FAKULTA STROJNÍHO INŽENÝRSTVÍ
ústav automatizace a informatiky

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
FACULTY OF MECHANICAL ENGINEERING

ÚSTAV AUTOMATIZACE A INFORMATIKY
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

GPU AKCELEROVANÉ METAHEURISTIKY VE VYBRANÝCH ÚLOHÁCH GLOBÁLNÍ A KOMBINATORICKÉ OPTIMALIZACE

GPU ACCELERATED METAHEURISTICS FOR GLOBAL AND COMBINATORIAL OPTIMIZATION TASKS

DISERTAČNÍ PRÁCE
DOCTORAL THESIS

AUTOR PRÁCE Ing. Ladislav Dobrovský
AUTHOR

ŠKOLITEL prof. Ing. Radomil Matoušek, Ph.D.
SUPERVISOR

BRNO 2024

ABSTRAKT

Získávání dostatečně dobrých řešení v rozumném čase je jednou z hlavních náplní inženýrské práce. Přesněji v kontextu inženýrské optimalizace je to hledání parametrů či vhodnější struktury s kompromisem protichůdných požadavků. Řešené problémy jsou stále větší a složitější výzvou. K jejich zvládnutí je třeba neustále inovovat řešící metody, či jejich implementace. Jednou skupinou těchto metod jsou právě metaheuristické algoritmy, kterými se tato práce podrobně zabývá. Jsou teoreticky představeny a prakticky implementovány dvě metody vhodné pro akceleraci na grafických procesorech pro obecné výpočty (GPGPU). První je rozšíření HC12 algoritmu o tzv. Tabu list. Druhá je nová varianta diferenciální evoluce s více ostrovními populacemi s GPU optimalizací genetických operátorů. Prostor je také věnován přehledu prostředků pro vysoce náročné výpočty (HPC). Metody jsou ověřeny na úlohách globální spojité a kombinatorické optimalizace (QAP a SAT). Pro problém SAT jsou verifikovány dvě varianty GPU akcelerace. Pro úplnost je provedeno porovnání implementací pro více jádrová CPU a pro GPU na HC12 a kombinatorické úloze QAP.

ABSTRACT

Achieving sufficiently good solutions in a reasonable time is one of the main focuses of engineering work. More precisely, in the context of engineering optimization, it involves searching for parameters' values or a better structure with. To reach a compromise is often needed for conflicting requirements.. New problems become increasingly challenging. To manage them, it is necessary to innovate methods and their implementation. One group of these methods are metaheuristic algorithms and this work describe their principles and implementation in detail. Two methods suitable for acceleration on general purpose graphics processors (GPGPUs) are theoretically presented and practically implemented. The first is the extension of the HC12 algorithm by the so-called Tabu list. The second is a new variant of differential evolution (DE) with multiple island populations suitable for GPU optimization of genetic operators. Space is also devoted to an overview of high-performance computing (HPC) hardware resources. The methods are verified on tasks of global continuous and combinatorial optimization (QAP and SAT). Two variants of GPU acceleration are verified for the SAT problem. For completeness, implementations for multi-core CPUs and for GPUs on HC12 and the combinatorial QAP task are compared.

KLÍČOVÁ SLOVA

Metaheuristiky, GPU akcelerace, CUDA, ISPC, HC12, diferenciální evoluce, QAP, SAT, HPC

KEYWORDS

Metaheuristics, GPU acceleration, CUDA, ISPC, HC12, differential evolution, QAP, SAT, HPC

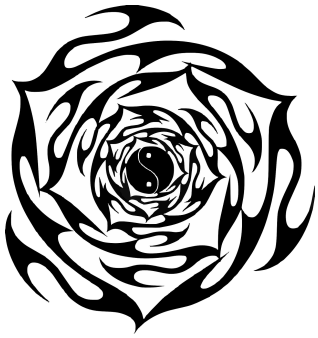
BIBLIOGRAFICKÁ CITACE

DOBROVSKÝ, Ladislav. *GPU akcelerované metaheuristiky ve vybraných úlohách globální a kombinatorické optimalizace*. Brno, 2024. Dostupné také z: https://www.vut.cz/studenti/zav-prace/detail/154877. Disertační práce. Vysoké učení technické v Brně, Fakulta strojního inženýrství, Ústav automatizace a informatiky. Školitel prof. Ing. Radomil Matoušek, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že disertační práci „GPU akcelerované metaheuristiky ve vybraných úlohách globální a kombinatorické optimalizace“ jsem vypracoval samostatně, za odborného vedení školitele. Dále prohlašuji, že jsem řádně citoval všechny použité prameny a literaturu.

V Brně, dne 29. 2. 2024 Ing. Ladislav Dobrovský



PODĚKOVÁNÍ

Tímto bych rád poděkoval svému školiteli prof. Ing. Radomilu Matouškovi, PhD, rodině, přátelům a kolegům za podporu při studiu a vypracování této disertační práce.

IF, AR, RA, MK, ZK, JR, VR, HP, PŠ, TH, TH, TN, MD, ML, JK, AH, LR, RH, JV, ...

OBSAH

1 ÚVOD

Život na Zemi je pro člověka udržováním určité rovnováhy s prostředím i rovnováhy vnitřní. Dalo by se říci, že je plný hledání řešení a kompromisů , hledáním východisek z konfliktních situací a plnění protichůdných požadavků. Určitým stínem či odrazem toho je v technické praxi hledání řešení technických problémů, které vyvstávají jak z individuálních lidských potřeb fyzických i duševních, tak z potřeb společnosti jako celku.

Optimalizace je v širším kontextu vylepšování. Předpokládá tedy možnost kvalitativního hodnocení a následného seřazení řešení od horšího k lepšímu. Stanovení důležitosti dílčích požadavků je samo o sobě kompromisem. V užším kontextu matematické optimalizace jde o hledání globálních extrémů funkcí. Tato práce se věnuje problémům z pohledu tzv. inženýrské optimalizace, což často představuje hledání dostatečně dobrých řešení v rozumném čase a za rozumnou cenu (feasible solutions in feasible time at feasible cost).

Práce je rozdělena na teoretickou a praktickou část. Praktická část představuje základní výsledek a zahrnuje implementaci zvolených metaheuristik s orientací na datově paralelní výpočty, zvolených úloh globální spojité i kombinatorické optimalizace (taktéž s vhodnými datově paralelními implementacemi) a provedení experimentů s vyhodnocením, včetně experimentů porovnávající zvolené hardwarové platformy.

Teoretická část uvádí základní pojmy a doplňuje popis praktické částí. Jsou zde uvedeny principy jednotlivých metaheuristických algoritmů, formální definice a vlastnosti zvolených úloh globální spojité i kombinatorické optimalizace, přehled hardwarových platforem dostupných v současnosti pro vysoce náročné výpočty (HPC). Také je zde popis přístupů k programování více jádrových centrálních procesorových jednotek (multi-core CPU), a také grafických procesorových jednotek s podporou obecných výpočtů (GPGPU). Vše je podložené relevantními zdroji literatury.

Mezi hlavní autorovy cíle pro tuto práci patří praktické využití hardware vhodného pro vysoce náročné výpočty (HPC), primárně GPGPU, zkoumání variant více populační diferenciálních evoluce a efektivní řešení úlohy QAP.

Kapitola 2 se věnuje přehledu optimalizačních metaheuristiky. Po stručném přehledu metod je do hloubky popsáno fungování algoritmů HC12, GAHC a diferenciální evoluce (DE). Metoda DE je zde v klasické variantě i s více ostrovními populacemi. Popsány jsou různé přístupy k výpočtu genetických operátorů mutace, křížení a selekce. Jsou také zmíněny strategie korekce nevhodných řešení (SDIS). U varianty s více populacemi jsou popsány různé strategie migrace jedinců mezi ostrovy.

Jak jsou definovány problémy globální spojitá optimalizace, která hledá vhodné hodnoty pro argumenty funkcí reálných proměnných je popsáno v kapitole 3 a celá sada testovacích funkcí pak v příloze A. Také je v této kapitole definována kombinatorická optimalizace, která se pokouší najít nejvhodnější kombinaci, permutaci či variaci. Konkrétními úlohami k řešení je problém kvadratického přiřazení (QAP) a problém splnitelnosti Booleovské formule (SAT, satisfiability problem).

Přehledem a popisem dostupných HW platforem pro potřeby vysoce náročných výpočtů (HPC) se zabývá kapitola 4.1. Jsou zde zmíněny jak komplexní řešení (superpočítače, grid, cloud), tak jednotlivé součástky, ze kterých se skládají zařízení od mobilních zařízení

po cloud (CPU, GPGPU, FPGA, ASIC). Konkrétní rozdíly v architektuře a preferovaných přístupech pro programování více jádrových CPU (kapitola 4.3, programovací jazyky C++ a ISPC) a GPGPU (kapitola 4.2, softwarové platformy CUDA a SYCL). Kapitole 4.3.1 je zajímavou zmínkou o stavu standardizace zápisu datově paralelních algoritmů v programovacím jazyce C++ v normě ISO/IEC 14882.

Popisem datově paralelních implementací jednotlivých metaheuristických algoritmů a optimalizačních úloh (16 spojitých funkcí, QAP a SAT) se zabývá kapitola 5. Metaheuristika HC12 je rozšířena o seznam zakázaných řešení, tzv. Tabu list, v implementaci vhodné pro GPGPU.

Diferenciální evoluce s více ostrovními populacemi je v unikátní implementací, která velice výhodně řeší problémy dostupných GPGPU implementací z pohledu aplikace genetických operátorů mutace a křížení. Po efektivní transformaci dat zachovává uspořádání pro vhodnou paralelizaci účelové funkce na GPGPU.

Text je doplněn vhodnými pseudokódy a výňatky zdrojového kódu v programovacích jazycích C++, CUDA C, Python a Matlab (m-kód). Také jsou podrobněji popsány relevantní části veřejného programového rozhraní (*public API*) a příklady uživatelem definovaných účelových funkcí pro použití s implementací řešičů, které metaheuristiky prakticky zpřístupňují. Pro programování GPGPU je použita platforma NVIDIA CUDA. V podkapitole 5.3.2.1 je také jako *benchmark* implementovaná metoda HC12 s úlohou QAP v pěti různých verzích. Čtyři pro více jádrová CPU s pomocí C++ a ISPC. Poslední, pátá, je upravená GPU verze pro co nejpřímější srovnání výkonu platforem.

Provedenými experimenty se zabývá text kapitoly 6. Nejprve je verifikována funkce implementace DE na příkladu analýzy vlivu volby strategie korekce nevhodných řešení SDIS a koeficientů *F*. Pak se u DE zkoumá vliv periody migrace u jednotlivých migračních strategií. Dalším experimentem je porovnání efektivity pěti různých implementací HC12 pro QAP na různých HW platformách CPU a GPGPU. Dva experimenty v podkapitole 6.3 ověřují úspěšnost implementace HC12 pro QAP. Posledním experimentem je zjišťování rozdílů mezi implementacemi (Python+NumPy, Matlab, CUDA C) testovací sady funkcí spojité optimalizace.

2 OPTIMALIZAČNÍ METAHEURISTIKY

Optimalizační metaheuristiky jsou algoritmy, které prohledávají diskrétní (nebo diskretizovaný) stavový prostor parametrů účelové funkce. Takových strategií existuje značné množství, dělí se hlavně na lokální a globální prohledávání. Liší se tím, zda se pokouší nalézt globální extrém účelové funkce nebo z principu zůstávají v blízkém lokálním extrému.

Všechny optimalizační algoritmy pracují s konceptem účelové funkce (objective function), v algoritmech inspirovaných evoluční teorií (*bio-inspired* algoritmy) je označovaná jako *fitness* funkce, či zkráceně *fitness*. Poznamenejme, že v biologickém kontextu je tato funkce definovaná jako nezáporná, tedy určená k maximalizaci. Parametry těchto funkcí mohou být obecně reálná čísla, grafové struktury, texty, matice, tenzory; jakákoliv kódovatelná data. Jejich výsledkem musí být datový typ, nad kterým je definována operace porovnání, aby šlo určit, které řešení zobrazuje účelová funkce na hodnotu interpretovanou jako lepší.

Parametry účelové funkce jsou nejčastěji:

- binární řetězce,
- celá čísla (mohou reprezentovat graf, text, obraz, atd.)
- a racionální čísla (funkce s reálnými argumenty).

Binární řetězce bývají kódovány buď přímo nebo v Grayově kódu. Celá čísla jsou v případě potřeby záporných hodnot kódována ve dvojkovém doplňku nebo kódu posunuté nuly. Mohou kódovat i jiná data, např. stromové struktury pro Genetické Programování (GP) nebo *codony* aplikace pravidel bezkontextové gramatiky pro Gramatickou Evoluci (GE), či se mohou přepočítávat do intervalu racionálních čísel.

Racionální číslo může být určeno [1, 2, 3, 4]:

- jako celé číslo, které se v průběhu výpočtu funkce přepočítává na daný interval,
- dvojicí celých čísel jako racionální zlomek,
- jako polynom součtu racionálních zlomků,
- jako desetinné nebo dvojkové necelé číslo s určenou či neomezenou přesností (například: GNU Multiple Precision Arithmetic Library pro jazyk C, modul decimal v Pythonu, Variable precision arithmetic v Matlabu),
- nebo jako číslo s plovoucí desetinnou čárkou (nejčastěji IEEE-754 single/double precision floating point nebo novější méně univerzální formát bfloat16 - brain float, či jiné formáty: NVIDIA TensorFloat, AMD fp24, Pixar PXR24).

Často se mluví o funkcích s reálnými argumenty. Žádná z těchto reprezentací však nepokrývá dokonale ani omezený rozsah reálných čísel. Dochází tak ke zjednodušení a určitému předpokladu, že se v praktickém užití nedostatek přesnosti a zaokrouhlování neprojeví negativně.

2.1 Přehled vybraných algoritmů

Od roku 1975, kdy byly představeny Genetické algoritmy (GA), byly vyvinuty stovky variant různých metod inspirovaných evolucí, biologicky či obecně přírodou. Taxonomie přírodních a populačních algoritmů uvedená v [5] jich zařazuje 323.

Klasické

Metoda náhodné procházky (random walk) – inspirována brownovým pohybem, prochází prostor náhodnými přírůstky. Přestože může působit naivně, stále nachází aplikace [6].

Metoda horolezce (hillclimb) – jedná se o hladový (greedy) algoritmus upřednostňující vždy nejlepší řešení z okolí současného bodu. Uniknout z lokálního extrému je možné pouze restartem algoritmu.

Simulované žíhání – modifikuje horolezeckou metodu o možnost přijetí horšího řešení a tedy potenciálně možnost opustit lokální minimum. Inspirace pochází z termodynamiky, kdy při vyšší teplotě mají vyšší pravděpodobnost i energeticky vyšší stavy. Jak teplota klesá, snižuje se pravděpodobnost přijetí horšího řešení až k nule [7].

Metoda Nelder-Mead – pro n-rozměrný problém je definován *simplex*, geometrický útvar s $(n+1)$ vrcholy, které jsou náhodně inicializované. Jsou definovány operace pro jeho pohyb prostorem ve snaze zlepšit ohodnocení nejhoršího vrcholu. Byly vyvinuty i paralelní implementace vhodné pro urychlení výpočtu mnohodimenzionálních úloh [9]. Metoda neklade na funkci žádné nároky na spojitost a diferencovatelnost.

Evoluční a populační [8]

Genetické algoritmy (GA) – inspirovány evoluční teorií provádí nad populací jedinců operace mutace (mírná změna jedinců), křížení (kombinace jedinců) a selekce (přežití jedinců do další generace).

Genetické programování (GP) – rozšiřuje GA o pravidla interpretace jedinců jako stromových či grafových struktur, které reprezentují například výrazy, programy nebo obvody (hlavně Kartézské genetické programování – CGP).

Gramatická evoluce (GE) – je podobná GP, ale využívá při převodu genotypu na fenotyp bezkontextové gramatiky.

Optimalizace hejnem částic (Particle Swarm Optimization – PSO) – inspirována chováním hejna ptáků, každá částice má svůj pohyb ovlivňovaný vlastní rychlostí a polohou ostatních částic. Protože mnoho druhů živočichů žije ve skupinách, existuje množství algoritmů inspirovaných chováním konkrétního druhu (včely, světlušky, bizoni, velryby, atd.).

Diferenciální evoluce (DE) – je inspirována genetickými algoritmy s tím rozdílem, že jedinci jsou přímo n-rozměrné vektory reálných čísel a operace mutace, křížení a selekce jsou adekvátně modifikovány. Tato práce se zabývá modifikací DE a je dále podrobněji popsána.

2.2 Algoritmy HC12 a GAHC

Oba algoritmy byly vyvinuté doc. Matouškem na VUT v Brně [10, 11, 12]. První algoritmus, HC12, je variantou horolezeckého algoritmu nad binárními řetězci. Jeho relativní zvláštností je využití XORu binárních masek a Grayova kódování pro efektivnější generování bodů okolí. Téměř všechny operace lze velmi dobře provádět paralelně. Druhý algoritmus, GAHC, je hybridní modifikace genetického algoritmu, který integruje HC12 jako mutační operátor náhodně zvolených podřetězců v rámci mutace jedince.

Pojmenování HC12 vychází z Hill Climb a Hammingových vzdáleností matic binárních masek, konkrétně M1 a M2, viz dále. Existují také logicky různé varianty, například HC123, HC13, HC12r3. Přidávání dalších masek však výrazně zvyšuje nároky na paměť a výpočetní výkon počítače. Pro matici masek M*h* s Hammingovou vzdáleností *h* je počet řádků, tedy počet potenciálních řešení pro *n* bitů, kombinační číslo *n* nad *h*.

2.2.1 Historie implementací HC12

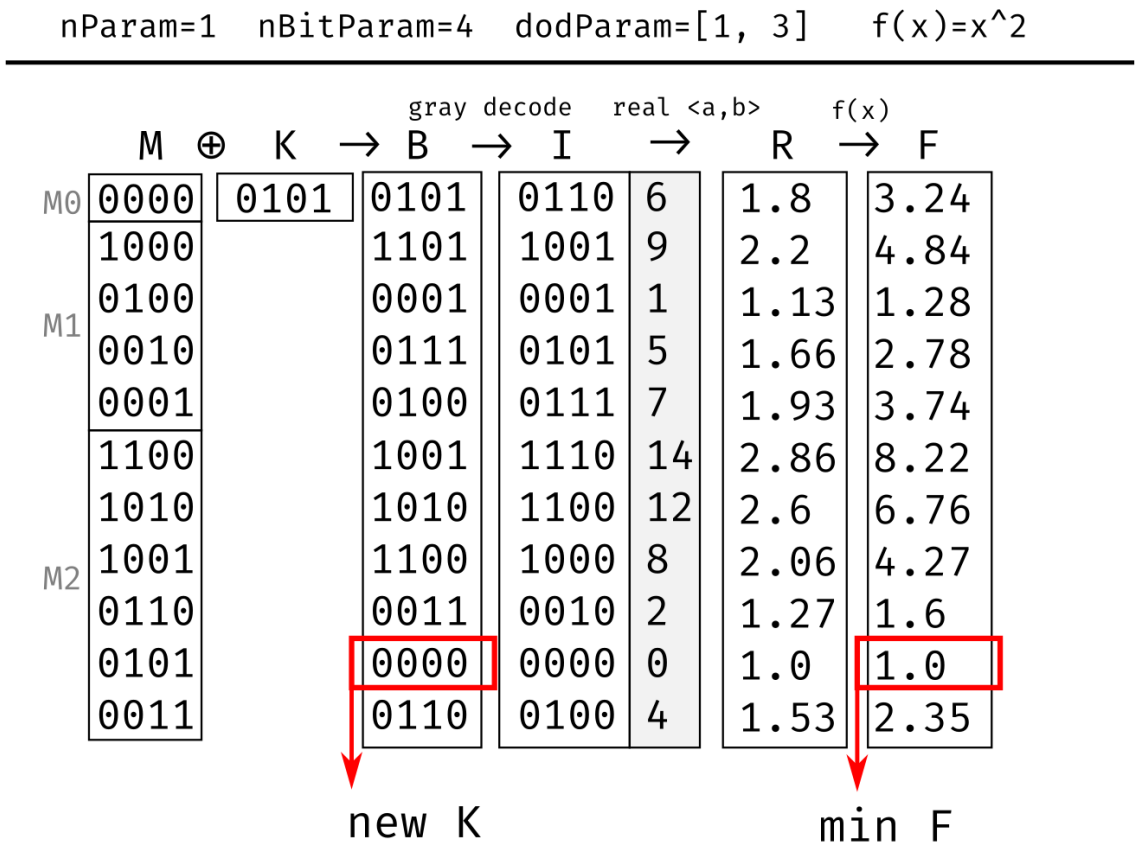
První implementace algoritmu HC12 využívaly čistě prostředí Matlabu. Prototypy se soustředily na snadnost analýzy výsledků a ladění funkčnosti algoritmu. Po ověření správnosti směru vývoje se začala zlepšovat postupně efektivnost a výkon implementace. Kolem roku 2009 stoupal význam obecných výpočtů na GPGPU firmy NVIDIA a nastupující softwarové platformy CUDA (první veřejná verze 2007) pro jejich programování. Algoritmus HC12 se dočkal první implementace na GPU [13] pomocí platformy CUDA.

Pokračovala optimalizace rychlosti výpočtu i na CPU s využitím jazyka C, kdy se některé funkce implementovaly jako MEX moduly (plug-in dynamické knihovny s daným rozhraním a linkované s knihovnami Matlabu, aby bylo jejich použití při nahrazení m-kódu transparentní). Tato implementace byla oproti čistě GPU kódu preferovaná kvůli jednoduššímu ladění a testování modifikací algoritmu.

Pro řešení kombinatorické úlohy QAP se postupně na CPU a GPU implementovala účelová funkce, nejprve jako modulární rozšíření původního algoritmu. Po té byl QAP integrován přímo do jádra varianty algoritmu HC12 (řešič *hc12qapswap2*), což vedlo ke zmenšení množství a režie volání CUDA kernelů a umožnilo vynechat některé parametry nepodstatné pro tuto úlohu s lepším využitím paměti konstant. Byly také prováděny pokusy o využití CUDA dynamického paralelismu, to však nepřineslo zlepšení výkonu a výrazně zhoršilo možnosti ladění algoritmu na dostupných kartách GeForce spolu se snížením odezvy operačního systému pokud tato karta obsluhovala i zobrazovací jednotku.

2.2.2 Princip algoritmu HC12

Základní varianta pro problémy s binárními řetězci (například problém batohu), celočíselnými (integer) nebo racionálními (IEEE-754 floating point) argumenty účelové funkce se kóduje podle následujícího obrázku 1.



Obr. 1: Schéma iterace algoritmu HC12

S významem parametrů, proměnných a matic:

- *nParam*: počet parametrů účelové funkce.
- *nBitParam*: počet bitů kódujících jeden parametr (může se pro každý parametr lišit).
- *dodParam*: intervaly <*a*, *b*>, na které se rovnoměrně pro každý parametr funkce rovnoměrně zobrazují hodnoty *I*.
- *f*(***x***): účelová funkce.
- **M**: matice masek, složená z pod-matic M0, M1 a M2, které obsahují hodnoty binárních řetězců s Hammingovými vzdálenostmi 0, 1 a 2.
- **K**: binární řetězec k transformaci (počáteční nebo vybrán v předchozí iteraci).
- **B**: matice binárních řetězců vzniklých aplikací operace XOR mezi řádky matice M a binárním řetězcem *K*.
- **I**: matice nezáporných (bez znaménka, unsigned) celých čísel vzniklá z částí binárních řetězců matice B převedených z Grayova kódu na přímý binární kód. Převod je uskutečňován pro každý parametr jednotlivě; varianta s převodem binárního řetězce jako celky byla také v průběhu testována, ale neposkytla výrazné změny výsledků za cenu horšího výkonu.

- **R**: matice racionálních (IEEE-754 floating point) argumentů pro výpočet hodnoty účelové funkce; jednotlivé parametry jsou na řádku.
- **F**: sloupcový vektor výsledků účelové funkce.

V každé iteraci se na řádku obsahujícím minimum vektoru F vybírá z matice **B** nová hodnota binárního řetězce *K* pro další iteraci. Pokud je nejmenší hodnota F v prvním řádku, algoritmus se ukončí.

V upravené verzi pro QAP problémy se do paměti ukládají pouze matice **M**, **B** a vektor F. Matice **I** je pouze jako mezivýsledek; stačí držet v lokálních proměnných pouze dvě poslední hodnoty a provést s nimi výměnu (*swap*) k postupné úpravě permutace., lze ji uložit úsporněji pomocí dvojic (číslo parametru, číslo bitu). U běžné varianty HC12 je k dispozici nastavení, které umožňuje zvolit, zda se má matice **I** ukládat a účelová funkce ji bude využívat, stejně tak lze zvolit, zda se má vytvářet matice **R**.

Matici **M** je nutné zkonstruovat částečně lineárně, protože není známa funkce, která by přiřadila přímo podle čísla řádku nastavené bity. Jakmile jsou však známy pozice bitů, nastavují se na GPU paralelně pro každý řádek. Pokud je matice **M** již velmi řídká lze ji takto rekonstruovat v průběhu výpočtu bez ztráty výkonu a ušetřit tak paměť pro data účelové funkce

Matici **M** není nezbytně nutné ukládat a lze ji rekonstruovat při zachování stejného výkonu. Přestože to vede k divergenci v rámci skupiny vláken, se zlepšujícím se HW je postih zanedbatelný.

2.2.3 GAHC

Genetický algoritmus (GA) je dnes již klasická metaheuristika [14] primárně vyvinutá pro účelové funkce nad binárními řetězci s následným překódováním. GA se inspiruje principy evoluční teorie a její fyzickou manifestací v nukleových kyselinách genotypu a jejich následném projevu jako fenotyp organismů. Základní kroky algoritmu jsou selekce (výběr jedinců do další generace/iterace), křížení (kombinace genotypu dvojice či více jedinců), mutace (náhodná úprava genotypu přeživších jedinců). Některé implementace sledují počet iterací, po které jedinec existuje a zavádí koncept stárnutí, jedinci po limitu “umírají” tj. jsou odstraňováni z populace [15]. Pokud varianta algoritmu nezaručuje nekonečné přežití nejlepšího jedince (některé varianty selekce a stárnutí), je třeba toto řešení vždy uschovat do paměti v kopii, aby algoritmus na konci neprohlásil horší řešení za finální výsledek.

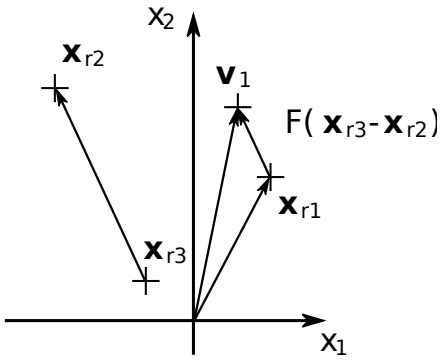
GAHC je využití algoritmu HC12 (a dalších variant) jako (přídavného) mutačního operátoru, který náhodnou část binárního řetězce genomu optimalizuje. Při zastavení konvergence genetického algoritmu lze provést ještě dodatečně HC12 pro celý řetězec a ověřit tak pozici lokálního extrému v rámci dostupného okolí [16].

2.3 Diferenciální evoluce (DE)

Diferenciální evoluce (DE) je algoritmus podobný genetickému algoritmu. Představen byl v roce 1995 v technické zprávě. Upravený text byl vydán v časopise v roce 1997. [17, 18]. První dvě použití byly pro hledání 18 parametrů IIR-filtru a jako účastník soutěže ICEC'96 v hledání extrémů desítky testovacích funkcí s reálnými (racionálními) argumenty. [19, 20]

Na rozdíl od genetického algoritmu nepracuje diferenciální evoluce s binárními řetězci převedenými na racionální čísla, ale s racionálními čísly přímo. Dalším rozdílem je, že operátory mutace a křížení jsou definovány jinak. Využívají tři a více jedinců (vektorů) tak, že k prvnímu je přičten v některých složkách rozdíl, diference; odtud název diferenciální evoluce. Vektor složený pouze z těchto diferencí je mutační. Vektory vzniklé křížením, tj. výběrem složek buď původního nebo mutačního vektoru, se nazývají *trial* vektory. Původní vektor v populaci nahrazují pouze při zlepšení hodnoty fitness funkce (selekce).

Obrázek 2 ilustruje jak mutační vektor $\mathbf{v}_i$ vzniká. Z populace jsou náhodně zvoleny tři různé vektory $\mathbf{x}_{r1}$, $\mathbf{x}_{r2}$ a $\mathbf{x}_{r3}$ tak, že k vektoru $\mathbf{x}_{r1}$ je přičten rozdíl vektorů $\mathbf{x}_{r2}$ a $\mathbf{x}_{r3}$ násobený koeficientem F. *Trial* vektor $\mathbf{y}_i$ (často v literatuře značeny jako $\boldsymbol{u}_i$) vzniká z původního vektoru $\mathbf{x}_i$ a z mutačního vektoru $\mathbf{v}$ operátorem křížení. Zda ve složce vektoru dojde ke křížení závisí na pravděpodobnosti křížení dané koeficientem *CR*. Některé varianty DE vybírají také náhodně jednu složku, která se nahradí vždy, nezávisle na výsledku porovnání s *CR*. To zaručí, aby se vždy *trial* vektor odlišoval od vektoru původního alespoň v jedné složce.



