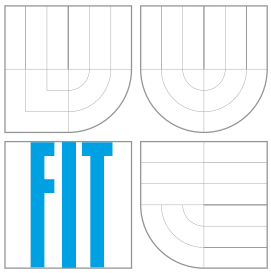
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE GENETICKÉHO ALGORITMU S VYUŽITÍM GPU

THE GENETIC ALGORITHM ACCELERATION USING GPU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PETR POSPÍCHAL

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JIŘÍ JAROŠ

BRNO 2009

Zadání práce

1. Seznamte se s problematikou genetických algoritmů a jejich standardní sekvenční implementací. Identifikujte časově nejnáročnější části tohoto algoritmu.
2. Prostudujte možnosti využití grafického adaptéru pro obecné výpočty. Seznamte se s dostupnými knihovnami pro využití GPU jako obecné výpočetní platformy.
3. Navrhněte techniku využívající grafického adaptéru jako akcelerační jednotky pro jednoduchý genetický algoritmus.
4. Navrženou koncepci implementujte v programovacím jazyce C/C++.
5. Zvolte vhodný optimalizační problém na kterém bude patrné zrychlení dosažené za použité grafického adaptéru oproti sekvenční verzi.
6. Diskutujte přínos vaší implementace pro řešení obecných problémů s využitím genetického algoritmu.

Abstrakt

Tento text představuje diplomovou práci se zaměřením na akceleraci Genetických algoritmů s použitím grafických čipů. První část popisuje Genetické algoritmy a s ním související populaci, chromozom, křížení, mutaci a selekci. Další část je věnována možnostem využití grafických karet jako prostředku pro obecné výpočty, kde jsou popsány jak možnosti programovatelné grafické pipeline s použitím DirectX/OpenGL a Cg, tak specializované knihovny pro GPGPU se zaměřením na architekturu CUDA. Další kapitola se zaměřuje na návrh implementace s použitím GPU, popsány jsou PGA modely a dílčí problémy, jako jsou rychlé řazení a generování náhodných čísel. Následují detaily implementace – migrace, křížení a selekce mapovaná na CUDA softwarový model. Závěrem je provedeno srovnání rychlosti a kvality CPU a GPU části.

Abstract

This thesis represents master's thesis focused on acceleration of Genetic algorithms using GPU. First chapter deeply analyses Genetic algorithms and corresponding topics like population, chromosome, crossover, mutation and selection. Next part of the thesis shows GPU abilities for unified computing using both DirectX/OpenGL with Cg and specialized GPGPU libraries like CUDA. The fourth chapter focuses on design of GPU implementation using CUDA, coarse-grained and fine-grained GAs are discussed, and completed by sorting and random number generation task accelerated by GPU. Next chapter covers implementation details – migration, crossover and selection schemes mapped on CUDA software model. All GA elements and quality of GPU results are described in the last chapter.

Klíčová slova

Genetický algoritmus, GA, Evoluční algoritmy, EA, Paralelní Genetické algoritmy, PGA, Optimalizace funkce, GPGPU, GPU, nVidia CUDA, Pixel shader, grafická karta, obecné výpočty na grafické kartě

Keywords

Genetic algorithm, GA, Evolutionary algorithm, EA, Parallel Genetic algorithm, PGA, Function optimisation, GPGPU, GPU, nVidia CUDA, Pixel shader, graphic card, General purpose computation on graphics card

Citace

Petr Pospíchal: Akcelerace genetického algoritmu s využitím GPU, diplomová práce, Brno, FIT VUT v Brně, 2009

Akcelerace genetického algoritmu s využitím GPU

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jiřího Jaroše

.....
Petr Pospíchal
24. května 2009

Poděkování

Tímto bych rád poděkoval Ing. Jiřímu Jarošovi za nadšené provázení světem Genetických algoritmů, ochotné konzultace i praktické rady. Díky patří i rodičům za jejich podporu při studiu.

Obsah

Kapitola 1

Úvod

Zatímco pro počítače platí již řadu let Moorův zákon hovořící o tom, že výkon a paměť se zdvojnásobí každých 18 měsíců, výkon grafických karet se za poslední 3 roky zdesetinásobil. Procesory na grafických kartách začaly být více než jednotky specializované na rasterizaci grafických primitiv. Staly se z nich několika-set jádrové, masivně-paralelní jednotky umožňující takový rozsah programovatelnosti, že je možné je nyní využívat pro obecné výpočty a dosáhnout tak velkého urychlení u širokého spektra výpočtů. Samo za sebe hovoří nasazení grafických karet pro akceleraci tak náročných výpočtů, jakými jsou simulace turbulencí a aerodynamiky [31], chemické interakce na atomární úrovni [40], nebo výpočet složení země ze seismických dat [33].

Genetické algoritmy jsou přírodním výběrem inspirované techniky řešení problémů, u kterých neexistuje analytické řešení, nebo se řešení mění v čase. S jejich pomocí se dá řešit velmi široká oblast problémů, od optimalizace výrobní linky [44] až po design antény [32] s ohledem na vyzařovací charakteristiku. Řešení však bývá zdlouhavé, protože metoda využívá kvazináhodného prohledávání stavového prostoru.

Tato práce se zabývá možností akcelerovat Genetické algoritmy s pomocí velkého výpočetního výkonu grafických karet a prozkoumává tak zajímavé odvětví informačních technologií. Cílem je zmapovat architekturu, teoretický i praktický potenciál grafických karet v této oblasti a porovnat jej s klasickou sekvenční implementací na procesoru.

1.1 Členění práce

Celá práce je členěna do kapitol s názvem podle tématického okruhu, které popisují.

V první části diplomové práce je podrobně prezentován stěžejní pojem, Genetický algoritmus. Popis se neomezuje jen na praktické hledisko, jsou diskutovány i teoretické podklady a principy, na kterých GA staví. Kromě širšího kontextu v optimalizaci jsou podrobně popsány i všechny jeho elementy, počínaje kódováním chromozomu přes selekci, křížení a mutaci až po parametry a jejich vliv na konvergenci.

Druhá část se zabývá grafickými kartami a možnosti jejich využití pro obecné výpočty. Z počátku kapitoly je nastíněna historie a směr vývoje GPU, následující podkapitoly rozebírají dva směry: výpočty s použitím programovatelné grafické pipeline, kde je kromě OpenGL a DirectX zmíněn i Cg, a specializované knihovny pro GPGPU. Podrobně je popsána především nVidia CUDA, která je využita jako implementační platforma.

Kapitola 4 prezentuje čtenáři kompletní návrh systému – zpočátku je profilací identifikována nejnáročnější část GA, kde je s předstihem porovnána rychlost vlastní CPU

implementace a implementace s použitím knihovny GALib. Ukazuje se, že jednoúčelová aplikace může být výrazně rychlejší, než obecná. Dále jsou prezentovány možnosti paralelizace GA a navržen model mapování problému na CUDA. Podrobně jsou analyzovány i dílčí podproblémy, jako je sběr statistik, řazení a generování náhodných čísel na GPU. V závěru je diskutován problém parametrizace při překladu.

Popis implementační části (kapitola 5) navazuje na návrh a zaměřuje se především na GPU část. Podrobně je popsána struktura kódu a použitého software, problematika ladění a překladu aplikace a veškeré dílčí podproblémy Genetických algoritmů realizovaných na GPU. Závěrem je zmíněn systém zpracování statistik a testovací sestava.

Vyvrcholením práce jsou naměřené výsledky kapitoly 6, převážně vizuální formou s komentáři je zhodnocena jak kvalita, tak rychlost implementace všech dílčích podproblémů (generování náhodných čísel, evaluace všech použitých fitness funkcí). Dále je diskutován vliv migrace jedinců, nastavení parametrů mutace a křížení na rychlost a celkové možnosti architektury GPU jakožto přídavného akcelerátoru. Závěrem je podrobně analyzována celá implementace GA.

Poslední kapitola, s číslem 7, shrnuje dosažené výsledky a nastiňuje směr dalšího vývoje práce.

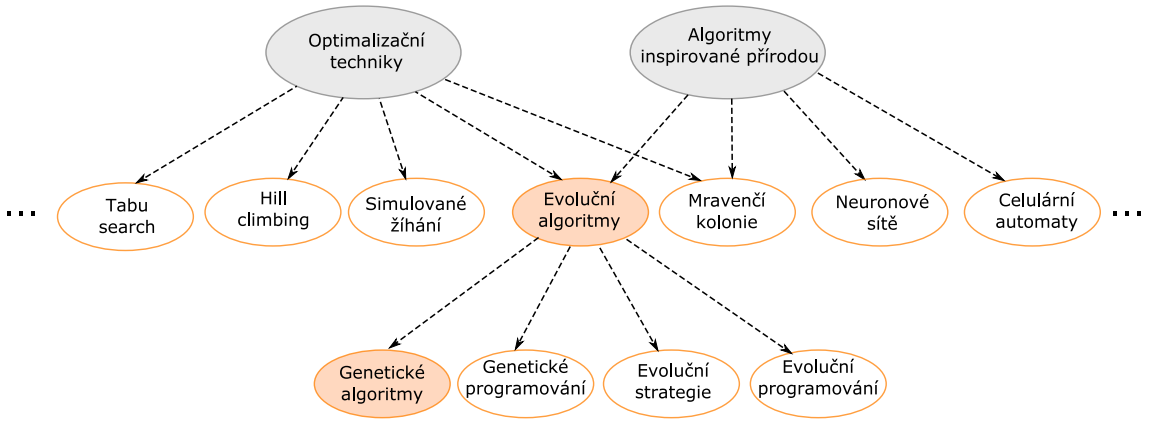
1.2 Návaznost na Semestrální projekt

Obsah této práce navazuje na dílčí řešení – Semestrální projekt. V rámci něho byla diskutována především teoretická část a nástin řešení celé problematiky. Konkrétně se jedná o body 1-3 ze zadání, resp. kapitoly 2-4 v práci. V průběhu řešení diplomové práce byla zjištěna řada skutečností a proto se kapitola návrh systému od Semestrálního projektu liší.

Kapitola 2

Genetický algoritmus

Tato kapitola analyzuje stěžejní pojem celé práce – Genetický algoritmus. Kromě historie a inspirace z přírody nastiňuje i souvislost s jinými optimalizačními metodami a do detailu popisuje chromozom, selekci, křížení i mutaci. V kapitole nechybí ani teoretická část pojednávající o schématech a konvergenci.



Obrázek 2.1: Příslušnost a rozdělení Evolučních algoritmů

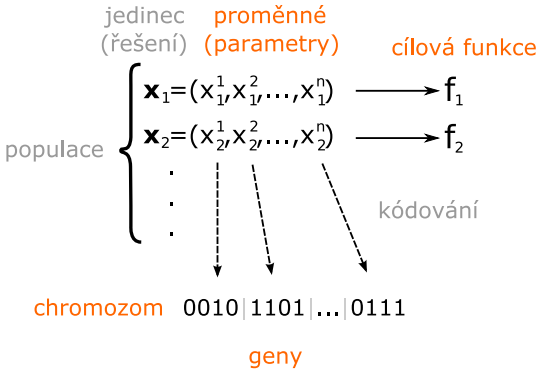
Genetické algoritmy [59] patří vedle Genetického programování [60], Evolučních strategií [56] a Evolučního programování [58] do rodiny Evolučních algoritmů [57] (viz obrázek 2.1). Jedná se o stochastické metody používané převážně pro optimalizaci a strojový návrh (Computational creativity) [54]. Genetické algoritmy jsou z metod Evolučních algoritmů nejpoužívanější. Hledají řešení problému s použitím mutace a křížení nad populací jedinců ve kterých jsou zakódovaná řešení.

Mezi hlavní výhody Genetických algoritmů patří jejich robustnost, modularita, možnost adaptace na dynamicky se měnící podmínky a schopnost řešit i problémy o kterých nemáme žádné větší znalosti. Hodí se i pro optimalizaci funkcí, které jsou multimodální. Vzhledem k jejich vysoké výpočetní náročnosti a problémech nalezení přesného extrému funkce se většinou používají jako poslední možnost, když všechny specializované techniky řešení selžou. Naproti tomu v těchto úlohách fungují překvapivě dobře.

2.1 Zobecnění evoluce a základní myšlenka Genetického algoritmu

Pojem Genetický algoritmus (GA) vznikl v roce 1975, kdy takto John Holland [22] označil skupinu algoritmů se shodnými rysy. Zpočátku byly využívány pro simulaci biologické evoluce, později se ukázal jejich velký potenciál pro optimalizační úlohy. S rostoucím výkonem počítačů se GA dostávají na výsluní jakožto zajímavý prostředek pro řešení širokého spektra problémů.

Genetické algoritmy se inspirovaly přirozeným výběrem podle Darwinovské evoluce. V něm je úspěšnost jedince v populaci charakterizována jeho schopností přežít a rozmnožit se. Úspěšný jedinec následně šíří svůj genom do další generace. Vymíráním jednotlivých jedinců jsou postupně eliminováni slabší jedinci a populace se časem dokonale adaptuje na okolní prostředí, jak je tomu vidět v přírodě. Pokud zobecníme tuto ideu na hledání optimálního řešení nějakého problému, pak jedinec představuje jedno konkrétní řešení z množiny všech možných řešení (kterých je velmi mnoho). Jeho DNA (*Chromozom*) složená z *genů* pak představuje zakódované řešení. Jedinec má tím vyšší pravděpodobnost, že svoji dědičnou informaci přenese do nové populace, čím je úspěšnější, tedy funguje přirozená *selekce*. Jedinec je posuzován pomyslnou mírou úspěšnosti, *cílovou/účelovou funkcí* (z hlediska evoluce pravděpodobností přežití), její míra pak vyjadřuje *sílu/fitness* (viz 2.3.5) jedince.

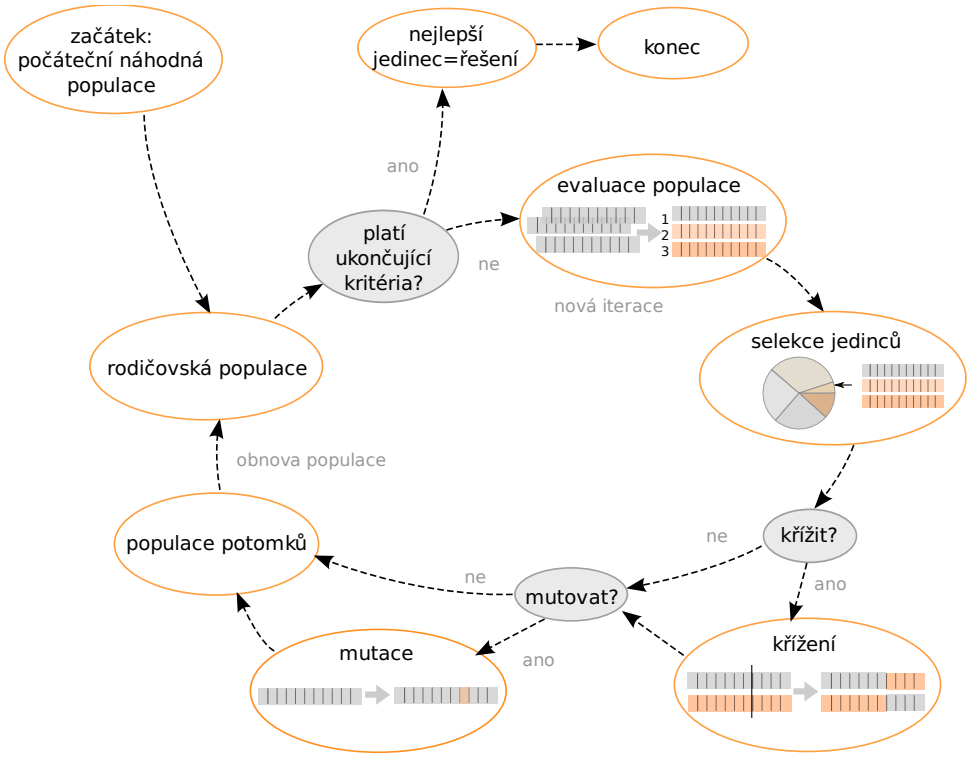


Obrázek 2.2: Názvosloví Genetického algoritmu

Jak ilustruje obrázek 2.2, populace v GA se skládá z jedinců, přičemž každý jedinec představuje jedno konkrétní vyhodnocení optimalizované funkce pro obecně více proměnných, které jsou zakódovány v genech. Chromozom je pak soubor všech genů jedince.

U řady živočichů probíhá rozmnožování pohlavně, kdy dva jedinci zplodí jedince nového, který nese od každého z rodičů část DNA (Chromozomu), dochází tedy ke *křížení* jejich dědičné informace. Tímto způsobem se postupně do jedinců prosazuje genom, který reprezentuje nejefektivnější řešení daného problému. Aby však nedocházelo pouze ke kombinování již existujícího genfondu populace, funguje při reprodukci ještě malá náhodná *mutace* genomu, která způsobí, že do populace přibývají nové, originální, řešení. *Mutaci*, *křížení* a *selekci* nazýváme *genetické operátory*.

Obecné schéma funkce Genetického algoritmu ilustruje obrázek 2.3. Jak je patrné, výchozím bodem je náhodná populace jedinců, která se stává počáteční rodičovskou populací. Dalším krokem iterace je evaluace populace, při které je každému jedinci na základě



Obrázek 2.3: Schéma funkce Genetického algoritmu

jeho chromozomu přiřazena úspěšnost (fitness) v populaci. Tento krok je obecně v algoritmu nejnáročnější, protože vyžaduje výpočet účelové funkce (která navíc bývá složitá) pro každého jedince zvlášť. Následně jsou některou ze selekčních metod (viz 2.6) pseudonáhodně vybíráni kvalitnější jedinci a s určitou pravděpodobností spolu zkříženi (viz 2.5), čímž vytvoří potomky, v opačném případě se jejich genom nemění a pouze se stanou potomky. Malý počet jedinců je navíc náhodně zmutován (viz 2.4), čímž se rozšíří biodiverzita populace[1]. Selekce s křížením a mutací se opakuje, dokud populace jedinců nedosáhne maximálního počtu, poté dojde k obnově rodičovské populace (viz 2.7). V případě, že jsou v nové iteraci splněny ukončující podmínky (viz 2.8.7), algoritmus končí a za řešení celého problému prohlásí chromozom nejsilnějšího jedince v populaci.

Protože jsou GA stochastické, obvykle je třeba více běhů algoritmu pro nalezení přijatelného řešení daného problému.

2.2 Schémata a konvergence GA

Schéma [28] je řetězec délky k nad množinou symbolů $\{0, 1, \#\} : \sigma \in \{0, 1, \#\}^k$, kde $\#$ je tzv. *wildcard symbol* za který můžeme nahradit libovolný jiný. Řekneme, že řetězec $\alpha \in \{0, 1\}^k$ je *příkladem schématu* σ, jestliže v každém indexu schématu σ s jiným než wildcard symbolem je hodnota znaku řetězce α totožná se znakem ze σ:

$$\alpha \in \sigma \Leftrightarrow (\forall i \in \{1, 2, \ldots, k\} : \sigma_i \in \{0, 1\} \Rightarrow \sigma_i = \alpha_i)$$

[1] některé implemtance mutují pouze jedince právě zkřížené, v této práci budeme uvažovat náhodnou míru mutace všech jedinců v nové populaci

Například binární řetězec délky $k = 7, \alpha = (1101011)$ je příkladem schématu $\sigma = (11\#\#0\#\#)$.

Množina $I(\sigma)$ pak obsahuje všechny příklady schématu σ: $I(\sigma) = \{\alpha; \alpha \in \sigma\}$. Můžeme ji sestrojit tak, že všechny wildcard symboly systematicky nahrazujeme symboly $\{0, 1\}$.

Řádem schématu σ značeným $o(\sigma)$ rozumíme počet symbolů $\{0, 1\}$ v σ: $o(\sigma) = |\{i; \sigma_i \in \{0, 1\}\}|$. Například pro $\sigma = (11\#\#0\#\#)$ platí $o(\sigma) = 3$, protože schéma obsahuje 3 ne-wildcard symboly.

Délka Δ *schématu* σ je určená jako rozdíl mezi indexem posledního binárního symbolu v σ a prvním binárním symbolu v σ: $\Delta(\sigma) = max\{i; \sigma_i \in \{0, 1\}\} - min\{i; \sigma_i \in \{0, 1\}\}$. Například pro $\sigma = (11\#\#0\#\#)$ platí $\Delta(\sigma) = 5 - 1 = 4$.

2.2.1 Globální optimum

Podle *Schema theoremu* [73] se schémata způsobující vyšší resp. nižší střední ohodnocení jedince vyskytují v další generaci v chromozomech s rostoucí resp. klesající frekvencí. Tento závěr je založený na předpokladu, že schéma má malou pravděpodobnost zániku, což je obvykle splněno pro schémata s malým $\Delta(\sigma)$ nazvané *stavební bloky*.

Z hlediska optimalizace je důležitý výzkum navazující na Schema theorem z posledních let: Nechť v Genetickém algoritmu posloupnost populací $P_0, P_1, ...$ je monotónní, tj. pro každou generaci t platí

$$\max_{\alpha \in P_{t+1}} F(\alpha) \quad \geq \quad \max_{\alpha \in P_t} F(\alpha) \tag{2.1}$$

potom Genetický algoritmus asymptoticky pro $t \to \infty$ dosáhne globálního optima

$$\alpha_{opt} \quad = \quad \lim_{t \to \infty} \left(\arg \max_{\alpha \in P_t} F(\alpha) \right) \tag{2.2}$$

Monotónnost GA lze dosáhnout s použitím *elitismu* (viz 2.8.4), ve kterém je nejlepší řešení z původní populace P_t vždy přeneseno do nové P_{t+1}. Tedy Genetické algoritmy lze úspěšně použít pro optimalizační úlohy.

2.3 Chromozom, jeho kódování a ohodnocení

Chromozom představuje zakódování jednoho jedince, neboli jeho *genotyp* [29]. Společně s prostředím (ohodnocením fitness funkcí) tvoří *fenotyp*, tedy to, jak se jedinec projevuje v populaci. Je rozdělen na jednotlivé *geny* představující dílčí podproblémy (parametry optimalizované funkce).

Kódování chromozomu, které je stěžejním prvkem implementace optimalizační techniky, by mělo splňovat následující vlastnosti:

neredundance v kódování chromozomu se nevyskytují dvě různá zakódování stejného řešení problému

kauzalita jeden konkrétní chromozom představuje právě jedno řešení daného problému

legalita všechna možná zakódování jedince by měla představovat možné řešení optimalizovaného problému, tedy v zakódování jedince se nevyskytují neplatná řešení (například kvůli omezení definičního oboru funkce)

kompletnost kódování chromozomu by mělo pokrývat všechna možná řešení optimalizovaného problému

Tyto vlastnosti nabývají na důležitosti především při permutačním a messy kódování chromozomu.

2.3.1 Binární kódování chromozomu a Hammingova bariéra

Binární kódování chromozmu patří mezi nejstarší a nejpoužívanější. Mezi jeho hlavní výhody patří nativní interpretace počítači a z toho plynoucí rychlá a efektivní implementace genetických operátorů nad nimi. Každý gen jedince v něm tvoří posloupnost binárních číslic.

Uvažujme následující dva geny:

$$x_1 = 00011111_b = 31_d$$

$$x_2 = 00100000_b = 32_d$$

v případě, že fitness funkce uvažuje kódování genu jako jeho hodnotu, jsou si tyto dvě dekadicky sousední hodnoty binárně velmi vzdálené. Extrém optimalizované funkce může být v blízkém okolí genu x_2, zatímco nejlepší jedinec x_1 se aplikací genetických operátorů k řešení v podobě x_2 dostane jenom s velmi malou pravděpodobností. Tomuto jevu říkáme *Hammingova bariéra*. Tento obecně negativní vedlejší efekt se dá omezit nebo eliminovat takto:

- použitím operátoru inverze nebo inverzní mutace (viz 2.4.1)
- použitím Grayova kódování [62] pro reprezentaci genu, v němž se hodnotou sousedící číslice liší vždy jen jedním bitem. To však má negativní dopad na výkon, protože při použití binárních operátorů musí být chromozomy opakovaně převáděny z a do Grayova kódu.

Mutace resp. křížení binárního chromozomu popisuje blíže kapitola 2.4.1 resp. 2.5.

2.3.2 Reálné kódování chromozomu

V reálném kódování genomu jedince jsou použity pro geny čísla s plovoucí řádovou čárkou, tedy gen může snadno vyjadřovat parametr spojité optimalizované funkce. Nevýhodou je v tomto případě nižší efektivita implementace genetických operátorů. Datový typ float je také standardním u grafických karet. Mutace a křížení se v případě reálného chromozomu liší od standardní binární verze (viz 2.4.2 a 2.5).

2.3.3 Permutační kódování chromozomu

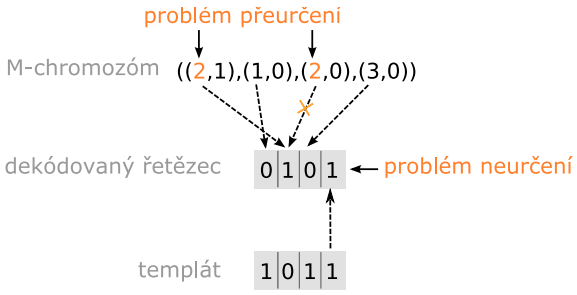
Permutační chromozom se skládá z fixního počtu číslic reprezentujících pořadí provádění nějaké činnosti, například optimální průchod výrobků výrobní linkou, nejkratší cestu mezi městy obchodního cestujícího (*TSP problém*[2]) a pod. Mutace se oproti klasickému binárnímu kódování liší, protože je třeba zachovávat všechna čísla v řetězci jak ilustruje kapitola 2.4.3 a 2.5.

[2]Travelling Salesman Problem je klasický testovací problém Evolučních algoritmů, účelem je najít optimální pořadí procestovaných měst tak, aby obchodník musel cestovat co nejkratší celkovou vzdálenost

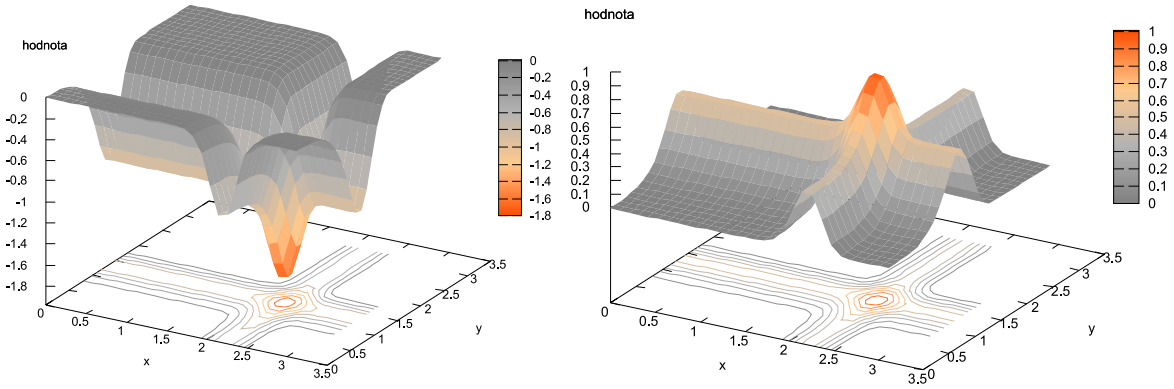
2.3.4 Messy chromozom

Messy chromozomy jsou vylepšením Genetických algoritmů pro hledání extrémů multimodálních funkcí. Na rozdíl od běžných kódování chromozomů nemají fixní délku a jejich interpretace je složitější: Nechť $Q = \{1, 2, \ldots, k\} \times \{0, 1\}$ je množina uspořádaných dvojic (μ, ν) kde μ je index a ν je binární hodnota $\{0, 1\}$. M-chromozóm délky i je pak definován jako $\chi = ((\mu_1, \nu_1), (\mu_2, \nu_2), \ldots, (\mu_i, \nu_i)) \in Q'$.

M-Chromozóm pak dekódujeme na binární vektor tak, že průchodem χ zleva doprava na μ_i-tý index dosazujeme ν_i-tou hodnotu. V případě, že je některý index výsledného vektoru definován opakovaně (tzv. *problém přeurčení*) bere se v potaz definice s nejnižším číslem indexu. V případě, že je některý index výsledného vektoru M-chromozomem nedefinován (tzv. *problém neurčení*), je pro definici použit *templát*, tj. vzorový jedinec vznikající v počáteční populaci náhodně a v dalších populacích z nejsilnějšího jedince. Celou situaci ilustruje obrázek 2.4.



Obrázek 2.4: Funkce M-chromozómu



Obrázek 2.5: Vztah účelové (vlevo) a normalizované Fitness funkce (vpravo)

2.3.5 Účelová a Fitness funkce

Účelová funkce představuje v GA zakódovaný problém, jenž optimalizujeme. Její sestavení je důležitým krokem v návrhu řešení problému. Obecně má N parametrů a optimalizací se snažíme najít její globální extrém (minimum nebo maximum) na předem stanoveném definičním oboru. Fitness funkce neboli *fenotyp* je vyjádřením úspěšnosti populace jedinců, kdy každý gen z chromozomu jedince představuje jeden parametr (osu) funkce a výsledkem

je míra úspěšnosti (fitness) jedince. Fitness funkce se definuje tak, aby nejsilnější jedinec měl přidělenou nejvyšší hodnotu a nejslabší nejnižší. Speciálním případem je pak *normalizovaná fitness funkce*[3], kdy je všem jedincům přiděleno číslo z intervalu $x \in \langle 0; 1\rangle$. V případě, že optimalizace řeší minimalizaci, tj. hledáme minimum účelové funkce, je fitness funkce vůči účelové funkci ve vztahu $u(x_1, x_2, \ldots, x_n) = -f(x_1, x_2, \ldots, x_n)$.

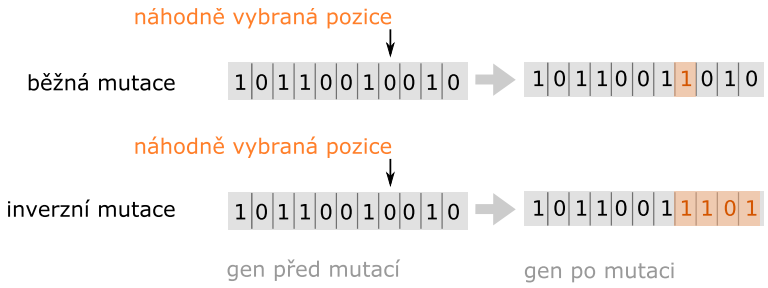
Situaci ilustruje obrázek 2.5: účelovou funkci vlevo minimalizujeme, má obor hodnot $u \in \langle -1,8; 0\rangle$, fitness funkce vpravo je převedená na maximalizaci a do oboru hodnot $f \in \langle 0; 1\rangle$, tedy jedinci blížící se minimu v účelové funkci dostávají maximální ohodnocení ve fitness funkci. Definiční obory se neliší. Obě funkce jsou definované pro dvě proměnné, chromozom jedince tedy bude obsahovat dva geny, jeden pro parametr x, druhý pro y.

