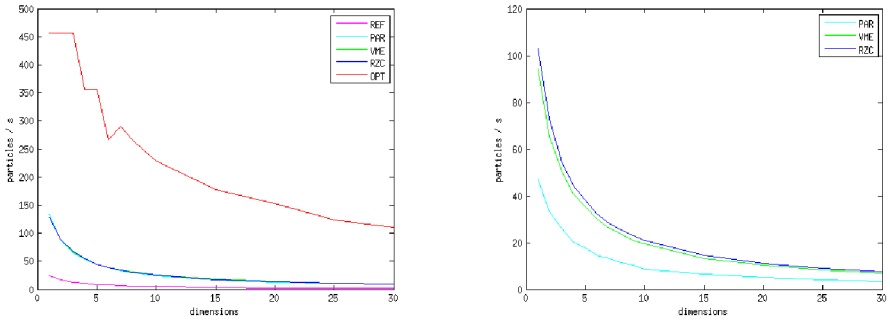
P je počet častíc, I je počet iterácií. Merané na INTEL® Core™ 2 DUO E8400 s teoretickým výkonom 24 GFLOPS. Tabuľka zachycuje priemernú hodnotu z 10 behov hľadania globálneho maxima v 5 dimenzionálnej variante funkcie Weierstrass.

		Verzia				
		REF	**PAR**	**VME**	**RZC**	**OPT**
P	**I**	**čas (s)**				
32	3000	0.88	0.53	0.43	0.43	0.27
64	3000	1.72	1.05	0.86	0.85	0.53
128	3000	3.47	2.11	1.72	1.72	1.08
32	6000	1.72	1.06	0.85	0.86	0.53
64	6000	3.54	2.11	1.71	1.72	1.06
128	6000	6.9	4.22	3.42	3.42	2.12
GFLOPS		0.84	1.40	1.71	1.72	1.86

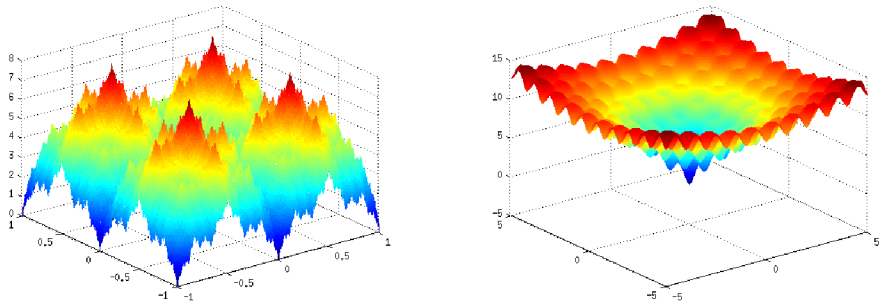
Tabuľka 5.3: Priemerná odchýlka od správneho riešenia.

Priemerná odchýlka v % od správneho riešenia 10 dimenzionálnej variante funkcie Weierstrass. P je počet častíc, I je počet iterácií.

P \ **I**	2000	3000	4000	5000	6000	7000	8000	9000	10000
32	4.5769	5.3016	1.0624	1.4683	0.5622	0.9490	0.5792	0.0487	0.0467
64	0.7260	1.9083	0.5712	0.3313	0.0179	0.0139	0.0128	0.0281	0.0182
96	0.1996	0.0070	0.0065	0.0608	0.0054	0.0069	0.0073	0.0076	0.0045
128	0.4989	0.0175	0.0162	0.1519	0.0135	0.0172	0.0182	0.0190	0.0112



Obrázok 5.3: Porovnanie výkonu verzií. Zachytenie závislosti počtu ohodnotených častíc za sekundu na počte dimenzií problému. Vľavo možno vidieť porovnanie všetkých verzií implementácií. Vpravo sú zachytené tie z nich u ktorých prínos z ľavého obrázka nie je zreteľný.



(a) 2-dim Weierstrass

(b) 2-dim Ackley

Obrázok 5.4: Multidimenzionálne funkcie Weierstrass a Ackley. Tieto funkcie sú klasickými predstaviteľmi výkonnostných testov heuristických metód. Na obr. je farebne zachytený prechod od jedného globálneho extrému k druhému. U funkcie Weierstrass je možné vidieť viac globálnych extrémov.

5.4 Sada testov

Správnosť implementácie (nájdenie správneho riešenia) a porovnanie výkonu optimalizovaných implementácií so základnou som testoval vyriešením jednoduchej rovnice 5.5 [12] a hľadaním globálnych extrémov v 2-dimenzionálnych variantách funkcií Weierstrass a Ackley (obr. 5.4). Globálne extrémy týchto funkcií sú zrejmé priamo z ich priebehu). Vo funkcii Weierstrass som hľadal maximum, v Ackley minimum.

Posledné dve funkcie (okrem dalších), vzhľadom k tomu, že sa jedná o multidimenzionálne funkcie a tým pádom je problém riešenia možné rozšíriť na ľubovolný počet dimenzií, sú vhodnými kandidátmi na testovanie výkonnosti implementácie algoritmu PSO.

$$f(x,y) = (2.8125 - x + xy^4)^2 + (2.25 - x + xy^2)^2 + (1.5 - x + xy)^2 \tag{5.5}$$

5.5 Adaptovanie programu za účelom riešenia MKP problému

Návrh prezentovaný v podkapitole 5.1 je príliš obecným. Došlo k prispôsobeniu návrhu, pretože program určený na riešenie zvolenej úlohy rieši len túto konkrétnu úlohu a žiadnu ďalšiu. Problém spomínanej prílišnej obecnosti tkvie v tom, že reálne problémy vyžadujú dáta, ktoré sú pre každý problém špecifické. Tieto dáta by bolo možné takisto popísať s vyššou mierou abstrakcie, avšak každá pridaná abstrakcia sa negatívne podpíše na dosiahnutom výkone. Takisto pri konkrétnych algoritmoch dochádza častokrát k úprave aj samotného algoritmu PSO, takže je výhodné prispôsobiť návrh vždy konkrétnemu problému.

V tejto podkapitole rozoberiem rozdiely v obidvoch návrhoch. Implementačné detaily už nebudú popisované, pretože nemajú vplyv na zvýšenie dosiahnutého výkonu. Prispôsobením prešla až plne optimalizovaná verzia programu.

Z predchádzajúceho návrhu boli odstránené moduly, ktoré nie sú zásadné a potrebné z pohľadu riešenia problému. Boli odstránené moduly **math**, **cpudb** a **performance**. Modul **benchmark** bol spojený s modulom **swarm** do modulu **knapsack**. Úloha modulu **knapsack** je v zásade rovnaká ako u modulu **swarm** z predchádzajúceho návrhu. Zmenou prešla akurát funkcia *update*, ktorá teraz implementuje diskrétnu variantu algoritmu PSO (podkapitola 2.4). Taktiež pribudol modul **data**, ktorý určuje formát vstupných dát.

Modul data

Formát vstupných dát bol zvolený rovnaký ako z voľne dostupnej databázy [9]. Tento formát popisuje nasledujúci príklad:

6	2	0				⇒ 6 objektov, 2 batohy, optimum (0 ak nie je známe)
1	3	7	5	9	2	⇒ Profity objektov
4	2	1	1	3	8	⇒ Matica cien (2×6)
5	3	9	2	3	7	
10	20					⇒ Vyhradený rozpočet (kapacita batohov)

5.6 Využité nástroje

V tejto podkapitole sú uvedené stručné informácie o nástrojoch využitých pri implementácii. U parametrov kompilátora sú uvedené len tie, ktoré majú aspoň nejaký vplyv na dosiahnutý výkon programu.

GCC

Verzia: 4.8.0
Kompilácia za využitia optimalizačných príznakov:

-O3 -fopenmp -ffast-math -funroll-loops -DNDEBUG

Príznaky automaticky zahrnuté využitím **-O3** a tiež ich význam je popísaný v dokumentácii GNU GCC [13].

OpenMP

OpenMP je platformne nezávislé API. Podporuje paralelné programovanie so zdieľanou pamäťou v jazykoch C/C++ a Fortran. Je jednotne definované skupinou významných výrobcov v oblasti hardwaru a softwaru. Využíva sa v aplikáciách pre širokú škálu platforiem od PC až po superpočítače. Vlastníkom značky OpenMP je nezisková organizácia *Architecture Review Board* [3].

PAPI

API s konzistentným rozhraním a metodológiou zabezpečujúcou prístup k dostupným hardwarovým počítadlám na meranie výkonu. Umožňuje užívateľovi takmer v reálnom čase vidieť vzťah medzi výkonom softwaru a udalosťami na procesore [19].

Metriky získané za použitia PAPI:

- MFLOPS
- Počet výpadkov v L1 cache

GNU gprof

Profiler využívajúci „vzorkovania“ programového počítadla (PC) profilovaného programu v špecifických intervaloch. Dodatočný kód, ktorým je program rozšírený v procese kompilácie, je spojený s významnou réžiou. Táto réžia sa prejaví na čase behu programu niekoľkonásobným spomalením. Je možné ju regulovať úpravou intervalu vzorkovania na úkor presnosti [2].

Na zapnutie profilovania pomocou gprofu [14] je nutné skompilovať program s dodatočným príznakom **-pg** (podpora u GCC a INTEL® kompilátorov).

Výstupom profilácie je *flat profile*, ktorý zobrazuje množstvo času stráveného v každej funkcii programu, ako aj počet volaní týchto funkcií.