Obr. 2: Vznik mutačního vektoru, DE/rand/1 (2D)

Tato základní mutační strategie se značí DE/rand/1. Mutační strategie se liší počtem použitých diferencí a volbou použitých vektorů (původní $\mathbf{x}_i$, náhodný $\mathbf{x}_{rk}$, nejlepší $\mathbf{x}_{best}$). Jsou klasifikovány dle [21]:

- DE/rand/1

$$\boldsymbol{v}_i = \boldsymbol{x}_{r1} + F(\boldsymbol{x}_{r2} - \boldsymbol{x}_{r3}) \tag{1}$$

- DE/rand/2

$$\boldsymbol{v}_i = \boldsymbol{x}_{r1} + F(\boldsymbol{x}_{r2} - \boldsymbol{x}_{r3}) + F(\boldsymbol{x}_{r4} - \boldsymbol{x}_{r5}) \tag{2}$$

- DE/best/1

$$\boldsymbol{v}_i = \boldsymbol{x}_{best} + F\left(\boldsymbol{x}_{r1} - \boldsymbol{x}_{r2}\right) \tag{3}$$

- DE/best/2

$$\boldsymbol{v}_i = \boldsymbol{x}_{best} + F\left(\boldsymbol{x}_{r1} - \boldsymbol{x}_{r2}\right) + F\left(\boldsymbol{x}_{r3} - \boldsymbol{x}_{r4}\right) \tag{4}$$

- DE/current-to-rand/1

$$\boldsymbol{v}_i = \boldsymbol{x}_i + F\left(\boldsymbol{x}_{r1} - \boldsymbol{x}_i\right) + F\left(\boldsymbol{x}_{r2} - \boldsymbol{x}_{r3}\right) \tag{5}$$

- DE/current-to-best/1

$$\boldsymbol{v}_i = \boldsymbol{x}_i + F\left(\boldsymbol{x}_{best} - \boldsymbol{x}_i\right) + F\left(\boldsymbol{x}_{r1} - \boldsymbol{x}_{r2}\right) \tag{6}$$

Většina funkcí je optimalizována s omezením hodnot definičního oboru, tedy pro každou složku x_{ij} je definován interval $\langle a_j; b_j \rangle$, do kterého se musí hodnota po mutaci zobrazit. Pokud hodnota v_{ij} neleží v tomto intervalu, je hodnota opravena, zde značena jako $\mathring{v}_{ij}$.

Několikrát použitá funkce fmod(*x*, *y*) odpovídá definici ve standardní knihovně jazyků C/C++ [23]. Zjednodušeně je „zbytkem po dělení reálných čísel“ neboli hledáním hodnoty (*x – ny)* takového, že je v absolutní hodnotě menší než *y* a znaménko je shodné s *x*.

Existuje několik těchto opravných zobrazení klasifikovaných dle [22]. Jsou nazývány jako *Strategy of dealing with infeasible solutions – SDIS*:

- *saturation (sat)* – „nasycení“ na porušené hranici:

$$\mathring{v}_{ij} = \begin{cases} a_j & ; v_{ij} < a_j \\ b_j & ; v_{ij} > b_j \end{cases} \tag{7}$$

- *mirror (mir)* – „zrcadlení“ na porušené hranici, předpokládá se, že není vzdálenost od hranice větší než šířka intervalu $\langle a_j; b_j \rangle$. Definice byla rozšířena, že pokud větší je, provede se *saturation*:

$$\mathring{v}_{ij} = \begin{cases} 2a_j - v_{ij} & ; v_{ij} < a_j \ \wedge \ a_j - v_{ij} < b_j - a_j \\ 2b_j - v_{ij} & ; v_{ij} > b_j \ \wedge \ v_{ij} - b_j < b_j - a_j \\ a_j & ; v_{ij} < a_j \ \wedge \ a_j - v_{ij} > b_j - a_j \\ b_j & ; v_{ij} > b_j \ \wedge \ v_{ij} - b_j > b_j - a_j \end{cases} \tag{8}$$

- *toroidal (tor)* – kraje intervalu se „spojí do kruhu“, u funkce dvou parametrů odpovídá pohybu po povrchu toroidu:

$$\mathring{v}_{ij} = \mathrm{fmod}\left(v_{ij} - a_j, b_j - a_j\right) + a_j \tag{9}$$

- *halfway-to-violated-bounds (HVB)* – „na půl cesty“ k porušené hranici. Využívá současné (z předchozí generace) řešení x_{ij} ke kterému se mutace v_{ij} generuje:

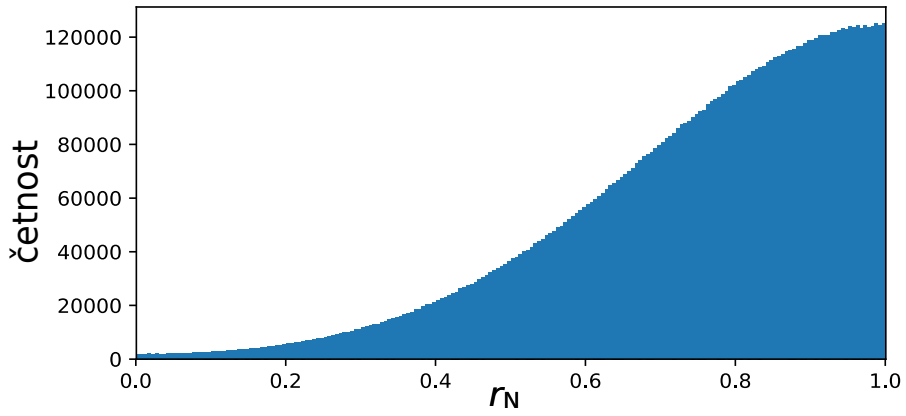
$$\mathring{v}_{ij} = \begin{cases} x_{ij} + \dfrac{x_{ij} - a_j}{2} & ; v_{ij} < a_j \\ x_{ij} + \dfrac{b_j - x_{ij}}{2} & ; v_{ij} > b_j \end{cases} \tag{10}$$

- *uniform* (*uni*) – $\mathring{v}_{ij}$ je náhodně zvoleno z intervalu $\langle a_j; b_j \rangle$ s rovnoměrným rozložením pravděpodobnosti.
- *complete one-sided truncated normal* (*COTN*) – použité pseudonáhodné číslo r_N má funkci hustoty pravděpodobnosti přibližně (11) v intervalu <0; 1>. Prakticky není třeba zjišťovat hodnotu koeficientu c a při generování r_N je použito pseudonáhodné číslo r_{N0} z normální rozdělení $N(0; 0{,}32^2)$ a korekce do intervalu <0; 1> dle (12). Hodnota σ byla volena tak, aby $f(r_N)$ odpovídala přibližně grafu (a) na Figure 1 v [22]. Přesné parametry normálního rozdělení zde ani v citovaných zdrojích nebyly uvedeny. Histogram generovaných pseudonáhodných čísel je na obrázku 3.

$$f(r_N) = 2\,\mathrm{N}(\mu; \sigma^2) + c \approx 2\,\mathrm{N}(0;\, 0{,}32^2) + c$$
$$\int_0^1 f(r_N)\,dx = 1 \tag{11}$$

$$r_N = \begin{cases} r_{N0} & ;r_{N0} \in \langle 0, 1 \rangle \\ |\mathrm{fmod}(r_{N0}, 1)| & ;r_{N0} < 0 \\ |\mathrm{fmod}(1 - r_{N0}, 1)| & ;r_{N0} > 1 \end{cases} \tag{12}$$

$$\mathring{v}_{ij} = \begin{cases} r_N(b_j - a_j) & ;v_{ij} < a_j \\ (1 - r_N)(b_j - a_j) & ;v_{ij} > b_j \end{cases} \tag{13}$$



Obr. 3: Histogram pseudonáhodných čísel pro SDIS COTN

Poznámka: Některé výzkumy ukazují, že diferenciální evoluci lze použít také pro řešení problémů kombinatorické optimalizace [24, 25].

2.3.1 Diferenciální evoluce s více ostrovními populacemi

Dosavadní implementace algoritmu diferenciální evoluce (DE) na CUDA [26, 27] neřeší některé problémy protichůdných požadavků efektivního výpočtu genetických operátorů na GPU. Také se zabývají faktem, že DE při přílišném zvýšení počtu jedinců v populaci má výrazně negativní vliv na rychlost konvergence [28]. Kompromisem je zavedení více populací, které se obrazně řečeno vyvíjejí samostatně na tzv. ostrovech (island). Každá ostrovní populace může mít jiné parametry a vyvíjí se samostatně. V obecném případě se

mohou populace lišit v počtech jedinců, koeficientech *F* a *CR*, mutaci i způsobu opravy nevyhovujících řešení (SDIS). Vždy po určitém počtu generací dojde k migraci, tj. přesunu (kopírování) části jedinců mezi jednotlivými ostrovy. V některé literatuře, například [29], je nazývána migrace jako propagace.

Základní migrační strategie jsou dle [29]:

- *One to one*, kdy se náhodně vyberou dvě populace a nahradí si vzájemně nejhoršího jedince za nejlepšího.
- *One to N*, kdy se náhodně vybere populace, která nabídne svého nejlepšího jedince všem ostatním populacím k nahrazení za jejich nejhoršího jedince.
- *N to one*, kdy se náhodně vybere jedna cílová populace, kam směřují své nejlepší jedince populace ostatních ostrovů.
- *N to N*, kdy všechny ostrovní populace informují ostatní populace o svém nejlepším jedinci. Kvůli množství komunikace není pro GPU implementaci nejvhodnější. V testech prováděných v [29] zvyšuje pravděpodobnost uváznutí v lokálním minimu, což je přisuzováno snížení diverzity populací.

Další dvě migrační strategie mohou být:

- *PermuteN*, kdy je každé populaci přiřazena jiná cílová populace, které předá nejlepšího jedince tak, že žádné dvě populace nemají stejný cíl.
- *RandTarget*, kdy si každá populace náhodně vybere jinou cílovou populaci, které předá svého nejlepšího jedince. Pokud je populace cílová pro více jedinců, dohodne se, který bude přijat.

Některé varianty DE také řeší pokud se někteří jedinci příliš přiblíží či ztotožní a provádějí operaci „odklonování“ (decloning). Toto rozšíření v kombinaci s ostrovními populacemi implementuje varianta IBDEA^{d} a její autoři tvrdí, že různě velké ostrovní populace mohou přispět k efektivnímu hledání řešení i bez migrace [30].

2.3.2 Současné varianty a aplikace DE

Pro diferenciální evoluci s jedinou populací jsou již delší dobu velmi populární adaptivní varianty. Ty v průběhu výpočtu upravují své chování dle historie hodnot účelové funkce a vlastností populace. Jedna z prvních adaptivních variant je FADE [31] a cituje jako přímou inspiraci algoritmus GAFIS, který je adaptivním genetickým algoritmem vyvinutým zde na VUT FSI ÚAI [32]. Novější literatura zmiňuje adaptivní varianty například L-SHADE [33] a ESADE [34].

Co se týče aplikací, autoři [35] zmiňují 55 článků vydaných v letech 2018 až 2020, které obsahují aplikace DE a hybridních algoritmů v oblastech: predikce, průmyslová automatizace a řízení, výpočetní systémy, elektrické a energetické systémy, výběr příznaků (feature selection), zpracování obrazu, shlukování (clustering), medicína, plánování cesty, bezdrátové přenosy a senzory, diferenciální rovnice. Příkladem jedné z nejnovějších aplikací DE v biochemii je [36].

3 OPTIMALIZAČNÍ ÚLOHY

3.1 Spojitá optimalizace

Spojitá optimalizace se zabývá hledáním globálních extrémů funkcí s reálnými (racionálními) argumenty. Mnoho technických problémů lze vyjádřit jako takovou funkci od hledání koeficientů polynomů po parametrickou simulaci nelineárních systémů. [37] Výsledný problém je pak formulován jako hledání globálního extrému funkce, minimalizace (14) nebo maximalizace (arg max), na definičním oboru D, který může být různě omezen pro jednotlivé parametry. Tato práce se zabývá metodami, které o funkci nepředpokládají diferencovatelnost ani spojitost na celém intervalu či po částech.

$$\arg\min_{\boldsymbol{x}\in D} f(\boldsymbol{x}) = \{\boldsymbol{x}\in D:\ f(\boldsymbol{x}) = \min_{\boldsymbol{y}\in D} f(\boldsymbol{y})\} \tag{14}$$

Aby bylo možné porovnávat robustnost jednotlivých metod, byly různými autory definovány vhodné testovací funkce. Při jejich návrhu byla snaha o dosažení různých vlastností:

- Obecný počet dimenzí
  vzorce obsahují “agregační” funkce jako je suma a produkt pro obecný počet dimenzí, u některých funkcí došlo k zobecnění dodatečně.
- Známost polohy optima
  jsou voleny podfunkce se známými globálními extrémy, které celé funkci dávají určitý ráz, například x_i^2 zaručí grafu funkce tvar paraboly (paraboloidu, hyperparaboloidu), který lze modifikovat dalšími funkcemi, aby byl povrch vhodně pokřivený.
- Vysoká hustota lokálních extrémů
  například Rastriginova F6 funkce tohoto dosahuje pomocí goniometrické funkce kosinus namodulované na parabolu a aplikované ve všech dimenzích.
- Nediferencovatelnost
  příkladem je funkce absolutní hodnoty; složitějším případem je Weierstrassova funkce: jde o aproximaci Fourierovy řady, která konverguje k hodnotám funkce spojité a zároveň nediferencovatelné v každém bodě, ovšem pro reálný výpočet je omezena na konečný součet.
- Nespojitost
  příkladem jsou funkce signum a zaokrouhlení
- Konstantní oblasti a oblasti se zanedbatelnou změnou
  představují pro některé metody velkou výzvu, například Easomova funkce; v kontextu problematiky nastavení váhových koeficientů umělé neuronové sítě odpovídá výstupu aktivační funkce ReLU (Rectified Linear Unit, rectifier) pro záporný vstup.
- Lineární neseparabilita
  pokud je funkce lineárně separabilní, tj. lze optimalizovat zvlášť v jednotlivých dimenzích, některé metody jsou zvýhodněny; neseparabilitu lze zajistit přidáním transformací celého prostoru (translace a rotace), pro více dimenzí lze využít Gramovu-Schmidtovu ortogonalizaci.

- Posunutí optima ze středu soustavy
  mnoho testovacích optimalizačních funkcí má globální extrém ve středu soustavy souřadnic, tj. v bodě [0; 0; 0; ...]; některé metody mají v takovém případě lepší výchozí pozici a zdají se efektivnější, než je pak jejich chování v obecném případě.

Pro testování byly vybrány funkce F1 až F12 z The Genetic and Evolutionary Algorithm Toolbox for Matlab (GEATbx) [38] a funkce F4 až F10 z "The 100-Digit Challange" [39], označované dále jako CEC04 až CEC10. A to v základní verzi pro více dimenzí bez transformací a s transformacemi (škálování, rotace a posun) ve 2 a 10 dimenzích, zde značené jako CEC04t až CEC10t. Transformace zajišťují, aby nebyly funkce lineárně separabilní, čímž jsou rovněž vhodnější pro porovnání robustnosti řešících metod. Upřesnění vzorců některých funkcí, například F11, bylo čerpáno také z Virtual Library of Simulation Experiments [40].

Například funkce „CEC08 Expanded Schaffer's F6 Function":

$$f_{CEC\,08}(\boldsymbol{x}) = g(x_1, x_D) + \sum_{i=1}^{D-1} g(x_i, x_{i+1})$$
$$g(x, y) = \frac{1}{2} + \frac{\sin^2\sqrt{x^2 + y^2} - \frac{1}{2}}{1 + \frac{x^2 + y^2}{1000}} \qquad (15)$$
$$x_i \in \langle -100\,;100 \rangle$$

Vzorce (21) až (36) funkcí s definičními obory jsou s uvedeny v **příloze A**.

Transformované CEC04t až CEC10t jsou definovány pro $x_i \in$ <-100; 100>, každá s různým vektorem posunutí $\boldsymbol{o}$, koeficientem škálování s a rotačními maticemi $\mathbf{M}_{2D}$ a $\mathbf{M}_{10D}$, aplikovány jako:

$$f_{CECxxt}(\boldsymbol{x}) = f_{CECxx}(\boldsymbol{M}(s(\boldsymbol{x} - \boldsymbol{o}))) \qquad (16)$$

Metoda HC12 vyžaduje diskretizaci hodnot argumentů na omezeném intervalu. Interval lze pak dále zmenšovat a dosáhnout tak postupně jemnějšího pokrytí. Naopak metoda diferenciální evoluce pracuje přímo s reálnými čísly (racionálními, s plovoucí desetinnou čárkou) a může fungovat na neomezeném intervalu. DE trpí opačným problémem, kdy je třeba zajistit, aby hodnoty argumentů nově navrhovaných řešení ležely v definičním oboru.

3.2 Kombinatorická optimalizace

Kombinatorická optimalizace se zabývá řešením problémů, u kterých mohou být kombinace, variace či permutace vstupem optimalizované funkce, tj. existuje vhodné zobrazení n-prvkové relace do množiny s definovanou operací porovnávání (většinou reálná/racionální/celá čísla). Při vhodném kódování se některé metaheuristiky, navržené pro nespojité či

nediferencovatelné funkce, ukazují jako vhodné nástroje pro získávání dostatečně dobrých řešení.

3.2.1 Problém kvadratického přiřazení (QAP)

Quadratic Assignment Problem (QAP) patří mezi klasické NP-těžké (NP-*hard*) problémy a je příbuzný problému obchodního cestujícího (TSP). Jeho použití v základní variantě jako *facility placement* metody při již známých možných umístěních (*places*) jednotlivých *facilities* nachází využití v logistice a technice [41]. Mezi umístěními je definována matice vzdáleností D a mezi *facilities* je definována matice toků F. Samotným problémem je nalezení permutace π, která by přiřadila *facilities* na místa tak, aby byl minimální součet všech součinů vzdáleností a toku (17). [42]

Následně je tedy QAP problém:

$$\min_{\pi \in S_n} \sum_{i=1}^{n} \sum_{j=1}^{n} D_{\pi(i)\pi(j)} F_{ij} \tag{17}$$

kde S_n je množina všech permutací množiny {1, 2, 3, …, n}. Pokud jsou matice D a F symetrické podle hlavní diagonály, jedná se o symetrický QAP (18) a lze jej vyjádřit pomocí sdružené matice T, kde jsou nad hlavní diagonálou prvky matice F a pod hlavní diagonálou prvky matice D.

$$\begin{gathered} \min_{\pi \in S_n} \sum_{i=1}^{n} \sum_{j=i+1}^{n} 2 T_{\pi(\max(i,j))\pi(\min(i,j))} T_{ij} \\ T_{ij} = \begin{cases} F_{ij} & pro\ i > j \\ D_{ij} & pro\ i < j \end{cases} \end{gathered} \tag{18}$$

Pro testování stochastických i exaktních řešičů QAP existuje knihovna problémů QAPLIB [43]. U některých instancí stále není ověřeno exaktně nejlepší řešení a je zde uvedena vzdálenost (GAP) od odhadu spodní meze.

Mezi základní metody odhadu spodní meze patří Gilmore-Lawler bound. Vychází z myšlenky, že pokud se prvky matice D seřadí vzestupně a prvky matice F sestupně, musí být součet násobků mezi nimi nejmenším možným pro tato čísla. Nebývá však možné pro původní matice sestavit permutaci, která by k této sumě dospěla.

Mezi úspěšné heuristiky, které v minulosti dosáhly nejlepších řešení na problémech v QAPLIB patří simulované žíhání (SA) [44], genetické algoritmy (GA) [45], optimalizace mravenčí kolonií (ACO) [46] a robust tabu search (RoTS) [47].

Řešiče exaktní, které na mnoha problémech v QAPLIB ověřily optimalitu řešení dosaženého již dříve heuristikami, jsou založeny převážně na metodě větví a mezí (branch and bound) a její paralelizaci [48].

3.2.2 Problém splnitelnosti Booleovské formule (SAT)

Tento klasický NP-úplný (NP-complete) problém je významný v teoretické i aplikované informatice. Při řešení obecného SAT problému (18) se hledá odpověď na jednoduchou otázku: „Pro jaké hodnoty logických literálů x_i je konjunkce klauzulí obsahujících tyto literály či jejich negace pravdivá?“.

Existují různé varianty SAT, například:

- 3SAT omezuje počet literálů v klauzuli na 3
- 1-in-3-SAT požaduje, aby v klauzuli byl pouze jeden pravdivý literál
- Horn-SAT používá Hornovy klauzule (formule tak reprezentují implikace)

Pokud některá klauzule obsahuje literál a zároveň jeho negaci, je triviální, že je instance problému nesplnitelná.

$$\begin{aligned} l_{po} &\in \{x_1, \neg x_1, x_2, \neg x_2, \dots,\} \\ q_i &= l_{i1} \vee l_{i2} \vee \dots \vee l_{io} \\ \boldsymbol{x} &= (x_1, x_2, \dots, x_n) \\ s(\boldsymbol{x}) &= q_1 \wedge q_2 \wedge \dots \wedge q_m \\ \mathrm{s}(\boldsymbol{x}) &= 1 \end{aligned} \tag{19}$$

Kde „1“ reprezentuje pravdivou hodnotu, q_i jsou klauzule, $\boldsymbol{x}$ je vektor literálů, l_{po} jsou literály x_o a jejich negace vyskytující se v klauzuli q_p. Funkce $s(\boldsymbol{x})$ je výsledek přiřazení konkrétních pravdivých či nepravdivých hodnot literálům a vyhodnocení konjunkce klauzulí. Ověřuje se zda je výsledek pravdivá hodnota.

K porovnání výkonu či vhodnosti metod pro řešení SAT problému je k dispozici knihovna instancí různých variant SATLIB [49].

4 GPU A HPC VÝPOČTY

Graphics Processing Unit (GPU) je označení datově paralelních procesorů historicky využívaných převážně pro vykreslování (rasterizaci) počítačové grafiky (Computer-generated imagery, CGI) v reálném čase. *General Purpose GPU* (GPGPU) je další evolucí těchto procesorů směrem k obecným datově paralelním výpočtům s podporou operací pro grafiku již nevyužívaných. Obecnější pojem *vysoce náročné výpočty* (High Performance Computing, HPC) jsou rychle rostoucím odvětvím aplikované informatiky (computer science). Poznamenejme, že stejně jako u termínu “big data” se i zde stále mění co je předmětem.

S růstem výkonu se původně HPC aplikace superpočítačů dostávají přes pracovní stanice až do kapes běžných uživatelů v podobě AI akcelerátorů integrovaných například na SoC mobilních telefonů, chytrých senzorů a bezpečnostních kamer (*edge computing*).

4.1 Prostředky HPC

Kromě klasických superpočítačů zahrnuje HPC cluster servery, grid computing i výkonné pracovní stanice vybavené více jádrovými procesory, GPU či jinými akcelerátory jako jsou přídavná zařízení založená na programovatelných hradlových polích (FPGA) a obvodech navržených přímo na zakázku (ASIC).

V souvislosti s tím, jak se dále nedaří příliš vylepšovat běh programu v jednom vlákně procesu (*single thread performance*), vzniká tlak na vývoj software s použitím více vláken, procesů či přímo jako distribuované systémy spolupracujících programů s využitím dalších akcelerátorů v hybridních architekturách. V případě *grid computingu* jde o formu distribuovaných výpočtů, kdy skupina různě umístěných výpočetních prostředků, často s velmi rozdílným HW – označeným jako výpočetní uzly (*computing node*), je spojena přes internet nebo privátní síť, aby společně pracovala na velkých výpočetních úlohách.

4.1.1 Superpočítače

Jedná se o úzce specializované a integrované distribuované systémy, které obsahují unifikované výpočetní uzly, propojené pomocí speciální sítě s vysokou rychlostí, případně jsou tyto uzly vybavené dalšími akcelerátory (GPU, FPGA, ASIC). Celý superpočítač je provozován jako celek se specializovaným operačním systémem, který se rovněž stará o rozdělení úlohy na jednotlivé výpočetní uzly více méně transparentně. Superpočítače jsou schopny provádět obrovské množství výpočtů za sekundu[1]. Jejich výkon se vyjadřuje v jednotkách FLOPS (počet operací v plovoucí desetinné čárce), dnes petaFLOPS (PFLOPS , biliarda, 10^{15}) nebo dokonce v exaFLOPS (EFLOPS, trilion, 10^{18}).

4.1.2 Clustery

Z hlediska HPC připomínají superpočítače, ale na každém výpočetním uzlu běží operační systém zvlášť s instancí výpočetní úlohy. Jejich předností je redundance, odolnost vůči částečným výpadkům a rozšiřitelnost. Jedním z druhů clusterů jsou i tzv. blade servery, které

[1] Dle žebříčku TOP500 je dnes nejvýkonějším superpočítačem Frontier (Oak Ridge National Laboratory v USA) s výkonem 1,194 exaFLOPS

dále posouvají koncept modulárnosti, ale zásadně i ceny zařízení. Z hlediska programování jde již o distribuovaný systém a je třeba řešit komunikaci mezi uzly samostatně nebo pomocí knihoven, jako jsou MPI-IO, Parallel HDF5, ADIOS, Hadoop, Spark, Lumify, Elasticsearch, Presto, Condor-G a mnohé další.

4.1.3 Cloud

Cloudy jsou často velké clustery (a soustavy clusterů), které používají virtualizované servery a kontejnery. Jsou flexibilnější z hlediska alokace dodatečných HW prostředků pro další instance úloh či jejich uvolnění a řízení těchto přidělených kapacit. Provozovatelé cloudů jako jsou společnosti Amazon, Google, IBM či Microsoft podporují vědecké výpočty na svých cloudech zpřístupňováním *datasetů* (datových množin, například pro učení umělé inteligence), slevami pro akademickou sféru, odměnami za úspěšný výzkum a přímou podporou granty [50].

4.1.4 Grid

Obecně se jedná o vysoce heterogenní distribuovaný systém, kde se mění dostupnost výpočetních uzlů s přerušováním výpočtu a to klade vysoké nároky na granularitu úlohy a vhodné sestavování mezivýsledků a případných změn v přiřazení uzlů při nedodržení *deadline* pro mezivýsledek. Vědecké projekty často pomocí svobodné softwarové platformy BOINC [51] využívají kombinaci vlastních počítačů, darovaného výpočetního výkonu osobních počítačů a serverů od dobrovolníků. V tomto případě jsou také zvýšené nároky na ověřování výsledků vícero výpočetními uzly kvůli chybám (běžné počítače bývají bez zabezpečených ECC pamětí) a podvrženým výsledkům. Příkladem jsou projekty jako Folding@Home (simulace složitých organických molekul, převážně proteinů a nukleových kyselin; výpočetní jádro GROMACS), SETI@Home (hledání důkazů existence mimozemského života analýzou radiových signálů z vesmírného prostoru), Einstein@Home (hledání gravitačních vln analýzou dat relevantních experimentů). Takové gridové systémy často využívají pouze zbytkový výkon počítače, který jeho uživatel v daném okamžiku nepotřebuje. Pro účely výzkumu viru SARS-Cov-2 se v rámci projektu Folding@Home podařilo dosáhnout výpočetního výkonu přes jeden exaFLOPS [52].

4.1.5 Více jádrové CPU (multi-core)

S růstem počtu diskrétních jader procesorů vzrůstají nároky na programovací jazyky, operační systémy a různá API v oblasti horizontálního škálování, optimalizace lokálnosti přístupů do paměti a vyhnutí se konfliktům paralelních procesů (uváznutí, vyhladovění).

Oproti GPU je menší poměr mezi počtem vláken a operační paměti. Přístup do paměti mívá také více úrovní vyrovnávací paměti (*cache*), která je z pohledu programátora vždy koherentní. V programovacím modelu tedy nemají vlákna vlastní skrytou paměť. Skutečné počty registrů viditelných programátorovi jsou také skryté, hlavně u *user space* kódu. Je třeba rovněž mít na paměti, že *cache* pracuje v řádcích (*cache line*, většinou 64 bytů). Synchronizace mezi vlákny, které pracují se stejnou pamětí, vyžaduje využití dalších synchronizačních primitiv implementovaných hlavně pomocí atomických instrukcí.

Superpočítače firmy Cray používaly v 70. letech 20. století odlišný koncept vektorových instrukcí, kde nebyly nutně pevně stanovené šířky registrů, ale parametricky zadaný počet hodnot pro zpracování. Tato data pak na dostupné SIMD (Single Instruction Multiple Data) či skalární ALU výpočetní jednotky procesor samostatně rozloží. Využití tohoto principu je navrhované jako rozšíření instrukční sady RISC-V [53]. Proprietární instrukční sady GPU a jiných paralelních procesorů mohou tyto principy také využívat, je to však předmětem obchodních tajemství a spekulací. U instrukční sady x86 se podobná vektorizace využívá u přesunu dat z/do paměti či periferních vstupů/výstupů přidáním opakovacího REP[2] prefixu přenosovým řetězcovým instrukcím (INS, MOVS, OUTS, LODS, STOS) a REP(N)E/REP(N)Z prefixům porovnávacím řetězcovým instrukcím (CMPS, SCAS). Pro výpočetní skalární a SIMD instrukce však tyto prefixy definovány nejsou [54].

4.1.6 FPGA

Field Programmable Gate Array, programovatelná hradlová pole, jsou zvláštním druhem obvodu, který se s výhodou využívá pro prototypování integrovaných obvodů a tvorbu specializovaných akcelerátorů. Programování probíhá dvojúrovňově jako konfigurace zapojení buněk, které pak realizují prvky obvodu: logická hradla, statická paměť a datové cesty. Samotné takto vzniklé zapojení může být programovatelné (tzv. *softcore*) nebo poskytovat pouze urychlení konkrétního výpočtu.

4.1.7 ASIC

Pro některé aplikace se vyplatí na zakázku vyrobit aplikačně specifický integrovaný obvod. Většinou s prototypem realizovaným FPGA, ovšem s dodatečnými simulacemi a testováním při výrobě. Oproti FPGA dosahuje obecně ASIC provozu na vyšších frekvencích a nižší spotřeby energie. Velkou nevýhodou je náročnost, délka a cena celého procesu, tzv. *tapeout*. Pro menší série se většinou nevyplatí použít nejnovější výrobní procesy a to dále znevýhodňuje ASIC realizaci oproti FPGA u malých sérií. V poslední době je však situace příznivější, hlavně pro programovatelná *embedded* řešení s využitím RISC-V instrukční sady s vlastními rozšířeními pro ovládání akcelerátorů [55]. Proces se tedy zlepšuje.

