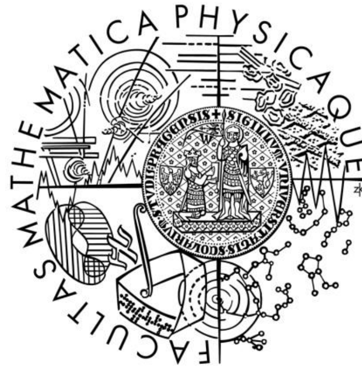
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Bc. Michal Hošala

Riešenie problému globálnej optimalizácie využitím GPU

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Martin Kruliš, Ph.D.

Studijní program: Informatika

Studijní obor: Softwarové systémy

Praha 2014

Na tomto mieste by som rád poďakoval všetkým ľuďom, ktorí mi boli psychickou oporou počas obdobia, kedy som tvoril túto prácu, najmä mojej priateľke a rodičom. Veľká vďaka patrí takisto vedúcemu mojej práce, RNDr. Martinovi Krulišovi, Ph.D., za množstvo času, ktoré bol ochotný obetovať na riešenie mojich problémov a za vecné pripomienky a rady pri konzultáciách aj mimo nich.

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 31. 7. 2014

Názov práce: Riešenie problému globálnej optimalizácie využitím GPU

Autor: Bc. Michal Hošala

Katedra: Katedra softwarového inženýrství

Vedúci bakalárskej práce: RNDr. Martin Kruliš, Ph.D.

Abstrakt: Problém globálnej optimalizácie, inými slovami problém hľadania globálnych extrémov funkcie v obmedzenom obore hodnôt, sa často objavuje v reálnych aplikáciách. Zvýšením účinnosti pri riešení tejto úlohy môže byť dosiahnuté zrýchlenie odozvy aplikácie, alebo poskytnutie presnejšieho výsledku, nakoľko sa úloha rieši pomocou aproximačných algoritmov. Táto práca je zameraná na praktické aspekty globálnej optimalizácie, najmä z oboru analýzy dát vo svete algoritmického obchodovania. Úspešné riešenia tejto úlohy za pomoci CPU sú už síce známe, ale ich hlavnou nevýhodou je veľká časová náročnosť. Hlavným cieľom tejto práce je preto navrhnúť riešenie problému globálnej optimalizácie za pomoci surovej výpočtovej sily GPU. Napriek neporovnateľne väčšiemu počtu výpočtových jadier, ktorými GPU oproti CPU disponuje, je však paralelizácia známych sériových algoritmov pomerne náročná, a to kvôli špecifikám GPU, ako sú napríklad výpočtový model, alebo architektúra pamäti. Druhotným cieľom tejto práce je preto preskúmať viacero možných prístupov k riešeniu úlohy globálnej optimalizácie a experimentálne porovnať dosiahnuté výsledky.

Kľúčové slová: globálna optimalizácia, extrémy funkcií, analýza dát, paralelizácia, GPU, CUDA

Title: Employing GPUs in Global Optimization Problems

Author: Bc. Michal Hošala

Department: Department of Software Engineering

Supervisor: RNDr. Martin Kruliš, Ph.D.

Abstract: The global optimization problem -- i.e., the problem of finding global extreme points of given function on restricted domain of values -- often appears in many real-world applications. Improving efficiency of this task can reduce the latency of the application or provide more precise result since the task is usually solved by an approximative algorithm. This thesis focuses on the practical aspects of global optimization algorithms, especially in the domain of algorithmic trading data analysis. Successful implementations of the global optimization solver already exist for CPUs, but they are quite time demanding. The main objective of this thesis is to design a GO solver that utilizes the raw computational power of the GPU devices. Despite the fact that the GPUs have significantly more computational cores than the CPUs, the parallelization of a known serial algorithm is often quite challenging due to the specific execution model and the memory architecture constraints of the existing GPU architectures. Therefore, the thesis will explore multiple approaches to the problem and present their experimental results.

Keywords: global optimization, extremes of function, data analysis, parallel, GPU, CUDA

Obsah

1. Úvod ... 1

1.1 Motivácia ... 2

1.2 Ciele ... 3

2. Globálna optimalizácia ... 4

2.1 Problém globálnej optimalizácie ... 4

2.2 Metódy globálnej optimalizácie ... 6

2.2.1 Metóda vetvenia a ohraničovania ... 7

2.2.2 Metóda simulovaného žíhania ... 8

2.3 Metóda časticovej optimalizácie ... 9

2.3.1 Algoritmus ... 10

2.3.2 Modifikácie algoritmu ... 12

2.4 Využitie metódy časticovej optimalizácie ... 15

2.4.1 Biomedicína ... 15

2.4.2 Siete ... 16

2.4.3 Financie ... 16

3. Architektúra grafickej karty ... 17

3.1 Princíp práce s GPU ... 17

3.2 Streamovací multiprocesor a jadro ... 19

3.3 Pamäť GPU ... 22

3.3.1 Registre ... 22

3.3.2 L1 cache a zdieľaná pamäť ... 23

3.3.3 Globálna pamäť a L2 cache ... 25

3.4 Programovanie GPU technológiou CUDA ... 26

3.4.1 Kompilácia kódu pre GPU ... 27

3.4.2 Získanie prístupu ku GPU ... 28

3.4.3 Práca s globálnou pamäťou GPU ... 28

3.4.4 Spúšťanie kódu na GPU 29
3.4.5 Komunikácia vlákien 31
4. Paralelizácia metódy časticovej optimalizácie technológiou CUDA 34
4.1 Súvisiace práce 34
4.2 Použité účelové funkcie 36
4.3 Testovací hardware 38
4.4 Implementácia 39
4.4.1 Aktualizácia lokálnych miním 41
4.4.2 Aktualizácia globálneho minima 42
4.4.3 Aktualizácia pozície a rýchlosti častíc 49
4.5 Vyhodnocovanie účelovej funkcie 51
4.5.1 Vyhodnocovanie častice vláknom 52
4.5.2 Paralelizácia vysoko dimenzionálnych funkcií 53
4.5.3 Paralelizácia funkcií inými parametrami 59
4.6 Všeobecné vylepšenia implementácie 63
5. Experimenty 65
5.1 Referenčná paralelná verzia pre CPU 65
5.2 Dosiahnuté výsledky 66
5.2.1 Funkcie f_1 až f_4 66
5.2.2 Funkcie f_6 až f_{10} 69
6. Záver 73
6.1 Budúce vylepšenia 74
Literatúra 76
Príloha A – Obsah priloženého disku DVD 81

1. Úvod

Už od počiatkov programovania sa vývojári stretávajú s problémom nedostatočnej rýchlosti nimi vytváraných aplikácií. Podstatou tohto problému je často len obmedzenie vyplývajúce z výpočtovej sily dostupného hardware. Jadrá dnešných vysoko výkonných procesorov (CPU) pracujú na hraniciach maximálnej dosiahnuteľnej frekvencie, a preto je v súčasnosti trendom týmto procesorom pridávať ďalšie jadrá, namiesto zvyšovania ich frekvencie. Tento trend má za následok okrem iného rýchly rozvoj odvetvia známeho ako paralelné programovanie, ktorého podstatou je vytvárať aplikácie a implementovať algoritmy tak, aby počas behu dokázali plne využiť výkon všetkých ponúkaných jadier.

S ohľadom na fakt, že v dnešnej dobe stále väčšina aplikácií pracuje len s využitím jedného jadra, sú dnešné CPU limitované prítomnosťou plnohodnotných a nezávislých jadier. Toto je silný prostriedok, pomocou ktorého sa dosahuje poskytovanie dostatočného výpočtového výkonu aj pre neparalelné aplikácie využívajúce len jedno jadro. Plnohodnotné jadrá ale dosahujú väčšie fyzické rozmery, produkujú viac tepla atď. a aj pre tieto dôvody sa ich počet v dnešnej dobe pohybuje aj pre špičkové CPU rádovo v desiatkach.

Obmedzený počet jadier procesorov a ich nezávislosť je hlavný bod, ktorý necháva v určitých prípadoch vyniknúť grafickým kartám (GPU). Tieto na rozdiel od CPU obsahujú skupiny jadier, ktoré vždy vykonávajú rovnaké inštrukcie, čo má za následok ich jednoduchšiu architektúru. To v konečnom dôsledku vedie k možnosti osadiť GPU vyšším počtom jadier, ktorých počet sa dnes pohybuje rádovo v tisícoch. Tento vysoký počet jadier spolu so zjednodušením vykonávania rovnakých inštrukcií v skupinách jadier súčasne je ideálnym prostriedkom na riešenie dátovo paralelných úloh.

Z vyššie uvedeného dôvodu sa GPU dlhú dobu používali na prácu s grafikou. Táto ich prirodzená úloha vyplývala práve z toho, že práca s grafikou typicky spočíva v aplikovaní rovnakých inštrukcií nad rôznymi dátami, jedná sa teda o dátovo

paralelné úlohy. Vo svete však existuje množstvo algoritmov, ktoré sú svojou povahou podobné práci s grafickými dátami, no bez podpory zo strany výrobcov hardware bolo často nemožné GPU na riešenie takýchto všeobecných problémov použiť. V súčasnosti však už existuje niekoľko zabehnutých technológii, ktoré priamo poskytujú prostriedky na spúšťanie všeobecných výpočtov na GPU, teda na prácu s negrafickými dátami. Súčasne tiež existuje silná podpora zo strany výrobcov smerom k vývojárom aplikácií pre GPU, počnúc kontinuálne sa zlepšujúcimi vývojovými prostriedkami a končiac kvalitnou dokumentáciou či sponzorovaním fór venovaných téme programovania GPU.

1.1 Motivácia

Zadanie tejto diplomovej práce vyplynulo na základe rozhovorov so zástupcami spoločnosti RSJ a.s.[1]. Hlavným predmetom činnosti tejto spoločnosti je algoritmické obchodovanie na burze bez nutnosti ľudského vstupu. Jedná sa teda o vydávanie automatizovaných príkazov na nákup a predaj akcií na základe predprogramovaného výpočtového modelu, ktorý je však v istých časových intervaloch nutné kalibrovať. Krok kalibrácie zaberá netriviálne množstvo času, a preto sa spoločnosť RSJ pokúšala rôznymi spôsobmi dosiahnuť jeho zrýchlenie. Použité verzie pre CPU však stále nedosahujú požadované výsledky.

Krok kalibrácie pozostáva z niekoľkých čiastkových úloh, medzi nimi aj z riešenia úlohy globálnej optimalizácie, teda hľadania globálnych extrémov určitej účelovej funkcie na vopred špecifikovanom intervale. Táto úloha je značne všeobecná, no v rôznych modifikovaných podobách nachádza široké uplatnenie aj v medicíne, robotike, metalurgii, či pri spracovaní obrazu a videa.

Účelové funkcie optimalizované spoločnosťou RSJ sú veľmi špecifické. Okrem vysokej výpočtovej náročnosti to spôsobuje fakt, že sú definované pomocou zozbieraných externých setov konfiguračných dát, ktoré ďalej určujú výpočet. Keďže proces kalibrácie má k dispozícii len obmedzené množstvo času, tak to pri jeho pomalej realizácii môže viesť k dosahovaniu menej presných výsledkov. Urýchlením

[1] http://www.rsj.com/

procesu by navyše bolo možné spracovávať väčší set externých dát, čo by rovnako skvalitňovalo výsledky a v konečnom dôsledku by to viedlo ku lepším rozhodnutiam pri obchodovaní na burze.

1.2 Ciele

Primárnym cieľom tejto práce je implementovať vybranú metódu časticovej optimalizácie pomocou GPU, ktorá sa používa práve na riešenie úlohy globálnej optimalizácie. Jedná sa o rovnakú metódu, ktorej CPU verziu na tento účel používa aj spoločnosť RSJ. Súčasťou práce je tiež zhrnutie teoretických poznatkov o súčasnej architektúre GPU a o metóde časticovej optimalizácie všeobecne, čo slúži ako teoretický základ pre efektívnu implementáciu riešenia problému použitím rôznych GPU architektúr.

Na záver je poskytnuté komplexné porovnanie implementovanej verzie pre GPU na rôznych zariadeniach a referenčnej verzie implementovanej pre viac jadrové CPU. Z výsledkov porovnania vychádza GPU verzia ako jasný víťaz, kedy dosahuje oproti verzii pre CPU zrýchlenie typicky o jeden rád a po obmedzení počtu aktívnych jadier CPU až o takmer dva rády.

2. Globálna optimalizácia

Táto kapitola je venovaná všeobecnému úvodu do problematiky globálnej optimalizácie, použitiu a stručnému popisu metód, ktoré sa v súčasnosti používajú na jej riešenie. Najväčší dôraz bude pritom kladený na metódu časticovej optimalizácie [1], ktorá je vďaka svojim vlastnostiam vhodným kandidátom na paralelizáciu.

2.1 Problém globálnej optimalizácie

Podľa Libertiho [2] je možné na problém globálnej optimalizácie nazerať ako na hľadanie bodu v množine prípustných riešení X, pre usporiadanú množinu T, v ktorom daná funkcia $f : X \to T$ nadobúda minimum alebo maximum. Táto práca je zameraná na spojitú globálnu optimalizáciu, čo zrejme platí aj pre funkciu f, pričom spojitosť optimalizovaných funkcií už v zvyšnom texte nebude špeciálne zdôrazňovaná. Formálnejší zápis problému pre prípad hľadania minima je preto možné prebrať od Pardalosa et al. [3]:

$$min\big(f(x)\big), x \in D \tag{2.1}$$

kde platí:

- $D = \{\, x: l \leq x \leq u;\ g_j(x) \leq 0\ j = 1, \dots, J \,\}$, l a u sú explicitné, konečné hranice a množina spojitých funkcií g tvorí ďalšie obmedzujúce podmienky pre vstup funkcie f
- $x \in \mathbb{R}^n$, vektor reálnych čísel dĺžky n
- $f : \mathbb{R}^n \to \mathbb{R}$ je spojitá účelová funkcia

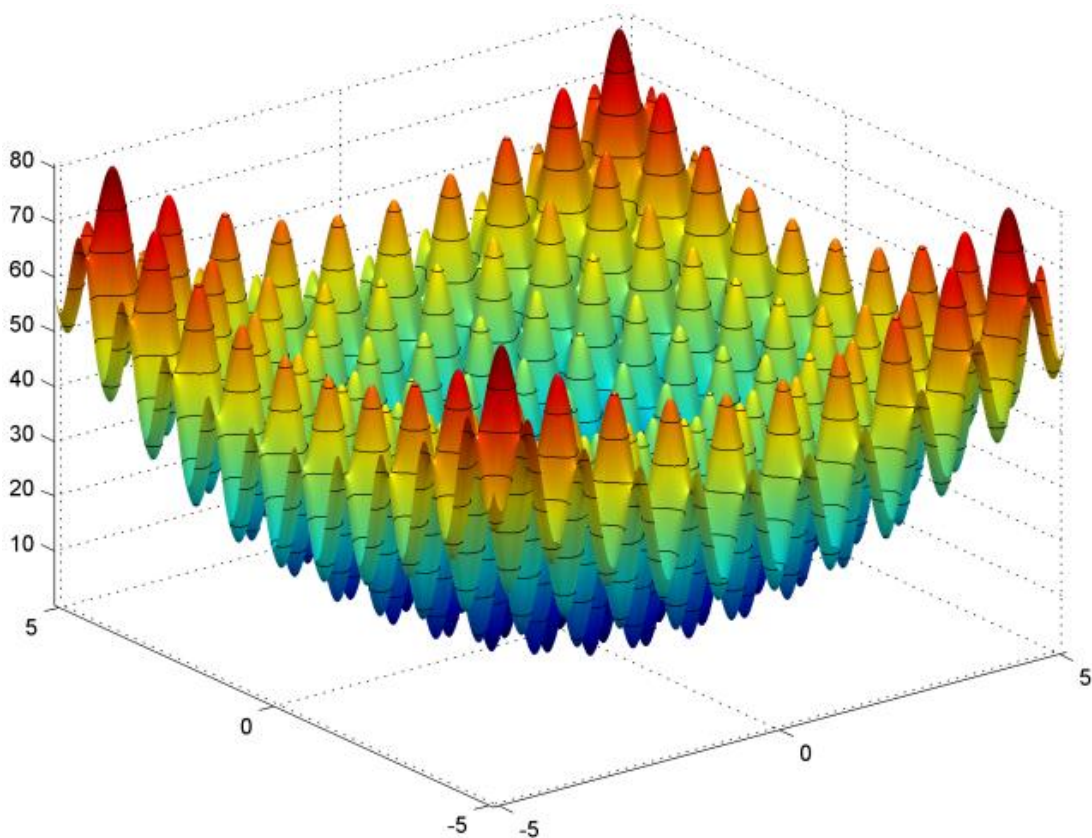
V prípade, ak je cieľom nájsť maximum, je možné použiť jednoduchú ekvivalenciu:

$$max\big(f(x)\big) \Leftrightarrow min\big(-f(x)\big), x \in D \tag{2.2}$$

Kvôli jednoduchosti prevoditeľnosti problému maximalizácie na minimalizáciu bude po zvyšok tohto textu globálna optimalizácia chápaná ako globálna minimalizácia a slovo optimum bude ekvivalentné slovu minimum.

Takáto formulácia problému, teda záruka spojitosti funkcie f a množiny D, v spojení s Weierstrassovým teorémom zaručuje, že optimálne riešenie vždy existuje [4]. Za predpokladu, že f je konvexná funkcia sa hľadanie optimálneho riešenia zjednoduší, pretože lokálne minimum funkcie je zároveň jej globálnym minimom a naopak. Na hľadanie lokálnych miním je potom možné použiť hneď niekoľko metód, napríklad lineárne programovanie, ktoré sú schopné určiť minimálne riešenie v konečnom čase. Tieto metódy sú v prípade nekonvexných funkcií, alebo funkcií s veľkým množstvom lokálnych miním prakticky nepoužiteľné, pretože akékoľvek nájdené lokálne minimum je automaticky považované aj za globálne minimum. Príkladom buď funkcia navrhnutá Rastriginom [5]:

$$f(x) = 10n + \sum_{d=1}^{n} [x_i^2 - 10\cos(2\pi x_i)]$$
$$x_i \in [-5.12, 5.12] \tag{2.3}$$



Obrázok 1: Rastriginova funkcia pre počet dimenzií 2, zdroj: cs.bham.ac.uk

Ako je dobre viditeľné z obrázku vyššie, Rastriginova funkcia obsahuje už pre počet dimenzií 2 na intervale $[-5.12, 5.12]$ veľké množstvo lokálnych extrémov, v ktorých by klasické metódy konvexnej optimalizácie jednoducho uviazli. Preto bolo k riešeniu problému globálnej optimalizácie nutné prísť s novými technikami, ktoré by sa s týmto problémom dokázali vyrovnať. Prehľad tých najzaujímavejších je uvedený v nasledujúcich podkapitolách.

2.2 Metódy globálnej optimalizácie

Problém globálnej optimalizácie pokrýva celé spektrum prípadov od špecifických až po všeobecné, ako napríklad riešenie Lipschitzovských optimalizačných problémov [6,7], alebo DC programovanie, ktorým sa zaoberá Horst et al. [8]. Preto sa dá očakávať, že efektívne metódy používané v konkrétnych prípadoch, riešiace len špecifické problémy, sa navzájom značne líšia a metódy, ktoré sa snažia riešiť čo najväčšiu podmnožinu problému doplácajú na nízku efektivitu [3]. Teoreticky je síce úlohou globálnej optimalizácie nájsť všetky riešenia problému formulovaného vzorcom (2.1), ale v praxi sa úloha formuluje ako hľadanie vhodnej aproximácie množiny optimálnych riešení. Táto praktická úloha je väčšinou riešená ako konečná postupnosť krokov, kedy je funkcia vyhodnocovaná v rôznych bodoch vstupného intervalu, pričom výber bodov riadi vopred zvolený algoritmus.

Aj kvôli vyššie uvedeným faktom existuje viacero spôsobov rozdelenia metód globálnej optimalizácie, napríklad na exaktné alebo heuristické, či deterministické a stochastické. Deterministické metódy síce zaručujú nájdenie optimálneho riešenia, no ich výpočtová náročnosť sa pre problémy vyšších dimenzií stáva privysokou aj vďaka tomu, že globálna optimalizácia spojitých funkcií je vo väčšine prípadov NP - ťažký problém [3]. Preto je aj nárast výpočtovej sily prakticky irelevantný a pre problémy vyššej dimenzionality je zvykom používať stochastické metódy.

Spoločným znakom stochastických optimalizačných metód je využívanie náhodných premenných, napríklad pri riadení stratégie prehľadávania vstupnej množiny prípustných riešení optimalizovanej funkcie. Aj napriek tomu, že pri ich použití nie je nájdenie optimálneho riešenia vôbec zaručené, tvoria tieto metódy dôležitú

skupinu, pretože za zlomok času výpočtu deterministickej metódy dokážu s určitou pravdepodobnosťou poskytnúť rozumne kvalitné riešenie. Jednotlivým metódam ako časticová optimalizácia [1], simulované žíhanie [9], ale aj stochastickej optimalizácii všeobecne je venovaných mnoho odborných publikácií [3,10], pre úplnosť sú preto niektoré z metód v nasledujúcich podkapitolách bližšie vysvetlené. Jadrom tejto práce je metóda časticovej optimalizácie, preto je podrobne rozobraná v samostatnej kapitole 2.3.

2.2.1 Metóda vetvenia a ohraničovania

Jedná sa o presnú metódu známu už od roku 1960, kedy ju navrhol Land et al. [11]. Bola vytvorená pre riešenie problémov diskrétneho programovania, no dnes je jej využitie oveľa širšie a zahŕňa aj spojitú globálnu optimalizáciu, teda riešenie problému formulovaného vzorcom (2.1). Metóda pozostáva z niekoľkých krokov:

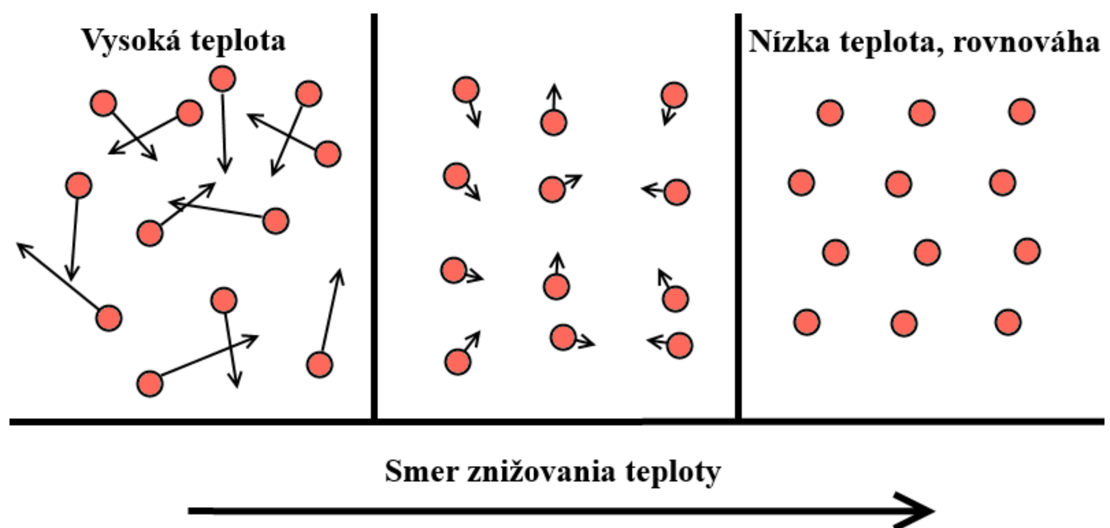
1. Rozdelí sa interval D vstupných kandidátov optimalizovanej funkcie na podintervaly $D_1, \dots, D_n$, kde $D = \bigcup_{i=1}^{n} D_i$.
2. Spočítajú sa horné a dolné hranice pre každé D_i, v závislosti od hodnôt, ktoré funkcia na tom ktorom podintervale dosahuje. Výpočet hraníc je netriviálnym krokom algoritmu, ktorým sa zaoberá aj Liberti [2], konkrétne sa snaží o výpočet čo najtesnejších hraníc.
3. Pretože minimom na D je najmenšie z miním dosiahnutých na podintervaloch, tak automaticky môžeme z ďalšieho vyhodnocovania vylúčiť tie podintervaly, ktorých dolná hranica je vyššia ako horná hranica ľubovoľného už ohraničeného podintervalu.
4. Proces sa rekurzívne vykonáva na podintervaloch, pričom na zastavenie sa používajú rôzne podmienky, napríklad rovnosť spočítanej dolnej a hornej hranice.

Metóda je okrem širokej uplatniteľnosti (DC programovanie, Lipschitzovská optimalizácia) zaujímavá aj tým, že jednak existujú jej deterministické verzie, ktorým sa venuje Scholz [12], ako aj stochastické verzie popísané Norkinom et al. [13]. Náhodným komponentom v prípade tejto metódy môže byť napríklad, nahradenie výpočtu dolnej a hornej hranice pre hodnoty, ktoré optimalizovaná

funkcia na podintervale dosahuje, ich odhadmi. Konvergovanie výpočtu k optimálnej hodnote aj v takomto prípade dokazuje Norkin et al. [13].

2.2.2 Metóda simulovaného žíhania

Simulované žíhanie patrí do skupiny heuristických, stochastických metód, ktoré sú inšpirované procesmi skutočného sveta, v tomto prípade procesom žíhania z oboru metalurgie. Analógiou s týmto procesom je používanie faktoru nazvaného teplota, ktorý riadi prehľadávanie vstupného intervalu problému formulovaného (2.1). Na začiatku algoritmu sa teplota nastaví na vysokú hodnotu, ktorá značí, že sa prehľadávajú veľké úseky vstupného intervalu. Obdobne je to aj v metalurgii, kedy sa molekuly v kove pri vysokej teplote pohybujú rýchlejšie. Algoritmus prebieha iteratívne, pričom teplota v každej iterácii klesá, takže sa prehľadávajú stále menšie časti vstupu, až sa proces pohybu po vstupe zastaví, rovnako ako sa zastaví pohyb molekúl kovu s jeho postupným chladnutím na optimálnych pozíciách, tak, aby bolo napätie materiálu minimálne. Názorná ukážka je na obrázku nižšie.



Obrázok 2: Chladnutie kovu z pohľadu vnútorného pohybu molekúl

Ako už bolo spomenuté, algoritmus prebieha v iteráciách, pričom podmienkou zastavenia je buď dosiahnutie dostatočne dobrého riešenia, alebo dosiahnutie maximálneho počtu iterácií. Oba tieto parametre sú zadané na vstupe a závislé od konkrétnej inštancie riešeného problému. Inicializácia výpočtu spočíva vo zvolení náhodného bodu vstupu označovaného x. Nasleduje iteratívny výpočet:

1. Zvolí sa susedný bod bodu x, značí sa x', na čo slúži procedúra závislá od konkrétnej implementácie. Nech $f(x) = xval$.
2. Spočíta sa teplota T v konkrétnej iterácii algoritmu, optimálnym rozvrhom znižovania teploty sa zaoberá napríklad Hajek [14].
3. Vyhodnotí sa funkcia f v bode x', $f(x') = xval'$
4. Na základe pravdepodobnostnej funkcie $P(xval, xval', T)$ sa buď proces presunie do bodu x', alebo zostáva v bode x, vlastnosti tejto funkcie sú bližšie popísané v nasledujúcom odseku.