6. Implementácia na GPU

Implementácia na CPU popísaná v predošlej kapitole bola ďalej akcelerovaná na GPU za cieľom dosiahnutia ešte vyššieho výkonu. Návrh CPU implementácie je z veľkej časti rovnaký pre GPU, preto sa v nasledujúcej podkapitole zameriam len na rozdiely v návrhu a následne na spôsob implementácie akcelerátora.

6.1 Návrh akcelerátora

Architektúra GPU aplikácie vychádza z jej CPU návrhu. Má takisto 5 modulov **main**, **knapsack**, **data**, **random** a **swarm**, pričom v poslednom spomínanom sa líši od modulu **particle** z CPU architektúry.

Rozdiel v týchto moduloch spočíva v rozdiele prístupu do pamäti. Pre CPU je optimálny prístup do matice (napr. pole štruktúr môže vystupovať ako matica) po riadkoch a to kvôli možnosti využiť veľkú cache, ktorou disponuje CPU. Na druhej strane pre GPU je optimálny prístup po stĺpcoch, pretože minimalizuje transakcie a celkový počet prenesených bajtov z pamäti [25]. Modul **swarm** už teda nepopisuje 1 časticu roja, ale celý roj.

Modul **swarm** obsahuje niekoľko tried. Trieda *Position* popisuje polohu častíc v priestore problému (*pos*), ich najlepšiu zatiaľ dosiahnutú polohu (*sol*) a fitness ich najlepšie dosiahnutej polohy (*bst*). Trieda ďalej obsahuje počet častíc v roji (*cnt*) a počet dimenzií problému (*dms*). Prístup k atribútom triedy je prostredníctvom metód, ktoré pridávajú úroveň abstrakcie a minimalizujú tak možné chyby, ktoré by mohli vzniknúť kvôli zvolenému formátu uloženia dát. Atribút *str* zabezpečuje, že počet dimenzií jednotlivých častíc je zarovnaný na najbližšiu mocninu čísla 2 (dôvod bude popísaný ďalej).

```
1  namespace Swarm
2  {
3    ...
4    class Position
5    {
6    ...
7    private:
8      const unsigned cnt, dms, str;
9      float * bst;
10     bit_t * pos, * sol;
11   ...
12   public:
13     inline bit_t position(unsigned p, unsigned d)
14     {
15       return (pos + p * str)[d];
```

```
16      }
17      ...
18      inline void setBest(unsigned i, float v)
19      {
20        if(v > bst[i])
21        {
22          bst[i] = v;
23
24          memcpy(sol + i * str, pos + i * str, dms * sizeof(bit_t));
25        }
26      }
27    ...
28    };
29    ...
30  }
```

Z povahy fungovania algoritmu by na zakódovanie polohy častice stačil v každej dimenzii 1 bit. Týmto spôsobom je to riešené na CPU, pretože dimenzie každej častice sú spracovávané sekvenčne. Na GPU je na každú dimenziu vyhradené 1 vlákno z bloku (bude popísané neskôr) a teda pri použití 1 bitu by táto oblasť vytvárala kritickú sekciu, pretože najmenšou adresovateľnou jednotkou je bajt. Z tohto dôvodu je poloha častice v každej dimenzii zakódovaná do 1 bajtu.

Ďalšou triedou, ktorá sa v tomto module nachádza je trieda *Data*, ktorá obsahuje maticu rýchlostí pre každú dimenziu každej častice (*vel*) a index lídra roja (*gbest*). Dôvod prečo nie je táto trieda zlúčená s triedou *Position* je, že na výpočet fitness riešenia postačuje poloha častíc, rýchlosť tým pádom nie je dôležitá.

```
1  namespace Swarm
2  {
3    ...
4    class Data
5    {
6      ...
7    private:
8      unsigned gbest,
9      float * vel;
10     Position pos;
11     ...
12   };
13   ...
14 }
```

Ostatné triedy, ktoré sa nachádzajú v tomto module zabezpečujú alokáciu pamäte na GPU pred zahájením výpočtu a takisto prenesenie dát medzi CPU a GPU (napr. prenesenie výsledku z GPU na CPU pred prezentovaním nájdeného riešenia).

6.2 Implementácia akcelerátora

Z profilácie CPU implementácie sa ukázali funkcie *update* a *fitness* ako časovo kritické a preto je dôležité sa zamerať práve na tieto dve časti aplikácie. Vzhľadom k tomu, že tieto funkcie realizujú kompletný výpočet, dáta musia byť po celý čas na GPU. Preto aj časti, ktoré by bolo možné ponechať v réžii CPU je nutné realizovať prostredníctvom GPU a to z dôvodu zamedzenia prenášania dát medzi CPU a GPU, ktoré drasticky znižuje výkon.

Nižšie je uvedená ukážka hlavnej časti celej aplikácie (a takisto algoritmu PSO) a to hľadanie riešenia. V každej iterácii algoritmu sa po spočítaní fitness jednotlivých častíc zisťuje, ktorá častica je lídrom roja, tj. časticou s najlepším nájdeným riešením.

```
Swarm::sol_t MKPSolver::solve(size_t epochs, size_t stop)
{
  ...
  for(size_t m = 0; n < epochs; ++n, ++m)
  {
    fitness<<<...>>>(...);
    ...
    if(swarm.findBest()) m = 0;

    if(m >= stop) break;

    update<<<...>>>(...);
  }
  ...
  return swarm.solution();
}
```

Hľadanie častice s najlepším nájdeným riešením prebieha v metóde *findBest*, v ktorej dôjde k spusteniu redukčného kernelu z knižnice **thrust**. Toto je práve časť programu, ktorej výpočet by bolo vhodnejšie realizovať s využitím CPU, pretože počet častíc je relatívne malý a spustenie kernelu zas relatívne náročná operácia. Ako ale už bolo spomínané, tento výpočet musí byť ponechaný v réžii GPU, pretože dáta sú uložené tam. V nasledujúcej ukážke je táto metóda zobrazená.

```
namespace Swarm
{
  ...
  typedef char bit_t;
  ...
  class Device
  {
    ...
    inline bool findBest()
    {
      thrust::device_ptr<float> dp(d_data.pos.bst);

      size_t bst = thrust::max_element(dp, dp + d_data.count()) - dp;

      if(d_data.gbest != bst)
      {
```

```
17          d_data.gbest = bst;
18
19          return true;
20        }
21
22        return false;
23      }
24      ...
25    };
26    ...
27  }
```

Funkcia $findBest()$ vracia príznak, či sa líder zmenil. Tento príznak je rozhodujúci pre zastavenie hľadania riešenia, ak sa užívateľ rozhodol využiť túto skutočnosť ako podmienku pre jeho ukončenie (štandardne sa však hľadá riešenie po celý počet iterácií). V tejto časti došlo k zmene oproti CPU verzii, ktorá v tejto sekcii algoritmu vracia, či došlo k zmene globálne najlepšieho fitness, tj. aj pre súčasne najlepšieho lídra. Aby táto funkčnosť bola zachovaná aj na GPU, bolo by nutné, aby si častice pamätali aj svoju predchádzajúcu fitness (alebo aspoň príznak či došlo k jej zmene), toto by však zväčšilo pamäťové nároky, zvýšilo počet prístupov do pamäti a znížilo tak celkový výkon aplikácie. Na druhú stranu neexistuje presný a ideálny počet, ktorý by udával koľko iterácií, pri ktorých nedôjde k zmene globálne najlepšieho fitness, postačuje na nájdenie optimálneho riešenia. Užívateľ teda môže voliť počet iterácií, počas ktorých nedôjde k zmene lídra roju.

Aj keď funkcia *init* nie je z pohľadu výkonu časovo kritickou časťou aplikácie, je jej implementácia jednoduchá. Nižšie je zobrazený kernel, ktorý inicializuje častice roja. Dôvod, prečo nie je inicializácia realizovaná na CPU, je zamedzenie zbytočne ďalšej alokácii pamäte pre častice na CPU pred tým, než by boli prenesené na GPU a následne z pamäti uvoľnené.

```
1  __global__ void init(Swarm::Data swarm, float vMax, Seed4 seed)
2  {
3    unsigned p = blockIdx.x * blockDim.x + threadIdx.x;
4
5    if(p >= swarm.count()) return;
6
7    Random4x32<float> rnd(p, 0xcaffe123, seed);
8
9    for(unsigned d = 0; d < swarm.dims(); ++d)
10   {
11     swarm.setPosition(p, d, rnd() < 0.5f ? 1 : 0);
12     swarm.velocity(p)[d] = rnd() * vMax;
13   }
14 }
```

Kernel update

V tejto podkapitole rozoberiem prístup k návrhu kernelu update. Popíšem 2 kernely, ktoré by bolo možné k riešeniu použiť. Prvý kernel *update1*, ktorý popíšem, je z pohľadu výkonu suboptimálny, ale nesie zo sebou výhodu, že počet dimenzií problému nie je limitovaný tak ako v prípade kernelu *update2* a takisto poloha častíc môže byť kódovaná 1 bitom v každej dimenzií.

```
__global__ void update1(Swarm::Data swarm, ... paramUpdate, ... seed)
{
  unsigned p = blockIdx.x * blockDim.x + threadIdx.x;

  if(p >= swarm.count()) return;

  Random4x32<float> rnd(p, 0xabcf0198, seed);

  for(unsigned d = 0; d < swarm.dims(); ++d)
  {
    float v = paramUpdate.iW * swarm.velocity(p)[d]
      + paramUpdate.lC1 * rnd()
      * (swarm.solution(p, d) - swarm.position(p, d))
      + paramUpdate.lC2 * rnd() * (swarm.solution(swarm.best(), d)
      - swarm.position(p, d));

    v = clamp(v, paramUpdate.vMax);

    swarm.velocity(p)[d] = v;

    float S = 1.f / (1.f + expf(-v));

    swarm.setPosition(p, d, (rnd() < S) ? 1 : 0);
  }
}
```