Dalším rozvojem je v této oblasti využití *chipletů* propojovaných na *interposeru* pro zvýšení výtěžnosti výroby a amortizací nákladů, které by se opakovaly pro monolitická řešení (masky pro litografii) [56]. ASIC řešení vyráběné v menších sériích jsou však vyráběná staršími technologiemi, dnes obvykle 28nm a horšími. Musí tak soupeřit s programovatelnými CPU/GPU vyráběnými často na technologiích 7nm a lepších. Výjimkou je například masově vyráběný AI akcelerátor Tensor Processing Unit (TPU) verze 4 firmy Google, vyráběný 7nm procesem. V první verzi byl však vyráběn 28nm procesem v roce 2016, kdy populární GPU NVIDIA GP104 užité v kartách GeForce 1080 GTX byly vyráběny 16nm (TSMC) procesem a populární CPU Intel Core i7-6700K byly vyráběny 14nm (Intel) procesem. Je třeba poznamenat, že porovnávání výrobních procesů různých výrobců jen podle počtu nanometrů, které uvádějí každý dle vlastní metodologie, je zavádějící.

[2] Prefix REP zajišťuje opakování řetězcové instrukce dokud se hodnota registru RCX nevyčerpá (x86-64).

4.1.8 Wafer-Scale Engine

Společnost Cerebras uvedla na trh produkt, o který se již v minulosti spíše neúspešně pokoušelo více společností, například Trilogy Systems již mezi léty 1980 až 1985. Při výrobě integrovaných obvodů jako křemíkových chipů používá kruhová křemíková destička (*wafer* neboli „oplatka“). Na tu se postupně pomocí litografických, mechanických a chemických procesů vytvoří funkční zapojení jednotlivých kopií obvodu a následně dojde k rozřezání na jednotlivé kusy. Protože proces není dokonalý, některé kusy je nutno vyřadit jako vadné. Někdy je možné část vadných kusů prodat jako méně kvalitní produkt, pokud to architektura obvodu umožňuje. Například 8 jádrový procesor, který má 6 jader v pořádku lze prodat s tím, že se dvě vadná deaktivují. Pokud však tato výrobní chyba byla v obvodech sdílené vyrovnávací paměti, paměťového řadiče, vnitřní sběrnice nebo řadiče externích sběrnic, nelze vadný kus použít vůbec. Procesor společnosti Cerebras je navržen tak, že může zabírat výrazně větší plochu (více než 50x) křemíkové destičky než jiné velké chipy. Je navržený modulárně tak, aby se vyrovnal s výrobními vadami výrazně lépe než konkurenční architektury, včetně chyb na sběrnicích.

Protože jde o nové řešení s velkou vstupní investicí jak cenou hardware, tak změnami v software, není jeho použití zatím rozšířené. Například Argonne National Laboratory v USA tyto procesory využívá k několika výzkumům napříč vědními obory v první i druhé verzi [57, 58]. Software tým Cerebras vynakládá nemalé úsilí, aby poskytl kompatibilitu s oblíbenými knihovnami pro neuronové sítě a strojové učení: TensorFlow a PyTorch. Je také možné si výpočtový čas pronajmout prostřednictvím společnosti Cirrascale Cloud Services.

4.1.9 GPGPU - General Purpose Graphics Processing Unit

Neustálé zvyšování nároků počítačových her na flexibilitu grafické pipeline vedlo k náhradě pomocí programovatelných jednotek zvaných *shadery*, ty byly zprvu rozděleny na jednotky pro zpracování prostorových bodů (vertexy) a jednotky pro zpracování obrazových bodů (pixely). Po sjednocení typů výpočetních jednotek jako “unified shaders” se objevily obecné výpočetní jednotky. S tím vznikaly programovací jazyky pro psaní “shader programů”, nejprve přímo ve specializovaném assembleru (v OpenGL postupně jako rozšíření firmy NVIDIA, následně standardizováno jako ARB_vertex_program a ARB_fragment_program v roce 2002), pak jazycích podobných C: Cg (C for Graphics, NVIDIA), GLSL (The OpenGL Shading Language, Khronos Group), HLSL (High-level shader language, Microsoft). Tyto jazyky však měly různá omezení, například nepracovaly s ukazatelem na data či funkce. Vykreslovaná (renderovaná) 3D grafika je založená na aplikaci lineární algebry 4 složkových vektorů a 4x4 transformačních matic. Paměť textur a texturovací jednotky lze chápat jako pole hodnot s různými režimy interpolace (bilineární, trilineární) a optimalizací pro 2D přístup. Postupně začaly pokusy o využití shaderů pro obecné výpočty, například simulace molekulární dynamiky. Převod výpočtů do jazyka grafiky byl však často velmi náročný a někdy nevedl k efektivnímu rozdělení výpočtu [59].

S příchodem SW platformy CUDA C společnosti NVIDIA bylo možné tyto jednotky programovat mimo grafický kontext. Skupina Khronos reagovala otevřenými standardy OpenCL a SYCL, též založených na programovacích jazycích C a C++. Firma AMD

prosazuje kromě otevřených standardů své řešení HCC a ROCm. Například firma Microsoft také reagovala rozhraními C++ AMP (označeno jako *deprecated;* zavrženo ve Visual Studio 2022), DirectCompute (Direct3D 10 a 11) a Compute shader (Direct3D 12) [60].

Další standardy pro GPU výpočty jsou implementovány v rámci grafických API OpenGL, Vulkan a DirectX jako speciální *compute shadery*. Ve světě mobilních zařízení s procesory ARM je populární také jazyk RenderScript. Jejich použití není závislé na grafice, může představovat možnost jak výpočty provozovat na HW, pro který existuje grafický driver, ale chybí podpora OpenCL nebo CUDA. Tato situace nastává u embedded řešení jako jsou mobilní telefony, tablety a různé minipočítače a vývojové desky.

Nově se pro webové aplikace zpřístupňují GPGPU výpočty přes dvojici rozhraní: WebGL 2.0 Compute (opuštěno, původně podporováno firmou Intel) a WebGPU, které je novější a nejspíše bude standardizováno W3C [61] (vývoj byl započat společnostmi Mozilla, Google a Apple).

Všechny tyto standardy buď posouvají (nebo dávají tu možnost) kompilaci GPU programu na runtime/driver nebo kompilují do zprostředkovatelské (*intermediate*) reprezentace, například PTX, SPIR-V a LLVM IR. Většinou lze vedle této reprezentace mít předem zkompilovaný strojový kód pro zvolené cílové architektury, které se použijí, pokud je instalován HW, který byl předpokládán a lze tak zrychlit spouštění programů (souvisí s pojmem *fat binary*).

Přestože v minulosti byly zvyšující se nároky počítačových her hlavním hnacím motorem vývoje GPU, obecné výpočetní aplikace hrají stále větší roli. Určitou změnu vývoje představovalo využití GPU pro akceleraci hashovacích funkcí v blockchainu pro kryptoměny. V současnosti hlavní směr vývoje GPGPU určují potřeby akcelerace hlubokých neuronových sítí (*deep neural networks*, DNN) a velkých jazykových modelů (*large language models*, LLM). Tento vývoj má však význam i pro grafiku, přidává se podpora akcelerace sledování paprsku (*ray tracing*) pro výpočty osvětlení a pomocí neuronové sítě se následně provede redukce šumu (*denoising*). Jednou z nejrychleji nasazených technik vyvinutou na GPGPU představuje umělé zvyšování rozlišení obrazu (*upscaling*), již dostupné ve spotřební elektronice (TV a herní konzole).

4.2 Architektura a programování GPU

Od běžných CPU se GPU fundamentálně liší v přístupu vícevláknového programování. Pro CPU je vlákno (*thread*) procesu poměrně vysoká abstrakce a je spravováno operačním systémem. Také v poměru k množství operační paměti je využíváno relativně málo vláken, které mohou pracovat na větším bloku paměti.

Využití SIMD jednotek je čistě závislé na kódu vlákna a operační systém neovlivňuje přímo jejich použití. Všechna vlákna v procesu jsou si rovna a mohou se vzájemně synchronizovat, OS je může uspávat a přesouvat mezi jádry CPU. Ve standardu jazyka C++ je definována třída `std::thread`, která je navržena čistě se zohledněním potřeb vláken CPU a pro programování GPU není vhodná.

GPU vlákna jsou velice “štíhlá” (lightweight). Organizují se do hierarchických skupin. Napříč skupinami nelze provádět synchronizaci, vlákna jiné skupiny nemusí ani zároveň běžet ani mít vytvořený kontext. Veškerý kontext vlákna v globální RAM musí být předem explicitně definován, jsou však také dostupné hierarchie lokálních pamětí vlákna i skupiny (CUDA nazývá shared memory) pro určitou míru synchronizace. O plánování spuštění a udržování kontextu skupin vláken se stará přímo HW plánovač v kooperaci s ovladačem OS. V rámci skupin nejnižší hierarchie (CUDA je nazývá *block*) jsou vlákna dále dělena na podskupiny, které už zaručeně běží zároveň (CUDA je nazývá *warp*) a přiřazují se na SIMD jednotky. Tento model NVIDIA nazývá *Single Instruction Multiple Thread* (SIMT). Velikost podskupin se může lišit podle dostupného HW, z hlediska programovacího modelu a kompilace není přímo podstatná. Znalost této velikosti však může rozhodovat o vhodné optimalizaci rozdělení výpočtu.

Spuštění celé úlohy sestává v definování hierarchie rozdělení skupin (v CUDA jako *grid* a *block*) a spuštění programu, který všechna vlákna sdílejí, zvaného *kernel*. Tento kernel je psaný z pohledu jednoho vlákna, u klasického případu paralelizace nahrazuje vnější cyklus for. Nejsou zde žádné triky vektorizace kompilátoru známé z programování CPU, veškerý paralelismus je explicitní. Organizace skupin může být vícerozměrná. Každé vlákno pak přímo v HW kontextu má k dispozici svoji pozici v hierarchii a velikost buněk. Není pak třeba u vláken řešit uložení kontextu indexů v globální paměti na rozdíl od CPU řešení.

Pokud je třeba ve výpočtu synchronizovat obecně vlákna napříč skupinami, je třeba výpočet rozdělit na spuštění více podúloh jako jednotlivých kernelů. Nevýhodou je zvýšená režie běhového prostředí (run-time) a ovladače (driver). Je tedy vhodné počet spuštění kernelů minimalizovat. Aby byly některé problémy řešitelné byla také přidána podpora atomických operací v globální paměti, avšak jejich využívání by mělo být omezeno na nezbytně nutné. Atomické operace nemohou sloužit ke vzájemné synchronizaci skupin (bloků), protože není zaručeno, že budou tyto skupiny spuštěny opravdu paralelně.

Na obrázcích 4 a 5 jsou znázorněny dvě různé architektury GPU. První je architektura Turing společnosti NVIDIA, která je použita v produktech pro profesionální grafiku, například Quadro RTX 8000, obecné výpočty, například Tesla T4, a produktech pro herní real-time grafiku, například GeForce RTX 2080. Architektura Turing představovala v roce uvedení 2018 velký vývojový skok, kdy byla kombinována standardní GPU výpočetní jádra s novými technologiemi pro real-time zpracování realistického osvětlení – *ray tracing*. V době uvedení poskytovala tato architektura bezkonkurenční výkon při zpracování grafických i výpočetních úloh. Zobrazená struktura GPC → TPC → SM architektury Turing na obrázku 4 zajišťuje, že výpočetní a grafické úkoly jsou efektivně rozděleny a zpracovány, což maximalizuje výkon a efektivitu GPU. Poznamenejme, že významným vylepšením byla Tensor Cores[3] (TC), představená již v architektuře Volta (V100). TC jsou součástí Streaming Multiprocessors (SP) a jsou výhodné pro akceleraci výpočtů pro AI a hluboké učení.

V současnosti je nejnovější architektura Hopper, dostupná pouze pro servery, herní grafiky a karty pro pracovní stanice používají architekturu Ada.

[3] Tensor Cores umožňuje operace Fused Multiply-Add (FMA) nad maticemi, tj. násobí prvky dvou matic a výsledky násobení sčítá s prvky třetí matice při pouze jediném zaokrouhlení.

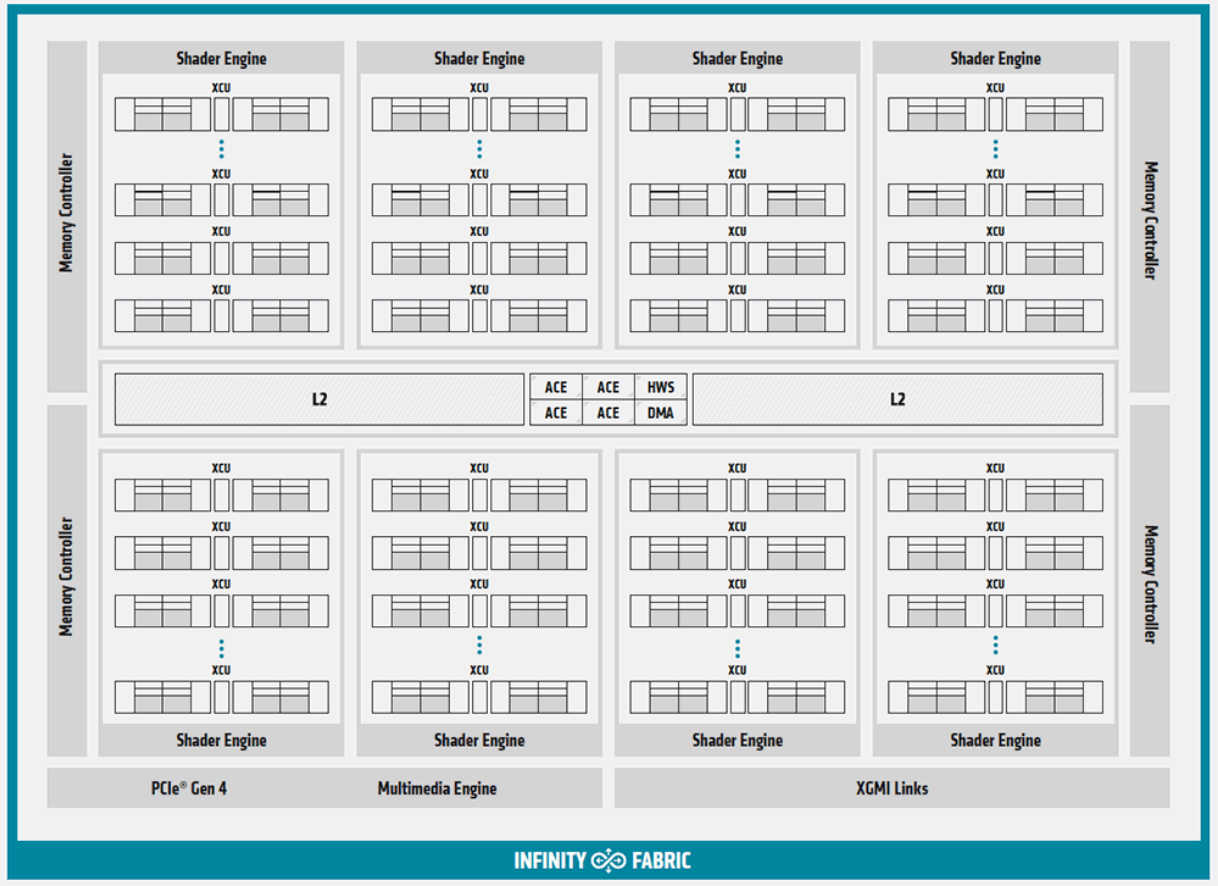
Obr. 4: GPU architektura NVIDIA Turing – čip TU102 [62].

Konkrétněji jádro Turing obsahuje 6 výpočetních klastrů (GPC – Graphics Processing Cluster), z nichž každý obsahuje více TPC (Texture Processing Clusters), SM (Streaming Multiprocessors) a RT Core (Ray Tracing Core), což jsou specializované jednotky pro urychlení výpočtů potřebných pro *ray tracing*. Uprostřed je 6 MB velká L2 *cache*.

Druhá architektura uvedená v roce 2020 je CDNA (Compute DNA) společnosti AMD. Tato architektura optimalizovaná pro produkty obecných výpočtů. Například GPU karta AMD Instinct MI100 pracuje nativně i se standardním formátem FP64. V současnosti je nejnovější CDNA3 obsažená v akcelerátorech Instinct MI300, silně orientovaná na podporu výpočtů v oblasti AI a ML. Pro real-time a herní grafiku prosazuje AMD architekturu RDNA, která také může v určité míře sloužit pro obecné výpočty, ale s omezenější softwarovou podporou (bez platformy ROCm: Radeon Open Compute). Druhá verze architektury, RDNA2, situaci pro herní GPU AMD zlepšil jen částečně a ROCm podporuje pouze nejvyšší modely, které sdílejí jádro s modely určenými pro pracovní stanice [63].

Obě architektury vykazují značnou podobnost. Podporují standardní PCI Express rozhraní pro komunikaci se zbytkem systému a zároveň vlastní řešení pro rychlou komunikaci více karet a jiných komponent stejného výrobce. V případě AMD jde o rozhraní *Infinity Fabric*, NVIDIA prosazuje své rozhraní *NVLINK*. Obě architektury využívají vyrovnávací paměť

druhé úrovně (L2 *cache*). Je viditelné rozdělení výpočetních jednotek, které pak odpovídá v programovacím modelu spouštění skupin (*group*, *block*) a podskupin (*warp*) vláken.



Obr. 5: GPU architektura AMD CDNA – Instinct MI100 [64]

Konkrétněji je jádro architektury CDNA složeno z 8 SE bloků (Shader Engine), přičemž každý je vybaven 16 bloky XCU (eXtended Compute Unit), z nichž každý obsahuje 64 SP (Streaming Processor) a uprostřed se nachází 8 MB velká L2 *cache*.

Při programování mikroprocesorů jsou velmi důležité vzory přístupu do paměti z hlediska *cache* paměti. Pro GPU to platí také stále více, ovšem je zde specifická potřeba využívat tzv. sdružené (*coalesced*) přístupy, kdy vlákna ve stejné skupině (nebo alespoň podskupině) přistupují k paměti přímo za sebou. To umožňuje paměťovému řadiči tyto přístupy sdružit do jednoho přenosu. Jelikož se ke každému přenosu váže latence, je nutno jejich počet minimalizovat pro dosažení vysoké propustnosti. Možnosti paměťových řadičů se neustále zlepšují, kdy zvládají takto sdružovat i přístupy permutované či adresově nezarovnané (*misaligned*).

Z těchto důvodů je preferované uložení dat jako struktura polí (Structure of Arrays, SOA) na rozdíl od skalárního kódu pro CPU, kde se preferuje rozdělení do polí struktur (Array of Structures, AOS), které jsou pro jedno vlákno rozprostřeny v co nejméně buňkách vyrovnávací paměti (*cache lines*). Předchozí tvrzení je zavádějící a pro některé výpočty se na CPU také preferuje uložení jako AOS. Pro 2 a více rozměrná pole se zarovnávají (alignment, padding, pitched) adresy začátků řádků v paměti na určité násobky (podle

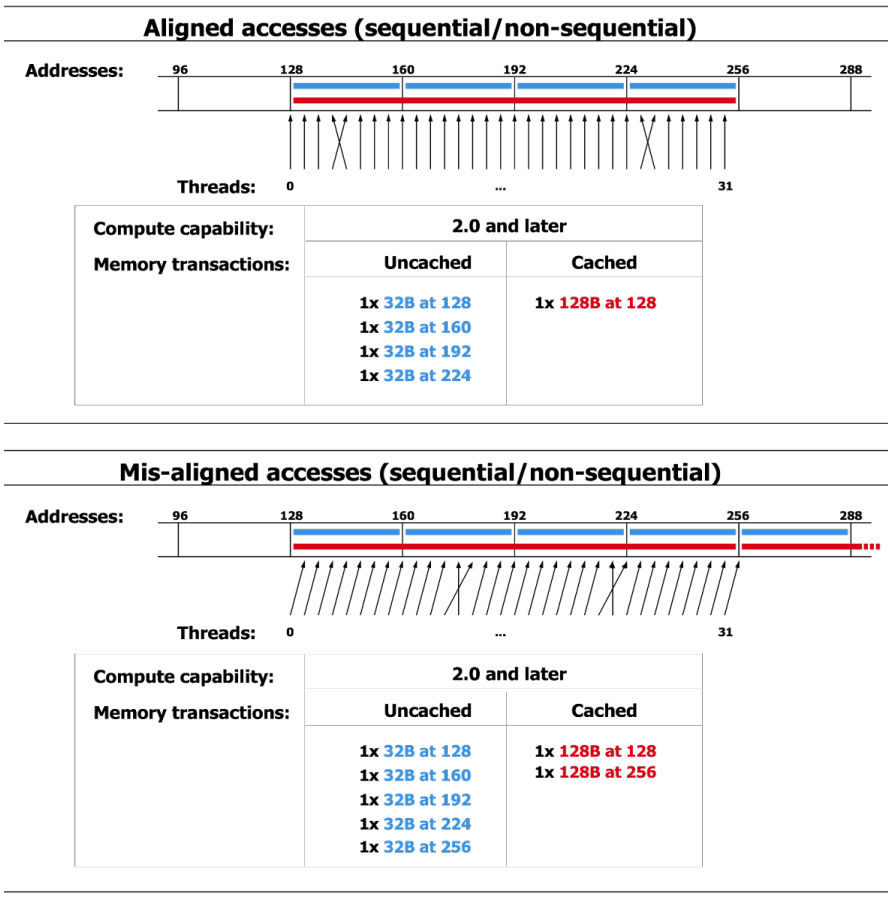
architektury paměťového řadiče, například na násobky 16). Pak je sdružený přístup zaručen. Jak se možnosti HW zvyšují a objevují se časté vzory, je sdružování přístupů i při určité permutaci v rámci pořadí sousedních vláken vůči pozicím v bloku paměti či zvýšené benevolence offsetu bloku paměti (misaligned). V první generaci CUDA však každý nesdružitelný přístup vedl na 16 oddělených přístupů, což často anulovalo výhody použití GPU pro urychlení daného problému. Obrázek 6 ilustruje příklad sdružitelných přístupů do paměti vlákny v podskupině (*warpu*), zarovnaný i nezarovnaný přístup.

Příklad pole struktur pro skalární CPU:

```
struct PersonStats {
  unsigned age, income; // atributy samostatně
};
PersonStats *persons_stats = new PersonStats[n_persons];
```

Příklad struktury polí pro GPU:

```
struct PersonsStats {
  unsigned number;  // počet osob
  unsigned *age, *income; // pole jednotlivých atributů
  PersonsStats(size_t number) {
    // alokace
  }
}
PersonsStats persons_stats;
```



Obr. 6: Příklady přístupu do paměti pro zařízení CUDA CC ≥ 2.0 [65]

Pro SIMD/SIMT je problémem divergence v rámci algoritmu, prakticky v rámci podskupiny vláken, která fyzicky běží zároveň na SIMD jednotkách. Menší problém je, pokud jen některá vlákna mají dělat práci navíc - neexistuje *else* větev, pak ji dělají všechna ovšem některé výsledky se nepoužijí. Pokud větev *else* existuje, může nastat příznivá situace, že pro všechna vlákna je podmínka nepravdivá. Jinak je třeba provést kód této větev se všemi vlákny v podskupině a uložit pouze relevantní výsledky (nazýváno jako *masking*). Právě zde může docházet ke značným ztrátám výkonu. Může se vyplatit data před spuštěním paralelního algoritmu uspořádat tak, aby k této situaci nedocházelo. [66]

Příklad nevhodné divergence algoritmu v podskupině (*warp*):

```
// vstup: A[] = {1, 0, 2, -1, 0, 4};  idx = 0..5
if(A[idx] > 0) { // prováděno kvůli idx∈{0, 2, 5}
  if(A[idx] > 2) {
    B[idx] = A[idx];   // prováděno kvůli idx∈{5}
  } else {
    B[idx] = -1;       // prováděno kvůli idx∈{0, 2}
  }
}
else {           // prováděno kvůli idx∈{1, 3, 4}
  if(A[idx]==0) {
    B[idx] = 0;        // prováděno kvůli idx∈{1, 4}
  } else {
    B[idx] = 2+A[idx]; // prováděno kvůli idx∈{3}
  }
}
```

Často lze divergenci podmínek skrýt přímo do výpočtu (*branchless arithmetic*) a vyhnout se tak instrukcím skoků, když už není možné se výpočtu pro všechna vlákna v podskupině vyhnout. Booleovské hodnoty pravdy a nepravdy se zobrazují na datové typy `int` a `float` jako 1 a 0. Pro předchozí příklad:

```
decltype(A) c1 = A[idx]>2 || A[idx]==0;
decltype(A) c2 = A[idx]>0 && A[idx]<2;
decltype(A) c3 = A[idx]<0;
B[idx] = c1 * A[idx] + c2 * -1 + c3 * (2+A[idx]);
```

Tato technika je používaná také pro skalární kód pro CPU pokud četné a nebo špatně předvídatelné větvení snižuje výkon *branch predictoru* a zabraňuje *pipeliningu* instrukcí [67]. Kompilátory (např. GCC a clang) v některých případech zvládnou generovat instrukce podmíněného přesunu, aby předešly použití instrukcí skoku a *branch predictor* je tak úplně vynechán [68].

4.2.1 Příklad programu v CUDA C

Kód pro CPU spouští GPU kernel speciální syntaxí `<<< b, tpb >>>` a předpokládá, že všechny ukazatele (`X`, `P`, `Y`) byly alokovány na GPU. Kompilátor NVCC tuto syntaxi nahradí vhodnou sekvencí volání CUDA Driver API (například funkce `cuLaunchKernel`) a předá již normální C/C++ kód kompilátoru pro CPU (GCC, MSVC) a kód `__global__` kernel funkcí a `__device__` funkcí předá kompilátoru pro GPU (dříve založený na Open64 a nyní na LLVM [70]).

```
__global__ void polyval_kernel(
    float *X, float *P, float *Y, int N, int P_n)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    float x = X[idx], y = 0;

    for(int j=0; j<P_n; j++) {

        y += P[j] * powf(x, P_n-j-1);

    }

    Y[idx] = y;
}

void polyval(float *X, float *P, float *Y, int N, int P_n)
{
    int tpb = 256; // počet vlánek v bloku
    int blocks = N/tpb + (N % tpb ? 1 : 0);
    polyval_kernel <<< blocks, tpb >>> (X, P, Y, N, P_n);
}
```

4.2.2 Příklad programu C++ SYCL

SYstem-wide Compute Language (SYCL) je jazyk založený na moderním C++. Počátek vývoje byl oznámen v roce 2014 s důrazem. Kód hojně využívá šablony, pomocné typy a anonymní *lambda* funkce. Například místo ukazatelů se používají přístupové (*accessor*) objekty, které hlídají režim přístupu do paměti (čtení, zápis). Na rozdíl od CUDA C, SYCL kód plně odpovídá normě ISO C++.

```
void polyval(
    cl::sycl::buffer<float, 1> &X_buf,
    cl::sycl::buffer<float, 1> &P_buf,
    cl::sycl::buffer<float, 1> &Y_buf,
    int N, int P_n,
    cl::sycl::queue &device_queue)
{
    device_queue.submit([&](cl::sycl::handler& cgh) {
        using cl::sycl::id;
        namespace access = cl::sycl::access;
        auto X = X_buf.get_access<access::mode::read> (cgh);
        auto P = P_buf.get_access<access::mode::read> (cgh);
        auto Y = Y_buf.get_access<access::mode::read_write> (cgh);
        cl::sycl::range<1> work_items{N};
        cgh.parallel_for<class polyval_kernel>(
            work_items,
            [X, P, Y](id<1> tid) {
                auto i = tid.get(0);
                float x = X[i];
                float y = 0;
                for(int j=0; j<P_n; j++) {
                    y += P[j] * pow(x, P_n-j-1);
                }
                Y[i] = y;
            }); // END cgh.parallel_for<class polyval_kernel>
    }

}
```

4.2.3 Generování pseudonáhodných čísel

Součástí vývojového balíku CUDA SDK je knihovna cuRAND, která obsahuje několik paralelních implementací více typů generátorů pseudonáhodných čísel. Každé vlákno programu musí mít vlastní kontext generátoru, alokuje pro něj paměť. Při inicializaci je zadáváno jak semínko (*seed*), tak číslo vlákna. To zjednodušuje inicializaci, protože není třeba generovat semínko pro každé vlákno zvlášť. Jsou dostupná dvě API, více kontroly nabízí tzv. *device API*, kdy se z kernelů řídí inicializace a generování čísel dle potřeby výpočtu [73].

Příklad použití cuRAND v CUDA kernelu s výchozím generátorem, nejprve inicializace a pak použití pro generování vektoru:

```
__global__ void init_rngs(curandState* rng, size_t N, uint64_t seed)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < N) {
        curand_init(seed, idx, 0, &rng[idx]);
    }
}

__global__ void generate_next_rnd_vector(
    curandState* rng, size_t N, float *V, float a, float b)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < N) {
        V[idx] = a + (b-a) * curand_uniform(&rng[idx]);

    }
}
```

4.3 Architektura a programování více-jádrových CPU

Současné více jádrové CPU se vyznačují poměrně velkými jádry, které jsou vnitřně paralelní, tzv. superskalární. Instrukce mohou být prováděny v jiném pořadí než jsou v kódu (*out of order execution*) a paralelně na více ALU (aritmeticko-logická jednotka).