U dynamických problémů se účelová i fitness funkce může měnit v čase, pro některé problémy je také vhodné uměle penalizovat některé jedince prostřednictvím fitness funkce, aby se z řešení problémů rychle vyloučily nevhodné genomy.

2.4 Mutace

Mutace má za úkol do genomu jedinců propagovat možná nová schémata a tím rozšiřovat biodiverzitu populace jedinců. Míra mutace je jeden z parametrů Genetickém algoritmu (viz kapitola 2.8.2).

2.4.1 Mutace binárního chromozomu



Obrázek 2.6: Porovnání běžné binární mutace a inverzní binární mutace

běžná binární mutace nejdříve zvolí náhodný index v genu, jehož bit následně zneguje, jak nastiňuje obrázek 2.6.

inverzní binární mutace je operátor využívaný v souvislosti s Hammingovou bariérou (viz 2.3.1). Stejně jako u běžné binární mutace chromozomu je v ní nejdříve vybrán náhodně index v genu. Následně ale není znegován pouze tento bit, ale všechny bity následující od něj vpravo, čímž je částečně kompenzováne efekt Hammingovy bariéry. Situaci ilustruje obrázek 2.6.

[3] V případě, že známe obor hodnot funkce, můžeme normalizovat celou fitness funkci, tj. udělat převod tak, aby globální extrémy měly ohodnocení buď $0,0$ nebo $1,0$ jak ilustruje obrázek. V případě, že obor hodnot výsledné funkce neznáme, můžeme normalizovat jenom v rámci jedné generace jedinců, kdy nejhorší jedinec z generace má ohodnocení $0,0$ a nejlepší $1,0$

2.4.2 Mutace reálného chromozomu

Mutace reálného chromozomu typicky využívá Normální distribuce [64]:

$$\varphi_{\mu,\sigma^2}(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}} = \frac{1}{\sigma}\varphi\Big(\frac{x-\mu}{\sigma}\Big) \tag{2.3}$$

kde $\sigma > 0$ je směrodatná odchylka a μ je střední hodnota. Konkrétní hodnoty se volí na základě charakteru optimalizační funkce, obvykle se uvažuje $\sigma = 1$ a $\mu = 0$. Další možností je volit číslo z náhodného intervalu z definičního oboru problému s rovnoměrným rozložením, to je však vhodné pouze pro speciální funkce nemající jako definiční obor celé $\mathbb{R}$.

2.4.3 Mutace permutačního chromozomu

Při mutaci permutačního chromozomu je potřeba zachovat obsah všech prvků v chromozomu, náhodná změna jednoho čísla tedy nepřipadá v úvahu. Proto se využívá náhodného určení dvou indexů, které se následně vzájemně prohodí. Tím dojde k malé náhodné změně, ale zůstane zachován permutační charakter chromozomu.

2.4.4 Mutace messy chromozomu

Messy chromozomy se mutují podobně jako binární chromozomy: je náhodně vybrána dvojice (μ_i, ν_i) ve které je zmutováno buď μ nebo ν.

2.5 Křížení

Křížení je proces, při kterém z genomu dvou rodičů vznikají dva potomci. Existuje řada variant křížení lišících se optimalizací pro konkrétní kódování chromozomu. V případě, že chromozom obsahuje více genů probíhá křížení separátním zkřížením jednotlivých genů. Jeho míra je v Genetickém algoritmu jeden z parametrů (viz kapitola 2.8.3).

2.5.1 Jednobodové, dvoubodové a uniformní křížení

binární Jednobodové, dvoubodové (také zvané cyklické) a uniformní křížení se typicky používá u křížení binárního chromozomu. Genom se při něm rozdělí na obecně N-místech, přičemž od rozdělení je genom vždy kopírován z rodiče do jiného potomka, jak ilustruje obrázek 2.7. Speciálním případem je pak uniformní křížení, ve kterém je pro každý bit genomu vybrán náhodně jeden potomek.

m-chromozom Křížení Messy-chromozomu je obdobné jako u binárního, na rozdíl od něho se ale s pracuje páry (μ_i, ν_i) místo bity genu.

permutační Permutační křížení musí zajistit obsažení všech permutovaných prvků v chromozomu, proto se používá upravená verze křížení, ve které jsou rodiče náhodně rozděleni podobně jako u binárního křížení, avšak posloupnost do tohoto bodu zkopírujeme do chromozomu potomka a zbytek doplníme nepoužitými čísly v pořadí stejném jako ve druhém rodičovi.

Obrázek 2.7: Srovnání různých typů binárního křížení

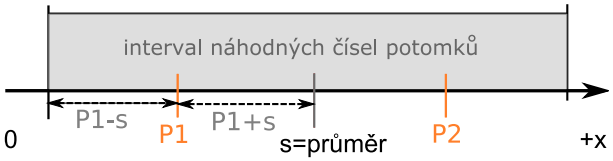
2.5.2 Aritmetické křížení

Aritmetické křížení se používá u křížení reálných chromozomů.

$$O_1 = a \cdot P_1 + (1 - a) \cdot P_2 \tag{2.4}$$
$$O_2 = (1 - a) \cdot P_1 + a \cdot P_2 \tag{2.5}$$

kde O_1 a O_2 jsou nově vzniklí potomci, P_1 a P_2 rodiče a a je náhodné číslo vybrané před každým křížením.

Některé implementace [29] používají variantu průměrování nebo průměrování s rozšířením intervalu (viz obrázek 2.8) pro předcházení konvergence směrem ke středu účelové funkce.



Obrázek 2.8: Princip křížení reálného chromozomu průměrováním s rozšířením intervalu

2.5.3 Heuristické křížení

Heuristické křížení se u reálného chromozomu snaží přizpůsobit směr hledání konvergenci účelové funkce. Využívá k tomu seřazení genomu rodičů podle jejich ohodnocení fitness:

$$O_1 = P_H + a \cdot (P_H - P_L) \tag{2.6}$$
$$O_2 = P_H \tag{2.7}$$

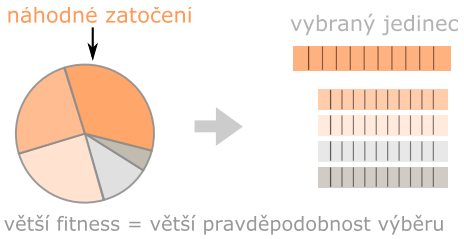
kde O_1, O_2 jsou nově vzniklí potomci, $a \in \langle 0; 1 \rangle$ je náhodné číslo a P_L, P_H jsou rodiče, přičemž P_H má lepší fitness ohodnocení.

2.6 Selekce

Selekce v Genetickém algoritmu slouží k simulaci přirozeného výběru jedinců v přírodě. Míru prosazování kvalitních jedinců do nové generace nazýváme *selekční tlak*. Jeho hodnota pak závisí na použité selekční metodě. Obecně platí, že pro počáteční generace by selekční tlak GA neměl být příliš vysoký, jinak mají jedinci tendence zkonvergovat k lokálnímu optimu. Naproti tomu v případě, že je selekční tlak příliš malý, řešení trvá neúměrně dlouho, protože je zbytečně prohledáván stavový prostor nevedoucí k řešení.

2.6.1 Ruletový výběr

Princip ruletového výběru ilustruje obrázek 2.9. Jedinec je vybrán s tím větší pravděpodobností, čím větší je jeho fitness ohodnocení. Tento přístup relativně dobře simuluje přirozený přírodní výběr, avšak má nevýhodu v tom, že jedinci s vůči ostatním relativně velkou fitness téměř znemožní postup slabých jedinců do další generace. Tento nedostatek se snaží eliminovat další techniky.



Obrázek 2.9: Princip ruletového výběru

2.6.2 Pořadový výběr

V případě, že ohodnocení (fitness) některého jedince v populaci výrazně převyšuje ostatní jedince je u klasického ruletového výběru tento jedinec neustále vybírán do nové generace, čímž zničí různorodost celé populace. Tento problém se dá kompenzovat použitím pořadového výběru (Rank Selection). V něm jsou jedinci vybíráni opět ruletovým stylem, ale místo ohodnocení fitness se používá pořadí jejich úspěšnosti v populaci. Tak mají díky menšímu selekčnímu tlaku větší šanci dostat se do další generace i méně úspěšní jedinci.

2.6.3 Turnajový výběr

Turnajový výběr (tournament Selection) se svou realizací inspiroval souboji mezi jedinci. Je při něm nejdříve náhodně nebo kvazináhodně vybráno N jedinců (nejčastěji $N = 2$, čím větší N, tím větší selekční tlak), kteří spolu simulovaně svedou souboj – vítězí jedinec s nejvyšší fitness a postupuje výběrem. Pro výběr jedinců se používá buď náhodný index, nebo ruletové výběry. Oproti klasickému ruletovému výběru tak mají zvýšenou šanci selekce i jedinci s horším ohodnocením.

2.6.4 Boltzmannův výběr

Tato metoda [37] se svou realizací inspirovala u techniky simulovaného žíhání [66] ve kterém je systém zpočátku ve stavu připouštějícím prozkoumávání širokého stavového prostoru

a později se snížením teploty a ustálením dostává ke globálnímu optimu. V GA tato metoda souvisí se selekčním tlakem, který je zpočátku mírný, aby se prozkoumalo široké spektrum řešení, a později stoupá, přičemž jedinci konvergují rychle k nejlepším blízkým řešením, čímž mají šanci najít globální optimum a přitom simulace neprobíhá neúměrně dlouho.

2.7 Populace

Populace v GA představuje množinu všech jedinců a tedy aktuálních řešení optimalizovaného problému. Kromě toho je sbírkou stavebních bloků (viz 2.2.1), které mohou být křížením (viz 2.5) resp. mutací (viz 2.4) potenciálně rekombinovány resp. změněny do vhodného řešení. Velikost populace s kterou GA pracuje je důležitým parametrem a je dále diskutován v kapitole 2.8.1.

Po každé iteraci algoritmu (viz 2.1) dochází k obnově rodičovské populace z populace potomků. Tento proces může probíhat řadou způsobů (nahrazení přidáním, pořadové nahrazení, nahrazení řazením, nahrazení prvního nejslabšího, náhrada bez duplicit, náhodné nahrazení...). V této práci budeme uvažovat následující postup:

1. Do rodičovské populace je elitismem (viz 2.8.4) vybrán nejlepší jedinec z populace potomků.
2. Zbylí jedinci jsou do rodičovské populace doplněni s použitím genetických operátorů (selekce, křížení, mutace)

2.7.1 Ostrovní a mřížkové modely

U ostrovních resp. mřížkových modelů je populace rozdělena na několik resp. mnoho malých subpopulací, které se vyvíjí s pouze omezenou možností interakce s ostatními subpopulacemi. Dochází tak k lepší aproximaci reálného světa a jejich implementace často dosahují lepší konvergence ke globálnímu optimu účelové funkce. Experimentálně se ukazuje [52, 8, 49], že mají v praxi velký potenciál. Ostrovní a mřížkové modely jsou podrobněji diskutovány v kapitole 4.3.

2.8 Parametry Genetického algoritmu

Parametry GA předurčují schopnost algoritmu najít správné řešení problému a dobu hledání optimálního řešení. Jejich nastavení by mělo reflektovat optimalizovanou účelovou funkci, na druhou stranu praktické řešení problémů se neobejde bez experimentování, protože neexistuje exaktní metodika jejich stanovení.

2.8.1 Velikost populace

Velikost populace v generaci jedinců patří mezi nejdůležitější parametry Genetického algoritmu. Její míra udává počet schémat (viz 2.2) v populaci a tím přímo ovlivňuje biodiverzitu populace a možnost konvergence nejlepších jedinců ke globálnímu optimu. Na druhou stranu příliš velké populace způsobují neúměrnou časovou náročnost výpočtu, protože v každé generaci dochází k velkému množství evaluace fitness funkce jedinců a stavební bloky se v populaci zbytečně opakují. Pro multimodální a mnohaparametrové účelové funkce jsou vhodné spíše větší populace zajišťující správnou konvergenci, pro jednoduché

funkce s jedním extrémem jsou velmi výhodné malé populace s elitismem (viz 2.8.4) rychle nalézající správné řešení [42].

Velikost populace se obvykle stanovuje na několik desítek až jednotek tisíc jedinců.

2.8.2 Pravděpodobnost mutace

Mutace do populace zanáší nová, potenciálně vhodná schémata (viz 2.2), ale také rozbíjí stávající. *Pravděpodobnost mutace* (mutation rate) $P_M \in \langle 0; 1\rangle$ je jeden z nejdůležitějších parametrů GA, specifikuje pravděpodobnost, že selekcí vybraní jedinci budou náhodně zmutováni (viz kap 2.4). Příliš velká pravděpodobnost mutace často rozbíjí vhodná schémata, způsobuje tak nezachování kvalitních jedinců v populaci a prodlužuje nebo dokonce úplně znemožňuje nalezení optimálních řešení. Naproti tomu příliš malá (nebo žádná) mutace zapříčiňuje malou biodiverzitu populace a statisticky tak zhoršuje nalezení globálního extrému účelové funkce. Její obvyklá hodnota se pohybuje v rozmezí asi $P_M \in \langle 0,01; 0,2\rangle$, tedy statisticky zmutuje $1\% - 20\%$ populace.

2.8.3 Míra křížení

Křížení v populaci kombinuje stávající schémata do nových (viz 2.2), ale stejně jako mutace také stávající schémata rozbíjí. *Míra křížení* (crossover rate) $P_C \in \langle 0; 1\rangle$ definuje pravděpodobnost, že dva selekcí vybraní jedinci budou pro vytvoření potomků zkříženi (viz kap 2.5). Příliš malá míra křížení zapříčiňuje pomalou konvergenci populace směrem k optimu, protože vhodná schémata vznikají primárně mutací. Příliš velká míra křížení může rozbíjet vhodná schémata a tak zapříčinit pomalou nebo žádnou konvergenci směrem ke globálnímu optimu. Někteří vědci se domnívají, že křížení je pouze forma velké mutace, jiní ho považují v Evolučních algoritmech za nepostradatelný [27]. Obvyklá hodnota míry křížení se pohybuje kolem $P_C \in \langle 0,5; 0,9\rangle$, tedy statisticky zkříží $50\% - 90\%$ selekcí vybraných párů.

2.8.4 Elitismus

Elitismus je parametr GA zaručující monotónnost fitness funkce v populaci, tedy neklesající úroveň nejlepšího jedince v jedné generaci. Dosahuje toho násilným výběrem N (často $N = 1$) nejlepších jedinců z předchozí generace do nové. Tím je zaručeno, že další generace jedinců bude obsahovat alespoň tak dobrého nejlepšího jedince, jako předchozí a nedojde mutací a křížením k jeho znehodnocení.

2.8.5 Funkce selekce

Volba vhodné selekční strategie (viz 2.6) v populaci dramaticky ovlivňuje selekční tlak a tím možnost a rychlost nalezení globálního optima v účelové funkci. Jedná se proto o důležitý parametr, který by měl být zvolen s ohledem na optimalizovaný problém.

2.8.6 Parametry ostrovních a mřížkových modelů

Míra migrace a *počet ostrovů* jsou parametry GA aplikované výhradně u ostrovních modelů. Počet ostrovů udává množství populací, které se vyvíjí spíše samostatně a v omezené míře spolu interagují skrze migraci jedinců. Vyšší počet ostrovů obecně vede k lepšímu nalezenému řešení, ale také zpomaluje konvergenci. Míra migrace určuje počet přesunutých

jedinců při migraci, *migrační interval* generační periodu, za kterou jsou jedinci mezi jednotlivými ostrovy přesunováni. S různými *způsoby migrace* se hojně experimentuje [46, 52, 8, 49]. U mřížkových modelů je situace podobná.

2.8.7 Ukončující podmínky

počet generací udává maximální dosažitelný počet generací za který GA skončí prohlášením chromozomu nejlepšího jedince za řešení problému

účelová funkce nejlepšího jedince je parametr udávající požadavek na ohodnocení nejlepšího jedince v populaci, jestliže je některý chromozom ohodnocen účelovou funkcí lépe než je tato hranice, algoritmus končí

minimální změna účelové funkce je parametr související s konvergencí populace k optimu. Udává minimální změnu ve statistickém ohodnocení populace, která je potřeba na to, aby algoritmus dále iteroval. Různé implementace používají různé statistiky (průměrnou fitness, počet generací kdy se nejlepší řešení nezměnilo a pod.)

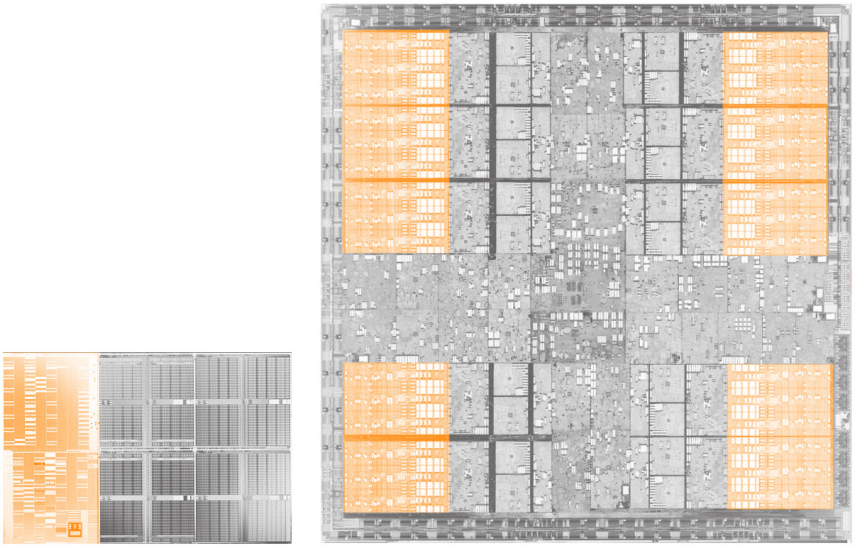
časové omezení se nepoužívá často, ale je možné ho využít pro ohodnocení poměru rychlosti konvergence ku časové náročnosti různých metod.

Výše uvedené ukončující podmínky se také často kombinují.

Kapitola 3

Využití GPU pro obecné výpočty

Tato kapitola prezentuje čtenáři vývoj a současnost grafických karet, jejich architekturu, výkonnost a trend vývoje. V další části popisuje možnosti využití programovatelné grafické pipeline pro obecné výpočty, ke konci jsou diskutovány knihovny pro GPGPU se speciálním zaměřením na nVidia CUDA.



Obrázek 3.1: Porovnání fyzické velikosti čipu CPU a GPU s vyznačenými jádry

3.1 Historie

Grafické karty byly původně určeny pouze jako D/A převodník dat z videopaměti na grafický výstup. Začátkem 70.let [68] vznikla myšlenka akcelerace rasterizačních operací s použitím čipu na grafické kartě. Následně některé firmy implementovaly vykreslování grafických primitiv, jako jsou úsečka a kružnice, do grafické karty. První 3D akcelerátory spatřily světlo světa počátkem 90.let, umožňovaly akceleraci zpracování polygonové sítě a nanášení textur na ně. Krátce poté se 3D akcelerátory začaly prosazovat do domácností, kde je velká hráčská komunita podporuje dodnes. Všechny grafické karty té doby měly

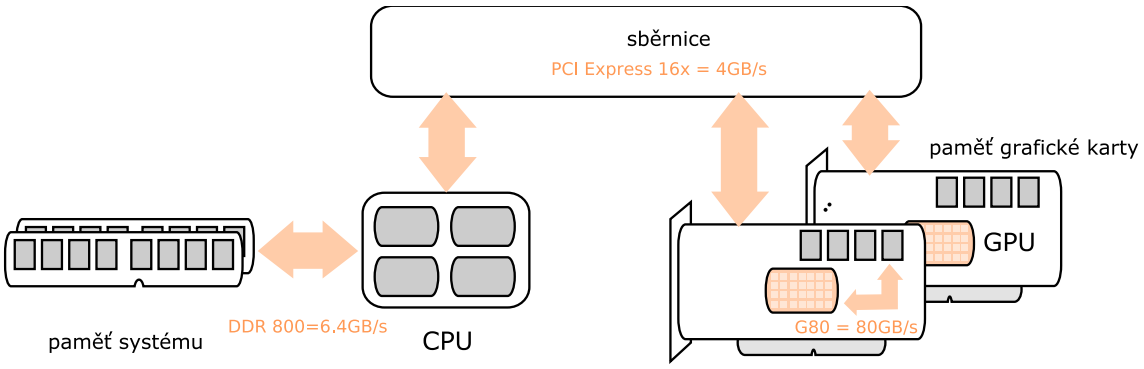
fixní vykreslovací řetězec bez možnosti provádět v GPU uživatelsky definované programy. V roce 2001 však firma nVidia uvedla na trh kartu GeForce 3, která odstartovala revoluci ve zpracování realtime grafiky – první programovatelný GPU. Od té doby rychle narůstá univerzalita, komplexnost i výkon grafických čipů.

Obrázek 3.1 porovnává fyzickou velikost jádra CPU Intel Penryn a GPU nVidia GT200 (oba vyrobené 65nm technologií) – je zřejmé, že grafický čip je komplexnější a větší plocha je v něm vyhrazena pro zpracování ALU operací, což jej předurčuje pro výpočetně-intenzivní aplikace.

Do budoucna je očekáván velký nárůst univerzality GPU, o čemž svědčí fakt, že firma Intel v současnosti pracuje na architektuře Larrabee [47], která bude plně programovatelným x86 kompatibilním mnohajádrovým procesorem s orientací na realtime grafiku.

3.2 Architektura grafických karet

3.2.1 Propojení s hostitelským systémem a přenosové rychlosti



Obrázek 3.2: Architektura PC a přenosové rychlosti

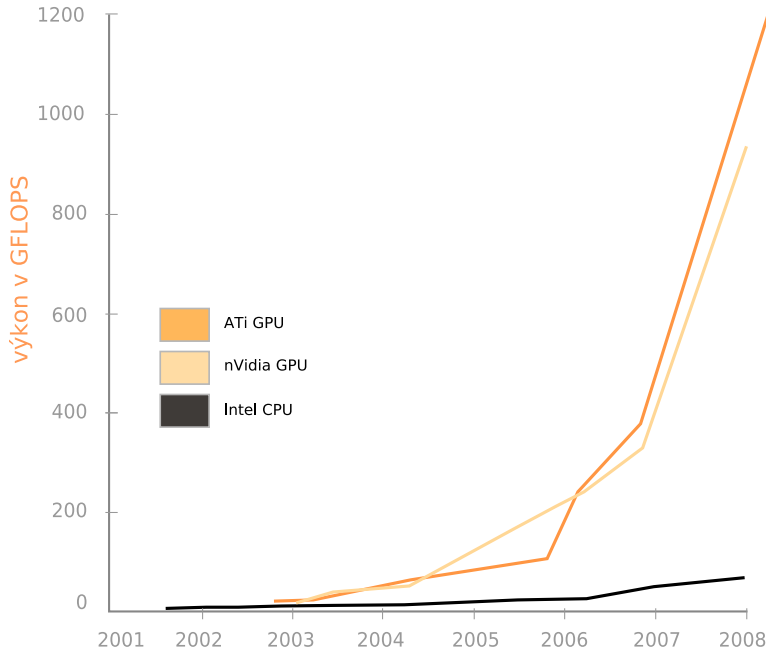
Jak ilustruje obrázek 3.2, grafická karta je do systému připojena formou externího adaptéru (v systému mohou být přítomny až 4), pro komunikaci s okolím využívá sběrnici. Typicky se v současnosti (2008) jedná o PCI-Express 16x s teoretickou přenosovou rychlostí 4GB/s, do budoucna nastoupí jeho dvojnásobně rychlá varianta PCI-Express 2. Paměť systému s procesorem komunikuje rychlostí v řádu jednotek GB/s, zatímco paměť grafické karty bývá mnohem rychlejší (u G80 80 GB/s), protože je potřeba zásobovat velké množství stream jednotek daty.

Jak je patrné, přesun dat lokálních pro procesor na grafickou kartu a naopak tedy naráží na úzké hrdlo, kterým je sběrnice propojující tyto komponenty. Tento problém se dá kompenzovat přesunutím veškerého zpracování na grafickou kartu nebo prokládáním výpočtu s přesunem dat[1].

3.2.2 Výkon

Grafické čipy jsou již od počátků programovatelnosti optimalizovány pro zpracování typu SIMD (Single Instruction Multiple Data, jedna operace nad mnoha daty) [36]. Takové zpra-

[1] prokládání přenosu dat s výpočtem na GPU umožňuje s použitím CUDA pouze novější hardware GT200 a výše



Obrázek 3.3: Srovnání vývoje teoretické výkonnosti CPU a GPU

cování je dobře paralelizovatelné, protože obsahuje minimální nebo žádné datové závislosti, zpracování tedy může probíhat v libovolném pořadí a na maximálním počtu jednotek. GPU také historicky potřebují jen minimální podporu cyklů a větvení. Tento trend s rozvojem velmi komplexní realtime grafiky pomalu upadá, nicméně i na moderních čipech je znatelný výkonový nárůst při eliminaci větvení [33, 40].

Tabulka 3.1: Porovnání komplexnosti některých CPU a GPU

čip	**tranzistorů** $\cdot 10^6$
AMD Athlon 64 X2 CPU	154
Intel Core 2 Duo CPU	291
Intel Pentium D 900 CPU	376
ATI X1950 XTX GPU	384
Intel Core 2 Quad CPU	582
nVidia G8800 GTX GPU	670
nVidia GTX 280 GPU	1400

Komplexnost grafických čipů v posledních letech velmi rychle stoupá, jak prezentuje tabulka 3.2.2. Nárůst výkonu probíhá cestou masivní paralelizace, nové GPU mají na čipu stovky jednotek schopných zpracovávat velké množství dat ve stejný čas. Mají tedy obrovský teoretický výkon (jak prezentuje obrázek 3.3), který je však dosahován jen s použitím velké paralelizace výpočtu [23]. Z tohoto faktu také plyne nízký výkon na jedno vlákno, na druhou stranu správa vláken je v kompetenci hardware GPU, a proto je režie vytváření a přepínání kontextu vláken nulová (což u CPU neplatí).

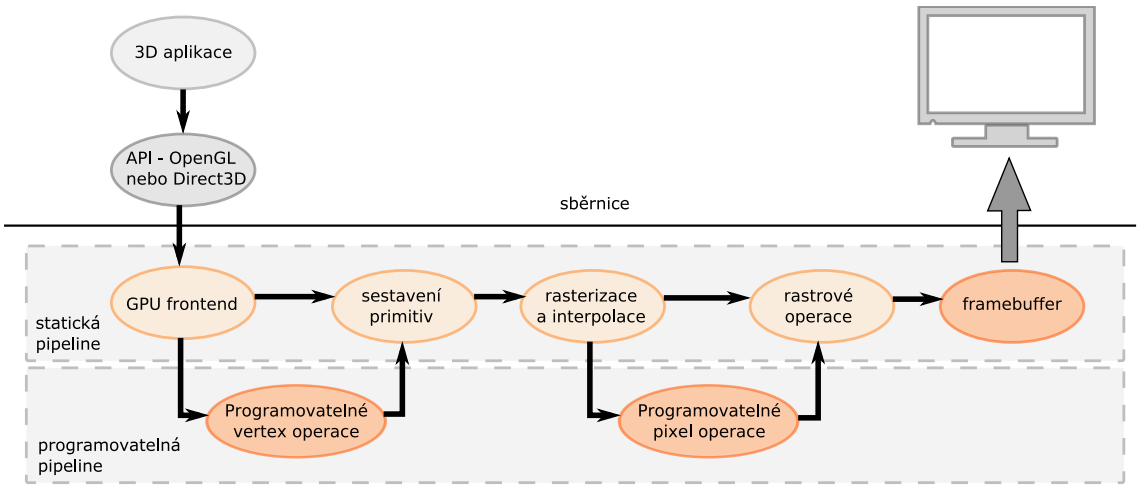
3.2.3 Datové typy

GPU historicky pracují se spojitým 3D prostorem ve kterém používají datovými typy s plovoucí desetinnou čárkou. Nejnovější GPU (GT200, 2008) podporují kromě 32bitové přesnosti výpočtu také dvojitou přesnost a přibližují se tak běžným procesorům. Kromě nich jsou v GPU podporovány celočíselné datové typy (fixed, většinou emulací z desetinných), malé celočíselné (half) a skaláry složené z těchto typů. Pointery jsou využívány výhradně jako odkazy na data textur a programy, nelze vytvářet vlastní datové struktury, jak je běžné u CPU. Z tohoto důvodu jsou některé implementace na GPU složitější, některé efektivně nelze realizovat vůbec.

3.2.4 Shrnutí použití GPU

- **výhody**
  - Obrovský teoretický výkon (přes 1TFLOP) za relativně malou cenu
  - Hardwarová plánování vláken
  - Méně obvodů pro logiku, více pro ALU a s tím spojený výkon na Watt
  - Vysoká propustnost paměti, rychlá sdílená paměť v rámci multiprocesoru
  - Externí adaptér, škálovatelnost
- **nevýhody**
  - Špatná nebo žádná podpora větvení, omezené datové typy
  - Omezená přesnost výpočtu
  - Nízký výkon na jedno vlákno, vyžaduje masivně paralelní problémy

3.3 Výpočty s využitím programovatelné grafické pipeline



Obrázek 3.4: Grafická pipeline

Vykreslovací řetězec prezentuje obrázek 3.4. Jak je zřejmé, vykreslování začíná v aplikaci využívající některou z dostupných grafických knihoven (viz 3.3.1). Posloupnost volání

knihovny je s prostřednictvím ovladače grafické karty v OS převedena na příkazy a data grafického čipu, které jsou po sběrnici zaslány do grafické karty (GPU frontend). Statický vykreslovací řetězec pak z těchto informací sestaví grafická primitiva (polygony, elipsy, úsečky, ...), které následně s použitím interpolace vykreslí (rasterizace). Závěrečnou částí je provádění rastrových operací odkud jsou výsledná data kopírována do framebufferu[2]. Odtud jsou RAMDAC převodníkem konvertována na obrazový signál a zasílána na monitor.

Programovatelná pipeline uvádí možnost uživatelsky definovaných operací nad každým vertexem (vrcholem grafického primitiva) a pixelem (také zvaný fragment). Tyto masivně paralelizovatelné operace jsou využívány pro všechny pokročilé efekty v realtime grafice.

Novější grafické karty navíc umí operovat s framebufferem a například vykreslený obraz použít jako texturu[3] pro 3D objekt, nebo jej zkopírovat přes sběrnici zpátky do aplikace. Toho se dá s výhodou využít právě pro GPGPU výpočty s použitím programovatelné grafické pipeline.