V kerneli sa nachádza cyklus, ktorý značne brzdí výkon aplikácie, pretože dimenzie sú vyhodnocované sekvenčne pre každú časticu. V podkapitole 6.3 je porovnaný výkon aplikácie využívajúcej tento kernel spolu s kernelom *fitness1* (popísaný ďalej, je ale riešený taktiež cyklom). Je možné vidieť, že výkon je oproti optimalizovanej aplikácii na CPU vyšší, ale až pri väčšom počte častíc.

```
__global__ void update2(Swarm::Data swarm, ... paramUpdate, ... seed)
{
  unsigned p = blockIdx.x;
  unsigned d = threadIdx.x;

  Random4x32<float> rnd(p * swarm.dims() + d, 0xabcf0198, seed);

  float v = paramUpdate.iW * swarm.velocity(p)[d]
    + paramUpdate.lC1 * rnd()
    * (swarm.solution(p, d) - swarm.position(p, d))
    + paramUpdate.lC2 * rnd() * (swarm.solution(swarm.best(), d)
    - swarm.position(p, d));

  v = clamp(v, paramUpdate.vMax);

  swarm.velocity(p)[d] = v;

  float S = 1.f / (1.f + expf(-v));

  swarm.setPosition(p, d, (rnd() < S) ? 1 : 0);
}
```

Optimalizovaný kernel *update2* ohodnocuje každú dimenziu častice jedným vláknom a teda neobsahuje cyklus. Kernel je spúšťaný s počtom blokov veľkostí roja, tj. počtu častíc a s počtom vlákien v každom bloku veľkosti problému, tj. počtu dimenzií. Takéto spúšťanie kernelu vytvára obmedzenie vo veľkosti problému a to konkrétne 512 dimenzií na GPU architektúry *Tesla* a 1024 pre architektúru *Fermi*. Takáto veľkosť problému je však plne postačujúca, pretože väčšie problémy už nie je vhodné riešiť pomocou jednej GPU, ale použiť cluster kariet a distribuovať výpočet medzi ne.

Kernel fitness

Táto podkapitola podobne, ako predchádzajúca, rozoberie prístup k tvorbe kernelov pre fitness funkciu. Uvediem takisto 2 možné kernely. Prvý kernel *fitness1*, zobrazený nižšie, je kernel obsahujúci cyklus. Má v zásade rovnaké výhody a nevýhody ako kernel *update1*.

```
__global__ void fitness1(Swarm::Position x, MKPSolver::ParamData data)
{
  unsigned p = blockIdx.x * blockDim.x + threadIdx.x;

  if(p >= x.count()) return;

  float f = 0;

  for(unsigned i = 0; i < data.n; ++i)
    f += data.d_pro[i] * x.position(p, i);

  float spl = 0.f;

  for(unsigned i = 0; i < data.m; ++i)
  {
    float m = 0;

    for(unsigned j = 0; j < data.n; ++j)
      m += data.d_wei[i * data.n + j] * x.position(p, j);

    spl += poslin(data.penalty * (m - data.d_cap[i]));
  }

  f -= spl;

  x.setBest(p, f);
}
```

Kernel *fitness2*, zobrazený nižšie, spracúva každú dimenziu problému 1 vláknom zas tak, ako v prípade kernelu *update2*. Kernel pracuje so zdieľanou pamäťou, nad ktorou je počítaná redukcia (*rdSum*). V tomto prípade je použité telo rozbaleného plne optimalizovaného kernelu vykonávajúceho súčet prvkov, publikovaného v [16]. Vďaka spôsobu akým funguje je nutné, aby bol počet dimenzií mocninou čísla 2 a týchto dimenzií bolo minimálne 64. Toto v prípade kernelu *fitness1* nie je potrebné.

```
template <unsigned blockSize>
__global__ void fitness2(Swarm::Position x, MKPSolver::ParamData data)
{
  unsigned p = blockIdx.x;
  unsigned d = threadIdx.x;

  __shared__ float sdata[blockSize];

  sdata[d] = data.d_pro[d] * x.position(p, d);

  rdSum<blockSize>(sdata);

  float f = sdata[0];

  float spl = 0.f;

  for(unsigned i = 0; i < data.m; ++i)
  {
    sdata[d] = data.d_wei[i * blockSize + d] * x.position(p, d);

    rdSum<blockSize>(sdata);

    spl += poslin(data.penalty * (sdata[0] - data.d_cap[i]));
  }

  f -= spl;

  if(d == 0) x.setBest(p, f);
}
```

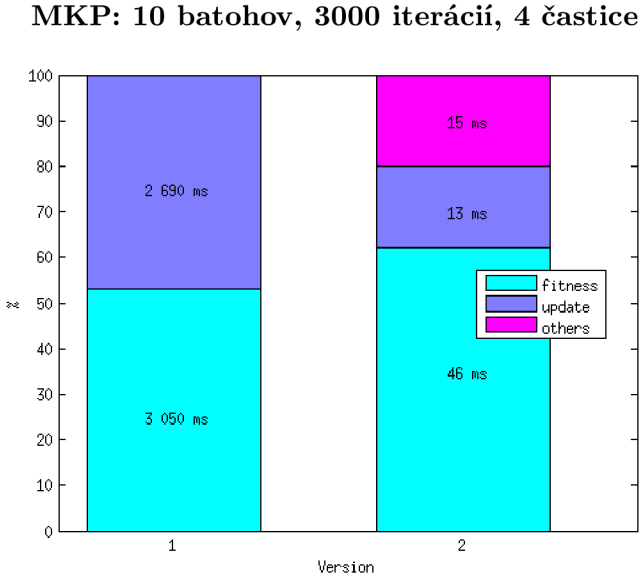
V kerneli je však naďalej možné vidieť cyklus, tentokrát ale cez počet batohov. Tomuto cyklu sa však nie je možné vyhnúť (tak aby to napr. negatívne neovplyvnilo veľkosť problému, ktorý je možný ešte riešiť). Tento cyklus obsahuje v zásade malý počet iterácií, pretože povaha MKP problému je prevažne v počte dimenzií.

Takisto vyvolanie redukcie v každej iterácii nie je možné presunúť až za cyklus (čo by zvýšilo výkon, pretože by nedochádzalo k neustálej synchronizácii vlákien). Toto nie je možné vykonať ani v prípade, že by bol kernel vyvolaný s veľkosťou zdieľanej pamäte násobenou počtom batohov problému. Jednak je veľkosť zdieľanej pamäte obmedzená, takže by sa tomuto problému nedalo vyhnúť pri väčšom počte batohov. Dôležitejšie je ale to, že redukciu by nebolo možné vykonávať nad maticou naplnenou váhami jednotlivých zahrnutých objektov, pretože na každý riadok tejto matice musí byť aplikovaná funkcia *poslin* (podkapitola 4.2), ktorá nie je lineárna (neplatí teda $f(a+b) = f(a) + f(b)$).

6.3 Profilácia a porovnanie výkonu

V tejto podkapitole porovnám 2 verzie aplikácie pre GPU spolu s optimalizovanou verziou pre CPU. Podrobné porovnanie a vyhodnotenie optimalizovaných verzií pre CPU a GPU ako aj celkový prínos akcelerácie s využitím GPU bude popísaný v samostatnej kapitole 7.

Verzia 1 označuje aplikáciu, v ktorej boli využité kerneli update1 + fitness1. Verzia 2



Obrázok 6.1: Profilácia implementácie GPU akcelerovanej aplikácie. Z obr. je možné vidieť jednak minimalizáciu réžie algoritmu PSO (funkcia *update*), ale aj približne 100-násobné zrýchlenie verzie 2 oproti verzii 1. Časy figurujúce v jednotlivých častiach stĺpcov sú časy strávené v jednotlivých kerneloch z celkového času behu aplikácie.

potom využíva kerneli update2 + fitness2. Na obr. 6.1 je z profilácie obidvoch verzií možno vidieť minimalizovanie réžie algoritmu (aktualizácia častíc kernelom *update*). Kým verzia 1 strávila približne 50% réžiou, verzia 2 už len cca 20%. Rovnako došlo aj k celkovému zrýchleniu aplikácie. Kernely vo verzii 2 vykazujú oproti verzii 1 priemerné zrýchlenie takmer 100×. Porovnanie výkonu obidvoch verzií spolu s optimalizovanou CPU verziou je zachytené na obr. 6.2. Z obrázku je očividné, že CPU vykazuje nižší výkon ako Verzia 1 (gpu 1) až pri vysokom počte častíc.