K paralelizaci na jednotlivá jádra lze využít procesy a vlákna poskytovaná operačním systémem. V jazyce C++ je od verze C++11 poskytována ve standardní knihovně třída `std::thread`, která abstrahuje použití vláken přenositelně napříč platformami. Spouštění a ukončování vláken vyžaduje určitou režii operačního systému. Pro HPC výpočty bývá vhodné použít tzv. *thread pool*, což je skupina předem spuštěných vláken (ne více než je dostupných výpočetních jader), které si postupně rozebírají úlohy k výpočtu. Jednou z implementací, která takto využívá vlákna ze standardní knihovny `std::thread` je [69] a poskytuje rozhraní ve třídách `BS::thread_pool` a `BS::thread_pool_light`.

Pro datovou paralelizaci vznikaly pro x86 procesory postupně SIMD (Single Instruction Multiple Data; jediná instrukce zpracovává více prvků) rozšíření instrukční sady: MMX, 3DNow!, SSE a AVX. K těmto instrukcím také přibývaly nové vektorové registry (XMM, YMM, ZMM). U procesorů ARM je definované SIMD rozšíření NEON. Jedním z produktů,

který se snaží principy ověřené na GPU architekturách použít i pro multi-core SIMD x86 (později i ARM) procesory je Intel Implicit SPMD Program Compiler, známy pod zkratkou ISPC.

Rozšíření instrukční sady x86 důležité pro datově paralelní výpočty:

- SSE: interpretace 128bit registrů XMM0-7 jako čtyři 32bitové IEEE-754 čísla, poměrně nízký počet instrukcí
- SSE2: rozšiřuje možnosti interpretace 128bit registrů na dvě 64bitové IEEE-754 čísla a celočíselné typy
- SSE3: přidává možnost pracovat s čísly v jednom vektorovém registru (redukce)
- SSE4: další rozšíření instrukcí, například o skalární součin vektorů v jedné instrukci
- AVX: možnost použít starší 128bit a nové 256bit registry YMM0-15, výrazné rozšíření množství instrukcí, instrukce se třemi operandy a maskování
- FMA3 a FMA4: skalární i vektorové „sdružené násobení a sčítání“ (*fused multiply add)*; má výhodu, že se provádí pouze jedno zaokrouhlení výpočtu x = a*b+c . Pro použití se musí kompilátoru povolit toto sdružení provést (například pro GCC parametry příkazové řádky `-mfma`, `-mfma4`, `ffp-contract=fast`)
- AVX2: přidává hlavně instrukce pro práci s pamětí, například *gather* (ve vektorovém registru jsou offsety v poli a hodnoty na nich se načtou na příslušné pozice)
- AVX-512: přidává 512bit registry ZMM0-31 a mnoho užitečných instrukcí, které jsou rozděleny do dalších skupin, které byly postupně zařazované a CPU je nemusí podporovat všechny (např. FP16 pro 16bit čísla s plovoucí desetinou čárkou, tzv. poloviční přesnost – *half floats*).

Příklad použití AVX *instricts* v C++ [71]:

```
using Vec3D = std::array<__m128, 3>;
__m128 scalar_product(Vec3D a, Vec3D b) {
return _mm_add_ps(_mm_add_ps(_mm_mul_ps(a[0], b[0]),
                             _mm_mul_ps(a[1], b[1])),
                  _mm_mul_ps(a[2], b[2]));
}
```

4.3.1 Datový paralelismus v C++ a standardizace

Ve standardu C++17 bylo přijaté rozšíření standardní knihovny pro možnost paralelního výpočtu standardních algoritmů, tj. těch definovaných v hlavičkovém souboru `<algorithm>`. Funkce algoritmů (např. `std::for_each`) jsou přetíženy, aby na prvním parametru měly šablonový parametr `ExecutionPolicy`, což umožňuje specializaci funkcí jednotlivých typů paralelního spuštění, *politiky vykonávání*. Ve jmenném prostoru `std::execution` jsou pro jejich odlišení definovány třídy `parallel_policy`, `parallel_unsequenced_policy` a od verze C++20 také `unsequenced_policy`. Každý typ garantuje různě přísná omezení paralelizace. To jestli paralelizace proběhne pomocí vláken nebo vektorizací (SIMD či jinak)

je pak ponecháno na schopnostech konkrétní implementace standardní knihovny a kompilátoru.

Dle [72] může implementace standardní knihovny definovat i dodatečné *politiky vykonávání* a očekávalo se použití např. `std::parallel::cuda` či `std::parallel::opencl`, k čemuž však zatím žádná implementace nepřistoupila. Naopak v současnosti je situace taková, že kompilátor NVCC podporuje pro CUDA pouze standardní *politiky vykonávání*, a to pomocí parametru příkazové řádky. Ve zdrojovém kódu tedy nelze explicitně vyjádřit zda se má provést paralelní výpočet na GPU nebo CPU; podle přepínače se všechny paralelní výpočty v modulu provedou buď na CPU nebo na GPU. Přesun paměti mezi CPU a GPU automaticky zajišťuje technologie CUDA Unified Memory. To však přidává kromě akcelerace samotného výpočtu, tzv. *offloading*, režii kopírování paměti [74]. Situace s GPU firmy AMD je obdobná a dle [75] za použití nástrojů jako HCC (již se nevyvíjí, pouze pro starší architekturu AMD GCN), AdaptiveCpp (dříve hipSYCL) a ROCm je prováděn *offloading* výpočtu na GPU s automatickým kopírováním paměti. Podpora se také aktivuje parametrem příkazové řádky kompilátoru. V polovině roku 2023 firma AMD také projevila zájem spolupracovat na přímé podpoře v kompilátoru LLVM clang [76]. Kontejnery a algoritmy implementované přímo pro GPU bez přesouvání paměti poskytují nestandardizované knihovny jako jsou například stdgpu [77] a Thrust/CCCL [78].

V technické specifikaci ISO/IEC TS 19570:2018 [79] jsou deklarovány postupy pro další rozšiřování standardu jazyka C++ o možnosti zápisu paralelních algoritmů i jinak než pomocí standardních algoritmů a vláken poskytovaných operačním systémem. Jedním ze vzniklých návrhů je dokument P1928R8 [80], který se již v osmé revizi zabývá implementací abstrakcí SIMD do standardní knihovny jako šablonová knihovna `std::simd`. Oproti *instrict* funkcím poskytovanými výrobcem jsou použitelné napříč platformami. Již nyní jsou dostupné pokusné implementace pro testování a vývoj normy v `std::experimental::simd` [81]. Pokud budou postupovat práce ve WG21 (pracovní skupina ISO pro C++) dle plánu, mělo by se rozšíření standardizovat ve verzi C++26. Příklad z [71] ukazuje rozdíl oproti *instricts*:

```
using std::experimental::native_simd;
using Vec3D = std::array<native_simd<float>, 3>;

native_simd<float> scalar_product(Vec3D a, Vec3D b) {
  return a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
}
```

4.3.2 SYCL C++

Stejně jako pro GPU je možné SYCL využít pro programování více jádrových CPU. Příklad použití viz kapitola 4.2.2. U některých CPU (AMD je nazývá APU) lze takto také využít integrované iGPU.

4.3.3 Intel ISPC

Jednou z možností jak implementovat datově paralelní algoritmy na CPU (x86, ARM) je využití Intel Implicit SPMD Program Compiler (ISPC) [82, 83]. Původně byl vyvíjen pod názvem “volta” [84]. Ten realizuje paradigma Single Program Multiple Data (SPMD) pomocí Single Instruction Multiple Data (SIMD) a maskování jednotlivých prvků ve vektorovém registru při podmíněném vyhodnocení instrukcí. ISPC poskytuje možnost použít o něco vyšší úroveň abstrakce než zatím nestandardní `std::simd`, ale dává větší kontrolu nad výpočtem než *politiky vykonávání* nad standardními algoritmy C++. Avšak při integraci s C/C++ kódem je v jiném jazyce (C podobnému) a musí se vždy překládat do samostatného modulu, ke kterému je vygenerován hlavičkový soubor zprostředkovávající rozhraní exportovaných funkcí. Funkce, které mají v rozhraní přímo vektorizované hodnoty, označované jako *varying*, exportovat a tak přímo využít v C++ kódu nelze. Stejně tak nelze exportovat funkce úloh (task). Exportovaná funkce nesmí mít návratovou hodnotu a všechny parametry musí být označeny *uniform*, tj. skalární hodnoty nebo běžná pole či ukazatele.

Podobně jako při použití CUDA je nutné pro efektivní fungování SIMD použít správné vzorce/pořadí přístupů do paměti tak, aby se minimalizoval počet operací nad jednotlivými řádky vyrovnávací paměti (*cache line*). Také je třeba minimalizovat množství instrukcí typu *gather*/*scatter*, které načtou/uloží hodnoty ve vektorových registrech na různé adresy v paměti dané bázovou adresou a offsety ve vektorovém registru; tyto instrukce kladou velké nároky na efektivitu implementace vyrovnávací paměti a paměťového řadiče CPU. Ekvivalent CUDA termínu *warp* (zároveň běžící mikrovlákna) je v ISPC termín *gang*. Další úroveň paralelizace se dosahuje spuštěním těchto SPMD na více jádrech CPU. Na to poskytuje ISPC rozdělení na úlohy (*task*), v CUDA částečně odpovídá bloku vláken, ale nepoužívá se technika přepínání *gangů* pro skrývání latence. Naopak CPU může využít *pipelining* a *branch predictor*, který v jednodušších jádrech GPU chybí. Na rozdíl od OpenCL a podobných API, které provádějí závěrečný krok kompilace před spuštěním, kompiluje ISPC kód přímo do strojového kódu (přes LLVM) a umožňuje jej sestavit se zbytkem programu. Nevýhodou je pak nutnost opětovné kompilace, pokud je třeba podporovat nové rozšíření instrukční sady. To nastává hlavně při rozšíření velikosti vektorových registrů a změny kódů instrukcí k nim příslušejících, v historii i současnosti: přechod z 64bit MMX na 128bit SSE a AVX, potom 256bit AVX2 a na 512bit AVX512.

Pro použití rozdělení na úlohy (*task*) klíčovým slovem `launch` je třeba zajistit 3 funkce, které budou úlohy přiřazovat vláknům (nejlépe jako *thread pool*). Lze využít ukázkovou implementaci `tasksys.cpp`, která je distribuovaná spolu s ISPC.

Zajímavostí je, že ISPC používá nový fyzikální engine Chaos, který je součástí populárního software Unreal Engine 5 [85]. Ten se využívá stále více i pro potřeby průmyslových a vědeckých aplikací jak pro čistou vizualizaci tak pro (co-)simulaci, například Automated Driving Toolbox – Unreal Engine Scenario Simulation v prostředí Matlab Simulink [86].

Příklad kódu ISPC pro výpočet hodnoty polynomu, více jader + SIMD:

```
task void polyval_task(
    uniform float X[], uniform float P[], uniform float Y[],
    uniform int32 N, uniform int32 P_n, uniform int32 per_task)
  {
    uniform int X_offset = taskIndex0 * per_task;
    uniform int X_end = X_offset + per_task;
    if(X_end > N) {
        X_end = N;
    }
    foreach(i = X_offset ... X_end) {
        varying float y = 0, x = X[i];
        for(uniform int32 j=0; j<P_n; j++) {
            y += P[j] * pow(x, P_n-j-1);
        }
        Y[i] = y;
    }
}

export void polyval(
    uniform float X[], uniform float P[], uniform float Y[],
    uniform int32 N, uniform int32 P_n,
    uniform int32 n_tasks)
{
    uniform int32 per_task = N / n_tasks;
    if(per_task == 0) {
        n_tasks = N;
        per_task = 1;
    } else if(N % n_tasks) {
        per_task += 1;
    }
    launch [n_tasks] polyval_task(X, P, Y, N, P_n, per_task);
    sync; // vyčká na dokončení všech úloh
}
```

V generovaném hlavičkovém souboru je pro C++ definováno rozhraní exportovaných funkcí (po odstranění maker):

```
namespace ispc {
  extern "C" {
    extern void polyval(
      float * X, float * P, float * Y,
      int32_t N, int32_t P_n, int32_t n_tasks);
  }

}
```

5 IMPLEMENTACE METOD

Byl modifikován algoritmus HC12 s novou implementací Tabu listu na GPU. Dále byla vyvinuta nová varianta Diferenciální evoluce s více ostrovními populacemi vhodná pro paralelní implementaci na GPU.

5.1 HC12 a Tabu list

Známý algoritmus HC12 vyvinutý prof. Matouškem byl postupně implementován i s využitím GPU. Jeho třetí verze na GPU ve variantách původního Matlab/C++ řešiče s obecnou fitness funkcí a `hc12qs2_v2` pro řešení QAP problému pomocí překódování *qapswap2* byla rozšířena o seznam zakázaných (již navštívených) řešení, tzv. tabu list. Tento princip se snaží o zamezení ukončení algoritmu v lokálním minimu účelové funkce. Z principu algoritmu HC12 vyplývá, že se neukládá do tabu listu celá populace řešení v iteraci, ale vždy jen to nejlepší z těch, které v něm nejsou obsaženy. [90]

Každé vlákno tedy musí nezávisle na ostatních provést kontrolu, zda jím generovaný jedinec se v tabu listu již nenachází. Kvůli tomuto je prvním požadavkem na zvolenou strukturu tabu listu snížení množství přístupů do paměti a zároveň jejich sdílení mezi vlákny. Takovou strukturou je například nepřímo seřazené pole, které bylo použito.

5.1.1 Algoritmy Tabu listu

Binární vyhledávání v seřazeném poli je klasický algoritmus. Zde jsou seřazenými prvky celé vektory a tak při vyhledávání postupně po sloupcích, tj. prvcích vektorů, je třeba ošetřit novou situaci. V případě když se rovnají hodnoty ve sloupci hledané hodnotě, zjišťuje se rozsah řádků se stejnou hodnotou a ty se pak stávají novými mezními řádky pro hledání v dalším sloupci. Pokud však hodnota shodná není, algoritmus se dalším prohledáním sloupce nezdržuje. Podrobný popis je uveden v algoritmu 2.

Pro efektivní přidávání položek tabu listu je seřazení struktury nepřímé. Zvlášť existuje neseřazené pole řádků a pole indexů seřazených řádků. Proces přidání položky ilustruje algoritmus 1 a obrázek 7.

Implementace využívá předpokladu, že většina přístupů bude provedena více než jednou skupinou vláken; první přístup na krajní a prostřední prvky prvního sloupce provádějí všechna vlákna. Mělo by tedy docházet často k *broadcastu* jedné hodnoty do více vláken a také k použití hodnot z vyrovnávací paměti druhé úrovně (L2 *cache*), která je mezi SM sdílená.

Prototypování implementace probíhalo ve 4 krocích:

1. Jednodušší program s NumPy vektorizací pro kontrolu správnosti výsledků při vyhledávání a přidávání řádků do tabu listu. Neřeší efektivnost, pouze jasnost kódu a správnost výsledků. Neprovádí binární vyhledávání a obsahuje vlastní funkci *sortrows* pro NumPy.

2. Implementace binárního vyhledávání pro více sloupců v Pythonu nad NumPy poli, ale pouze skalární přístup po jednom prvku. Kontrola shodnosti výsledků s předchozí

implementací. Po ověření správnosti byly výsledky převedeny do jazyka C++ ve formě hlavičkového souboru s konstantními daty.

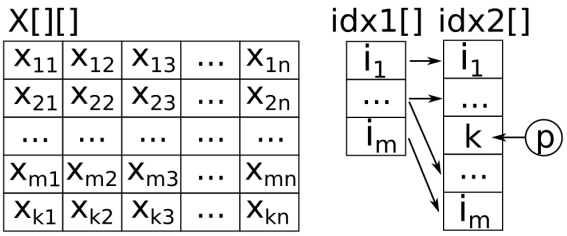
3. Program v C++ pro hledání a přidávání položek tabu listu na CPU s omezením na čisté C ve funkcích pro další krok. Ověřování správnosti výsledků z vloženého hlavičkového souboru z předchozí implementace.

4. Výsledná implementace tabu listu s kernely v CUDA C. Testy ověřují správnost výsledků pro hledání více hodnot paralelně ve více vláknech. Výsledky jsou shodné s předchozími implementacemi.

Tabu list se rozšiřuje přidáním řádku na konec a pak úpravou pole indexů. Po neúspěšném vyhledání řádku algoritmus již zná pozici pro vložení indexu. Udržují se dvě pole indexů, vždy z jednoho se kopíruje a do druhého se vkládá upravená verze a následně se vymění jejich ukazatele. Může tak probíhat paralelně kopírování indexů s vynecháním místa pro vkládaný prvek.

Algoritmus 1: vložení řádku do nepřímo seřazeného pole

```
vstup:
   X[1 … m, 1 … n] současný obsah tabu listu, m řádků, n sloupců
   V[1 … n] vkládaný řádek
   p pozice pro vložení řádku
   idx1[1 … m] indexy seřazených řádků, staré
   idx2[1 … m+1] indexy seřazených řádků, nové
inicializace:
   k ⟵ m+1
paralelně pro j ⟵ 1 … n
└   X[k][j] ⟵ V[j]
paralelně pro i ⟵ 1 … m
|   pokud i < p potom
|   └   idx2[i] ⟵ idx1[i]
|   jinak pokud i ≥ p potom
|   └   idx2[i+1] ⟵ idx1[i]
|  jinak
└   └   idx2[p] ⟵ k
výměna_ukazatelů(idx1, idx2)
výstup:
   X; idx1; idx2
```



Obr. 7: Stav paměti po přidání řádku do tabu listu

Algoritmus 2: nalezení řádku v nepřímo seřazeném poli

```
vstup:
X[1 … m, 1 … n] současný obsah tabu listu, m řádků, n sloupců
V[1 … n] hledaný řádek (paralelně pro každé vlákno)
idx1[1 … m] indexy seřazených řádků

inicializace:
   a ⟵ 1   levý okraj intervalu
   b ⟵ m   pravý okraj intervalu
   f ⟵ -1  pozice nalezeného řádku
   p ⟵ 1   pozice kam přidat řádek, pokud není nalezen
   funkce Xi(r, s) → X[idx1[r]][s]
pro j ⟵ 1 … n   (cyklus A)
|   pokud Xa > Xb potom
|   └   b ⟵ a
|   pokud V[ j ] < Xi(a, j) potom
|   |   p ⟵ a
|   └   přerušení cyklu A
|   pokud V[ j ] > Xi(b, j) potom
|   |   p ⟵ a
|   └   přerušení cyklu A
|   dokud b - a ≥ 2    (cyklus B)
|   |   s ⟵ a + ⌊ (b - a) / 2) ⌋
|   |   Xsj ⟵ Xi(s, j)
|   |   pokud Xs = V[ j ] potom
|   |   |   na ⟵ a
|   |   |   pro i ⟵ m … a   (cyklus C1) sestupně
|   |   |   |   pokud Xsj = Xi(i, j) potom
|   |   |   |   └ na ⟵ i
|   |   |   |   jinak
|   |   |   └   └   přerušení cyklu C1
|   |   |   a ⟵ na
|   |   |   nb ⟵ b
|   |   |   pro i ⟵ b … m   (cyklus C2)
|   |   |   |   pokud Xs = Xi(i, j) potom
|   |   |   |   └   nb ⟵ i
|   |   |   |   jinak
|   |   |   └   └   přerušení cyklu C2
|   |   |   b ⟵ nb
|   |   └   přerušení cyklu B    byly nalezeny meze pro další sloupec
|   |   jinak pokud Xs > V [ j ] potom
|   |   └   b ⟵ s
|   |   jinak
|   |   └   a ⟵ s
|   |
|   |   pokud       Xi(a, j) = V[ j ]
|   |   a zároveň Xi(b, j) ≠ V[ j ] potom
|   |   └   b ⟵ a
|   |   jinak pokud Xi(a, j) ≠ V[ j ]
|   |   a zároveň Xi(b, j) = V[ j ] potom
|   |   └   a ⟵ b
|   |
|   |   pokud b - a = 1 a zároveň Xi(a, j) < V[j] < Xi(b, j)
└   └   └   p ⟵ a + 1
výstup: (f, p)
```

5.1.2 Programové rozhraní obecného řešiče (API)

Řešič HC12 s obecně volitelnou uživatelskou účelovou funkcí je řešen v C++ jako C MEX pro Matlab (dynamická knihovna s funkcí `mexFunction`). Její veřejné rozhraní je uvedeno v tabulce 1. Účelová funkce je také C/C++ MEX funkce nebo běžná Matlab m-funkce a její volání (a případné hledání souboru) využívá funkci `mexCallMATLAB`. Středobodem implementace je struktura `HC12_pars` a příslušné funkce s prefixem `HC12_pars_`. Další třídy pro management CUDA kernelů a Tabu listu již používají C++ třídy a dědičnost. Volba datového typu užívaného pro hodnoty fitness funkce ve vektoru **F** a matice reálných parametrů **R** se provádí definováním makra `USE_FLOAT` v době překladu. Volí se mezi `float` a `double`. Pro matice **K**, **M**, **I**, **B**, je použito bez znaménkové 32 bitové celé číslo (`unsigned int`).

Tabulka 1: Parametry MEX funkce Matlab rozhraní obecného řešiče HC12

mode	**Význam, další parametry a návratové hodnota**
`'set-device'`	Nastavuje aktivní CUDA zařízení pořadovým číslem. Parametry: `deviceNumber`
`'params'` `'params:noR'`	Vytvoření `instance`, alokace paměti. Varianta `noR` zabraňuje alokaci matice **R**, účelová funkce pak počítá z matice **B** nebo **I**. Parametry povinné: `nParam`, `nBitParam` (počet bitů na parametr, může být vektor hodnot) `dodParam` (dvě hodnoty nebo řádek pro každý parametr) Parametry nepovinné s výchozí hodnotou: `maskOrder=2`, `startBit=0`, `bitLength=`*`(vypočítáno)`* Vrací `instance` jako pole `char`.
`'set-tabu-list'`	Inicializuje Tabu list. Parametry: `instance`, `maxLength`
`'run'` `'run:optsList'`	Spuštění HC12 algoritmu Za dvojtečkou mohou být v `optsList` modifikátory (oddělené čárkou, středníkem nebo svislicí): `taboo` – Tabu list povolen `consts_reload` – Opětovné načtení paměti konstant Parametry: `instance`, `mCode` – kódování (`'GCi'`, `'GC'`, `'BC'`), `nRuns` – počet restartů algoritmu, `startSeed` – vektor R → přepočítá se na K nebo `'bit-zeros'`, `funName` – jméno m-funkce, pokud je před jménem znak `@` chápe se jako CUDA funkce v MEXu (`functionParams`), `funOpt` – `'min'` nebo `'max'` Parametry nepovinné s výchozí hodnotou: `activationLimit=+-REAL_MAX` – práh aktivace Tabu listu `limitRows=100` – maximální délka Tabu listu

`'func-data'`	Nastavení přídavných `customData` pro CUDA funkci v MEXu s rozhraním `functionParams`. Parametry: `instance, funcName,` *`další parametry podle funkce ...`*
`'func-data-clean'`	Uvolnění přídavných `customData` pro CUDA funkci v MEXu s rozhraním `functionParams`. Parametry: `instance, funcName`
`'kernel-verbose'`	Výpisy detailů aktivace CUDA kernelů Parametry: `instance, verbose`
`'profiler-on'`	Aktivace měření statistik spouštění CUDA kernelů `Parametry: instance`
`'profiler-output'`	Vrací naměřené statistiky spouštění CUDA kernelů Parametry: `instance`
`'info'`	Uvolnění paměti Parametry: `instance`
`'free'`	Uvolnění paměti Parametry: `instance`

Uživatelem definovaná m-funkce dostává na vstup matici R v řádcích a očekává na výstupu vektor F:

```
function [ F ] = F_happycat( R )
    [~, N]  = size(x);
    alfa   = 1/8;
    norma  = sqrt(sum(R.^2,2)).^2;
    F = ((norma-N).^2).^alfa+1./N.*(0.5*norma+sum(R,2))+0.5;
end
```

Komunikace mezi řešičem a uživatelskou CUDA MEX funkcí zajišťuje předávání kopie struktury `functionParams` jako vektor typu `char` přímo přes Matlab:

```
typedef struct TfunctionParams
{
    unsigned char magic[2]; // 0×A5 0×D7
    REAL *R;
    size_t R_pitch;
    REAL *F;
    unsigned nParam, rows, tpb;

    char initCustom;
    void *custom; // data inicializovaná mode='func-data'

    unsigned *B;
    unsigned *I;
    char optionGrayCode;
    unsigned lastMinIndex;

    unsigned iterationNumber;

} functionParams;
```

Uživatelem poskytnutá CUDA MEX funkce má následující strukturu:

```
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])

{
    const mxArray **inputs=prhs;
    mxArray **outputs=plhs;
    int numInputs=nrhs, numOutputs=nlhs;

    if(numInputs==0) {
       mexErrMsgTxt("ERROR in test_func: missing parameters.");
       return;
    }

    input0_classId = mxGetClassID(inputs[0]);

    // (…) funkce se může podle typu parametru rozhodnout provést jiný
    //      výpočet, použito v geatbx_cec_bundle

    // postoupení k běžnému chování: první parametr má bých char pole
    // obsahující kopii struktury functionParams
    if(input0_classId ≠ mxCHAR_CLASS) {

        mexErrMsgTxt("ERROR: functionParams structure must be passed "
                     "as char array.");
        return;
    }

    if(mxGetNumberOfElements(inputs[0]) ≠ sizeof(functionParams)) {

        mexErrMsgTxt("ERROR: size of char array is not "
                     "sizeof(functionParams) structure.");
        return;
    }

    if(pars→initCustom == 0) { // běžné volání
       // (…) volání CUDA kernelu

    } else if(pars→initCustom == 1) // inicializace customData
       // (…) pokud to funkce vyžaduje může inicializovat customData
       // a využít k tomu další parametry

       // změnu je třeba vrátit na výstupní parametr
        unsigned char *data=(unsigned char*)mxGetData(plhs[0]);
        memcpy(data, pars, sizeof(functionParams));

    } else if(pars→initCustom == 2)) {
       // (…) pokud funkce alokovala customData, je povinna je uvolnit

       // změnu je třeba vrátit na výstupní parametr
        unsigned char *data=(unsigned char*)mxGetData(plhs[0]);
        memcpy(data, pars, sizeof(functionParams));

    } else {
        exErrMsgTxt("ERROR: initCustom has wrong value!");
    }
}
```

5.1.3 Programové rozhraní C++ (API)

Jazyk C++ je na CPU hlavní implementací řešiče `hc12qs2_v2` a poskytuje hlavně tři třídy jako rozhraní: `Params`, `ParamsInstance` a `ParamsInstanceDF`. Každá postupně zjednodušuje uživateli práci s řešičem a případnou výrobu *wrapper* knihoven.