Po ukončení sa algoritmus nachádza v bode, ktorý považuje za globálne optimum, ale keďže sa jedná o stochastickú metódu, optimalita riešenia nie je zaručená.

Pre pravdepodobnostnú funkciu, ktorá sa používa pri zamietaní alebo akceptovaní nových kandidátov na minimálne riešenie väčšinou platí niekoľko pravidiel:

- Pre $xval' > xval$ sa s klesajúcou teplotou T pravdepodobnosť blíži k nule, pričom $T = 0$ implikuje $P(xval, xval', T) = 0$.
- Pre $xval' < xval$ je výsledná pravdepodobnosť rovná 1, takže kandidát dosahujúci nižšiu hodnotu je vždy akceptovaný, aj v prípade $T = 0$.
- Typicky s narastajúcou hodnotou $xval' - xval$ klesá pravdepodobnosť presunu do bodu x'.

Na základe vlastností funkcie P je odvoditeľné, že s jej pomocou je možné prekonať hlavnú prekážku pri globálnej optimalizácii – uviaznutie v lokálnom extréme. Pokiaľ teplota neklesne až na nulu, tak je stále možné, že sa proces presunie do susedného bodu x' aj napriek tomu, že $xval' > xval$.

2.3 Metóda časticovej optimalizácie

Metóda časticovej optimalizácie je rovnako heuristikou, ktorá sa používa na prehľadávanie vstupného intervalu problému globálnej optimalizácie, a tak, ako metóda simulovaného žíhania, má aj táto metóda základ v reálnom svete. Konkrétne je založená na pozorovaniach správania sa kŕdľa vtákov, zhlukov rýb a podobne. V kontexte časticovej optimalizácie sú členovia týchto zvieracích spoločenstiev

považovaní za častice, čo sú body rozmiestnené po vstupnom intervale. Hľadanie minima je analogické napríklad vyhýbaniu sa spoločenstva predátorovi, alebo hľadaniu potravy, pretože ako sa pohybujú členovia spoločenstva po priestore, tak sa aj častice iteratívne pohybujú po intervale a v každej iterácii je vyhodnocovaná optimalizovaná funkcia za účelom identifikácie častice, ktorá dosahuje najnižšiu hodnotu.

Hlavným princípom časticovej optimalizácie prebratým zo zvieracej ríše je to, že tak, ako si členovia spoločenstva navzájom predávajú informácie, si aj častice pri optimalizácií udržujú povedomie o ostatných časticiach a aj na základe týchto informácií určujú smer, v ktorom sa vydajú v ďalšom kroku prehľadávania vstupného intervalu. Zatiaľ, čo interakcie v zvieracej ríši sú pomerne zložité [15], pri bežnej implementácii metódy časticovej optimalizácie si častice udržujú len informáciu o najnižších hodnotách, ktoré v doterajšom prehľadávaní vstupného intervalu dosiahli ostatné častice.

2.3.1 Algoritmus

Algoritmus vo svojej základnej verzii je pomerne jednoduchý, vstupom je popis optimalizovanej funkcie $f\colon \mathbb{R}^n \to \mathbb{R}$, vstupný interval $D \in \mathbb{R}^n$ a počet častíc, ktoré sa pri prehľadávaní intervalu použijú, označme m. Podľa Poliho et al. [16] sú bežnými hodnotami pre m hodnoty z intervalu $[20, 50]$. Ďalej definujeme:

- $x_i \in \mathbb{R}^n, i = \{1, \dots, m\}$ je aktuálna pozícia i-tej častice,
- $y_i = f(x_i)$ je aktuálna hodnota účelovej funkcie i-tej častice,
- $v_i \in \mathbb{R}^n, i = \{1, \dots, m\}$ je aktuálna vektorová rýchlosť i-tej častice,
- $p_i \in \mathbb{R}^n, i = \{1, \dots, m\}$ je pozícia, na ktorej i-tá častica dosiahla svoje doterajšie minimum,
- $pbest_i = f(p_i),\ i = \{1, \dots, m\}$ je najnižšia hodnota účelovej funkcie, ktorú dosiahla i-tá častica,
- $g \in \{p_1, \dots, p_m\}, f(g) = min(f(p_1), \dots, f(p_m))$, je pozícia, na ktorej ľubovoľná z častíc dosiahla najnižšiu hodnotu účelovej funkcie,
- $gbest = f(g)$ je najnižšia nájdená hodnota

Samotný algoritmus pozostáva z inicializácie a z iteratívneho prehľadávania vstupného intervalu, až kým nie je nájdené dostatočne dobré riešenie, alebo kým nie je dosiahnutý maximálny počet iterácií, rovnako, ako v prípade metódy simulovaného žíhania. Inicializácia pozostáva z týchto krokov:

1. Uniformným vzorkovaním intervalu D je získaných m pozícií, ktoré sa použijú na inicializovanie hodnôt x_i, p_i a $pbest_i$.
2. Interval D musí mať dolné aj horné hranice v každej dimenzii $d \leq n$, označme lo_d, hi_d, rozdiel označme $\Delta_d = abs(lo_d - hi_d)$. Rýchlosť každej častice je potom inicializovaná ako $[rand(-\Delta_1, \Delta_1), \ldots, rand(-\Delta_n, \Delta_n)]$, kde $rand$ je generátor náhodných čísel zo zvoleného intervalu.

Po inicializácii prebieha prehľadávanie v nasledovných krokoch až kým nie sú splnené podmienky ukončenia, kroky sú vykonávané pre všetky častice:

1. Vygenerujú sa dve náhodné čísla $r1, r2$ z intervalu $[0, 1]$.
2. Nová vektorová rýchlosť i-tej častice sa spočíta nasledovným vzorcom:

$$v_i = \omega v_i + \varphi_p r_1(p_i - x_i) + \varphi_g r_2(g - x_i) \qquad (2.4)$$

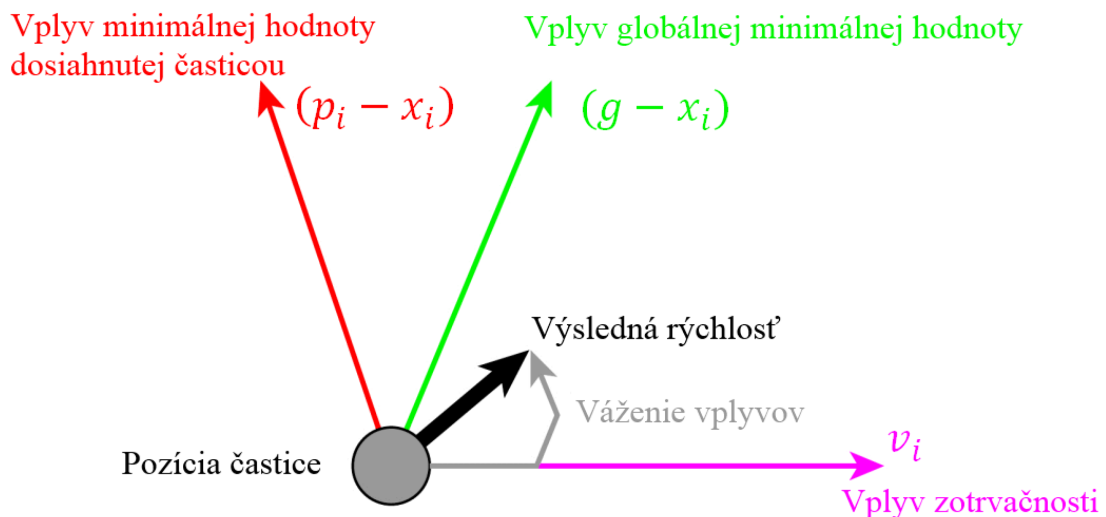
   Premenné $\omega, \varphi_p, \varphi_g$ predstavujú váhy, ktorých význam je vysvetlený nižšie.
3. Spočíta sa nová pozícia i-tej častice:
$$x_i = x_i + v_i$$
4. Aktualizuje sa hodnota účelovej funkcie:
$$y_i = f(x_i)$$
5. Prebehne aktualizácia lokálnych miním:
$$(y_i < pbest_i) \Rightarrow pbest_i = y_i, p_i = x_i$$
6. Prebehne aktualizácia globálneho minima:
$$(y_i < gbest) \Rightarrow gbest = y_i, g = x_i$$

Po splnení podmienok ukončenia obsahuje premenná $gbest$ minimálnu dosiahnutú hodnotu a premenná g obsahuje súradnice, na ktorých bola táto minimálna hodnota dosiahnutá.

Jediným netriviálnym krokom vyššie popísaného algoritmu je výpočet vektorovej rýchlosti. Nová rýchlosť vzniká zložením troch základných častí, ktorých význam je vysvetlený na obrázku nižšie. Veľmi dôležitými sú ale aj váhy, ktoré prislúchajú týmto trom zložkám:

- $\omega, \varphi_p, \varphi_g$ sú závislé od konkrétnej implementácie metódy časticovej optimalizácie a ich vhodným hodnotám sa venuje napríklad Eberhart et al. [17] a prehľadné zhrnutie vypracoval aj Poli et al. [16].
- r_1, r_2 tvoria stochastickú komponentu tejto metódy, vďaka čomu je možné preskúmať stále inú časť vstupného intervalu pre rôzne behy algoritmu.

Na tomto mieste je ešte vhodné podotknúť, že pôvodná verzia algoritmu časticovej optimalizácie od Kennedyho vôbec nepoužívala váhu pre zotrvačnosť, ω, tá bola uvedená až Shiim et al. [18]. Jej používanie sa ale stalo takmer štandardom, a preto je uvedená ako súčasť základnej verzie algoritmu.



Obrázok 3: Znázornenie vplyvov pri výpočte vektorovej rýchlosti častice

2.3.2 Modifikácie algoritmu

Kvôli veľkej obľúbenosti metódy časticovej optimalizácie, vďaka jej rýchlosti, schopnosti pokryť veľký vstupný interval atď. sa tejto metóde a jej možným vylepšeniam venuje veľký počet odborných publikácií. Pár známych modifikácií základnej verzie algoritmu je popísaných ďalej v tejto kapitole. Užitočné je

mnohokrát aj spájanie časticovej optimalizácie s inými metódami, ako je napríklad optimalizácia pomocou hľadania vzorov. Kombinácii týchto dvoch konkrétnych prístupov sa venuje napríklad Vaz et al. [19], kde je podrobne rozobratý aj vplyv na rýchlosť výpočtu oproti štandardnej verzii.

LBEST model

LBEST model bol jednou z prvých modifikácií pôvodnej časticovej optimalizácie, s ktorou v roku 1995 prišiel Eberhart et al. [20]. Jedná sa o modifikáciu topológie častíc, nakoľko v pôvodnej verzii bol pohyb častice počítaný na základe jej najlepšieho dosiahnutého výsledku a celkového najlepšieho výsledku zo všetkých častíc. V LBEST modeli je ale častica ovplyvňovaná len svojím výsledkom a výsledkami častíc z jej okolia. Definícia okolia závisí od vstupného parametru K, $K \in \mathbb{N} \wedge K \neq 0 \wedge K \, mod \, 2 = 0$, typicky $K = 2$ a teda pohyb i-tej častice závisí od častíc $i - \left(\frac{K}{2}\right), \ldots, i + \left(\frac{K}{2}\right)$. Samotný algoritmus zostáva takmer rovnaký, jedine definícia pre člen g vo vzorci (2.4) sa zmení na:

$$g \in \left\{ p_j \mid i - \left(\frac{K}{2}\right) \leq j \leq i + \left(\frac{K}{2}\right) \right\}$$
$$f(g) = min\left(f(p_j) \mid i - \left(\frac{K}{2}\right) \leq j \leq i + \left(\frac{K}{2}\right) \right) \quad (2.5)$$

Výhodou LBEST modelu voči pôvodnej verzii je odolnosť voči uviaznutiu v lokálnom minime, nakoľko sa pri behu algoritmu zvyknú vytvárať nezávislé skupiny častíc, ktoré konvergujú k výsledku v rôznych častiach vstupného intervalu [20]. Nevýhodou je vyšší počet iterácií potrebných na dosiahnutie rovnakého výsledku.

Trecie koeficienty

Použitie trecích koeficientov navrhuje Clerc et al. [21] a ich úlohou bolo zaručiť konvergovanie častíc, či tiež mapovanie ich rýchlosti do rozumných medzí, bez potreby explicitného orezávania na vopred špecifikované maximálne hodnoty. Vďaka vykonanej analýze bolo takisto možné nahradiť krok „hádania“ hodnôt pre parametre φ_p, φ_g zo vzorca (2.4) a nahradiť ich univerzálnymi hodnotami. Jeden zo

spôsobov implementácie trenia pre metódu časticovej optimalizácie spočíva v modifikácii vzorca (2.4) na tvar:

$$v_i = \chi\big(v_i + \varphi_p r_1 (p_i - x_i) + \varphi_g r_2 (g - x_i)\big) \tag{2.6}$$

Kde platí $\varphi = \varphi_p + \varphi_g > 4$ a zároveň:

$$\chi = \frac{2}{\varphi - 2 + \sqrt{\varphi^2 - 4\varphi}} \tag{2.7}$$

Pri použití uvedenej metódy trenia sa typicky nastavuje $\varphi = 4.1$ a $\varphi_p = \varphi_g$, čo vedie k $\chi \approx 0.7298$. Je jednoduché všimnúť si, že vzorce (2.4) a (2.6) sú algebraicky ekvivalentné, čo v tomto prípade vedie pre vzorec (2.4) ku nasledovným váham jednotlivých zložiek rýchlosti:

- $\omega = 1.49618$,
- $\varphi_p = \varphi_g = 0.7298$

Plne informované častice

Rovnako ako LBEST model, aj táto modifikácia prichádza so zmenou výpočtu ďalšieho pohybu častíc v každej iterácii algoritmu. S touto modifikáciou prišiel Kennedy et al. [22], pričom na vektor rýchlosti častice majú vplyv všetky častice z jej okolia a nielen tá, ktorá dosiahla najlepší doterajší výsledok. Ak okolie i-tej častice obsahuje K_i častíc, tak vektor rýchlosti vzniká zložením $K_i + 1$ vektorov, čo predstavuje nasledovný vzorec:

$$v_i = \chi\left(v_i + \frac{1}{K_i}\sum_{j=1}^{K_i} \varphi_p r_1 \left(p_{nbr_j} - x_i\right)\right) \tag{2.8}$$

Význam členov je rovnaký, ako v pôvodnej verzii, p_{nbr_j} označuje najlepšiu pozíciu dosiahnutú j-tou susednou časticou i-tej častice, χ je trecí koeficient. Porovnaním so vzorcom (2.4) je možné odvodiť, že pre $K_i > 2$ vzrastá výpočtová náročnosť oproti pôvodnej verzii. Naproti tomu táto modifikácia prináša zlepšenie v tom, že

minimálne riešenie nájde väčšinou v kratšom čase a stačí jej na to menší počet iterácií metódy časticovej optimalizácie, pričom je pevne spätá s konkrétnou topológiou populácie častíc, pretože táto určuje hodnotu parametru K_i. Vplyvom rôznych topológií na výpočet sa veľmi obšírne venuje Mendes [23].

2.4 Využitie metódy časticovej optimalizácie

Metóda časticovej optimalizácie má nepreberné množstvo využití, ktoré sú popísané v mnohých vedeckých prácach, pričom ich prehľad a kategorizáciu pomerne dôkladne vypracoval Poli [24]. Jeho práca je postavená na analyzovaní viac než 700 odborných publikácií, ktoré sa nachádzajú v IEEE[1] databáze, pričom časticová optimalizácia je používaná na riešenie širokého spektra problémov. Mnohé z problémov využívajú modifikované verzie časticovej optimalizácie, základná idea, teda hľadanie optimálneho riešenia použitím časticovej optimalizácie, je ale vždy rovnaká. Niektoré zo zaujímavých kategórií riešených problémov sú priblížené vo zvyšku tejto kapitoly. Využitie ale nie je limitované len spomenutými kategóriami, metóda nájde okrem iného podľa Poliho uplatnenie aj v metalurgii, robotike, predpovedaní, plánovaní ale napríklad aj pri navrhovaní antén, či rozpoznávaní tvárí na obrázkoch a vo videách.

2.4.1 Biomedicína

Využitie časticovej optimalizácie napríklad pri diagnostike Parkinsonovej choroby navrhol Eberhart et al. [25] už v roku 1999 a spočíva v analýze ľudského chvenia pomocou neurónovej siete, ktorá pri svojom vývoji využíva práve časticovú optimalizáciu.

Nemenej zaujímavou aplikáciou je klasifikácia rakoviny, ktorá zohráva kľúčovú úlohu pri samotnej diagnostike a následnej liečbe tejto choroby. Využiteľnosť metódy časticovej optimalizácie spočíva podľa Xua et al. [26] v tom, že sa pomocou nej identifikujú gény relevantné pre istý druh rakoviny, čo následne zvyšuje presnosť navrhovaného klasifikátora rakoviny.

[1] http://www.ieee.org/

2.4.2 Siete

Časticová optimalizácia nachádza uplatnenie pri riešení takých problémov, ako je ideálne umiestnenie základných staníc pre mobilné zariadenia rôznych druhov, a to jednak s ohľadom na vysoké pokrytie ale aj účinnosť. Tejto problematike sa venuje medzi inými aj Zhang et al. [27].

Iným problémom z oblasti sietí je efektívne využívanie frekvenčného rozsahu mobilnými zariadeniami, teda prideľovanie komunikačných kanálov. Zhang et al. [28] navrhuje za týmto účelom používať časticovú optimalizáciu a zároveň tvrdí, že prístup dosahuje dobrých výsledkov, čo do minimalizácie využívaného frekvenčného spektra, aj do naplnenia komunikačných požiadaviek. Zároveň pripomína, že rozšíreným prístupom na riešenie tohto problému je aj použitie metódy simulovaného žíhania, ktorá je spomenutá aj v tejto práci v kapitole 2.2.2.

2.4.3 Financie

Podľa Poliho výskumu [24] tvoria odborné publikácie z oboru financií využívajúce časticovú optimalizáciu len 1% všetkých ním preskúmaných publikácií. Aj napriek tomu je táto kategória obzvlášť dôležitá z pohľadu tejto práce, nakoľko cieľom je paralelizovať výpočet časticovej optimalizácie práve za účelom uplatnenia na finančných trhoch.

Konkrétnu úlohu, ktorú zohráva časticová optimalizácia popisuje napríklad Nenortaite et al. [29]. Riešeným problémom v tomto prípade je vytvorenie inteligentného modelu postaveného na neurónových sieťach schopného činiť rozhodnutia ohľadom možného investovania na burze. Tento model je nutné v istých časových úsekoch trénovať, čo spočíva v prispôsobovaní setu neurónových sietí pomocou optimálnych váh nájdených práve metódou časticovej optimalizácie.

3. Architektúra grafickej karty

Za účelom paralelizácie akéhokoľvek algoritmu pomocou grafickej karty (GPU), časticovú optimalizáciu popísanú v kapitole 2.3 nevynímajúc, je nutné hlbšie pochopenie základných princípov GPU, architektúry a prostriedkov, ktoré ponúka. Bez týchto znalostí, respektíve obmedzení, ktoré z nich vyplývajú, je nemožné vytvárať efektívne aplikácie pre GPU, a preto je im venovaná táto kapitola. Terminológia aj architektúra samotných GPU sa líši v závislosti od konkrétneho výrobcu, pričom text tejto kapitoly bude založený na terminológii a architektúre, ktorú používa spoločnosť NVIDIA. Na to existujú dva hlavné dôvody:

- Experimenty, ktoré sú súčasťou tejto práce prebiehajú na zariadeniach od spoločnosti NVIDIA a boli navrhované s ohľadom na tento fakt. Všetky použité zariadenia sú postavené buď na architektúre Kepler [30], alebo staršej Fermi [31] a bude na ne po zvyšok tejto kapitoly braný osobitný zreteľ.
- Aj napriek tomu, že existuje niekoľko dostupných technológii na vytváranie paralelných aplikácií pre GPU, ako OpenCL [32], alebo OpenACC [33], tak voľba v rámci tejto práce padla na technológiu CUDA, ktorú vytvorila priamo spoločnosť NVIDIA a jej základom sa venuje podkapitola 3.4. Cieľom práce je totiž dosiahnuť čo najväčšie zrýchlenie metódy časticovej optimalizácie pomocou paralelizmu na GPU a existuje logický predpoklad, že technológia navrhnutá konkrétnym výrobcom kariet dokáže najlepšie využiť prostriedky, ktoré ich vlastné GPU poskytujú. Okrem iného to je možné aj vďaka špeciálnym inštrukciám, ktoré poskytuje CUDA, no nenájdeme ich v iných technológiách, ako OpenCL, ktorá dokáže pracovať s GPU rôznych výrobcov, na ktorých jednoducho pre tieto inštrukcie neexistuje rovnaká hardwarová podpora ako na GPU spoločnosti NVIDIA.

3.1 Princíp práce s GPU

Jedným z hlavných princípov práce s GPU je komunikácia CPU s GPU, ktorá je nevyhnutná, nakoľko GPU nie je samostatne funkčná jednotka, nebeží na nej operačný systém a nie je schopná vykonávať žiadnu prácu bez toho, aby jej ju nepridelil CPU. GPU má svoju vlastnú pamäť, ktorú využíva počas svojich

výpočtov, a ktorá sa na nej fyzicky nachádza. CPU k tejto pamäti nemá priamy prístup a môže s ňou pracovať len cez presne definované rozhranie inštrukcií GPU, ktoré slúžia na kopírovanie pamäte CPU do pamäte GPU a naopak. Typický beh programu s využitím GPU preto vyzerá nasledovne:

1. CPU overí, či má k dispozícii vhodnú GPU, ak nie, tak výpočet končí.
2. CPU zavolá špeciálne funkcie na GPU, ktorými alokuje množstvo pamäte potrebnej pre výpočet, a následne do tejto pamäte skopíruje dáta výpočtu, ktoré má k dispozícii v operačnej pamäti.
3. CPU vyšle požiadavku na spustenie riešenej úlohy na GPU, pričom pri spustení určí aj to, v koľkých vláknach súčasne sa má úloha vykonať.
4. Hneď ako GPU obdrží požiadavku na vykonanie úlohy, tak pomocou svojho rozvrhovača začne prideľovať vlákna svojim fyzickým jadrám a nechá ich operovať nad dátami, ktoré CPU poslalo ako parametre danej úlohy.
5. CPU môže následne čakať na dokončenie úlohy spustenej na GPU, alebo vykonávať inú prácu, prípadne prideliť GPU ďalšiu úlohu.
6. CPU po ukončení úlohy skopíruje výsledné dáta z pamäti GPU do operačnej pamäti za účelom ich ďalšej interpretácie.

Z vyššie uvedeného vyplýva, že GPU obsahuje veľké množstvo jadier, na ktorých sa paralelne vykonávajú vlákna, čo tvorí základ paralelizmu GPU, ktorý má oproti paralelným vláknam CPU svoje špecifiká a obmedzenia. GPU využíva takzvaný model SIMT[1], tu sú jeho základné vlastnosti a porovnanie s inými modelmi, ktoré ilustruje tiež Obrázok 4:

- V tradičnom ponímaní využívajú CPU model SISD[2], keď jeden procesor spracúva jeden stream inštrukcií nad jednou pamäťou.
- Alternatívou k tradičnému modelu SISD je model SIMD[3], ktorý dokáže aplikovať jednu inštrukciu na viac ako jeden prvok súčasne, čím sa v ideálnom prípade skráti čas výpočtu na zlomok, podľa počtu prvkov, na ktoré bola inštrukcia aplikovaná. O vykonanie inštrukcie sa stále stará jedno

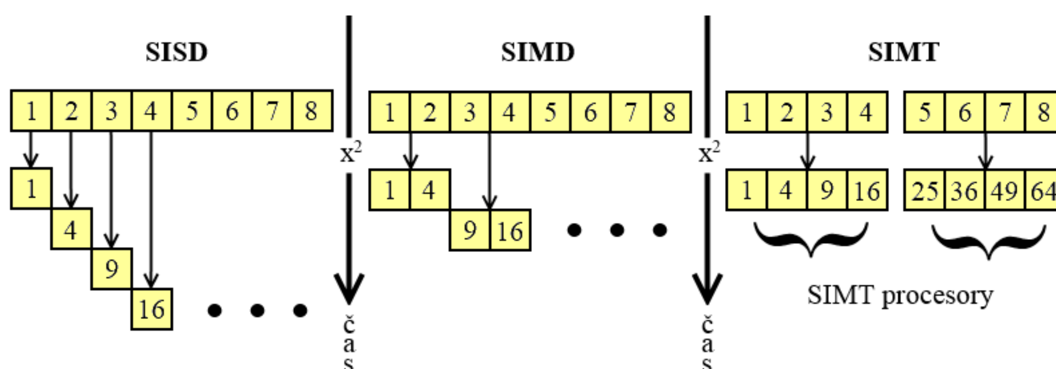
[1] Skratka z anglického Single Instruction Multiple Threads

[2] Skratka z anglického Single Instruction Single Data

[3] Skratka z anglického Single Instruction Multiple Data

jadro CPU, respektíve vlákno, ktoré na jadre beží. Jedná sa o paralelizmus na úrovni dát dosahovaný volaním špeciálnych inštrukcií v danom vlákne.

- GPU sú navrhnuté a postavené za účelom vykonania rovnakej inštrukcie na veľkom množstve prvkov súčasne, čo sa na rozdiel od modelu SIMD dosahuje na úrovni vlákien. Kód pre GPU vyzerá ako typický sériový kód, s tým rozdielom, že sa vykonáva súčasne vo veľkom množstve vlákien.
- Vlákna, respektíve jadrá GPU, môžu byť rozdelené do skupín, ktoré spravuje a riadi takzvaný SIMT procesor, ktorý sa v terminológii NVIDIA nazýva streamovací multiprocesor a pojednáva o ňom nasledujúca podkapitola. Tento procesor riadi výpočet vo všetkých jeho jadrách súčasne, čo v konečnom dôsledku vedie k jednoduchšiemu hardware, ako v prípade multi jadrových CPU, a teda aj k možnosti umiestniť na GPU väčší počet jadier, ako majú dnešné moderné CPU a dosiahnuť vyššiu úroveň paralelizácie.



Obrázok 4: Ilustrácia spracovania inštrukcie mocniny na ôsmich prvkoch rôznymi modelmi

3.2 Streamovací multiprocesor a jadro

Streamovací multiprocesor (SM), je základnou stavebnou jednotkou GPU, pričom zhruba odpovedá vysoko paralelnému CPU jadru [34]. Oproti CPU jadru má viacero výhod v tom, že na GPU nebeží operačný systém, pamäť je prideľovaná priamo bez stránkovania atď., takže GPU postačuje menší set inštrukcií, čo v konečnom dôsledku vedie k jednoduchšiemu hardware.

SM pracuje na báze inštrukcií vykonávaných súčasne v skupine vlákien, ktorých počet závisí od konkrétnej architektúry, pričom táto skupina vlákien sa nazýva warp. Práca je jednotlivým SM prideľovaná dynamicky na základe ich aktuálneho využitia

vo forme bloku, čo je skupina niekoľkých warpov. Túto prácu je možné si predstaviť ako set inštrukcií, pre ktoré je zaručené, že budú vykonané sériovo. Tento set je po zvyšok kapitoly označovaný ako úloha. Pre súčasné architektúry sa môžu jednému SM prideľovať len bloky tej istej úlohy. SM si teda dokáže udržovať v pamäti stavy viacerých blokov úlohy, obmedzením je veľkosť jeho registrov a zdieľanej pamäte. Vďaka tomu SM jednoducho spúšťa a uspáva warpy jemu priradených blokov bez drahého prepínania kontextu. Táto vlastnosť SM umožňuje efektívne skrývanie latencie pri čítaní alebo zapisovaní do pamäte a iných časovo náročných operáciách, pretože je možné s minimálnou réžiou uspať určitý warp, kým sa čaká na realizáciu operácie a začať vykonávať inštrukcie nad iným warpom.