Tabulka 3.2: Porovnání vybraných vlastností různých verzí Pixel Shaderů

verze pixel shaderu	**2.0**	**2.0a**	**2.0b**	**3.0**	**4.0**
instrukčních slotů	32 textura + 64 ALU	512	512	≥512	≥65536
vykonaných instrukcí	32 textura + 64 ALU	512	512	65536	nelimitováno
pomocné registry	12	22	32	32	4096
konstantní registry	32	32	32	224	16x4096

3.3.1 Grafické knihovny

V současnosti jsou využívány dvě hlavní knihovny pro realtime grafiku:

DirectX [55] je knihovna firmy Microsoft určená výhradně pro OS Windows[4] a programovací jazyk C++. Kromě akcelerace 3D grafiky (Direct3D) umožňuje i unifikovaný přístup k ovládacím prvkům (DirectInput), zvukovým zařízením (DirectSound) a dalším funkcím vhodných především pro programátory her. Je využívána v nejnovějších herních titulech, kde jako první poskytuje novou funkcionalitu grafických čipů. Microsoft nyní ve spolupráci s výrobci GPU vydává specifikace nových verzí DirectX a určuje tak směr a tempo vývoje grafických čipů. Nastala tak paradoxní situace, kdy se podle specifikace/implementace software vytváří hardware. Poměrně často se používá verze DirectX, jehož plnou podporu funkcionality daný GPU má, pro vyjádření míry pokročilosti grafického hardware.

DirectX verze 8 uvedla možnost využití programovatelné grafické pipeline s použitím pixel resp. vertex shaderů (PS resp. VS). V první verzi PS a VS 1.0 bylo možné provádět nad pixely a vertexy pouze několik málo instrukcí, v pokročilejších verzích,

[2] framebuffer je paměť určená k přímému zobrazení na monitor, ve 3D grafice se většinou používají dva, které se po každém snímku prohodí, tj. jeden je zobrazen uživateli na monitoru a do druhého probíhá vykreslení nového snímku

[3] textura je mapa bodů, které jsou kvůli vysoké komplexnosti nanášeny formou „malování“ na grafická primitiva při vykreslování. Je tak možné dosáhnout realistického vzhledu povrchů s použitím malé komplexnosti rasterizace.

[4] s použitím implementace Windows knihoven WINE je možný provoz i na jiných OS, běh je však pomalejší a nepodporuje všechny funkce

které nastartovaly revoluci ve zpracování realtime grafiky, se funkcionalita výrazně zvýšila, jak ukazuje tabulka 3.3.

Verze 9 spatřila světlo světa v roce 2002 a postupnou evolucí se do verze 9.0c udržela na výsluní až současnosti (2008), kdy je nejnovější podporovanou verzí na systémech Windows XP. Představila především značné vylepšení Pixel a Vertex shader modelu až do verze 3.0. U většiny vysokorozpočtových herních titulů je podporována zároveň s verzí 10.

DirectX verze 10 (dostupná pouze na Windows Vista) kromě kompletního přepracování knihovny zavedla sjednocení pixel a vertex shaderů 4.0 do tzv. unified shaderů. Počínaje architekturou G80 tak mají grafické čipy shodné jednotky pro všechny operace a přibližují se tak spíše obecné multiprocesorové ALU jednotce, než specializovanému čipu.

Verze 11 kromě GPU akcelerace reálné fyziky ve hrách plánuje zavést novinku v podobě compute shaderu, programu pro ovecné GPGPU výpočty, což ještě více potvrzuje vývoj GPU směrem k obecnému použití. Tato verze bude formou instalace v budoucnu dostupná i pro Windows Vista, její uvedení se však plánuje až s příchodem Windows 7 [72].

Knihovna DirectX používá pro popis shaderů na vysoké úrovni programovací jazyk HLSL (High Level Shading Language) [63], který je syntakticky téměř shodný s Cg používaným v knihovně OpenGL. V novějších verzích však budou uvedeny další změny oproti Cg.

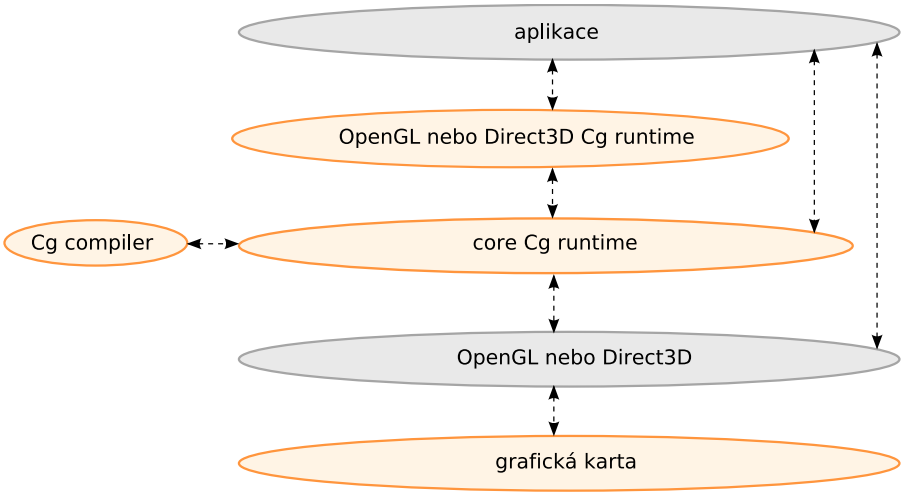
OpenGL (Open Graphics Library) [1] je knihovna vyvíjená konsorciem firem KHRONOS stojícími i za dalšími otevřenými standardy jako je OpenCL (viz 3.4.4), OpenMP a další. Je založená čistě na programovacím jazyce C a je navržena pro maximální míru přenositelnosti. Obsahuje pouze funkcionalitu přímo spojenou s prací na GPU, pro obsluhu klávesnice, tvorbu komplexnějších objektů, načítání souborů a další funkce se používají doplňkové knihovny, například GLUT (openGL Uility Toolkit) [65]. Pro vyjadřování funkcionality hardware, na kterém je knihovna provozována, používá skupinu tzv. extensions, což jsou řetězce specifikující konkrétní funkce, které daný hardware umožňuje provádět bez emulace softwarem.

OpenGL používá pro popis shaderů od verze 2.0 GLSL (openGL Shading Language) [61], který má však mezi vývojáři relativně malou podporu. Kromě toho je možné shadery popisovat s použitím programovacího jazyku Cg, kterému se dále budeme věnovat podrobněji.

3.3.2 Jazyk Cg

První shadery byly jen několik instrukcí dlouhé, ručně psané programy v asembleru grafické pipeline konkrétního čipu. S vývojem shaderů však brzy přišla nutnost vysokoúrovňového, hardwarově-nezávislého popisu. Iniciativy v tomto směru se chopila firma nVidia, která vyvinula specifikaci a s ní i první překladače programovacího jazyka Cg (C for Graphics) [53]. Jedná se o vysokoúrovňový, na platformě nezávislý popis shaderů pro grafické karty, který později vytvořením překladačů pro svůj hardware adaptovala i firma ATi.

Jak je patrné na obrázku 3.5, překladač Cg působí v runtime fázi grafické knihovny OpenGL nebo Direct3D. Protože až při spuštění aplikace je zřejmé, na jakém hardware shader poběží, kompilace probíhá dynamicky při spouštění aplikace [38]. Výrobci grafických



Obrázek 3.5: Architektura prostředí Cg

čipů si pak dodávají vlastní Cg překladače, většinou jako součást balíku ovladačů DirectX nebo OpenGL.

Cg používá syntaxi [10] podobnou programovacímu jazyku C, avšak reflektuje rozdíly GPU proti CPU. Podporuje proto jenom omezenou škálu datových typů a neumožňuje vytváření nových, sémanticky používá implicitně skalární operace a má velmi omezené použití pointerů.

3.3.3 Výpočet

Samotný GPGPU výpočet probíhá následovně:

1. Inicializace grafické knihovny a alokace textur, které jsou použity jako vstupní data
2. Nastavení scény tak, aby jeden bod textury odpovídal jednomu bodu počítaného obrazu
3. Přenos dat z CPU do GPU příkazy pro přiřazení textury
4. Spuštění shaderu provádějícího nad texturou požadované operace pomocí rasterizace do paměti grafické karty (nikoliv vykreslením na monitor)
5. Přenos dat z GPU do CPU s použitím čtení z paměti grafické karty

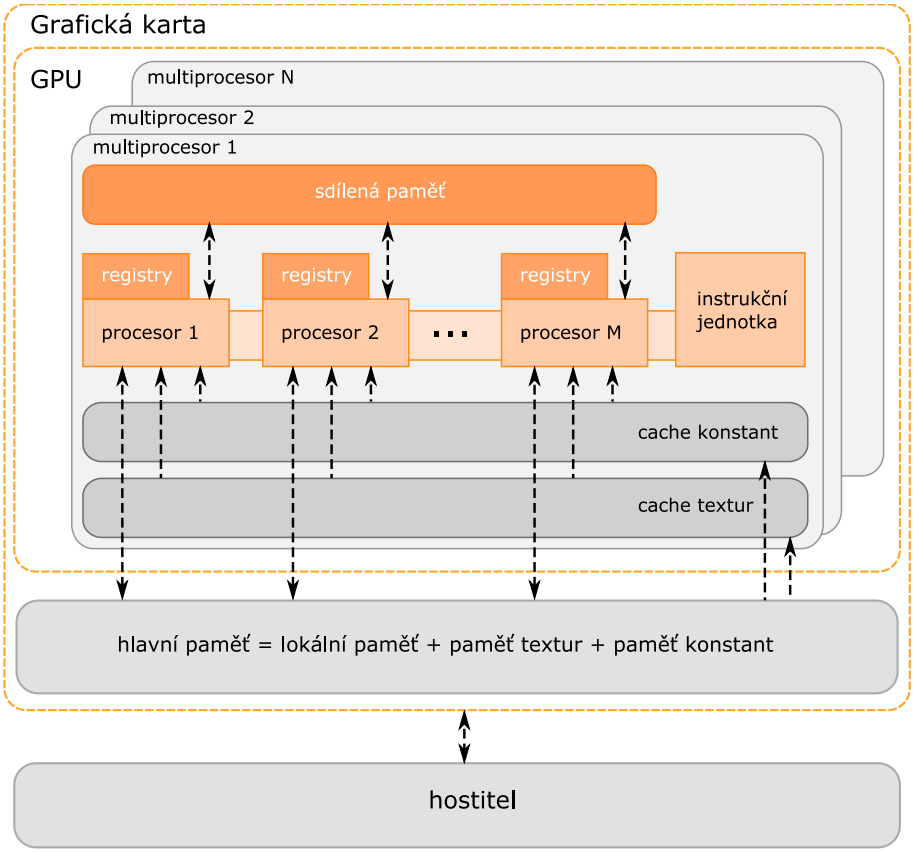
Příklad zdrojového kódu pro OpenGL a Cg je uveden v příloze A [35].

3.4 Specializované knihovny pro GPU výpočty

Perspektivnost GPGPU výpočtů dala vzniknout řadě knihoven, které abstrahují grafický hardware a umožňují jednodušší popis výpočtů s pomocí GPU. Některé knihovny jsou implementovány na konkrétní hardware (CUDA viz 3.4.1, AMD Brook viz 3.4.2), jiné využívají grafické knihovny a shadery pro platformní nezávislost (GPUTech EcoLib viz 3.4.3), rýsuje se i vývoj nových programovacích jazyků (OpenCL viz 3.4.4).

3.4.1 nVidia CUDA

CUDA (Compute Unified Device Architecture) je knihovna firmy nVidia určená pro obecné výpočty s použitím GPU. Funguje pod OS Windows a Linux. Podporuje pouze grafické karty této firmy a je založená na syntaxi C s některými rozšířeními [21]. Zpočátku byla tato knihovna určena pro použití ve specializovaných grafických kartách určených pro GP-GPU, později byla uvolněna i pro volné použití ve veřejné sféře na běžných desktopových systémech. S výhodou se tak dá využít cenově dostupného hardware pro akceleraci výpočtů prostřednictvím GPU.



Obrázek 3.6: Hardwarový model CUDA knihovny a grafických karet nVidia

Obrázek 3.6 ilustruje hardwarový model knihovny CUDA, kterou používají grafické karty firmy nVidia. Jak je patrné, grafická karta je z hlediska této hierarchie rozdělena na externí paměť a grafický čip (GPU). Hostitelský systém s kartou komunikuje prostřednictvím zápisu a čtení do/z hlavní paměti. Grafický čip je rozdělen do řady multiprocesorů z nichž každý obsahuje svou cache konstant a textur, instrukční jednotku rozdělující práci mezi jednotlivé procesory a společnou sdílenou paměť pro všechny procesory v rámci multiprocesoru. Tato architektura má výhodu ve škálovatelnosti, tj. levnější čipy obsahující méně multiprocesorů mohou být z hlediska software popisovány stejně jako dražší s více multiprocesory.

Paměť je strukturována následovně:

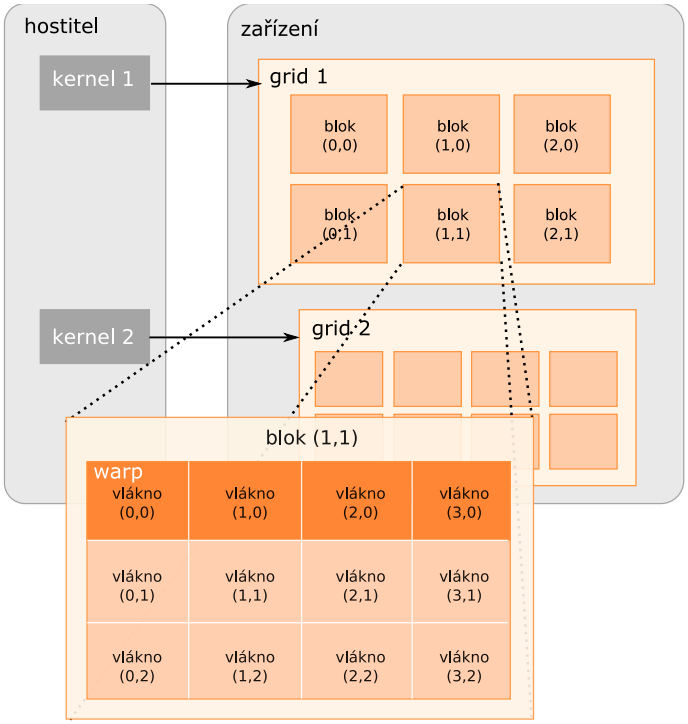
hlavní paměť je nejpomalejší a největší datové úložiště v rámci grafické karty, slouží

jakožto komunikační médium s hostitelem a jsou v ní uchovány textury, konstanty i pracovní data vláken. U architektury G80 je kapacita až 1.5GB (u desktopových verzí 768MB) a propustnost 80GB/s.

cache konstant a textur slouží jako rychlá vyrovnávací paměť mezi procesory a hlavní pamětí, u G80 má kapacitu 64KB, je pouze pro čtení a má nízkou přístupovou dobu. Má větší hit-ratio v případě, že data jsou načítána ve 2D sousedící lokalitě [23].

sdílená paměť na multiprocesoru je paměť kapacity 16KB v rámci každého multiprocesoru do které mají přístup všechny procesory v jednom bloku jak pro čtení, tak pro zápis. Jedná se o velmi vhodnou paměť pro pracovní data, v případě správné optimalizace synchronizace s hlavní pamětí může díky ní dojít k razantnímu urychlení celé aplikace.

registry jsou, stejně jako u CPU, pracovním prostorem každého procesoru, rychlost a propustnost jsou nejvyšší v hierarchii, kapacita je však nejmenší.



Obrázek 3.7: Softwarová hierarchie v knihovně CUDA

CUDA hierarchii z hlediska softwarové stránky ilustruje obrázek 3.7 [7]:

vlákno je nejmenší jednotkou provádění v knihovně, na rozdíl od klasických vláken v OS má téměř nulovou režii vytvoření a přepnutí, cenou je relativně nízký výkon. Pro využití výkonu GPU je potřeba až tisíců aktivních vláken [17].

warp je několik vláken vykonávaná fyzicky paralelně. Skupina warpů je vykonávána logicky paralelně, tj. nemá definované pořadí provádění.

blok vláken je skupina warpů vykonávaná na jednom multiprocesoru, může sdílet data a knihovna podporuje jejich rychlou hardwarovou synchronizaci.

grid je skupina bloků vláken na které běží logicky paralelně jeden CUDA program (kernel). Grid nemůže být implicitně synchronizován, avšak s použitím více kernelů (které nejsou vykonávány paralelně) toho lze docílit.

sériová implementace

```
void saxpy(uint n, float a,
                   float *x, float *y)
{
 for (uint i = 0; i<n; ++i)
    y[i] = a*x[i] + y[i];
}


void serial_sample ()
{ // zavolej SAXPY funkci
    saxpy (n, 2.0, x,y);
}
```

paralelní implementace

```
__global__ void saxpy(uint n, float a,
                              float *x, float *y)
{
 uint i = blockIdx.x*blockDim.x
           +threadIdx.x;
  if( i < n ) y[i] = a*x[i]+y[i]
}

void parallel_sample ()
{// zavolej paralelni SAXPY kernel
  // s pouzitim [n/256] bloku, kazde po
  // 256 vlaknech
    saxpy<<<ceil(n/256),256>>>(n,2,x,y);
}
```

Obrázek 3.8: Porovnání zdrojového kódu sériové a paralelní implementace s využitím CUDA

Obrázek 3.8 [17] ilustruje syntaktické rozšíření jazyka C pro popis paralelismu a mapování do paměti:

- Volání funkce `funkce<<<bloku,vlaken>>>(param1,param2,...,paramN)` je použito pro popis paralelismu. Syntakticky zapisuje spuštění definovaného počtu výpočetních bloků a počtu vláken na blok paralelně. Jak je patrné, lze takto snadno vyjádřit zpracování typu SIMD.
- `__shared__` se používá jako prefix proměnné a vyjadřuje uložení ve sdílené paměti
- `__global__` jako prefix funkce vyjadřuje, že se jedná o funkci volanou na hostitelském systému a vykonávanou na zařízení

Detailnější popis rozšíření jazyka C poskytuje [11].

Dobu trvání některých operací nastiňuje tabulka 3.4.1 [7]. Jak je patrné, latence hlavní paměti je proti době vykonávání ALU instrukcí velmi vysoká, je proto vhodné použít pro data spíše rychlou sdílenou paměť v rámci multiprocesoru než globální paměť pro všechny multiprocesory společně. Z tohoto faktu také plyne, že pro využití potenciálu GPU jsou potřeba výpočetně-intenzivní aplikace.

S pomocí nVidia CUDA byly dosaženy reálné urychlení na GPU až 100x proti CPU [33]. Vzhledem k vysoké optimalizaci, velké komunitě vývojářů [4] a dostupnosti na běžný hardware byla vybrána pro implementaci.

3.4.2 AMD Brook+

AMD Brook+ (součástí Stream SDK) [6] je knihovna pro akceleraci výpočtů pro grafické karty bývalé firmy ATi. Donedávna byla určena pro využití na specializovaných akcelerátorech FireStream, v roce 2008 však byla uvolněna pro volné použití na desktopových

Tabulka 3.3: Doba trvání vybraných operací v architektuře G80

instrukce	**počet cyklů na warp**
FADD, FMUL, FMAD, IADD, bitové operace	4
SQRT	16
přístup do registrů	0
přístup do sdílené paměti	≥4
čtení z hlavní paměti	400-600
čtení z paměti konstant a textur	≥0 (z cache), jinak 400-600
synchronizace vláken v rámci bloku	4+potenciální čekání

řadách grafických akcelerátorů. Stejně jako CUDA používá rozšíření jazyka C pro vyjádření paralelismu a tzv. kernely pro mapování funkcí na GPU. Mezi vývojáři zatím není tolik rozšířena a dosahuje dobré výsledky při konverzi videa.

3.4.3 GPU-Tech EcoLib

Je komerční knihovna [18] firmy GPUTech využívající programovatelnou grafickou pipeline (viz 3.3). S použitím objektového návrhu abstrahuje zápis a čtení do/z framebufferu a odstiňuje tak režii kolem použití grafických knihoven při per-fragment operacích. Dosahuje řádových urychlení GPU u generování náhodných čísel, Monte Carlo metody a dalších.

3.4.4 OpenCL

Skupina KHRONOS, čítající více něž 100 největších firem působících v oblasti hardware a software, odpovědná za takové standardy jako je OpenGL (Open-Graphics Library [1]) nebo OpenMP (Open-Multi Processing [2]) v současnosti (konec 2008) dokončuje specifikaci nového programovacího jazyka OpenCL (Open-Compute Language) [48]. Plánuje v něm sjednotit popis výpočtu na grafických kartách a vícejádrových procesorech. Jazyk je založen na syntaxi C s rozšířeními podobnými CUDA syntaxi, používá standardní datové typy a odstiňuje konkrétní hardware na kterém program poběží popisem pomocí výpočetních jednotek (compute kernels). Knihovna také umožňuje snadné plánování paměti a výpočtu. Firmy ATi a nVidia se vyjádřily k podpoře tohoto jazyka kladně a společně s ostatními 5. prosince 2008 dokončily specifikaci 1.0, na základě které budou moci vytvořit OpenCL překladače pro svoje grafické karty. Tento jazyk tak do budoucna slibuje jednotný, plaformně nezávislý popis paralelních výpočtů jak na procesorech, tak na grafických kartách.

Kapitola 4

Návrh systému

Kapitola, kterou právě čtete, diskutuje problematiku návrhu výsledného systému. V první části je prezentován výběr použité platformy a identifikována nejnáročnější část Genetického algoritmu. Následuje popis teoretických možností paralelizace GA, aby byla v zápětí podrobně popsána použitá varianta. V závěru kapitoly jsou diskutovány dílčí problémy GPU implementace – generování náhodných čísel a efektivní řazení prvků. Nechybí ani popis umělých testovacích funkcí použitých pro analýzu výsledků.

4.1 Platforma

Vzhledem k povaze GA a platforem umožňujících jejich akceleraci pomocí GPU byl zvolen programovací jazyk C++. Splňuje následující vlastnosti:

- Podporuje všechny nejrozšířenější grafické knihovny i knihovny pro akceleraci obecných výpočtů na GPU
- Je kompilovaný a dostatečně nízkoúrovňový pro možnost pokročilé optimalizace, poskytuje dobrý základ pro výkonnou implementaci
- Zdrojové kódy jsou snadno přenositelné na jiné platformy
- Umožňuje objektový návrh a s ním dobrou znovupoužitelnost kódu v jiných aplikacích

Jako knihovna pro výpočty pomocí GPU byla zvolena nVidia CUDA, mimo jiné má tyto vlastnosti:

- Je použitelná na běžně dostupném hardware, použití je přímočaré a efektivní
- Má dostatečně odladěnou implementaci, širokou sbírku ukázek zdrojových kódů a největší komunitu vývojářů
- S pomocí CUDA byly dosaženy dosud nejlepší výsledky v GPGPU oblasti díky blízké vazbě na konkrétní hardware (sdílená paměť v rámci multiprocesoru) a vysoké optimalizaci

Tabulka 4.1: Nastavení Genetického algoritmu při profilaci

parametr	**hodnota**
počet generací	100 000
velikost populace	128 jedinců
optimalizovaná funkce	Griewank (viz 4.10), reálný chromozom, 2 geny (rozměry)
křížení	jednobodové (viz 2.5), pravděpodobnost 0.9
mutace	Gaussovské (viz 2.4), pravděpodobnost 0.01
selekce	turnajový (viz 2.6)
elitismus	vypnutý

Tabulka 4.2: Porovnání profilování běhu vlastní implementace GA a GALibu

GALib

čas [%]	**funkce**
34.21	`garan2()`
11.90	`GARouletteWheelSelector::select()`
6.88	`GAPopulation::QuickSortDescendingRaw()`
6.09	`GAGenome::evaluate()`
5.56	`GAPopulation::QuickSortDescendingScaled()`
4.10	`GAGenome::fitness()`
2.98	`GAGenome::score()`
celkem 11.9s	

Vlastní implementace

čas [%]	**funkce**
59.77	`CPUGeneticSolver::RouletteSelectionFunction()`
9.69	`CPUGeneticSolver::GetThisPopulationSize()`
6.90	`CPUGeneticSolver::CrossoverIndividuals()`
5.90	`CPUGeneticSolver::GriewankEvaluator()`
3.48	`CPUGeneticSolver::GetThisPopulationSize()`
2.98	`CPUGeneticSolver::Solve()`
2.86	`CPUGeneticSolver::Random()`
celkem 9.10s	

4.2 Identifikace nejnáročnější části GA

Pro identifikaci nejnáročnější části Genetického algoritmu byla použita standardní utilita `GNU Profiler` [15], konkrétně výstup prezentující relativní dobu vykonávání jednotlivých funkcí. Měření probíhalo na CPU verzi realizované v rámci této práce, jako reference posloužila implementace s využitím knihovny `GALib` [51]. Testovací hardware je uveden v kapitole 5.5, parametry GA pak shrnuje tabulka 4.1.

Jak je patrné z tabulky, kromě běžně užívaných hodnot parametrů pravděpodobnosti křížení a mutace byl zvolen relativně velký počet generací, aby byla profilace statisticky kvalitní. Celkový čas byl měřen (na neprofilové verzi) Unixovou utilitou `time`, která slouží pro přibližné porovnání rychlosti obou implementací, kromě samotného běhu algoritmu zahrnuje také dobu inicializace OS.

Výstup profilace pro funkce, ve kterých řízení programu strávilo nejvíce času, shrnuje tabulka 4.2. Je zřejmé, že v obou implementacích pro nastavené parametry hraje významnou

Tabulka 4.3: Profilování běhu vlastní implementace pro turnajový výběr a Mersenne-Twister

čas [%]	**funkce**
15.44	`CPUGeneticSolver::GriewankEvaluator()`
11.99	`MTRand::rand()`
10.60	`MTRand::randInt()`
8.07	`CPUGeneticSolver::CrossoverIndividuals(()`
5.53	`MTRand::reload()`
5.30	`CPUGeneticSolver::TournamentSelectionFunction()`
4.26	`MTRand::twist(()`
4.15	`CPUGeneticSolver::UpdatePopulationNormFitnessFunction()`
celkem 6.5s	

roli selekce. Vlastní implemetnace je přibližně o čtvrtinu rychlejší, protože není tak robustní a obsahuje horší generátor náhodných čísel. Ten v případě GALibu představuje celou třetinu doby běhu programu. Evaluace fitness funkce hraje v obou případech zanedbatelnou roli, avšak praxe ukazuje, že pro reálné řešení je tomu přesně naopak [32]. Pro jednoduché optimalizační problém je však patrný především význam selekce a doprovodných funkcí jako je řazení.

V případě testování stejných parametrů a turnajového výběru spolu s kvalitním generátorem náhodných čísel MersenneTwister se situace mění, jak prezentuje tabulka 4.3. Většinu času běhu zabere vyhodnocení PRNG a fitness funkce, implementace se zrychluje téměř o polovinu. Protože GPU mají velký potenciál akcelerovat hlavně matematicky-intenzivní výpočty, je vhodné použít rychlý paralelní generátor náhodných čísel a turnajový výběr.

4.3 Paralelizace Genetického Algoritmu

4.3.1 Modely

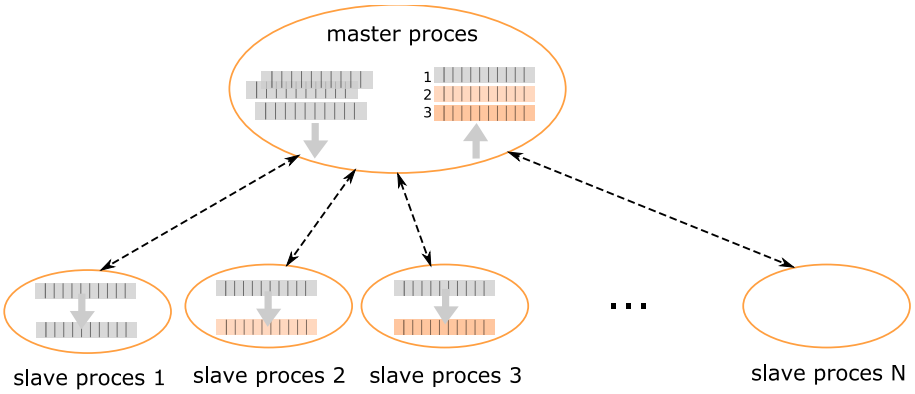
Klasický sekvenční model Genetického Algoritmu předpokládá jedinou populaci jedinců (globální model), kteří mezi sebou bez restrikcí interagují. U paralelního modelu genetického Algoritmu (*Parallel Genetic Algorithm*, PGA) [43] se používá buď více populací (ostrovní model) s omezenou interakcí mezi jedinci různých populací, nebo jedna populace rozdělaná na mnoho subpopulací.

4.3.2 Paralelizace globálního modelu

V tomto přístupu (také zvaný *Farming model*) je paralelizováno zpracování klasické verze globálního modelu:

- Evaluace jedinců není závislá na ostatních jedincích, a proto je snadno paralelizovatelná. Jedná se také o jednu z nejnáročnějších částí zpracování jedné iterace algoritmu, proto skýtá velký potenciál pro urychlení celého výpočtu.
- Použití genetických operátorů obecně vyžaduje datovou synchronizaci s ostatními procesy zpracovávajícími populaci, aby nedošlo k nekonzistenci. U selekce a následného křížení je možné brát v potaz každého jedince zvlášť a jeho šanci se reprodukovat se všemi ostatními.

V případě synchronní implementace paralelizace globálního modelu jsou všechny procesy zpracovávající populaci synchronizovány na konci každé iterace algoritmu. Výsledek je pak stejný, jako u sekvenční verze algoritmu. Asynchronní běh se svými výsledky od sekvenční verze liší, avšak může být vhodný pro implementaci.



Obrázek 4.1: Příklad implementace paralelizace globálního modelu

Jednu z možných implementací [41] paralelizace globálního modelu ilustruje obrázek 4.1:

master proces udržuje kompletní populaci jedinců, zajišťuje synchronizaci a rozesílá práci ostatním procesům. V některých implementacích také zajišťuje křížení a mutaci

slave procesy slouží primárně k evaluaci fitness funkce jedinců, přijmají části populace od master procesu a vrací mu ohodnocené jedince

Master i slave procesy mohou pracovat na jednom nebo více fyzických počítačích. Nevýhodou této implementace je vysoké zatížení master procesu a nutnost časté datové synchronizace mezi procesy, výhodou pak urychlení plynoucí z paralelní evaluace fitness funkce jedinců, která patří mezi nejnáročnější část výpočtu.