Zástupné názvy typů v šablonách reprezentují:

- `EL_TYPE` – prvky matic (**K**, **M**, **I**, **B**)
- `IORD_TYPE` – prvky permutace, `0` až `nElements-1`
- `REAL` – prvky matice **T** (`DF`) a hodnoty účelové funkce ve vektoru **F** (také prvky matice **R**, kterou QAP nepotřebuje)
- `CC_type` – pomocné konstanty (CC = ConstantsClass) k `REAL` typu (`float` konstantu nelze použít přímo jako parametr šablony, je využit `static constexpr`)
- `f_type` – účelová funkce ve výsledku (přetypování `REAL`, ideálně jsou typy shodné)

Příklad pomocného typu `CC_type` s informací o `REAL`, konkrétně `float`:

```
struct CC_float
{
    static constexpr float REAL_MAX = FLT_MAX;
    static constexpr const char* fmt = "%f";
};
```

Struktura pro vrácení výsledků:

```
template<typename f_type>
struct Result
{
    f_type f = 0; // nejlepší fitness
    std::vector<unsigned short> order; // nejlepší permutace
    unsigned lastIteration = 0;
    std::vector<unsigned> K_GC; // nejlepší K v Grayově kódu

};
```

Veřejné rozhraní šablonové třídy pro instanci řešiče a (opakovaný) výpočet:

```
template <typename EL_TYPE, typename IORD_TYPE,
          typename REAL, typename CC_type>
class Params
{
    Params(
      unsigned swaps,  // počet výměn
      unsigned nElementsIN,  // velikost QAP problému
      unsigned nBitParamIN, // počet bitů na parametr
      REAL* DF_IN, // matice T
      unsigned nSlicesIN, // počet matic T pro stochastickou variantu
      unsigned maskOrder, // počet masek, max 3
      unsigned startBitIn, // počáteční bit v masce
      unsigned endBit,  // počet bitů v masce
      cudaDeviceProp& devicePropIN // informace o zvoleném CUDA zařízení
   );
```

```
    void initTabuList(unsigned maxLength); // aktivace Tabu listu

    unsigned getNParam();
    unsigned getNElements();
    REAL getBestFitness();
    unsigned getIterationNum();

    // spuštění výpočtu s nastavením výchozí permutace,
    // která se bude měnit a výchozího K
    template<typename Ftype>
    auto run(unsigned short* initOrder, unsigned* initKernel);

    // konrola zda order obsahuje permutaci 0 až nElements-1;
    // order_d není využito
    void checkValidOrder(
       IORD_TYPE*order, GPUPointer<IORD_TYPE> *order_d=nullptr);

    ~Params();
};
```

V metodě `run` jsou prováděny kontroly platnosti vlastností typů, oba výroky musí být pravdivé:

```
std::is_integral<FType>::value == std::is_integral<REAL>::value

!(std::is_integral<FType>::value  && std::is_integral<REAL>::value
  && sizeof(FType) < sizeof(REAL))
```

Pro automatický výběr co nejmenších datových typů pro prvky matic a vektorů je definována šablonová třída `ParamsInstance`. Její uživatel již volí jako parametr šablony pouze typ `REAL` pro uložení prvků `DF` matice a k němu příslušející pomocný `CC_type`. Obaluje instanci třídy `Params` s výběrem typů podle hodnot parametrů `nBitParam` a `nElements`, kterou ukládá jako ukazatel `void* p`. V každé metodě pak provádí přetypování `reinterpret_cast` podle hodnot parametrů `nBitParam` a `nElements`. Kompilátor je tak nucen vygenerovat všechny potřebné specializace šablonové třídy `Params`. Pokud jsou hodnoty parametrů nepodporované, dojde k vyhození výjimky `std::runtime_error`. Také je `Params` předána struktura `cudaDeviceProp` získaná pro již zvolené CUDA zařízení. Veřejné rozhraní třídy:

```
template <typename REAL, typename CC_type>
class ParamsInstance
{
    ParamsInstance(unsigned swaps, unsigned nElements,
        unsigned nBitParam, REAL* DF, unsigned nSlices,
        unsigned maskOrder, unsigned startBit, unsigned endBit);

    template <typename Ftype>
    auto run(unsigned short* initOrder, unsigned* initKernel);

    void initTabuList(unsigned maxLength);

    ~ParamsInstance();
};
```

Pro další zjednodušení výběru typů se specializací šablonových typů a zjednodušení implementace dynamických knihoven C a Matlab *wrapperů* je také definovaná pomocná třída `ParamsInstanceDF`. Její definice konstruktorů a metod využívají maker preprocesoru, které

opakující se kód vkládají pro různé typy. Využívá typové introspekce RTTI (Run-time type information), což je nutné povolit při kompilaci. Obaluje šablonovou třídu `ParamsInstance` tak, že vytváří její specializace pro omezenou množinu typů pomocí přetěžování konstruktorů a `if-else-if-(...)-else` řetězců v metodách. Tedy vyhýbá se tomu, aby sama byla šablonová. Šablonám se však nevyhne metoda `run`, která tak explicitně řeší různost v návratovém typu (všechny podporované typy musí navzájem podporovat implicitní konverzi, což u základních numerických typů platí; v deklaraci se využívá klíčové slovo `auto`). Pokud se narazí na nepodporovaný typ, vyhodí se výjimka `std::runtime_error`. Všechny veřejné členské proměnné jsou nastaveny při konstrukci a jsou definovány konstantní. Umožňují tak snadný přístup k parametrům oproti použití *getter* metod. Veřejné rozhraní třídy:

```
class ParamsInstanceDF
{
public:
    const std::string p_typename;
    const unsigned swaps, nElements, nBitParam;
    const unsigned nSlices, maskOrder, startBit, endBit;

    ParamsInstanceDF( // příklad konstruktoru generovaného makrem
        unsigned swaps, unsigned nElements, unsigned nBitParam,
        float* DF, unsigned nSlices, unsigned maskOrder,
        unsigned startBit, unsigned endBit);
    template <typename Ftype>
    auto run(unsigned short* initOrder, unsigned* initKernel);

    void initTabuList(unsigned maxLength);

    ~ParamsInstanceDF();
};
```

Neveřejná členská proměnná `void* p` je ukazatelem na konkrétní zvolenou `ParamsInstance` a v tělech metod je vždy nutné provádět přetypování podle uloženého jména z `typeid` objektu v konstantní veřejné členské proměnné `p_typename`. S těmito kontrolami a konverzemi pomáhají makra, například pro definici části těla metody `run`:

```
#define ParamsInstanceDF_run(tn) \
    if (p_typename == typeid(tn).name()) { \
        auto ps = reinterpret_cast<ParamsInstance<tn, CC_##tn >*>(p); \
        return ps→template run<FType>(initOrder, initKernel); }
```

Byla vyvíjena snaha o co největší využití konstantní paměti, která má zvlášť *cache*. Protože `K` a `initOrder` jsou různě dlouhé a různých typů je definována struktura a makra:

```
struct Constants {
    unsigned rows, pars, nSlices, nElements;
    size_t M_pitch;
    unsigned initOrderOffset; // offset in buf, 4-bytes aligned
    char buf[60000];  // some headroom for kernel parameters
};
__constant__ Constants constants_cd;

#define K_cd ((EL_TYPE*)constants_cd.buf)

#define initOrder_cd ((IORD_TYPE*) \
          constants_cd.buf+constants_cd.initOrderOffset))
```

5.1.4 Programové rozhraní C (API)

Rozhraní řešiče `hc12qs2_v2` pro jazyk C a kompatibilní je řešeno jako dynamická knihovna `libhc12qap`. Většina programovacích jazyků nějakým způsobem umožňuje volat dynamické knihovny. Tato knihovna je řešena jako *wrapper* s *handle* ukazateli. *Handle* je ukazatel na blíže nespecifikovaný typ, v tomto případě C++ třídu `ParamsInstanceDF`, která je z pohledu C rozhraní pouze ukazatelem do paměti bez dalšího významu. Všechny veřejné metody, konstruktory a destrukce přes `delete` operátor jsou obaleny (*wrap*) C funkcemi. V případě přetížených funkcí (stejný název, jiné parametry) je třeba každou variantu dát do jiné C funkce s vlastním názvem. Případné přetížení operátorů je nutné taktéž obalit vhodně pojmenovanými funkcemi. Taktéž je nutné obalit a skrýt případné šablony, které rovněž není možné do rozhraní C dynamické knihovny exportovat. Také je nezbytně nutné obsloužit všechny výjimky a převést je na chybové kódy a případně vrátit kopii chybové zprávy.

Veřejné C rozhraní je definováno jako:

```
extern "C" { // zajišťuje použití C ABI

// zjištění platnosti GPU paměti
uint8_t is_mem_space_valid();

// pořadové číslo aktivního CUDA zařízení
Int32Result get_cuda_device_number(void);

// nastavení aktivního CUDA zařízení pořadovým číslem
Int32Result set_cuda_device_number(int32_t deviceNumber)

// uvolnění obecného C pointeru získaného z rozraní, např. chybové zprávy
 void free_pointer(void *p);

// vytvoření instance
ParamsResult params(
   uint32_t swaps, uint32_t nElements, uint32_t nSlices,
   // datový typ matice DF je zadán jménem, povoleny jsou názvy:
   //  uint32, uint64, int32, int64,
   // single, float, float32, double, float64
   void *DF_src, const char *DF_type_name_in,
   uint32_t maskOrder,
   int32_t startBit, int32_t bitLength);

// nastavení Tabu listu
void init_tabu_list(ParamsInstanceDF* instance, uint32_t maxLength);

// spuštění algoritmu
RunResult run(
   ParamsInstanceDF* instance, uint16_t* initOrder, uint32_t* initKernel)

// použití operátoru delete na instanci
void free_params(ParamsInstanceDF* instance)

} // extern "C"
```

Pomocné datové typy mají výchozí hodnoty a tedy mají výchozí konstruktor. To však nemění jejich datovou strukturu a jsou tak kompatibilní s C ABI, tzv. *plain old data* (POD) či *passive data structure*. Problém by nastal například při definici virtuálních metod. Také u nich není definován destruktor (jinak by byla třeba pomocná funkce pro uvolnění).

```
enum ErrorCodes {
    EC_OK = 0,
    EC_STD_EXCEPTION = 1,
    /* more codes here*/
    EC_UNKNOWN = 999999
};

struct ParamsResult
{
    int32_t error_code = EC_OK;
    // [WARNING] caller shoud free error_message with free_pointer
    char* error_message = nullptr;
    ParamsInstanceDF* instance = 0;
};

struct Int32Result
{
    int32_t result;
    int32_t error_code = EC_OK;
    // [WARNING] caller shoud free error_message with free_pointer
    char* error_message = nullptr;
};

struct RunResult
{
    // výsledek je v f_t{result_type}, 0⇒chyba
    int32_t result_type=0;
    int64_t f_t1=0;
    uint64_t f_t2=0;
    float f_t3=0;
    double f_t4=0;

    uint16_t *order=nullptr;
    uint16_t orderLength=0;  // should be nElements
    uint32_t lastIteration=0;
    uint32_t *K_GC=nullptr; // kernel in gray code
    uint32_t K_GCLength=0;

    int32_t error_code=0;
    // [WARNING] caller shoud free error_message with free_pointer
    char* error_message = nullptr;

};
```

V době kompilace je také vynuceno, aby byly typy kompatibilní s C ABI jako POD:

```
static_assert(std::is_trivially_copyable<ParamsResult>::value,
    "ParamsResult must be trivial type to copy!");

static_assert(std::is_trivially_copyable<Int32Result>::value,
    "Int32Result must be trivial type to copy!");

static_assert(std::is_trivially_copyable<RunResult>::value,
    "RunResult must be trivial type to copy!");
```

5.1.5 Programové rozhraní Python (API)

Je pro řešič `hc12qs2_v2` řešeno jako *wrapper* dříve popsané C knihovny `libhc12qap`, která je sama wrapperem původního C++ kódu. Toto bylo zhodnoceno jako nejjednodušší přístup, který nevyžaduje další kompilaci C/C++ modulu pro Python. Pro přímý přístup k C dynamickým knihovnám a pro definici C datových typů poskytuje Python modul `ctypes` ve standardní knihovně. Alternativou, která volí jiný přístup k definicím funkcí a typů je například knihovna CFFI (C Foreign Function Interface for Python) [87]. Pro případnou tvorbu C++ modulu do Pythonu se jeví jako nejpohodlnější možnost použití knihovny pybind11 [88].

Modul `libhc12qap.py` definuje pomocí `ctypes` všechny veřejné C datové typy a C funkce. Jsou definovány tři pomocné funkce a třída `HC12QAP`, která poskytuje pohodlné a bezpečné rozhraní k instanci a spouštění algoritmu. Matice jsou předávány pomocí NumPy `ndarray`.

Relativní cesta k binárnímu souboru dynamické knihovny je podle použitého operačního systému `linux64/libhc12qap.so` nebo `win64\libhc12qap.dll`. Jeho nalezení a načtení zajišťuje kód:

```
__module_path = pathlib.Path(__file__).parent.absolute()

match (platform.system(), platform.architecture()[0]):
    case ('Linux', '64bit'):
        _lib = ctypes.CDLL(str(
                  __module_path / 'linux64' / 'libhc12qap.so'))

    case ('Windows', '64bit'):
        _lib = ctypes.CDLL(str(
                  __module_path / 'win64' / 'libhc12qap.dll'))

    case _:
        raise ValueError('not supported platform')
```

Pro funkce z knihovny, která je načtená v proměnné `_lib`, jsou pak definovány pomocné typy a typy vstupních a výstupních parametrů funkcí. Například funkce `params`:

```
ParamsInstanceDF_p = ctypes.c_void_p

class ParamsResult(ctypes.Structure):
   _fields_ = [("error_code", ctypes.c_int32),
               ("error_message", ctypes.c_char_p),
               ("instance", ParamsInstanceDF_p)]

_lib.params.argtypes = (ctypes.c_uint32,  # uint32_t swaps
                        ctypes.c_uint32,  # uint32_t nElements
                        ctypes.c_uint32,  # uint32_t nSlices
                        ctypes.c_void_p,  # void *DF_src
                        ctypes.c_char_p,  # const char *DF_type_name_in
                        ctypes.c_uint32,  # uint32_t maskOrder
                        ctypes.c_int32,   # int32_t startBit
                        ctypes.c_int32,   # int32_t bitLength
                        )

_lib.params.restype = ParamsResult
```

Ve třídě `HC12QAP` je funkce `params` volána přímo v inicializaci objektu a předtím jsou provedeny kontroly parametrů:

```
def __init__(self, swaps: int, DF: np.ndarray,
             start_bit=0, bit_len=0, mask_order=2):
    self.instance = None
    if len(DF.shape) == 3:
        self.n_slices = DF.shape[2]
    elif len(DF.shape) == 2:
        self.n_slices = 1
    else:
        raise RuntimeError('unsupported DF dimensions')

    if DF.shape[0] != DF.shape[1]:
        raise RuntimeError('DF must be a square matrix '
                           '(or array of square matrices)')

    self.swaps = swaps
    self.mask_order = mask_order
    self.n_elements = DF.shape[0]
    self.n_param = 2 * swaps
    self.n_bit_param = np.ceil(np.log2(self.n_elements))
    self.DF = DF

    params_result: ParamsResult = _lib.params(
        swaps,  # uint32_t swaps
        self.n_elements,  # uint32_t nElements
        self.n_slices,  # uint32_t nSlices
        DF.ctypes.data_as(ctypes.c_void_p),  # void *DF_src
        DF.dtype.name.encode(),  # const char *DF_type_name_in
        mask_order,  # uint32_t maskOrder
        start_bit,  # int32_t startBit
        bit_len  # int32_t bitLength
    )

    if params_result.error_code != 0:
        raise RuntimeError(f'error_message:'
                           f'"{params_result.error_message}"'
                           f', error_code: {params_result.error_code}')

    self.instance = params_result.instance
```

Další veřejné metody třídu `HC12QAP` jsou:

```
def init_tabu_list(self, max_length)
def run(self, init_order=None, init_kernel=None)
def free(self)

@property
def zero_kernel(self)
```

Modul také definuje veřejné funkce:

```
def get_cuda_device_number()
def set_cuda_device_number(device_number)
def is_mem_space_valid()
```

5.1.6 Programové rozhraní Matlab (API)

Pro spojení řešiče `hc12qs2_v2` s Matlabem je použito novější C++ rozhraní pro vytváření MEX souborů (*plug-in* dynamické knihovny s předdefinovaným rozhraním). Je povinné definovat třídu, která dědí `matlab::mex::Function`. Dříve použité C MEX rozhraní vyžadovalo přímo definici funkce `mexFunction`, kterou si C++ MEX definuje samo. Přechod na C++ MEX byl spojen s určitými problémy a bylo nutné správně párovat verze Matlabu a kompilátorů (GCC v Linuxu a MSVC ve Windows) kvůli kompatibilitě ABI. C MEX rozhraní tímto netrpí, protože ABI pro C dynamické knihovny je dáno konvencí operačního systému. Bohužel však neposkytuje bezpečnější datové typy.

Povinné rozhraní je, že třída musí přetížit operátor volání funkce:

```
operator()(ArgumentList outputs, ArgumentList inputs)
```

Funkce má proměnný počet vstupních i výstupních parametrů (počet položek `inputs` a `outputs`), první parametr `mode` je povinný a jeho hodnota definuje význam dalších parametrů dle tabulky 2. Chyby jsou uvnitř ošetřovány vyhozením výjimky `std::runtime_error` v C++, která je chycena a následně je její zpráva předána Matlab funkci `error`, která vyhodí výjimku již v Matlabu.

Protože je instance předávána jako pole `uint8`, je při použití kontrolováno zda počet bytů odpovídá velikosti ukazatele na `ParamsInstanceDF`, což je skutečným obsahem. Dodatečná kontrola však žádná není. Starší obecný HC12 řešič bez QAP s C MEX rozhraním předával místo ukazatele celou strukturu, která na začátku měla „magic header“, neboli identifikační sekvenci několika bytů. V případě potřeby je možné podobnou kontrolu zavést i zde.

Tabulka 2: Parametry MEX funkce Matlab rozhraní HC12qs2_v2 řešiče

mode	**Význam, další parametry a návratové hodnota**
`'get-device'`	Vrací pořadové číslo aktivního CUDA zařízení.
`'set-device'`	Nastavuje aktivní CUDA zařízení pořadovým číslem. Parametry: `deviceNumber`
`'params'`	Vytvoření `instance`, alokace paměti Parametry povinné: `swaps`, `nElements`, `nBitParam`, `DF` Parametry nepovinné s výchozí hodnotou: `nSlices=1`, `maskOrder=2`, `startBit=0`, `bitLength=`*`(vypočítáno)`* Vrací `instance` jako pole `uint8`.
`'set-tabu-list'`	Inicializuje Tabu list. Parametry: `instance`, `maxLength`
`'run'`	Spuštění HC12 algoritmu Parametry: `instance`, `initOrder`, `initKernel`
`'free'`	Uvolnění paměti Parametry: `instance`

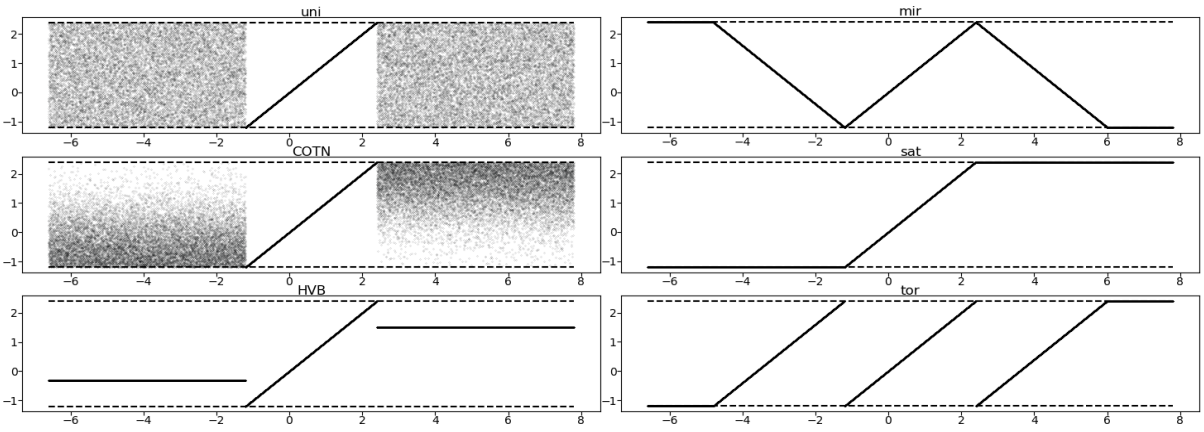
5.2 Diferenciální evoluce s více ostrovními populacemi

Základní verze diferenciální evoluce byly implementovány na CPU v programovacích jazycích Matlab/Octave, Julia, Python (NumPy). Pro implementaci čistě na GPU není základní verze DE příliš vhodná, často je však vhodné využít GPU pro urychlení výpočtu optimalizované funkce. Kompromisem této implementace je, že ostrovní populace mají stejný počet jedinců a sdílejí pseudonáhodná čísla pro genetické operátory mutace a křížení.

V implementaci je konzistentně používáno značení:

- číslo populace *p*, počet populací `n_populations`
- číslo jedince *m*, počet jedinců v každé populaci `n_members`
- číslo dimenze problému *n*, počet dimenzí problému `n_dims`

Strategie korekce chybných řešení (*Strategy of dealing with infeasible solutions – SDIS*) byly implementovány a verifikovány. Korekci pro interval <–1,2; 2,4> znázorňuje obrázek 8. Pro strategie *uni* a *COTN* je třeba použít náhodné číslo. Aby nebylo třeba alokovat kontext cuRAND generátoru pro každou složku každého jedince v každé populaci anebo přidávat synchronizaci na úrovni atomických operací v globální paměti, využijí se náhodná čísla generovaná pro operaci křížení. Ta jsou však sdílená všemi populacemi. Autor se domnívá, že je toto přípustné hlavně z důvodu, že vůbec použití SDIS je výjimečná situace a většina jedinců v generaci by v mutačním vektoru neměla korekci vyžadovat. Strategie *COTN* vyžaduje číslo s normálním rozdělením pravděpodobnosti. Takové číslo je možné získat ze dvojice čísel s rovnoměrným rozložením pravděpodobnosti pomocí Box-Muller transformace. Druhé číslo se získá z dalšího vedlejšího jedince (pro posledního se použije číslo od prvního). Strategie *HVB* se počítá od předchozího řešení, které se pro test volí do středu intervalu.



Obr. 8: Korekce SDIS na intervalu <-1,2; 2,4>

Uložení v paměti, které umožňuje sdružený přístup pro genetické operátory do matice **X** s prvky $x_{p,m,n}$ (ve *warpu* se index mění pro číslo populace *p*), ale selhává pro efektivní transformaci uložení *trial* vektorů v matici **Y** s prvky $y_{p,m,n}$. Matice **Y** musí být uložená tak,

aby sdružený přístup byl možný při výpočtu účelové funkce (ve *warpu* se index mění pro číslo jedince *m*).

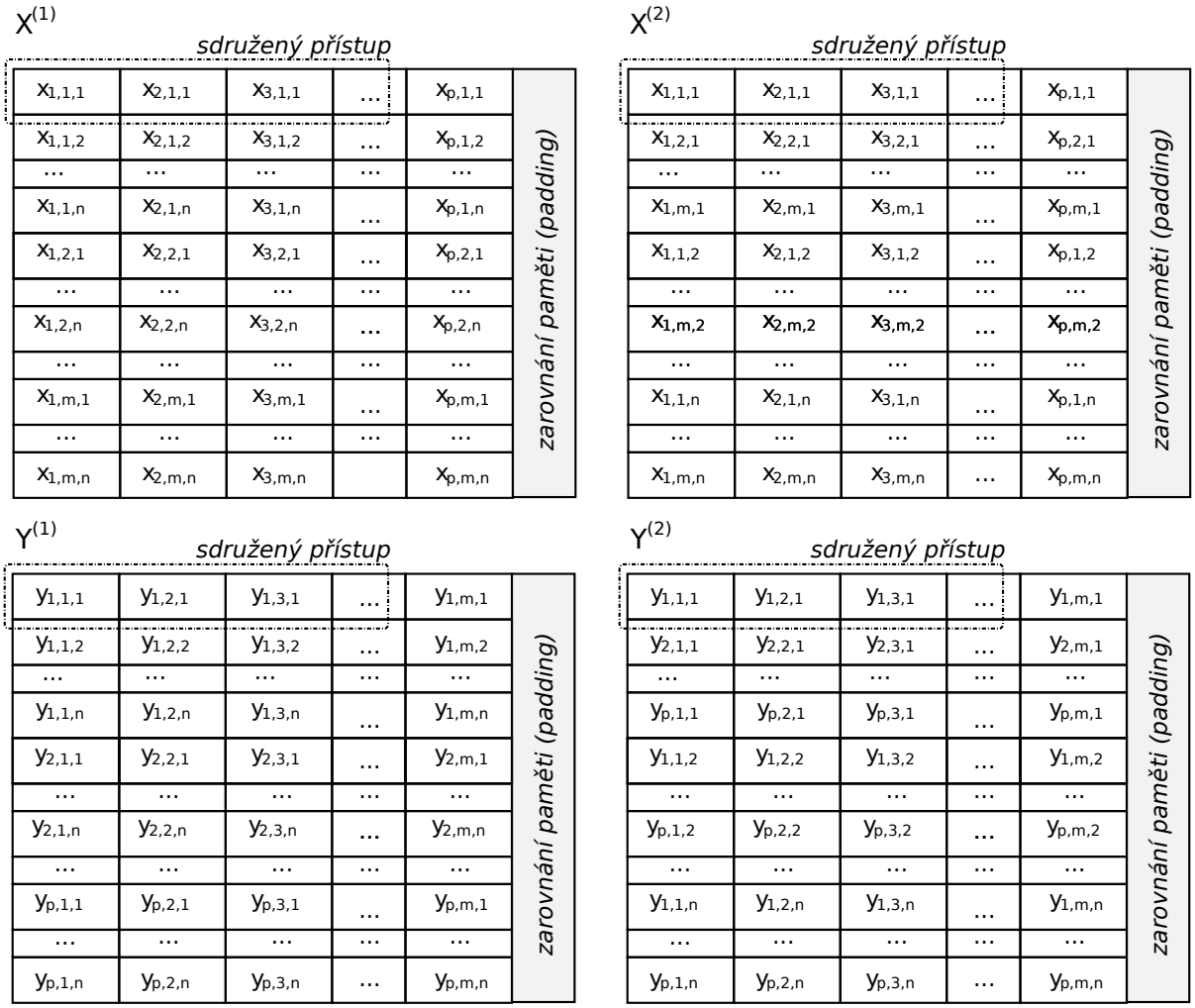
Pro samotné genetické operátory není důležité jestli roste dříve index jedince *m* nebo dimenze *n*. Pro výpočet účelové funkce naopak nezáleží na pořadí růstu indexů jedince *m* a populace *p*. Čtyři možnosti organizace matic v paměti jsou znázorněny na obrázku 9. Liší se v pořadí růstu indexů: $\mathbf{X}^{(1)}$: *p*, *m*, *n*; $\mathbf{X}^{(2)}$: *p*, *n*, *m*; $\mathbf{Y}^{(1)}$: *m*, *n*, p; $\mathbf{Y}^{(2)}$: *m*, *p* , *n*.

Při aplikaci genetických operátorů mutace a křížení je třeba načítat se sdruženým přístupem do paměti vektory z populace X a vytvořené *trial* vektory do matice Y také se sdruženým přístupem do paměti ukládat. Protože se uložení liší právě v tom indexu, který roste nejrychleji, není možné jej provést ve vlákně přímo. Vlákna v CUDA, která jsou ve stejném bloku, se mohou synchronizovat bariérou (vestavěná CUDA funkce `__syncthreads`). Také mohou pracovat se sdílenou (`__shared__`) pamětí, která nevyžaduje sdílený přístup. Díky této vlastnosti je vhodná (a je pro tento účel přímo navržená) pro urychlení algoritmů, které transponují matice či obecněji manipulují se skupinou buněk. Pro nejvyšší výkon je však třeba předcházet konfliktu (současnému přístupu) takzvaných *bank*, které lze v tomto případě zajistit vhodným rozměrem 2D pole a zvětšením druhé dimenze o 1; byl volen rozměr 16×17, skutečně je využito však pouze 16×16, tedy 256 vláken na blok.

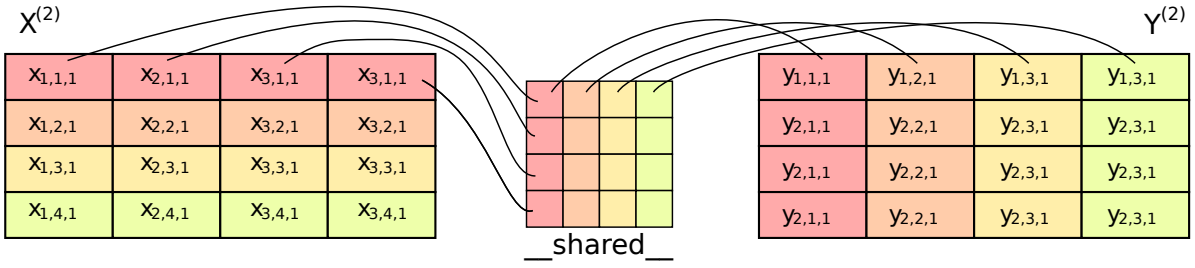
S využitím sdílené paměti je možné dodržet sdružený přístup při čtení **X** pouze u varianty $\mathbf{X}^{(2)}$, ale pro zápis **Y** lze použít obě varianty $\mathbf{Y}^{(1)}$ a $\mathbf{Y}^{(2)}$. V kódu byla použita varianta $\mathbf{Y}^{(2)}$. Výsledná transformace je naznačena na obrázku 10. Celý proces výroby *trial* vektoru je paralelní po složkách vektorů a tak jsou bloky vláken definovány trojdimenzionálně: index *x* odpovídá populaci *p*, index *y* odpovídá jedinci *m*, a index *z* odpovídá dimenzi *n*. Po synchronizační bariéře se však význam indexů *x* a *y* obrací na jedince *m* a populaci *p*. Při vytváření mutačních vektorů $\mathbf{v}_i$, je zaručen sdílený přístup do matice **X** tak, že pro jedince stejného čísla *m* jsou sdílena pseudonáhodná čísla napříč populacemi *p*. Protože jsou populace vytvořeny nezávisle na sobě, je toto sdílení náhodných čísel pro výběr jedinců k mutaci a složek ke křížení přípustné. Vždy je třeba důsledně kontrolovat zda vypočítané indexy nezasahují mimo hranice matic, protože ty nebývají dělitelné velikostí bloku.

Ukázalo se, že ladit CUDA kernely, které pracují se sdílenou pamětí může být velmi problematické, při špatném přístupu do paměti se může sdílená paměť zneplatnit, ale chyba není nijak indikována ani při kompilaci s ladícími informacemi a spuštění v debuggeru cuda-gdb. S korektností výpočtu indexů výrazně napomohla simulace paměťových přístupů, uvedená v **příloze D**.

Matice **X** je alokována s jedním řádkem navíc, kde se v každé populaci ukládá duplicitně nejlepší současné řešení. To se používá pro některé typy mutace, při migraci a také při ukončení algoritmu je zde výsledek ihned k dispozici. Právě pro použití v mutaci to zaručuje sdružený přístup, protože každá populace *p* má jako nejlepší řešení jiného jedince *m*.



Obr. 9: Varianty uložení populace DE v paměti



Obr. 10: Kopírování X do Y s využitím sdílené paměti bloku vláken

Schéma na obrázku 10 ukazuje princip zaručeného sdruženého přístupu i pro mutaci, kde sloupec ve sdílené (`__shared__`) paměti vzniká z několika řádků matice **X**.