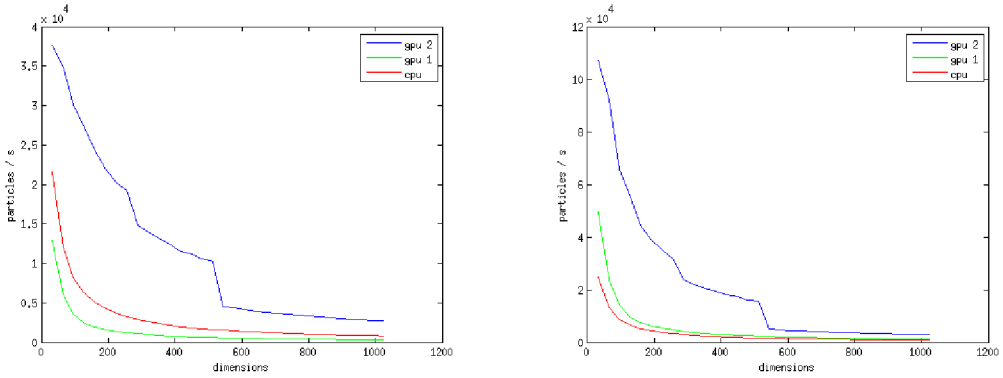
6.4 Využité nástroje

Na implementáciu aplikácie bola využitá multiplatformná a bezplatná sada nástrojov *CUDA Toolkit v5* [29] ako aj kompilátor g++ (v. 4.8.0) z *GNU GCC* [13]. Z CUDA Toolkit boli konkrétne využité nástroje *NVIDIA CUDA Compiler* (nvcc), knižnica *Thrust* a *NVIDIA Visual Profiler*.

Parametre kompilácie pre g++

-msse4.1 -ffast-math -funroll-loops -pedantic -O3 -DNDEBUG

MKP: 1 batoh, 100 iterácií



(a) 256 častíc

(b) 1024 častíc

Obrázok 6.2: Porovnanie výkonu. Z obidvoch zobrazených obrázkov je očividná prevaha druhej verzie GPU implementácie nad ostatnými. Verzia 1 oproti optimalizovanej CPU verzii vykazuje zvýšenie výkonu až pri vysokom počte častíc, ktoré ale už nemajú prínos na kavlitu riešenia (rozobrané v kap. 7).

Parametre kompilácie pre nvcc

-gencode arch=compute_10,code=sm_10 -gencode arch=compute_20,code=sm_20 -gencode arch=compute_30,code=sm_30 -gencode arch=compute_35,code=sm_35 -DNDEBUG -O3 -m64 –use_fast_math -D__STDC_CONSTANT_MACROS

7. Výsledky

Optimalizované verzie CPU a GPU implementácie (v kontexte na ne bude odkazované v skratke ako na CPU a GPU) boli podrobené testom za účelom vyhodnotenia prínosu akcelerácie. V tejto kapitole sú aplikácie vyhodnocované jednak z výkonového hľadiska, ale takisto z hľadiska kvality nájdeného riešenia.

Na ohodnocovanie výkonu boli vygenerované vlastné dáta, pretože bolo potrebné vyhodnocovať aplikácie na špecifických množinách rôznej veľkosti MKP problému. Vzhľadom k tomu, že sa porovnáva len dosiahnutý výkon a nie kvalita, tak nezáleží na tom, na akých dátach sa aplikácie testujú. Dokonca by nevadilo ani keby problém nemal riešenie, alebo parametre PSO by boli zvolené nevhodne (pokiaľ rovnako nevhodne pre obidve testované aplikácie).

Na ohodnocovanie kvality riešenia ako aj na experimenty s parametrami algoritmu PSO boli použité voľne dostupné dáta, ktoré zahrňujú rôzne problémy a niektoré aj s ich riešením, tj. s nájdeným globálnym optimom. Využité databázy [9] sú vo formáte aký je použitý v aplikáciách, ďalšie dostupné databázy [17] sú ale už v inom formáte (ten však nie je problém prispôsobiť).

7.1 Testovacia zostava

Problém bol riešený na PC zostave, ktorá disponovala CPU s 4 jadrami **Intel®Core™ i7 920** a grafickou kartou **NVIDIA GeForce GTX 580**.

Parametre CPU sú zobrazené v tab. 7.1, parametere GPU v tab. 7.2. Výpočet teoretického výkonu P, ktorý je v tabuľkách uvedený, vychádza z rovnice [39]:

$$P = \text{chips} * \text{cores} * \text{width}_{\text{vector}} * \frac{\text{FLOPs}}{\text{cycle}} * \text{clock} \tag{7.1}$$

Z tabuliek je možno vidieť, že teoretický výkon GPU oproti CPU je približne 36-násobný (1536/42).

Tabuľka 7.1: Parametre CPU.

Intel Core i7 920	
CPU clock rate	2664 MHz
Number of cores	4
L3 cache size	8 MB
Theoretical performance	42 GFLOPS

Tabuľka 7.2: Parametre GPU.

NVIDIA GeForce GTX 580	
CUDA capability	2.0
Total amount of global memory	1536 MB
CUDA cores	512
GPU clock rate	1564 MHz
Memory clock rate	2004 MHz
L2 cache size	786432 B
Total amount of shared memory per block	49152 B
Total number of registers per block	32768
Warp size	32
Maximum number of threads per MP	1536
Maximum number of threads per block	1024
Theoretical performance	1572 GFLOPS

7.2 Experimenty

Aplikácie boli podrobené sade testov (s rôznymi vstupnými dátami o rôznych veľkostiach problému), pričom uvediem výsledky len niektorých z nich, ostatné možno nájsť v prílohe A. Všetky uvedené výsledky sú namerané a spriemerované z 30 behov. Grafy zobrazujú priemerný výsledok bez extrémov (pokiaľ tam nie sú explicitne uvedené). V tab. 7.3 sú uvedené parametre algoritmu PSO aké boli využité k testovaniu aplikácií.

Tabuľka 7.3: Parametre PSO.

Parameter	**Hodnota**
Stopping criteria	0
Inertia coefficient	1
Cognitive coefficient	2
Social coefficient	2
Maximum velocity	2
Penalty	200

Prvým testom, ktorým boli aplikácie podrobené, bol test výkonu. Boli zvolené dve metriky na porovnávanie výkonnosti. Konkrétne počet miliárd operácií s pohyblivou čiarkou za sekundu (GFLOPS) a počet ohodnotených častíc za sekundu (particles / s). Druhý test je zameraný na porovnávanie kvality nájdeného riešenia.

Výkon aplikácií

V grafe na obr. 7.1 sú uvedené výsledky testov, ktoré zachycujú závislosť výkonu aplikácie v počte častíc ohodnotených za jednotku času na veľkosti problému. V grafe je možné vidieť, že GPU dosahuje celkovo vyššieho výkonu a globálne vyššieho až od určitej veľkosti problému alebo počtu častíc. Pre veľkosť problému, ktorý je menší ako cca 100 dimenzií je pri počte 32 častíc možno vidieť, že CPU dosahuje vyššieho výkonu. Takáto veľkosť pro-

blému pri tomto počte častíc je príliš malá na to, aby dostatočne vyťažila GPU, takisto sa prejaví réžia spojená so spúšťaním kernelu. Pri menších problémoch je však možné hľadať riešenie na GPU s vyšším počtom častíc a nižším počtom iterácií, takže by v konečnom dôsledku mohlo byť dosiahnuté aj pre takto malé problémy rovnakého výkonu. V nasledujúcej tabuľke je zhrnuté zrýchlenie dosiahnuté akceleráciou na GPU. Priemerné zrýchlenie bolo výpočítané ako podiel priemerných zrýchlení (cez všetky dimenzie) GPU a CPU verzie. Podobne maximálne zrýchlenie ako podiel GPU a CPU verzie, pričom bola vybraná maximálna nájdená hodnota. Pre vyšší počet častíc by zrejme mohlo byť dosiahnutého ešte vyššieho nárastu zrýchlenia.

Tabuľka 7.4: Priemerné a maximálne zrýchlenie dosiahnuté akceleráciou.

MKP: 4 batohy, 3000 iterácií

	Zrýchlenie GPU oproti CPU	
Počet častíc	Priemerné	Maximálne
32	1.9	3.5
64	2.7	5.0
128	3.7	7.0
256	5.2	9.6

V grafe 7.1 je v GPU verzii aplikácie zrejmý skokový pokles výkonu. Toto je zapríčinené tým, že dimenzie problému sú zaokrúhľované na najbližšiu vyššiu mocninu čísla 2, tak ako je popísané v kap. 6. Značný výkonový schodok teda nastáva ak je prekročených 64, 128 alebo 512 dimenzií, pretože počet dimenzií je zdvojnásobený. V skratke, skokové zvýšenie obnosu dát znamená skokové zníženie výkonu. Ak ale platí, že 513 dimenzií je automaticky 1024 dimenzií, potom by mali zobrazené krivky vykazovať plne schodovitý tvar. Pritom je možné vidieť, že pri počte 513 dimenzií je dosiahnutého vyššieho výkonu ako pri 1024 dimenziách. Tento jav je spôsobený tým, že kernel *update*, ktorého úlohou je aktualizácia častíc roju (popísaný v kap 6), nemusí a teda ani neaktualizuje častice v dimenziách, ktoré nemajú na nájdené riešenie vplyv.