O odoberanie, plánovanie a delegovanie vykonania inštrukcií nejakej úlohy sa starajú takzvané warpové rozvrhovače a ich podriadení dispečeri. Rozvrhovače z aktívnych úloh odoberajú inštrukcie a delegujú ich na vykonanie k fyzickým jadrám SM, pričom počet týchto jadier záleží na architektúre. Je veľmi dôležité dbať na fakt, že inštrukciu vykonáva warp jadier súčasne, a že inštrukcia sa vždy vykoná v každom z týchto jadier. Napríklad ak má for cyklus v jednom vlákne dĺžku 4 a v ostatných vláknach warpu má dĺžku 1, tak bude dĺžka cyklu vo všetkých vláknach warpu rovná 4. GPU to rieši zamaskovaním vykonania troch iterácií for cyklu pre všetky vlákna okrem jediného, kde mal mať cyklus skutočne dĺžku 4. To je dôsledok toho, že SM pracuje na báze SIMT, čo môže v prípade zle navrhnutého programu s množstvom vetvenia pomocou if/else, prípadne s for a while cyklami rôznej dĺžky, viesť k nízkemu využitiu GPU. V záujme optimalizácie využitia je preto pre GPU aplikácie typické, že najlepšie sériové riešenie problému nemusí byť nevyhnutne najlepším pre GPU. Často sa volí iný prístup alebo modifikácie sériového riešenia, ktoré dokážu naplno využiť ponúkané prostriedky hardware, s ktorým sa pracuje.

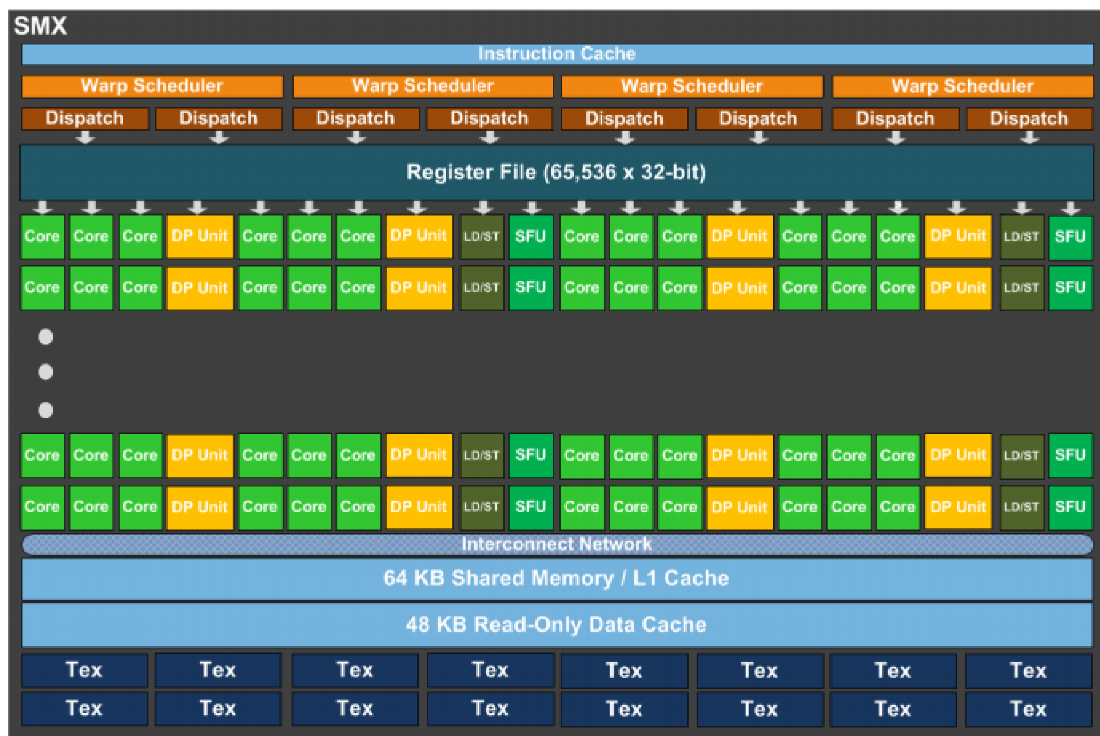
Mnoho uvedených informácií je závislých od použitej architektúry, preto nasleduje malé porovnanie architektúr Fermi a Kepler použitých v experimentoch:

- Karty Kepler môžu obsahovať až 2880 jadier rozdelených do 15 SM. Oproti tomu karty Fermi majú maximálne 512 jadier v 16 SM. Rozdiel v počte jadier ale nevedie k priamo úmernému zvýšeniu výkonu, pretože medzi architektúrami je veľké množstvo zmien, okrem iného aj zníženie frekvencie

jadier v architektúre Kepler takmer na polovicu. Vysoký počet jadier preto tento úbytok rýchlosti kompenzuje.

- Na prácu s týmto veľkým množstvom jadier sa počet warpových rozvrhovačov v Kepler SM zdvojnásobil, k dispozícii sú štyri.
- Ďalším spôsobom, ako vykompenzovať nižšiu frekvenciu jadier architektúry Kepler je delegovanie inštrukcie z warpu vždy na 32 fyzických jadier. Fermi SM na GPU použitej v experimentoch využíva iba 16 jadier, a preto potrebuje až dva takty na vykonanie jednej inštrukcie.

Zhrnutie vyššie uvedených informácií pre architektúru Kepler ilustruje Obrázok 5, kde je okrem iného možné vidieť aj jednotky slúžiace na výpočet špeciálnych funkcií ako sínus, odmocnina a podobne (SFU). Čím je ale konkrétne zobrazený čip GK110 výnimočný je prítomnosť špeciálnych jednotiek na počítanie so 64 bitovými hodnotami (DP Unit), čo je veľmi dôležitý aspekt pre aplikácie, ktoré využívajú 64 bitovú presnosť čísel. LD/ST predstavujú jednotky na čítanie a zapisovanie do pamäte, o ktorej pojednáva nasledujúca podkapitola.



Obrázok 5: SM čipu GK110, označovaný ako SMX, použitého aj v GPU NVIDIA Tesla K20, zdroj: nvidia.com

3.3 Pamäť GPU

Pamäť GPU má niekoľko dôležitých rozdielov oproti operačnej pamäti (RAM), ktorú používa CPU, pričom jej vlastnosťou je aj hierarchická členitosť. Pochopenie tejto hierarchie a pamäte ako takej, spolu s aplikovaním týchto vedomostí pri vytváraní aplikácií pre GPU je jedným z hlavných faktorov rozhodujúcich o tom, či bude výsledný kód efektívny. To je spôsobené niekoľkými faktami, ako napríklad:

- Prístupy do istých druhov pamäte sú typicky najdrahšími krokmi výpočtu, a preto je potrebné venovať sa ich optimalizácii.
- Pre pamäť GPU je typický paralelný prístup do pamäte a pri zlom návrhu kódu môže dochádzať k veľkému počtu konfliktov, ktoré vyústia v serializáciu prístupov a v konečnom dôsledku k mohutnému spomaleniu.

3.3.1 Registre

Ako znázorňuje Obrázok 5, každý SM má na čipe k dispozícii určitý počet registrov (napríklad 64KB v architektúre Kepler), do ktorých si ukladá stavy všetkých aktívnych vlákien, ako bolo popísané v predchádzajúcej kapitole. Práve veľkosť registrov ovplyvňuje maximálny počet aktívnych vlákien na jeden SM. Konkrétne vlákno má prístup len k svojim prideleným registrom, pričom registre sú najrýchlejšou pamäťou, ku ktorej má vlákno prístup.

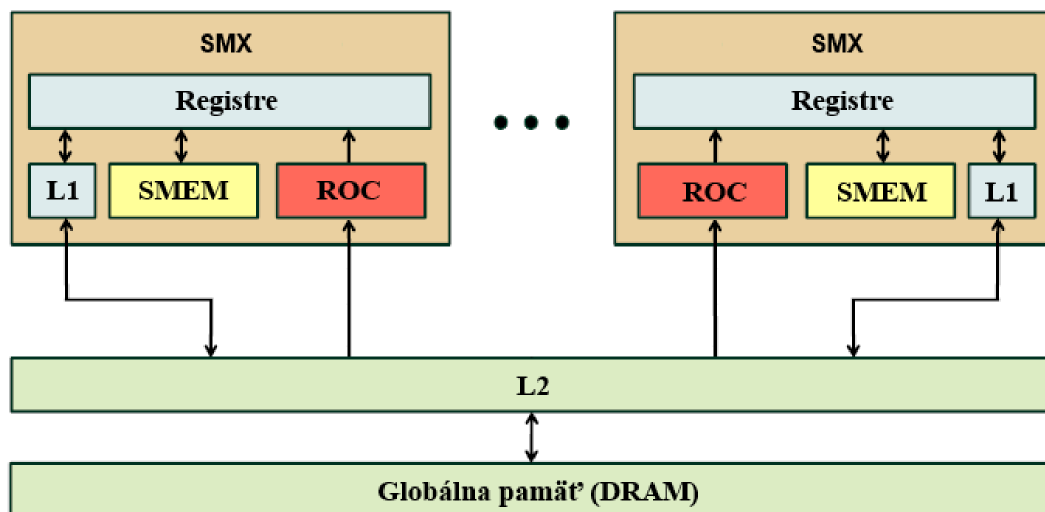
Lokálna pamäť

Problém s registrami nastane v situácii, ak chce vlákno používať viac ako maximálny povolený počet registrov, alebo dynamicky alokovať pamäť. V takom prípade je potrebný priestor alokovaný v takzvanej lokálnej pamäti, čo je kus globálnej pamäte vyhradený na exkluzívny prístup len pre konkrétne vlákno. Prístup do globálnej pamäte je zhruba o dva rády pomalší, ako prístup do registrov, a preto je vhodné sa tejto situácii vyhýbať, aj keď situáciu zlepšuje prítomnosť L1 a L2 cache. Problému a jeho vplyvu na výkon sa podrobnejšie venuje Micikevicius [35] a globálnej pamäti všeobecne sa venuje kapitola 3.3.3.

3.3.2 L1 cache a zdieľaná pamäť

L1 cache a zdieľaná pamäť (shared memory) predstavujú fyzicky ten istý druh pamäte, každá sa ale používa iným spôsobom. Každý SM má túto pamäť vlastnú, ako ukazuje Obrázok 6, a ich súhrnná veľkosť pre architektúry Fermi a Kepler je 64KB. Pomer rozdelenia medzi tieto pamäte je variabilný, a to 25%, 50% alebo 75% celkovej kapacity.

- L1 cache je transparentná, pričom jej úloha je priamočiara – slúži na zrýchlenie čítania a zapisovania z lokálnej pamäte, ktorej význam bol zmienený v predchádzajúcej podkapitole. Toto je novinka v architektúre Kepler oproti starším architektúram ako Fermi, pretože tie používali L1 cache aj na cachovanie prístupov do globálnej pamäte [36].
- Zdieľaná pamäť slúži na zdieľanie hodnôt, napríklad medzivýsledkov, medzi vláknami, pričom je neporovnateľne rýchlejšia ako globálna pamäť a za optimálnych podmienok dosahuje takmer rýchlosť registrov. Čítanie a zapisovanie do zdieľanej pamäte si vlákno rieši vo vlastnej réžii.



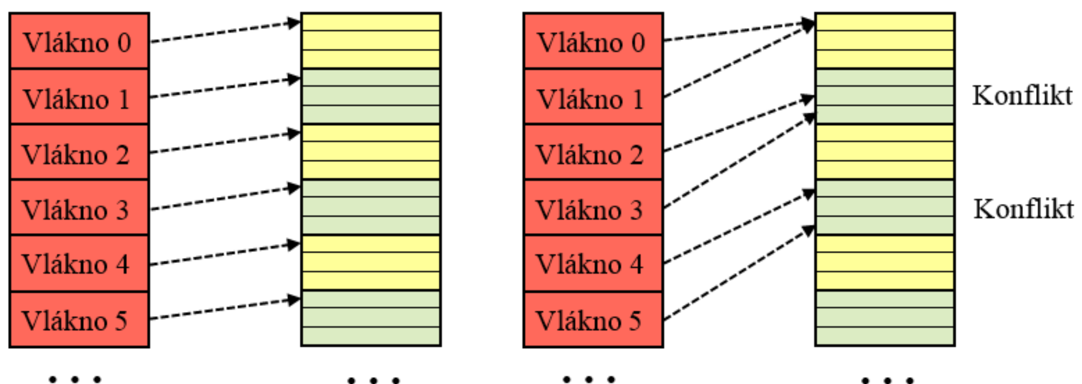
Obrázok 6: Hierarchia pamäte GPU, kde L1 a L2 sú cache, SMEM je zdieľaná pamäť a ROC je cache len na čítanie

Novinkou architektúry Kepler a čipu GK110 je špeciálna L1 cache o veľkosti 48KB určená len na čítanie, ako ilustruje Obrázok 6. Jej úlohou je načítavanie dát z L2 cache, respektíve globálnej pamäte, za účelom odľahčenia L1 cache a zdieľanej pamäte a zvýšenia priepustnosti. Tento mechanizmus slúži ako náhrada za to, že

architektúra Kepler necachuje prístupy do globálnej pamäte v štandardnej L1 cache, je ale dostupný až pre GPU podporujúce compute capability 3.5 [36].

Rýchlosť zdieľanej pamäte spočíva v spôsobe, akým sa k nej pristupuje. Je rozdelená na časti rovnakej veľkosti nazývané banky, ktorých je aj v architektúrach Fermi aj Kepler 32. Každá banka má konkrétnu priepustnosť (pre Kepler je to 64 bitov) a po sebe idúce banky sú obsadzované po sebe idúcimi slovami. Veľkosť slova typicky odpovedá priepustnosti jednej banky, ale nie je to pravidlom, napríklad architektúra Kepler podporuje rôzne veľkosti slov. Vďaka tomuto dokáže zdieľaná pamäť v ideálnom prípade obslúžiť až 32 prístupov súčasne, ak vlákna pristupujú do rôznych bánk, alebo ak pristupujú na to isté slovo v jednej banke.

Pri práci so zdieľanou pamäťou je nutné dbať na riziko takzvaných bankových konfliktov. Tieto vznikajú ak aspoň dve vlákna z jedného warpu pristupujú do tej istej banky súčasne, pričom každé z vlákien pristupuje k inému slovu v banke. Výsledkom bankových konfliktov je serializácia konfliktných pamäťových prístupov, čo môže mať dramatický vplyv na výkon. Konfliktom je väčšinou možné predchádzať analýzou prístupu do bánk a ich následnou optimalizáciou. Prirodzeným riešením je, aby vlákna z jedného warpu adresovali tie isté slová v jednej banke, alebo aby každé vlákno využívalo svoju banku. Príklad bezkonfliktného a konfliktného prístupu do zdieľanej pamäte ukazuje Obrázok 7. Hlbšie o zdieľanej pamäti a bankových konfliktoch v závislosti od konkrétnej compute capability pojednáva CUDA dokumentácia [36].



Obrázok 7: Prístup do zdieľanej pamäte, vľavo bez bankových konfliktov, vpravo s dvoma bankovými konfliktami, kedy vlákna pristupujú na iný prvok v jednej banke

3.3.3 Globálna pamäť a L2 cache

Globálna pamäť, označuje sa aj DRAM, je analógiou operačnej pamäte CPU. Je dostupná jednak z GPU ale aj CPU a sprostredkúva komunikáciu medzi týmito dvoma zariadeniami, viď nasledujúca podkapitola. Jej veľkosť je v dnešných profesionálnych GPU zhruba na úrovni 6 – 12GB. Táto pamäť je dostupná všetkým SM a prístup do nej je zo všetkých typov GPU pamätí najdrahší.

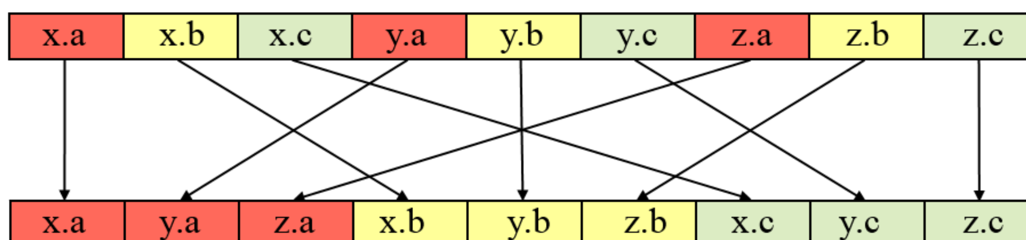
L2 cache je rovnako ako L1 cache transparentná, rozdielom je ale to, že je zdieľaná medzi všetkými SM, ako znázorňuje Obrázok 6, čo má za následok aj jej väčšiu veľkosť (pre architektúru Kepler to je 1536kB). Jej hlavným a jediným účelom je urýchliť opätovné prístupy do globálnej pamäte.

Vzhľadom na to, že prístup do globálnej pamäte prebieha vždy zo všetkých vlákien v jednom warpe, tak sa ich GPU snaží zlučovať do čo najmenšieho počtu jednotlivých transakcií. S týmto je ale spojené veľké riziko vplývajúce na výkon, ak sú dáta v pamäti zle organizované. Problém vzniká nasledovne:

- Súbežné prístupy vlákien jedného warpu do globálnej pamäte sú rozdelené na toľko požiadaviek, koľko riadkov v cachi je potrebných alokovať na ich obslúženie. Veľkosť riadkov závisí od konkrétnej architektúry, pretože riadok v L1 a L2 cachi má rozdielnu veľkosť.
- Ak vlákna pristupujú k dátam roztrúseným v globálnej pamäti, je možné, že pre každé vlákno vo warpe bude musieť byť vytvorená vlastná pamäťová transakcia, čo je v prípade globálnej pamäte neefektívne.

Problému je možné predchádzať, ak budú dáta v pamäti zarovnané podľa veľkosti riadku cache a vlákna vo warpe budú k dátam pristupovať po zarovnaných kompaktných blokoch, inak je zlučovanie jednotlivých transakcií fakticky nemožné. To je často možné dosiahnuť napríklad používaním štruktúry polí namiesto poľa štruktúr, ako znázorňuje Obrázok 8. Rozdiel v organizácii pamäte a používanie štruktúry polí je jedným z hlavných logických rozdielov medzi algoritmami navrhovanými pre CPU a pre GPU, čo opäť vyplýva z toho, že GPU pracuje na báze SIMT, ako bolo uvedené v kapitole 3.1. Tento problém má v závislosti od

architektúry GPU svoje špecifiká, ktorým sa spoločnosť NVIDIA dôkladne venuje vo svojej dokumentácii [36].



Obrázok 8: Hore: pole štruktúr, alebo objektov, typické pre programovanie na CPU, dole: štruktúra polí, typická pre GPU

3.4 Programovanie GPU technológiou CUDA

CUDA, Compute Unified Device Architecture, predstavuje platformu pre všeobecné paralelné výpočty navrhnutú a vytvorenú spoločnosťou NVIDIA, pričom podpora práce s touto platformou je štandardom pre všetky GPU, ktoré spoločnosť v posledných rokoch produkuje. Samotná NVIDIA poskytuje niekoľko knižníc využívajúcich paralelizmus na GPU za účelom urýchlenia ľubovoľných aplikácií bez hlbších znalostí tejto technológie. K najobľúbenejším patrí napríklad knižnica cuFFT na výpočet rýchlej Fourierovej transformácie [37], alebo knižnica základných operácií lineárnej algebry (BLAS[1]) známa ako cuBLAS [38]. Dôležitejším aspektom technológie CUDA je fakt, že poskytuje jasné, silné a dobre zdokumentované rozhranie [36] na tvorbu vlastných aplikácií, ktoré môžu za účelom urýchľovania využívať surovú výpočtovú silu dnešných GPU.

Základný koncept využívania technológie CUDA je takmer totožný so spôsobom komunikácie medzi CPU a GPU popísaným v kapitole 3.1, pričom spoločnosť NVIDIA poskytuje SDK, ktoré vytvára kompletné vývojové prostredie pre C a C++ programátorov za účelom realizácie tohto konceptu. SDK okrem iného obsahuje špeciálny CUDA kompilátor, základné matematické knižnice a nástroje pre ladenie aplikácií. Niektoré dôležité aspekty práce s technológiou CUDA sú vysvetlené v nasledujúcich podkapitolách.

[1] Skratka z anglického Basic Linear Algebra Subprograms

3.4.1 Kompilácia kódu pre GPU

Kód aplikácie využívajúcej technológiu CUDA vyzerá v základe rovnako, ako bežný kód v jazyku C/C++ s tým, že CUDA knižnice poskytujú niekoľko rozšírení:

- Súbor klasických funkcií poskytnutý napríklad za účelom práce s globálnou pamäťou GPU atď. sa súhrnne nazýva CUDA runtime.
- Špeciálna syntax na označenie metód bežiacich na GPU a tiež na ich spúšťanie. Takto označené metódy sa nazývajú kernely.

Označovanie kernelov je nutné na to, aby ich CUDA kompilátor (NVCC) rozpoznal a vo svojej réžii skompiloval na kód spustiteľný na GPU. Zvyšný neoznačený kód sa následne kompiluje štandardným kompilátorom, ako GCC alebo Visual C++. Toto je len veľmi zjednodušený pohľad na kompiláciu, ktorá je spolu s parametrami pre NVCC podrobne vysvetlená v CUDA dokumentácii [36]. Označovanie kódu pre GPU prebieha pomocou dvoch kľúčových slov:

- `__global__` označuje metódy volané z CPU a spúšťané na GPU.
- `__device__` označuje metódy volané z GPU a spúšťané tiež na GPU.

Špeciálna syntax slúžiaca na spúšťanie kernelov je vysvetlená v kapitole 3.4.4. Z hľadiska kompilácie je dôležité, že volanie metódy kernelu sa v kóde nahradí vygenerovaným tzv. stubom, ktorý slúži ako proxy pre skutočné volanie kernelu. Po zavolaní stubu spustí CUDA na pozadí príslušný kernel.

Veľmi užitočnými nástrojmi z hľadiska kompilácie CUDA aplikácií sú poskytnuté makrá, ktoré tiež pomáhajú identifikovať úseky kódu určené pre NVCC:

- `__NVCC__` je definované, ak je zdrojový kód práve kompilovaný s NVCC.
- `__CUDACC__` je definované, ak NVCC práve kompiluje CUDA zdrojový súbor, čo sa určuje podľa prípony súboru (*.cu).
- `__CUDA_ARCH__` vracia hodnotu identifikujúcu verziu compute capability, pre ktorú NVCC práve kompiluje kód. Toto makro je vhodné používať, ak sa problém rieši iným spôsobom v závislosti od rozličných prostriedkov, ktoré sú dostupné v GPU podporujúcich rôzne verzie compute capability.

3.4.2 Získanie prístupu ku GPU

CUDA runtime poskytuje niekoľko základných metód na detekciu GPU a jej základných vlastností, tieto sú:

- `cudaGetDeviceCount`, ktorá má jeden výstupný parameter, do ktorého sa zapíše počet dostupných GPU,
- `cudaSetDevice`, ktorá berie ako parameter index jedného z dostupných GPU, indexuje sa od nuly, pričom všetky ďalšie CUDA metódy, ako napríklad alokovanie pamäti, viď nasledujúca kapitola, sú po jej vykonaní smerované na GPU s daným indexom,
- `cudaGetDeviceProperties`, ktorá tiež dostáva na vstupe index jednej z GPU a do výstupného parametru zapíše všetky dôležité informácie o danej GPU, ako napríklad súhrnná veľkosť globálnej alebo zdieľanej pamäte.

3.4.3 Práca s globálnou pamäťou GPU

Pred každým netriviálnym výpočtom na GPU je na nej potrebné alokovať kus globálnej pamäte, do ktorého sa následne skopírujú dáta z pamäte CPU, prípadne sa táto pamäť ponechá prázdna a slúži na predávanie výsledku z GPU naspäť do CPU. Tiež je dobrým zvykom nečakať na ukončenie aplikácie a alokovanú pamäť uvoľniť hneď, ako je to možné. Na tieto účely slúžia postupne metódy `cudaMalloc`, `cudaMemcpy` a `cudaFree`, ktorých použitie znázorňuje Úryvok kódu 1. Proces práce s dynamicky alokovanou pamäťou GPU je veľmi podobný tomuto procesu na CPU s využitím metód `malloc`, `copy` a `free` známych zo štandardnej knižnice jazyka C. Ukazovatele na pamäť na GPU fungujú úplne rovnako, ako ukazovatele na pamäť CPU, pričom je na programátorovi, aby rozlišoval, či premenná typu ukazovateľ v skutočnosti ukazuje na pamäť alokovanú na GPU alebo na pamäť CPU.

```
float* d_Ptr;
cudaMalloc((void**)&d_Ptr, count * sizeof(float));

float* h_Ptr = new float[count];
cudaMemcpy(d_Ptr, h_Ptr, count * sizeof(float),
cudaMemcpyHostToDevice);

cudaFree(d_Ptr);
```

Úryvok kódu 1: Alokácia, kopírovanie a uvoľňovanie pamäte GPU, na ktorú ukazuje d_Ptr

Nakoľko CPU nemá priamy prístup k pamäti GPU, tak aj na kopírovanie dát z GPU na CPU je nutné použiť metódu `cudaMemcpy` s obmenenými parametrami. Oproti použitiu `cudaMemcpy` v kóde vyššie sa zmení cieľ a zdroj kopírovania a takisto posledný parameter, ktorý určuje smer kopírovania.

3.4.4 Spúšťanie kódu na GPU

Pri volaní kernelov z CPU, označených kľúčovým slovom `__global__`, je nutné použiť špeciálnu syntax, viď Úryvok kódu 2, kde ukazovateľ posielaný ako parameter kernelu, `arg`, musí ukazovať na pamäť alokovanú na GPU metódou `cudaMalloc`. Špecifikom volania kernelu sú parametre uvedené v `<<< >>>`, ktorých význam je spätý s tým, ako sú kernely na GPU spúšťané. Koncept spúšťania, ktorý čiastočne ilustruje Obrázok 9, je možné zhrnúť do niekoľkých bodov:

- Absolútnou podstatou a zmyslom existencie kernelu je, aby bol spúšťaný na veľkom počte GPU jadier súčasne, bez toho nemá použitie GPU na riešenie daného problému zmysel, pretože výpočet nie je paralelný.
- V jednom okamihu beží na jednom GPU jadre práve jedno vlákno.
- Tieto vlákna sú zoskupené do blokov rovnakej veľkosti, pričom GPU rozvrhuje tieto bloky jednotlivo na konkrétne SM. Z toho okrem iného vyplýva, že vlákna v jednom bloku majú prístup do tej istej zdieľanej pamäte, keďže tá sa nachádza na SM.
- Bloky sú začlenené do takzvanej mriežky (grid), ktorá obsahuje všetky vlákna, v ktorých beží jeden kernel.