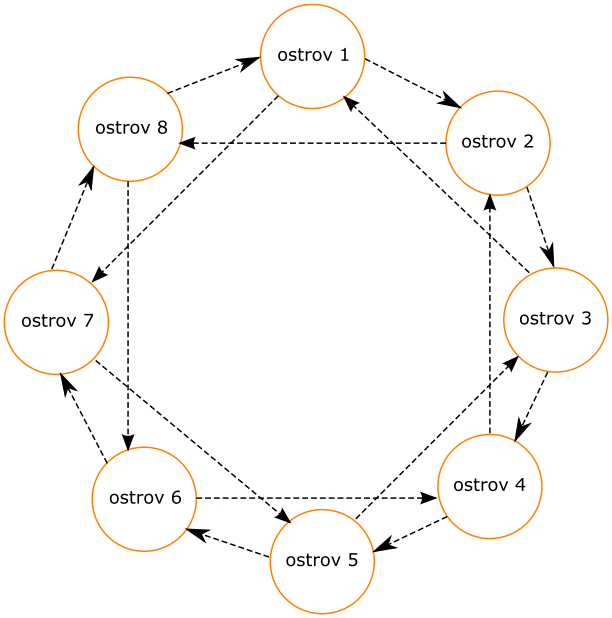
4.3.3 Hrubá granularita

U hrubé granularity (*Coarse-Grained PGA*, také zvaný Migrační model) se využívá rozdělení do několika populací ostrovního modelu, které se vyvíjejí spíše izolovaně, pouze občas si mezi sebou vymění tzv. migrací jedince. Přidělení hardwarových prostředků pak závisí na návrhu, často se používá jeden ostrov pro jedno fyzické vlákno programu. Počet migrovaných jedinců a způsob migrace je předmětem experimentování [26], příkladem je obrázek 4.2.

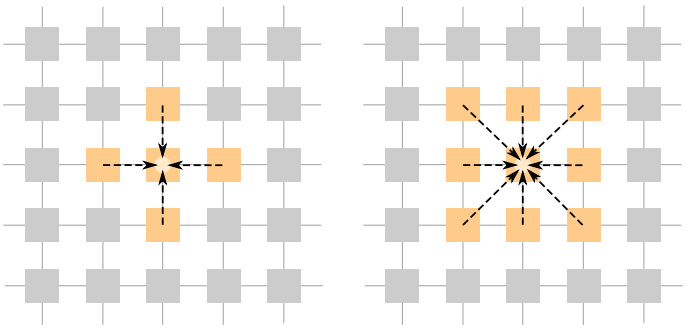
Výsledky ostrovního modelu se liší od klasického globálního modelu, v některých případech lépe prohledávají stavový prostor než klasické GA.

4.3.4 Jemná granularita

V případě jemného granularity (*Fine-Grained PGA*, také zvaný difúzní nebo mřížkový model) je celá populace rozdělena do velmi velkého počtu subpopulací (ideálně jeden jedinec na jeden proces) a zpracovávání tak probíhá masivně paralelně [26]. Protože grafické karty



Obrázek 4.2: Možná migrace jedinců u ostrovního modelu



Obrázek 4.3: Výběr partnera pro reprodukci ze 4 a 8mi okolí v mřížkovém modelu

mají obrovský potenciál v počtu vláken a jejich hardwarové synchronizace, jeví se tento přístup jako nejnadějnější.

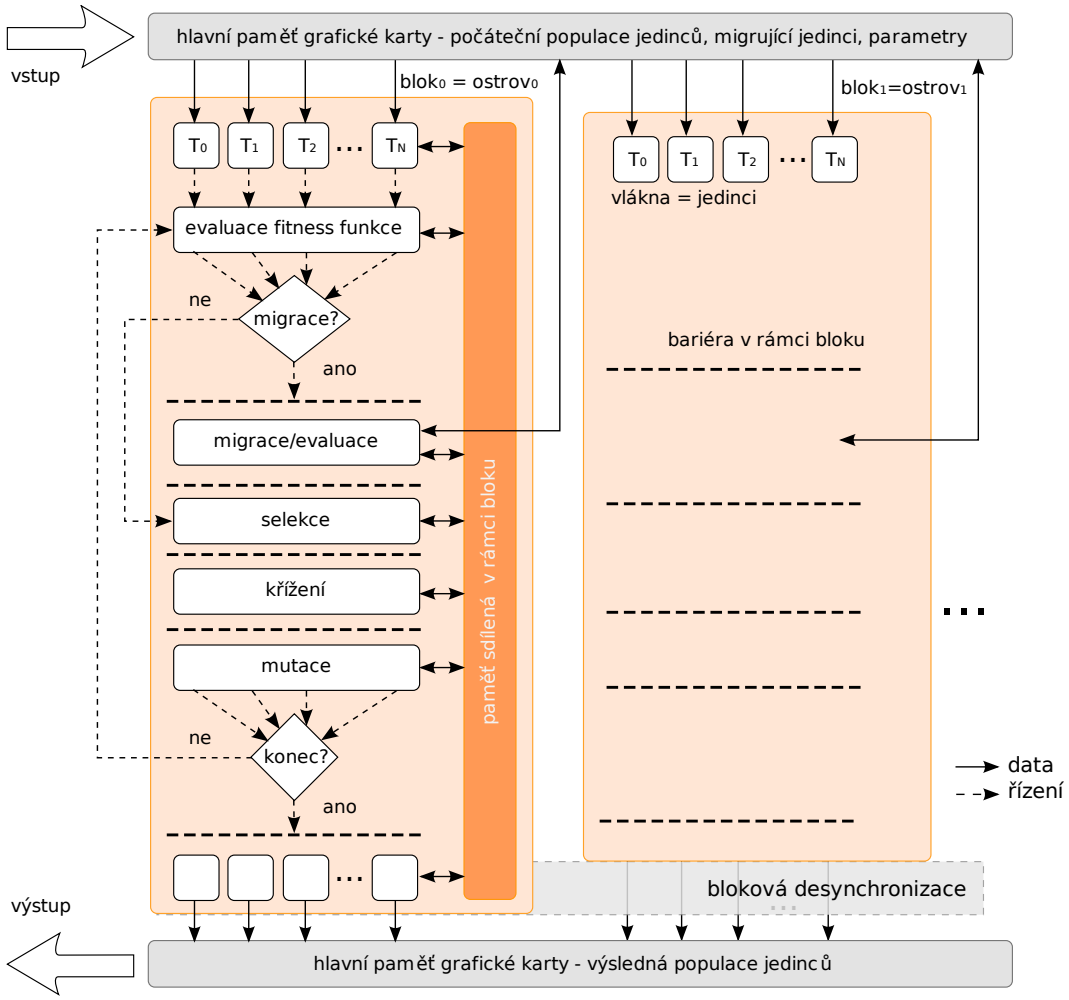
Stejně jako u ostrovních modelů se výsledky liší od klasického globálního modelu tím, že na rozdíl od nich berou v potaz genetické operátory blíže ke vztahu k jedinci, ne celé populaci. Difúzní model v některých případech dosahuje lepších výsledků [37] než klasické sekvenční GA.

4.3.5 Hybridní model

Hybridní přístup se snaží různou kombinací všech přístupů eliminovat nevýhody všech metod.

4.4 Návrh paralelizace GA na GPU

Vzhledem k povaze GPU je pro výraznou akceleraci Genetických algoritmů potřeba realizovat především masivní paralelismus. Ten je možné snáze dosáhnout s použitím jemné granularity, tedy přiřadit každému jedinci v populaci ideálně jedno vlákno programu. Kromě toho je také vhodné využít rychlé sdílené paměti v rámci multiprocesoru a minimalizovat přístupy do hlavní paměti, která má velké latence.



Obrázek 4.4: Schéma paralelizace GA na GPU

S ohledem na softwarový a hardwarový model CUDA (viz 3.4.1) byla navržena realizace GA na GPU, jak prezentuje obrázek 4.4.

Jak je patrné, celý běh GA je rozdělen do ostrovů, v rámci kterých jsou jedinci obsluhováni vlákny. Jednotlivé ostrovy jsou pak v CUDA modelu realizovány bloky, ve kterých je možné provádět vláknovou synchronizaci a využívat rychlou sdílenou paměť v rámci multiprocesoru na grafickém čipu. Pořadí vykonávání bloků je však nedefinované a není možné provádět jejich synchronizaci bez zásahu CPU (což by mělo drastický dopad na výkon). Proto je použit asynchronní model migrace, kdy ostrovy zapisují do hlavní paměti po několika generacích a ostatní ostrovy jej v nedefinovaném pořadí čtou. Pro výměnu po-

pulace je využita hlavní paměť sloužící pro přenos počáteční populace jedinců do grafické karty, což má tyto výhody:

- šetří se paměť grafické karty, protože pro migraci není třeba alokovat novou
- je zajištěna konzistence v případě čtení jedinců před zápisem jiného ostrova, protože v tomto případě je načtena část počáteční populace z definičního oboru optimalizované funkce a nikoliv nedefinovaná data

Migrace jsou dále podrobně popsány v kapitole 5.1.5, detailům selekce a křížení se věnuje kapitola 5.1.7.

V rámci bloku je dále v maximální možné míře využita rychlá sdílená paměť a každému jedinci je přiřazeno jedno softwarové vlákno, což vede k velkému paralelismu. Konzistentní přístup k populaci jedinců a dalším datům více vlákny je zajištěn použitím bariéry mezi jednotlivými kroky algoritmu. Pomalá hlavní paměť je využita pouze pro komunikaci s hostitelským systémem a v případě migrace (která obecně nenastává každou iteraci). Jde vidět, že použitý návrh má velký potenciál ohledně škálovatelnosti a dosažených zrychlení. Konkrétní naměřené hodnoty jsou předmětem kapitoly 6.

4.5 Získávání statistik

Genetické algoritmy, jakožto stochastická optimalizační metoda, pracují především s náhodou, a proto se pro vyhodnocování jejich funkce používají převážně statistické metody. V rámci práce je třeba vyhodnocovat kvalitu řešení na základě statistik výsledné populace. Pro tyto účely je vhodné použít tyto ukazatele u fitness hodnoty:

minimální a maximální hodnota – ukazuje nejhoršího, resp. nejlepšího jedince v populaci

průměrnou hodnota – je vhodným ukazatelem celkové kvality populace, je definována jako:

$$\overline{x} = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{4.1}$$

kde N je počet prvků a x_i jsou jednotlivé prvky

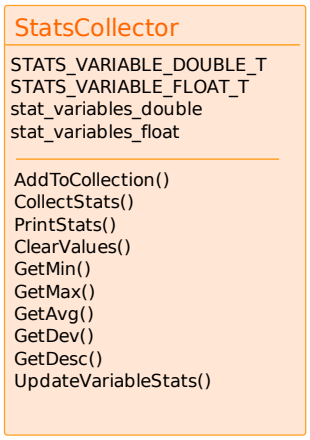
směrodatnou odchylka – slouží jako ukazatel odlišnosti jedinců, potažmo biodiverzity populace. Je definovaná jako odmocnina rozptylu náhodné veličiny [14]:

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - \overline{x})^2} = \sqrt{\left(\frac{1}{N} \sum_{i=1}^{N} x_i^2 \right) - \overline{x}^2} \tag{4.2}$$

kde N je počet prvků, x_i jsou jednotlivé prvky a $\overline{x}$ je průměrná hodnota

Vhodnou implementací je možné statistiky spočítat jednoprůchodově. Kromě vyhodnocování fitness hodnoty populace je také důležité měření času, potažmo rychlosti. Protože každá měřená veličina vyžaduje několik pomocných proměnných, pro větší množství parametrů se tak může stát implementace nepřehlednou. Z tohoto důvodu je vhodné navrhnout specializovanou část systému zajišťující snadný sběr a vyhodnocení statistik pro více veličin.

Navrženou třídu ilustruje obrázek 4.5. Implementace je blíže popsána v kapitole 5.3.



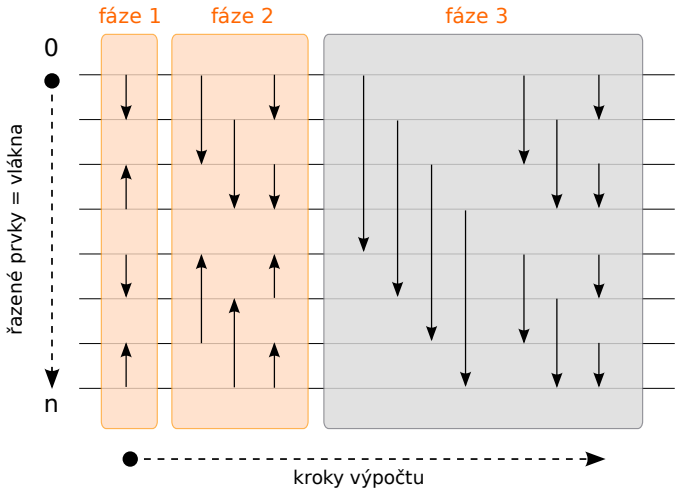
Obrázek 4.5: Třída StatsCollector

4.6 Řazení na GPU

Pro vyhodnocení kvality jedinců v populaci a případné migrace mezi ostrovy je třeba implementovat v GPU řazení. Efektivní využití je možné díky vysoké paralelizaci, optimálně by jednomu vláknu měl být přidělen jeden prvek řazených dat. Klasické sekvenční algoritmy používané v CPU jako QuickSort a HeapSort jsou však pro takovou implementaci nevhodné.

Možnosti řazení na GPU jsou detailně diskutovány v publikaci [39]. Základní řadící algoritmus Odd-Even sort je ve své podstatě paralelní implementací Bubble sortu. Jako takový je velmi jednoduchý na implementaci, ale má složitost $\Theta(n^2)$, což není optimální.

Další možností je vylepšení Odd-Even sortu na Odd-Even Merge sort, který využívá větší paralelizace a má proto rychlejší implementaci.



Obrázek 4.6: Schéma funkce Bitonic-Merge sort algoritmu

Nejvhodnější je však použití Bitonic-Merge sort algoritmu, jehož princip práce prezentuje obrázek 4.6. Jak je patrné, je prováděn paralelně pro všechny řazené elementy. Sestává

ze tří hlavních fází [39]: v první resp. druhé jsou sestaveny vzestupné, resp. sestupné bitonické subsekvence[1]. V poslední fázi jsou pak tyto sekvence spojeny dohromady.

Při použití stejného počtu vláken jako je řazených elementů je složitost algoritmu $\Theta(log^2(n))$ [25]. V některých GPU implementacích bylo dosaženo zrychlení až 20x proti QuickSort běhu na CPU [3]. Algoritmus je tak dostatečně rychlý a vhodný pro použití v GA implementaci.

4.7 Získávání náhodných čísel

Genetické algoritmy, jakožto stochastická optimalizační metoda, vyžadují zdroj náhodných čísel pro selekci, křížení i mutaci. Z existujících možností získávání náhodných čísel je pro potřeby použití v GA vhodná PRNG[2] (*Pseudo-Random Number Generator*) [71], tedy skupina algoritmů implementovaných v software s datovým výstupem splňujícím ze statistického hlediska náhodné rozložení. Zdroj náhodných čísel by měl mít tyto vlastnosti:

kvalita – především u stochastických optimalizačních metod je třeba, aby měl výstup statisticky kvalitní náhodné rozložení a tedy algoritmus jím řízený byl schopný prohledat co nejlépe stavový prostor. U paralelních implementací je dále třeba, aby po sobě jdoucí čísla byla co nejméně vzájemně korelovaná (tedy například nebyla násobkem jiného), potažmo paralelně běžící vlákna byla na sobě nezávislá. V neposlední řadě je také vhodné, aby měl generátor co nejdelší periodu (poskytoval stejný výstup po co nejdelší době běhu).

rychlost – generátor náhodných čísel je v GA využíván často, proto by měl být velmi rychlý, aby svým během nebrzdil celou implementaci. Je tedy vhodné aby stavěl na bitových operátorech a jiných operacích zpracovaných obvykle v jednom taktu časovače hardware.

nízká náročnost na paměť – každý přístup do globální paměti GPU je časově náročný, proto je vhodné pro lokální data jednotlivých generátorů využívat sdílenou paměť. Ta je ale kapacitou silně omezená, proto by použitý PRNG neměl mít příliš velké nároky na paměť.

Pro selekci a křížení jsou v GA třeba náhodná čísla s uniformním rozložením (všechny hodnoty intervalu jsou zastoupeny se stejnou pravděpodobností). Pro mutaci reálného chromozomu se také využívá náhodné číslo z Normálního (Gaussovského) rozložení [64]. Většina PRNG generuje čísla s Uniformním rozložením, které je v případě potřeby možné převést transformací do Normálního rozložení. Pro GPU je vhodná Box-Mullerova transformace [34, 67], protože nevyužívá podmínek a komplikovaných přístupů do paměti:

$$r_0 = sin(2\pi u_0)\sqrt{-2log(u_1)} \tag{4.3}$$

$$r_1 = cos(2\pi u_0)\sqrt{-2log(u_1)} \tag{4.4}$$

kde

[1] Bitonická je taková sekvence, která obsahuje nejvýše dvě změny monotónnosti 0 na 1 nebo naopak. Obecněji Bitonická sekvence čísel je taková, která má nejvýše jedno lokální maximum a jedno lokální minimum

[2] Vedle PRNG (Pseudorandom Number Generators) existují také TRNG (True Random Number Generator) využívajících jako zdroj náhodnosti skutečný fyzický jev a QRNG (Quasirandom Number Generator) snažící se vyplnit bez shlukování n-dimenzionální prostor body

- u_0 a u_1 jsou dvě náhodná čísla Uniformního rozložení z interval $\langle 0; 1 \rangle$
- r_0 a r_1 jsou dvě výsledná čísla z Normálního rozložení

Mezi uniformní Pseudonáhodné generátory patří:

Linear Congruent generator [69] (LCG) je klasický generátor náhodných čísel použitý v funkci `rand()`. Je založen na přechodové funkci ve tvaru

$$x_{n+1} = (a \cdot x_n + c) \bmod m \tag{4.5}$$

Pro vhodně zvolené konstanty a,c a m má statistické vlastnosti uniformního rozložení. Jeho maximální perioda je však 2^m, což je pro masivně paralelní implementace o několika tisících vláknech málo. Jeho výhodou je velká rychlost a jednoduchá implementace s minimálními nároky na paměť.

Mersenne-Twister [70] je považován za jeden z nejkvalitnějších generátorů náhodných čísel. Jeho název je odvozen z Mersennova prvočísla, což je prvočíslo ve tvaru 2^n-1. Má velmi velkou periodu $2^{19937} - 1$ a vykazuje nekorelovanost po sobě jdoucích hodnot až do 602 rozměrů. Využívá však relativně velké množství paměti pro uchovávání předchozích hodnot, které musí být sériově aktualizovány při generování každého čísla, a proto je pro GPU implementaci příliš pomalý.

Combined Tausworthe generator na rozíl od Mersenne-Twister algoritmu vyžaduje výrazně méně paměti (pouze 2-4 slova), ale zároveň má mnohem delší periodu než LCG. Klasický Tausworthe generátor je založen na výpočtu v rámci jednotlivých bitů ve tvaru[24] :

$$x_{n+1} = (A_1 x_n + A_2 x_{n-1} + \cdots + A_k x_{n-k+1}) \bmod 2 \tag{4.6}$$

kde $x_i, A_i \in \{0, 1\}$ pro všechna i

Combined Tausworthe s pomocí funkcí XOR kombinuje více binárních řetězců dohromady, čímž vzniká vyšší kvalita výstupu a přitom se zachovává jednoduchost srovnatelná s LCG.

4.8 Parametrizace

Klasické aplikace pro GA předpokládají parametrizaci algoritmu (velikost populace, míra mutace, ...) například s použitím příkazové řádky, což má výhodu v tom, že program nemusí být opětovně překládán pro různá nastavení.

GPU implementace s použitím CUDA však vyžaduje odlišný přístup, protože:

- velikost sdílené paměti by měla být známa v době překladu, což umožňuje překladači optimalizovat přidělení registrů jednotlivým vláknům a lépe tak využít potenciál GPU. Z tohoto důvodu je nevhodné nastavovat dynamicky (v době běhu) velikost populace a počet jedinců na ostrově.

- nejsou podporovány ukazatele na funkce, z tohoto důvodu by různé optimalizační funkce musely být vybírány podmínkou při běhu programu, což je krajně nevhodné vzhledem k rychlosti implementace a požadavku na SIMD typ zpracování dat.
- GPU nejsou vhodné pro zpracování cyklů, proto překladač optimalizuje kód převodem cyklů na lineární kód (tzv. *static unroll*) [11]. Protože jedna generace zpracování jedinců je typicky záležitost programového cyklu, tato optimalizace není možná, pokud v době překladu není znám počet generací v GA.
- Velké množství volání jednoduchých funkcí může mít také relativně významný dopad na výkon (viz 4.2), některé se dají optimalizovat statickým překladem konstanty.

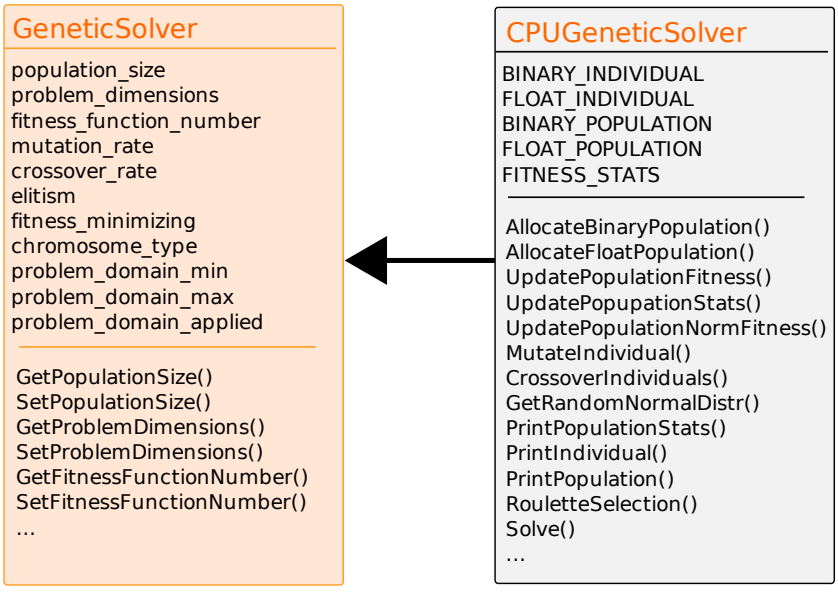
Z těchto důvodů je vhodnější použít pro parametrizaci direktivy preprocesoru překladače a pro každé nastavení provést překlad programu.

4.9 Návrh CPU verze GA

Cílem práce je co nejlépe prozkoumat možnosti běhu GA na GPU, proto je implementována také jednoduchá CPU verze sloužící jako reference z hlediska rychlosti a kvality výsledků.

Původním záměrem bylo vytvořit třídu definující společné parametry pro CPU a GPU a ty následně dědičností mapovat na konkrétní hardware. V průběhu testování CUDA však vyšlo najevo, že rychlost běhu na GPU těží hlavně z optimalizace překladačem, a proto je nevhodné použít dynamickou parametrizaci (viz kapitola 4.8). Implementace jsou tak ve finále oddělené a pro porovnání jsou použity statistiky jejich výstupů.

Jádro CPU části tvoří třída `GeneticSolver` definující základní parametry běhu GA. Tu dědí `CPUGeneticSolver`, která implementuje konkrétní funkci na CPU jak ilustruje obrázek 4.7.



Obrázek 4.7: UML návrh systému

Detaily CPU implementace jsou blíže prezentovány v kapitole 5.2.

4.10 Testování

Pro zhodnocení výsledků jsou použity umělé spojité optimalizační funkce.

Spojité funkce jsou definovány obecně na N rozměrech a jsou z praktického hlediska omezeny intervaly definičního oboru, které platí pro každý rozměr. Práce se zabývá následujícími: Griewankova funkce, Michalewiczova funkce a Rosenbrockova funkce, definice je uvedena níže.

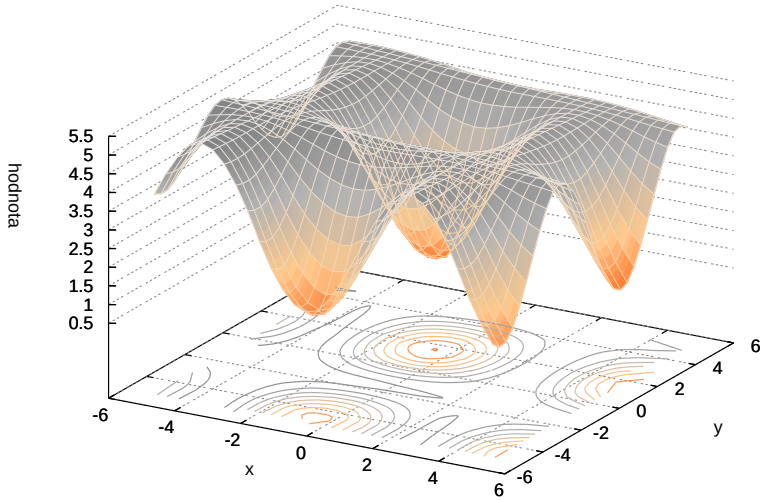
$$f_{Griewank}(x_0 \dots x_n) = \frac{1}{4000}\sum_{i=0}^{n-1}(x_i - 100)^2 - \prod_{i=0}^{n-1} cos\Big(\frac{x_i - 100}{\sqrt{i+1}}\Big) + 1 \quad \text{interval}[-5, 5]^n \tag{4.7}$$

$$f_{Michalewicz}(x_0 \dots x_n) = -\sum_{i=0}^{n-1} sin(x_i) sin^{20}\Big(\frac{(i+1)x_i^2}{\pi}\Big) \quad \text{interval}[0, \pi]^n \tag{4.8}$$

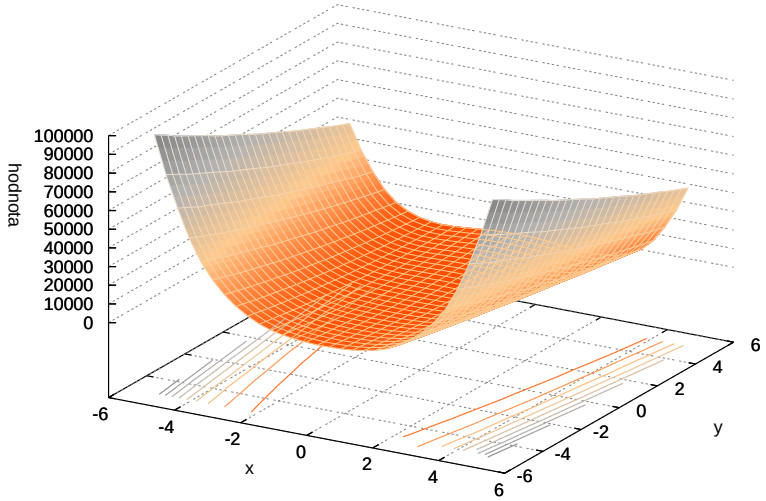
$$f_{Rosenbrock}(x_0 \dots x_n) = \sum_{i=0}^{n-2} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2 \quad \text{interval}[-5.12, 5.12]^n \tag{4.9}$$

- kde n je rozměr problému (počet genů v chromozomu)
- i je index genu v chromozomu
- výsledek funkce je hodnota fitness celého chromozomu

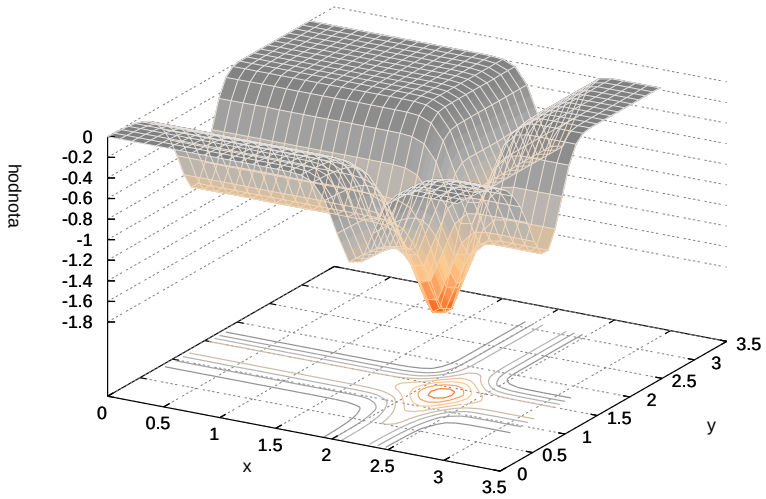
Dvourozměrné varianty výše uvedených funkcí prezentují obrázky 4.8(a), 4.8(b) a 4.8(c).



(a) Griewankova funkce



(b) Rosenbrockova funkce



(c) Michalewiczova funkce

Obrázek 4.8: Spojité testovací funkce vizualizované pro dvě proměnné

Kapitola 5

Implementace a testování

Tato kapitola popisuje implementaci navrženého systému. Důraz je kladen především na popis GPU části, detailně jsou diskutovány dílčí podproblémy jako jsou migrace, selekce a křížení. V závěru je uveden systém sběru a vyhodnocení statistik a stručný přehled referenční CPU verze. Kapitola také obsahuje odkazy na naměřené výsledky.

5.1 GPU implementace

S ohledem na navrženou koncepci a problémy s parametrizací diskutované v kapitolách 4.8 a 4.9 je GPU a CPU verze implementace oddělená. CPU verze, blíže popsaná v kapitole 5.2, je běžnou konzolovou aplikací využívající standardní systémové knihovny, není proto problém ji překládat na více platformách, spouštět bez CUDA hardware a porovnávat tak snadno rychlost různých procesorů proti GPU.

Naproti tomu GPU implementace je relativně úzce specializovanou záležitostí, vyžaduje grafickou kartu řady nVidia GeForce, Quatro nebo Tesla, nainstalovaný CUDA toolkit, nové ovladače a CUDA SDK (detaily viz projektová dokumentace na přiloženém datovém nosiči).