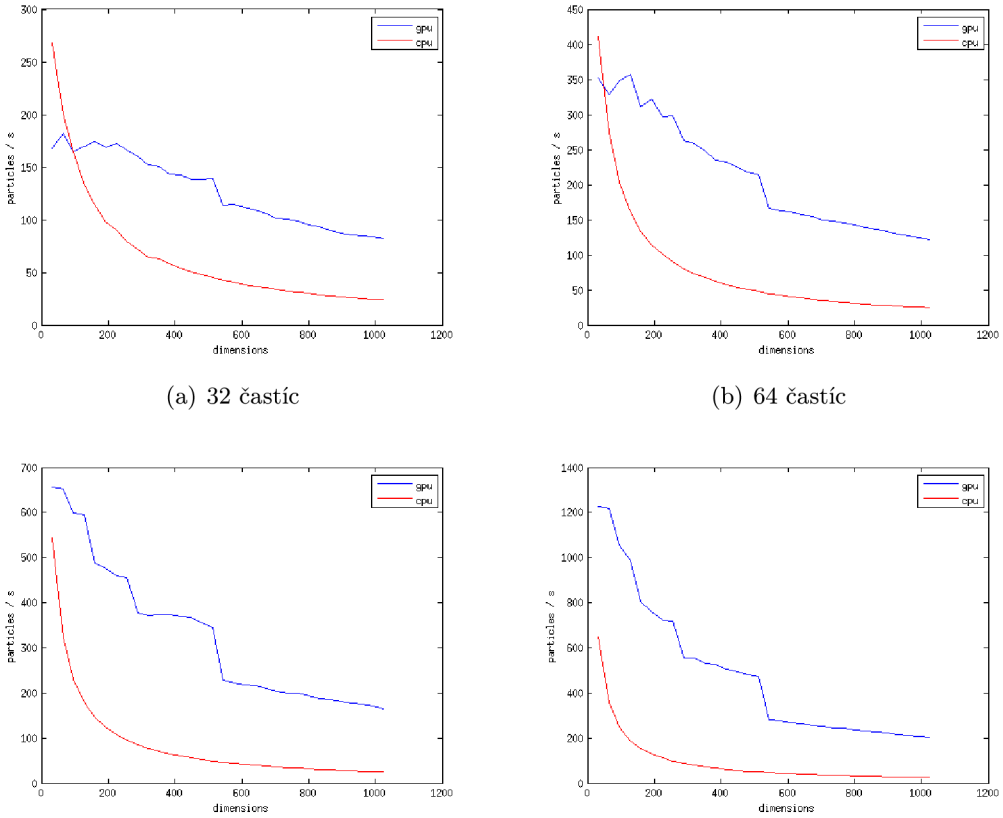
V grafe na obr. 7.1 figurujú konštanty – počet batohov problému a počet iterácií algoritmu. Ich zmenšovanie alebo zväčšovanie ovplyvňuje nameranú výkonnosť približne lineárne pre CPU aj GPU. Počet častíc ovplyvňuje CPU taktiež približne lineárne, avšak na dosiahnutý výkon aplikácie na GPU by mala mať vplyv len minimálny. Je to nakoniec možné vidieť v grafe na obr. 7.2. Z tohto obr. taktiež vidno, že na plnú saturáciu GPU pri 500 dimenziách je treba cca 600 častíc.

Výsledok druhej výkonnostnej metriky ukazuje závislosť veľkosti problému na počte GFLOPS. Je zobrazený na obr. 7.3. Z grafu je zrejmé, že rovnako ako pri predchádzajúcej metrike je výkon GPU celkovo vyšší. Takisto ako u prvej výkonnostnej metriky je aj u tejto možné vidieť pokles výkonu s narastajúcim počtom dimenzií, čo sa dá predpokladať. Obidve metriky zo sebou nepriamo súvisia, takže je možné odvodiť podobný úsudok ako pri prvej metrike.

Kvalita nájdeného riešenia

Nasledujúce testy sú zamerané na kvalitu nájdeného riešenia. Aplikácie sú porovnávané za cieľom potvrdenia správnosti akcelerácie využitím GPU. Okrem overenia či akcelerácia

MKP: 4 batohy, 3000 iterácií



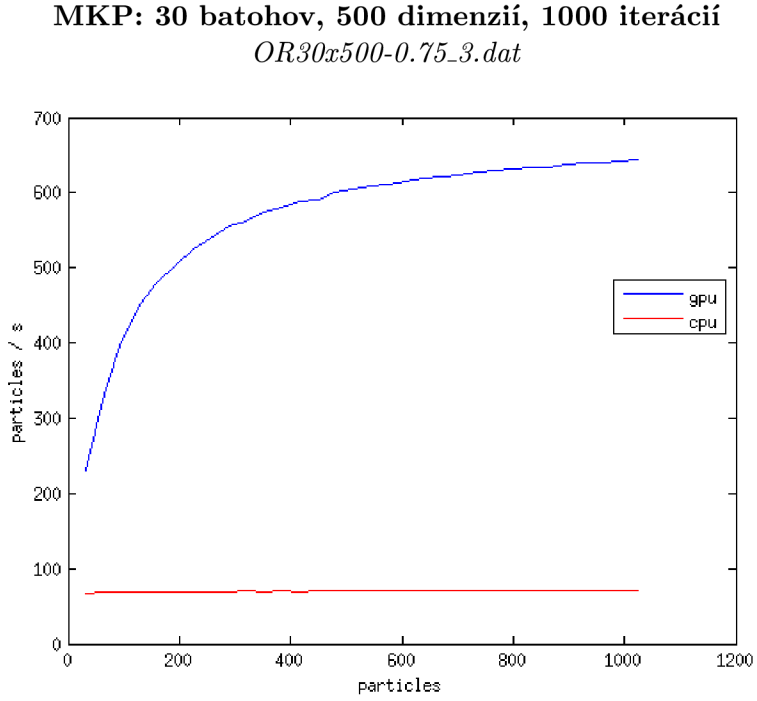
(a) 32 častíc

(b) 64 častíc

(c) 128 častíc

(d) 256 častíc

Obrázok 7.1: Porovnanie výkonu aplikácií CPU a GPU. S pribúdajúcim počtom častíc stúpa rozdiel vo výkone medzi CPU a GPU verziami aplikácií. Stúpa takisto aj celkový výkon obidvoch aplikácií, tj. počet ohodnotených častíc za sekundu (graf na obr. 7.2).



Obrázok 7.2: Závislosť výkonu na počte častíc. CPU aj GPU verzie aplikácií vykazujú nárast výkonu s pribúdajúcim počtom častíc. V CPU však dôjde k saturácií výkonu o mnoho skôr než v GPU.

nezaniesla chyby do vyhľadávania riešenia, sú aplikácie testované aj z hľadiska priblíženia sa (nájdenia) k optimálnemu riešenia (na dátach ktoré túto informáciu obsahujú). Rovnako ako pri predošlých metrikách uvediem len niektoré výsledky, ostatné je možno nájsť v prílohe.

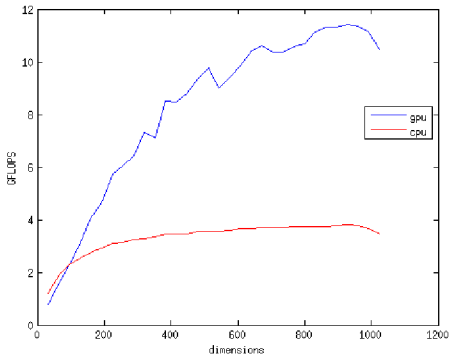
Graf na obr. 7.4 ukazuje vplyv počtu častíc na kvalitu nájdeného riešenia. Priebeh je zrejme pre CPU aj GPU približne rovnaký, čo potvrdzuje aj graf na obr. 7.5. Odchýlky sú spôsobené stochastickými zložkami algoritmu. Pokiaľ by boli aplikácie spustené s rovnakou počiatočnou konfiguráciou pseudonáhodného generátoru, tak by dospeli k rovnakým výsledkom.

Je nutné poznamenať, že u grafov 7.4 a 7.5 je treba vnímať skôr „pásmo“ než krivku, ktorá do neho spadá, pretože prechod medzi iteráciami nie je plynulý. Počet iterácií je vzorkovaný po 100 dimenziách, pričom v každej vzorke dochádza k znovuspusteniu algoritmu.

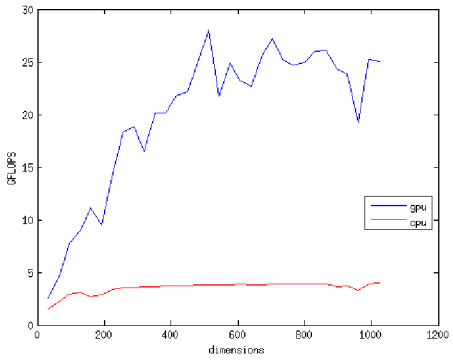
Z grafu 7.4 je vidno, že so zvyšujúcim sa počtom iterácií kvalita nájdeného riešenia nerastie, na rozdiel od zvyšujúceho sa počtu častíc. Toto je spôsobené tým, že časticový roj skonverguje o mnoho skôr než dobehne celkový počet iterácií. Ak však roj pozostáva z väčšieho počtu častíc, má vyššiu šancu preskúmať priestor problému a priblížiť sa tak k optimu. Algoritmus PSO je silne parametrizovateľný, takže neznamená, že kvalita riešenia nutne znamená zvýšiť počet častíc, poukazuje to len na to, že šanca nájsť optimum je vyššia. Vplyv počtu častíc na kvalitu riešenia je zachytený v tab. 7.5. Z tabuľky vidno, že zdvojnásobujúci sa počet častíc zmenšuje chybu riešenia približne na polovicu.

Na obr. 7.6 je uvedený graf, ktorý taktiež potvrdzuje závery o vplyve počtu častíc na kvalitu nájdeného riešenia. V grafe je ale možné vidieť, že približne od počtu častíc 256 nárast fitness nie je tak strmý. V grafe sú explicitne zobrazené extrémy nájdené v 30 meraných behoch pre každú vzorku.