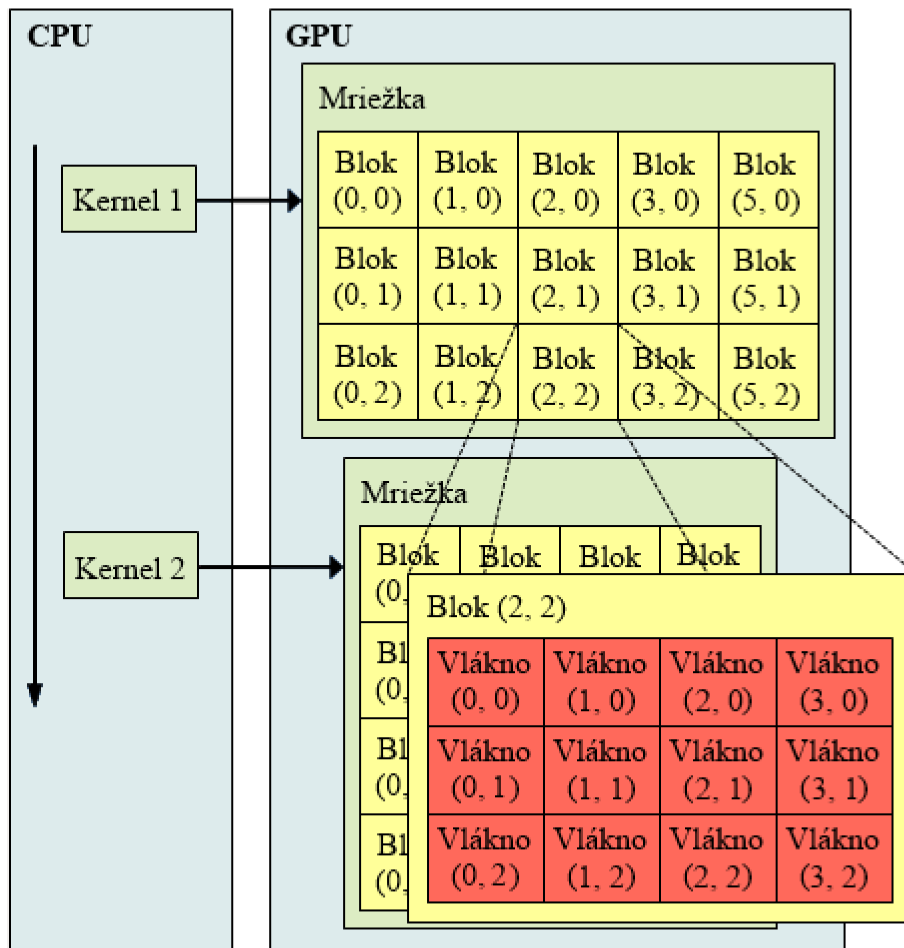
```
__global__ void gpu_method(float* arg) { … }

gpu_method<<<gridSize, blockSize, smemSize>>>(arg);
```

Úryvok kódu 2: Spúšťanie kódu na GPU z CPU

Na to nadväzuje vysvetlenie významu parametrov spustenia kernelu:

- `gridSize` je počet blokov v mriežke.
- `blockSize` je počet vlákien v každom bloku mriežky.

- `smemSize` je veľkosť zdieľanej pamäte v bajtoch, ktorú môže každý blok kernelu využívať. Ak sú dva bloky jedného kernelu rozvrhnuté na rovnaký SM, tak má každý z nich vyhradenú svoju vlastnú časť zdieľanej pamäti.
- Aby bolo možné pohodlne organizovať vlákna pre rôzne typy problémov, tak `gridSize` aj `blockSize` sú jedna až tri dimenzionálne, Obrázok 9 demonštruje počet dimenzií dva.



Obrázok 9: Organizácia vlákien pri spúšťaní kernelu na GPU

Organizácia vlákien má niekoľko dôležitých dôsledkov:

- Kvôli využitiu GPU je vhodné zarovnávať počty vlákien bloku na násobky veľkosti warpu, pretože podľa kapitoly 3.2 sa každá inštrukcia vykoná vždy v každom vlákne warpu, čo pre architektúru Kepler odpovedá inštrukcii vykonanej 32 jadrami súčasne.

- Každé vlákno a blok má svoj index, ktorý ho jednoznačne identifikuje v bloku a v mriežke. Vlákno pozná aj svoj index aj index bloku, v ktorom sa nachádza, a na základe toho je možné spočítať jednoznačnú identifikáciu každého vlákna mriežky.
- Identifikácia vlákien prebieha na základe štruktúr, ktoré sú dostupné v každom kerneli, tieto sú:
  - `threadIdx` určuje index vlákna v bloku, ktorého je súčasťou,
  - `blockDim` určuje počet vlákien bloku,
  - `blockIdx` určuje index bloku v mriežke, ktorého je súčasťou,
  - `gridDim` určuje počet blokov mriežky.

  Všetky tieto štruktúry majú členské premenné `x`, `y` a `z`, keďže bloky a mriežky majú 1 – 3 dimenzie.

3.4.5 Komunikácia vlákien

Jednou zo zásadných podúloh, ktoré typicky musí riešiť kernel, je odovzdávanie čiastkových výsledkov, alebo dát všeobecne, medzi vláknami, čo sa zabezpečuje niekoľkými rozdielnymi spôsobmi. Príkladom využitia môže byť rozdelenie vyhodnocovania účelovej funkcie medzi viacero vlákien, z ktorých je následne nutné agregovať spočítané hodnoty. Na to, aby bolo možné začať agregáciu čiastkových výsledkov je nutné overiť, či všetky vlákna dokončili svoju prácu. Na tento aspekt komunikácie sú zamerané aj príklady použitia po zvyšok tejto kapitoly.

Komunikácia globálnou pamäťou

Komunikácia globálnou pamäťou je pomalá, pretože ako popisuje kapitola 3.3.3, prístup do nej je zo všetkých pamätí najdrahší a navyše si vyžaduje zložitejší spôsob synchronizácie. Jediným spôsobom synchronizácie globálnej pamäti je zapojiť atomické inštrukcie, ktoré prinášajú značné spomalenie, nakoľko z ich podstaty vyplýva, že ich vykonanie je nutné serializovať. Kompletný zoznam funkcií je uvedený v CUDA dokumentácii [36], k najbežnejším patria `atomicAdd`, `atomicSub` na atomické sčítanie a odčítanie, či `atomicAnd` a `atomicOr`, čo sú atomické bitové operácie.

Na overenie toho, že všetky vlákna mriežky spočítali svoju časť úlohy a čiastkový výsledok je pripravený v globálnej pamäti sa typicky používa:

- Atomická funkcia `atomicInc`, ktorá pracuje nasledovne:
    - Ako prvý parameter funkcia dostáva ukazovateľ na premennú, ktorej hodnotu má zvýšiť o jedna.
    - Druhým parametrom je číselná hodnota predstavujúca limit. Ak by premenná po zvýšení tento limit dosiahla, tak `atomicInc` premennú namiesto zvýšenia o jedna vynuluje.
- Pamäťová bariéra `__threadfence`, ktorá okrem iného zaručuje, že zápisy do globálnej pamäte z volajúceho vlákna sú po jej volaní viditeľné vo všetkých ostatných vláknach mriežky.

To, ako overiť, že všetky vlákna v mriežke zapísali čiastkový výsledok do globálnej pamäte ukazuje Úryvok kódu 3. Vlákno, v ktorom platí `(value == 0)` je tým posledným, ktoré svoj čiastkový výsledok zapísalo do globálnej pamäte a tým pádom sú všetky čiastkové výsledky uložené v poli `arg` pripravené na agregáciu.

```
__device__ int count = 0;
__global__ void kernel(float* arg)
{
    int partialResult = ...;
    int idx = (blockDim.x * blockIdx.x) + threadIdx.x;
    arg[idx] = partialResult;

    __threadfence();
    int value = atomicInc(&count, gridDim.x * blockDim.x);

    if (value == 0) { /* aggregate */ }
}
```

Úryvok kódu 3: Práca s globálnou pamäťou a jej synchronizácia

Komunikácia zdieľanou pamäťou

Ďalším spôsobom je komunikácia vlákien skrz zdieľanú pamäť, kde jedno z vlákien hodnotu do zdieľanej pamäti zapíše a druhé vlákno si na tomto mieste môže príslušnú hodnotu prečítať. Možný spôsob použitia zdieľanej pamäte zachytáva Úryvok kódu 4, na ktorom je demonštrované, že:

- Deklarácia premennej nachádzajúcej sa v zdieľanej pamäti prebieha pomocou kľúčového slova `__shared__`.

- Synchronizácia vlákien po zápise do zdieľanej pamäte prebieha pomocou volania metódy __syncthreads, pričom za túto bariéru sa žiadne vlákno nedostane, až kým ju nedosiahnu všetky vlákna v bloku. Takýchto synchronizačných metód poskytuje CUDA niekoľko druhov [36].
- Pozícia idx v poli alokovanom v zdieľanej pamäti, na ktorú bude vlákno zapisovať sa typicky spočíta pomocou premenných identifikujúcich vlákno, ktoré sú popísané v predchádzajúcej podkapitole.

```
extern __shared__ float sharedMemory[];

int idx = threadIdx.x;
sharedMemory[idx] = value;
__syncthreads();
```

Úryvok kódu 4: Práca so zdieľanou pamäťou

Komunikácia zdieľanou pamäťou sa vo veľmi jednoduchej forme používa aj na akési explicitné cachovanie hodnôt pomalej globálnej pamäte do rýchlejšej zdieľanej pamäte. To je možné realizovať tak, že každé vlákno prečíta prvok globálnej pamäte a uloží ho do zdieľanej pamäte. Tým každé vlákno sprístupní ním načítané dáta aj ostatným vláknam v bloku. Zvyšné vlákna potom nemusia dáta znova čítať z pomalej globálnej pamäte, ale využijú dáta nacachované v zdieľanej pamäti.

Komunikácia warpovými inštrukciami

Ďalším spôsobom vymieňania dát medzi vláknami dostupným v GPU podporujúcich compute capability 3.0, je využívanie __shfl inštrukcií operujúcimi v rámci warpu. S ich použitím dokáže vlákno pristupovať do registrov iného vlákna nachádzajúceho sa v rovnakom warpe. Výhodou oproti predošlým spôsobom komunikácie je rýchlosť, keďže registre sú podľa kapitoly 3.3.1 najrýchlejšou pamäťou na GPU. Prístup pomocou __shfl inštrukcií má ale aj niekoľko nevýhod:

- Cez __shfl inštrukcie komunikujú len vlákna vo warpe a nie v celom bloku, takže nedokážu úplne zastúpiť komunikáciu cez zdieľanú pamäť.
- __shfl inštrukcie dokážu presúvať len 32 bitové hodnoty, kombináciou s inými funkciami je ale možné dosiahnuť presun 64 bitových hodnôt, pričom sa dvakrát vykoná __shfl inštrukcia pre 32 bitov. Tento postup popisuje napríklad Demouth [39].

4. Paralelizácia metódy časticovej optimalizácie technológiou CUDA

V kapitole 2.3 bola predstavená metóda časticovej optimalizácie používaná pri hľadaní globálneho minima účelovej funkcie, teda pri globálnej optimalizácii. Táto metóda je vďaka svojim vlastnostiam vhodným kandidátom na paralelizáciu použitím GPU, konkrétne použitím technológie CUDA, ktorá bola spolu so základmi architektúry GPU predstavená v kapitole 3. V tomto smere existuje niekoľko odborných publikácií, ktoré sú priblížené v nasledujúcej podkapitole.

Po tomto úvode nasleduje podkapitola venovaná implementácii, konkrétne paralelizácii jednotlivých krokov metódy časticovej optimalizácie na GPU spolu s porovnaním ich rýchlosti pre rôzne implementované verzie. Za týmto účelom je vždy používaný rovnaký hardware, ktorého kľúčové vlastnosti sú popísané v podkapitole 4.3 a set optimalizovaných účelových funkcií popísaný v kapitole 4.2, Tabuľka 1. Následne v kapitole 5 bude poskytnuté komplexné porovnanie referenčnej verzie pre CPU, spolu s jej stručným popisom, a paralelnej verzie pre GPU, ktorej vytvorenie je hlavným cieľom tejto práce.

4.1 Súvisiace práce

Časticovej optimalizácii je venované veľké množstvo odborných článkov a iných publikácií, niekoľko z nich však priamo pojednáva o využití GPU pri paralelizácii tejto metódy.

Prvým, kto písal o využití GPU za týmto účelom bol Li et al. [40], ktorý tak učinil ešte pred vznikom technológie CUDA, teda v čase, keď boli GPU zamerané výlučne na prácu s grafikou. V tejto práci sa najskôr muselo zadanie problému prácne namapovať na textúry, ktoré boli následne spracovávané shadermi GPU a výsledok bolo nutné spätne previesť z textúr na čísla. Aj napriek týmto komplikáciám a faktu, že toto riešenie je v dnešnej dobe značne zastarané, poslúžila práca ako dôkaz životaschopnosti urýchľovania časticovej optimalizácie pomocou GPU, pretože

týmto riešením bolo dosiahnuté pre problémy vysokej dimenzie až desaťnásobné zrýchlenie.

V čase keď spoločnosť NVIDIA začala produkovať GPU podporujúce technológiu CUDA prišiel s vlastnou implementáciou paralelnej časticovej optimalizácie na GPU aj Zhou et al. [41]. Vo svojej práci popisuje všetky kroky riešenia, pričom niektoré problémy, ktorým venuje priestor vo svojej práci sú už v dnešnej dobe neaktuálne a prekonané. Jedným z problémov bola absencia kvalitného a rýchleho generátora náhodných čísel na GPU, čo vyriešila spoločnosť NVIDIA vytvorením knižnice na tento účel, nazvanej cuRAND [42]. Najväčší dôraz je v tejto práci kladený na paralelizáciu vyhodnocovania účelovej funkcie, pričom na tento účel je použitý počet vlákien rovný počtu častíc. Vyhodnocovanie prebieha po jednotlivých dimenziách s ohľadom na zlučovanie pamäťových prístupov. Na záver je výsledná hodnota spočítaná ako kombinácia, čiastkových výsledkov, typicky suma.

Mussi et al. [43] prichádza so zaujímavou myšlienkou rozšíriť paralelizáciu, a to tak, že na vyhodnocovanie účelovej funkcie pre jednu časticu je použitý celý blok vlákien, ktorý má veľkosť o počte dimenzií riešeného problému. Veľkosť bloku je kvôli zlučovaniu prístupov do globálnej pamäte vždy zaokrúhlená nahor na najbližší násobok veľkosti warpu. Tento prístup je vhodný napríklad pre účelové funkcie tvaru sumy, ktoré sčítajú cez jednotlivé dimenzie vstupu a sčítance sú dostatočne aritmeticky náročné na vyhodnotenie, aby sa oplatilo vyhodnocovanie rozdeľovať medzi väčší počet vlákien. Ako príklad bola v práci poskytnutá Rastriginova funkcia, ktorá je spomenutá aj v kapitole 2.1.

Prístup Mussiho et al. [43] vo svojej práci kritizuje Hung et al. [44], pretože si podľa neho vyžaduje dodatočnú réžiu na komunikáciu medzi vláknami, v prípade, ak nie je možné účelovú funkciu rozdeliť na výpočet jednotlivých nezávislých zložiek pre každú dimenziu. Ním navrhovaný prístup spočíva rovnako ako v prípade práce Zhoua et al. [41] vo vyhodnocovaní účelovej funkcie pre jednu časticu v jednom vlákne a svoj postup popisuje do väčších detailov.

Existuje tiež niekoľko článkov zameraných na rozšírenie už existujúcich paralelných implementácií časticovej optimalizácie, ako napríklad:

- V náväznosti na prácu Zhoua et al. [41] vypracoval ten istý autorský kolektív rozšírenie zamerané na optimalizáciu viacerých účelových funkcií súčasne [45], pričom ale autori reálne posudzovali len set funkcií o veľkosti dva.
- S asynchrónnou verziou tejto metódy pre GPU prišiel Mussi et al. [46], pričom implementácia využíva LBEST modifikáciu topológie častíc popísanú v kapitole 2.3.2. Implementácia je však obmedzená maximálnym počtom blokov, ktoré dokáže GPU paralelne vykonávať, každá častica je vyhodnocovaná práve jedným blokom.
- Zhu et al. [47] vypracoval článok venovaný kombinácii paralelnej časticovej optimalizácii s metódou hľadania vzorov, ktorá podľa nameraných výsledkov vedie k celkovému zrýchleniu metódy a zachováva kvalitu nájdených výsledkov.

Prínos tejto práce oproti vyššie popísaným prácam spočíva v tom, že dôkladne analyzuje a porovnáva rôzne prístupy paralelizácie metódy časticovej optimalizácie. Na základe tejto analýzy je následne zvolený najefektívnejší prístup. Špecifikom je tiež zameranie sa na konkrétny druh účelových funkcií, ktoré majú vysokú výpočtovú zložitosť aj v prípade nízkej dimenzionality. Takýto druh problému žiadna z popísaných prác nerieši. Výhodou tejto práce je tiež, že vzniká s miernym časovým odstupom, pričom GPU hardware spolu s technológiou CUDA za tento čas značne pokročil a poskytuje programátorom silnejšie prostriedky, ako napríklad špeciálne inštrukcie na rýchlu prácu s pamäťou. To do tejto práce prináša aj preskúmanie možných riešení pre viacero v súčasnosti využívaných architektúr GPU, aby zvolené riešenie podporovala široká škála zariadení.

4.2 Použité účelové funkcie

Text kapitoly 4.3 pojednávajúcej o implementácii paralelizácie metódy časticovej optimalizácie sa často odkazuje na konkrétne účelové funkcie. Popis týchto funkcií uvádza Tabuľka 1. Pôvod netriviálnych funkcií je nasledovný:

- Funkcie f_1 a f_2 definoval Krishnakumar et al. [48] a boli spolu s funkciami f_3 a f_4 použité aj Hungom et al. [44] v jednej zo súvisiacich prác.
- Funkcia f_4 bola uvedená Rastriginom [5] a je spomenutá už aj v kapitole 2.1.
- Funkcia f_5 predstavuje zovšeobecnený tvar funkcií, ktoré sú používané firmou RSJ a.s. na riešenie konkrétnej úlohy vo svete investovania na burze cenných papierov. Podoba funkcie f_5 je závislá od externého konfiguračného súboru, pričom popis týchto súborov pre verzie funkcie f_6 až f_{10} uvádza Príloha A – Obsah priloženého disku .

Funkcia	**Interval**	**Minimum**
$f_1\langle n\rangle = \sum_{d=1}^{n}\left(\sin(x[d]) + \sin\left(\frac{2x[d]}{3}\right)\right)$	$[3, 13]^n$	$-1.21598n$
$f_2\langle n\rangle = \sum_{d=1}^{n-1}\left(\sin(x[d] + x[d+1]) + \sin\left(\frac{2x[d]x[d+1]}{3}\right)\right)$	$[3, 13]^n$	$\approx -2(n-1)$
$f_3\langle n\rangle = \sum_{d=1}^{n}(x[d]^2 - 10\cos(2\pi x[d]) + 10)$	$[-5.12, 5.12]^n$	0
$f_4\langle n\rangle = \sum_{d=1}^{n} x[d]^2$	$[-5.12, 5.12]^n$	0
$f_5\langle n, P\rangle = \sum_{j=1}^{P}\left(b_j - \sum_{d=1}^{n} x[d]^2 a_j[d]\right)^2$	–	–
$f_6 = f_5\langle 16, 16384\rangle$	$[-100, 100]^n$	0
$f_7 = f_5\langle 16, 65536\rangle$	$[-100, 100]^n$	0
$f_8 = f_5\langle 16, 272144\rangle$	$[-100, 100]^n$	0
$f_9 = f_5\langle 16, 1088576\rangle$	$[-100, 100]^n$	0
$f_{10} = f_5\langle 4, 6980011\rangle$	$[-1000, 1000]^n$	≈ 323428599

Tabuľka 1: Popis testovaných funkcií, n je počet dimenzií. Funkcia f_5 je parametrizovaná pomocou n a P, ktoré sú odvodené z externých konfiguračných súborov, ktoré definujú aj hodnoty polí a, b. Popis týchto súborov obsahuje Príloha A – Obsah priloženého disku

Z hľadiska prínosu tejto práce je najdôležitejšia paralelizácia funkcie f_5, nakoľko tomuto druhu účelových funkcií sa nevenuje žiadna zo spomenutých súvisiacich prác. Z tohto dôvodu obsahuje Tabuľka 1 väčšie množstvo jej variant.

Sémantický význam minimalizácie účelovej funkcie f_5 je možné chápať nasledovne:

- Skalárne hodnoty z b a vektory z a tvoria set P dvojíc, o ktorých je známe, že pre každú dvojicu vektor a_j a skalár b_j existuje určitý vektor koeficientov x rovnakej dĺžky ako a_j, pre ktorý platí:

  $$\mathrm{b}_j = \sum_{d=1}^{n} x[d]a_j[d]$$

  Pre $j = \{1, \dots, P\}$
- Problémom je, že vektory koeficientov x nie sú známe, a preto je cieľom aproximovať čo najpresnejší spoločný vektor koeficientov pre všetky dvojice, aby rozdiel medzi stranami vyššie uvedenej rovnosti bol čo najmenší. Prístup sa dá chápať ako aplikovanie metódy najmenších štvorcov.
- Tento spoločný vektor sa hľadá časticovou optimalizáciou a teda nájdená pozícia častice po skončení optimalizácie predstavuje vektor koeficientov s požadovanými vlastnosťami.

4.3 Testovací hardware

V nasledujúcich kapitolách, ktoré pojednávajú o implementácii časticovej optimalizácie na GPU a jej porovnaní s verziou pre CPU je používaný set konkrétneho hardware. Jeho kľúčové vlastnosti majú veľký vplyv na dosiahnuté výsledky, a preto Tabuľka 2 obsahuje stručný popis tých najdôležitejších z nich. Po zvyšok tejto práce budú GPU počas testovania odkazované pomocou mien uvedených v tejto tabuľke, teda M2090, GTX660Ti a K20.

Ako ukazuje Tabuľka 2, GPU M2090 sa môže zdať oproti zvyšným dvom použitým modelom slabšia najmä kvôli nízkemu počtu CUDA jadier. To súvisí s veľkými zmenami medzi architektúrami Fermi a Kepler, čo vyústilo k zníženiu frekvencie jadier pre architektúru Kepler. Stratu v počte jadier teda doháňa GPU M2090 pomocou ich vyššej frekvencie. Bližšie o týchto zmenách pojednáva kapitola 3.2.

	M2090	GTX660Ti	K20
Architektúra	Fermi	Kepler	Kepler
Compute Capability	2.0	3.0	3.5
Počet jadier	512	1344	2496
Počet SM	16	7	13
Rýchlosť jadier	1300MHz	980MHz	706MHz
Veľkosť pamäte	6GB	2GB	5GB
Rýchlosť pamäte	1.85GHz	3GHz	2.6GHz

Tabuľka 2: Popis základných vlastností GPU použitých na testovanie, zdroj: NVIDIA [49,50,51]

Popri použitých GPU je počas testovania často uvádzané aj porovnanie výkonu s verziou časticovej optimalizácie pre CPU. Za týmto účelom je tiež vždy používaný rovnaký hardware, jedná sa o procesor Intel i7-2600K [52] so:

- štyrmi fyzickými jadrami a ôsmymi vláknami (hyper threading),
- základnou pracovnou frekvenciou jadier na úrovni 3,4GHz a turbo frekvenciou dosahujúcou až 3,8GHz,
- L3 cache o veľkosti 8MB.

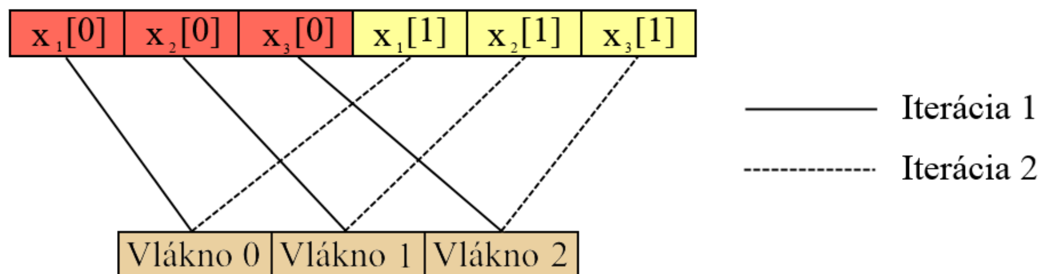
4.4 Implementácia

Konkrétna verzia metódy časticovej optimalizácie zvolená pre potreby paralelizácie na GPU vychádza z algoritmických krokov popísaných v kapitole 2.3.1, z tejto kapitoly je prebraté aj značenie a význam symbolov, ktoré budú po zvyšok textu tejto kapitoly slúžiť na označovanie príslušných dátových štruktúr. Celý výpočet sa odohráva na GPU, a preto je logickým prvým krokom alokovanie príslušných dátových štruktúr na GPU. Za predpokladu, že počet dimenzií je `n` a počet častíc je `pCount` sa jedná o:

- pole pozícií častíc, premenná `x`, veľkosť `[n * pCount]`,
- pole aktuálnych hodnôt účelovej funkcie vyhodnotenej nad pozíciami častíc v poli `x`, premenná `y`, veľkosť `[pCount]`,
- pole vektorov rýchlosti častíc, premenná `v`, veľkosť `[n * pCount]`,

- pole pozícií kde častice dosiahli svoje doterajšie minimum, premenná p, veľkosť [n * pCount],
- pole najnižších hodnôt účelovej funkcie vyhodnotenej nad pozíciami častíc v poli p, premenná pbest, veľkosť [pCount].

Polia x, v a p v skutočnosti neobsahujú nezávislé hodnoty, ale súradnice bodov v priestore, s ktorými by malo byť zachádzané ako so štruktúrami. V typickom sériovom návrhu v objektovom jazyku z rodiny C/C++ by zrejme skutočne tieto polia obsahovali pCount štruktúr o veľkosti n. To by však spôsobilo problém pri prirodzenej paralelnej verzii časticovej optimalizácie, kedy je jedno vlákno zodpovedné za vykonanie jedného kroku algoritmu nad jednou časticou. V takom prípade by totiž prístupy z jednotlivých vlákien do globálnej pamäte neboli zlúčené, ako popisuje kapitola 3.3.3 a znázorňuje Obrázok 8. To by viedlo k značnému spomaleniu prístupov k týmto poliam. Z tohto dôvodu sú pozície častíc uložené v globálnej pamäti ako štruktúra polí, teda po jednotlivých dimenziách, čo ilustruje Obrázok 10. Výnimku z tohto pravidla tvorí špeciálna verzia paralelizácie výpočtu účelovej funkcie, o ktorej pojednáva kapitola 4.5.2.



Obrázok 10: Štruktúra polí obsahujúca pozície častíc pre počet dimenzií 2 a počet častíc 3, zlučovaný prístup je v prvom kroku značený plnou čiarou, v druhom kroku prerušovanou

Po alokovaní pamäte potrebnej na výpočet je nutná jej inicializácia hodnotami. Na generovanie hodnôt v poliach x a v je použitá knižnica cuRAND [42]:

- V závislosti od použitej presnosti desatinných čísel je použitá buď funkcia curandGenerateUniform alebo curandGenerateUniformDouble.
- Obe funkcie generujú náhodné čísla s rovnomerným pravdepodobnostným rozdelením, ako parameter prijímajú ukazovateľ na polia x a v.