5.1.1 Struktura

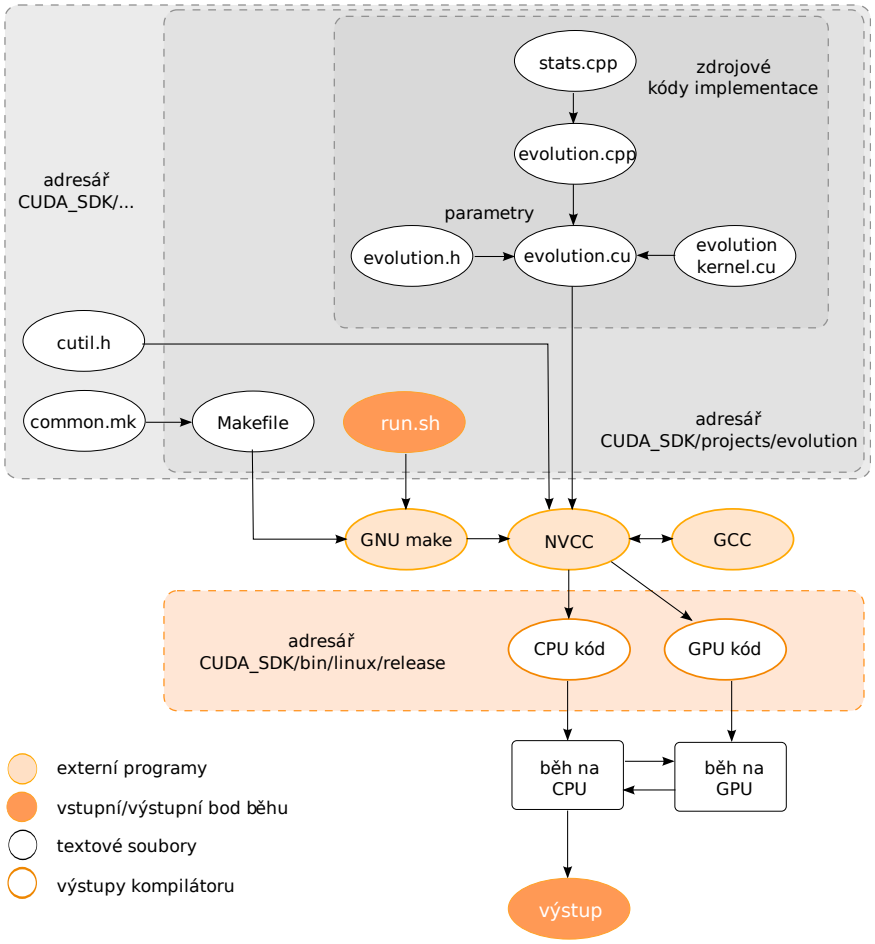
Při návrhu rozdělení funkcionality GA na GPU do zdrojových kódů sloužila jako inspirace řada ukázkových programů z CUDA SDK. Ty rozdělují kód na CPU, GPU a společnou část podle přípony souboru – `.cpp` jsou určeny pro CPU, `.cu` pak GPU nebo kombinaci. Výsledná implementace má tuto strukturu:

evolution_kernel.cu obsahuje jádro GPU implementace (kernel) a doprovodné funkce pro vyhodnocování fitness a generování náhodných čísel.

evolution.cpp je zdrojový kód CPU části, která výslednou populaci po běhu na GPU statisticky vyhodnocuje z hlediska kvality.

stats.cpp obsahuje třídu `StatsCollector` (viz 5.3) použitou ve `evolution.cpp` pro vyhodnocení výsledné GPU populace.

evolution.h definuje za pomocí direktiv preprocesoru překladače parametry běhu programu pro dávku `run.sh` a `runemu.sh`. Kromě toho na začátku, resp. na konci načítá soubory `evolution-prolog.h` resp. `evolution-epilog.h` sloužící k definici pomocných direktiv a ověření validity nastavených parametrů.



Obrázek 5.1: Struktura GPU implementace pro dávku run.sh

pomocné skripty slouží pro usnadnění překladu a vyhodnocení statistik, jsou prezentovány níže v této kapitole.

Protože projekty mezi podporovanými platformami Linux a Windows nejsou snadno přenositelné[1], implementace se úzce zaměřuje na použitou platformu Linux a využívá řadu systémových utilit a skriptovacího jazyku BASH[2] pro analýzu a zpracování výstupních dat.

Pro usnadnění překladu a spouštění GPU implementace slouží skripty `run.sh`, `runemu.sh` a `runbench.sh`:

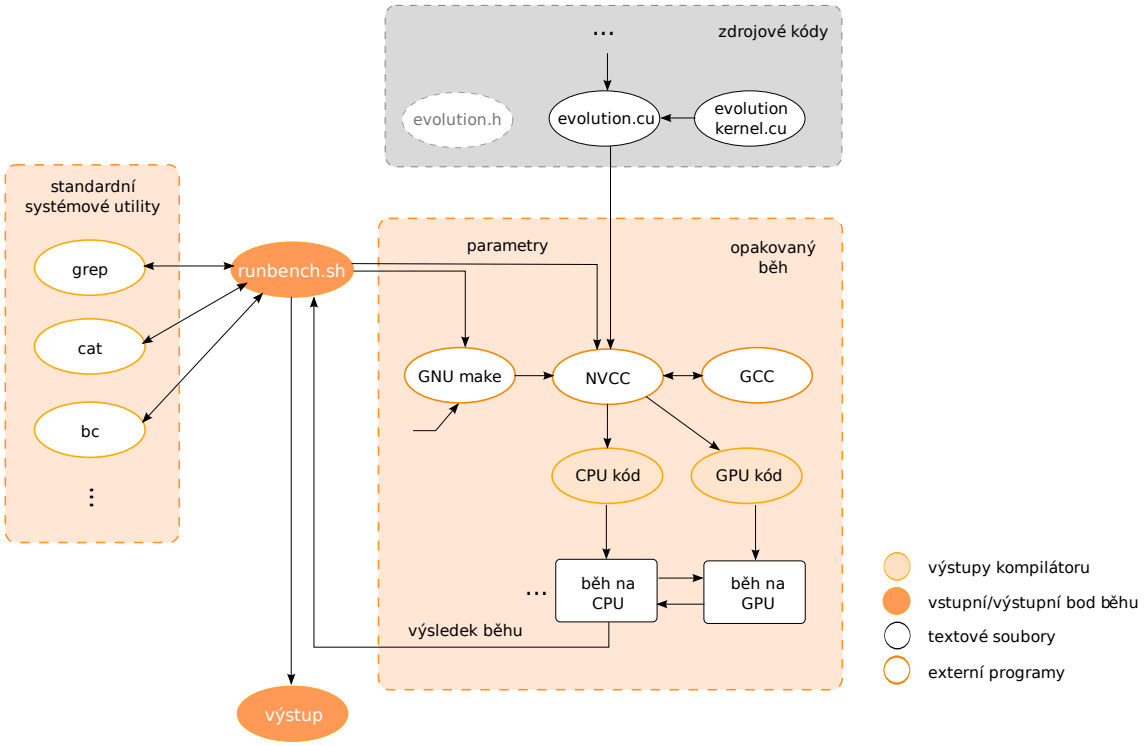
run.sh je určen pro jednorázový překlad a běh GA na GPU s použitím parametrů uvedených v hlavičkovém souboru `evolution.h` (viz kap. 5.1).

Jak ilustruje obrázek 5.1, skript volá utilitu `GNU make`, která s použitím souboru `Makefile` z implementace, `common.mk` z CUDA SDK, driveru kompilátoru `NVCC` a `GCC`[3]

[1]CUDA pro Windows vyžaduje Microsoft Visual Studio a projekt založený v něm, proto není možné použít pro multiplatformní překlad například program make

[2]Bourne-Again SHell je jeden z příkazových řádků na operačních systémech typu UNIX, kromě běžné práce se systémem podporuje také skriptování v syntaxi podobné programovacímu jazyku C

[3]GCC (původně GNU C Compiler, nyní GNU Compiler Collection) je sada překladačů různých programovacích jazyků používaná převážně v operačních systémech typu UNIX



Obrázek 5.2: Struktura implementace pro dávku runbench.sh

(viz 5.1.2) provede překlad. Výsledkem je kód pro CPU a GPU, který je skriptem při úspěšném překladu spuštěn. Inicializace dat a vyhodnocení proběhne na straně CPU, výsledky jsou vypsány na standardní výstup, mohou tedy být v případě potřeby snadno přesměrovány do souboru určenému k dalšímu zpracování (čehož je využíváno pro generování grafů, viz kap. 5.4).

runemu.sh je skript určený pro překlad a následně emulovaný běh. Od dávky `run.sh` se struktura volání programů liší pouze tím, že jak CPU, tak GPU kód proběhnou na CPU. Je vhodný především pro ladění, v souboru `evolution.h` má oddělené parametry, takže je možné mít separátní nastavení pro emulovaný a reálný běh na GPU.

Kromě nich obsahuje také direktivy pro ladící výstupy (viz 5.1.3), protože emulovaný běh, na rozdíl od reálného, podporuje volání CPU funkcí.

runbench.sh je skript určený pro statistické vyhodnocování běhů pro rozsah různých parametrů, používá k tomu standardní Unixové utility jako `cat`, `grep`[4], `bc`[5]. Jak bylo nastíněno v kapitole 4.8, je kvůli optimalizace kódu na GPU pro odlišné parametry prováděn nový překlad. Běh se kromě vícenásobného volání překladače a spouštění programu od `run.sh` liší také tím, že data z výstupu běhu zpracovává s použitím skriptu, jak ilustruje obrázek 5.2. Parametry ze souboru `evolution.h` jsou v tomto případě nahrazeny direktivou překladače parametry ze skriptu (viz 5.1.2).

[4]grep je utilita sloužící k filtrování řádků ze standardního vstupu/souboru na základě regulárního výrazu nebo obsažených slov. V práci je použit pro výběr řádků obsahujících parametry běhu a další informace.

[5]bc je kalkulátor bez omezení přesnosti pracující se standardním vstupem a výstupem, což ho předurčuje ke snadnému použití ve skriptech, kde je potřeba spočítat průměr a další statistiky.

5.1.2 Překlad

Nvidia dodává ke frameworku CUDA ovladač překladače (*compiler driver*) NVCC sloužící k předzpracování a překladu GPU kódu. Jako takový využívá i standardního systémového překladače zdrojových kódů v jazyce C, v případě Microsoft Windows `cl`, v případě Linuxu `GCC` [13]. V CUDA frameworku pro Linux je k jeho snadnému použití připraven soubor `common.mk` určený pro `GNU make`. V `Makefile` souboru vytvářeném v rámci implementace pak stačí nadefinovat zdrojové soubory s použitím proměnných `CUFILES` (v případě `.cu` souborů), `CCFILES` (v případě `.cpp` souborů), popř. další parametry překladu a provést jednoduše překlad s použitím příkazu `make`.

Kompilace s použitím `NVCC` používá, kromě standardních parametrů jako jsou `-D` v případě definic direktiv preprocesoru, také tyto specifické přepínače:

-use_fast_math parametr (v souboru `common.mk` se zapíná nastavením proměnné `fastmath =1`) při překladu způsobí použití rychlejších, ale méně přesných matematických funkcí. Testování ukazuje, že to může mít za následek až ztrojnásobení výkonu (viz 6.3) za cenu relativně malého zmenšení přesnosti výsledků.

-deviceemu (v souboru `common.mk` se zapíná nastavením proměnné `emu=1`) způsobí překlad GPU kódu emulovaně na CPU. V tomto režimu je možné volat z GPU kódu hostitelské funkce, což může být velmi vhodné pro ladění (viz 5.1.3).

V implementaci je pro urychlení běhu cyklů v GPU často používána direktiva `#pragma unroll` sloužící pro převod smyček do sekvenčního kódu.

5.1.3 Ladění

Výpočet na grafické kartě s použitím CUDA má vzhledem k procesoru spíše charakteristiku přídavného akcelerátoru – data jsou na ni nejdříve poslána, poté je spuštěn kód a následně jsou zkopírovány výsledky do paměti počítače, kde mohou být dále zpracovávány procesorem. Vzhledem k tomu, že je zbytečné mrhání výkonem zobrazovat prostřednictvím 3D akcelerovaného GUI průběh výpočtu, je interakce s uživatelem v průběhu výpočtu silně omezená, v případě CUDA Compute Capability 1.0 prakticky nulová[6]. Proto je ladění GPU kódu mnohem problematičtější. `NVCC` ovladač kompilátoru naštěstí podporuje (sekvenční) softwarovou emulaci hardware grafické karty, ve kterém je možné volat z `device` částí kódu i funkce pro CPU a je tak možné vypisovat ladící informace. Pro tyto účely jsou v implementaci následující parametry preprocesoru:

- `DEBUG_INDEXING` vypisuje informace o indexech vstupní, výstupní a průběžné populace jedinců
- `DEBUG_POPULATION` průběžně vypisuje celou populaci jedinců včetně fitness ohodnocení
- `DEBUG_FITNESS_STATS` způsobí výpočet a výpis statistik fitness funkce po každé generaci ještě na GPU

[6] Novější hardware s Compute Capability 1.3 podporuje asynchronní vyvolávání událostí s použitím ovladače grafické karty, lze tak například prokládat výpočet s kopírováním dat na grafickou kartu nebo efektivně spouštět výpočet zároveň na CPU a GPU. V rámci této práce však nebyl testován.

- `DEBUG_MIGRATION_SORT` poskytuje informace relevantní pro řadící algoritmus v migraci
- `DEBUG_MIGRATION_INDEXING` vypisuje ladící informace pro indexování globální i lokální populace v průběhu migrací
- `DEBUG_MUTATION` vypisuje informace o genech před a po mutaci a mutovaných indexech
- `DEBUG_CROSSOVER` slouží k ladění křížení, pro tyto účely vypisuje informace o vítězích selekce sebe a sousedního vlákna (viz 5.1.7)

V případě, že je některý výše uvedený parametr v souboru `evolution.h` definován s použitím direktivy `#define`, překladač na příslušném místě zahrne kód způsobující danou činnost. Výpisy tak nejsou testovány podmínkou za běhu programu a nezpomalují proto implementaci v reálném režimu. Navíc jsou tyto parametry zahrnuty podmíněným překladem `#ifdef` pouze v případě, že překlad probíhá bez emulace, potažmo je možné volat funkce hostitele (CPU) z device kódu (GPU).

5.1.4 Generování náhodných čísel

Jak nastiňuje kapitola 4.7, pro GPU implementaci GA je potřeba generátor pseudonáhodných čísel, který umožňuje dobrou paralelní implementaci (tj. má dlouhou periodu a produkuje nekorelované čísla), je dostatečně rychlý a pro běh vyžaduje minimum paměti.

Tato problematika je s ohledem na GPU podrobně diskutována v knize [34] publikované přímo výrobcem grafických čipů nVidia. Zmiňuje upravený generátor náhodných čísel, který sestává z kombinace Linear Congruential, čtyř Tausworthe generátorů a vhodně zvolených konstant jako parametrů.

Výsledná implementace (viz příloha B) je velmi dobře paralelizovatelná, má minimální požadavky na paměť (pouze 4 proměnné pro každý generátor) a periodu asi 2^{113}.

Testování v praxi ukazuje, že implementace je i dostatečně rychlá, při vysoké paralelizaci v případě Uniformního rozložení dosahuje zrychlení proti sekvenční CPU verzi až 90x, u Normálního rozložení s použitím přepínače kompilátoru `-use_fast_math` dokonce 700x (viz 6.3). Vykazuje také téměř nulovou chybu při výpočtu v porovnání s CPU implementací (viz 6.3) a statisticky kvalitní hodnoty.

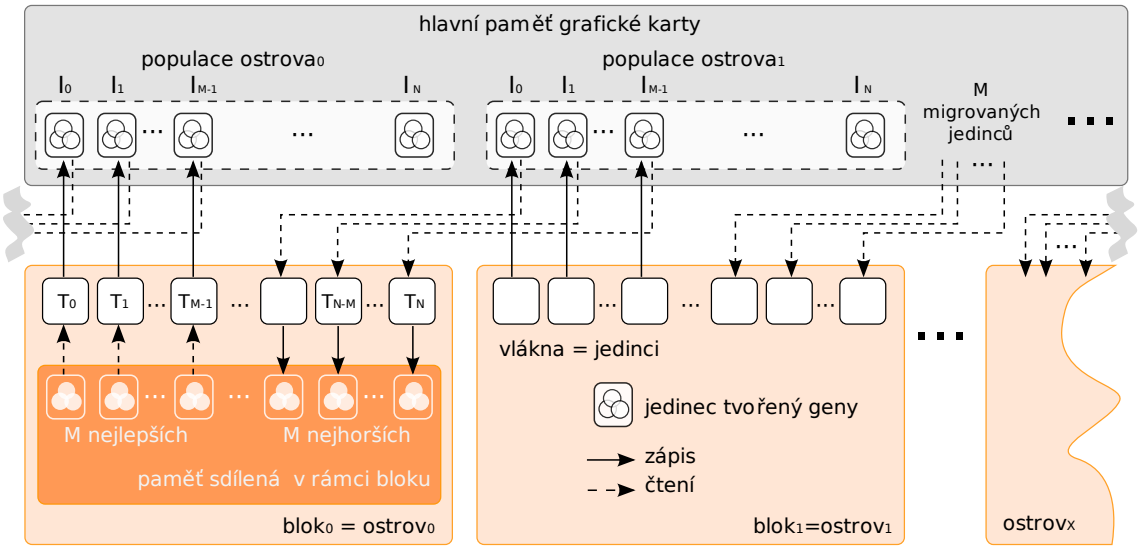
Protože splňuje všechny podmínky vhodného pseudonáhodného generátoru, je použit v rámci této práce. Pro náhodnou inicializaci vláken je použit klasický LCG na straně CPU s předáním přes hlavní paměť před zahájením výpočtu.

5.1.5 Migrace mezi ostrovy

Jak již bylo nastíněno v kapitole 4.4, každý ostrov jedinců je simulován v samostatném bloku a využívá v maximální míře sdílenou paměť v rámci multiprocesoru. Pro přenos dat mezi ostrovy je však potřeba přístup do hlavní paměti grafické karty.

Při migraci je třeba vyměnit se sousedními ostrovy obecně $M \in \mathbb{N}$ jedinců (kde N je velikost populace na ostrově). Migrace tedy sestává ze zápisu M nejlepších jedinců do hlavní paměti a čtení M jedinců z hlavní paměti jiného ostrova.

Protože CUDA nepodporuje blokovou synchronizaci a pořadí vykonávání bloků je nedefinované, nastává migrace v nedefinovaném pořadí. Z toho důvodu je pro výměnu jedinců



Obrázek 5.3: Schéma migrace jedinců mezi ostrovy

využita hlavní paměť grafické karty sloužící pro přenos počáteční populace z CPU do GPU (důvody jsou zmíněny v kapitole 4.4).

Samotná migrace pak probíhá následovně:

řazení – všichni jedinci v rámci bloku jsou nejdříve paralelně seřazeni Bitonic-Merge sort (viz 4.6) algoritmem. Protože operace prohození je obecně tím dražší, čím více genů chromozom jedince obsahuje, je pro ni použito pomocné pole sdílené paměti `temp_indexes`. V tomto poli je po seřazení na každém indexu odkaz na jedince příslušného pořadí, tj. nultý index pole `temp_indexes` obsahuje odkaz na nejlepšího jedince, první na druhého nejlepšího, poslední na nejhoršího a pod.

Pole `temp_indexes` je použito také při turnajovém křížení, aby se zbytečně neplýtvalo sdílenou pamětí.

zápis – po řazení mají všechna vlákna prostřednictvím pole `temp_indexes` přístup k jedincům daného kvalitativního pořadí v populaci. Indexování probíhá podle ilustrace na obrázku 5.3 – prvních M vláken v bloku zapíše s pomocí `temp_indexes` M nejlepších chromozomů z ostrova do své populace, v případě první migrace tak přepisují počáteční populaci. Z obrázku je také patrný použitý cyklický migrační model, kdy sousedem posledního ostrova je první.

čtení a evaluace – posledních M vláken v rámci bloku načítá ze sousedního ostrova prvních M (nejlepších) jedinců, kterými ve sdílené paměti přepisují M nejhorších v populaci svého ostrova. Tyto nové jedince hned v zápětí vyhodnotí z hlediska kvality a zapíší jejich novou fitness hodnotu to sdílené proměnné `fitness_values`. Fitness hodnota není kopírována do hlavní paměti na základě doporučení Cuda Programming Guide [11], ve kterém je opětovně zdůrazňována potřeba minimalizace přístupů do hlavní paměti i za cenu zvýšené intenzity výpočtu.

Čtení i zápis může probíhat paralelně v případě, že $2 \cdot M \leq N$, kde M je počet migrovaných jedinců a N je počet jedinců na ostrov. Toho je docíleno podmíněným překladem

`_syncthreads()` funkce mezi bloky čtení a zápisu. Maximálního paralelismu je dosaženo, pokud $2 \cdot M = N$, kdy polovina vláken při migraci čte z hlavní paměti nové jedince a druhá polovina je zapisuje.

Protože Bitonic-merge sort algoritmus pracuje nativně s počtem prvků o velikostech mocniny dvou (v případě, že by tomu tak nebylo, by se zbytečně plýtvalo výkonem), obsahuje implementace omezení počtu jedinců pro jeden ostrov na tyto hodnoty.

Soubor `evolution.h` obsahuje v souvislosti s migrací tyto parametry:

- `MIGRATION` určuje, zda se vůbec budou provádět migrace. V případě zakomentování tohoto parametru nejsou do výsledného kódu zahrnuty části spojené s migrací, není prováděno ani řazení jedinců podle kvality a dochází tak k výraznému urychlení běhu za cenu snížené konvergence jedinců ke globálnímu optimu (viz 6.5).
- `MIGRATION_INTERVAL_GENERATIONS` definuje, po kolika generacích se provádí migrace. Vzhledem k nedefinovanému pořadí provádění bloků se jedná pouze o přibližný parametr. Měření však ukazují, že výsledky se liší a migrace funguje správně (viz 6.5).
- `MIGRATION_INDIVIDUALS` určuje počet migrovaných jedinců, v popisu výše je shodný s parametrem M.

5.1.6 Elitismus

Elitismu lze v GPU jednoduše dosáhnout nastavením jednoho ostrova s migrací – ostrov bude periodicky zapisovat nejlepší jedince do hlavní paměti, odkud s nimi přepíše své nejhorší jedince. Řazení se však velmi negativně projevuje na výkonnosti aplikace (viz 6.5), proto je pro GPU vhodnější operovat nad velkou populací (která svůj potenciálně dobrý genofond neztrácí tak snadno a zvyšuje paralelismus), než uměle udržovat nejlepšího jedince v malé populaci.

5.1.7 Selekce a křížení

Pro implementaci selekce byl zvolen turnajový výběr (viz 2.6.3), protože profilování ukazuje jeho vysoký výkon (viz 4.2), má dobrý selekční tlak, nepotřebuje řazení z hlediska kvality a potenciálně neobsahuje tolik datových závislostí jako ruletový výběr. Je tedy dobře paralelizovatelný a vhodný pro akceleraci na GPU. Z podobných důvodů bylo pro implementaci vybráno aritmetické křížení (viz 2.5.2).

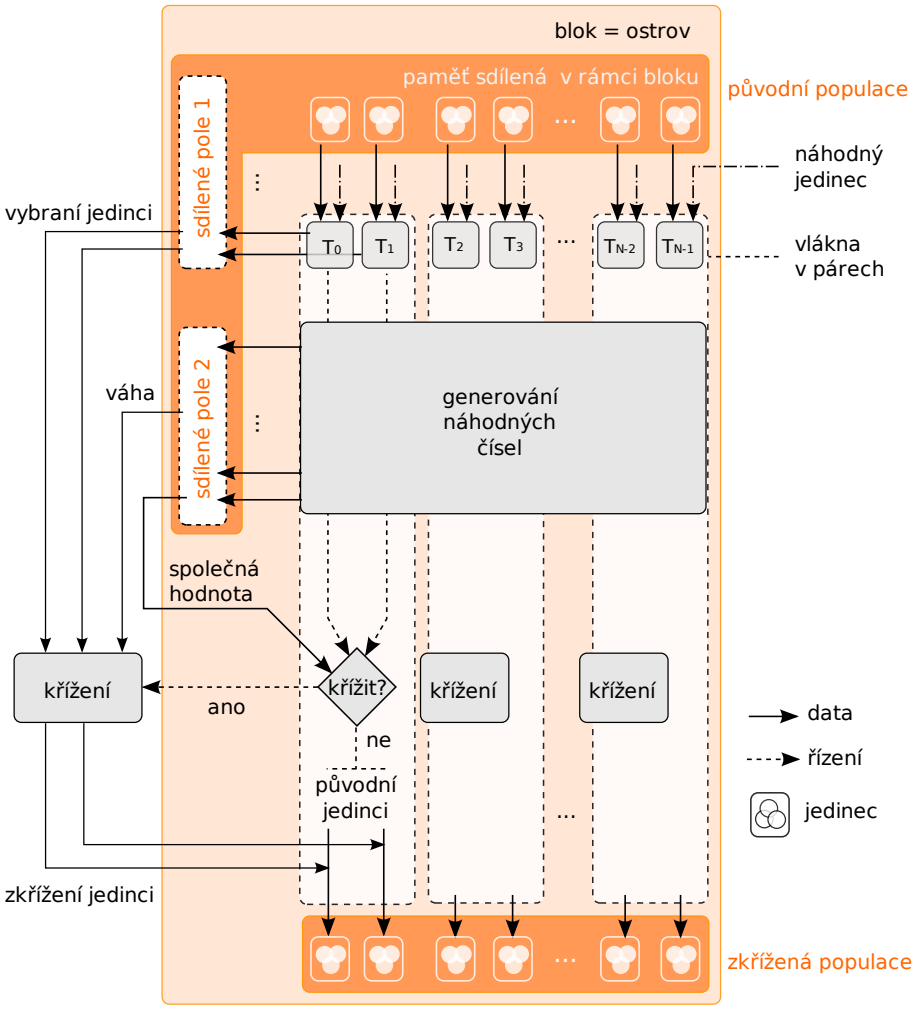
Selekce i křížení jsou kvůli rychlosti prováděny ve sdílené paměti v rámci jednoho bloku na GPU a jsou silně provázány, jak prezentuje obrázek 5.4.

Jak je patrné, všechna vlákna paralelně nejdříve vyberou náhodného jedince v rámci sdílené populace a porovnají ho s jedincem indexu stejného jako je index vlákna. Turnajem prochází lepší z nich, jehož index je (opět paralelně) zapsán do sdíleného pole 1.

Následuje paralelní generování náhodných čísel uniformního rozložení, kterým je naplněno sdílené pole 2 použité v pozdějších fázích běhu.

Následně vlákna rozdělená do párů vstupují do podmínky, zda bude probíhat křížení. V této fázi je použita pro obě vlákna společná hodnota načtená z první poloviny sdíleného pole 2.

Pokud dojde ke křížení, je druhá polovina sdíleného pole 2 použita jako náhodná váha, oba zkřížení jedinci jsou v tomto případě paralelně vygenerováni na základě turnajových vítězů zapsaných ve sdíleném poli 1. Tím je dosaženo vysokého paralelismu při provádění.



Obrázek 5.4: Schéma funkce selekce a křížení na GPU

Tato implementace provádí zbytečný turnajový výběr v případě, že nakonec nedojde ke křížení na základě podmínky. Mohlo by se zdát, že to má negativní dopady na výkon, nicméně jak ukazuje praxe, GPU jsou silně optimalizovaná na zpracování typu SIMD, a proto je pro ně vhodné provádět minimum kódu různou cestou řízení. Z tohoto důvodu je použitá implementace v závislosti na parametrech o 0.1-2% rychlejší, než když je podmíněně prováděna i selekce.

Soubor `evolution.h` obsahuje v souvislosti s křížením tyto parametry:

- `CROSSOVER_RATE` vyjadřuje míru křížení, jak je popsaná v kapitole 2.5. Validní hodnoty jsou 0,0 pro žádné křížení až 1,0 pro 100% zkřížených jedinců v populaci. V běžných implementacích se hodnota tohoto parametru pohybuje kolem 0,7-0,9.
- `SEED_RANDOM_INIT` při zakomentování zajišťuje, že generátor náhodných čísel nebude inicializován pseudonáhodnou hodnotou (časem). Tím lze dosáhnout toho, že každý běh se stejnými parametry bude shodný, čehož bylo využito při ladění a examinaci konvergence populace na GPU.

5.1.8 Mutace a evaluace fitness funkce

Implementace GA na GPU je realizována podle návrhu popsaném v kapitole 4.4 s jemnou granularitou, tedy každému jedinci v populaci je přiřazeno jedno softwarové vlákno v CUDA modelu. Evaluace fitness funkce tak probíhá na začátku iterace algoritmu paralelně pro všechny jedince v populaci (s výjimkou migrovaných jedinců). Fitness hodnoty jedinců jsou zapisovány do rychlé sdílené paměti, kde k nim při křížení a migraci mají přístup i ostatní vlákna. Jak ukazuje testování, díky absenci datové závislosti a vysoké úrovni paralelizace dochází až k několika-set násobnému urychlení proti jednomu vláknu procesoru (viz 6.4).

Mutace probíhá typicky pouze pro malé procento jedinců populace v závěru iterace algoritmu. V implementaci nejdříve všechna vlákna paralelně vygenerují náhodné číslo z Uniformního rozložení, následně otestují na základě pravděpodobnosti mutace, zda budou mutovat. V případě, že ano, proběhne generování náhodného čísla z Gaussovského rozložení a následný zápis do výsledné populace jedinců, v opačném případě pouze zápis jedinců.

Jak ukazuje testování, pravděpodobnost mutace ovlivňuje rychlost implementace v případě, že často nedochází k žádným mutacím v celé populaci (viz kap. 6.6), v opačném případě se totiž kvůli synchronizace zastaví celý blok vláken.

Soubor `evolution.h` obsahuje v souvislosti s evaluací fitness funkce a mutací tyto parametry:

- `FITNESS_FUNCTION` určuje testovací optimalizovanou funkci, může nabývat hodnot `FITNESS_FUNCTION_GRIEWANK`, `FITNESS_FUNCTION_MICHALEWICZ` nebo `FITNESS_FUNCTION_ROSENBROCK`. Kromě určení funkce soubor obsahuje také intervaly optimalizace, které jsou použity při generování počáteční populace a v případě penalizace.
- `INTERVAL_PENALISATION` určuje, zda bude prováděna penalizace fitness funkce za každý gen, který je mimo definiční obor, čímž efektivně udržuje jedince ve vymezeném prostoru.
- `MUTATION_RATE` vyjadřuje míru mutace, jak je popsaná v kapitole 2.4. Validní jsou hodnoty v intervalu $\langle 0,0; 1,0\rangle$, jak ukazují měření v kapitole 6.6, může mít velká míra mutace negativní dopad na výkon.
- `MUTATION` určuje, zda vůbec budou prováděny mutace. Jak ukazuje měření v kapitole 6.6, může to mít vliv na výkonnost aplikace, avšak pro evoluci je v tomto případě použit pouze genofond počáteční populace.

5.2 CPU implementace

Práce se zaměřuje spíše na využití GPU, proto je CPU implementaci věnován jen malý prostor.

Zdrojové kódy jsou rozděleny do těchto souborů:

GeneticSolver.cpp implementuje třídu GeneticSolver. Ta je určena k dědění třídami implementující konkrétní funkcionalitu, jako taková poskytuje především metody pro nastavení základních parametrů běhu GA. Mezi ty patří například velikost populace, použitá fitness funkce, ukončující podmínky běhu, funkce pro selekci a podobně.

CPUGeneticSolver.cpp implementuje třídu CPUGeneticSolver, která dědí metody třídy GeneticSolver. Proti ní přidává konkrétní funkcionalitu na CPU, tedy obsahuje metody pro alokaci paměti, výpočet fitness funkcí, realizuje všechny genetické operátory

atd. Kromě vlastní implementace je v programu využit také kvalitní generátor náhodných čísel Mersenne Twister implementovaný Richard J. Wagnerem [50].

constants.h je hlavičkový soubor definující direktivami preprocesoru výchozí parametry GA, intervaly optimalizace a možnosti nastavení pro ladění.

main.cpp obsahuje ukázku použití implementovaných tříd.