Pro první iteraci je třeba náhodně vygenerované populace jedinců **X** pouze překopírovat do **Y**, aby mohlo vzniknout prvotní ohodnocení *fitness funkcí*. Tento kód demonstruje využití sdílené paměti stejně jako je tomu v případě následné výroby *trial* vektorů:

```
template<typename FP, size_t TILE_SIZE>
__global__ void copy_X_Y(
    PitchedPtrGPU<FP> X, PitchedPtrGPU<FP> Y,
    uint32_t n_populations, uint32_t n_members, uint32_t n_dims)
{
    __shared__ FP buffer[TILE_SIZE][TILE_SIZE + 1];
    int p = blockIdx.x * blockDim.x + threadIdx.x;
    int m = blockIdx.y * blockDim.y + threadIdx.y;
    int n = blockIdx.z;
    int X_row = n * n_members + m;
    int X_col = p;


    if(m < n_members && p < n_populations) {
        FP x = X(X_col, X_row);
        buffer[threadIdx.x][threadIdx.y] = x;
    }

    __syncthreads(); // synchronizační bariéra

    int p_blk = blockIdx.x * blockDim.x;
    int m_blk = blockIdx.y * blockDim.y;

    // změna významu indexu x na jedince m, indexu y na populaci p
    int Y_col = m_blk + threadIdx.x;
    int Y_row = p_blk + n_populations * n + threadIdx.y;


    if(Y_col < n_members && p_blk + threadIdx.y < n_populations) {

        // transpozice ze sdílené paměti
        Y(Y_col, Y_row) = buffer[threadIdx.y][threadIdx.x];

    }
}
```

5.2.1 Programové rozhraní C++ (API)

Hlavním rozhraním je šablonová třída `DEMultipop<FP, FPf>` a pomocné typy pro nastavení režimů genetických operátorů, migrace a korekce neplatných řešení (SDIS). Kód odpovídá normě ISO C++17. Šablonový typ `FP` je použitý pro uložení matic X a Y, typ `FPf` je použitý pro uložení výsledku fitness funkce.

```
template<typename FP, typename FPf>
class DEMultipop
{
public:
  // konstruktor se základním nastavením
  DEMultipop(
    unsigned n_populations, unsigned n_members, unsigned n_dims,
    // omezení intervalu v jednotlivých dimenzích
    const std::vector<Interval<FP>> &limits_in,
```

```
    // korekce chybných řešení SDIS v jednotlivých dimenzích
    const std::vector<DEBoundsCorrection> &bounds_correction_in,
    // dodatečná nastavení, která mají výchozí hodnoty
    const OptionsDict& options = OptionsDict{});

  // dodatečná inicializace (lze měnit)
  void init(
    // pointer na fitness funkci – využívá pozdní vazbu
    FitnessFunctorInterface<FP, FPf> *fitness_func_ptr_in,
    DEMutation mutation_in, // mutační stragegie
    DEMigration migration_in, // migrační strategie
    // změna dodatečných nastavení
    const OptionsDict& optionsUpdate = OptionsDict{})

  iteration(); // iterace algoritmu
  // ladící výpis nejlepších fitness populací a nejlepšího řešení celkově
  printBests();
  // vektor nejlepších hodnot fitness v každé populaci
  std::vector<FPf> getBestsFitness();
  // vektor nejlepšího řešení v populaci p
  std::vector<FP> getBest(unsigned p);
  // vektor/jedinec m v populaci p
  std::vector<FP> getX(unsigned p, unsigned m);
};
```

Pomocné typy pro nastavení:

```
struct DEMigration {
  enum class Type {
    None, One2One, RandTarget, PermuteN, One2N
    // (N2One, N2N neimplementováno)
  };
  DEMigration(int period, Type type_);
  bool is_period(int iteration_number);
};

enum class DEBoundsCorrection {
    Uniform,   // uni
    CompleteOnesidedTruncatedNormal,   // COTN
    HalfwayToViolatedBounds,   // HVB
    Mirror,   // mir
    Saturation,   // sat
    Toroidal   // tor

};

enum class DEMutation {
    DE_rand_1,  // DE/rand/1
    DE_best_1,  // DE/best/1
    DE_rand_2,  // DE/rand/2
    DE_best_2,  // DE/best/2
    DE_current_to_rand_1, // DE/current-to-rand/1
    DE_current_to_best_1 // DE/current-to-best/1
};
```

Pro inicializaci pomocí textových řetězců, například z příkazové řádky jsou definovány pomocné funkce. V případě neplatné hodnoty vyhazují výjimku `std::runtime_error`:

```
// name ve formátu DE/method/N
DEMutation DEMutation_by_name(const std::string name);

// name: None, One2One, RandTarget, PermuteN, One2N, N2One, N2N
DEMigration::Type DEMigration_by_name(const std::string name);

// name: uni, COTN, HVB, mir, sat, tor
DEBoundsCorrection DEBoundsCorrection_by_name(const std::string name)
```

Pomocná struktura `OptionsDict` (detail implementace v 5.4.3) obsahuje dodatečná nastavení, která mají výchozí hodnoty (tabulka 3). Pro replikovatelnost výsledků experimentů jsou výchozí semínka pseudonáhodných generátorů konstantní.

Tabulka 3: Význam a výchozí hodnoty polí v `OptionsDict`

Identifikátor	Datový typ	Význam	Výchozí hodnota
`F_coefs_mode`	`string`	Režim nastavení koeficientů *F* (linspace, const_min)	`"linspace"`
`F_coefs_min`	`FP`	Konstantní nebo minimální hodnota *F*	`0.1f`
`F_coefs_max`	`FP`	Maximální hodnota *F*	`1.5f`
`CR_coefs_mode`	`string`	Režim nastavení koeficientů *CR* (const)	`"const"`
`CR_coef_const`	`float`	Konstantní hodnota *CR*	`0.5f`
`init_rng_seed`	`uint64_t`	Semínko PRNG pro inicializaci populací	`123`
`mutation_rng_seed`	`uint64_t`	Semínko PRNG pro operátor mutace	`345`
`crossover_rng_seed`	`uint64_t`	Semínko PRNG pro operátor křížení	`457`
`migration_rng_seed`	`uint64_t`	Semínko PRNG pro migrační strategii	`581`

Fitness funkce je předávána pomocí ukazatele na šablonovou abstraktní bázovou třídu. U testovací sady funkcí se používá metoda `bound` pro nastavení rozsahu hodnot před konstrukcí `DEMultipop` objektu. Uživatel ji ve vlastní funkci použít nemusí, `DEMultipop` objekt ji sám nevolá:

```
template<typename FP, typename FPf>
class FitnessFunctorInterface
{
public:
    virtual void operator()(
        PitchedPtrGPU<FP> X, PitchedPtrGPU<FP> Y,
        PitchedPtrGPU<FPf> Fx, PitchedPtrGPU<FPf> Fy,
        uint32_t n_populations, uint32_t n_members, uint32_t n_dims
        ) = 0; // pure virtual function
    virtual Interval<FP> bounds(uint32_t dim) = 0;
    virtual ~FitnessFunctorInterface() = default;
};
```

Třídy, které dědí `FitnessFunctorInterface` by měly dodržovat Liskovové princip zastoupení (Liskov substitution principle). Sada testovacích funkcí je implementována jako dvojice třídy a příslušejícího CUDA kernelu. Pro CEC funkce je kvůli verzi s transformací

přidána do hierarchie třída `CECfitness`, která přidává povinnost vracet informaci o svém čísle (4 až 10) a výpočtu z transformovaných hodnot `Yt`:

```
template<typename FP, typename FPf>
struct CECfitness: FitnessFunctorInterface<FP, FPf>
{
    virtual int get_func_num() = 0;
    virtual void call_transformed(
        PitchedPtrGPU<FP> X, PitchedPtrGPU<FP> Y, PitchedPtrGPU<FP> Yt,
        PitchedPtrGPU<FPf> Fx, PitchedPtrGPU<FPf> Fy,
        uint32_t n_populations, uint32_t n_members, uint32_t n_dims) = 0;
    virtual ~CECfitness() = default;

};
```

O samotnou transformaci `CECfitness` funkce se stará třída `FitnessCECTransform` s příslušným CUDA kernelem. Protože dědí `FitnessFunctorInterface`, ale obaluje instanci `CECfitness` (drží na ni ukazatel), není třeba aby byla `FitnessFunctorInterface` děděna jako virtuální báze. Její konstruktor vyžaduje ukazatel `CECfitness` funkci k obalení a informaci jestli ukazatel vlastní (má uvolnit v destruktoru), případně lze dopředu předat i informace potřebné k alokování matice `Yt`:

```
FitnessCECTransform(CECfitness<FP, FPf> *wrapfunc, bool owns_func_ptr)

FitnessCECTransform(CECfitness<FP, FPf> *wrapfunc,
        uint32_t n_populations, uint32_t n_members, uint32_t n_dims,
        bool owns_func_ptr):
```

Pro konstrukci testovací funkce názvem v textovém řetězci je definována pomocná funkce. V případě neplatné hodnoty vyhazuje výjimku `std::runtime_error`:

```
// name: CEC04 až CEC10, CEC04t až CEC10t, F1 až F12
template<typename FP, typename FPf>
std::unique_ptr<FitnessFunctorInterface<FP, FPf>>
    make_fitness_by_name(const std::string &name);
```

Pro kód CUDA kernelu jsou definovány dvě makra, která se mají použít před a po výpočtu. První definuje konstantní proměnné *m* a *p*. Druhé provede selekci (uložení *trial* vektoru z Y do původní matice X, pokud má lepší ohodnocení), předpokládá se existence proměnné `FPf f`, která obsahuje výsledek fitness funkce pro jedince *m* v populaci *p*.

```
#define DEMULTIPOP_FITNESS_m_p \
    int m = blockIdx.x * blockDim.x + threadIdx.x; \
    int p = blockIdx.y * blockDim.y + threadIdx.y;

#define DEMULTIPOP_FITNESS_COPY_BETTER_Y2X \
    Fy(m, p) = f; \
    if(f < Fx(m, p)) { \
      Fx(m, p) = f; \
      int X_col = p, Y_col = m; \
      for(int n=0; n<n_dims; n++) { \
        int X_row = n * n_members + m, Y_row = n_populations * n + p; \
           X(X_col, X_row) = Y(Y_col, Y_row); \
    }  }
```

5.3 Implementace zvolených optimalizačních úloh

Vhodně zvolené optimalizační úlohy v následující podkapitole „Testovací funkce“ a aplikačně orientované inženýrské problémy kombinatorické optimalizace prezentované v následujících podkapitolách, zkráceně „QAP“ a „SAT“, ukazují komplexnost řešení i robustnost a efektivitu prezentovaných metod a jejich netriviálních implementací.

5.3.1 Testovací funkce s reálnými argumenty

Všechny funkce F1 až F12 a CEC04 až CEC10 byly implementovány ve 4 verzích (příklad implementací funkce CEC07 je uveden v **příloze B**):

- Matlab, nativní vektorizace přes sloupce (CPU)
- Python+NumPy, nativní vektorizace přes řádky (CPU)
- CUDA C, sdružený přístup přes sloupce odpovídající HC12 řešiči (GPU).
- CUDA C, pro více populační DE (GPU)

Bylo ověřeno, že všechny implementace počítají podobné výsledky. U Matlabu není zdokumentováno v jakém pořadí dochází k výpočtům u operací násobení matic a suma prvků vektoru/matice. U NumPy je dostupný zdrojový kód; suma prvků vektorů zde vede k paralelní binární redukci. Ta je využita kvůli předpokladu podobné velikosti prvků vektoru a pak jsou i mezisoučty podobně velká čísla a tak dochází k menšímu vlivu zaokrouhlování než při postupné sumaci do akumulátoru (či skupiny akumulátorů). Současná implementace v CUDA C binární redukci pro sumu prvků vektoru neprovádí, ale přičítá prvky postupně do akumulátoru.

Pro přímé spouštění CUDA C funkcí pro verifikaci jejich správnosti a rozdílům výsledků oproti ostatním implementacím byla rozšířena CUDA MEX funkce pro obecný HC12 řešič `geatbx_cec_bundle`. Pokud je prvním parametrem matice **X** typu `REAL` (makrem definován jako `float` nebo `double`) a druhý parametr je název testovací funkce, je sémantika stejná jako verze pro Matlab:

```
X = rand(1000, 10);
Y = geatbx_cec_bundle(X, 'CEC05t');

// s využitím anonymní funkce
CEC05t = @(X) geatbx_cec_bundle(X, 'CEC05t');
Y = CEC05t(X);
```

5.3.2 Problém kvadratického přiřazení (QAP)

CUDA C implementace QAP spojená s HC12 řeší v hlavním CUDA kernelu vyváří permutaci prohazováním dvojic parametrů v lokální paměti vlákna. K tomu potřebuje vlákno vždy pouze dvě hodnoty z matice **I**, která se tak nemusí celá ukládat. To je nejméně efektivní část implementace. Jakmile je testovaná permutace známá, pokračuje se k výpočtu ohodnocení pomocí sdružené matice **T** pro symetrický QAP.

Každé vlákno uloží výsledné ohodnocení pouze do sdílené paměti bloku (`__shared__`) a v rámci bloku se synchronizují v kritické sekci pomocí bariéry (vestavěná CUDA funkce `__syncthreads`). Pouze první vlákno bloku (`threadIdx.x=0`) vybere ze sdílené paměti nejlepší ohodnocení bloku a uloží je do globální paměti spolu s indexem příslušného řádku. Následuje další synchronizace v kritické sekci v rámci bloku a pokud se jedná o vlákno prvního řádku v prvním bloku, uloží svoji permutaci do globální paměti pro případ, že je iterace poslední. V dalším kernelu se v prvním vlákně najde nejlepší ohodnocení ze všech bloků a pro vítězný řádek se znovu provede převod XOR kernelu s maskou a vzniká tak kernel nový. Tímto opakováním výpočtu odpadá potřeba ukládat matici **B**. Pokud je nejlepším řádek první, tj. starý kernel, jedná se o závěrečnou iteraci a vrací se již uložená permutace.

```
// dynamicky alokovaná sdílená paměť má stejný typ
// pro všechny instance šablony
extern __shared__ char smem[];
REAL* localFBuf = (REAL*)smem; // přetypování na typ šablony
// index vlákna
unsigned row = blockIdx.x * blockDim.x + threadIdx.x;

// permutace v paměti vlákna
IORD_TYPE order[MAX_nElements];
for(unsigned e=0; e<nElements_cd; e++)
    order[e]=initOrder_cd[e];

REAL F = CC_type::REAL_MAX; // pro vlákna, která se neúčastní výpočtu

if(row<rows_cd) { // vlákna účastnící se výpočtu
    unsigned lastI=0;
    for(unsigned par = 0; par<pars_cd; ++par)
    {
       EL_TYPE G = K_cd[par] ^ PITCHED(M, row, par, M_pitch_cd, EL_TYPE);
       // (…) převod G na I; Gray2Binary (…)

       if(par%2) { // každý druhý parametr dojde k výměně
           unsigned p1=lastI % nElements_cd;
           unsigned p2=I % nElements_cd;
           IORD_TYPE tmp=order[p1];
           order[p1]=order[p2]; order[p2]=tmp;
       }
       lastI = I;
    }
    F = 0.0; // inicializace fitness
    // (…) výpočet QAPu, součet všech D * F
    localFBuf[threadIdx.x] = F;  // uložení do sdílené paměti bloku
}
__syncthreads(); // bariéra, localFBuf je naplněn
// (…) první vlákno v bloku najde nejlepší fitness v localFBuf
__syncthreads(); // nejlepší F v bloku a jeho index (row) bylo nalezeno
// (…) první vlákno v prvním bloku uloží permutaci pro případ,
//     že je iterace poslední
```

5.3.2.1 Benchmark pro CUDA a ISPC

Pro porovnání bylo vyvinuto několik implementací řešiče HC12 s překódováním *qapswap2*. Pro GPU je s CUDA C mírně upravený řešič, aby bylo srovnání co nejpřímější. Převážně jde o využití 32 bitových celých čísel tam, kde obecný CUDA řešič volí co nejmenší datové typy podle velikosti problému a délky bitového řetězce HC12, což nemusí být nutně nejvýhodnější. Instance problému byla zvolena tho150 z QAPLIB. Důvody jsou: matice jsou symetrické, rozměr problému není mocnina dvou (je třeba optimalizovat modulo), matice vzdáleností a toků obsahují poměrně málo nul, problém je obtížný a není ověřena optimálnost řešení. Pro CPU jsou implementovány 4 varianty: *Normal*, *NormalBS*, *ISPCqap* a *ISPCtasks*.

Normal je referenční implementací. Jednovláknový kód, případná vektorizace je ponechána na schopnostech kompilátoru *clang LLVM*.

NormalBS má stejný kód výpočtu jako u varianty *Normal*. Ten je vložený do *lambda* funkce s rozdělením do více úloh. Tyto úlohy jsou předávány k výpočtu skupině předem spuštěných vláken (*thread pool*), využívá `BS::thread_pool_light` [69]:

```
auto task = [&, this](int taskIdx) {
  // ( ... inicializace)
  int row_max = std::min((taskIdx+1)*rowsPerTask, rows);
  for(int row=taskIdx*rowsPerTask; row<row_max; row++) {
    /* ( ... výpočet ) */
  }
  // ( ... uložení výsledků )
}

for(int i=0; i < n_tasks; i++) {
  normal_thread_pool.push_task(task, i); // předání úloh vláknům
}

normal_thread_pool.wait_for_tasks(); // čekání na konec výpočtu
```

ISPCqap varianta vytváří permutace (`order`) skalárně, výpočet QAP je paralelizován pro tuto jednu permutaci. Pro paralelizaci na jednotlivá jádra CPU je také použita skupina spuštěných vláken (*thread pool*) `BS::thread_pool_light`. Sdružená matice T (`DF`) je transponovaná T^T (`DFt`) a tak je třeba pouze jediná *gather* operace, když je SIMD vektorizována vnitřní smyčka:

```
varying int32 result_v; // vektorová hodnota (v simd registru)
for(uniform int32 i = 0; i<nElements; i++) {
  foreach(j = i+1 ... nElements) {
    uniform int32 iD=i, D=j, iF=order[i];
    varying int32 jF=order[j];
    // swap if iD < jD; resp. iF<jF
    varying int32 iFA=iF*(int32)(jF⩽iF) + jF*(int32)(jF>iF);
    varying int32 jFA=iF*(int32)(jF>iF) + jF*(int32)(jF⩽iF);
    varying int32 d = DFt[iD*nElements + jD];
    varying int32 f = DFt[iFA*nElements + jFA];
    result_v += d * f;
  }
}
result = 2 * reduce_add(result_v); // suma složek simd registru
```

ISPCtasks využívá pro paralelizaci na jednotlivá jádra CPU *thread pool* dodávaný přímo v příkladech ISPC `tasksys.cpp`. Lze tak použít v kódu klíčová slova `task` a `launch`. Pro SIMD je paralelizována v každém *tasku* vnější smyčka. Pro větší kontrolu nad výpočtem není použitá konstrukce `foreach`, ale index řádku (jedince) je počítám pomocí vestavěných proměnných `programCount` (šířka SIMD registrů) a `programIndex` (číslo SIMD *lane*). Využívá se též vestavěná proměnná pořadového čísla úlohy `taskIndex0` (podobně jako v případě CUDA bloků lze definovat i více rozměrů indexů úloh). Kontrola překročení počtu řádků (jedinců) pak stejně jako v CUDA C musí být prováděna ručně (kostra *tasku*):

```
uniform int row_offset = taskIndex0 * rowsPerTask;
uniform int row_end = row_offset + rowsPerTask;
if(row_end > rows) { row_end = rows; }

// výsledek QAPu; hodnota nejlepšího a jeho index pro každou SIMD lane
varying int32 F = 0, v_min_F = 2147483647, v_min_row_idx = 0;

for(uniform int32 row_base = row_offset;
    row_base < row_end; row_base += programCount) {

  varying int32 row_v = row_base + programIndex;
  if(row_v < row_end) {
    // (…) výpočet Iv = Gray2Bin(K xor M) ;
    I[(taskIndex0*nParam + par) * I_pitch + programIndex] = Iv;

  }

  // rozkopírování výchozí permutace
  for(uniform int32 ei=0; ei<nElements; ei++) {
    for(uniform int32 ri = 0;
         ri < programCount && row_base+ri < row_end; ri++) {
      orders[(taskIndex0*nElements + ei) * orders_pitch + ri] =
         initOrder[ei];
  } } // konec for ei

  if(row_v < rows) {
    // (…) datově paralelní výpočet QAP z permutací orders
    F *= 2;  // výsledek symetrického QAPu

    if(F < v_min_F) { // ukládání lepších řešení
        v_min_F = F;
        v_min_row_idx = row_v;
} } }  // konec for row_base

// nejlepší řešení a jeho index pro task
uniform int32 min_F = extract(v_min_F, 0); // složka 0 ze SIMD registru
uniform int32 min_row_idx = extract(v_min_row_idx, 0);
for (uniform int i = 1; i < programCount; ++i) {
  uniform int32 f = extract(v_min_F, i); // složka i ze SIMD registru
  // zachováním shodného chování jako má CUDA implementace:
  // při rovnosti hodnot se používá ta, která má menší index řádku
  if(f < min_F ||
     (f == min_F && extract(v_min_row_idx, i) < min_row_idx)) {
    min_F = f;
    min_row_idx = extract(v_min_row_idx, i);
} }

FBuf[taskIndex0] = min_F;    idxBuf[taskIndex0] = min_row_idx;
```

Skalární kód (a CUDA kód také) potřebuje z matice **I** vždy pouze dva po sobě jdoucí prvky, které stačí držet v lokálních proměnných, které mapuje kompilátor přímo na registry. U varianty *ISPCtasks* se však nepodařilo tuto optimalizaci použít a musí si podle šířky SIMD registrů ukládat příslušný počet řádků matice **I**, aby je mohl využít k paralelnímu vytváření permutací. Na rozdíl od typických algoritmů vhodných pro SIMD implementace, např. násobení matic, zde nepomáhá rozdělení do bloků kvůli dvojité nepředvídatelnosti přístupu do paměti: při výrobě permutací prohazováním prvků v poli a pak přístup do matice toků *F*.

Protože ani rozšíření instrukční sady AVX2 a AVX512 neobsahují instrukci SIMD pro operaci modulo (zbytek po celočíselném dělení), je ve všech benchmarcích rozměr úlohy N dán jako konstanta známá v době překladu a tedy pro problém jiné velikosti je třeba buď program upravit a překompilovat nebo přidat `switch` pro seznam podporovaných hodnot *N*. Přestože ISPC hlásí při kompilaci: *„Performance Warning: Modulus operator with varying types is very inefficient,“* bylo ověřeno, že LLVM výslednou optimalizací zvládne toto modulo konstantou vyjádřit jako příslušnou ekvivalentní sekvenci aritmetických a bitových operací:

C kód

```
unsigned m150(
    unsigned Iv)
{
    return Iv % 150;
}
```

x86-64 assembler (LLVM clang 16)

```
    mov   eax, edi
    imul  rax, rax, 458129845
    shr   rax, 36
    imul  edx, eax, 150
    mov   eax, edi
    sub   eax, edx
    ret
```

ISPC kód

```
varying uint32 m150(
   varying uint32 Iv)
{
    return Iv % 150;
}
```

x86-64 assembler (ISPC 1.21.0+LLVM)

```
.LCPI1_0:
    .long   458129845 ; stejné konstanty
.LCPI1_1:
    .long   150

m150___vyu:
    vpshufd ymm1, ymm0, 245
    vpbroadcastd ymm2, dword ptr [rip+.LCPI1_0]
    vpmuludq  ymm1, ymm1, ymm2
    vpmuludq  ymm2, ymm0, ymm2
    vpshufd   ymm2, ymm2, 245
    vpblendd  ymm1, ymm2, ymm1, 170
    vpsrld  ymm1, ymm1, 4
    vpbroadcastd ymm2, dword ptr [rip+.LCPI1_1]
    vpmulld ymm1, ymm1, ymm2
    vpsubd  ymm0, ymm0, ymm1
    ret
```

V případě použití proměnné místo konstanty však varování ISPC platí a generovaný kód je sekvenční s postupným kopírováním jednotlivých složek vektorového registru do běžného registru a výpočtu pomocí instrukce DIV, která je na běžných x86 CPU jedna z nejpomalejších (počet potřebných cyklů) a zároveň nepodporuje *pipelining*:

C kód

```
unsigned md(
   unsigned Iv,
   unsigned d)
{
    return Iv % d;
}
```

x86-64 assembler (LLVM clang 16)

```
    mov     eax, edi
    xor     edx, edx
    div     esi
    mov     eax, edx
    ret
```

ISPC kód

```
varying uint32 md(
  varying uint32 Iv,
  uniform uint32 d)
{
    return Iv % d;
}
```

x86-64 assembler (ISPC 1.21.0+LLVM)

```
md___vyuunu:
    vextracti128    xmm1, ymm0, 1
    vpextrd eax, xmm1, 1
    xor     edx, edx
    div     edi
    mov     ecx, edx
    vmovd   eax, xmm1
    xor     edx, edx
    div     edi  // obyčejné skalární dělení
    vmovd   xmm2, edx
 ; (…)  opakuje se 7 dalších bloků extrahující
 ; prvky vektorových registrů a DIV instrukce
    vinserti128   ymm0, ymm0, xmm1, 1
    ret
```

5.3.3 Problém splnitelnosti Booleovské formule (SAT)

Pro SAT je specifické, že jde o rozhodovací problém a tedy formulace jako účelová funkce hodnotí počet splněných dílčích formulí. To lze využít buď jako pomocnou heuristiku v jiném řešiči a nebo pro specifické úlohy SATu podobné (aplikace například v bioinformatice). Metaheuristikou není možné dokázat případnou nesplnitelnost a výzkum se dále touto variantou zabývá. Vhodnou účelovou funkcí lze také optimalizovat například pro nejmenší počet pravdivých literálů.

Implementace výpočtu obecného SAT problému – klauzule obsahují různý počet literálů a jejich negací. Klauzule jsou kódovány klasicky jako vektory celých čísel *i*, kde |*i*| je index literálu $x_{|i|}$ a záporné hodnoty značí negaci literálu $\neg x_{|i|}$. Datový typ se volí co nejmenší celočíselný (znaménkový; *signed integer;* `int`**`N`**`_t`), aby obsáhl hodnoty od –max|*i*| do max|*i*| . Vektor hodnot literálů je uložen jako jednotlivé bity ve vektoru složeném z prvků 32-bitového nezáporného celočíselného (bez-znaménkového, *unsigned integer;* `uint32_t`) datového typu. Například pro 164 literálů by byl zvolen typ `int16_t` pro uložení klauzulí (rozsah hodnot typu `int8_t` je pouze od –127 do 128 a žádný typ mezi není definován) a pro vektor hodnot literálů 5-ti prvkový vektor `uint32_t` (v pátém zůstane 30 bitů nevyužito).

Byly implementovány dvě varianty funkcí vyhodnocení SAT na GPU pomocí platformy CUDA, otestované na datech generovaných v jednoduché implementaci v Pythonu:

1. **MCSX** je paralelní pro jednotlivé klauzule a počítá pouze jeden vektor hodnot literálů **x**. Protože klauzule nejsou uloženy v řádcích *pitched* paměti, může docházet

k horšímu výkonu paměťového systému. Výpočet je rozdělen na dvě fáze, které musí být spuštěny jako samostatné CUDA kernely:

I. Paralelní vyhodnocení všech dílčích literálů v klauzulích a jejich počtu.

II. Vyhodnocení splnění celého SATu.

2. **DXAC** je paralelní pro vektory hodnot literálů **x**, ale jednotlivé klauzule se pro každý vektor počítají sekvenčně. Vektory hodnot literálů **x** jsou uloženy v *pitched* paměti, stejně tak výstupy algoritmu o počtech platných literálů v jednotlivých klauzulích. Aby bylo GPU dostatečně vytěžované, je třeba aby bylo jednotlivých vektorů větší množství. Tedy při napojení na metaheuristiku je třeba, aby generovala velké množství nových potenciálních řešení (například HC12 a varianty).

Implementace obou metod s bitovými optimalizacemi jsou v **příloze C** a **Příloha E** obsahuje skript v jazyce Python pro generování testovacích dat pro verifikaci prototypu CUDA implementace.

5.4 Pomocná funkcionalita

Implementace algoritmů na GPU v CUDA C při použití C++ si vyžadoval pohodlnější a bezpečnější způsob pro práci se surovými ukazateli (*raw pointers*) do paměti na GPU. Po sléze sice vyšla knihovna s šablonovými kontejnery Trust (nyní součást CCCL), ale nepodporovala starší karty, které bylo stále nutné používat. Automatický management kopírování paměti mezi CPU a GPU v tzv. *CUDA Unified memory* také nebyl k dispozici.

5.4.1 gpupointer.h

V hlavičkovém souboru `gpupointer.h` jsou definovány pomocné šablonové třídy pro obalení (wrapping) ukazatelů do GPU paměti na CPU. Také je zjednodušeno použití 2D polí se zarovnáním, tzv. „*pitched pointers*“ alokované pomocí funkce `cudaMallocPitch`. V pozdější verzi použité při vývoji DE také šablonová struktura, která „*pitched pointer*“ předává do CUDA kernelu jako parametr. Přetěžuje operátor volání funkce pro přístup k prvkům, která obaluje definované makro `PITCHED`.

U funkcí a tříd je uvedena pouze deklarace a veřejné rozhraní. Funkce označené pomocí `__global__` jsou CUDA kernely, `__device__` jsou označení funkcí, které lze volat pouze z kernelů či z jiných `__device__` funkcí.