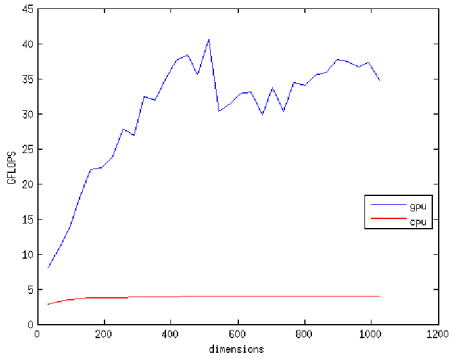
MKP: 1 batoh, 3000 iterácií



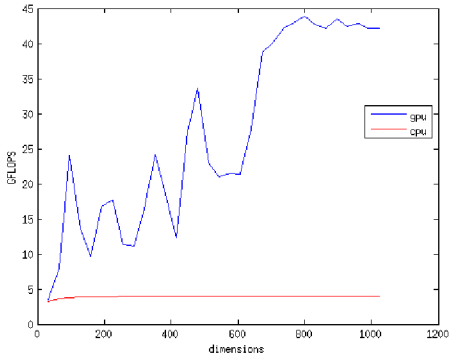
(a) 32 častíc



(b) 64 častíc



(c) 128 častíc



(d) 256 častíc

Obrázok 7.3: Porovnanie výkonu aplikácií. Grafy zachycujú podobne ako na obr. 7.1, vplyv počtu dimenzií (a častíc) na výkon aplikácie. Tentokrát je však zobrazená závislosť GFLOPS na počte dimenzií.

Tabuľka 7.5: Vplyv počtu častíc na kvalitu nájdeného riešenia.

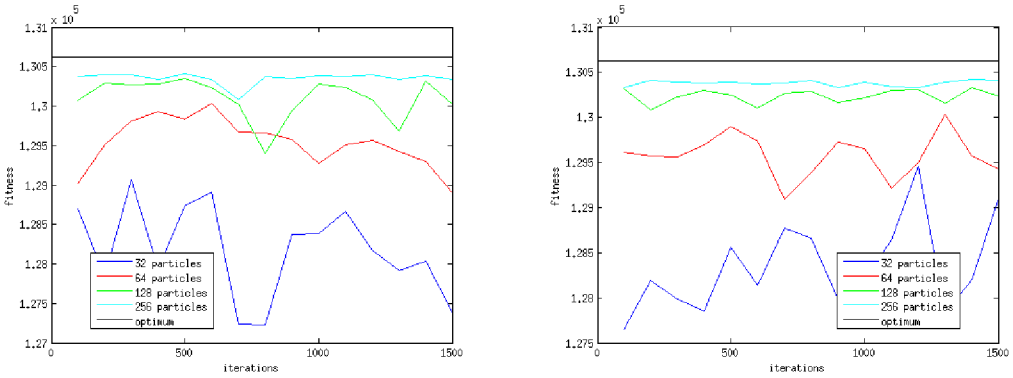
MKP: 2 batohy, 28 dimenzií

weing6.dat

Počet častíc	**Odchýlka od optima (%)**
32	1.88
64	0.83
128	0.40
256	0.21

MKP: 2 batohy, 28 dimenzií

weing6.dat

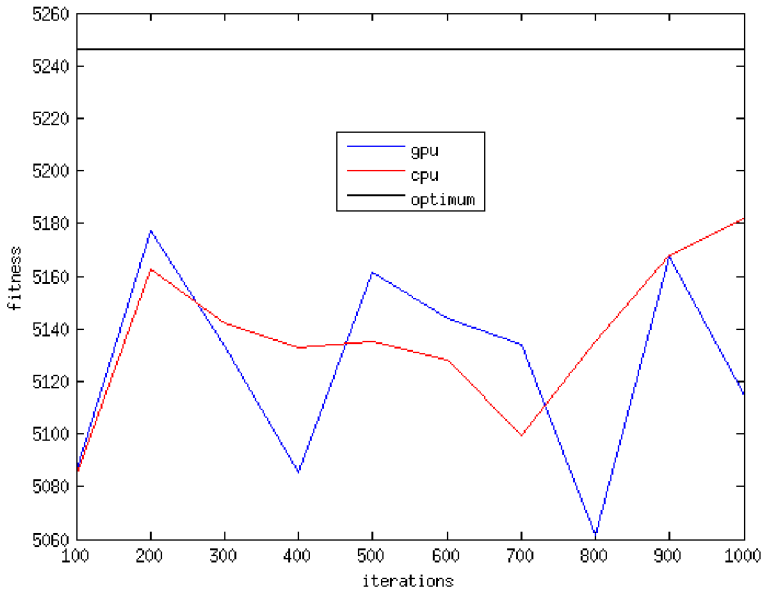


(a) GPU (b) CPU

Obrázok 7.4: Vplyv počtu častíc na fitness riešenia. Graf zobrazuje priemernú fitness riešenia z 30 behov. S množstvom častíc sa kvalita riešenia čím ďalej viac približuje k optimu. Pri zobrazení maximálnej fitness z 30 behov, by pri počte častíc 256 bola krivka totožná s hľadaným optimom.

MKP: 5 batohov, 40 dimenzií, 128 častíc

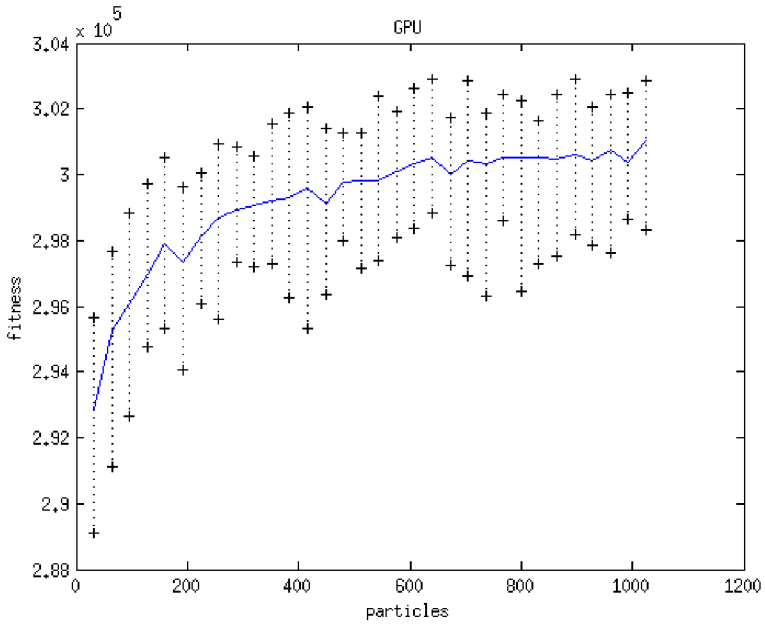
weish09.dat



Obrázok 7.5: Porovnanie fitness riešení CPU a GPU. Graf ukazuje ekvivalenciu v možnostiach nájdenia riešenia pre verzie CPU aj GPU. Ukazuje časť zloženia obidvoch grafov z obr. 7.4, tentokrát však pre inú inštanciu problému.

MKP: 30 batohov, 500 dimenzií, 1000 iterácií

OR30x500-0.75_3.dat



Obrázok 7.6: Závislosť fitness na počte častíc. V grafe je zobrazená GPU verzia, avšak CPU vykazuje rovnaký priebeh tak ako je ukázané v grafe na obr. 7.5. V grafe sú takisto zobrazené extrémy, tj. minimálna a maximálna nájdená fitness z 30 behov.

Na obr. 7.7 je možné vidieť časy behov riešenia problému z obr. 7.4. Verzia GPU nevykazuje tak výrazné rozdiely ako CPU čo sa týka časov behov pri použití rôzneho množstva častíc. Vzhľadom k tomu, že sa jedná o menšiu inštanciu problému sa u GPU v zásade nejedná o žiadny nárast času stráveného výpočtom. Pri väčšom počte dimenzií by sa rozdiely zväčšili, ale zas nie tak výrazne ako v prípade CPU, kde už pri takto malom počte dimenzií sú rozdiely zreteľné.

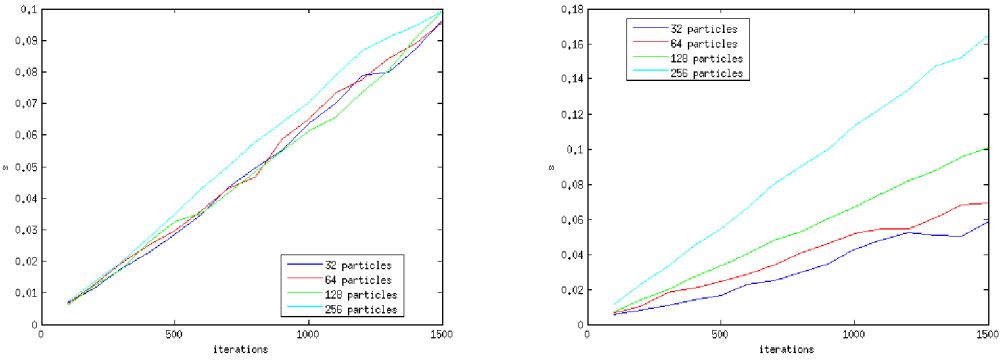
7.3 Zhodnotenie

Akcelerácia s využitím GPU si kládla za cieľ urýchliť výpočet riešenia, pričom kvalita nájdeného riešenia nebude ovplyvnená. Zo sady experimentov vyplynulo, že prínos akceleráciou je nesporný a na kvalite riešenia sa negatívne nepodpísal. Akceleráciou bolo dosiahnuté 5-násobné priemerné a takmer 10-násobné maximálne zrýchlenie výpočtu pri počte 256 častíc. Pri väčšom počte častíc má zrýchlenie GPU voči CPU tendenciu sa naďalej zvyšovať, avšak vyšší počet častíc už nemá výrazný vplyv na kvalitu dosiahnutého riešenia a zbytočne zvyšuje celkový čas strávený hľadaním riešenia.