Zdrojové kódy jsou přeložitelné s použitím standardních systémových knihoven C++ na více platformách. K implementaci je dodáván soubor `Makefile`, který umožňuje pohodlný překlad a spouštění s použitím `GNU Make`:

make přeloží implementaci do binárního kódu.

make run spustí zkompilovaný zdrojový kód.

make valgrind slouží pro testování práce implementace s pamětí, analyzuje především množství provedených alokací a dealokací.

make profile přeloží, spustí a vyhodnotí implementaci s použitím aplikace `GNU profiler`. Z výstupu je možné identifikovat nejnáročnější části programu pro případné optimalizace.

Implementace byla úspěšně testována programem `valgrind` a řadou zkušebních běhů, ve kterém byla statisticky vyhodnocována z hlediska rychlosti i kvality (viz 6). Prokázala svou plnou funkčnost a vykazuje zrychlení proti GA knihovně GALib (viz 4.2).

5.3 Získávání statistik

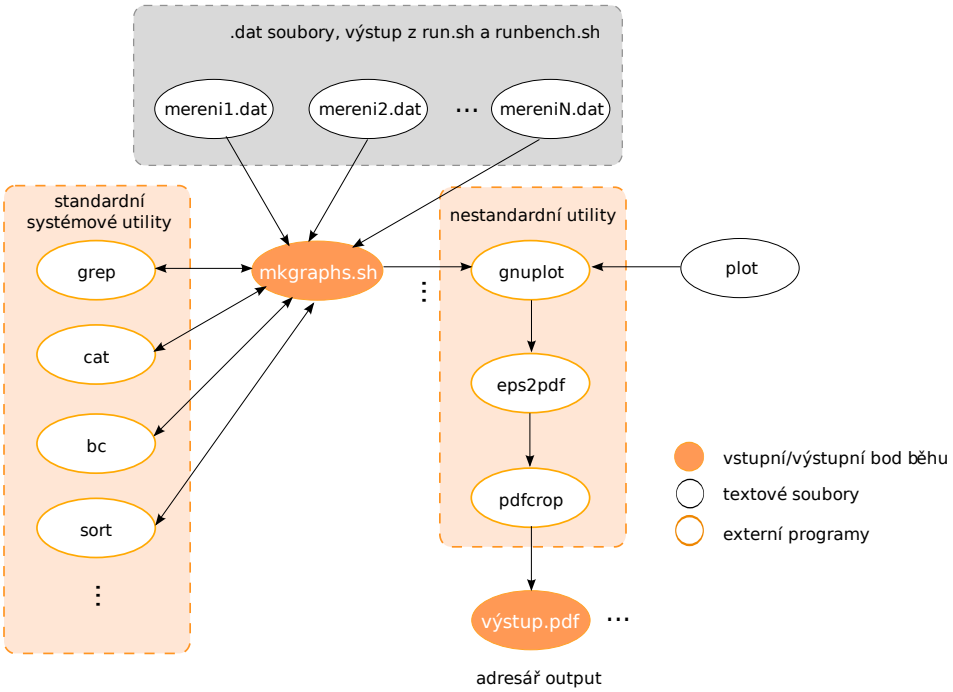
Pro vyhodnocení běhu GPU implementace je na straně CPU určena třída navržená v rámci kapitoly 4.5. Je určená pro snadný sběr a manipulaci se statistikami z většího množství proměnných. Její základ tvoří dva vektory struktur `STATS_VARIABLE_DOUBLE_T` a `STATS_VARIABLE_FLOAT_T`, které jsou plněny při volání metody pro přidání proměnné do kolekce `AddToCollection()` a při samotném sběru dat metodou `CollectStats()`. Obě datové struktury obsahují ukazatel na vyhodnocovanou proměnnou, který je při sběru přečten a uložen do vektoru hodnot. Při dotazu na statistiky jsou pak z vektorů privátní metodou `UpdateVariableStats()` spočítány jednoprůchodově všechny statistiky a ve struktuře je nastavena proměnná `stats_are_actual` na `true`. Při opětovném dotazu na některou ze sbíraných statistik jsou údaje přečteny z paměti a nedochází tak ke zbytečnému novému výpočtu až do doby, než se vektory s hodnotami změní.

K vynulování již nasbíraných hodnot slouží metoda `ClearValues()`, samotné statistiky je pak možné buď vypsat jako celek metodou `PrintStats()`, nebo jednotlivě získávat metodami `GetMin()`, `GetMax()`, `GetAvg()` a `GetDev()`.

Implementovaná třída byla testována, výsledky konfrontované s ručně provedenými výpočty ukazují plnou funkčnost. Zdrojový kód je možné díky multiplatformnosti použít i v budoucích projektech zaměřených na stochastické metody.

5.4 Vyhodnocování statistik

Jak bylo diskutováno v předchozích kapitolách, GA jsou stochastické optimalizační metody, jejichž běh by měl být vyhodnocován statisticky. Optimální formu prezentace statistických



Obrázek 5.5: Struktura implementace pro dávku mkgraphs.sh

dat představuje vizualizace – dává dobrou představu především o závislostech více parametrů.

Pro generování grafů je v OS typu Unix dostupná OpenSource, výborně parametrizovatelná, utilita `Gnuplot` [5]. V novějších verzích má možnost generování 3D grafů, jako výstup lze použít vektorový terminál `eps` podporující barevné gradienty. V implementaci je, spolu s dalšími utilitami pro převod do formátu `pdf`, využita pro grafické zpracování výsledků.

Jak ukazuje obrázek 5.5, obslužná dávka `mkgraphs.sh` využívá standardní systémové utility a výstupních souborů z běhu `run.sh`. Pro výpočet zrychlení jsou ve vstupních souborech ručně doplněny informace o rychlosti běhu GPU. Skript sám o sobě obsahuje několik parametrů, pro nastavení grafického výstupu slouží také soubor `plot`, který je utilitou `gnuplot` načítán v průběhu generování grafů. Ve většině případů je pro vykreslení použito více průchodů.

5.5 Testovací sestava

Tabulka 5.5 prezentuje použitý hardware a software pro implementaci a testování. Jedná se o desktopový počítač s nejmodernějším procesorem (2009) a první generací grafické karty podporující DirectX 10.0, potažmo nVidia CUDA framework[7]. Procesor je tak technologicky mnohem vyspělejší. Práce se zaměřuje na paralelizaci s použitím GPU, na CPU je tedy testována klasická sekvenční verze GA (přestože použitý procesor podporuje vícevláknové zpracování).

Jako operační systém slouží poslední 64bitová verze Ubuntu Linuxu s nejnovějším do-

[7] V době psaní práce již byly na trhu dlouho dostupné grafické karty řady GTX 280 s dvojnásobným teoretickým výkonem, v nejbližší době je očekávána nová generace.

Tabulka 5.1: Použitá testovací sestava

	hardware
procesor	Core i7 920 na 3.2Ghz, 4x256KB L2 cache, 8MB L3 cache, 45nm
operační paměť	4GB DDR3 1600Mhz DualChannel
grafická karta	GeForce 8800GTX, 768MB GDDR3 900Mhz 384bit, 90nm
	software
operační systém	Ubuntu 9.04 x86_64 „Jaunty“
jádro	2.6.28-11-generic x86_64 GNU/Linux
překladač GCC	4.1.3 20080623 (4.1.2-24ubuntu1)
ovladač grafické karty	kernel modul verze 180.44, CUDA toolkit verze 2.0

stupným ovladačem grafické karty. Použitý CUDA toolkit 2.0 vyžaduje GCC překladač verze 4.1, proto byl v systému instalován místo poslední dostupné stabilní verze 4.3. BIOS grafické karty nebyl modifikován od doby nákupu v roce 2007.

Teoretické možnosti GPU G80 jsou blíže diskutovány v kapitole 3, následující kapitola shrnuje dosažené výsledky.

Kapitola 6

Zhodnocení výsledků

Tato kapitola shrnuje hodnoty naměřené v průběhu a na závěr implementace. V první části je prezentována především architektura GPU a závislost množství zpracovávaných dat na výkonu, následuje měření rychlosti a přesnosti dílčích problémů Genetických algoritmů. V závěru kapitoly je analyzována implementace GA na GPU jak z hlediska kvality dosahovaných výsledků, tak z hlediska rychlosti běhu.

6.1 Architektura

Pro analýzu architektury systému a grafické karty byly vybrány následující testy:

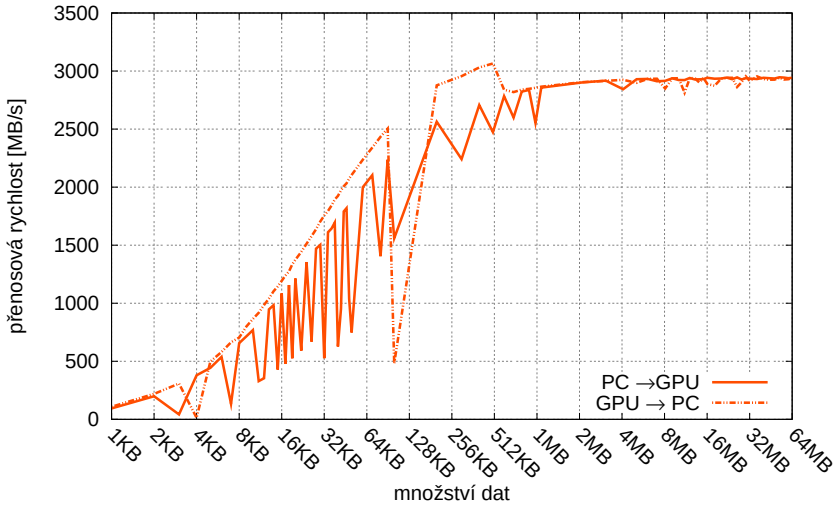
- měření propustnosti nahrávání dat ze systémové paměti do grafického adaptéru
- měření propustnosti nahrávání dat z grafického adaptéru do systémové paměti
- měření propustnosti kopírování dat v rámci grafického adaptéru

Dále byl u přenosů na sběrnici testován rozdíl mezi použitím paměti alokovanou běžnou funkcí `malloc()` a specializovanou `cudaMallocHost()` pro alokaci paměti uzamčené proti odswapování systémem (*page-locked* nebo také pinned memory).

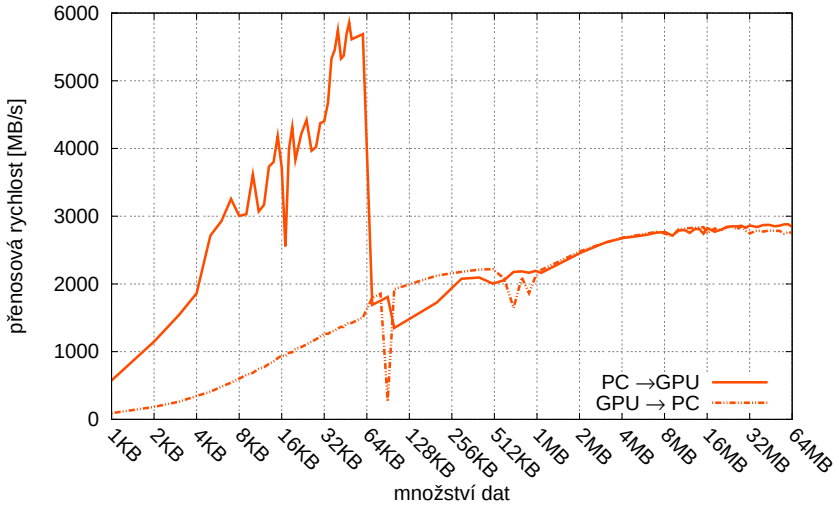
Podle předpokladů se reálné přenosové rychlosti výrazně lišily podle celkového množství přenášených dat, proto bylo provedeno měření pro 81 hodnot velikosti dat M_{size} v rozmezí 1KB až 64MB:

$$M_{size} = \begin{cases} \langle 1; 20 \rangle \text{KB} & \text{po 1KB} \\ \langle 22; 50 \rangle \text{KB} & \text{po 2KB} \\ \langle 60; 100 \rangle \text{KB} & \text{po 10KB} \\ \langle 200; 1000 \rangle \text{KB} & \text{po 100KB} \\ \langle 1; 20 \rangle \text{MB} & \text{po 1MB} \\ \langle 22; 32 \rangle \text{MB} & \text{po 2MB} \\ \langle 36; 64 \rangle \text{MB} & \text{po 4MB} \end{cases}$$

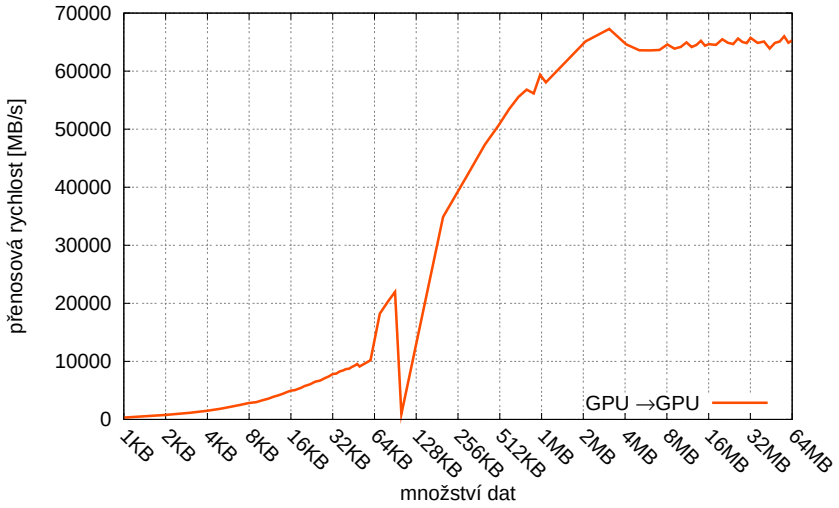
Měření probíhalo pro 100 běhů pro každou velikost dat, výsledek vynesený do grafů je průměrem pro tyto hodnoty. Pro test byl použit demonstrační zdrojový kód `BandwidthTest.cu` z CUDA Software Development Kitu dodávaného přímo výrobcem GPU [12].



(a) hostitelská paměť alokovaná CUDA funkcí cudaMallocHost()



(b) hostitelská paměť alokovaná funkcí malloc()



(c) paměť na grafické kartě

Obrázek 6.1: Porovnání závislostí přenosových rychlostí na velikosti přenášených dat

Dosažené výsledky shrnuje trojice grafů 6.1. Na všech grafech je patrný výkonový propad při čtení z paměti grafické karty pro velikost přenášených dat 100KB. Přenosy obecně vykazují přímou úměru mezi propustností a objemem přenášených dat. Jedinou výjimku tvoří zápis do paměti grafické karty v případě použití běžné alokace paměti (viz obr. 6.1(b)). Vzhledem k tomu, že naměřené hodnoty v tomto případě překračují teoretickou propustnost sběrnice PCI-Express 1.0 16x, která je použita pro propojení hostitelského systému s grafickou kartou, pravděpodobně se jedná o optimalizaci ovladače grafické karty. Ten například může spouštět kopírování do GPU částečně paralelně s během kódu CPU nebo si data uložit do vlastní cache a přenést je až v případě potřeby.

Graf 6.1(c) prezentující přenosové rychlosti v rámci paměti grafické karty vykazuje řádově vyšší rychlosti, než v případě přenosu přes sběrnici. Je tak patrné úzké hrdlo architektury, jak je diskutováno v kapitole 3.2.

Při kopírování z a do GPU s použitím paměti uzamknuté proti swapování je patrné maximum přenosové rychlosti na sběrnici kolem 3GB/s (viz obr. 6.1(a)), což je asi 75% teoretické přenosové rychlosti použitého rozhraní. Těchto rychlostí je však dosaženo pouze s použitím relativně velkého množství dat. Zápis do GPU také značně ztrácí na výkonu v případě dat o velikosti lichých násobků 1024, v případě čtení se však tento nedostatek neprojevuje.

Je zřejmé, že režie inicializace přenosu dat dělá GPU nevhodné pro řadu malých transakcí (například evaluace fitness funkcí malých populací na GPU). Tato vlastnost je částečně kompenzována asynchronním přenosem dat a spouštěním výpočtu na GPU, je však podporována pouze na novějších čipech s Compute Capability 1.2 a výše, které nebyly v rámci této práce testovány.

6.2 Způsob měření

Měření pro dílčí problémy (generování náhodných čísel a evaluace fitness funkcí) probíhalo pro 2^{23} (8388608) položek, u CPU i GPU na datovém typu `float` (celkem tedy 32MB dat). Pokud v textu není řečeno jinak, byl jako výsledek použit průměrem z pěti hodnot. Pro měření času jsou použity integrované funkce CUDA utilit `cutCreateTimer()`, `cutStartTimer()` a `cutStartTimer()`, které mají výstup v milisekundách. Způsob měření času je inspirovaný projekty na [3] – nejdříve je spuštěn časovač, následně GPU kernel, poté je funkcí `cudaThreadSynchronize()` zajištěno dokončení a závěrem je časovač opět zastaven. Je tak změřena doba výpočtu na GPU bez režie přenosu. Pro hodnocení režie přenosu jsou použity dodatečné časovače, které zahrnují také volání funkcí pro přenos dat z a do grafické karty po systémové sběrnici. Pro porovnávání zrychlení proti CPU verzi byly použity nejlepší běhy sekvenční implemetnace.

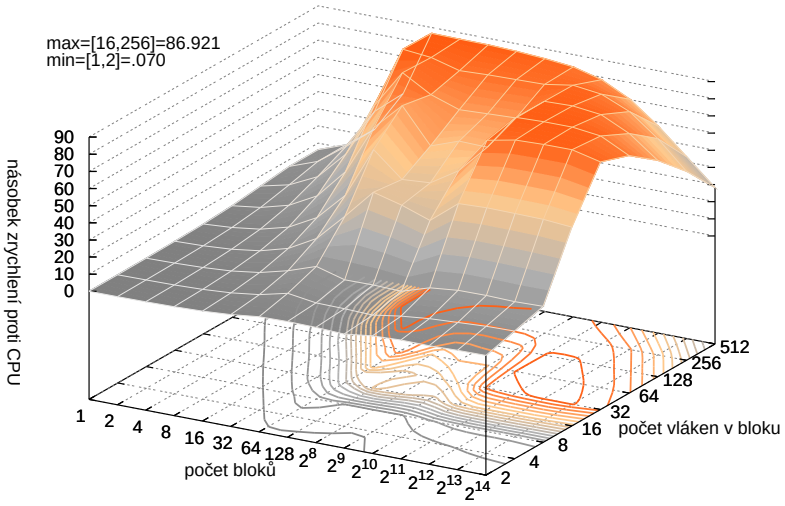
Všechna měření proběhla na testovací sestavě popsané v kapitole 5.5.

6.3 Generování náhodných čísel

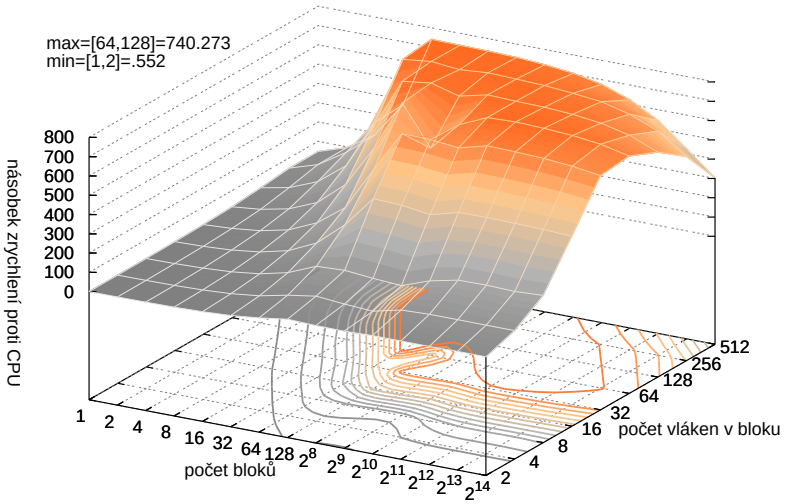
Měření probíhalo podle popisu v kapitole 6.2. Stejně jako všechny ostatní měření je na rychlosti běhu GPU verze velmi výrazně poznat nastavení počtu bloků a vláken v bloku, což prezentují grafy 6.3 – potvrzuje se teoretický předpoklad, že pro dosažení velkého urychlení je potřeba masivní paralelismus. Jde také vidět, že rychlost běhu na GPU velmi silně kolísá (je v obou případech v rozsahu tří řádů).

Tabulka 6.1: Porovnání výkonnosti různých PRNG na CPU a GPU

architektura	**funkce**	$\cdot 10^6$ **vyhodnocení za s**
CPU	Standardní Rand()	114 [50]
	MersenneTwiseter	270 [50]
	HybridTauss	220
	Box-Muller	12
GPU	HybridTauss	10 až 13400
	Box-Muller	2 až 3004
	Box-Muller fastmath	6 až 8933



(a) Uniformní (HybridTauss)



(b) Gaussovský – fastmath (Box-Mullerova transformace z Uniformního)

Obrázek 6.2: Vliv parametrů na rychlost generátorů náhodných čísel v porovnání s CPU

Různé CPU a GPU PRNG pak porovnává tabulka 6.1[1].

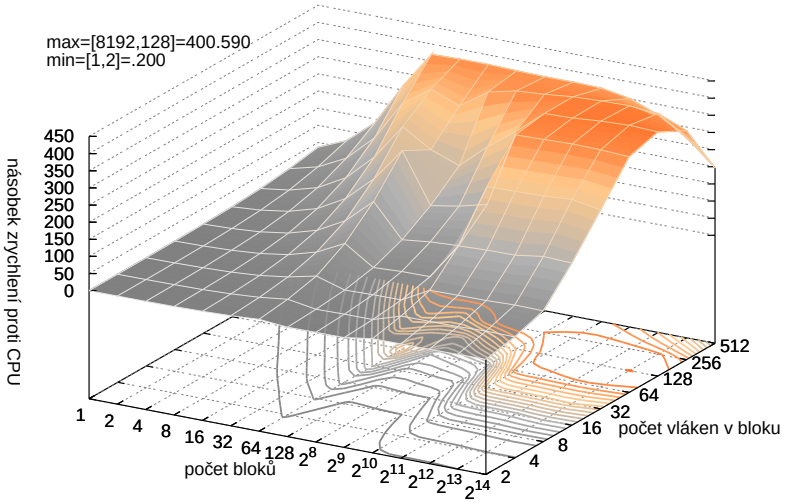
Maximální hodnoty urychlení jsou velmi dobré, v případě Uniformního generátoru náhodných čísel dosahují 85-násobku rychlosti CPU, u Gaussovského dokonce 740-násobku.

Výsledky PRNG se v případě Uniformního generátoru proti CPU téměř neliší, u Gaussovského je v závislosti na použitém parametru kompilátoru absolutní shoda pro 14, resp. 39% výsledků, což prezentuje tabulka následující tabulka.

Tabulka 6.2: Porovnání přesnosti PRNG funkcí proti CPU

funkce	**prům.výstup**	**prům.chyba**	**max.chyba**	**shoda**
BoxMuller-fastmath	0.05636	$1.679777 \cdot 10^{-7}$	$4.41081 \cdot 10^{-5}$	14.39%
BoxMuller	0.05636	$1.159606 \cdot 10^{-10}$	$9.53674 \cdot 10^{-7}$	39.26%
HybridTauss-fastmath	0.500000	$1.159606 \cdot 10^{-10}$	$5.96056 \cdot 10^{-8}$	99.22%

6.4 Evaluace fitness funkcí



Obrázek 6.3: Závislost urychlení Griewankovy funkce na parametrech

Vyhodnocení probíhalo podle stejných kritérií, jako v případě PRNG, tedy dle kapitoly 6.2. Všechny zpracované hodnoty (viz příloha C) vykazují podobné tendence jako graf 6.3 – opět je zřejmé, že na využití potenciálu grafické karty je potřeba především velké množství paralelně zpracovávaných dat. Kromě toho je patrný také výkonový propad pro 256 a 512 bloků (ostrovů) u výpočtů vykazujících výraznější nárůst výkonu. Zpomalení se projevuje také u parametrů, kde se blíží velikost zpracovávaných dat počtu vláken. Je pravděpodobně způsobené omezeným množstvím registrů, které nemohou být hardwarovým plánovačem dostatečně rychle plněny k vytížení ALU jednotek.

Absolutní hodnoty rychlosti a porovnání s během na CPU je shrnuto v tabulce 6.3.

[1] v případě řádků Rand() a MersenneTwister byly pro měření využity skripty dodávané společně s implementací [50]

Tabulka 6.3: Porovnání výkonnosti běhu optimalizovaných funkcí na CPU a GPU

funkce	**architektura**	$\cdot 10^6$ **vyhodnocení za s**
Michalewicz	CPU	6.4
	GPU	1.0 až 2077
	GPU-fastmath	1.8 až 4460
Rosenbrock	CPU	82.3
	GPU	1.5 až 1630
Griewank	CPU	21.5
	GPU	2.3 až 5111
	GPU-fastmath	4.3 až 8630

Tabulka 6.4: Porovnání přesnosti vyhodnocení optimalizovaných funkcí na GPU

funkce	rozsah vstupu	rozsah výstupu	prům.chyba	max.chyba	shoda
Griewank	$\langle$-4.99 ; 4.99$\rangle$	$\langle 2.17 \cdot 10^{-14}$; 2.0$\rangle$	$1.1619 \cdot 10^{-6}$	$3.9339 \cdot 10^{-6}$	6.12%
GriewankF	$\langle$-4.99 ; 4.99$\rangle$	$\langle 6.75 \cdot 10^{-13}$; 2.0$\rangle$	$1.8268 \cdot 10^{-6}$	$7.1525 \cdot 10^{-7}$	2.83%
Michalewicz	$\langle 0 ; \pi \rangle$	$\langle 0$; 0.80130$\rangle$	$4.4286 \cdot 10^{-8}$	$1.0728 \cdot 10^{-6}$	7.38%
MichalewiczF	$\langle 0 ; \pi \rangle$	$\langle 0$; 0.81013$\rangle$	$1.0177 \cdot 10^{-7}$	$2.6822 \cdot 10^{-6}$	5.37%
Rosenbrock	$\langle$-5.12 ; 5.12$\rangle$	$\langle 2.71 \cdot 10^{-5}$; $9.8 \cdot 10^{4} \rangle$	$1.0623 \cdot 10^{-3}$	$2.3438 \cdot 10^{-2}$	33.34%

Podle předpokladů na přesnost vyhodnocení nemělo vliv nastavení počtu vláken a bloků výpočtu na grafické kartě.

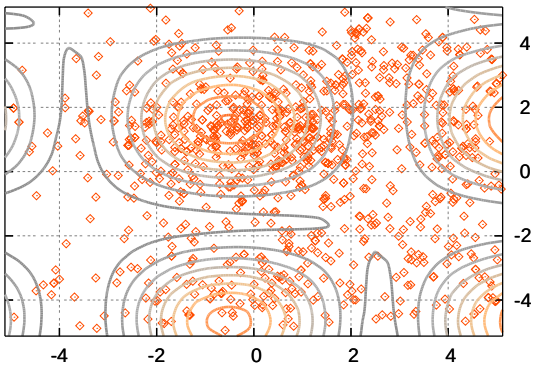
Tabulka 6.4 shrnuje naměřené hodnoty – je vidět, že nastavení kompilátoru `-use_fast_math` (funkce názvem končící na F) mělo vliv na přesnost u funkcí Griewank a Michalewicz, kde se změnila jak průměrná chyba, tak procentuální množství shodných výsledků pro CPU a GPU. U Rosenbrockovy funkce se však zrychlené vyhodnocení matematických funkcí neprojevilo ani na přesnosti, ani na rychlosti výpočtu.

6.5 Migrace jedinců

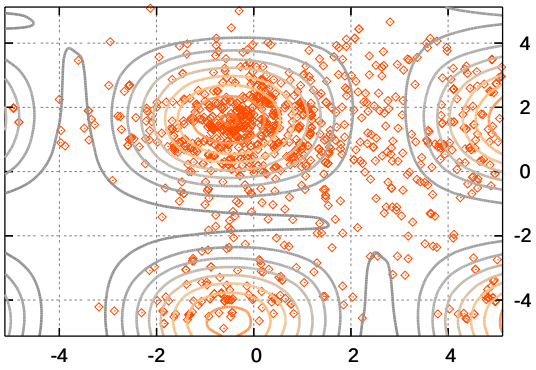
Migrace jedinců mezi ostrovy (viz 4.3) se pozitivně projevuje při konvergenci populace směrem ke globálnímu optimu. Pro testování byla vybrána GPU implementace popsaná v kapitole 5.1.5. Rozdíly jsou nejlépe patrné pro velký počet ostrovů a o malém počtu jedinců, výsledky jsou prezentované pro 128 ostrovů po 8 jedincích. V prvním běhu nebyla prováděna migrace, ve druhém byli migrováni dva nejlepší jedinci každých 5 generací. Vyhodnocována byla Griewankova optimalizační funkce (viz 4.10) pro dva rozměry (geny).

Obrázek 6.5 prezentuje populace jedinců vynesené do vrstevnicového 2D grafu použité fitness funkce. Pro generace 2 a 4 resp. 6.4(a) a 6.4(b) je patrný posun jedinců od náhodné inicializace směrem k lokálním extrémům. Generace 50 pro případ bez migrace (6.4(d)) vykazuje uváznutí jedinců ve všech lokálních extrémech funkce. S použitím migrace se situace lepší (6.4(f)) a je na první pohled zřejmé, že jedinci jsou více soustředění kolem globálního optima přibližně v bodě (-0.6, 1.4). Migrace tedy funguje dle předpokladů a vylepšuje výsledky běhu GA na GPU.