Makra výpočtu offsetu do zarovnané paměti. Řádek a sloupec je pro matici uloženou po sloupcích (tj. při lineárním průchodu polem `row` roste první a pak se zvyšuje `column`):

```
#define PITCHED_P(P, row, column, pitch, TYPE)  \
    (((TYPE*)((char*)(P) + (column) * (pitch))) + (row))

#define PITCHED(P, row, column, pitch, TYPE)  \
    *PITCHED_P(P, row, column, pitch, TYPE)
```

Šablonová struktura pro předání „*pitched pointeru*“ do kernelu na GPU (neumožňuje kontrolu mezí, pouze správné přepočítání offsetu):

```
template <typename T>
struct PitchedPtrGPU {
    T *p;  // raw pointer
    size_t pitch;  // zarovnání paměti

    __device__ __forceinline__ T& operator()(size_t row, size_t column) {
        return PITCHED(p, row, column, pitch, T);
    }
};
```

Šablonová funkce pro paralelní vyplnění prvků pole jednou hodnotou (CUDA knihovna obsahuje pouze funkci pro vylnění nulami):

```
template <typename T>
__global__ void gpu_fill_1D(T* V, unsigned n, T value)
```

Šablonová funkce pro paralelní vyplnění prvků 2D pole se zarovnáním („pitched pointer“) jednou hodnotou:

```
template <typename T>
__global__ void gpu_fill_2D(
    PitchedPtrGPU<T> M, unsigned rows, unsigned cols, T value);
```

Šablonová třída pro alokaci a správu paměti na GPU. Každá instance má povinné jméno, které je vypisováno při ladění definováním makra a v chybových zprávách výjimek. Výjimkami jsou ošetřeny stavy, kdy paměť nebyla explicitně alokována nebo pokud operace nedává smysl. Například počet řádků metodou `rows` nelze získat u 1D pole. Výhodou použití také je, že pokud dojde k přealokování a tedy změně surového ukazatele, reference na příslušný `GPUPointer<T>` zůstávají stále platné. Výběr z rozhraní, které je aktivně využíváno v implementaci algoritmů:

```
template <typename T>
class GPUPointer {
public:
    GPUPointer(const std::string& name); // konstruktor
    // alokace 1D pole; pokud bylo již alokováno paměť uvolní
    void alloc(size_t size, const T* init=nullptr);

    // alokace 2D pole zarovnané pitched paměti; také uvolní paměť
    void allocPitch(size_t rows, size_t cols, void *init=nullptr);

    size_t size() const;  // počet prvků
    size_t sizeByte() const; // velikost prvků v bytech
    size_t sizeByteReal() const // skutečná velikost v paměti

    size_t rows() const; // počet řádků 2D pole
    size_t cols() const; // počet sloupců 2D pole
    size_t pitch() const; // počet bytů se zarovnáním paměti

    T* p() const; // surový ukazatel do GPU paměti
    T* operator()() const; // surový ukazatel do GPU paměti
    PitchedPtrGPU<T> pp() const; // předání 2D pole do kernelu
```

```
    // lineární přístup k prvkům, může pracovat i nad 2D polem,
    // kontroluje meze a pracuje pouze s kopií
    T& operator[](const size_t location);
    T& at(size_t location); // stejné jako operátor []
    T& at(size_t row, size_t col); // čtení prvku 2D pole

    // kopie do CPU paměti, může alokovat pokud je dst=nullptr
    T* copyToCpu(T* dst=nullptr) const;
    std::vector<T> toVector() const; // kopie jako std::vector
    void copyFromCpu(const T* m) const; // kopírování do GPU paměti

    // nastavení jedné hodnoty (vhodné pro ladění a pokusy)
    void setAt(size_t location, const T& value);
    void zero(); // nastavení všech prvků na 0
    void fill(T value); // nastavení všech prvků na hodnotu

    void free(); // ruční uvolnění paměti

    // výměna POUZE surových pointerů; nekontrolují se rozměry!
    // (použití pro prohození polí indexů tabu listu HC12)
    static void swap(GPUPointer<T> &a, GPUPointer<T> &b);
};
```

Lze definovat makro pro zapnutí výpisů paměťových operací nad `GPUPointer<T>` na CPU:

```
#define GPUPOINTER_REPORT_MEMORY_OPS
```

5.4.2 cerror.h

Protože CUDA C knihovna neobsahuje funkci pro převod chybového kódu na textovou chybovou zprávu je tato funkcionalita doplněna; texty jsou čerpány z dokumentace. K převodu slouží funkce v hlavičkovém souboru `cerror.h`:

```
const char* cudaErrorString(cudaError_t ec)
```

Některé chyby, například `cudaErrorLaunchFailure`, způsobují zneplatnění celého paměťového prostoru GPU pro přiřazený CPU procesu. Proto funkce `cudaErrorString` při převodu kódu chyby také nastavuje globální proměnnou `g_isMemSpaceValid`, kterou vyžaduje definovanou. Jeden vybraný modul programu (typicky ten s hlavní funkcí `main`) musí definovat proměnnou `bool g_isMemSpaceValid`. Třída `GPUPointer<T>` hlídá, aby v destruktoru neuvolňovala GPU paměť pokud není paměťový prostor validní. Jinak by nastal nepříjemný pád programu namísto rozumného výpisu chybových zpráv a obsluhy výjimek.

Pro výpisy konfigurace spuštění CUDA kernelu jsou definovány pomocné funkce:

```
std::string launchParsStringAndInfo(const std::string& kernelName,
    unsigned blocks, unsigned tpb, unsigned smemBytes)

std::string launchParsStringAndInfo(const std::string& kernelName,
    dim3 blocks_d, dim3 tpb_d, unsigned smemBytes)
```

5.4.3 utils.h (DE)

Diferenciální evoluce používá pomocnou funkcionalitu z `utils.h`.

Výjimka `bad_alloc_message` umožňuje udat důvod neúspěšné alokace. To základní výjimka `std::bad_alloc` nemůže, protože nemá zaručen dostatek paměti pro zprávu. Nová výjimka to však předpokládat může, protože se používá pouze pro alokaci velkých polí.

Šablonová funkce pro výpočet 2^n–1, která nepřeteče až do limitu datového typu:

```
template<typename T>
inline T safe_pow2m1(int power)
```

Rozdělení na bloky:

```
inline size_t split_to_blocks(size_t n_elements, size_t block_size)
```

Šablonová funkce pro rozdělení intervalu na stejnoměrné úseky:

```
template<typename FP>
std::vector<FP> linspace(FP a, FP b, size_t n)
```

Šablonová funkce na zjištění přítomnosti položky v kontejneru (využívá `std::find`):

```
template<typename TC, typename TI>
bool contains(const TC& container, const TI& item)
```

Asociativní kontejner pro snadné předávání dodatečných nastavení. `std::map` se dědí jako neveřejné a tak se zabraňuje nevhodnému použití operátoru indexace `[]`, který neexistující položky rovnou přidává výchozím konstruktorem. `std::variant` je lépe ošetřená alternativa k `union` typům:

```
using OptDictItem =
   std::variant<int, uint64_t, float, double, std::string>;

struct OptionsDict: std::map<std::string, OptDictItem>
{
  template<typename T>
  void set(const std::string &name, const T &value);

  template<typename T>
  T get(const std::string &name) const;

  template<typename T>
  T get(const std::string &name, T default_value) const
};
```

Definice intervalu pro inicializaci populací a kontrolu mezí s opravou (SDIS), `FP` by měly být pouze základní číselné typy s plovoucí desetinnou čárkou (`float`, `double` či jiné podporované CUDA C), protože `std::vector<FP>::data()` se přenáší přímo na GPU jako pole `FP` pomocí `reinterpret_cast`:

```
template<typename FP>
struct Interval
{
  FP a, b;
};
```

6 EXPERIMENTY

6.1 Diferenciální evoluce s více ostrovními populacemi

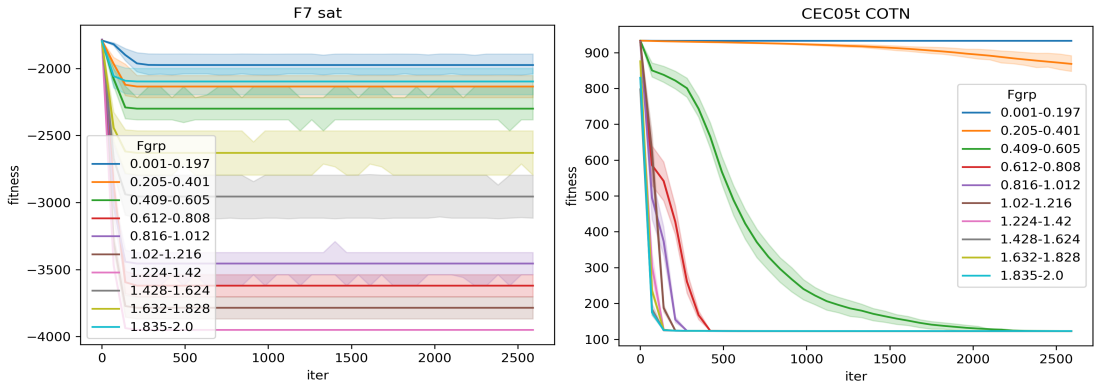
Dále se rozšiřuje značení definované v kapitole 5.2: počet populací `n_populations` = N_p, počet jedinců v každé populaci `n_members` = N_m, počet dimenzí problému `n_dims` = N_n.

Principiálně lze pro každou populaci zvlášť nastavit koeficienty *F*, *CR*, mutační operátor a SDIS (strategie opravy nevyhovujících řešení). Pro celý algoritmus se volí strategie migrace a její perioda. Nastavení těchto parametrů je z hlediska rychlosti dosažení co nejlepšího řešení (či možnosti jej vůbec dosáhnout) pro každou účelovou funkci individuální. Současná implementace spouštění testovacích funkcí z příkazové řádky umožňuje variabilitu pro jednotlivé populace v rámci koeficientu *F*.

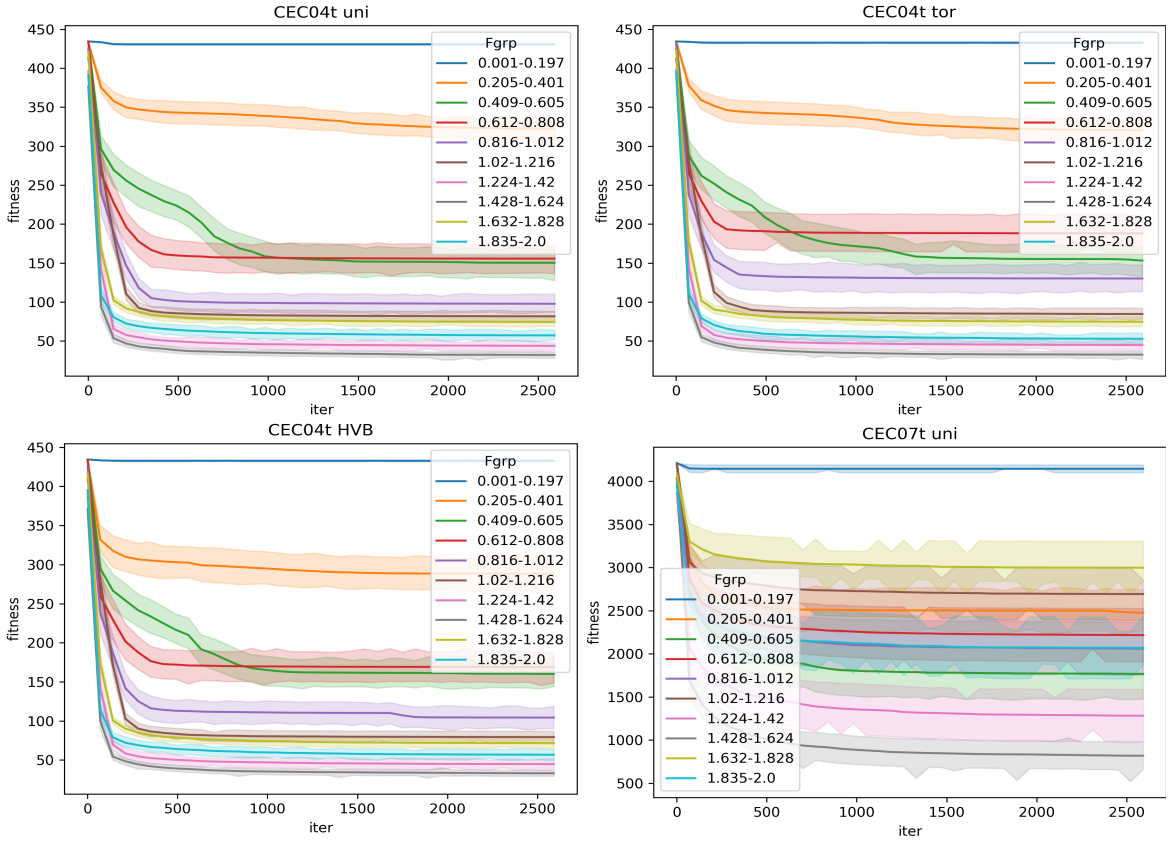
První porovnání se věnuje zkoumání vlivu *SDIS* na konvergenci při určitém koeficientu F_p. Migrace je zakázána (*None*), každá populace *p* je nezávislá. (Výraz pro F_p odpovídá použití funkce `linspace`). Testovací účelové funkce jsou použity CEC04t až CEC10t a F7.

$$\begin{gathered} SDIS \in \{sat, mir, tor, HVB, uni, COTN\} \\ F_p \subset (0{,}001; 2) \rightarrow F_p = 0.001 + \frac{p(2-0.001)}{N_p - 1} \\ CR = 0{,}7 \;\; ; \;\; N_p = 256 \end{gathered} \tag{20}$$

Průběhy hodnot účelové funkce (*fitness*) byly seskupeny pro 10 intervalů hodnot F_p. Pro skupinu zobrazují grafy průměrnou hodnotu a interval spolehlivosti 50%. Pro všechny funkce CEC*XX*t a všechny varianty *SDIS* byl nejúspěšnější (nebo stejný) interval koeficientů F_p = <1,428; 1,828>. Obrázek ukazuje, že naopak u funkce F7 byl nejúspěšnější interval koeficientů F_p = <1,224; 1,42>, ale na různá nastavení *SDIS* také nebyla pozorována žádná podstatná reakce. Také zobrazuje, že funkce CEC05t je k volbě *F* koeficientů tolerantnější. Obrázek 12 ilustruje, že u funkce CEC04t k největším rozdílům vedly *SDIS* = {*uni*, *tor*, *HVB*} pro některé intervaly F_p. Naopak pro funkci CEC07t neměla *SDIS* žádný zřejmý vliv a grafy vypadají stejně jako u *SDIS*=*uni*. Autor považuje za možné, že sada testovacích funkcí nevykazuje velkou citlivost na volbu *SDIS*, protože se ani v jedné dimenzi globální extrém nenachází blízko hranice intervalu.

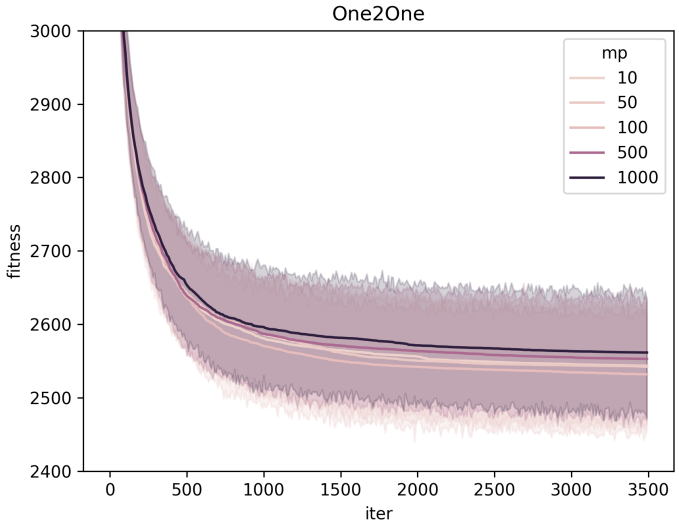


Obr. 11: Průběhy fitness u funkcí F7 a CEC05t pro různé intervaly F koeficientů

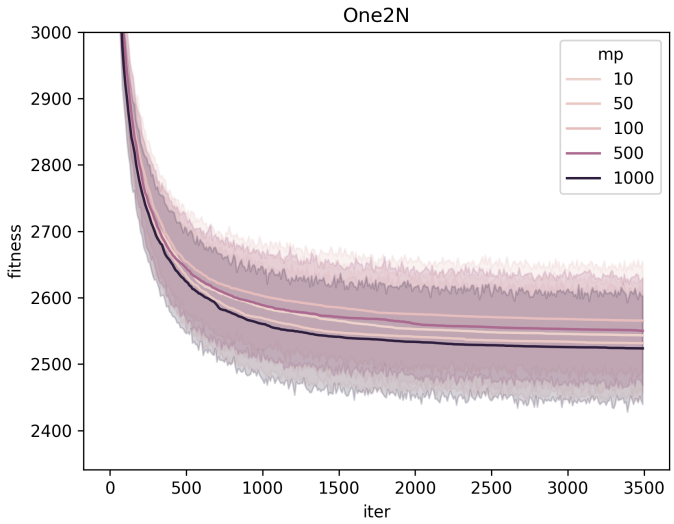


Obr. 12: Výběr průběhů fitness CEC04t a CEC07t pro různé skupiny koeficientů F a SDIS

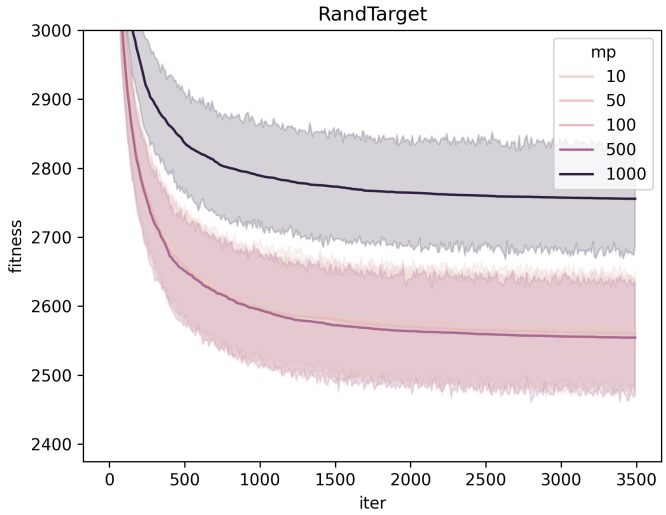
Následující druhý experiment zkoumal vliv periody migrace na hodnotu fitness pro různé migrační strategie. Parametry DE jsou téměř stejné jako u předchozího experimentu, SDIS je volena COTN a fitness funkce CEC07t. Grafy na obrázcích 13, 14, 15, a 16 zobrazují kolem průměrné fitness pro periodu migrace interval spolehlivosti 50%. Poměrně překvapivá je bifurkace u strategie *RandTarget*.



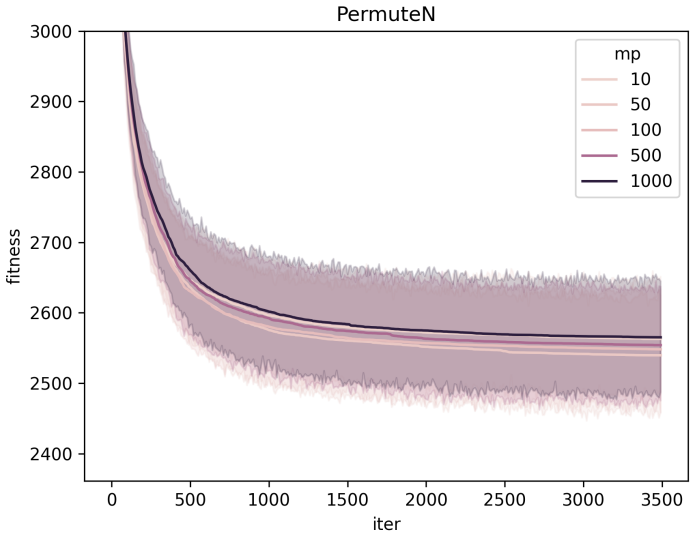
Obr. 13: Vliv periody migrace na fitness u strategie One to one (CEC07t)



Obr. 14: Vliv periody migrace na fitness u strategie One to N (CEC07t)



Obr. 15: Vliv periody migrace na fitness u strategie RandTarget (CEC07t)

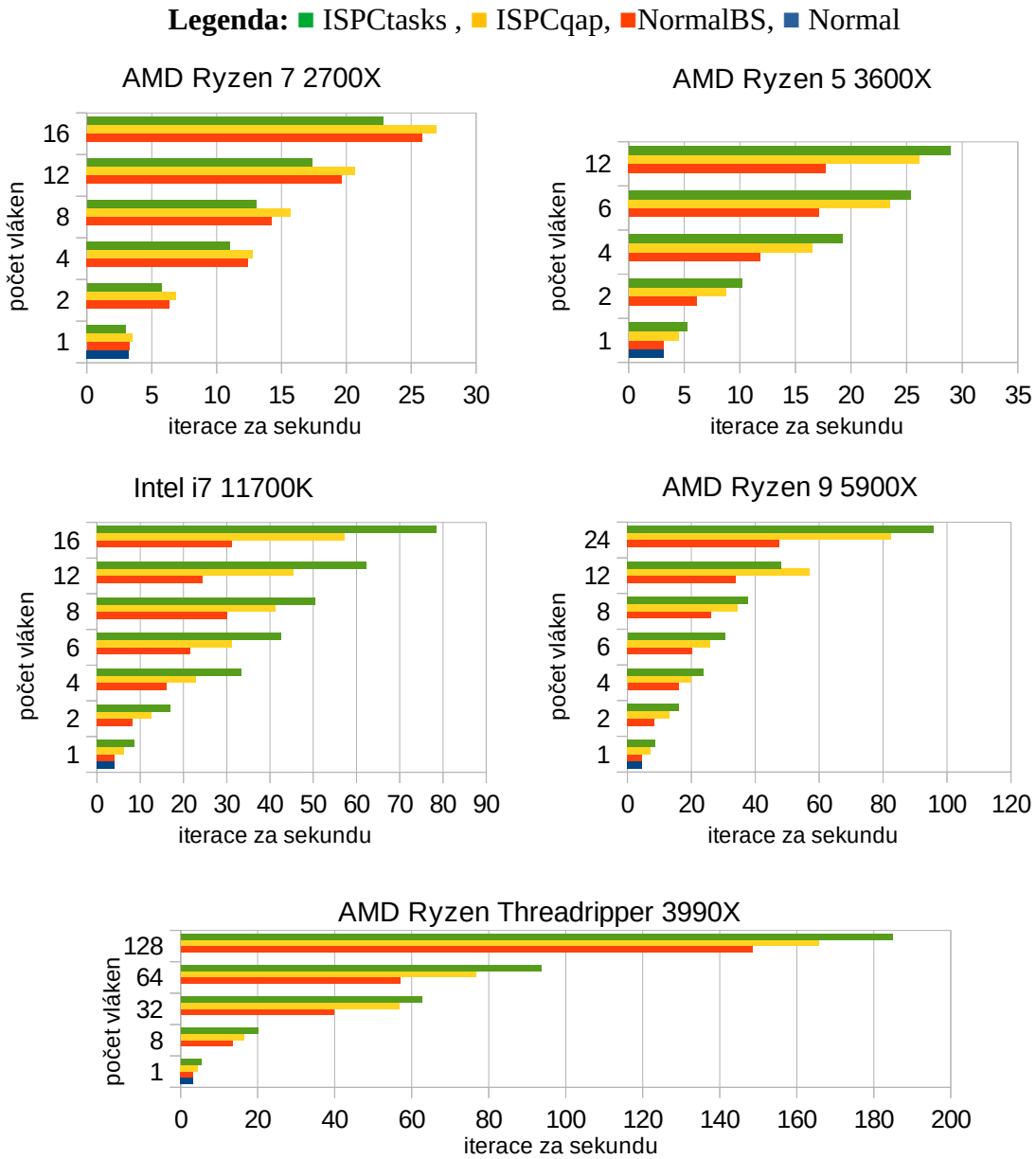


Obr. 16: Vliv periody migrace na fitness u strategie PermuteN (CEC07t)

6.2 Porovnání implementací HC12 pro QAP na CPU a GPU

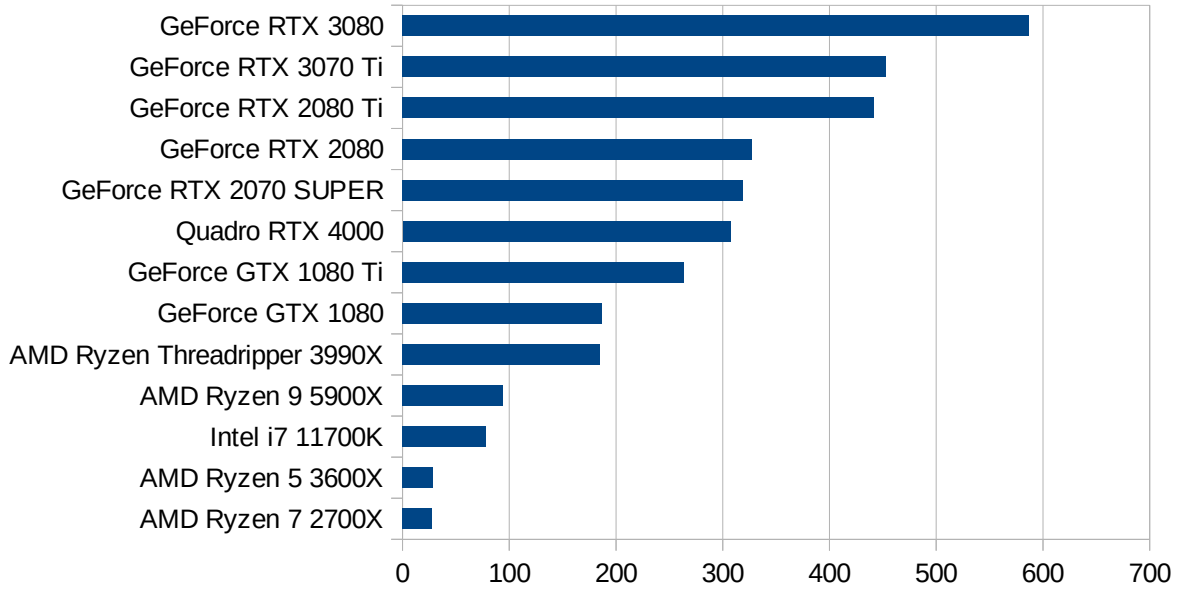
Varianta *ISPCqap* byla vyvíjena dodatečně, protože hlavní stroj použitý pro vývoj byl s procesorem AMD Ryzen 7 2700X, na kterém varianta *ISPCtasks* byla pomalejší než základní varianty *Normal(BS)* v rozporu s hypotézou autora. Další testy však prokázaly, že při použití procesorů s novější architekturou je naopak *ISPCtasks* výhodnější. Benchmark použil QAP úlohu *tho150* a HC12 pro 15 výměn (swapů), tedy 30 parametrů a 28 921 řádků.

Následující grafy na obrázku 17 ilustrují výkon jednotlivých CPU a také poměr výhodnosti jednotlivých variant pro danou architekturu. Každý graf má jiné měřítko. Všechny testované procesory podporovaly maximálně dvě aktivní vlákna na jádro (AMD nazývá jako Simultaneous Multi Threading; Intel jako Hyper-Threading Technology).



Obr. 17: Porovnání výkonu jednotlivých HC12qapswap2 na CPU

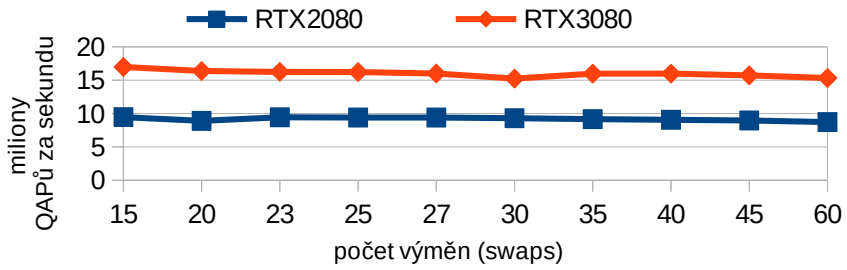
Přestože se podařilo pomocí ISPC dosáhnout výhodného urychlení algoritmu na CPU, v porovnání s CUDA implementací vycházejí GPU výhodněji (obrázek 18). Použitá implementace pro benchmark v zájmu objektivnosti porovnání používá shodné datové typy.



Obr. 18: Porovnání CPU a GPU v benchmarku HC12qapswap2

Ani 64 jádrový procesor AMD Ryzen Threadripper 3990X se základní frekvencí 2,9 GHz (4,3 GHz *boost*) a tepelným výkonovým limitem TDP 280W nestačil na starší grafickou kartu MSI GeForce GTX 1080 ARMOR 8G OC (chip NVIDIA GP104, generace Pascal) s frekvencí 1,657 GHz (1,797GHz *boost*) a tepelným výkonovým limitem TDP 180W. Chip GP104 má 7,2 miliard tranzistorů, byl vyráběn 16nm procesem u TSMC a má pouze 3968 KB *cache* se dvěma úrovněmi. Naopak zmíněný Threadripper je soustava 9 *chipů* (či *chippletů*), 8 z nich vyráběných u TSMC výrazně novějším 7nm procesem a I/O *chip* vyráběný 14nm procesem GlobalFoundries. Celkově procesor obsahuje 30,4 miliard tranzistorů. Velká část z nich je však využita na obří 292MB *cache* se třemi úrovněmi, řídící logiku a spekulativní vyhodnocování instrukcí.

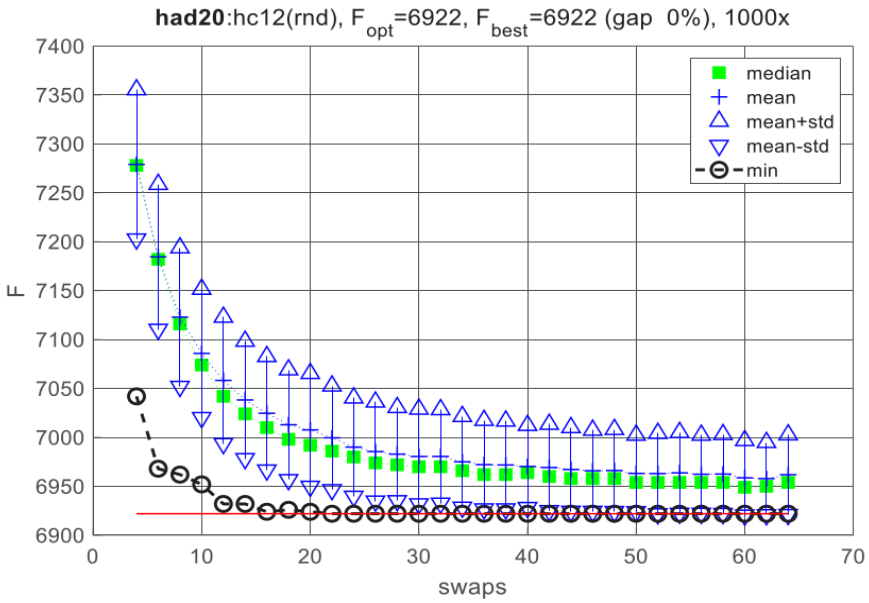
Také bylo ověřeno, že CUDA implementace udržuje od 15 do 60 výměn (28921 až 461281 řádků) GPU plně vytíženo (obrázek 19).