Hodnoty v poli p sú následne inicializované skopírovaním hodnôt z poľa x. Kopírovanie pamäte na GPU stručne spomína kapitola 3.4.3.

Po inicializácii už nasleduje samotná iteratívna časť algoritmu, ktorú je možné rozdeliť na niekoľko nezávislých menších celkov:

1. Vyhodnotenie účelovej funkcie $f(x_i)$ pre všetky častice,
2. Aktualizácia lokálnych miním častíc `pbest` a príslušných pozícií `p`,
3. Aktualizácia indexu častice, ktorá dosiahla v doterajšom výpočte globálne minimum,
4. Presun častíc do nových pozícií aplikovaním vektorov rýchlosti `v`.

Tieto celky a ich implementácia sú detailne popísané vo zvyšku tejto kapitoly, vyhodnocovaniu účelovej funkcie je následne venovaná celá kapitola 4.5, nakoľko sa jedná o ten najdôležitejší a najzložitejší krok výpočtu.

Komunikácia medzi krokmi prebieha pomocou polí `x`, `y`, `v`, `p` a `pbest` uloženými v globálnej pamäti. Spomenuté premenné v skutočnosti obsahujú len ukazovatele na príslušné polia alokované v pamäti GPU a sú posielané ako parametre metódam implementujúcim jednotlivé kroky popísané vyššie. Po ukončení iteratívnej časti výpočtu nasleduje kopírovanie dosiahnutých výsledkov do pamäte CPU. Opäť sa jedná o spôsob kopírovania pamäte GPU popísaný v kapitole 3.4.3.

Je vhodné spomenúť, že kopírovanie je za určitých podmienok nutné aj za účelom inicializácie účelovej funkcie. Napríklad funkcia f_5, ktorú definuje Tabuľka 1, je závislá na externom konfiguračnom súbore uloženom na pevnom disku. Tento problém sa rieši tak, že dáta sú najskôr štandardne načítané do dátových štruktúr pamäte CPU, odkiaľ sú následne skopírované do pamäte GPU.

4.4.1 Aktualizácia lokálnych miním

Aktualizácia lokálnych miním častíc pri výpočte časticovej optimalizácie spočíva v aktualizácii dvoch polí:

- pole pozícií kde častice dosiahli svoje doterajšie minimum, `p`,

- pole najnižších hodnôt účelovej funkcie vyhodnotenej nad pozíciami častíc v poli `p`, `pbest`.

Keďže aktualizácia môže prebiehať pre všetky častice súčasne, tak je prirodzenou voľbou paralelizácia tejto úlohy tak, že jedno vlákno aktualizuje obe polia pre jednu časticu. Ostatné použité premenné sú:

- pole aktuálnych pozícií častíc `x`,
- pole aktuálnych dosiahnutých hodnôt častíc `y`,
- id vlákna, ktoré ho jednoznačne identifikuje v celej mriežke spusteného kernelu je `tId`, výpočet hodnoty tejto premennej prebieha pomocou štruktúr identifikujúcich bloky a vlákna v mriežke, ich popis uvádza kapitola 3.4.4.

Kernel na aktualizáciu lokálnych miním najskôr načíta práve dosiahnutú hodnotu účelovej funkcie pre časticu s indexom `tId` a ak je táto nižšia ako najlepšia dosiahnutá hodnota, tak aktualizuje hodnoty v oboch poliach `p` a `pbest`. Kód vyzerá nasledovne:

1. `tId = threadIdx.x + (blockDim.x * blockIdx.x)`,
2. Ak `(y[tId]< pbest[tId])` tak
   a. `pbest[tId] = y[tId]`,
   b. Pre `dim = 0, dim < n, ++dim`
      i. p_{tId}`[dim] = `x_{tId}`[dim]`.

Polia `x` a `p` majú rovnakú štruktúru, ktorá medzi nimi umožňuje jednoduché kopírovanie pozícií častíc. Obe polia obsahujú dáta vo formáte štruktúra polí, čo je popísané v úvode kapitoly 4.4, pričom vďaka tomuto jednoduchému konceptu sú všetky pamäťové prístupy z kernelu na aktualizáciu lokálnych miním zlučované.

4.4.2 Aktualizácia globálneho minima

Aktualizácia globálneho minima tvorí dôležitý krok výpočtu časticovej optimalizácie, ktorý sa svojou implementáciou na GPU navyše značne líši od typickej verzie pre CPU. Podstata úlohy je napriek tomu jednoduchá, na vstupe sa nachádza pole najlepších dosiahnutých hodnôt účelovej funkcie pre všetky častice,

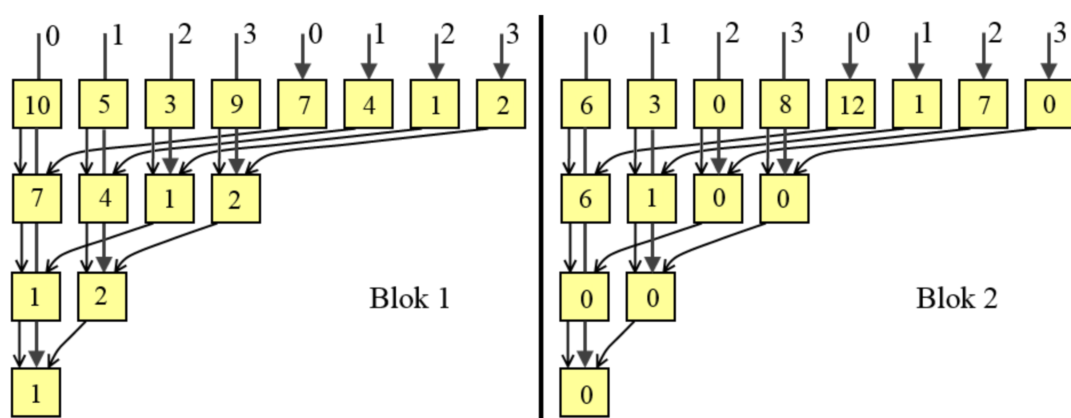
v ktorom je potrebné nájsť tú najnižšiu hodnotu. Podľa toho, na ktorej pozícii sa táto hodnota nachádza je následne možné identifikovať aj časticu, ktorá ju dosiahla.

Samotná implementácia tohto kroku pre GPU sa očividne líši od teoretického návrhu algoritmu popísaného v kapitole 2.3.1, čo vyplýva najmä z toho, že vyhodnocovanie účelovej funkcie prebieha pre všetky častice súčasne. Kvôli tomu by bez drahej synchronizácie medzi vláknami nebolo možné priebežne aktualizovať jednak lokálne minimá a vo výsledku ani globálne minimum.

4.4.2.1 Redukcia na úrovni bloku

Základom pre implementáciu je algoritmus nazývaný redukcia, ktorého konkrétna podoba závisí od použitého binárneho operátora. V prípade časticovej optimalizácie sa jedná o operátor minimum, ale redukcia môže prebiehať za použitia akéhokoľvek asociatívneho binárneho operátora. V prípade minima nájde algoritmus redukcie jednoducho vo vstupnom poli hodnôt tú najnižšiu.

Verzia redukcie implementovaná v tejto kapitole pre GPU spočíva v tom, že vstupné pole sa najskôr zredukuje po častiach a následne sa rekurzívne redukujú tieto čiastkové výsledky. Na redukovanie častí je použitá práve redukcia na úrovni bloku.



Obrázok 11: Zjednodušený pohľad na krok algoritmu redukcie operátorom minimum nad 16 prvkami pomocou dvoch blokov, každý o veľkosti 4 vlákien, vlákna značené plnou šípkou, indexované hodnotami 0, 1, 2, 3

Redukcia na GPU navrhnutá v tejto kapitole je realizovaná po častiach:

1. Zredukujú sa časti vstupu o dvojnásobnej veľkosti použitých blokov, čo poskytne čiastkové výsledky. Postup pre dva bloky ilustruje aj Obrázok 11.
2. Ak bolo použitých viac blokov ako jeden, tak výsledok ešte nie je k dispozícii a redukcia sa znova spustí na výsledkoch z jednotlivých blokov.
3. Kroky 1 a 2 sa opakujú, až kým nie je spustený jediný blok, ktorý už dokáže poskytnúť finálny výsledok.

Redukcia počas svojho výpočtu používa nasledujúce premenné:

- pole vstupných hodnôt `y`,
- pole výstupných hodnôt, `y_out`, o veľkosti počtu blokov spusteného kernelu, ktoré je potrebné v prípade rekurzívneho spúšťania kernelu,
- pole `y_shared` o veľkosti počtu vlákien v bloku uložené v zdieľanej pamäti, slúžiace na zdieľanie čiastkových výsledkov medzi vláknami,
- premennú `id`, ktorá na základe identifikátorov blokov a vlákien v mriežke popísaných v kapitole 3.4.4 určuje úsek vstupu, na ktorého redukcii sa podieľa aktuálne vlákno v závislosti aj od toho, v ktorom bloku spusteného kernelu sa nachádza.

Zjednodušený kód výpočtu redukcie pre každý zo spustených blokov vyzerá nasledovne:

1. `id = blockIdx.x * (blockDim.x * 2) + threadIdx.x`,
2. `y_shared[threadIdx.x] = min(y[id], y[id + blockDim.x])`,
3. `__syncthreads()`,
4. Pre `stride = blockDim.x / 2, stride > 0, stride /= 2`
   a. Ak `(threadIdx.x < stride)` tak
      i. `y_shared[threadIdx.x] = min(`
         `y_shared[threadIdx.x],`
         `y_shared[threadIdx.x + stride])`,
   b. `__syncthreads()`,
5. `Ak (threadIdx.x == 0) tak`
   a. `y_out[blockIdx.x] = y_shared[threadIdx.x]`.

Synchronizácia vlákien v krokoch 3 a 4.b je nutná, pretože zaručuje, že po dosiahnutí bariéry všetky vlákna z bloku vykonali všetky predchádzajúce zápisy do zdieľanej pamäte, teda do premennej `y_out`.

Popísaný kernel ale nerieši úlohu z úvodu tejto kapitoly, t.j. nájdenie indexu prvku s minimálnou hodnotou. To je možné dosiahnuť modifikáciou:

- Kernel bude prijímať na vstupe nielen pole hodnôt, ale aj pole ich indexov, `idx`, ktoré majú jednotlivé hodnoty vo vstupnom poli.
- Pri prvej iterácii (`itr`) redukcie je pole `idx` prázdne a namiesto neho sa priamo používajú indexy hodnôt vo vstupnom poli, ktoré sú ešte známe.
- Pole `idx` má tak, ako pole `y` svoj náprotivok v podobe poľa `idx_shared` uloženého v zdieľanej pamäti, ktoré slúži na komunikáciu medzi vláknami.
- Vždy, ak sa hodnota `y_shared[threadIdx.x]` v krokoch algoritmu 2 a 4.a.i. zmení, tak sa do `idx_shared[threadIdx.x]` zapíše hodnota `idx_shared[threadIdx.x + stride]`.

Kernel po adaptovaní sa na vyššie popísané zmeny vyzerá nasledovne, krok 2 bol rozdelený medzi kroky 2, 3, 4, rovnaký princíp na aktualizáciu poľa indexov je potrebné uplatniť aj pre pôvodný krok 4.a.i., teraz krok 6.a.i.:

1. `id = blockIdx.x * (blockDim.x * 2) + threadIdx.x`,
2. `y_shared[threadIdx.x] = y[id]`,
3. `idx_shared[threadIdx.x] = (itr == 0) ? id : idx[id]`,
4. Ak `(y[id + blockDim.x] < y_shared[threadIdx.x])` tak
    a. `y_shared[threadIdx.x] = y[id + blockDim.x]`,
    b. `idx_shared[threadIdx.x] = (itr == 0)`
       `? id + blockDim.x : idx[id + blockDim.x]`,
5. `__syncthreads()`,
6. Pre `stride = blockDim.x / 2, stride > 0, stride /= 2`
    a. Ak `(threadIdx.x < stride)` tak `...`
7. Ak `(threadIdx.x == 0)` tak
    a. `y_out[blockIdx.x] = y_shared[threadIdx.x]`,
    b. `idx[blockIdx.x] = idx_shared[threadIdx.x]`.

Ako je možné vidieť, pole `idx` slúži aj na čítanie aj na zápis, takže nie je potrebné udržovať aj pole `idx_out` ako v prípade poľa `y` a `y_out`. Je to vďaka tomu, že pole indexov je len umelo vytvorené pre potreby redukcie a zmeny v ňom neovplyvnia ďalšie kroky algoritmu časticovej optimalizácie. Navyše sú z neho všetky hodnoty prečítané skôr, ako sa doňho zapisuje v poslednom kroku.

Nevýhodou uvedenej implementácie je, že kvôli podmienke v kroku 6.a zostáva veľký počet vlákien bloku neaktívnych. Túto situáciu je možné značne vylepšiť od hodnoty `(stride == 32)` a menej, pretože vtedy sú v bloku aktívne len vlákna z prvého warpu. Vďaka tomu je možné for cyklus z kroku 6.a ukončiť už vo chvíli, kedy je premenná `stride` rovná veľkosti warpu a zvyšné kroky výpočtu nahradiť jednoduchšou verziou redukcie na úrovni warpu, ktorú popisuje nasledujúca kapitola.

V tejto práci bola vyvinutá a testovaná aj verzia redukcie, kedy warpy už od začiatku redukujú časť vstupu prislúchajúci bloku po menších úsekoch. Každý warp by následne čiastkový výsledok zapísal do zdieľanej pamäte, kde by sa o finálnu redukciu postaral prvý warp. Modifikovaný prístup dosahoval takmer totožné časy s verziou predstavenou v tejto kapitole, a preto detaily implementácie nie sú uvedené.

4.4.2.2 Redukcia na úrovni warpu

Vlákna vo warpe vykonávajú všetky inštrukcie súčasne. Túto ich vlastnosť je pre GPU od compute capability 3.0 možné využívať pomocou rozšírenia o špeciálne, takzvané warpové inštrukcie. Do tejto skupiny patria aj inštrukcie `__shfl`, o ktorých pojednáva kapitola 3.4.5 v časti „Komunikácia warpovými inštrukciami“.

Použitím warpových inštrukcií je možné dosiahnuť rýchlejšiu redukciu, ako použitím tradičného prístupu cez zdieľanú pamäť popísaného v predchádzajúcej kapitole. Uvedená verzia redukcie na úrovni warpu pracuje podobne, ako redukcia na úrovni bloku popísaná v predchádzajúcej kapitole. Je teda schopná redukovať počet prvkov maximálne o veľkosti dvojnásobku počtu vlákien vo warpe.

So značením premenných prebratým z predchádzajúcej kapitoly by výpočet kernelu redukujúceho pole na úrovni warpu vyzeral nasledovne, `warpSize` predstavuje veľkosť warpu:

1. `result = min(y[threadIdx.x],y[threadIdx.x + warpSize])`,
2. Pre `stride = warpSize / 2, stride > 0, stride /= 2`
    a. `result = min(result, __shfl_xor(result, stride))`.

Inštrukcia `__shfl_xor` umožňuje vymieňať hodnoty lokálnych premenných medzi aktívnymi vláknami jedného warpu. Inštrukcia v uvedenom príklade vytvára v každom kroku páry vlákien, ktoré si tieto hodnoty vymieňajú, pričom pracuje podobne, ako výpočet redukcie, ktorý ilustruje Obrázok 11, teda napríklad pre `(warpSize == 32)` by pre:

- `(stride == 16)` vo vlákne s `threadIdx.x` postupne rovným 0, 1, ..., 31 vrátila hodnotu premennej `result` z vlákien postupne 16, 17, ..., 0,
- `(stride == 8)` vo vlákne s `threadIdx.x` postupne rovným 0, 1, ..., 31 vrátila hodnotu premennej `result` z vlákien postupne 8, 9, ..., 23,
- `(stride == 4)` vytvorí páry (0, 4), (1, 5), (2, 6), (3, 7), ..., (27, 31) a tak ďalej pre nižšie hodnoty `stride`.

Pre jednoduchosť a prehľadnosť kódu je znova uvedená len verzia, ktorá nájde minimálny prvok v poli a nie jeho index. Po skončení výpočtu sa minimálny prvok nachádza v premennej `result` vo vlákne, kde platí `(threadIdx == 0)`. Nájdenie indexu by bolo možné docieliť rovnakými jednoduchými úpravami ako pre verziu redukcie na úrovni bloku z predchádzajúcej kapitoly.

Problém tohto prístupu spočíva v tom, že __shfl inštrukcie sú dostupné len pre GPU od compute capability 3.0. Riešenie problému pre staršie architektúry ako Fermi popisuje Harris [53], ktorý okrem toho rozoberá aj niekoľko ďalších potenciálnych vylepšení výkonu algoritmu redukcie na GPU použitím technológie CUDA. Jeho prístup k redukcii na úrovni warpu spočíva v použití poľa `vol`, o veľkosti `warpSize` v zdieľanej pamäti označeného kľúčovým slovom `volatile`, teda

všetky zápisy do tohto poľa sú vykonané okamžite. Po aplikovaní ním navrhnutého postupu sa kód pre redukciu na úrovni warpu zmení nasledovne:

1. `vol[threadIdx.x] = result = min(y[threadIdx.x], y[threadIdx.x + warpSize]),`
2. Pre `stride = warpSize / 2, stride > 0, stride /= 2`
    a. `vol[threadIdx.x] = result = min(result, vol[threadIdx.x ^ stride]).`

	16384	**65536**	**262144**	**1048576**
SHARED	12 µs	27 µs	79 µs	281 µs
SHFL	8 µs	18 µs	51 µs	177 µs
VOL	9 µs	20 µs	54 µs	189 µs

Tabuľka 3: Porovnanie času výpočtu redukcie na úrovni bloku pre rôzne verzie na rôzne veľkých vstupoch, použitá GPU: GTX660Ti

Tabuľka 3 ukazuje porovnanie výkonu pre tri verzie redukcie na úrovni bloku:

- SHARED je verzia popísaná v predchádzajúcej kapitole bez špeciálneho ošetrovania redukcie na úrovni warpu.
- SHFL je verzia s redukciou na úrovni warpu vyriešenou pomocou warpových inštrukcií `__shfl_xor`.
- VOL je verzia s redukciou na úrovni warpu vyriešenou pomocou zdieľanej pamäte označenej kľúčovým slovom `volatile`.

Z prezentovaných čísel je jasne vidieť, že najlepšie výsledky podáva verzia SHFL, ktorá dosahuje ešte mierne lepšie časy ako verzia VOL, ktorá používa volatilnú pamäť. Rozdiel v časoch pre tieto dve verzie je však pomerne malý a z tohto dôvodu ani staršie architektúry ako Fermi nebudú citeľne znevýhodnené. Z tabuľky je tiež vidieť, že špeciálne ošetrenie redukcie na úrovni warpu prináša v oboch verziách SHFL aj VOL časovú úsporu vo výpočtovom čase pre redukciu na úrovni bloku oproti pôvodnej verzii SHARED.

4.4.3 Aktualizácia pozície a rýchlosti častíc

Proces aktualizácie pozície a vektoru rýchlosti častíc je opäť veľmi jednoduchý a keďže je možné ho vykonávať bez problémov súčasne pre všetky častice, tak jeho paralelizácia pomocou GPU je priamočiara, jedno vlákno aktualizuje hodnoty pre jednu časticu.

Celý proces aktualizácie kopíruje kroky popísané v kapitole 2.3.1, nový vektor rýchlosti vzniká použitím vzorca (2.4), pozícia sa aktualizuje pripočítaním aktualizovaného vektora rýchlosti k pôvodnej pozícii po jednotlivých dimenziách. Kernel implementovaný za účelom aktualizácie používa nasledovné premenné:

- pole aktuálnych pozícií častíc `x`,
- pole pozícií dosiahnutých lokálnych miním častíc `p`,
- pole aktuálnych vektorových rýchlostí častíc `v`,
- index častice, ktorá dosiahla globálne minimum, `gId`, získaný pomocou aktualizácie globálneho minima popísanej v predchádzajúcej kapitole,
- konštanty ω, φ_p a φ_g, význam a hodnoty sú rovnaké ako v kapitole 2.3.1,
- id vlákna, ktoré ho jednoznačne identifikuje v celej mriežke spusteného kernelu je `tId`, výpočet hodnoty tejto premennej prebieha pomocou štruktúr identifikujúcich bloky a vlákna v mriežke, ich popis uvádza kapitola 3.4.4.

Pseudokód kernelu používaného na účely aktualizácie pozícií a vektorov rýchlosti vyzerá nasledovne:

1. `tId = threadIdx.x + (blockDim.x * blockIdx.x)`,
2. Pre `dim = 0, dim < n, ++dim`
    a. $v_{tId}[dim] =$
       $(\omega * v_{tId}[dim]) +$
       $(\varphi_p * rand() * (p_{tId}[dim] - x_{tId}[dim])) +$
       $(\varphi_g * rand() * (p_{gId}[dim] - x_{tId}[dim]))$,
    b. $x_{tId}[dim] = x_{tId}[dim] + v_{tId}[dim]$.

Tento kernel demonštruje aj to, že ak je index častice, ktorá dosiahla doterajšie globálne minimum známy, v tomto prípade premenná `gId`, tak nie je potrebné si

v pamäti explicitne udržiavať pozíciu tejto častice, ktorá je v kapitole 2.3.1 označovaná ako `g`. Z podstaty uvedeného kódu vyplýva, že všetky vlákna vo warpe budú v prípade prístupu k p_{gId}`[dim]` čítať túto hodnotu z tej istej pozície v pamäti, takže nie je nutné dbať na akékoľvek zlučovanie pamäťových prístupov.

Jediným problémom, ktorý je potrebné pre uvedený pseudokód vyriešiť je generovanie náhodných čísel z intervalu $[0,1]$, čo je úlohou funkcie `rand()`. Ako bolo spomenuté aj v úvode kapitoly 4.1, spoločnosť NVIDIA pripravila za týmto účelom knižnicu cuRAND [42], ktorá poskytuje rozhranie na generovanie náhodných čísel aj v kóde kernelu. Funkcie použité na tento účel sú:

- `curand_uniform_double` pri použití desatinných čísel s presnosťou 64 bitov,
- `curand_uniform` pri použití nižšej, 32 bitovej presnosti.

Obe vyššie spomenuté funkcie prijímajú jediný parameter, ktorým je premenná typu `curandState`. Inicializácia tejto premennej je pomerne drahá, a preto boli v tejto práci implementované jej dve rôzne verzie:

- Vo verzii GLOB sa na začiatku výpočtu časticovej optimalizácie vytvorí v globálnej pamäti GPU pole s položkami typu `curandState` o veľkosti počtu použitých častíc. Toto pole je následne posielané ako parameter kernelu zodpovedného za aktualizáciu pozícií a vektorov rýchlosti. Každé vlákno si potom za účelom generovania náhodných čísel prečíta z tohto poľa jeden objekt typu `curandState` na pozícii `tId`.
- Vo verzii REG si pri každom spustení kernelu aktualizujúceho pozície a vektory rýchlosti vytvorí každé jeho vlákno svoj vlastný objekt typu `curandState`, ktorý následne použije na generovanie náhodných čísel. Objekt je teda uložený v registroch daného vlákna. Objekty sa vždy vytvoria s iným seedom, napríklad použitím funkcie `clock()` v kombinácii s premennou `tId`.

Evidentnou výhodou prístupu REG je fakt, že nezaberá zbytočne miesto v globálnej pamäti, ktoré by inak mohlo byť využité za účelom zvýšenia celkového počtu použitých častíc a podobne. Objekty typu `curandState` totiž zaberajú až 48B.

	65536
GLOB	1589 µs
REG	1551 µs

Tabuľka 4: Porovnanie času výpočtu aktualizácie pozícií a vektorov rýchlosti častíc pre rôzne verzie pre vstup o veľkosti 65536 častíc, použitá GPU: GTX660Ti

Tabuľka 4 uvádza porovnanie dosiahnutých časov týchto dvoch verzií, z ktorého jasne vyplýva, že verzia REG je mierne rýchlejšia, takže inicializácia objektu typu `curandState` vo vlákne je rýchlejšia, ako prečítanie 48B dát z globálnej pamäte. Z tohto dôvodu je vo výslednej práci na aktualizáciu pozícií a vektorov rýchlosti častíc použitá verzia REG.

Ako verzia REG, tak aj verzia GLOB spôsobujú za určitých podmienok problém. Ak sa pri spustení kernelu použije maximálna povolená veľkosť blokov, pre Fermi a Kepler je to 1024 [30], tak to vyústi do veľmi vysokého počtu registrov, ktoré každý blok využíva, najmä pre veľkosť objektov typu `curandState`. Architektúra Kepler túto situáciu dokáže zvládnuť vďaka vysokému počtu 32b registrov, ktoré má každé SM k dispozícii, konkrétne 65536 [30]. Naproti tomu Fermi disponuje len polovičným počtom registrov. Jednoduchým riešením problému, ktoré bude mať minimálny dopad na výkon je v prípade tohto kernelu používať bloky napríklad o polovičnej veľkosti, teda o počte 512 vlákien.

4.5 Vyhodnocovanie účelovej funkcie

Vyhodnocovanie účelovej funkcie je najdôležitejším krokom výpočtu, a to preto, že typicky zaberá najviac z celkového času výpočtu časticovej optimalizácie. Jeho paralelizácii je venovaný najväčší priestor, pričom väčšina princípov uplatňovaných pri vyhodnocovaní je závislá od konkrétnej účelovej funkcie, jej výpočtovej zložitosti a podobne. Táto kapitola, ako aj celá práca je preto venovaná optimalizácii obmedzeného setu účelových funkcií, ktoré definuje Tabuľka 1 v kapitole 4.2.

4.5.1 Vyhodnocovanie častice vláknom

Prirodzeným spôsobom paralelizácie výpočtu účelovej funkcie je každej častici priradiť vlastné vlákno, ktoré pre jej súčasnú pozíciu vyhodnotí účelovú funkciu. Jedným zo spôsobov, ako pristúpiť k vyhodnoteniu účelovej funkcie pre časticu je použiť kernel so zhruba nasledujúcimi krokmi:

1. Vlákno načíta z globálnej pamäte do registrov pozíciu jemu priradenej častice. Vďaka usporiadaniu pozícii častíc v pamäti popísanému v kapitole 4.3 prebieha tento krok po jednotlivých dimenziách, napríklad vo for cykle, ktorého prácu fakticky ilustruje aj Obrázok 10.
2. Účelová funkcia sa vyhodnotí na pozícii načítanej z globálnej pamäte.
3. Hodnota účelovej funkcie sa zapíše naspäť do globálnej pamäte.

Pre funkcie, ktorých vyhodnocovaniu je venovaná táto práca a definuje ich Tabuľka 1 je jednoduché tento prístup výrazne vylepšiť. Vďaka tvaru sumy je možné príspevky k výsledku v jednotlivých dimenziách pre tieto funkcie vyhodnocovať priebežne a takto šetriť množstvo registrov potrebných na udržovanie častice v pamäti. Na sčítavanie priebežných výsledkov sa zavedie premenná akumulátor (`acc`) a algoritmické kroky 1 a 2 sa ocitnú v tele jedného for cyklu. Modifikovaný kód by konkrétne pre funkciu f_4 (počíta sumu druhých mocnín vo všetkých dimenziách) vyzeral nasledovne, pre ostatné testované funkcie sa mení len krok 2.b:

1. `acc = 0`,
2. Pre `dim = 0, dim < n, ++dim`
    a. Do premennej `pos` načítaj pozíciu častice v dimenzii `dim`,
    b. `acc += pos * pos`.
3. Zapíš hodnotu uloženú v `acc` do globálnej pamäte.