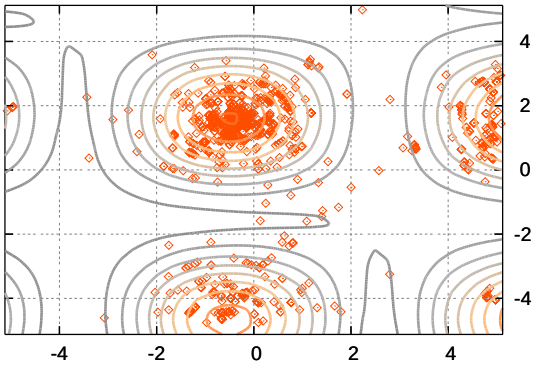
Testování výkonnosti migrace probíhalo pro nastavení uvedené v tabulce 6.5, pro každý parametr 5x, výsledky jsou jejich aritmetickým průměrem. Jak ukazuje graf 6.5, počet



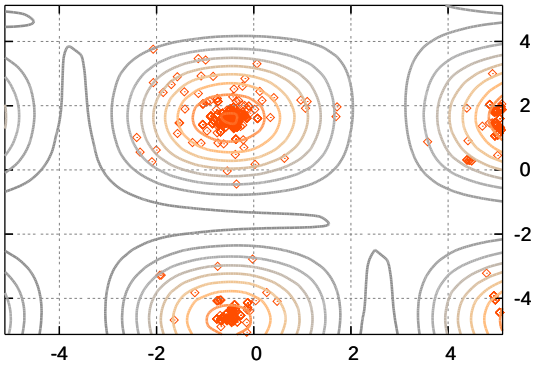
(a) generace 2



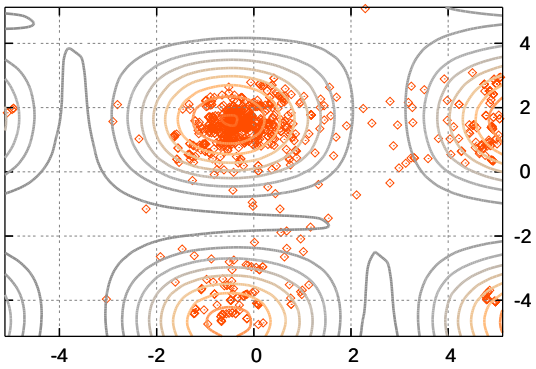
(b) generace 4



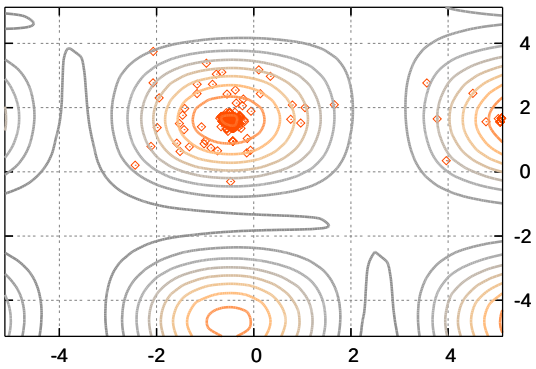
(c) generace 10 bez migrace



(d) generace 50 bez migrace



(e) generace 10 s migrací

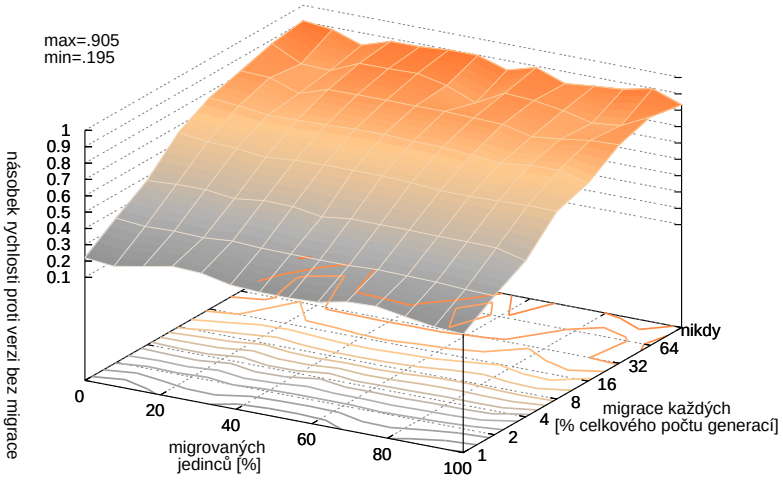


(f) generace 50 s migrací

Obrázek 6.4: Vliv migrace na konvergenci populace ke globálnímu optimu na Griewankově optimalizační funkci

Tabulka 6.5: Nastavení GA při testování dopadu migrace na výkon

parametr	**nastavení**
funkce	Rosenbrock, 2 rozměry (geny)
pravděpodobnost mutace	0.05
pravděpodobnost křížení	0.7
počet jedinců na ostrově	128
počet ostrovů	16
maximální počet generací	100



Obrázek 6.5: Vliv migrace na výkon

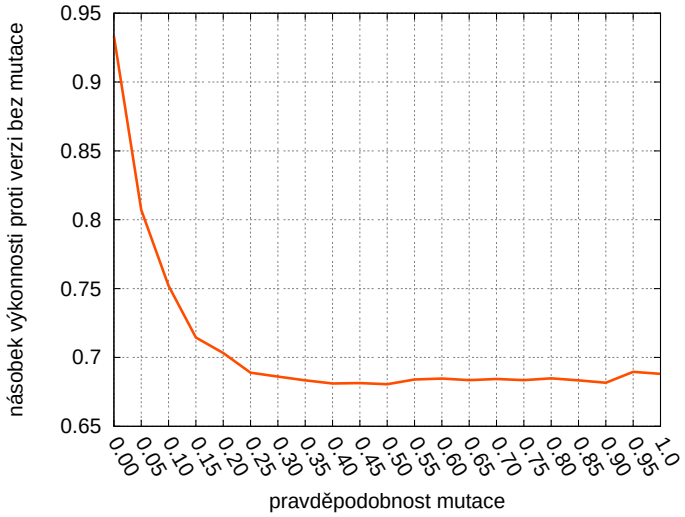
migrovaných jedinců nemá vliv na výkonnost aplikace, protože vlákna jsou po migracích synchronizována bariérou. Naproti tomu počet generací, po kterých jsou jedinci migrováni, má na výkon zcela zásadní dopad – přesun jedinců každou generaci vede k degradaci výkonu na pouhou pětinu, což poukazuje na pomalou hlavní paměť. I v případě, že kvůli podmínce nedojde ani k jedné migraci, je výkonnost snížena přibližně o 10%, což opět potvrzuje optimalizaci GPU na SIMD typ zpracování dat s minimem podmínek a vhodnost parametrizace kódu preprocesorem překladače.

6.6 Mutace a křížení

Podle předpokladů se nastavení míry mutace a křížení projevuje na výkonnosti aplikace. Statistiky se neliší pro různý počet ostrovů, prezentované výsledky jsou pro 8 jedinců na ostrově, 100 generací a Griewankovu funkci na dvou genech. Výsledek je pak průměrem 10ti běhů.

U křížení dopad na výkon není nijak drastický, pravděpodobnost 0,0 a 1,0 se liší maximálně o 10%, navíc tento propad nastává již u hodnoty 0,1, tedy pro běžné míry křížení kolem 0,7 nemá cenu uvažovat nad urychlením změnou nastavení tohoto parametru.

Vliv mutace je o něco větší, při pravděpodobnosti křížení 0,7 ho shrnuje graf 6.6. Jak je vidět, proti běhu zcela bez mutace je výkonový propad již u hodnoty 0,25 více než 30%. Proto se může vyplatit nastavit spíše menší hodnoty mutace a vynahrazovat prohledávání stavového prostoru větší populací, která zároveň vede k většímu využití GPU.



Obrázek 6.6: Vliv pravděpodobnosti mutace na výkon

6.7 Vliv přenosu na sběrnici na celkové urychlení výpočtu

Jak již bylo nastíněno v kapitole 6.1, sběrnice je úzkým hrdlem architektury. Statistiky zahrnující režii přenosu dat z a do grafického adaptéru mají podobný charakter, jako graf 6.7(b) a jsou předmětem přílohy C.2. Jak je zřejmé z porovnání u Michalewiczovy optimalizované funkce na obrázku 6.7, přenos dat na sběrnici, tedy použití GPU jako přídavného akcelerátoru CPU, reálné urychlení silně degraduje. V kombinaci s naměřenými výsledky v kapitole 6.1 (které ukazují, že sběrnice není vhodná na přenos malých objemů dat) vychází najevo, že testovaná architektura není příliš vhodná pro offloading práce z CPU. Inicializace CUDA ze strany grafického ovladače také zabírá při prvním spuštění kernelu až 300ms, na GPU by proto pro dosažení smysluplného urychlení mělo běžet maximum kódu aplikace.

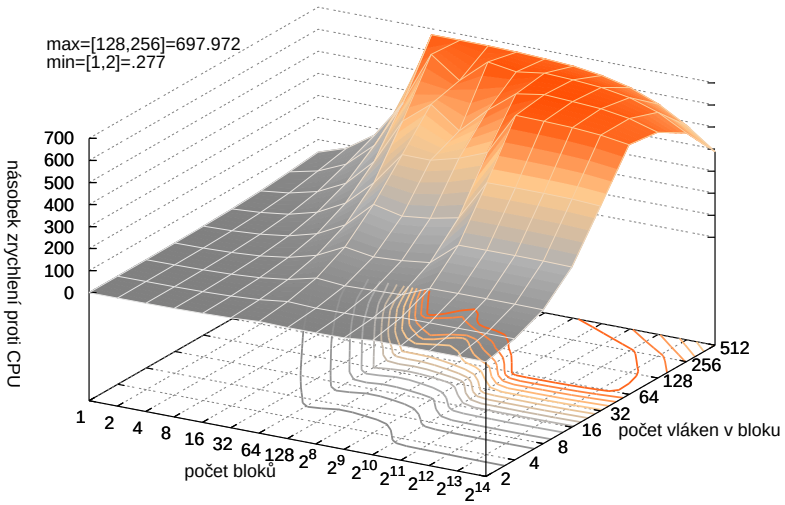
6.8 Rychlost GA na GPU

Vzhledem k tomu, že akcelerace všech dílčích podproblémů GA na GPU je velmi závislá na míře paralelismu, je pro vyjádření rychlosti GA potřeba jednotka, pro kterou bude velikost populace invarianta. Takovou je například počet iterací s geny za jednotku času:

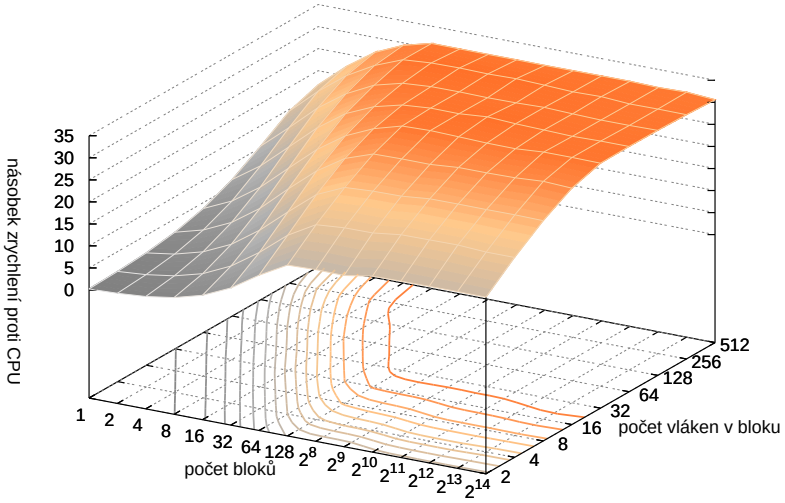
$$S = \frac{J \cdot O \cdot D \cdot G}{t} \tag{6.1}$$

kde

- J je počet jedinců na jednom ostrově (v implementaci `INDIVIDUALS_PER_ISLAND`)
- O je počet ostrovů (v implementaci `ISLANDS`)
- D je počet genů v chromozomu (v implementaci `PROBLEM_DIMENSIONS`)
- G je počet provedených generací (v implementaci `GENERATIONS`)
- t je čas v sekundách (výstup funkce `cuGetTime` vynásobený 1000)



(a) bez režie přenosu, maximální urychlení 700x

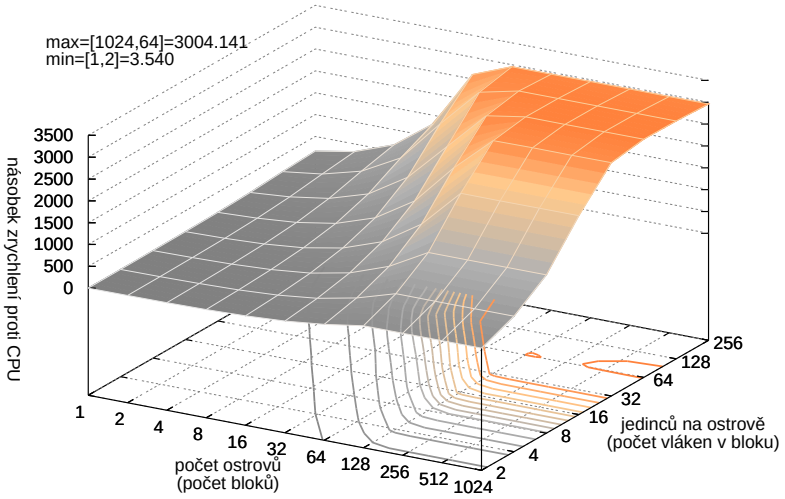


(b) s režií přenosu, maximální urychlení jen 35x

Obrázek 6.7: Vliv režie přenosu po sběrnici na celkové urychlení vyhodnocení Michalewiczovy optimalizované funkce

Tabulka 6.6: Porovnání výkonnosti celého GA na CPU a GPU

architektura	**funkce**	$\cdot 10^6$ **gen-iterací za sekundu**
CPU	Rosenbrockova funkce	5.4 až 5.9
	Michalewiczova funkce	2.9 až 3.3
	Griewankova funkce	3.9 až 4.0
GPU	Rosenbrockova funkce	14.2 až 8877
	Rosenbrockova funkce – fastmath	18.5 až 11914
	Michalewiczova funkce	6.9 až 5893
	Michalewiczova funkce – fastmath	11.7 až 9894
	Griewankova funkce	9.6 až 7108
	Griewankova funkce – fastmath	15.9 až 10507



Obrázek 6.8: Celkové urychlení GA na GPU pro Michalewiczovu funkci

Jak ukazuje měření, běh na CPU testovaný milion generací a 4 až 256 jedinců v populaci má pro tuto jednotku téměř konstantní výkonnost (viz tabulka 6.6) – hodnoty se liší pro rozdíl velikosti populace téměř 2 řády jen o 10%, může tedy na základě ní dojít k objektivnímu porovnání výkonnosti CPU a GPU implementace. Stejná tabulka také shrnuje dosažené absolutní hodnoty výkonu pro GPU. Při výpočtu relativního zrychlení proti CPU verzi byl brán v potaz nejrychlejší běh na CPU plus GPU implementace se stejnými parametry bez použití migrací. Hodnoty ukázaly být velmi pozitivní, k akceleraci dochází již u minimálního paralelismu, pro velké populace jedinců je zrychlení dokonce v řádu jednotek tisíců (viz graf 6.8).

Je otázkou, zda má cenu provádět GA nad tak velkými populacemi jedinců, jaké jsou nutné k dosažení vysokého výkonu na GPU. Vzhledem k tomu, že ostrovy mohou být díky absenci migrace zcela izolované, je možné vyvíjet paralelně velké množství nezávislých populací a simulovat tak typicky mnohokrát opakovaný běh na CPU. Tím dojde k lepšímu prohledání stavového prostoru a bude nalezeno potenciálně lepší optimum. Z grafů je navíc patrné, že 95% maximálního urychlení je možné dosáhnout ve všech případech již pro populaci 32 jedinců na ostrov, což není nijak přehnaná velikost. Jedná se tedy o velmi pozitivní výsledky a praktický přínos je neoddiskutovatelný.

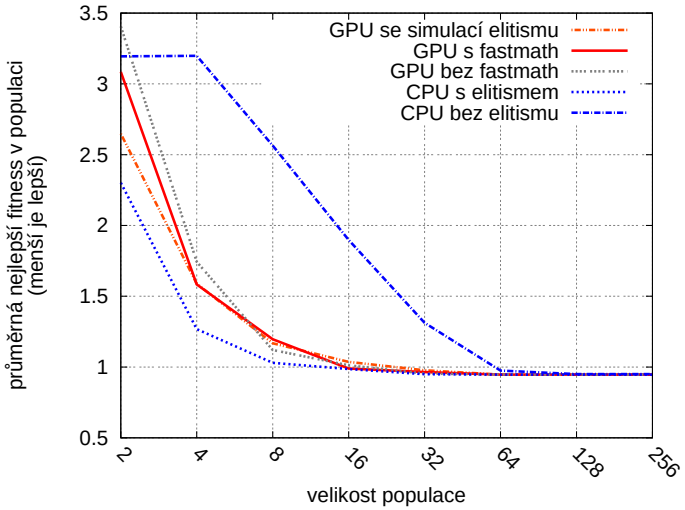
Ukazuje se, že GPU mají obrovský výpočetní potenciál, který je možné s pomocí sdílené paměti a vhodného rozložení problému na softwarový CUDA model efektivně využít.

Kompletní statistika urychlení GA na GPU je pro různé problémy a parametry uvedeny v příloze C.3.

6.9 Kvalita GA na GPU

Testování kvality GA proběhlo pro 50 generací na Griewankově optimalizační funkci (má nejvíce extrémů) pro jednu populaci jedinců variabilní velikosti. Do grafu 6.9 byl vynášen průměr nejlepší dosažené fitness ze 100 běhů pro různé parametry.

Jak je patrné, nejlepší resp. nejhorší výsledky dosahuje CPU verze bez resp. s použitím elitismu. GPU implementace se kvalitativně blíží CPU verzi s elitismem, výsledky jsou tedy pozitivní i v případě přesnosti výpočtu a GPU mají dobrou šanci dosáhnout kvalitních řešení. Simulace elitismu s použitím migrace měla na kvalitu výsledků jen minimální



Obrázek 6.9: Porovnání kvality výsledků pro variabilní velikost populace na CPU a GPU

vliv, došlo však k výraznému propadu ve výkonnosti, pro GPU je tedy vhodnější provádět spíše prohledávání stavového prostoru hrubou silou s využitím velkých populací. Parametr kompilátoru `-use_fast_math` zrychlující vyhodnocení funkcí za cenu přesnosti měl ve vypnutém stavu proti očekávání na kvalitu spíše mírně negativní dopad. Cenou byla ztráta výkonu o desítky procent, proto je pro GA jednoznačně vhodnější běh se zrychleným vyhodnocením matematických funkcí.

V tomto měření nebyla brána v potaz migrace mezi ostrovy, která výsledky ještě dále vylepšuje, jak prezentuje kapitola 6.5.

Kapitola 7

Závěr

Tato závěrečná kapitola shrnuje dosažené výsledky v technické zprávě i implementaci, diskutuje přínos práce v oboru a nastiňuje možnosti dalšího rozvoje.

7.1 Dosažené výsledky

GPU implementace je realizována s použitím vhodně mapovaného softwarového modelu nVidia CUDA frameworku na GA. Používá dělení na ostrovní populace a asynchronní migraci mezi nimi s využitím hlavní paměti. Pro všechny optimalizované funkce dosahuje při vysoké paralelizaci velmi výrazného urychlení proti sekvenční CPU verzi – až 3000 krát, 95% maxima je ve všech případech dosaženo již pro 32 jedinců na ostrov, viz poslední příloha. I pro nízkou úroveň paralelizace je však GPU verze také rychlejší. Z hlediska kvality výsledků je srovnatelná s CPU verzí s použitím elitismu, v případě migrací má lepší konvergenci ke globálnímu optimu. Potvrzuje teoretické předpoklady úzkého hrdla na systémové sběrnici, výrazně těží z rychlosti sdílené paměti v rámci multiprocesoru a potvrzuje fakt, že GPU jsou optimalizované pro masivní paralelismus a SIMD typ zpracování.

Pro CPU běh je s použitím objektového návrhu v C++ implementována třída realizující jednoduchý běh Genetického algoritmu pro různé parametry. Využívá PRNG z [50]. V technické zprávě je porovnána s implementací s použitím GALibu (vykazuje proti ni zrychlení asi o třetinu) a slouží jako reference pro GPU verzi z hlediska rychlosti i kvality.

Analýzu výsledků byla provedena řadou testovacích a vyhodnocovacích skriptů realizovaných s použitím programovacího jazyka `BASH` a systémových utilit jako `GREP`, `GNUPLOT`, `BC` a dalších.

Zdrojové kódy celé práce mají dohromady přibližně 4000 řádků.

Práce[1] vychází ze studia řady podkladů, které jsou citovány v použité literatuře, přílohu tvoří ukázkový zdrojový kód GPGPU využití OpenGL s Cg, použité GPU generátory náhodných čísel a podrobné statistiky urychlení.

7.2 Přínos práce

Možnostmi akcelerace Evolučních výpočtů s použitím grafických čipů se zabývala řada prací, ve většině případů se však jednalo o Genetické programování nebo jejich dílčí podproblémy [30, 20, 9, 19, 75, 45]. Genetické algoritmy jako celek byly akcelerovány pouze v několika

[1] pro vytváření vektorových ilustrací byl použit program InkScape, sazba textu je docílena s použitím systému LaTeX.

málo případech[2], na relativně starém hardware bez použití CUDA [16, 74] a dosahované urychlení se pohybovalo v řádu desítek. Tato práce dosahuje inovativním způsobem realizace GA na GPU urychlení v řádu tisíců a zároveň mapuje možnosti nových GPU. Ukazuje se, že i relativně stará grafická karta GeForce 8 má velký potenciál urychlit GA v porovnání s nejnovějším procesorem (2009) Core i7. Dosažený výkonnostní nárůst umožňuje řešit během sekund úlohy, které dříve trvaly hodiny.

V neposlední řadě je výhodou také cena. Použitou grafickou kartu lze koupit nesrovnatelně levněji, než cluster procesorů podobného výkonu. Implementace šetří i provozní náklady v podobě spotřeby elektrické energie, použitá grafická karta měla přibližně 230-krát lepší výkon na Watt[3], než CPU.

Ostrovy jedinců se mohou vyvíjet samostatně bez migrací, lze proto očekávat, že implementace je výborně škálovatelná na více grafických karet a budoucí GPU a lze tak snadno dosáhnout ještě výraznějších urychlení.

7.3 Možnosti využití

Genetické algoritmy umožňují řešit prakticky libovolný problém, jehož řešení je kódovatelné do chromozomu a je možné provést porovnání z hlediska kvality u dvou kandidátů. Proto je spektrum potenciálních využití aplikace opravdu široké (automatické vytváření inovativních a patentovatelných technologií/technik, optimalizace výroby/návrhu, rychlé řešení NP problémů, hledání extrému u složitých funkcí, ...).

Potenciál skýtají hlavně úkoly vyžadující rychlou adaptaci na měnící se externí podmínky. Vhodné je i nasazení na složité problémy s náročnou fitness funkcí, kde je možné buď prohledávat větší stavový prostor za stejný čas (a najít tak potenciálně lepší řešení), nebo cestu k řešení výrazně urychlit.

7.4 Možnosti dalšího rozvoje

V rámci budoucí práce by mohly být lépe prozkoumány zařízení s lepším CUDA Compute Cability, především možnosti synchronizace s CPU. S pomocí GPU by mohly být urychlovány také obecné Evoluční algoritmy a jiné optimalizační techniky inspirované přírodou. Do budoucna se také rýsuje použití programovacího jazyka OpenCL, který slibuje podobnou syntaxi jako CUDA a běh na všech grafických kartách i vícejádrových procesorech. Ty by tak mohly být porovnány z hlediska potenciálu akcelerovat GA.

[2] v době dokončení práce je v přípravě soutěž v rámci konference GECCO 2009, kde lze očekávat mnoho nových implementací GA na GPU

[3] počítáno pro ideální paralelizaci na 4 jádra + 50% zvýšení výkonu vlivem HyperThreadingu, Thermal Design Power 130W pro Core i7, 185W pro GeForce 8800 GTX a nejlepší dosažené výsledky na každé platformě

Literatura

[1] Internetové stránky grafické knihovny pro realtime zobrazování OpenGL. [Online; navštíveno 14.12.2008].
URL http://opengl.org/

[2] Internetové stránky knihovny pro paralelní výpočty OpenMP. [Online; navštíveno 14.12.2008].
URL http://openmp.org/wp/

[3] Nejlepší studentské projekty paralelního řazení s použitím CUDA na Illinois University. [Online; navštíveno 25.4.2009].
URL http://courses.ece.illinois.edu/ece498/al/HallOfFame.html

[4] Stránky GPGPU komunity. [Online; navštíveno 19.10.2008].
URL http://www.gpgpu.org

[5] Oficiální stránky projektu GNUPLOT. 2009, [Online; navštíveno 18.5.2009].
URL http://www.gnuplot.info/

[6] Advanced Micro Devices, I.: AMD Stream Computing User Guide. 2007.

[7] Breitbart, J.: *Case studies on GPU usage and data structure design*. Diplomová práce, Universität Kassel, Germany, 1999.

[8] Cantu-Paz, E.; Goldberg, D. E.: Efficient Parallel Genetic Algorithms: Theory and Practice. In *Computer Methods in Applied Mechanics and Engineering*, press, 2000, str. 2000.

[9] Chitty, D. M.: A data parallel approach to genetic programming using programmable graphics hardware. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, New York, NY, USA: ACM, 2007, ISBN 978-1-59593-697-4, s. 1566–1573, doi:http://doi.acm.org/10.1145/1276958.1277274.

[10] nVidia Corporation: Cg reference manual. [Online; navštíveno 2.2.2009].
URL http://developer.download.nvidia.com/cg/Cg_2.0/2.1.0016/Cg-2.1_November2008_ReferenceManual.pdf

[11] nVidia Corporation: NVIDIA CUDA Programming Guide 2.0. [Online; navštíveno 2.2.2009].
URL http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf

[12] nVidia Corporation: stránky příkladů použití nVidia CUDA. [Online; navštíveno 11.4.2009].
URL http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html

[13] nVidia Corporation: The CUDA Compiler Driver NVCC. 2007, [Online; navštíveno 18.5.2009].
URL http://sbel.wisc.edu/Courses/ME964/2008/Documents/nvccCompilerInfo.pdf

[14] Fajmon, B.; Růžičková, I.: *skripta Matematika 3*. FEKT VUT, 147–165 s.

[15] Fenlason, J.; Stallman, R.: The GNU Profiler. [Online; navštíveno 4.5.2009].
URL http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html

[16] Fok, K.-L.; Wong, T.-T.; Wong, M. L.: Evolutionary Computing on Consumer Graphics Hardware. *IEEE Intelligent Systems*, ročník 22, č. 2, 2007: s. 69–78.

[17] Garland, M.; Le Grand, S.; Nickolls, J.; aj.: Parallel Computing Experiences with CUDA. *Micro, IEEE*, ročník 28, č. 4, 2008: s. 13–27.
URL http://dx.doi.org/10.1109/MM.2008.57

[18] GPUTech: Stránky grafické knihovny EcoLib. [Online; navštíveno 2.2.2009].
URL http://www.gpucomputing.eu/index3.php?lang=en&page=_library1.php&id=3

[19] Harding, S.; Banzhaf, W.: Fast Genetic Programming and Artificial Developmental Systems on GPUs. In *HPCS '07: Proceedings of the 21st International Symposium on High Performance Computing Systems and Applications*, Washington, DC, USA: IEEE Computer Society, 2007, ISBN 0-7695-2813-9, str. 2, doi:http://dx.doi.org/10.1109/HPCS.2007.17.

[20] Harding, S.; Banzhaf, W.: Fast Genetic Programming on GPUs. In *EuroGP*, *Lecture Notes in Computer Science*, ročník 4445, editace M. Ebner; M. O'Neill; A. Ekárt; L. Vanneschi; A. Esparcia-Alcázar, Springer, 2007, ISBN 978-3-540-71602-0, s. 90–101.

[21] Harris, M.: Introduction to NVIDIA CUDA. [Online; navštíveno 19.10.2008].
URL http://www.gpgpu.org/s2007/slides/06-CUDA-intro.pdf

[22] Holland, J.: *Adaptation in natural and artificial systems*. Ann Harbor: University of Michigan Press, 1975.

[23] Houston, M.: Understanding GPUs through benchmarking. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, New York, NY, USA: ACM, 2007, [Online; navštíveno 14.12.2008].
URL www.gpgpu.org/s2007/slides/08-performance-overview.pdf

[24] Kapadia, A.: Tausworthe Generators. 2001, [Online; navštíveno 23.4.2009].
URL http://www.cs.dartmouth.edu/~akapadia/project2/node9.html

[25] Kider, J.: GPU = Parallel Machine II: Sorting and Hashing. 2005, [Online; navštíveno 25.4.2009].
URL http://www.cis.upenn.edu/~suvenkat/700/lectures/19/sorting-kider.pdf

[26] Kohlmorgen, U.; Schmeck, H.; Haase, K.: Experiences with fine-grained parallel genetic algorithms. *Annals of Operations Research*, ročník 90, 1999: s. 203–219.

[27] Kvasnička, V.: přednášky předmětu Evolučné Algoritmy: Formalizácia Darwinovej evolučnej tórie.

[28] Kvasnička, V.: přednášky předmětu Evolučné Algoritmy: Teória genetického algoritmu.

[29] Kvasnička, V.; Pospíchal, J.; Tiňo, P.: *Evolučné algoritmy*. Bratislava: Slovenská technická univerzita ve vydavatelství STU, 2000, ISBN 80-227-1375-5.

[30] Langdon, W. B.; Banzhaf, W.: A SIMD interpreter for Genetic Programming on GPU Graphics Cards. In *EuroGP*, LNCS, Naples: Springer, 26-28 Březen 2008, forthcoming.
URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/langdon_2008_eurogp.pdf

[31] LeGresley, P.: High Performance Computing with CUDA: Computational Fluid Dynamics. [Online; navštíveno 14.12.2008].
URL http://www.gpgpu.org/sc2008/M02-08_CFD.pdf

[32] Lohn, J. D.; Linden, D. S.; Hornby, G. S.; aj.: Evolutionary Design of an X-Band Antenna for NASA's Space Technology 5 Mission. In *EH '03: Proceedings of the 2003 NASA/DoD Conference on Evolvable Hardware*, Washington, DC, USA: IEEE Computer Society, 2003, ISBN 0-7695-1977-6, str. 155.

[33] Morton, S.: Seismic Imaging on NVIDIA GPUs: Algorithms and Porting & Production Experiences. [Online; navštíveno 14.12.2008].
URL http://www.gpgpu.org/sc2008/M02-05_Seismic_imaging.pdf

[34] Nguyen, H.: *Gpu gems 3*. Addison-Wesley Professional, 2007, ISBN 9780321545428, [Online; navštíveno 11.4.2009].
URL http://developer.nvidia.com/object/gpu-gems-3.html

[35] Nógráadi, D.: Lattice simulation on graphics cards. [Online; navštíveno 19.10.2008].
URL http://www.physics.utah.edu/lat06/abstracts/sessions/alg_mach_net/s1/Nogradi_Daniel.pdf

[36] Owens, J.: GPU Architecture Overview. [Online; navštíveno 19.10.2008].
URL http://www.gpgpu.org/s2007/slides/02-gpu-architecture-overview-s07.pdf

[37] Pavlovič, J.: *Multikriteriální hybridní evoluční algoritmy pro výběr a optimalizaci dekontaminačních technologií*. Dizertační práce, Masarykova Univerzita, 2006, [Online; navštíveno 14.12.2008].
URL http://is.muni.cz/th/4035/fi_r/

[38] Pelikán, J.; Herout, A.: studijní materiály předmětu Pokročilá Počítačová Grafika. 2008, [Online; navštíveno 1.12.2008].