V práci bol vytvorený nástroj, ktorý akceleruje algoritmus PSO. Namerané a vyhodnotené výsledky poukazujú na to, že algoritmus funguje. Práca si však nekládla za cieľ dosiahnuť čo najkvalitnejšieho riešenia MKP problému, či už vyladením parametrov PSO alebo využitím iných fitness funkcií, prípadne niektorých špecializovaných derivátov algoritmu

MKP: 2 batohy, 28 dimenzií

weing6.dat



(a) GPU (b) CPU

Obrázok 7.7: Porovnanie časov behov pre GPU a CPU verziu implementácie. Graf ukazuje čas strávený výpočtom v závislosti na počte častíc a iterácií. Použitie väčšieho množstva častíc nevykazuje u GPU verzie výrazne navýšenie času potrebného k vyriešeniu danej úlohy, na rozdiel od CPU.

PSO zameraných na MKP problém. Vytvorený nástroj slúži skôr ako odladená platforma pre problémy vhodné na riešenie pomocou PSO (prípadne jeho varianty). Adaptovanie nástroju na iný problém vyžaduje už len zmenu fitness (prípadne tiež update) funkcie a formátu vstupných dát.

Výhľad do budúcna

V práci bol vytvorený program, pomocou ktorého je možné riešiť problémy algoritmom PSO. Ak uvážime napr. problém MKP, na ktorom bol testovaný, tak užívateľ musel popísať inštanciu tohto problému v externom súbore a následne spustiť výpočet. Možným vylepšením, ktoré by sa naskytlo, by bolo prepojiť vytvorený program so známymi nástrojmi určenými k vedecko-technickým výpočtom, ako napr. Matlab. Konkrétne v prípade Matlabu by to mohlo byť realizované ako MEX funkcia. Takto by bolo možné využiť výkonný program priamo v procese tvorby niečoho väčšieho.

Ďalšou možnosťou k úprave programu by bolo podrobnejšie zbieranie štatistík, napr. o pohybe častíc v každej iterácii algoritmu. Tieto údaje by bolo možné následne vyzualizovať a použiť napr. na výukové účely zamerané na princípy fungovania algoritmu PSO.

V neposlednom rade by bolo možné upraviť vytvorený program za účelom využitia viacerých GPU, medzi ktorými by dochádzalo k distribúcii výpočtu. Táto úprava by zmiernila limity v podobe počtu dimenzií a množstve častíc riešiacich daný problém. Týmto by bolo umožnené riešiť ďaleko väčšie problémy, než tie, na ktoré je určená vytvorená aplikácia.

8. Záver

Táto práca bola zameraná na algoritmus PSO (*Particle Swarm Optimization*), ktorý dokáže efektívne riešiť aj také problémy, ktoré sú pri použití konvenčných prístupov výpočtovo príliš náročné. Práca si kládla za cieľ akcelerovať tento algoritmus a tým pádom znížiť čas potrebný k nájdeniu riešenia daného problému.

Najprv som vytvoril sekvenčnú implementáciu algoritmu PSO za účelom získania nadhľadu nad fungovaním tohto algoritmu. Vo vytvorenej aplikácii a taktiež v jej nasledujúcich verziách boli riešené multidimenzionálne funkcie Ackley, Sumcan a Weierstrass. Po overení funkčnosti sekvenčnej implementácie som pristúpil k jej ďalším optimalizáciám za cieľom získania čo najväčšieho výkonu na CPU. Prvá optimalizácia spočívala v paralelizácii algoritmu, tj. v možnosti využitia viacerých jadier, ktorými disponuje CPU. Táto optimalizácia, tak ako sa dalo očakávať priniesla značné zvýšenie výkonu. Ďalšia optimalizácia bola zameraná na využitie inštrukcií, ktoré realizujú výpočet nad vektorom dát. Na tento prípad by sa dalo pozerať ako na „paralelizáciu“ na úrovni vykonávania niektorých inštrukcií. Nasledujúca optimalizácia bola zameraná na rozbalenie cyklov algoritmu. Posledná optimalizácia, ktorá sa však najviac prejavila na dosiahnutom výkone, spočívala v úprave fitness funkcie a v nahradení matematických funkcií za ich optimalizované varianty, ktoré boli využívané pri riešení problému. Optimalizáciami bolo dosiahnuté priemerné cca 40-násobné zrýchlenie oproti základnej sekvenčnej implementácii.

Po tom čo bola vytvorená optimalizovaná CPU verzia aplikácie, došlo k jej adaptovaniu na riešenie MKP problému. Posledná fáza úprav sa týkala využitia GPU k akcelerovaniu optimalizovanej CPU verzie. Maximálnej efektivity GPU verzie bolo dosiahnuté priradeným jedneho vlákna pre každú dimenziu každej častice, využitím zdieľanej pamäti a použitím rozbaleného redukčného kernelu k získaniu fitness riešenia.

Po vytvorení akcelerovanej verzie, boli obidve verzie riešiace MKP problém, tj. CPU aj GPU podrobené vzájomnému porovnávaniu. Porovnávané boli len plne optimalizované verzie CPU a GPU implementácie z toho dôvodu, aby sa plne prejavili vlastnosti daných architektúr. Z experimentov sa ukázalo, že akcelerovaná verzia zrýchľuje výpočet riešenia priemerne 5-násobne a maximálne 10-násobne, pričom zachováva kvalitu nájdeného riešenia. Prínos v podobe 10-násobného zrýchlenia výpočtu sa prejaví na dobe strávenej riešením problému o to viac, čím je tento problém zložitejší. V praxi to znamená, že CPU verzia na vyriešenie problému už nemusí stačiť, ak je toto riešenie nutné dodať v určitom časovom okne. Zrýchlenie získané akceleráciou má ale trend sa naďalej navyšovať, pretože výkon grafických kariet v dnešnej dobe rastie rýchlejšie ako výkon CPU. Akcelerovaná implementácia zrýchľuje výpočet, pričom zachováva kvalitu nájdeného riešenia, cieľ práce bol teda splnený.

Táto práca však skôr než nástroj na riešenie MKP problému s najvyššou možnou kvalitou, poskytuje odladenú platformu, na ktorej je možné ďalej stavať. Kvalitu riešenia MKP problému môže užívateľ zlepšiť vhodnejšou voľbou parametrov PSO prípadne úpravou algoritmu PSO špecificky na MKP problém.

Literatura

[1] Agner, F.: Optimizing software in C++. http://www.agner.org/optimize/optimizing_cpp.pdf, 2012-02-29.

[2] Aldaham, M.; Anderson, J.; Brown, S.; aj.: Low-Cost Hardware Profiling of Run-Time and Energy in FPGA Embedded Processor. 2011.

[3] ARB: OpenMP. http://openmp.org/wp/about-openmp/, [cit. 2013-01-03].

[4] Arizona, T. U. O.: Nvidia Graphics Processing Unit (GPU). http://www2.engr.arizona.edu/ yangsong/gpu.htm, [cit. 2013-01-03].

[5] Bansal, J. C.; Saraswat, P. K. S. M.; Verma, A.; aj.: Inertia Weight Strategies in Particle Swarm Optimization. http://www.softcomputing.net/nabic11_7.pdf, 2011.

[6] Chandra, R.; Dagum, L.; Kohr, D.; aj.: *Paralell Programming in OpenMP*. Academic Press, 2001, iSBN 1-55860-671-8.

[7] Diubin, G. N.; Korbut, A. A.: Greedy Algorithms for the Minimization Knapsack Problem: Average Behaviour. Journal of Computer and Systems Sciences International, New York, USA, vol. 47, 2008.

[8] Dorigo, M.; aj.: Particle Swarm Optimization. http://www.scholarpedia.org/article/Particle_swarm_optimization, 2008 [cit. 2012-12-26].

[9] Drake, J.: MKP - Problem Instances. http://www.cs.nott.ac.uk/j̃qd/mkp/#gk, [cit. 2013-05-06].

[10] Farber, R.: Running CUDA Code Natively on x86 Processors. http://www.drdobbs.com/parallel/running-cuda-code-natively-on-x86-proces/231500166, 2011.

[11] Friedrich-Alexander-Universität: Multi-Objective Evolutionary Algorithms. http://www12.informatik.uni-erlangen.de/research/evolivo/, [cit. 2013-05-09].

[12] Gandhi, M.: Particle Swarm Optimization (PSO) Sample Code using Java. http://gandhim.wordpress.com/2010/04/04/particle-swarm-optimization-pso-sample-code-using-java/, 2010-04-04 [cit. 2012-12-02].

[13] GNU: GCC. http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, [cit. 2013-01-03].

[14] GNU: gprof. http://sourceware.org/binutils/docs/gprof/index.html, [cit. 2013-01-03].

[15] Group, K.: OpenCL - The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/, [cit. 2013-01-07].

[16] Harris, M.: Optimizing Parallel Reduction in CUDA. http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf, [cit. 2013-05-02].

[17] Heitkötter, J.; aj.: MP-Testdata. http://www.zib.de/index.php?id=921&no_cache=1&L=0&cHash=fbd4ff9555f8714ac6238261e3963432&type=98, [cit. 2013-05-06].

[18] HeuristicLab: Particle Swarm Optimization. http://dev.heuristiclab.com/trac/hl/core/wiki/Particle%20Swarm%20Optimization, [cit. 2013-05-09].

[19] ICL Group: Performance Application Programming Interface. http://icl.cs.utk.edu/papi.

[20] Kennedy, J.; Eberhart, R.: *Particle Swarm Optimization*. IEEE, 1995, iSBN 0-7803-2768-3.