Táto verzia prístupu k paralelizácii časticovej optimalizácie, po zvyšok kapitoly bude nazývaná THREAD, má niekoľko výhod:

- Je jednoduchý na implementáciu a nevyžaduje si žiadnu synchronizáciu medzi vláknami bloku ani mriežky a ani použitie zdieľanej pamäte.
- Je obzvlášť výhodný pre výpočtovo menej náročné účelové funkcie tvaru sumy, čo vyplýva napríklad z ich nízkej dimenzionality. Príkladom buď funkcia f_4 pre nižší počet dimenzií, ako napríklad $n = 2$, ktorú definuje

Tabuľka 1. Kernel totiž podáva najlepší výkon, ak je práca vlákien výpočtovo náročná, to ale nie je prípad výpočtu dvoch druhých mocnín vo funkcii f_4 pre $n = 2$. Ak je už samotná funkcia príliš jednoduchá na vyhodnotenie, tak nemá zmysel, a niekedy to ani nie je možné, jej ďalšie vyhodnocovanie pre jednu časticu rozdeľovať medzi viacero vlákien.

Naopak, použitie verzie THREAD nie je vhodné v prípade nízkeho počtu častíc v kombinácii s výpočtovo náročnou účelovou funkciou. Pre tento prípad existuje niekoľko základných pravidiel:

- Ak nie je počet častíc aspoň niekoľkonásobne vyšší ako počet fyzických jadier GPU, tak by kernel spustený za účelom vyhodnotenia účelovej funkcie pre všetky častice za žiadnych podmienok nedokázal naplno vyťažiť všetky jadrá, ktoré GPU ponúka. Preto je pre výpočtovo náročné účelové funkcie vhodné použiť iný spôsob paralelizácie.
- Ak je počet dimenzií vstupného argumentu účelovej funkcie aspoň o veľkosti warpu na GPU a príspevky argumentu k výsledku sú v jednotlivých dimenziách nezávislé, tak to signalizuje možnosť paralelizácie vyhodnotenia účelovej funkcie pre časticu aspoň warpom vlákien.
- Funkcie parametrizované nielen počtom dimenzií, ako je napríklad funkcia f_5 z Tabuľka 1, signalizujú aj iné možnosti paralelizácie aplikovateľné aj v prípade nízkeho počtu použitých častíc alebo dimenzií.

4.5.2 Paralelizácia vysoko dimenzionálnych funkcií

Z predchádzajúcej kapitoly vyplýva, že pre výpočtovo náročnejšie účelové funkcie je často vhodné vyhodnocovanie účelovej funkcie rozdeliť medzi viacero vlákien, napríklad po jednotlivých dimenziách, ako to popisuje aj Mussi et al. [43]. Tento prístup nie je použiteľný v prípade závislých členov argumentu účelovej funkcie, kedy nie je možné ich nezávislé vyhodnotenie, ale nachádza uplatnenie pre rôzne sumy, či polynómy. Medzi inými napríklad pre funkcie f_1 až f_4 popísané v Tabuľka 1. Aj napriek miernej závislosti členov vo funkcii f_2 je táto stále dobre paralelizovateľná, vďaka jednoduchému triku popísanému v nasledujúcej časti.

Koncept paralelizácie vyhodnocovania vysoko dimenzionálnej účelovej funkcie pre jednu časticu warpom, alebo celým blokom vlákien je pomerne jednoduchý. Kroky algoritmu sú v oboch prípadoch tiež takmer totožné, preto je v nasledujúcom texte vysvetlená len verzia s použitím bloku, pozvyšok kapitoly nazývaná BLOCK. Nech:

- pole aktuálnych pozícií častíc je `x`,
- pole aktuálnych dosiahnutých hodnôt častíc `y`,
- počet vlákien v bloku je `blockDim.x` (viď kapitola 3.4.4),
- jednoznačné id bloku v mriežke je `blockIdx.x` (viď kapitola 3.4.4),
- jednoznačné id vlákna v bloku je `threadIdx.x` (viď kapitola 3.4.4).

Potom by kernel na vyhodnotenie účelovej funkcie f_4 blokom vlákien mohol vyzerať nasledovne:

1. `acc = 0`,
2. Pre `dim = threadIdx.x, dim < n, dim += blockDim.x`
    a. `pos = x`$_{\texttt{blockIdx.x}}$`[dim]`,
    b. `acc += pos * pos`,
3. Vytvor sumu `sum` všetkých premenných `acc` z vlákien bloku,
4. Ak `(threadIdx.x == 0)` tak `y[blockIdx.x] = sum`.

Tretí krok kernelu je netriviálny, jedná sa o redukciu na úrovni bloku vlákien, ktorú detailne popisuje kapitola 4.4.2.1. Jediným rozdielom oproti redukcii popísanej v spomenutej kapitole je, že namiesto operácie minimum sa v tomto prípade použije operácia sčítania za účelom vytvorenia sumy. Jedná sa o asociatívny binárny operátor, takže použitie redukcie je v tomto prípade možné.

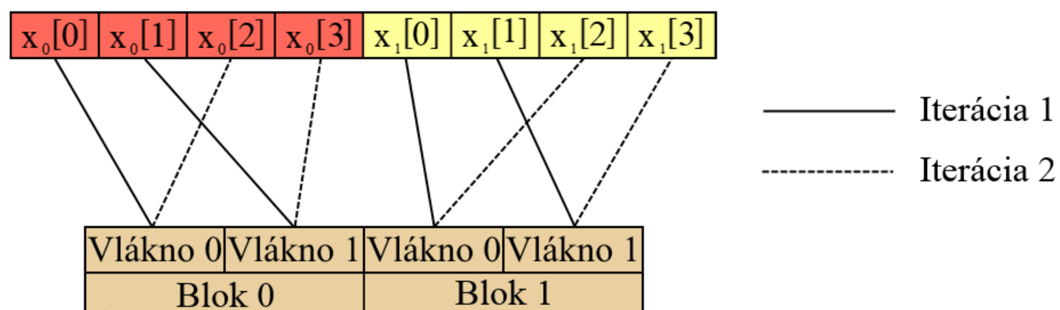
Prístup sa môže zdať nevýhodný pre funkcie typu f_2, ktoré obsahujú závislé členy ako $x_i x_{i+1}$, pretože by každé vlákno muselo dvakrát čítať z globálnej pamäte. Riešenia sú nasledovné:

- V starších GPU architektúrach ako Fermi sa podľa kapitoly 3.3.2 prístupy do globálnej pamäte cachujú v L1 cachi. Preto by vlákno zrejme našlo hodnotu x_{i+1} v L1 cachi, nakoľko ju v momente, keď vlákno čítalo hodnotu x_i, prečítalo susedné vlákno, ktoré operuje v dimenzii $i + 1$. Výnimku z tohto

pravidla tvoria hranice warpov, pretože inštrukcie nie sú vykonávané súčasne vo všetkých warpoch bloku. Posledné vlákno vo warpe by teda hodnotu x_{i+1} v L1 cachi nájsť nemuselo.

- V novšej architektúre Kepler sa za týmto účelom môže oplatiť použiť zdieľanú pamäť. Po kroku 2.a by každé vlákno uložilo načítanú hodnotu `pos`, čo je x_i, do zdieľanej pamäte. Nasledovala by synchronizácia pomocou `__syncthreads` a potom by už vlákno určite našlo hodnotu x_{i+1} v zdieľanej pamäti. O tomto využití zdieľanej pamäte pojednáva aj kapitola 3.4.5 v časti „Komunikácia zdieľanou pamäťou“.
- Podľa kapitoly 3.3.2 je od compute capability 3.5 je možné vynútiť si cachovanie čítania z globálnej pamäti v špeciálnej L1 cache len pre čítanie.

Problém s popísanou verziou BLOCK je ten, že ak by pozície častíc boli uložené v globálnej pamäti podľa jednotlivých dimenzií, ako to popisuje kapitola 4.3, tak by znova globálne pamäťové prístupy neboli zlúčené. Vlákna z bloku by totiž nepristupovali na susediace adresy v pamäti, miesto toho by boli tieto adresy od seba vo vzdialenosti o počte vlákien v bloku.



Obrázok 12: Zjednodušená demonštrácia riešenia problému so zlučovaním prístupu do globálnej pamäti pre verziu BLOCK pre počet dimenzií 4 a počet vlákien v bloku 2

Problém vzniká, lebo vlákna bloku vyhodnocujúceho jednu časticu sa nachádzajú v rovnakých warpoch. Riešenie, s ktorým prišiel Mussi et al. [43] spočíva v tom, že pozície budú v pamäti uložené po jednotlivých časticiach a nie po dimenziách, teda ako v klasickej sériovej verzii časticovej optimalizácie pre CPU. Toto riešenie ilustruje aj Obrázok 12. Takýto prístup ale vedie k znižovaniu využitia GPU, lebo

ako už bolo spomenuté, počet dimenzií je pri jeho použití nutné zaokrúhliť nahor na najbližší násobok veľkosti warpu.

Kvôli spomenutým komplikáciám je v tejto práci predstavená modifikácia naivného prístupu, po zvyšok kapitoly označovaná ako MOD. Tá spočíva v tom, že vlákna vyhodnocujúce funkciu pre jednu časticu síce nebudú v jednom warpe, ale stále budú v jednom bloku. Podmienka vlákien v jednom bloku je dôležitá, inak by redukcia musela prebiehať na úrovni globálnej pamäte, čo by bolo drahšie. Nech:

- pole aktuálnych pozícií častíc je `x`,
- pole aktuálnych dosiahnutých hodnôt častíc `y`,
- počet vlákien vo warpe je `warpSize`, pre architektúry Fermi a Kepler platí `(warpSize == 32)`,
- id vlákna, ktoré ho jednoznačne identifikuje v rámci warpu je `laneId`,
- počet warpov v bloku je `warpCount`,
- id warpu, ktoré ho jednoznačne identifikuje v rámci bloku je `warpId`.

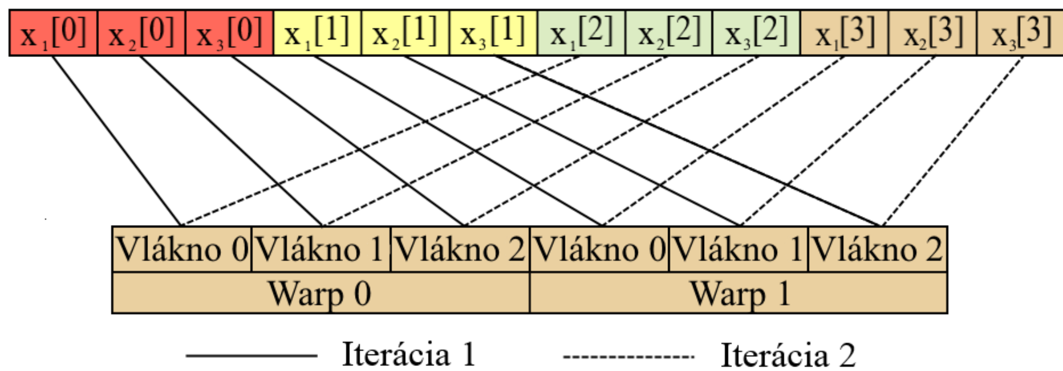
Modifikovaný kernel pre funkciu f_4, ktorý by nebránil zlučovaniu pamäťových prístupov vyzerá nasledovne:

1. `laneId = threadIdx.x % warpSize`,
2. `warpCount = blockDim.x / warpSize`,
3. `warpId = threadIdx.x / warpSize`,
4. `acc = 0`,
5. Pre `dim = warpId, dim < n, dim += warpCount`
    a. `pos =` x_{laneId}`[dim]`,
    b. `acc += pos * pos`,
6. Vytvor sumu `sum` všetkých premenných `acc` z vlákien bloku, ktoré majú rovnakú hodnotu `laneId`,
7. Ak `(warpId == 0)` tak `y[laneId] = sum`.

Krok 6 je znova možné realizovať pomocou redukcie na úrovni bloku podľa kapitoly 4.4.2.1, použitím operácie sčítania. Rozdiel je ten, že redukcia nedobehne až do konca, ale zastaví sa vtedy ak `(stride < 32 == warpSize)`. To je okamih,

kedy sú spočítané výsledky pre častice, ktoré vyhodnocoval daný blok. Tento fakt vyplýva z toho, že:

- Všetky vlákna v bloku s rovnakým `laneId` vyhodnocujú účelovú funkciu pre tú istú časticu, napríklad vlákna `{ threadIdx.x, threadIdx.x + warpSize, threadIdx.x + 2*warpSize, … }` vyhodnocujú časticu s indexom `threadIdx.x`.
- Redukcia sčítava postupne hodnoty z vlákien vzdialených od seba o počet pozícií o veľkosti mocniny 2. Veľkosť warpu je pre všetky známe architektúry GPU rovná 32, čo je 2^5 a teda až kým neplatí spomenutá podmienka `(stride < 32)`, tak sú sčítavanými hodnotami práve hodnoty z vlákien `{ threadIdx.x, threadIdx.x + warpSize, … }`.



Obrázok 13: Zlúčené pamäťové prístupy v modifikovanej verzii paralelizácie vyhodnocovania účelovej funkcie, MOD

Princíp modifikovaného prístupu ilustruje Obrázok 13, na ktorom je dobre vidieť, že:

- Vďaka inicializácii for cyklu `(dim = warpId)`, operujú vlákna prvého warpu nad prvou dimenziou a vlákna druhého warpu nad druhou.
- Vďaka kroku for cyklu `(dim += warpCount)` operujú vlákna z prvého warpu v ďalšej iterácii for cyklu nad treťou dimenziou atď., nakoľko počet warpov v tomto zjednodušenom príklade je 2.

Prístup má oproti verzii BLOCK nevýhodu v tom, že maximálny počet warpov na blok je limitovaný, momentálne architektúry podporujú 32 warpov v bloku. Z tohto dôvodu nie je riešenie z pohľadu počtu dimenzií ľubovoľne škálovateľné, naproti tomu dobre škáluje s rastúcim počtom častíc.

	(128, 128)	(256, 256)	(512, 512)	(64, 4096)	(131072, 256)
THREAD	18 µs	42 µs	87 µs	633 µs	2404 µs
BLOCK	4 µs	8 µs	27 µs	30 µs	2516 µs
MOD	4 µs	9 µs	30 µs	49 µs	2391 µs

Tabuľka 5: Časy vyhodnotenia funkcie f_4, údaje v zátvorkách predstavujú (počet častíc, počet dimenzií), použitá GPU: GTX660Ti

Tabuľka 5 obsahuje porovnanie troch prístupov k paralelizácii funkcie f_4, kde:

- THREAD predstavuje verziu kde jedno vlákno vyhodnocuje jednu časticu, ktorá bola bližšie popísaná v kapitole 4.5.1,
- BLOCK je verzia navrhnutá Mussim, kde časticu vyhodnocuje celý blok, ktorej popis sa nachádza skôr v tejto kapitole,
- MOD je modifikovaná verzia predstavená a popísaná tiež v tejto kapitole, ktorej základným rozdielom oproti verzii BLOCK je to, že blok vlákien vyhodnocuje účelovú funkciu pre toľko častíc, koľko vlákien obsahuje jeden warp.

Prínos testovania pre nižšie počty častíc a dimenzií je diskutabilný, nakoľko výpočtové časy boli v tých prípadoch také nízke, že by zrejme nemalo zmysel o paralelizácii danej funkcie na GPU vôbec uvažovať. Tabuľka 5 ukazuje niekoľko zaujímavých informácií:

- Verzia BLOCK a verzia MOD dosahujú v prvých troch prípadoch veľmi podobné výsledky. To je možné dosiahnuť testovaním rôznych konfigurácií pre veľkosti blokov a mriežok kernelu, vďaka čomu obe verzie dosahujú rovnaké využitie GPU rovnomerným vyťažovaním jej jadier.
- V prípade malého počtu častíc a vysokého počtu dimenzií je favoritom verzia BLOCK, keďže pre tak nízky počet častíc nedokáže verzia MOD vyťažiť všetky SM na GPU. Jednoducho spustí kernel s príliš nízkym počtom blokov. Takto vysoko dimenzionálne problémy sú ale značne netypické a metóda časticovej optimalizácie ich často nie je ani schopná účinne riešiť.
- Verzia THREAD trpí podobným nedostatkom, ako verzia MOD v predchádzajúcom bode. Až na prípad extrémne vysokého počtu častíc, čo

pri metóde časticovej optimalizácie nie je bežné, nedokáže na GPU spustiť dostatočne veľký počet blokov, ktorý by vyťažil všetky SM.

- Verzia BLOCK naopak mierne zaostáva pre najvyšší testovaný počet častíc. Dôvodom je zrejme príliš vysoký počet vlákien, ktoré táto verzia využíva a následná réžia potrebná na ich rozvrhovanie a podobne.
- V prípade extrémneho počtu použitých častíc počas vyhodnocovania je očakávaným výsledkom, že časy verzií THREAD, BLOCK a MOD sú veľmi podobné. Je to jednoducho preto, že pri takom objeme práce nehrá takmer žiadnu rolu spôsob, akým bude táto práca rozdelená, je jej dostatok na vyťaženie GPU pre všetky tri testované verzie.

Verzia MOD sa na základe testu zdá byť vhodným kompromisom, aj keď nedosiahla v bežných prípadoch lepšie výsledky ako verzia BLOCK. Vyhodnocovanie účelovej funkcie je navyše nutné brať ako súčasť väčšieho celku – metódy časticovej optimalizácie. V tomto dosahuje výraznú výhodu práve metóda MOD, pri ktorej je možné udržiavať pozície častíc vo formáte štruktúra polí. To je predpoklad pre kroky aktualizácie lokálnych miním, kapitola 4.4.1, a aktualizácie pozícií a vektorov rýchlosti, kapitola 4.4.3, aby dokázali zlučovať pamäťové prístupy do globálnej pamäte. V opačnom prípade, teda pri použití metódy BLOCK a formáte dát pole štruktúr, to možné nie je, čo by viedlo k zníženému výkonu týchto krokov.

4.5.3 Paralelizácia funkcií inými parametrami

Mnohé účelové funkcie majú na rozdiel od funkcií ako f_1 až f_4 vysokú výpočtovú náročnosť aj napriek nízkej dimenzionalite. Príkladom je funkcia f_5, ktorej zložitosť závisí nielen od počtu dimenzií, ale aj od parametru P závislého na externom konfiguračnom súbore. Preto je v podstate nemožné poskytnúť univerzálne pravidlá paralelizácie aplikovateľné na všetky účelové funkcie a v konečnom dôsledku je na programátorovi, aby správne určil potenciál pre paralelizáciu tej ktorej účelovej funkcie.

Táto práca je zameraná na príklady použitia funkcie f_5, pre ktoré parameter P dosahuje veľmi vysokých hodnôt, čo presne odpovedá verzii tejto funkcie označenej

ako f_{10}, teda $P = 6980011$. Počet dimenzií je typicky menší ako 15. Prirodzeným prístupom k paralelizácii tejto funkcie teda nie je paralelizovať jednotlivé dimenzie vyhodnocovania funkcie pre časticu, ale paralelizovať vonkajšiu sumu vo vzorci funkcie, ktorá je ohraničená parametrom P. Za týmto účelom sú implementované dve verzie, kde na vyhodnotenie sumy pre jednu časticu sa použije buď warp vlákien, alebo celý blok. Tieto verzie sú následne porovnávané s verziou vyhodnocovania THREAD, predstavenou v kapitole 4.5.1 a upravenej pre potreby funkcie f_5. Obe verzie majú s vyhodnocovaním vysoko dimenzionálnych funkcií popísaným v predchádzajúcej kapitole spoločné to, že tiež používajú premennú akumulátor, `acc`, na sčítavanie čiastkovým výsledkov.

Za účelom použitia warpu na vyhodnotenie účelovej funkcie f_5 je v tejto práci vytvorený kernel, ktorý využíva nasledovné premenné:

- pole aktuálnych pozícií častíc je `x`,
- pole aktuálnych dosiahnutých hodnôt častíc je `y`,
- polia `a` a `b` určujú tvar funkcie f_5, sú definované externým konfiguračným súborom, `a` obsahuje vektory dĺžky `n` a `b` je pole skalárnych hodnôt,
- počet vlákien vo warpe je `warpSize`, pre architektúry Fermi a Kepler platí `(warpSize == 32)`,
- id vlákna, ktoré ho jednoznačne identifikuje v rámci warpu je `laneId`,
- id warpu, ktoré ho jednoznačne identifikuje v rámci bloku je `warpId`.

S použitím týchto premenných vyzerá pseudokód kernelu pre verziu WARP_2 nasledovne:

1. `laneId = threadIdx.x % warpSize`,
2. `warpId = threadIdx.x / warpSize`,
3. `acc = 0`,
4. Pre `j = laneId, j < P, j += warpSize`
    a. `partialAcc = b[j]`,
    b. Pre `dim = 0, dim < n, ++dim`
        i. `partialAcc -= (`x_{warpId}`[dim] * `a_j`[dim])`,
    c. `acc += partialAcc * partialAcc`,

5. Vytvor sumu `sum` všetkých premenných `acc` v rámci každého warpu, teda z vlákien, ktoré majú rovnakú hodnotu `warpId`,
6. Ak `(laneId == 0)` tak `y[warpId] = sum`.

Podobne, ako tomu bolo pre vyhodnocovanie vysoko dimenzionálnych funkcií, krok 6 vyžaduje algoritmus redukcie. Tento krát ale len na úrovni warpu, čo je podrobne popísané v kapitole 4.4.2.2. Rozdielom je len použitie operátora sčítanie, namiesto operátora minimum.

Verzia WARP_2 má nedostatok, ktorý ju výrazne limituje v prípade nízkeho počtu častíc použitého pri hľadaní minima. Typicky by nedokázala spustiť kernel s dostatočným počtom blokov, aby boli využité všetky SM na GPU, čo je limitáciou hlavne v prípade vysokej hodnoty `P`. Preto bola v tejto práci implementovaná aj verzia BLOCK_2, ktorá na vyhodnotenie účelovej funkcie pre jednu časticu využije celý blok vlákien. Pseudokód kernelu verzie BLOCK_2 je koncepčne veľmi podobný tomu pre verziu WARP_2, líši sa len v nasledovných bodoch:

- Použitie premennej `laneId` je nahradené použitím `threadIdx.x`, nakoľko vlákna potrebujeme odlíšiť na úrovni celého bloku a nie len warpu.
- Použitie premennej `warpId` je nahradené použitím `blockIdx.x`, keďže celý blok vlákien vyhodnocuje funkciu pre jednu časticu, a tak je spracovávaná častica určená na základe id bloku v mriežke.
- Použitie premennej `warpSize` je nahradené použitím `blockDim.x`, lebo v každej iterácii for cyklu v kroku 4 sa paralelne spracuje `blockDim.x` prvkov polí `a` a `b`, keďže toľko vlákien beží pre jednu časticu paralelne.
- Takisto sa zmení krok 5, čo fakticky vyplýva aj z toho, že použitie premennej `warpId` je nahradené premennou `blockIdx.x`. Teda v prípade verzie BLOCK_2 prebieha redukcia na úrovni bloku, ako popisuje kapitola 4.4.2.1.

Tabuľka 6, Tabuľka 7 a Tabuľka 8 obsahujú porovnanie troch variant vyhodnocovania funkcie $f_5\langle n, P\rangle$, kde n je počet dimenzií a P je parameter definovaný externým konfiguračným súborom:

- THREAD predstavuje verziu kde jedno vlákno vyhodnocuje jednu časticu, čo je pôvodná verzia algoritmu popísaná v kapitole 4.5.1.

- WARP_2 – na vyhodnocovanie jednej častice je použitý warp vlákien,
- BLOCK_2 – na vyhodnocovanie jednej častice je použitý blok vlákien.

Všetky verzie boli spustené pre rôzne veľkosti blokov a zobrazené výsledky sú najlepšie, ktoré jednotlivé verzie pre jednu konkrétnu veľkosť bloku dosiahli. Tabuľky vykazujú niekoľko zaujímavých znakov:

- Pre nízke počty častíc, ako ukazuje Tabuľka 6, je verzia THREAD veľmi pomalá, pretože pri spustení kernelu nevytvorí dostatočný počet blokov, aby dokázala vyťažiť všetky SM, ktoré sa na GPU nachádzajú.
- Naopak, pre veľmi vysoké počty častíc, ako ukazuje Tabuľka 8 je verzia THREAD vďaka svojej jednoduchosti najrýchlejšia. Výsledok sa dá interpretovať tak, že ak je práce dostatok, tak na jej rozdelení nezáleží, pretože dokáže vyťažiť dostupné prostriedky. Navyše verzia THREAD ponúka najjednoduchšie rozdelenie práce, pretože si nevyžaduje krok redukcie medzi vláknami, na rozdiel od verzií WARP_2 a BLOCK_2.
- Verzie WARP_2 a BLOCK_2 sú najvýhodnejšie vtedy, ak verzia THREAD zaostáva, teda pre malý počet častíc, ako ukazuje Tabuľka 6. Je to jednoducho preto, že deľba práce, ktorú tieto verzie používajú dokáže lepšie využiť všetky SM na GPU.
- Verzia WARP_2 v žiadnom z uvedených príkladov extrémne nevyniká, je však zaujímavá tým, že podáva najviac vyrovnané výkony zo všetkých verzií. Pre počet častíc 1024 dokonca dosahuje miestami najlepšie výsledky, ako ukazuje Tabuľka 7.
- Verzia BLOCK_2 najlepšie odpovedá požiadavkám kladeným na funkciu f_5 vyplývajúcich z typického príkladu použitia, kedy parameter `P` dosahuje veľmi vysokých hodnôt. V takom prípade je navyše často nemožné používať vysoké počty častíc, pretože by sa jednoducho nezmestili do globálnej pamäte GPU, ktorá by bola z väčšej časti obsadená hodnotami z konfiguračného súboru, teda `a` a `b`, alebo by bol výpočet príliš časovo náročný na reálne použitie.

	$f_5\langle 8, 4096\rangle$	$f_5\langle 8, 16384\rangle$	$f_5\langle 8, 65536\rangle$	$f_5\langle 8, 131072\rangle$
THREAD	2431 μs	13149 μs	47905 μs	96554 μs
WARP_2	128 μs	510 μs	1980 μs	3943 μs
BLOCK_2	89 μs	297 μs	1029 μs	2017 μs

Tabuľka 6: Časy vyhodnotenia funkcie f_5 pre počet častíc 64, použitá GPU: GTX660Ti

	$f_5\langle 8, 64\rangle$	$f_5\langle 8, 256\rangle$	$f_5\langle 8, 1024\rangle$	$f_5\langle 8, 65536\rangle$
THREAD	59 μs	151 μs	659 μs	47029 μs
WARP_2	25 μs	86 μs	263 μs	15506 μs
BLOCK_2	29 μs	109 μs	594 μs	14430 μs

Tabuľka 7: Časy vyhodnotenia funkcie f_5 pre počet častíc 1024, použitá GPU: GTX660Ti

	$f_5\langle 8, 64\rangle$	$f_5\langle 8, 256\rangle$	$f_5\langle 8, 1024\rangle$	$f_5\langle 8, 4096\rangle$
THREAD	1949 μs	7139 μs	28408 μs	112201 μs
WARP_2	2894 μs	8508 μs	29984 μs	112993 μs
BLOCK_2	3134 μs	12500 μs	66270 μs	151662 μs

Tabuľka 8: Časy vyhodnotenia funkcie f_5 pre počet častíc 131072, použitá GPU: GTX660Ti

4.6 Všeobecné vylepšenia implementácie

Jedno z vylepšení implementácie metódy časticovej optimalizácie na GPU je možné aplikovať len na GPU podporujúcich compute capability 3.5 a vyššie. Je to z toho dôvodu, že tieto zariadenia obsahujú na každom SM špeciálnu L1 cache určenú len na čítanie. O tejto cachi bližšie pojednáva kapitola 3.3.2.