[39] Pharr, M.; Fernando, R.: *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005, ISBN 0321335597, [Online; navštíveno 25.4.2009].
URL http://developer.nvidia.com/object/gpu_gems_2_home.html

[40] Phillips, J.: High Performance Computing with CUDA: Molecular Dynamics. [Online; navštíveno 14.12.2008].
URL http://www.gpgpu.org/sc2008/M02-07_Molecular_dynamics.pdf

[41] Pit, L. J.: *Parallel Genetic Algorithms*. Diplomová práce, Leiden University, 1995.

[42] Pospíchal, P.: Projekt předmětu Evoluční Algoritmy. 2008, [Online].
URL http://petr.pospichal.biz/EVO/dokumentace.pdf

[43] Pošík, P.: Paralelní genetické algoritmy. [Online; navštíveno 11.12.2008].
URL http://www.volny.cz/posa/skola/pga/theory/pga-theory.htm

[44] Psarras, J.: GA's based decision support systems in production scheduling. *Int. J. Intell. Syst. Technol. Appl.*, ročník 2, č. 1, 2007: s. 58–76, ISSN 1740-8865, [Online; navštíveno 20.12.2008].
URL http://dx.doi.org/10.1504/IJISTA.2007.011574

[45] Robilliard, D.; Marion-Poty, V.; Fonlupt, C.: Population Parallel GP on the G80 GPU. In *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, *Lecture Notes in Computer Science*, ročník 4971, editace M. O'Neill; L. Vanneschi; S. Gustafson; A. I. Esparcia Alcazar; I. De Falco; A. Della Cioppa; E. Tarantino, Naples: Springer, 26-28 Březen 2008, s. 98–109, doi:doi:10.1007/978-3-540-78671-9_9.

[46] Schwehm, M.: Parallel population models for genetic algorithms. Technická zpráva, Universitat Erlangen-Nurnberg, 1996.

[47] Seiler, L.; Carmean, D.; Sprangle, E.; aj.: Larrabee: A Many-Core x86 Architecture for Visual Computing. Technická zpráva, 2008.

[48] Trevett, N.: OpenCL – The Open Standard for Heterogeneous Parallel Programming. 2008, [Online; navštíveno 11.12.2008].
URL http://khronos.org/opencl/presentations/OpenCL_Summary_Nov08.pdf

[49] Vertanen, K.: Genetic Adventures in Parallel: Towards a Good Island Model under PVM. Technická zpráva, Oregon State University, 1998.
URL http://www.inference.phy.cam.ac.uk/kv227/papers/island_model_pvm.pdf

[50] Wagner, R.: Mersenne Twister Random Number Generator – C++ Class. 2003, [Online; navšíveno 16.5.2009].
URL http://www-personal.umich.edu/~wagnerr/MersenneTwister.html

[51] Wall, M.: GALib — C++ Knihovna pro optimalizaci s použitím genetických algoritmů. [Online; navštíveno 4.5.2009].
URL http://lancet.mit.edu/ga/

[52] Whitley, D.; Rana, S.; Heckendorn, R. B.: The island model genetic algorithm: On separability, population size and convergence. *Journal of Computing and Information Technology*, ročník 7, 1999: s. 33–47.

[53] Wikipedia: Cg (programming language) — Wikipedia, The Free Encyclopedia. 2008, [Online; navštíveno 2.2.2009].
URL
http://en.wikipedia.org/w/index.php?title=Cg_(programming_language)&oldid=252967015

[54] Wikipedia: Computational creativity — Wikipedia, The Free Encyclopedia. 2008, [Online; navštíveno 17.12.2008].
URL http://en.wikipedia.org/w/index.php?title=Computational_creativity&oldid=257164405

[55] Wikipedia: DirectX — Wikipedia, The Free Encyclopedia. 2008, [Online; navštíveno 2.2.2009].
URL http://en.wikipedia.org/w/index.php?title=DirectX&oldid=261057811

[56] Wikipedia: Evolution strategy — Wikipedia, The Free Encyclopedia. 2008, [Online; navštíveno 17.12.2008].
URL http://en.wikipedia.org/w/index.php?title=Evolution_strategy&oldid=258402550

[57] Wikipedia: Evolutionary algorithm — Wikipedia, The Free Encyclopedia. 2008, [Online; navštíveno 17.12.2008].
URL http://en.wikipedia.org/w/index.php?title=Evolutionary_algorithm&oldid=255654009

[58] Wikipedia: Evolutionary programming — Wikipedia, The Free Encyclopedia. 2008, [Online; navštíveno 17.12.2008].
URL http://en.wikipedia.org/w/index.php?title=Evolutionary_programming&oldid=185676198

[59] Wikipedia: Genetic algorithm — Wikipedia, The Free Encyclopedia. 2008, [Online; navštíveno 17.12.2008].
URL http://en.wikipedia.org/w/index.php?title=Genetic_algorithm&oldid=258298132

[60] Wikipedia: Genetic programming — Wikipedia, The Free Encyclopedia. 2008, [Online; navštíveno 17.12.2008].
URL http://en.wikipedia.org/w/index.php?title=Genetic_programming&oldid=257886368

[61] Wikipedia: GLSL — Wikipedia, The Free Encyclopedia. 2008, [Online; navštíveno 2.2.2009].
URL http://en.wikipedia.org/w/index.php?title=GLSL&oldid=246721750

[62] Wikipedia: Gray code — Wikipedia, The Free Encyclopedia. 2008, [Online; navštíveno 15.12.2008].
URL http://en.wikipedia.org/w/index.php?title=Gray_code&oldid=258138341

[63] Wikipedia: High Level Shader Language — Wikipedia, The Free Encyclopedia. 2008, [Online; navštíveno 1.12.2008].
URL
http://en.wikipedia.org/w/index.php?title=High_Level_Shader_Language&oldid=248081635

[64] Wikipedia: Normal distribution — Wikipedia, The Free Encyclopedia. 2008, [Online; navšíveno 17.12.2008].
URL http://en.wikipedia.org/w/index.php?title=Normal_distribution&oldid=257016524

[65] Wikipedia: OpenGL Utility Toolkit — Wikipedia, The Free Encyclopedia. 2008, [Online; navštíveno 2.2.2009].
URL http://en.wikipedia.org/w/index.php?title=OpenGL_Utility_Toolkit&oldid=245061076

[66] Wikipedia: Simulated annealing — Wikipedia, The Free Encyclopedia. 2008, [Online; navšíveno 17.12.2008].
URL http://en.wikipedia.org/w/index.php?title=Simulated_annealing&oldid=258047207

[67] Wikipedia: Box-Muller transform — Wikipedia, The Free Encyclopedia. 2009, [Online; navštíveno 23.4.2009].
URL
http://en.wikipedia.org/w/index.php?title=Box%E2%80%93Muller_transform&oldid=282607751

[68] Wikipedia: Graphics processing unit — Wikipedia, The Free Encyclopedia. 2009, [Online; navštíveno 2.2.2009].
URL http://en.wikipedia.org/w/index.php?title=Graphics_processing_unit&oldid=261435528

[69] Wikipedia: Linear congruential generator — Wikipedia, The Free Encyclopedia. 2009, [Online; navštíveno 23.4.2009].
URL http:
//en.wikipedia.org/w/index.php?title=Linear_congruential_generator&oldid=283719976

[70] Wikipedia: Mersenne twister — Wikipedia, The Free Encyclopedia. 2009, [Online; navštíveno 23.4.2009].
URL http://en.wikipedia.org/w/index.php?title=Mersenne_twister&oldid=284860642

[71] Wikipedia: Random number generation — Wikipedia, The Free Encyclopedia. 2009, [Online; navštíveno 23.4.2009].
URL http://en.wikipedia.org/w/index.php?title=Random_number_generation&oldid=285565890

[72] Wikipedia: Windows 7 — Wikipedia, The Free Encyclopedia. 2009, [Online; navštíveno 6.1.2009].
URL http://en.wikipedia.org/w/index.php?title=Windows_7&oldid=262132348

[73] Wright, A. H.: The exact schema theorem. Technická zpráva, University of Montana, 2000.

[74] Yu, Q.; Chen, C.; Pan, Z.: Parallel Genetic Algorithms on Programmable Graphics Hardware. In *Advances in Natural Computation, First International Conference, ICNC 2005, Proceedings, Part III*, *Lecture Notes in Computer Science*, ročník 3612, editace L. Wang; K. Chen; Y.-S. Ong, Changsha, China: Springer, August 27-29 2005, ISBN 3-540-28320-X, s. 1051–1059, doi:doi:10.1007/11539902_134.
URL http://www.cad.zju.edu.cn/home/yqz/projects/gagpu/icnc05.pdf

[75] Šimek, V.; Vašíček, Z.; Slaný, K.: Can the performance of GPGPU really beat CPU in evolutionary design task? In *4th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, Masaryk University, 2008, ISBN 978-80-7355-082-0, s. 264–264.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=8804

Seznam použitých zkratek a symbolů

BIOS (*Basic Input-Output System*, základní vstup-výstupní systém) — část softwarového vybavení počítače nebo jiného zařízení poskytující nejnižší úroveň přístupu k hardware

Cg (*C for graphics*, C pro grafiku) – programovací jazyk původně vyvinutý firmou nVidia pro popis shaderů v syntaxi podobné jazyku C

CPU (*Central Processing Unit*, centrální výpočetní jednotka) – procesor počítače

CUDA (*Compute Unified Device Architecture*, jednotná architektura pro výpočty na zařízeních) – knihovna firmy nVidia pro obecné výpočty s použitím grafických čipů, funguje na grafických kartách řady GeForce 8 a výše

GA (*Gentic Algorithm*, Genetický algoritmus) – stochastická optimalizační metoda pro prohledávání stavového prostoru inspirovaná přírodou

GFLOP (*Giga Floating Point Operations per second*, počet miliard operací v plovoucí desetinné čárce za sekundu) – jednotka používaná v souvislosti s měřením výkonnosti grafických čipů a procesorů v počítačích

GLSL (*OpenGL Shading Language*, jazyk pro popis shaderů v OpenGL) – alternativa Cg pro OpenGL

GPU (*Graphics Processing Unit*, grafická výpočetní jednotka) – čip na grafické kartě, obvykle optimalizovaný na zpracování masivně paralelních, výpočetně-intenzivních úloh, v minulosti spíše hardwarový rasterizér grafických primitiv

GPGPU (*General Purpose computing on GPU*, obecné výpočty s použitím grafického čipu) – v minulosti byly realizovány s použitím textur a shaderovacích programů, dnes již existují pro tyto operace knihovny jako CUDA nebo AMD Brook+. V poslední boě nabývají na důležitosti s rychle rostoucím výkonem a obecností grafických čipů

HLSL (*High Level Shading Language*, vysokoúrovňový jazyk pro popis shaderů) – alternativa Cg pro DirectX

SDK (*Software Development Kit*, softwarový vývojový balík) – sada programů, utilit a příkladů pro usnadnění vývoje aplikací

SIMD (*Single Instruction, Multiple Data*, jedna instrukce, více dat) – typ zpracování dat při kterém je architektura uzpůsobená pro vykonávání jedné instrukce nad větším množstvím dat paralelně

OS (*Operation System*, operační systém) – softwarová mezivrstva mezi hardware počítače a uživatelskými programy

OpenGL (*Open Graphics Library*, otevřená grafická knihovna) – vedle DirectX nejpoužívanější knihovna pro akceleraci realtime grafiky, je vyvíjená konsorciem firem KHRONOS

PRNG (*Pseudorandom Number Generator*, generátor pseudonáhodných čísel) – algorimus pro generování sekvence čísel, která aproximuje vlastnosti náhodných čísel

Seznam příloh

Příloha A

Příklad použití Cg pro GPGPU

Výpočet $x_i = y_i + z_i, i = 1 \ldots 4 \cdot m \cdot n$

Shader v Cg

```
struct FragmentOut { float4 color0:COLOR0; };
FragmentOut example( in float2 myTexCoord:WPOS,
 uniform samplerRECT y,uniform samplerRECT z )
{
 FragmentOut c;
 c.color0 = texRECT( y, myTexCoord ) + texRECT( z, myTexCoord );
 return c;
}
```

Program v C/OpenGL

```
GLuint X;
glGenTextures( 1, &X);
glBindTexture( . . ., X);
glTexParameteri(. . .);
glTexImage2D( . . ., n, m, . . ., 0 ); ..... // také pro Y a Z
// přenos dat z CPU do GPU prostřednictvím OpenGL
glBindTexture( . . ., Y);
glFramebufferTexture2DEXT( . . ., Y, . . . );
glTexSubImage2D( . . ., n, m, . . ., y );..... // také pro Z
// spustí samotný výpočet s použitím Cg shaderu
cgGLSetTextureParameter( . . ., Y);
cgGLEnableTextureParameter( . . ., "y" );..... //  také pro Z
glFramebufferTexture2DEXT( . . ., X, 0 );
glDrawBuffer( . . . );
cgGLBindProgram( . . . );
glBegin( . . . );
{
 glVertex2f( -n, -m);
 glVertex2f( n, -m);
 glVertex2f( n, m);
 glVertex2f( -n, m);
}
glEnd( );
// přenos dat GPU-CPU v OpenGL
glFramebufferTexture2DEXT( . . ., X, . . . );
glReadBuffer( . . . );
glReadPixels( . . ., n, m, . . ., x);
```

zdrojový kód pro CPU v C

```
for( i = 0; i < 4 * n * m; i++ ) x[i] = y[i] + z[i];
```

Příloha B

Použitý generátor náhodných čísel na GPU

Všechny generátory jsou zmíněny v knize [34] publikované na stránkách firmy nVidia.

```
// Combined Tauss generátor
unsigned TausStep(unsigned &z, int S1, int S2, int S3, unsigned M)
{
  unsigned b=(((z << S1) ^ z) >> S2);
  return z = (((z & M) << S3) ^ b);
}

// Linear Congruent generátor
unsigned LCGStep(unsigned &z, unsigned A, unsigned C)
{
  return z=(A*z+C);
}


// generátor Uniformního rozložení využívající dvou předchozích
// každé vlákno využívá svoje proměnné z1-z4
float HybridTaus()
{
  // Combined period is lcm(p1,p2,p3,p4)~ 2^121
   return 2.3283064365387e-10 * (              // Periody
    TausStep(z1, 13, 19, 12, 4294967294UL) ^  // p1=2^31-1
    TausStep(z2, 2, 25, 4, 4294967288UL) ^    // p2=2^30-1
    TausStep(z3, 3, 11, 17, 4294967280UL) ^   // p3=2^28-1
    LCGStep(z4, 1664525, 1013904223UL)        // p4=2^32
   );
}


// generátor Normálního rozložení, využívá dvou generování Uniformního rozložení
float2 BoxMuller()
{
  float u0=HybridTaus (), u1=HybridTaus ();
  float r=sqrt(-2 log(u0));
  float theta=2*PI*u1;
  return make_float2(r*sin(theta),r*cos(theta));
}
```

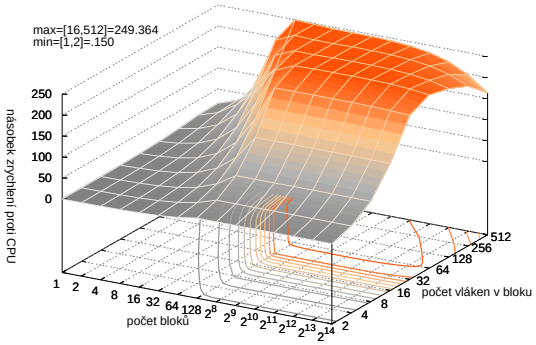
Příloha C

Podrobné statistiky urychlení na GPU

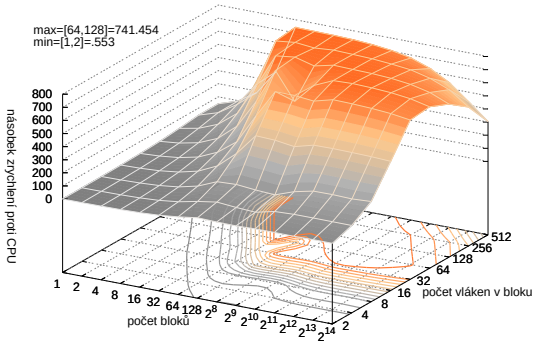
Příloha shrnuje naměřené výsledky z hlediska urychlení pro různé nastavení parametrů. První série grafů (C.1) představuje urychlení dílčích problémů bez zahrnutí režie přenosu přes sběrnici (tj. vykonávání samotného GPU kernelu plus vláknové synchronizace, aby bylo zajištěno blokující provedení). Naproti tomu druhá skupina grafů (C.2) ukazuje stejné výsledky se zahrnutím režie přenosu a ukazuje tak nevhodnost GPU pro částečný offloading práce z CPU.

Závěrečná kompilace C.3 představuje celkové urychlení GA na GPU v závislosti na parametrech proti jednovláknové verzi na testovaném CPU.

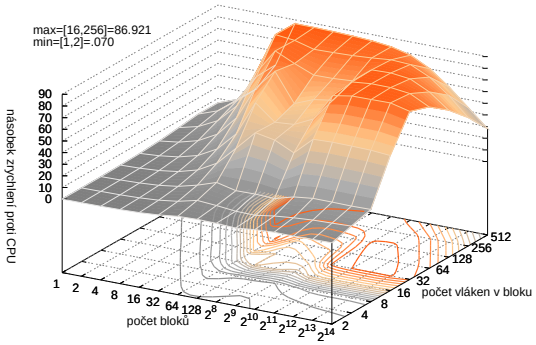
Detaily měření a kometáře jsou uvedeny v kapitole 6.



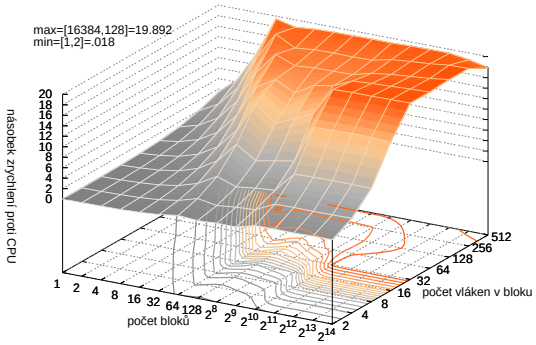
(a) Gaussovský PRNG bez fast-math



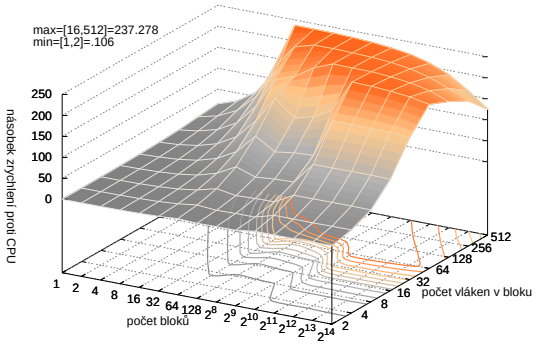
(b) Gaussovský PRNG s fast-math



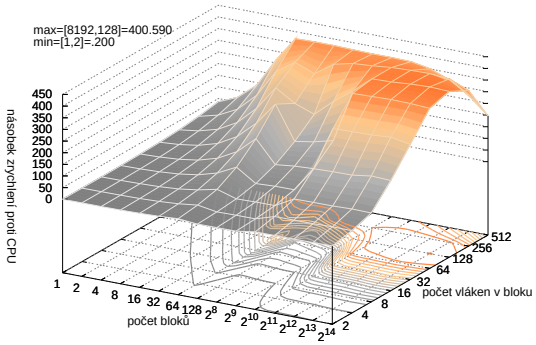
(c) Uniformní PRNG (fast-math se neprojevuje)



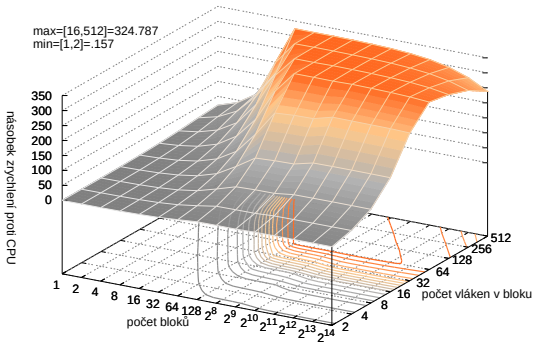
(d) Rosenbrockova fce (fast-math se neprojevuje)



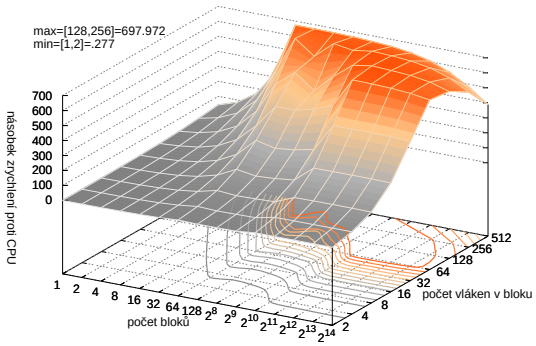
(e) Griewankova funkce bez fast-math



(f) Griewankova funkce s fast-math

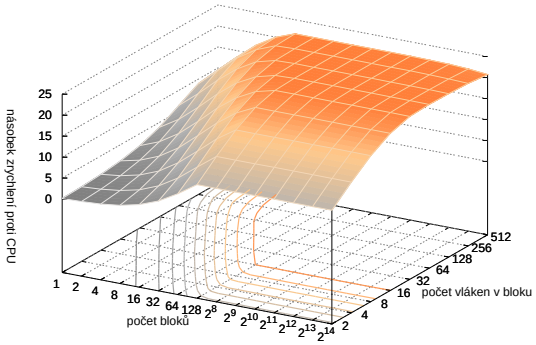


(g) Michalewiczova funkce bez fast-math

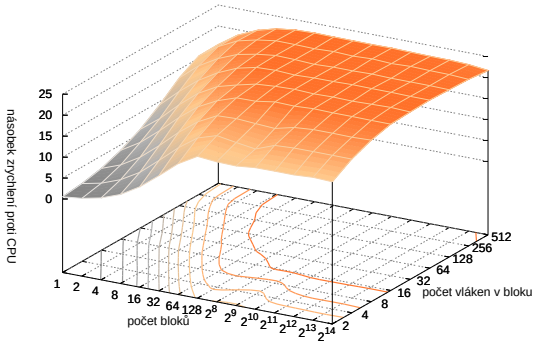


(h) Michalewiczova funkce s fast-math

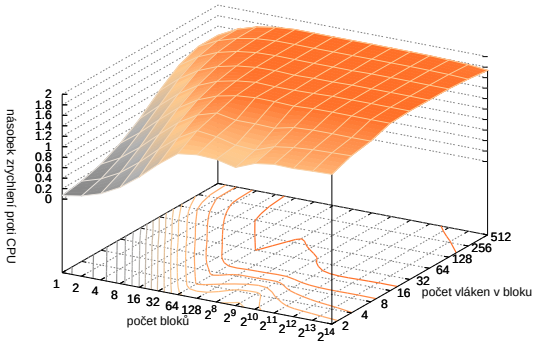
Obrázek C.1: Urychlení dílčích podproblémů proti jednovláknové CPU verzi bez režie



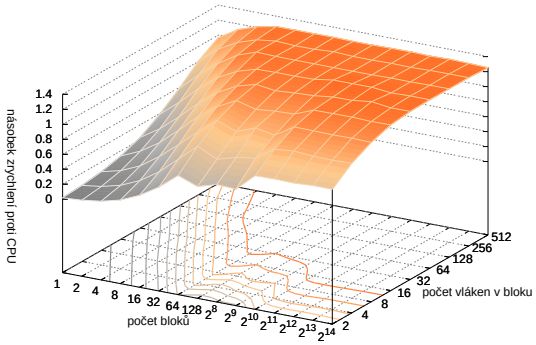
(a) Gaussovský PRNG bez fast-math



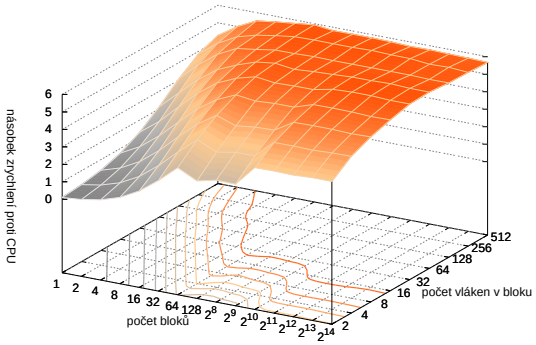
(b) Gaussovský PRNG s fast-math



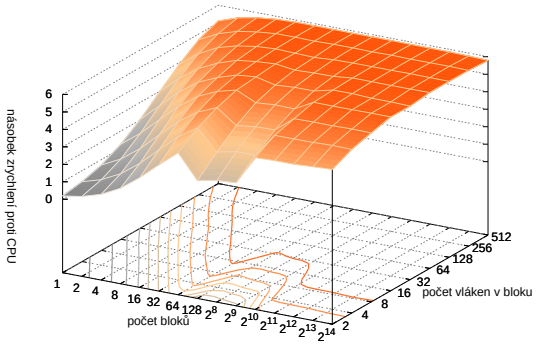
(c) Uniformní PRNG (fast-math se neprojevuje)



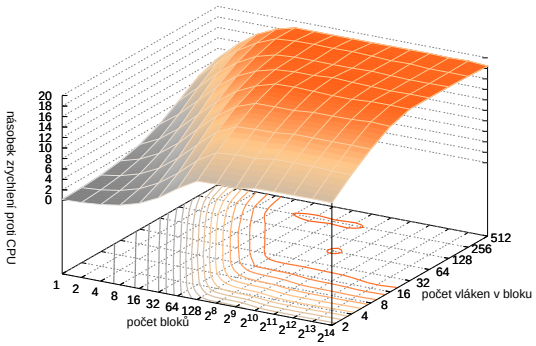
(d) Rosenbrockova fce (fast-math se neprojevuje)



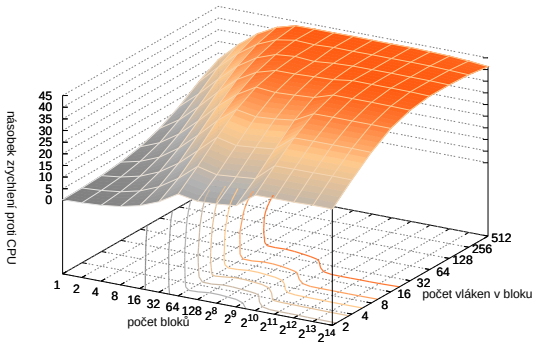
(e) Griewankova funkce bez fast-math



(f) Griewankova funkce s fast-math

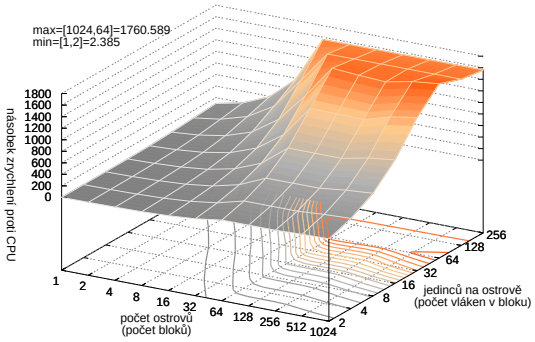


(g) Michalewiczova funkce bez fast-math

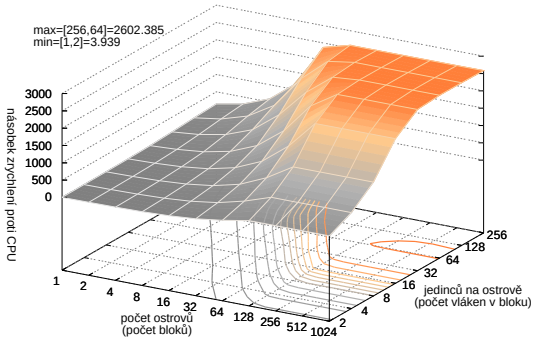


(h) Michalewiczova funkce s fast-math

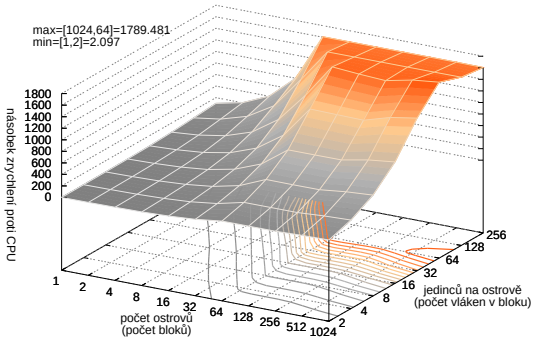
Obrázek C.2: Urychlení dílčích podproblémů proti jednovláknové CPU verzi s režií



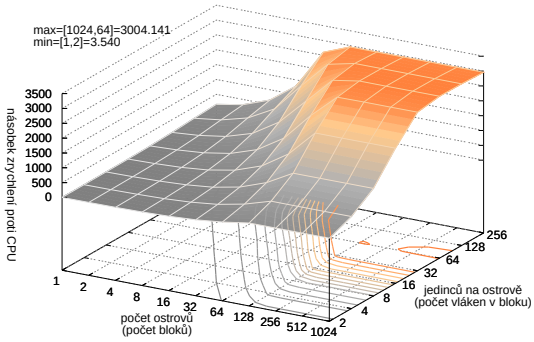
(a) Griewankova funkce bez fast-math



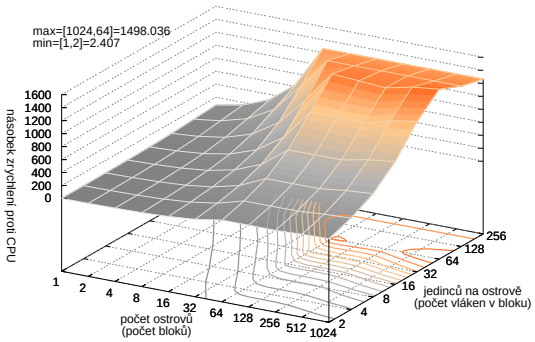
(b) Griewankova funkce s fast-math



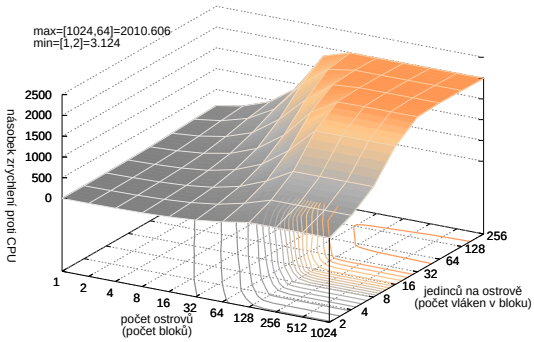
(c) Michalewiczova funkce bez fast-math



(d) Michalewiczova funkce s fast-math



(e) Rosenbrockova funkce bez fast-math



(f) Rosenbrockova funkce s fast-math

Obrázek C.3: Urychlení celého Genetického algoritmu proti sekvenční CPU verzi