[21] Kennedy, J.; Eberhart, R. C.: A Discrete Binary Version of the Particle Swarm Algorithm. International Conference, Systems, Man and Cybernetics, Computational Cybernetics and Simulation, 1997.

[22] Khanesar, M. A.; Tavakoli, H.; Teshnehlab, M.; aj.: *Novel Binary Particle Swarm Optimization*. In-tech, 2009, iSBN 978-953-7619-48-0.

[23] Lee, V. W.; Kim, C.; Chhugani, J.; aj.: Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. http://pcl.intel-research.net/publications/isca319-lee.pdf, 2010.

[24] Lorie, J.; Savage, L.: Three problems in capital rationing. Journal of Business, Vol. 28, 1955.

[25] Luitjens, J.: Global Memory Usage and Strategy. http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_GlobalMemory.pdf, 2011.

[26] Martello, S.; Toth, P.: *KNAPSACK PROBLEMS Algorithms and Compouter Implementations*. John Wiley & Sons, 1990, iSBN 0 471 92429 2.

[27] Miranda, V.; Fonseca, N.: EPSO - Evolutionary Particle Swarm Optimization, a New Algorithm with Applications in Power Systems. Transmission and Distribution Conference and Exhibition 2002: Asia Pacific. IEEE/PES, Vol. 2, 2002.

[28] NVIDIA: What is GPU Computing? http://www.nvidia.com/object/what-is-gpu-computing.html, 2012 [cit. 2012-12-27].

[29] NVIDIA: CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2012 [cit. 2013-01-03].

[30] NVIDIA: WHAT IS CUDA? http://www.nvidia.com/object/cuda_home_new.html, [cit. 2013-01-07].

[31] Oken, A. L.: Penalty Functions and the Knapsack Problem. IEEE World Congress on Computational Intelligence, Proceedings of the First IEEE Conference on, Orlando, FL, USA, 1994.

[32] OpenACC: OpenACC Home. http://www.openacc-standard.org/, [cit. 2013-01-07].

[33] Petersen, C.: Computational experience with variants of the balas algorithm applied to the selection of r&d projects. Management Science, Vol. 13, 1967.

[34] Pirkul, H.: Heuristic Solution Procedure for the Multiconstraint Zero-One Knapsack Problem. Naval Research Logistics, 1987.

[35] Premalatha, K.; Natarajan, A. M.: *Hybrid PSO and GA for Global Maximization.* ICSRS, 2009, iSSN 1998-6262.

[36] Qingqing, Z.; Xingshi, H.; Na, S.: Convergence Analysis and Parameter Select on PSO. Information Science and Engineering (ISISE), 2009 Second International Symposium on, 2009.

[37] Rutkowski, L.; Tadeusiewicz, R.; Zadeh, L. A.; aj.: Artificial Intelligence And Soft Computing - Icaisc 2006. 8th International Conference, Zakopane, Poland, 2006.

[38] del Valle, Y.; Venayagamoorthy, G. K.; Mohagheghi, S.; aj.: *Particle Swarm Optimization: Basic Concepts, Variants and Applications in Power Systems.* IEEE, 2008, iSSN 1089-778X.

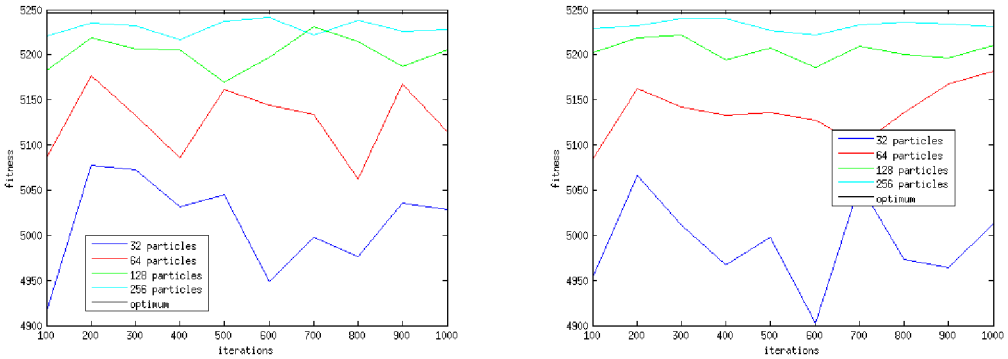
[39] Varbanescu, A. L.: Performance. http://www.st.ewi.tudelft.nl/~varbanescu/ASCI_A24.2k12/ASCI_A24_Day1_Part4_Performance.pdf, [cit. 2013-05-08].

[40] Wang, L.; Wang, X.; Fu, J.; aj.: A Novel Probability Binary Particle Swarm Optimization Algorithm and Its Application. Journal of Software, Vol. 3, No. 9, 2008.

[41] Zhang, W.-J.; Xie, X.-F.: DEPSO: Hybrid Particle Swarm with Differential Evolution Operator. Systems, Man and Cybernetics, 2003. IEEE International Conference on, Vol. 4, 2003.

A. Experimenty

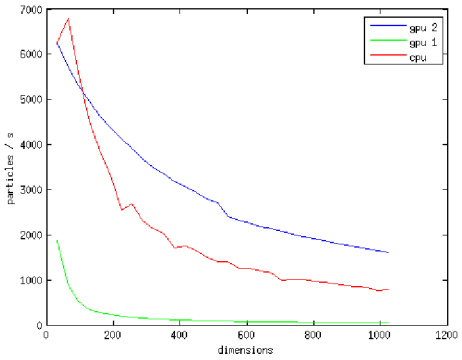
MKP: 2 batohy, 28 dimenzií
weish09.dat



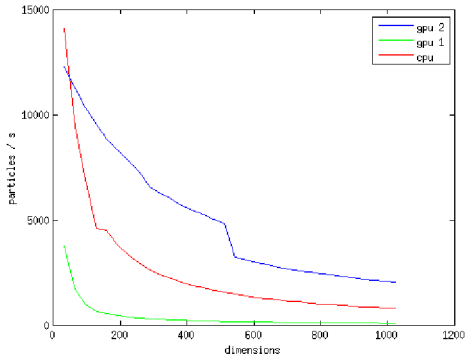
(a) GPU (b) CPU

Obrázok A.1: Vplyv počtu častíc na fitness riešenia

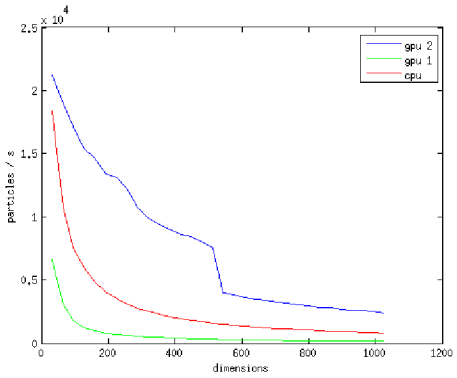
MKP: 1 batoh, 100 iterácií



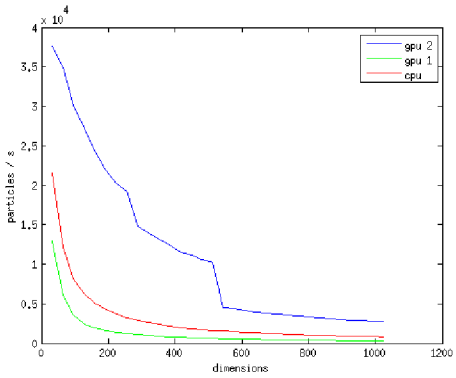
(a) 32 častíc



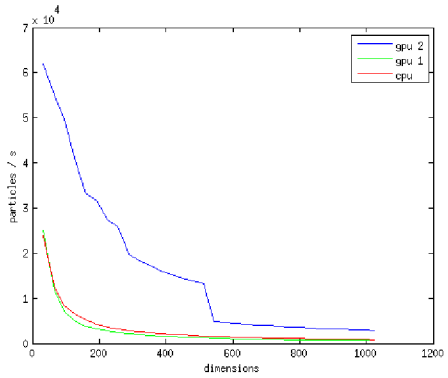
(b) 64 častíc



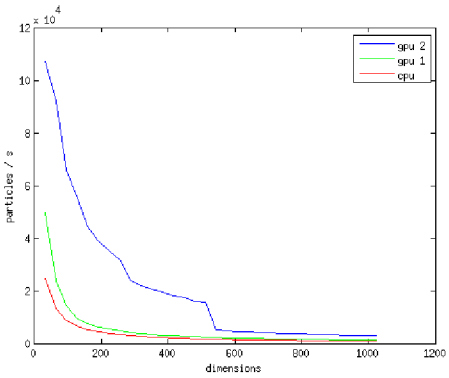
(c) 128 častíc



(d) 256 častíc



(e) 512 častíc



(f) 1024 častíc

Obrázok A.2: Porovnanie výkonu dvoch GPU verzií a CPU verzie

B. Ovládanie programu

```
POUŽITIE: mkp [VOĽBY] <SÚBOR>

VOĽBY:
    -m n  Maximálny počet iterácií
    -s n  Počet iterácií v ktorých nedôjde k zmene lídra roja
    -n n  Počet častíc
    -w r  Váha zotrvačnosti
    -1 r  Kognitívny koeficient
    -2 r  Sociálny koeficient

    -p r  Penalizačná konštanta (P)
    -v r  Absolutná hodnota maximálneho zrýchlenia častice

    -V    Výpis štatistík
    -h    Zobrazenie nápovedy

PRÍKLAD:
    $ mkp -n 256 -m 2000 -p 500 -v 10 weing1.dat
```