Implementácia popísaná v predchádzajúcej kapitole veľmi intenzívne načítava dáta z globálnej pamäte. Čítanie z globálnej pamäte je súčasťou každého kroku algoritmu a najviac zrejme pri vyhodnocovaní účelovej funkcie. Tabuľka 1 definuje mimo iné aj funkciu f_{10}, ktorej konkrétny tvar závisí od dvoch externe definovaných polí. Tieto polia sú počas výpočtu uložené v globálnej pamäti a ich hodnoty sú pri vyhodnocovaní účelovej funkcie neustále opätovne čítané z pamäti bez toho, aby sa

kedykoľvek zmenili. To je presne prípad, kedy by bolo vhodné tieto hodnoty ukladať v spomenutej L1 cachi určenej len na čítanie.

CUDA za týmto účelom poskytuje špeciálnu inštrukciu `__ldg`, ktorá dostáva ako parameter adresu v globálnej pamäti GPU. Funkcia pracuje nasledovne:

- Najskôr prehľadá L1 cache určenú len na čítanie a ak v nej nájde požadovanú hodnotu, tak ju jednoducho vráti.
- Ak sa požadovaná hodnota v L1 cachi nenachádza, tak je prečítaná z globálnej pamäte a následne uložená v tejto cachi pre prípadné ďalšie použitie.

	f_{10}
s `__ldg`	58157 µs
bez `__ldg`	125326 µs

Tabuľka 9: Vplyv funkcie __ldg pri čítaní z globálnej pamäte na rýchlosť výpočtu optimalizácie funkcie f_{10}, časy normalizované na jednu iteráciu časticovej optimalizácie, použitá GPU: K20

Modifikácia celej aplikácie na globálnej úrovni teda spočíva v jednoduchom nahradení každého čítania z globálnej pamäte volaním inštrukcie `__ldg`. Tabuľka 9 demonštruje rozdiel medzi výpočtovými časmi aplikácie pred a po aplikovaní tejto zmeny. Dosiahnutý čas pre funkciu f_{10} je za použitia 64 častíc a verzie BLOCK_2 po aplikovaní zmeny o viac než polovicu nižší. Takéto dramatické zlepšenie výsledkov však nie je dosiahnuté len vďaka urýchleniu čítania samotných hodnôt použitím funkcie `__ldg`, ale aj vďaka tomu, že jej využitím sa odľahčujú zbernice, ktoré môžu byť lepšie využívané štandardnou L1 cache a zdieľanou pamäťou [30].

S použitím `__ldg` v kóde je nutné dbať na opatrnosť. Ak sa napríklad čítanie globálnej pamäte nachádza zanorené vo for cykle, prípadne je ukazovateľ na túto pamäť členskou premennou objektu uloženého v globálnej pamäti, tak si kompilátor NVCC nemusí s optimalizáciou kódu správne poradiť. Toto tvrdenie je založené na skúsenostiach získaných pri tvorbe tejto práce, kedy jednoduché nahradenie čítania globálnej pamäte pomocou inštrukcie `__ldg` predĺžilo čas výpočtu až štvornásobne kvôli nesprávne vykonanej optimalizácii kompilátorom NVCC.

5. Experimenty

Na základe teoretických poznatkov z predchádzajúcich kapitol bolo v rámci tejto práce implementované riešenie problému globálnej optimalizácie použitím GPU a technológie CUDA. Z uvedených informácií je možné očakávať, že poskytnuté riešenie problému pomocou GPU bude niekoľkonásobne rýchlejšie, ako rovnaký algoritmus implementovaný pomocou CPU. Hlavným predpokladom je vysoký počet jadier, ktorými GPU disponujú. Zrýchlenie však nie je možné overiť bez podrobného porovnania rýchlosti výpočtu oboch verzií, ktorému je venovaná práve táto kapitola.

5.1 Referenčná paralelná verzia pre CPU

Za účelom dosiahnutia objektívnych výsledkov počas porovnávania CPU a GPU verzie časticovej optimalizácie vznikla ako súčasť práce aj referenčná CPU verzia. Vytvorenie vlastnej CPU verzie na účely porovnávania je užitočné najmä preto, aby bolo možné predpokladať, že GPU aj CPU verzia implementuje kľúčové časti výpočtu rovnakým spôsobom, a že kód neobsahuje netriviálne modifikácie. Vďaka tomu sú výsledky neskreslené, na rozdiel od porovnávania s verziou časticovej optimalizácie pre CPU poskytnutou tretími stranami. Potenciálne modifikácie by v takom prípade mohli viesť k predlžovaniu alebo skracovaniu výpočtového času, čo by znižovalo význam porovnávania s implementovanou GPU verziou.

Implementovaná verzia pre CPU je paralelizovaná, aby dokázala využiť vyšší počet jadier dnešných CPU. Testy obsiahnuté v tejto práci využívajú CPU Intel i7-2600k, ktorého kľúčové vlastnosti sú uvedené v kapitole 4.3 (4 fyzické jadrá, 8 vlákien). Na účely paralelizácie je dnes možné použiť niekoľko stabilných voľne dostupných knižníc, no pre potreby tejto práce bola zvolená knižnica TBB[1] od spoločnosti Intel [54], a to z niekoľkých dôvodov:

- TBB poskytuje prostriedky ako paralelný for cyklus a paralelná redukcia. Tá je pri metóde časticovej optimalizácie použitá pri aktualizácii globálneho minima. Krok redukcie popisuje kapitola 2.3.1 a paralelnej redukcii na GPU sa neskôr venuje kapitola 4.4.2.

[1] Skratka z anglického Threading Building Blocks

- Vyššie spomenuté prostriedky poskytujú aj iné knižnice, ako OpenMP [55], na rozdiel od nich TBB prináša možnosť vyššieho stupňa kontroly nad detailmi paralelizácie. Ďalším plusom knižnice TBB je fakt, že je vytvorená priamo spoločnosťou Intel, takže je možné predpokladať lepšie využitie prostriedkov, ktoré ponúka testovací hardware.

Z dôvodu, že dnešné vysoko výkonné CPU majú stále niekoľkonásobne nižší počet jadier ako GPU je pri vyhodnocovaní účelovej funkcie na CPU vhodné využiť jednoduchú verziu paralelizácie, ktorú pre GPU popisuje kapitola 4.5.1, teda jedno vlákno vyhodnocuje jednu časticu. To vedie k jednoduchému kódu, ktorý zároveň už pre nízke počty častíc (o počte jadier CPU) dokáže využiť všetky CPU jadrá.

5.2 Dosiahnuté výsledky

Táto kapitola obsahuje hodnoty výsledkov dosiahnutých pre všetky testované funkcie, ktorých popis obsahuje Tabuľka 1 v kapitole 4.2 a za použitia setu testovacieho hardware, ktorého základné vlastnosti sú uvedené v kapitole 4.3. Grafy s dosiahnutými výsledkami obsahujú na porovnanie aj časy referenčnej paralelnej verzie pre CPU. Dosiahnuté minimá z pohľadu výsledkov zrýchlenia nie sú dôležité, pretože sa po algoritmickej stránke jedná na GPU aj CPU o totožné výpočty, použitý hardware nemá na kvalitu výsledku žiaden vplyv. Z tohto dôvodu dosiahnuté minimá v tejto kapitole nie sú uvedené.

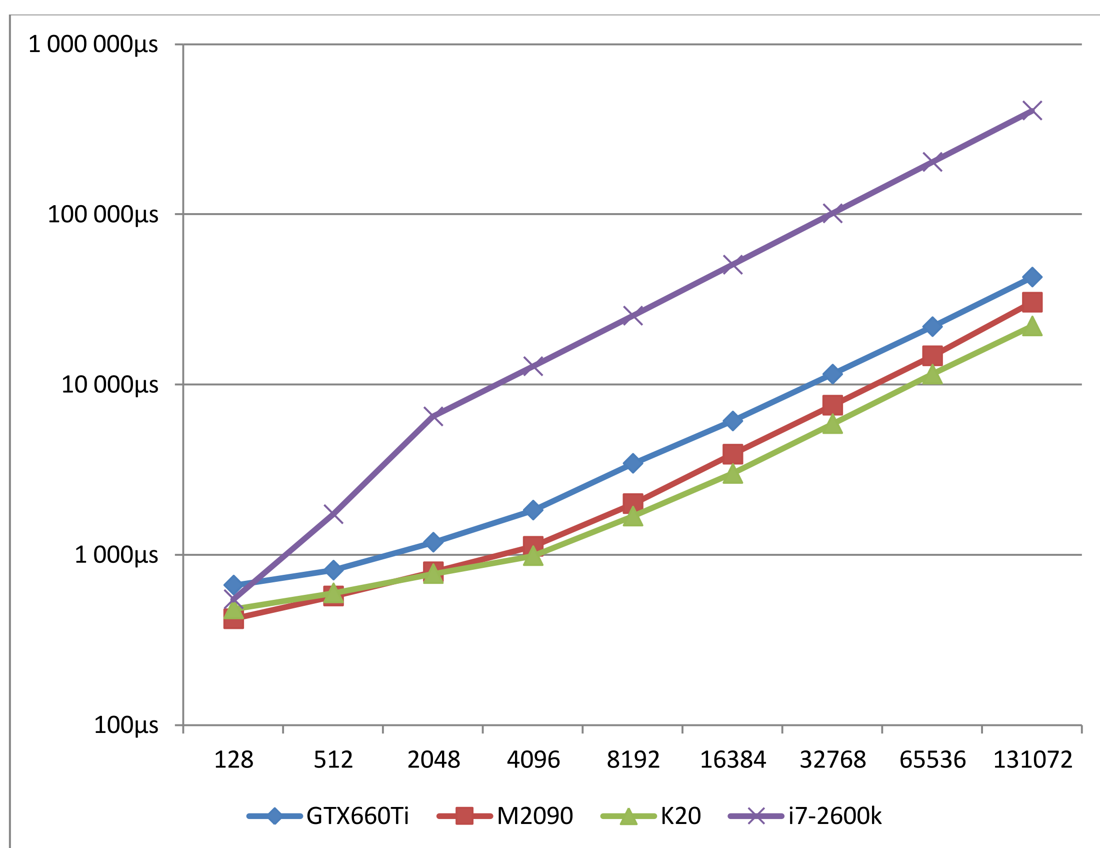
Pre všetky uvádzané výsledky platí, že sa jedná o čas normalizovaný na jednu iteráciu výpočtu metódy časticovej optimalizácie. Výpočet času jednej iterácie prebiehal typicky na základe troch nezávislých meraní po 1000 iteráciách algoritmu.

5.2.1 Funkcie f_1 až f_4

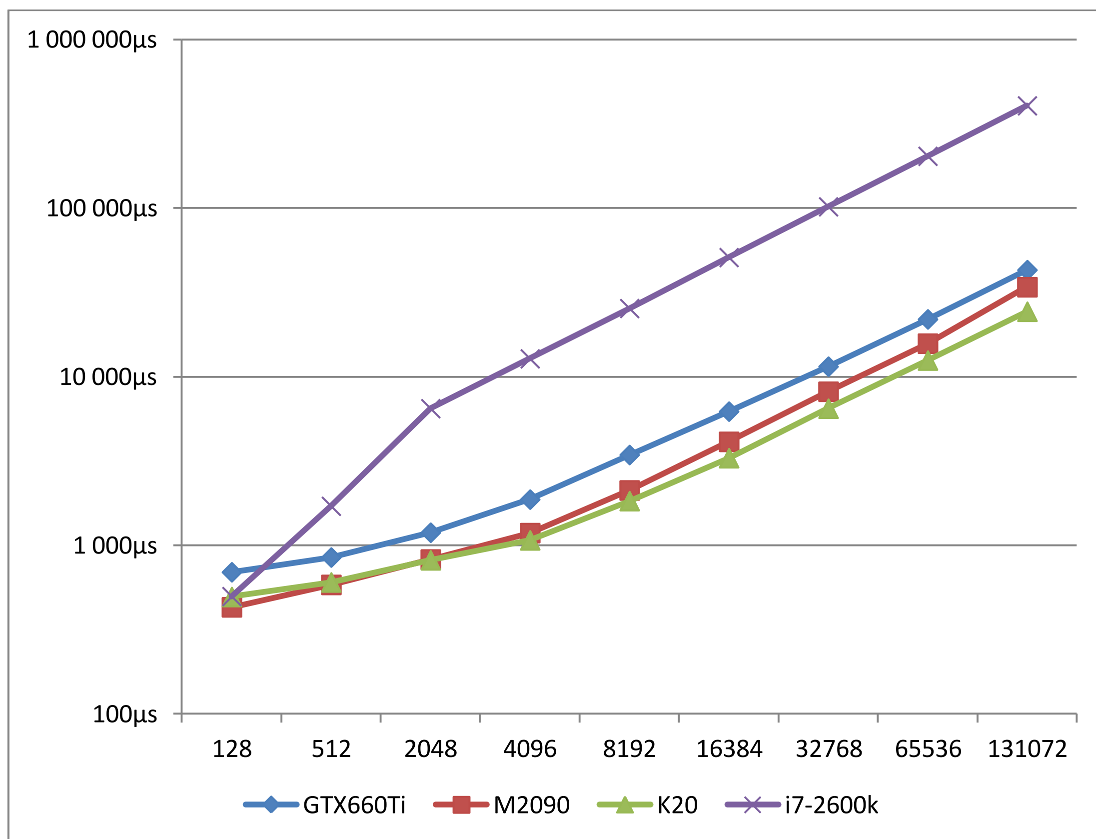
Pre funkcie f_1 až f_4 bola za účelom experimentov na GPU použitá verzia ich vyhodnocovania predstavená v kapitole 4.5.2 a označovaná ako WARP. Na účely testovania bol zvolený počet dimenzií $n = 256$, horizontálne osy v grafoch predstavujú rôzne počty častíc zúčastňujúcich sa na výpočte časticovej optimalizácie.

Z grafov 1 – 4 je možné odvodiť niekoľko informácií špecifických pre dané funkcie:

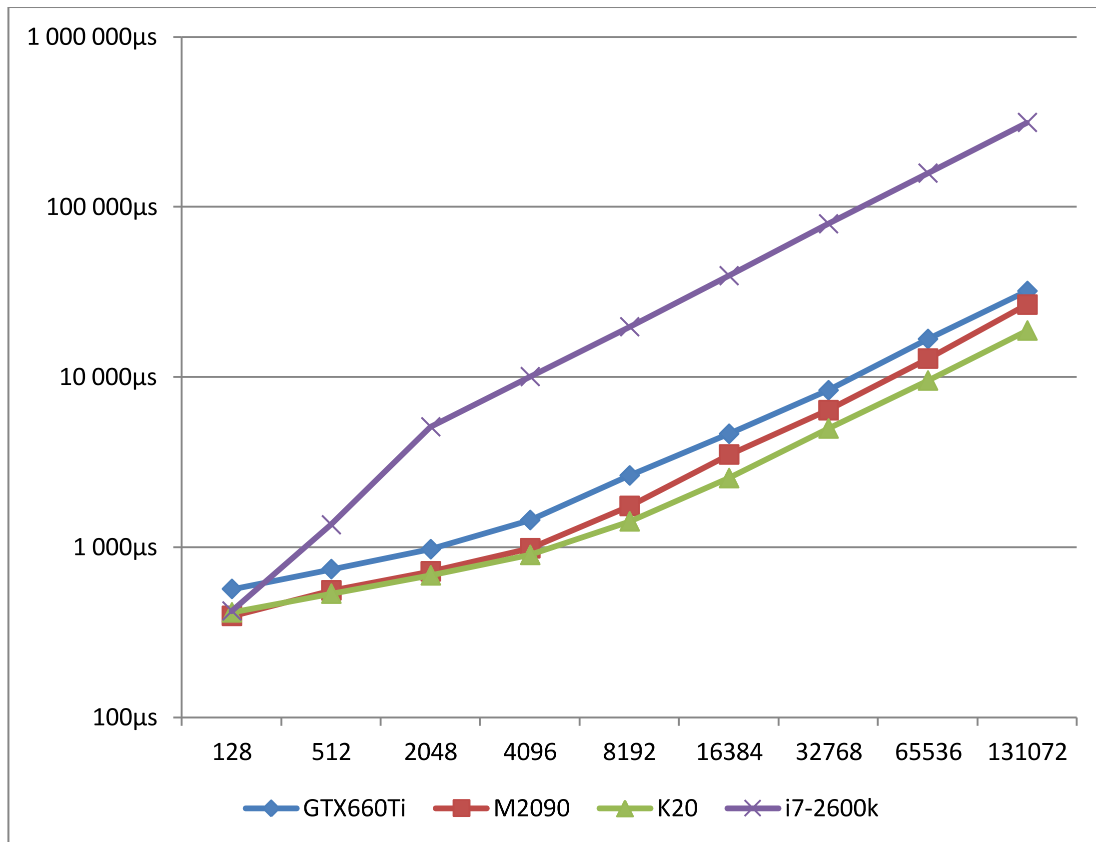
- S narastajúcim počtom častíc dosahuje najsilnejšia GPU K20 oproti CPU verzii 20x kratší výpočtový čas.
- Pre nízke počty častíc (128) môže byť vhodné použiť aj CPU verziu, ktorá vždy dokázala prekonať najslabšiu testovanú GPU GTX660Ti.
- Profesionálne GPU M2090 a K20 dosahujú najlepšie výsledky na funkciách f_1 až f_3, ktoré obsahujú výpočet trigonometrických funkcií. To je dôsledok špeciálnych jednotiek na výpočet týchto funkcií, ktoré obsahuje každý SM, čo pre GPU K20 ilustruje aj Obrázok 5.
- Naopak pre funkciu f_4, ktorá obsahuje len sčítanie a násobenie dosahuje pre vyššie počty častíc veľmi dobré výsledky aj GPU GTX660Ti a prekonáva aj profesionálnu GPU M2090, ktorá je ale o jednu generáciu staršia.
- Všetky GPU škálujú pre nižšie počty častíc lepšie, čo je dané tým, že pri nízkom počte častíc (128, 512) nedokáže výpočet plne zaťažiť všetky SM.



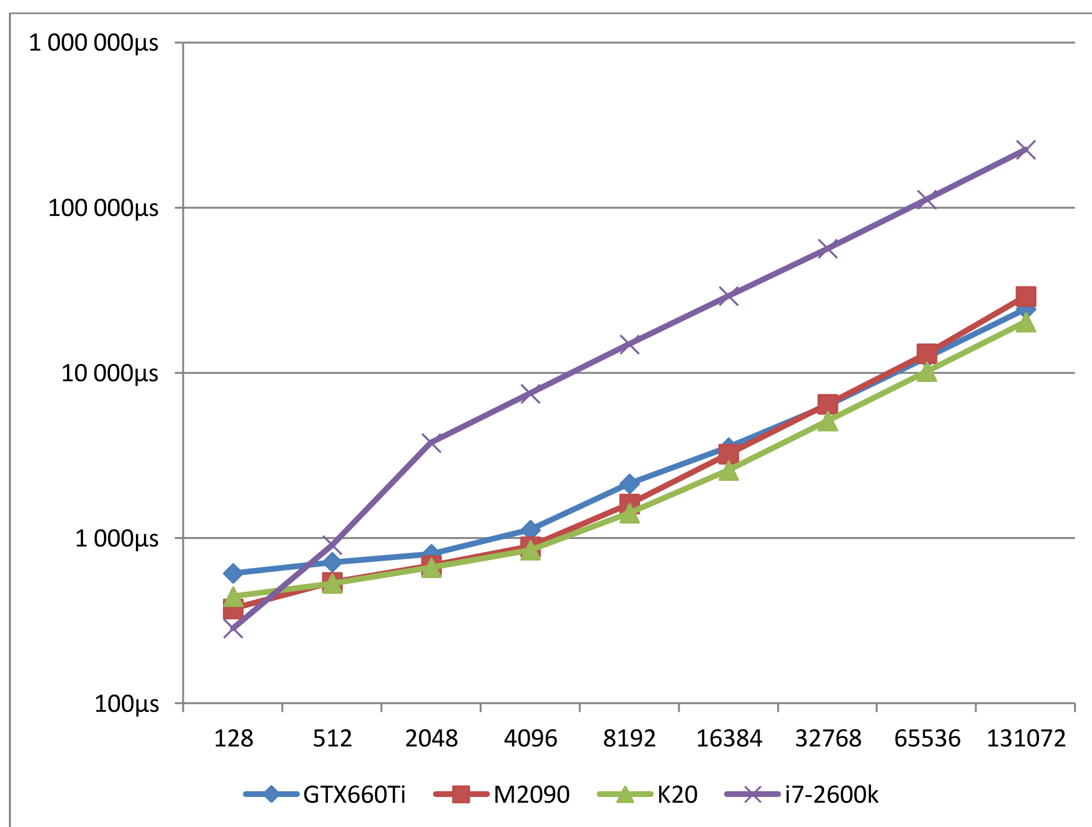
Graf 1: Výpočtové časy pre funkciu f_1<256>, časová os s logaritmickým škálovaním



Graf 2: Výpočtové časy pre funkciu f_2<256>, časová os s logaritmickým škálovaním



Graf 3: Výpočtové časy pre funkciu f_3<256>, časová os s logaritmickým škálovaním



Graf 4: Výpočtové časy pre funkciu f_4<256>, časová os s logaritmickým škálovaním

5.2.2 Funkcie f_6 až f_{10}

Na vyhodnocovanie účelových funkcií f_6 až f_{10} odvodených zo všeobecnej funkcie f_5 bola použitá verzia BLOCK_2 predstavená v kapitole 4.5.3. Najdôležitejším bodom pri experimentovaní s týmito funkciami je čas dosahovaný v závislosti na škálovaní veľkosti parametru P, ktorý spoluurčuje tvar konkrétnej verzie funkcie f_5.

S ohľadom na dôležitosť škálovania voči parametru P obsahujú grafy 5 – 7 porovnanie nameraných hodnôt pre funkcie f_6 až f_9, nakoľko tieto sa líšia len v narastajúcej hodnote parametra P. Z grafov je možné odvodiť:

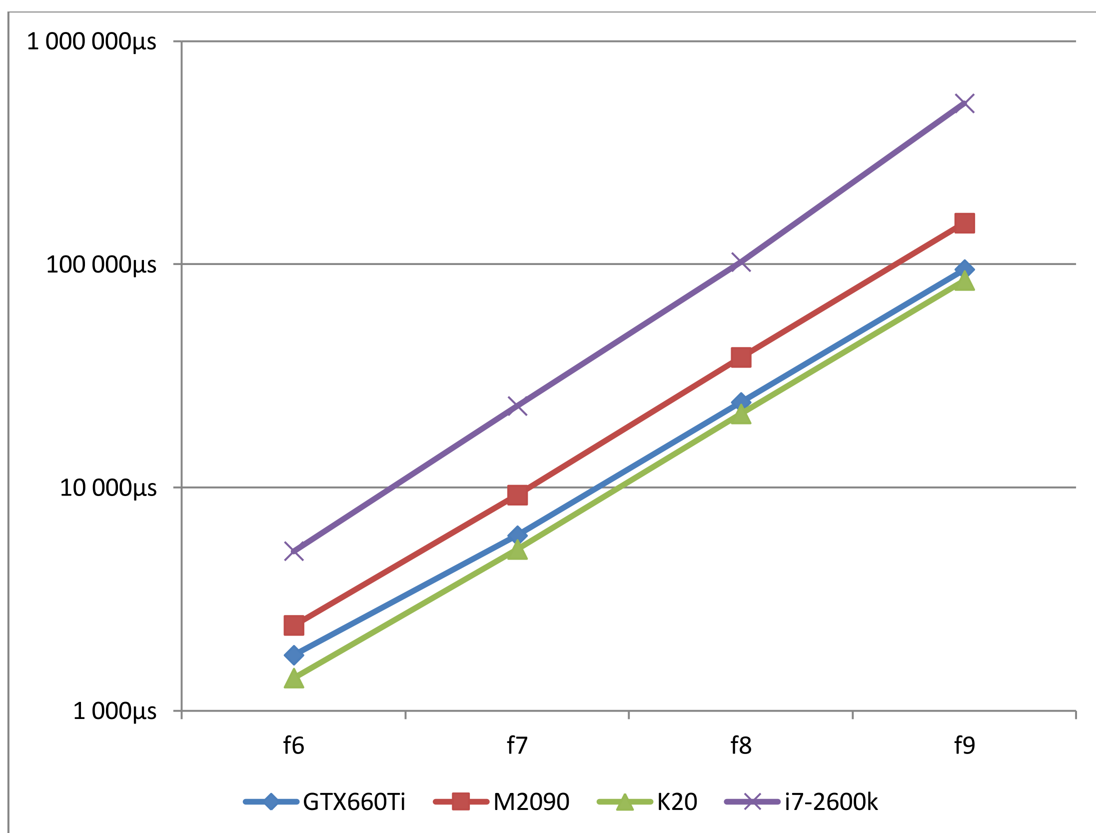
- Graf 5 demonštruje, že už od nízkej hodnoty P (funkcia f_6) a pre nízky počet častíc dokáže GPU účinne škálovať, čo je spôsobené použitím verzie BLOCK_2 spúšťajúcej dostatočné množstvo vlákien na vyťaženie GPU.
- Verziu BLOCK_2 znevýhodňuje nutná komunikácia medzi vláknami bloku v kroku redukcie. Zrejme z tohto dôvodu klesá zrýchlenie oproti verzii pre CPU, ktorá krok redukcie pri vyhodnocovaní funkcie vôbec nepoužíva.

Zrýchlenie je podľa grafov 5 – 7 limitované zhruba na úrovni 6x, čo je značný pokles oproti hodnotám dosiahnutým v predchádzajúcej kapitole.

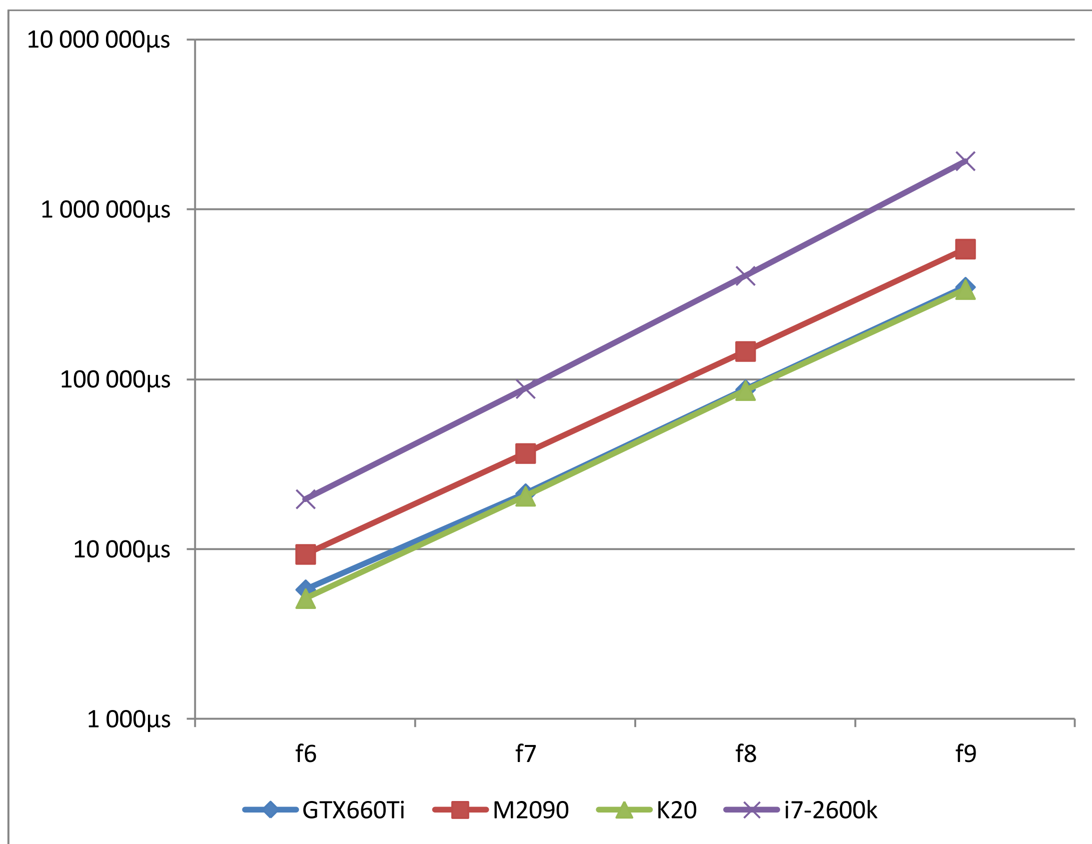
- Zobrazené údaje podporujú tvrdenie, že verzia GPU dodržuje priamu úmeru medzi časom a počtom častíc, ale najmä medzi časom a parametrom P.
- Naopak zvýšenie hodnoty parametru P pre CPU verziu má na výsledný čas horší vplyv, čo dobre ilustrujú grafy 5 – 7, kde fialová čiara predstavujúca CPU i7-2600k zviera s horizontálnou osou grafu väčší uhol.

Funkcii f_{10}, ktorá je najtypickejším predstaviteľom rodiny funkcií tvaru f_5, je venovaný Graf 8, vyplývajú z neho niektoré zaujímavé závery:

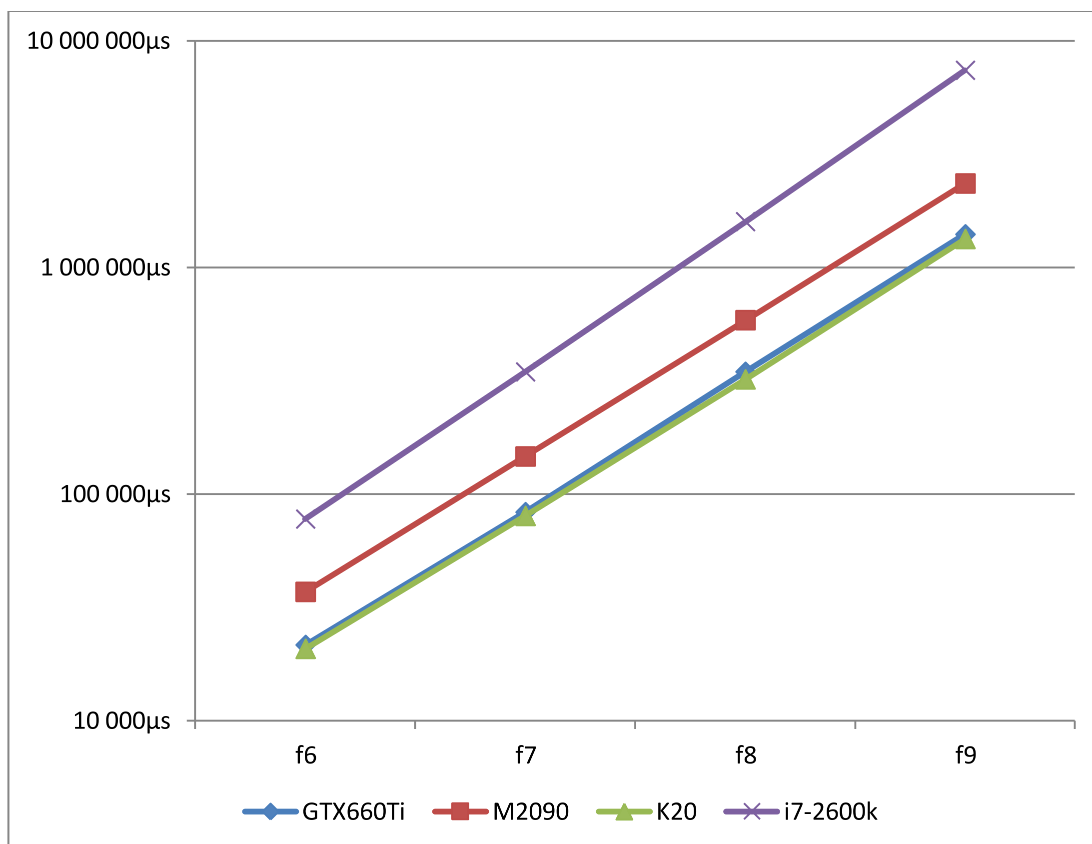
- Verzia CPU prestáva s rastúcim parametrom P stačiť GPU verzii a zrýchlenie sa opäť dostáva až na úroveň 20x. Hlavným faktorom je v tomto prípade zrejme nižšia priepustnosť pamätí CPU.
- Už z predošlých grafov vyplývalo, že GPU M2090 začína citeľne strácať v porovnaní s novšími GPU, ktoré síce pracujú na nižšej frekvencii, ale M2090 porážajú vďaka niekoľkonásobne vyššiemu počtu jadier.



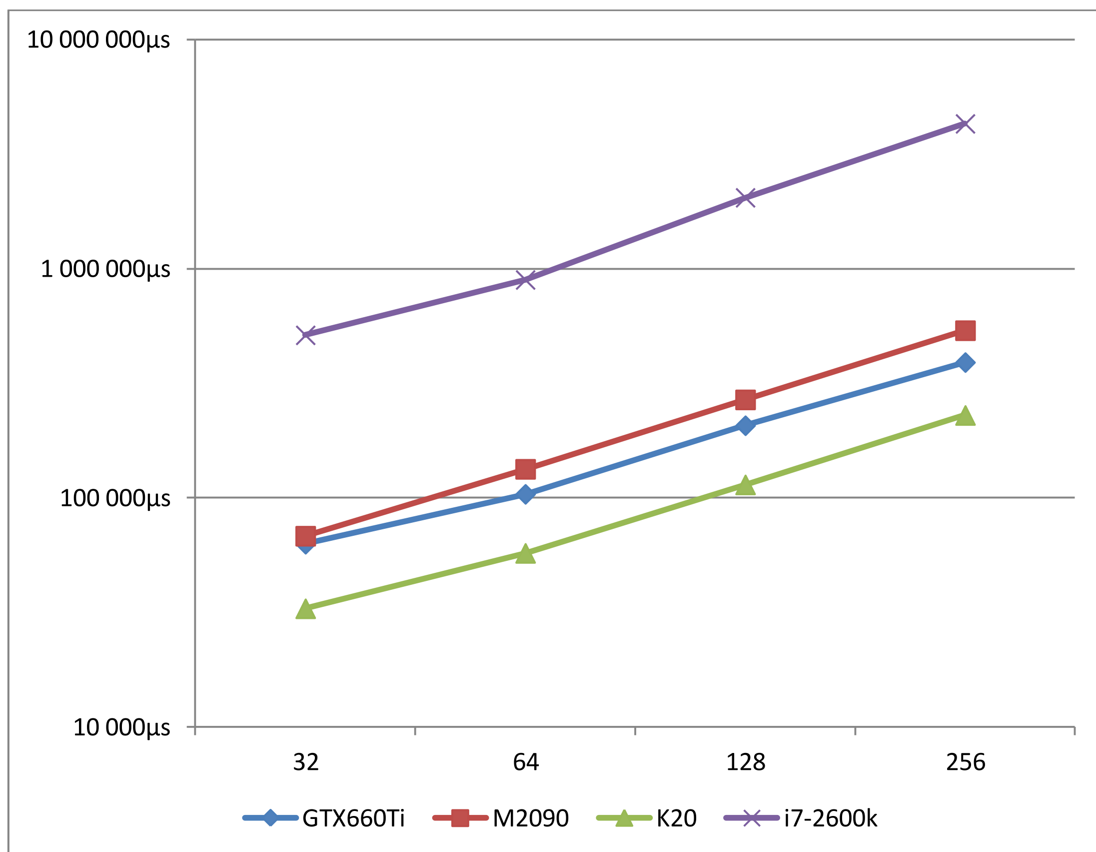
Graf 5: Výpočtové časy funkcií f_6 až f_9, počet častíc 128, časová os s logaritmickým škálovaním



Graf 6: Výpočtové časy funkcií f_6 až f_9, počet častíc 512, časová os s logaritmickým škálovaním



Graf 7: Výpočtové časy funkcií f_6 až f_9, počet častíc 2048, časová os s logaritmickým škálovaním



Graf 8: Výpočtové časy pre funkciu f_{10}, časová os s logaritmickým škálovaním

6. Záver

Teoretický rozbor metódy časticovej optimalizácie poskytnutý v kapitole 2 poslúžil ako základ implementácie tejto metódy používanej na riešenie problému globálnej optimalizácie pomocou GPU. Spolu s princípmi práce s GPU neskôr uvedenými v kapitole 3 bolo následne možné implementáciu zefektívniť a sprístupniť pre široké spektrum dnes bežne používaných a dostupných GPU. Takmer pre každý algoritmický krok implementácie časticovej optimalizácie bolo ponúknutých niekoľko odlišných verzií, čo však niekedy vyplývalo aj z obmedzení rôznych architektúr GPU. Na základe dôsledného porovnávania výsledkov dosiahnutých použitím týchto verzií bola vždy zvolená tá, ktorá najlepšie odpovedá požiadavkám na vysokú priemernú rýchlosť dosahovanú pri optimalizácii testovaného setu účelových funkcií.

Výsledkom celého tohto snaženia je komplexná implementácia metódy časticovej optimalizácie použiteľná na optimalizovanie širokého setu účelových funkcií. Najväčšia pozornosť je pritom venovaná najmä špeciálnym účelovým funkciám, ktorých vyhodnocovanie je definované externým konfiguračným súborom obsahujúcim typicky stovky megabajtov dát. Vďaka testovaniu výslednej verzie pre GPU bola ukázaná jej opodstatnenosť, keďže dosiahla oproti referenčnej paralelnej CPU verzii až dvadsaťnásobné (20x) zrýchlenie. Na testovanie paralelnej CPU verzie bol použitý procesor s ôsmimi vláknami a štyrmi fyzickými jadrami, pričom po obmedzení počtu vlákien používaných výpočtom na CPU narastá doba výpočtu až v násobkoch pôvodného času, čo len podtrháva význam paralelizácie časticovej optimalizácie.

Namerané výsledky tiež poukázali na fakt, že značné zrýchlenie je dosiahnuteľné aj použitím GPU nižšej triedy určenej primárne na masové použitie pri hraní počítačových hier. Testovaná GPU dosiahla dokonca často lepšie výsledky ako o generáciu staršia profesionálna GPU bez video výstupu, ktorá je určená výlučne na všeobecné paralelné výpočty. Z toho je možné usudzovať, že prechod v reálnych aplikáciách z existujúcej verzie pre CPU na verziu pre GPU nemusí byť nutne spojený s vysokými nákladmi na obmenu používaného hardware.

6.1 Budúce vylepšenia

Implementácia metódy časticovej optimalizácie pre GPU vytvorená ako súčasť tejto práce stále poskytuje mnoho priestoru na potenciálne budúce vylepšenia. Tieto vylepšenia je možné rozdeliť do dvoch základných skupín, a to vylepšenia algoritmu časticovej optimalizácie ako takého a zlepšenie využitia prostriedkov, ktoré ponúka GPU. Táto kapitola stručne popisuje niekoľko zástupcov oboch skupín.

Počas testovania vytvoreného prototypu GPU verzie metódy časticovej optimalizácie bežne dochádzalo k situácii, že častice použité počas výpočtu opúšťali oblasť definovanú vstupným intervalom. To samozrejme môže viesť k poskytovaniu nesprávnych výsledkov, nakoľko nájdené minimum sa mohlo nachádzať mimo zadaného intervalu. Na to, aby bolo možné tento prototyp reálne použiť by bolo nutné spomenutý problém vyriešiť, pričom na tento účel existuje v CPU verziách niekoľko prístupov. Jedným z nich je napríklad zahadzovanie častíc, ktoré sa dostanú mimo intervalu, no aj takéto jednoduché riešenie by si vyžadovalo dôsledné zváženie dopadu na implementáciu pomocou GPU.

Špecifické implementácie časticovej optimalizácie pre CPU používané v reálnom svete často využívajú spájanie s inými metódami za účelom dosiahnutia efektívnejšieho alebo presnejšieho procesu globálnej optimalizácie. Jedným z príkladov je proces kalibrácie výpočtového modelu používaného na algoritmické obchodovanie na burze spoločnosťou RSJ spomenutou v úvode tejto práce. Na tento účel používajú voľne dostupnú implementáciu, ktorá v sebe spája metódu časticovej optimalizácie a metódu hľadania vzorov. Detaily popisuje Vaz et al. [19], ktorý je takisto autorom implementácie. Použitie metódy umožňuje rýchlejšie eliminovať väčšie časti vstupného intervalu a urýchľuje tak celý proces, pričom tento prístup by zrejme bolo možné adaptovať aj na použitie na GPU.

Čo sa týka zlepšenia využitia prostriedkov, ktoré GPU ponúka, tak to je typicky dosahované napríklad dôkladnou analýzou prístupov do pamäte GPU a ich následným obmedzovaním za použitia rôznych spôsobov cachovania. V tomto ohľade ponúka vytvorená implementácia stále priestor na zlepšenie.

Iným spôsobom, ako lepšie využiť GPU je možnosť zapojiť do výpočtu viacero GPU súčasne, keďže v dnešnej dobe ich môže byť do slotov na matičných doskách zapojených hneď niekoľko. Spoločnosť NVIDIA označuje technológiu vyvinutú na tento účel ako SLI[1] a metóda časticovej optimalizácie je vhodným kandidátom na jej využitie. Prirodzeným spôsobom ako to dosiahnuť by bolo napríklad na všetkých dostupných GPU spustiť výpočet nad rovnakým vstupným intervalom. Dosiahnuté výsledky by bolo po určitom čase možné buď synchronizovať, aby mali častice na oboch GPU vedomosť o najlepšej globálnej pozícii, alebo by tieto GPU celý čas operovali nezávisle a až na záver by sa z oboch zvolil ten lepší dosiahnutý výsledok.

[1] Skratka z anglického Scalable Link Interface

Literatúra

[1] J. Kennedy and R. Eberhart, *Particle Swarm Optimization*. Perth: IEEE, 1995.

[2] Leo Liberti. (2008, Február) Introduction to Global Optimization. [Online]. http://www.lix.polytechnique.fr/~liberti/teaching/globalopt-lima.pdf

[3] P. M. Pardalos and E. H. Romeijn, *Handbook of Global Optimization Volume 2*.: Kluwer Academic Publishers, 2002.

[4] R. Horst, P. Pardalos, and N. V. Thoai, *Introduction to Global Optimization*, 2nd ed.: Springer, 2001.

[5] L. A. Rastrigin, *Systems of extremal control*.: Nauka, 1974.

[6] P. Hansen and B. Jaumard, *Lipschitz Optimization, Nonconvex Optimization and Its Applications*.: Springer, 1995, vol. 2.

[7] J. D. Pinter, *Global Optimization in Action*.: Springer, 1996.

[8] R. Horst and N. V. Thoai, "DC Programming: Overview," in *Journal of Optimization Theory and Applications*.: Springer, 1999, vol. 103, pp. 1-43.

[9] V. Černý, "Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm," in *Journal of Optimization Theory and Applications*.: Springer, 1985, vol. 45, pp. 41-51.

[10] A. Zhigljavsky and A. Zilinskas, *Stochastic Global Optimization*.: Springer, 2008.

[11] A. H. Land and A. G. Doig, "An automatic method of solving discrete programming problems," in *Econometrica*.: Wiley-Blackwell, 1960, pp. 497-520.

[12] D. Scholz, *Deterministic Global Optimization*.: Springer, 2012.

[13] V. I. Norkin, G. Ch. Pflug, and A. Ruszczyński, "A branch and bound method for stochastic global optimization," in *Mathematical Programming*.: Springer, 1998, vol. 83, pp. 425-450.

[14] B. Hajek, "Cooling Schedules for Optimal Annealing," in *Mathematics of Operations Research*.: INFORMS, 1988, vol. 13, pp. 311-329.

[15] E. O. Wilson, *Sociobilogy: The New Synthesis*.: Belknap Press, 2000.

[16] R. Poli, J. Kennedy, and T. Blackwell, "Particle swarm optimization," in *Swarm Intellingence.*: Springer, 2007, vol. 1, pp. 33-57.

[17] R. C. Eberhart and Y. Shi, "Particle swarm optimization: developments, applications and resources," in *Proceedings of the 2001 Congress on Evolutionary Computation.*: IEEE, 2001, vol. 1, pp. 81-86.

[18] Y. Shi and R. Eberhart, "A modified particle swarm optimizer," in *Proceedings of the IEEE international.*: IEEE, 1998, pp. 69-73.

[19] A. I. F. Vaz and L. N. Vicente, "A particle swarm pattern search method for bound constrained global optimization," in *Journal of Global Optimization.*: Springer US, 2007, vol. 39, pp. 197-219.

[20] R. C. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *Proceedings of the sixth international symposium on micro machine and human science.*: IEEE, 1995, pp. 39-43.

[21] M. Clerc and J. Kennedy, "The particle swarm - explosion, stability, and convergence in a multidimensional complex space," in *IEEE Transactions on Evolutionary Computation.*: IEEE, 2002, vol. 6, pp. 58-73.

[22] J. Kennedy. and R. Mendes, "Population structure and particle swarm performance," in *Proceedings of the 2002 Congress on Evolutionary Computation.*: IEEE, 2002, vol. 2, pp. 1671-1676.

[23] R. Mendes, *Population topologies and their influence in particle swarm performance, PhD Thesis.*, 2004.

[24] R. Poli, *An Analysis of Publications on Particle Swarm Optimisation Applications.*, 2007.

[25] R. C. Eberhart and X. Hu, *Human tremor analysis using particle swarm optimization.*: IEEE.

[26] R. Xu, G. C. Anagnostopoulos, and D. C. Wunsch, "Multiclass Cancer Classification Using Semisupervised Ellipsoid ARTMAP and Particle Swarm Optimization with Gene Expression Data," in *IEEE/ACM Transactions on Computational Biology and Bioinformatics.*: IEEE, 2007, pp. 65-77.

[27] Y. Zhang, C Ji, Y. Ping, and M. Li, "Particle swarm optimization for base station placement in mobile communication," in *2004 IEEE International*

Conference on Networking, Sensing and Control.: IEEE, 2004, vol. 1, pp. 428-432.

[28] Y. Zhang and D. O'Brien, "Fixed Channel Assignment in Cellular Radio Networks Using Particle Swarm Optimization," in *Proceedings of the IEEE International Symposium on Industrial Electronics, 2005. ISIE 2005.*, 2005, vol. 4, pp. 1751-1756.

[29] J. Nenortaite and R. Simutis, "Adapting particle swarm optimization to stock markets," in *5th International Conference on Intelligent Systems Design and Applications, 2005. ISDA '05. Proceedings.*: IEEE, 2005, pp. 520-525.

[30] Nvidia corp. (2012) NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. [Online]. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[31] Nvidia corp. NVIDIA Fermi Compute Architecture Whitepaper. [Online]. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[32] Khronos. OpenCL. [Online]. https://www.khronos.org/opencl/

[33] OpenACC. OpenACC. [Online]. http://www.openacc-standard.org/

[34] B. Leback, D. Miles, and M. Wolfe. Tesla vs. Xeon Phi vs. Radeon A Compiler Writer's Perspective. [Online]. https://cug.org/proceedings/cug2013_proceedings/includes/files/pap199.pdf

[35] P. Micikevicius. (2011) Local Memory and Register Spilling. [Online]. http://on-demand.gputechconf.com/gtc-express/2011/presentations/register_spilling.pdf

[36] Nvidia corp. CUDA Toolkit Documentation. [Online]. http://docs.nvidia.com/cuda/index.html

[37] Nvidia corp. cuFFT. [Online]. http://docs.nvidia.com/cufft/index.html#axzz34SfA3Gvv

[38] Nvidia corp. cuBLAS. [Online]. http://docs.nvidia.com/cublas/index.html#axzz34SfA3Gvv

[39] J. Demouth. (2013) Shuffle: Tips and Tricks. [Online]. http://on-

demand.gputechconf.com/gtc/2013/presentations/S3174-Kepler-Shuffle-Tips-Tricks.pdf

[40] J. Li, D. Wan, Z. Chi, and X. Hu, "An efficient fine-grained parallel particle swarm optimization method based on GPU-acceleration," *International Journal of Innovative Computing, Information and Control*, vol. 3, no. 6, pp. 1707-1714, 2007.

[41] Y. Zhou and Y. Tan, "GPU-based parallel particle swarm optimization," in *Congress on Evolutionary Computation*.: IEEE, 2009, pp. 1493-1500.

[42] Nvidia corp. cuRAND. [Online]. http://docs.nvidia.com/cuda/curand

[43] L. Mussi and S. Cagnoni. (2009) Particle Swarm Optimization within the CUDA Architecture. [Online]. http://www.gpgpgpu.com/gecco2009/1.pdf

[44] Y. Hung and W. Wang, "Accelerating parallel particle swarm optimization via GPU," in *Optimization Methods and Software*.: Taylor & Francis, 2012, vol. 27, pp. 33-51.

[45] Y. Zhou and Y. Tan, "GPU-based parallel multi-objective particle swarm optimization," in *GPU-based parallel multi-objective particle swarm optimization*.: CESER Publications, 2011, vol. 7, pp. 125-141.

[46] L. Mussi, Y. S. G. Nashed, and S. Cagnoni, "GPU-based asynchronous particle swarm optimization," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*.: ACM, 2011, pp. 1555-1562.

[47] W. Zhu and J. Curry, "Particle Swarm with graphics hardware acceleration and local pattern search on bound constrained problems," in *Swarm Intelligence Symposium*.: IEEE, 2009, pp. 1-8.

[48] K. Krishnakumar, R. Swaminathan, S. Garg, and S Narayanaswamy, "Solving large parameter optimization problems using genetic algorithms," in *Guidance, Navigation, and Control Conference*.: AIAA, 1995.

[49] Nvidia corp. (2011) TESLA M2090 DUAL-SLOT COMPUTING PROCESSOR MODULE. [Online]. http://www.nvidia.com/docs/IO/43395/Tesla-M2090-Board-Specification.pdf

[50] Nvidia corp. GeForce GTX 660 Ti Specifications. [Online]. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-

660ti/specifications

[51] Nvidia corp. (2012) TESLA K20 GPU ACCELERATOR. [Online]. http://www.nvidia.com/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v05.pdf

[52] Intel corp. Intel Core i7-2600K Processor. [Online]. http://ark.intel.com/products/52214/Intel-Core-i7-2600K-Processor-8M-Cache-up-to-3_80-GHz

[53] M. Harris. Optimizing Parallel Reduction in CUDA. [Online]. http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

[54] Intel corp. Threading Building Blocks. [Online]. https://www.threadingbuildingblocks.org/

[55] OpenMP Architecture Review Board. OpenMP. [Online]. http://openmp.org/wp/

Príloha A – Obsah priloženého disku DVD

Súčasťou tejto práce je disk DVD, ktorý obsahuje niekoľko zložiek, ktorých popis je obsahom tejto prílohy.

.\bin

- CUDAPso.exe – spustiteľný súbor pre systém Windows7 a novšie, predstavujúci aplikáciu implementujúcu metódu časticovej optimalizácie použitím technológie CUDA, ktorej vytvorenie a testovanie bolo cieľom tejto práce.

.\data

V tejto zložke sa nachádzajú súbory obsahujúce binárne dáta, ktoré definujú použité testovacie funkcie definované v Tabuľka 1. Všeobecne platí, že ak jeden z týchto súborov definuje výpočet funkcie $f_5\langle n, P\rangle$, tak bude obsahovať $(n+1) * P$ čísel so 64 bitovou presnosťou, ktoré spoločne definujú polia a a b. Pre bližšie vysvetlenie pojmov viď kapitola 4.2, ktorá obsahuje detailný popis tvaru funkcie f_5 a z nej odvodených testovacích funkcií f_6, f_7, f_8, f_9 a f_{10}.

- FunctionRsj_ConfigData_4_6980011.bin – súbor dát určujúci tvar funkcie f_{10}. Dáta boli poskytnuté spoločnosťou RSJ a.s. ako aproximácia nimi riešených účelových funkcií.
- Ostatné súbory:
  - FunctionRsj_ConfigData_16_16384.bin,
  - FunctionRsj_ConfigData_16_65536.bin,
  - FunctionRsj_ConfigData_16_272144.bin,
  - FunctionRsj_ConfigData_16_1088576.bin,

  postupne určujú tvar funkcií f_6, f_7, f_8 a f_9. Jedná sa o syntetické dáta vygenerované za účelom testovania výkonu poskytnutej paralelnej implementácie metódy časticovej optimalizácie použitím technológie CUDA. Dáta sú umelo vytvorené tak, aby funkcie dosahovali minimum 0.0 v bode $[1.0]^n$. To bolo možné dosiahnuť najprv vygenerovaním poľa vektorov a, podľa ktorého boli následne spočítané skalárne hodnoty v poli b.

.\demos

- Zložka obsahuje vzorové .bat súbory demonštrujúce správne použitie aplikácie CUDAPso.exe zo zložky .\bin. Použitie je demonštrované pre všetky funkcie, ktoré uvádza Tabuľka 1.

.\doc

- CPUPso_html – zložka obsahuje HTML dokumentáciu vygenerovanú nástrojom doxygen pre projekt .\projects\CPUPso,
- CUDAPso_html – zložka obsahuje HTML dokumentáciu vygenerovanú nástrojom doxygen pre projekt .\projects\CUDAPso,
- experiments.xlsx – súbor obsahuje detailné dáta namerané počas experimentov, ktoré poslúžili ako základ pri tvorbe kapitoly 5.

.\install

- zložka obsahuje inštalátory pre všetky knižnice a nástroje tretích strán, ktoré boli použité počas implementácie metódy časticovej optimalizácie pre CPU aj pre GPU. Bližšie informácie sa nachádzajú v súboroch README.txt umiestnenými v zložkách jednotlivých projektov.

.\projects

- CPUPso – zložka s projektom pre Microsoft Visual Studio 2012, ktorý obsahuje implementáciu referenčnej paralelnej verzie metódy časticovej optimalizácie použitím CPU a knižnice TBB.
- CUDAPso – zložka s projektom pre Microsoft Visual Studio 2012, ktorý obsahuje implementáciu paralelnej verzie metódy časticovej optimalizácie použitím GPU a technológie CUDA.