Obr. 19: Efektivní vytížení GPU (HC12qapswap2)

6.3 HC12qs2 na QAPLIB

Na osmi menších úlohách z QAPLIBu, které mají známé optimální řešení, bylo testováno vhodné množství výměn (swapů) a poměr úspěšnost k množství restartů algoritmu. Obrázek 20 ilustruje jak zvyšující se počet výměn vede k lepšímu průměrnému řešení a zvyšuje se pravděpodobnost dosáhnutí optima. Uváděný čas v tabulce 4 je počítán do objevení prvního optimálního řešení pro uvedený počet výměn. Byla použita grafická výpočetní karta NVIDIA RTX 2080 (8 GB). Výsledky byly prezentovány na konferenci GECCO [90].



Obr. 20: HC12 QAPLIB had20: Závislost fitness na počtu výměn

Tabulka 4: HC12 8xQAPLIB: minimální počet výměn, úspěšnost a čas do optima

QAP	Počet výměn	Poměr úspěšnosti [%]	Čas do optima [s]
esc16a	62	100,0	0,0014
esc32a	52	0,1	5,9462
had16	64	34,8	0,0288
had18	64	12,3	0,1476
had20	62	6,7	0,3072
rou12	60	11,6	0,0338
rou15	50	2,0	0,2378
rou20	44	0,1	8,4109

Další aplikací řešiče bylo porovnání jeho řešení inicializovaných náhodně a řešení, kterých dosáhl pomocí inicializace permutacemi získanými projekcí odhadů spodní hranice (lower bound projection). Hlavní výsledky shrnuje tabulka 5. Použita byla rovněž výpočetní grafická karta NVIDIA RTX 2080 (8GB). Výsledky byly publikovány v kanadském odborném časopise International Journal of Industrial Engineering Computations [89].

Použité zkratky mají následující význam:

NZŘ – nejlepší známé řešení,

GLB – Gilmore-Lawler bound,

HRW – Hadley-RendlWolkowitz,

AB – convex quadratic bound,

PE – semidefinite bound.

Tabulka 5: Porovnání metod inicializace permutací pro HC12qs2.

QAP	NZŘ (tučně opt.)	Náhodně	GLB	HRW	AB	PE	Chybí do NZŘ [%] (tučně ≤ 2%)
chr12a	**9552**	**9552**	**9552**	**9552**	**9552**	**9552**	**0,00**
chr12b	**9742**	**9742**	**9742**	**9742**	**9742**	**9742**	**0,00**
chr12c	**11156**	11186	**11156**	**11156**	**11156**	**11156**	**0,00**
chr15a	**9896**	10094	10010	9980	10106	**9978**	**0,82**
chr15b	**7990**	8626	8210	9096	8458	8452	2,68
chr15c	**9504**	10118	**9504**	10426	9940	10002	**0,00**
chr18a	**11098**	**11682**	**11682**	12396	12004	12424	5,00
chr18b	**1534**	1538	**1534**	**1534**	1538	**1534**	**0,00**
chr20a	**2192**	2480	2532	2592	**2398**	2402	8,59
chr20b	**2298**	2612	2608	**2598**	2674	2618	11,55
chr20c	**14142**	**14610**	14988	15636	17274	14876	3,20
chr22a	**6156**	6408	6342	6354	6456	**6336**	2,84
chr22b	**6194**	6522	6526	6534	6410	**6352**	2,49
chr25a	**3796**	5062	4678	5056	4970	**4230**	10,26
had20	**6922**	6924	6928	**6922**	6956	**6922**	**0,00**
kra30a	**88900**	93460	92480	93850	93460	**92300**	3,68
kra30b	**91420**	95020	94570	94690	93620	**92380**	**1,04**
kra32	**88700**	**91660**	92420	92270	92650	92320	3,23
nug18	**1930**	1958	**1936**	1950	1938	1938	**0,31**
nug20	**2570**	2598	2614	**2590**	2602	2598	**0,77**
nug21	**2438**	2458	2452	2472	**2450**	2452	**0,49**
nug22	**3596**	3628	**3610**	3628	3628	3610	**0,39**
nug24	**3488**	3552	3554	3582	3546	**3528**	**1,13**
nug25	**3744**	3806	3788	3800	**3760**	3762	**0,43**
nug27	**5234**	5298	5298	5328	**5294**	5304	**1,13**
nug28	**5166**	5314	5284	5288	5272	**5260**	**1,79**

QAP	NZŘ (tučně opt.)	Náhodně	GLB	HRW	AB	PE	Chybí do NZŘ [%] (tučně ≤ 2%)
nug30	**6124**	6272	6260	6316	6254	**6220**	**1,54**
scr15	**51140**	**51140**	52340	**51140**	**51140**	51140	**0,00**
scr20	**110030**	111078	111938	110802	112660	**110772**	**0,67**
sko42	15812	16304	16282	16290	**16106**	16172	**1,83**
sko64	48498	50090	**49904**	49932	49970	49942	2,82
sko72	66256	68298	68182	68264	**67902**	68140	2,42
sko81	90998	93684	93492	93968	93840	**93148**	2,31
sko90	115534	119630	119064	119078	119092	**118446**	2,46
sko100a	152002	157426	157034	156934	**156116**	156820	2,64
sko100b	153890	159060	**158002**	158456	158220	158184	2,60
sko100c	147862	152742	152592	153186	152476	**152374**	2,96
sko100d	149576	154708	154340	**153896**	153978	154144	2,81
sko100e	149150	**153880**	154522	153930	154010	154536	3,07
sko100f	149036	154284	153994	153788	153766	**153558**	2,94
ste36a	**9526**	10234	10126	10052	**9790**	10266	2,70
ste36b	**15852**	17786	17140	17112	**16724**	17770	5,21
ste36c	**8239110**	8701576	8678652	8738822	8695554	**8578694**	3,96
tai25a	**1167256**	1194194	**1177180**	1188890	1188248	1187984	**0,84**
tai50a	4938796	5148702	**5119448**	5131652	5130768	5132594	3,53
tai60a	7205962	7486562	7506384	7499212	7434242	**7427410**	2,98
tai80a	13499184	14034018	14027470	**13966388**	14050896	13993668	3,35
tai100a	21052466	21951138	21932812	21957694	**21893240**	21932070	3,84
tho30	**149936**	154134	156170	153558	154874	**152020**	**1,37**
tho40	240516	251428	249370	248116	**247260**	251034	2,73
tho150	8133398	8511942	8461186	**8454432**	–	–	3,80
wil50	48816	49504	49472	49652	**49434**	49470	**1,25**
wil100	273038	278428	**277210**	277730	277230	277458	**1,50**

6.4 Rozdíly přesnosti výpočtu funkcí spojité optimalizace

Norma IEEE-754 pro aritmetiku v plovoucí desetinné čárce (floating point arithmetic) určuje pravidla a doporučení pro výpočty: bitovou reprezentaci, přesnost (single, double, a další) a způsoby zaokrouhlování. Ve vyšších programovacích jazycích však mnohdy nelze přesně vše nastavit. Výsledky též ovlivňuje použitý HW a nastavení kompilace. Například na 32bit x86 může FPU jednotka interně používat vyšší přesnost a pokud jsou v ladícím režimu mezivýsledky přenášeny do RAM, můžou se lišit od „release“ verze. Též se pak výsledky liší od 64bit x86. Při kompilaci lze také povolit kompilátoru použití méně přesných funkcí či optimalizací, které předpokládají například platnost distributivního a asociativního zákona u reálných čísel. Čísla s plovoucí desetinou čárkou však podmínky distributivního ani asociativního zákona nesplňují. Další potíže přináší také fakt, že norma se nijak nevěnuje tomu jak mají být implementovány abstraktnější operace jako suma či produkt složek vektorů, skalární součin vektorů, násobení matic nebo diskrétní konvoluce. V případě užití optimalizovaných paralelních a SIMD implementací se mohou výsledky lišit poměrně hodně.

Tabulka 6 ukazuje výsledky porovnání implementací v Matlabu, NumPy a CUDA C s přesnější metodou (`vpa`). Pro porovnání bylo v 10 dimenzích vygenerováno 10 000 bodů a byla spočítána funkční hodnota pomocí Variable Precision Arithmetic (`vpa` funkce) z Matlab Symbolic Math Toolbox s přesností na 50 desetinných míst (v desítkové soustavě). Porovnávané implementace využívaly datový typ IEEE-754 double precision, který je přesný maximálně na 16 desetinných míst (v desítkové soustavě).

Mezi různými verzemi Matlabu se také výsledky mohou lišit podle architektury (x86, x86-64, arm64) a verze použité knihovny Intel oneMKL (oneAPI Math Kernel Library), kterou Matlab interně používá pro základní operace. NumPy lze také provozovat ve verzi s Intel oneMKL. Byl proveden dodatečný test s Matlabem 2019a x86-64 (Windows) a výsledky byly absolutně shodné s Matlabem 2023b x86-64 (Linux).

Tabulka 6: Statistika absolutní odchylky implementací testovacích funkcí

Funkce	maximum			průměr			směrodatná odchylka		
	CUDA	NumPy 1.23.5	Matlab 2023b	CUDA	NumPy 1.23.5	Matlab 2023b	CUDA	NumPy 1.23.5	Matlab 2023b
F1	5,97E-13	**5,68E-13**	**5,68E-13**	6,05E-15	**5,75E-15**	6,48E-15	1,44E-14	1,43E-14	1,45E-14
F2	4,89E-12	4,89E-12	4,89E-12	**2,78E-14**	3,01E-14	3,15E-14	9,26E-14	9,37E-14	9,37E-14
F3	9,75E-10	9,75E-10	9,75E-10	**1,00E-11**	1,01E-11	1,03E-11	2,08E-11	2,07E-11	2,13E-11
F4	2,46E-11	2,46E-11	2,46E-11	**1,35E-13**	1,51E-13	1,56E-13	4,62E-13	4,63E-13	4,65E-13
F5	7,82E-11	7,82E-11	7,82E-11	3,94E-13	**3,81E-13**	4,21E-13	1,12E-12	1,11E-12	1,12E-12
F6	3,30E-12	3,30E-12	3,30E-12	2,26E-14	**2,21E-14**	2,25E-14	7,10E-14	7,10E-14	7,09E-14
F7	2,96E-12	**2,84E-12**	2,96E-12	5,26E-13	5,26E-13	5,27E-13	4,14E-13	4,14E-13	4,16E-13
F8	1,71E-13	**1,14E-13**	1,71E-13	2,45E-14	**2,19E-14**	2,45E-14	2,88E-14	2,72E-14	2,88E-14
F9	4,17E-14	4,22E-14	4,17E-14	1,55E-16	1,21E-16	**1,46E-16**	5,74E-16	5,74E-16	5,74E-16
F10	1,14E-13	1,14E-13	1,14E-13	1,45E-15	**1,44E-15**	1,45E-15	2,21E-15	2,20E-15	2,20E-15
F11	4,86E-16	1,39E-15	4,86E-16	1,43E-19	2,88E-19	**1,36E-19**	5,72E-18	1,67E-17	5,64E-18
F12	2,09E-13	2,09E-13	2,09E-13	1,15E-15	**1,13E-15**	1,14E-15	3,16E-15	3,17E-15	3,17E-15
CEC04	3,30E-12	3,30E-12	3,30E-12	2,26E-14	**2,21E-14**	2,51E-14	7,10E-14	7,10E-14	7,14E-14
CEC05	1,71E-13	**1,14E-13**	1,71E-13	2,45E-14	**2,19E-14**	2,45E-14	2,88E-14	2,72E-14	2,88E-14
CEC06	4,59E-11	**4,59E-11**	**4,59E-11**	**1,41E-12**	1,41E-12	1,41E-12	1,19E-12	1,19E-12	1,19E-12
CEC07	3,64E-12	3,64E-12	3,64E-12	7,94E-13	**7,92E-13**	7,93E-13	6,78E-13	6,78E-13	6,78E-13
CEC08	7,82E-14	7,82E-14	7,82E-14	**4,53E-16**	4,98E-16	4,68E-16	9,58E-16	9,64E-16	9,63E-16
CEC09	9,95E-14	**9,86E-14**	9,95E-14	4,90E-16	**4,76E-16**	4,95E-16	1,31E-15	1,32E-15	1,33E-15
CEC10	1,85E-13	1,85E-13	1,85E-13	**4,18E-15**	4,18E-15	4,18E-15	4,23E-15	4,23E-15	4,23E-15
CEC04t	**5,12E-13**	6,82E-13	5,68E-13	**5,99E-14**	6,48E-14	6,65E-14	5,72E-14	6,16E-14	6,17E-14
CEC05t	6,82E-13	6,82E-13	6,82E-13	6,82E-14	**6,79E-14**	7,09E-14	7,64E-14	7,56E-14	7,83E-14
CEC06t	**1,76E-11**	1,78E-11	1,78E-11	2,84E-12	**2,79E-12**	2,86E-12	2,25E-12	2,23E-12	2,30E-12
CEC07t	2,46E-11	**2,27E-11**	2,41E-11	**1,42E-12**	1,55E-12	1,56E-12	1,23E-12	1,36E-12	1,36E-12
CEC08t	**9,77E-15**	1,24E-14	1,07E-14	**6,60E-16**	8,05E-16	7,78E-16	8,80E-16	1,02E-15	1,02E-15
CEC09t	9,21E-13	1,53E-12	**1,18E-12**	**1,92E-15**	1,98E-15	2,01E-15	9,62E-15	1,55E-14	1,21E-14
CEC10t	3,73E-13	3,73E-13	3,73E-13	**8,16E-15**	8,74E-15	8,91E-15	8,00E-15	8,40E-15	8,70E-15

7 ZÁVĚR

S růstem celkového výkonu počítačů, a díky zvyšování hustoty tranzistorů v křemíkových integrovaných obvodech (ICs, chipech), je stále obtížnější je efektivně využívat a programovat. Datově paralelní procesory jako jsou GPGPU představují velkou výzvu v oblasti architektury a tvorby programů, které se stávají součástí stále složitějších softwarových systémů. Pro mnohé úlohy mají GPGPU lepší poměr využité plochy čipu výpočetními jednotkami ku ploše využité pro řídící struktury a koherentní cache paměti. Při vhodném programování tak umožňují řešit řádově složitější problémy než jejich CPU ekvivalenty. Ovšem to platí pouze pro některé úlohy nebo jejich části, na ty se pak GPGPU použijí jako akcelerátory v komplexním heterogenním systému. Téměř celá práce se zabývá převážně využitím GPGPU společnosti NVIDIA pomocí softwarové platformy CUDA a jazyka C++. Moderní více jádrové CPU však v mnohém nezůstávají pozadu a přidávají také stále širší vektorové SIMD jednotky. To v kombinaci s velkou hierarchickou koherentní cache může u některých úloh poskytnout prostor pro efektivnější implementaci. Kapitoly 4.3 a 5.3.2.1 se věnovaly jejich programování pomocí jazyků C++ a ISPC obecně i na konkrétní úloze. Kapitola 4.1 se široce věnuje přehledu hardwarových systémů a prostředků používaných v oblasti vysoce náročných výpočtů (HPC). Kromě více jádrových CPU a GPGPU jsou uváděny osvědčená řešení jako superpočítače, clustery, grid a cloud, programovatelná hradlová pole FPGA a zakázkové obvody ASIC. Také je zmíněna nadějná novinka Wafer-Scale Engine společnosti Cerebras. V určitých aplikacích i CPU mění některé strategie. Příkladem jsou ARM procesory firmy Apple pro notebooky, SoC různých výrobců pro mobilní zařízení, RISC-V embedded řešení různých výrobců a ARM procesory pro servery velkých hráčů v oblasti cloud computingu (Amazon, Microsoft, Google).

Jedním z hlavních cílů práce bylo vytvořit a demonstrovat plně funkční, komplexní a datově paralelní implementace vybraných a pro daný účel vhodných metaheuristických algoritmů a úloh spojité a kombinatorické optimalizace. Metaheuristiky prokazatelně umožňují nacházet dostatečně dobrá řešení v aplikacích inženýrské optimalizace. Protože prohledávané prostory parametrů v kombinatorické optimalizaci rostou exponenciálně s počtem prvků a pro spojitou optimalizaci to jsou principiálně nekonečné multidimenzionální množiny. Tento fakt často znemožňuje hledat exaktní řešení. Pak je třeba v praktické aplikaci přistoupit k tzv. inženýrské optimalizaci a najít dostatečně vhodná řešení v rozumném čase (feasible solution in feasible time) a za přijatelnou cenu (feasible solution at acceptable cost).

Prvním implementovaným metaheuristickým algoritmem je HC12 rozšířený o novou implementaci seznamu zakázaných řešení (Tabu list), která je vhodná pro GPU. Obecný řešič spouštěný z Matlabu umožňuje uživateli dodat vlastní účelovou funkci pro GPU (CUDA), a také jí před spuštěním samotného algoritmu připravit obecně jakákoliv data. Lze nastavit mnoho parametrů, včetně domény reprezentace, tj. zda se účelová funkce bude počítat nad binárními řetězci, celými čísly nebo reálnými čísly. Podle toho se aktivují příslušné transformace a může se ušetřit alokace GPU paměti i čas výpočtu nepotřebných transformací. Je také implementován a popsán specializovaný řešič *hc12qs2_v2*, který efektivně kombinuje HC12 algoritmus přímo s řešením QAP problému pomocí překódování *qapswap2*. Také je část implementace a jejího popisu věnována praktickému propojování software a tvorbě knihoven a *wrapperů* pro různé programovací jazyky a SW balíky (C++ šablonové třídy, C dynamická knihovna, Python *wrapper* C dynamické knihovny, Matlab MEX dynamická

knihovna) s konkrétním příkladem právě na *hc12qs2_v2*. Tímto způsobem poskytuje předložená práce velký prostor k dalšímu rozšíření.

Druhou implementovanou metaheuristikou je zcela nová varianta diferenciální evoluce (DE) s více ostrovními populacemi vhodná právě pro GPU jak z pohledu genetických operátorů (mutace, křížení, selekce) tak pro výpočet účelové funkce. Navržená implementace DE vhodně využívá sdílenou paměť dostupnou na Streaming Multiprocesorech v GPU NVIDIA. Při vývoji byl kladen důraz na podporu více variant algoritmu a různých nastavení (feature complete). Návrh je realizován tak, aby bylo poměrně snadné dále rozšiřovat funkcionalitu. Dodatečná nastavení mají výchozí hodnoty a jsou předávány pomocí třídy *DictOptions*, která je inspirovaná slovníky v Pythonu. Uživatelem definovaná fitness funkce se předává jako objekt s rozhraním definovaným pomocí šablonové abstraktní bázové třídy a může si tak udržovat jakýkoliv kontext svých dat. Ve veřejném API i interně implementace využívá C++ šablony (*templates*). Bylo implementováno 6 variant mutačního operátoru, 6 variant strategií korekce chybných řešení SDIS a 4 varianty migrace s možností ji vypnout (nezávislé populace pro experimenty). Celkově je implementace využitelná jak pro výzkum vlastností nových variant a závislostí parametrů, tak pro praktické aplikace inženýrské optimalizace. Kapitola 6.1 prezentuje dva experimenty. Prvním je zkoumání vliv SDIS a koeficientů F na fitness na nezávislých populacích pro různé funkce. Druhý experiment zkoumá vliv periody migrace na fitness pro čtyři migrační strategie u transformované funkce CEC07t.

Bylo implementováno šestnáct známých testovacích účelových funkcí spojité globální optimalizace z literatury ve třech různých programovacích prostředích: Matlab, Python+NumPy, CUDA C. Verze v CUDA C jsou dvě, první pro spolupráci s HC12 řešičem a druhá pro řešič Diferenciální evoluce s více ostrovními populacemi. K sedmi funkcím byly také přidány další modifikace v podobě transformací, aby se lépe porovnávala robustnost metod, tak jak byly definovány v literatuře. Bylo verifikováno jak moc se výsledky jednotlivých implementací liší vůči přesnější metodě výpočtu (Variable precision arithmetic) a výsledky experimentu byly prezentovány v kapitole 6.4. Pro tyto testovací funkce jsou rozdíly zanedbatelné, ale v případě optimalizace účelových funkcí týkajících se simulace a řízení chaotických, tuhých anebo vysokofrekvenčních systémů je třeba dbát na přesnost výsledků a případnou nepřenositelnost výsledného nastavení mezi implementacemi.

Klasický kombinatorický optimalizační problém kvadratického přiřazení (QAP) byl s HC12 rigorózně testován a výsledky experimentů v kapitole 6.3 byly publikovány na konferenci a v odborném časopise.

QAP v kombinaci s HC12 byl také využit pro srovnání GPU (CUDA) a CPU (ISPC) implementací. Přestože se podařilo zefektivnit využití SIMD jednotek pomocí explicitního popisu paralelnosti výpočtu v jazyce ISPC na CPU a urychlit výpočty oproti optimalizujícímu C++ kompilátoru LLVM/clang, ukázalo se v experimentech popsaných v kapitole 6.2, že pro HC12 a QAP je architektura GPU významně výhodnější. Algoritmus na CPU dobře škáloval s počtem použitých jader. Nicméně ani 64 jádrový procesor AMD Ryzen Threadripper 3990X nestačil na starší grafickou kartu GeForce GTX 1080. Přestože je procesor Ryzen Threadripper vyráběn novějším procesem, běží na vyšší frekvenci,

spotřebovává více energie a má k dispozici řádově více tranzistorů, pro řešení tohoto problému je pomalejší.

Problém splnitelnosti Booleovské formule (SAT) byl analyzován z pohledu akcelerace výpočtu paralelního pro více klauzulí při jediném ohodnocení a pak paralelního pro vícero ohodnocení. Verifikovaná implementace dostupná v příloze minimalizuje velikost použitých datových typů a maximálně využívá bitových operací pro efektivní určení platnosti formule a počtu platných klauzulí. Akcelerovanou implementaci vyhodnocování SAT bude vhodné integrovat do otevřeného SAT řešiče, který používá heuristiku (například CaDiCaL).

V rámci dalšího vývoje praktického použití je žádoucí poskytnout pro více populační diferenciální evoluci knihovny/wrappery pro jazyky C, Python, Matlab, Octave a Julia. Určitou výzvou budou adaptéry pro možnost definovat účelovou funkci pro GPU v příslušném programovacím jazyce či za pomoci knihoven k němu dostupných (Python: CuPy, Numba, „CUDA Python“; Matlab gpuArray; Julia CUDA.jl).

Další žádoucí rozšíření více populační diferenciální evoluce je možnost u některých populací aktivovat adaptivnost parametrů a zkoumat jak se potenciálně budou adaptivní a neadaptivní populace vzájemně obohacovat. Také bude vhodné více analyzovat jaký je vztah mezi počtem populací, strategií migrace a periodou migrace.

V oblasti spojité optimalizace by se autor chtěl dále věnovat využití své varianty diferenciální evoluce při návrhu parametrů modelů pro simulaci a řízení komplexních dynamických systémů popsaných soustavou diferenciálních rovnic a implementací s tím souvisejících paralelních implementací řešičů.

Autor by se rád v budoucnu také soustředil na úpravu metaheuristiky GAHC zobecněním schématu HC12 mutace a také bude vhodné pokusit se implementovat více populační variantu podobně jako u DE. Jistě zde existuje potenciál pro řešení základních kombinatorických úloh (QAP, SAT, problém batohu – knapsack, atd.) i složitějších aplikací.

Pro problémy kombinatorické optimalizace SAT a QAP jsou potenciálně velice zajímavé pro implementaci na programovatelném hradlovém poli (FPGA). Je otázkou zda takto akcelerovat pouze účelovou funkci nebo se pokusit na pole přenést i metaheuristiky jako HC12 a GAHC.

Mnoho reálných aplikací využívá hybridní algoritmy a komplexní kombinace metaheuristik, rozhodovacích algoritmů, různých klasifikátorů, hlubokých neuronových sítí a posilovaného učení. Hybridní jsou také použité HW prostředky v rámci jediné aplikace od cloudu po mobilní zařízení jak v běžném životě (GPS, nákupy, zábava) tak v průmyslové praxi (Industry 4.0). Dalším cílem autora pokračovat ve výzkumu v rámci tohoto velmi dynamického oboru aplikace metod umělé inteligence a vysoce náročných výpočtů (HPC).

8 LITERATURA

[1] BERTOT, Yves. A simple canonical representation of rational numbers. *Electronic Notes in Theoretical Computer Science* [online]. 85(7), 1-16 [cit. 2021-08-10]. ISSN 15710661. Dostupné z: doi:10.1016/S1571-0661(04)80754-0

[2] *IEEE Standard for Floating-Point Arithmetic: IEEE Std 754-2008 (Revision of IEEE Std 754-1985)*. New York: IEEE Computer Society. Dostupné také z: https://irem.univ-reunion.fr/IMG/pdf/ieee-754-2008.pdf

[3] *GNU multiple precision arithmetic library maunal: Introduction to GNU MP* [online]. Free Software Foundation [cit. 2021-08-10].
Dostupné z: https://gmplib.org/manual/Introduction-to-GMP

[4] GOUALARD, Frédéric. MicroFloatingPoints. *Github.io* [online]. [cit. 2021-08-10]. Dostupné z: https://goualard-f.github.io/MicroFloatingPoints.jl/stable/

[5] MOLINA, Daniel, Javier POYATOS, Javier Del SER, Salvador GARCÍA, Amir HUSSAIN a Francisco HERRERA, 2020. Comprehensive Taxonomies of Nature- and Bio-inspired Optimization: Inspiration Versus Algorithmic Behavior, Critical Analysis Recommendations. Cognitive Computation [online]. 12(5), 897-939 [cit. 2024-02-07]. ISSN 1866-9956. Dostupné z: doi:10.1007/s12559-020-09730-8

[6] ALEV, Vedat Levi a Lap Chi LAU, 2020. Improved analysis of higher order random walks and applications. In: *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing* [online]. New York, NY, USA: ACM, 2020-06-22, s. 1198-1211 [cit. 2024-02-07]. ISBN 9781450369794. Dostupné z: doi:10.1145/3357713.3384317

[7] DELAHAYE, Daniel, Supatcha CHAIMATANAN a Marcel MONGEAU, 2019. Simulated Annealing: From Basics to Applications. In: GENDREAU, Michel a Jean-Yves POTVIN, ed. Handbook of Metaheuristics [online]. Cham: Springer International Publishing, s. 1-35 [cit. 2023-11-02]. International Series in Operations Research & Management Science. ISBN 978-3-319-91085-7. Dostupné z: doi:10.1007/978-3-319-91086-4_1

[8] KRUSE, Rudolf, Christian BORGELT, Frank KLAWONN, et al., 2013. Introduction to Evolutionary Algorithms. In: *Computational Intelligence* [online]. London: Springer London, s. 167-195 [cit. 2023-11-02]. Texts in Computer Science. ISBN 978-1-4471-5012-1. Dostupné z: doi:10.1007/978-1-4471-5013-8_11

[9] KLEIN, Kyle a Julian NEIRA, 2014. Nelder-Mead Simplex Optimization Routine for Large-Scale Problems: A Distributed Memory Implementation. *Computational Economics* [online]. 43(4), 447-461 [cit. 2024-02-07]. ISSN 0927-7099. Dostupné z: doi:10.1007/s10614-013-9377-8

[10] MATOUSEK, Radomil. HC12: Highly Scalable Optimisation Algorithm. GONZÁLEZ, Juan R., David Alejandro PELTA, Carlos CRUZ, Germán TERRAZAS a Natalio KRASNOGOR, ed. *Nature Inspired Cooperative Strategies for Optimization* (NICSO 2010) [online]. 1. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, s. 177-183 [cit. 2022-01-21]. Studies in Computational Intelligence. ISBN 978-3-642-12537-9. Dostupné z: doi:10.1007/978-3-642-12538-6_15

[11] MATOUSEK, Radomil a Eva ZAMPACHOVA. Promising GAHC and HC12 algorithms in global optimization tasks. *Optimization Methods and Software* [online]. 2011, 26(3), 405-419 [cit. 2021-06-10]. ISSN 1055-6788. Dostupné z: doi:10.1080/10556788.2011.556826

[12] MATOUSEK, Radomil a Petr MINAR. Stabilization of Chaotic Logistic Equation Using HC12 and Grammatical Evolution. *Nostradamus 2013: Prediction, Modeling and Analysis of Complex Systems*. 1. Heidelberg: Springer International Publishing, 2013, 2013, s. 137-146. *Advances in Intelligent Systems and Computing*. ISBN 978-3-319-00541-6. ISSN 2194-5357. Dostupné z: doi:10.1007/978-3-319-00542-3_14

[13] MATOUSEK, Radomil. HC12 Implemented Using the CUDA Platform. In: *APPLIED COMPUTER SCIENCE: International Conference on Applied Computer Science (ACS)*. Malta: WSEAS, 2010, s. 649-652. ISBN 978-960-474-225-7. ISSN 1792-4863. Dostupné z: http://www.wseas.us/e-library/conferences/2010/Malta/ACS/ACS-106.pdf

[14] GOLDBERG, David Edward. *Genetic algorithms in search, optimization, and machine learning*. Boston: Addison-Wesley. ISBN 978-0201157673.

[15] GHOSH, Ashish, Shigeyoshi TSUTSUI a Hideo TANAKA. Individual aging in genetic algorithms. In: *1996 Australian New Zealand Conference on Intelligent Information Systems. Proceedings. ANZIIS 96* [online]. Adelaide, SA, Australia: IEEE, s. 276-279 [cit. 2021-10-15]. ISBN 0-7803-3667-4. Dostupné z: doi:10.1109/ANZIIS.1996.573957

[16] MATOUSEK, Radomil. GAHC: Hybrid Genetic Algorithm. AO, Sio-Iong, Burghard RIEGER a Su-Shing CHEN, ed. *Advances in Computational Algorithms and Data Analysis* [online]. Dordrecht: Springer Netherlands, 2009, s. 549-562 [cit. 2021-06-10]. Lecture Notes in Electrical Engineering. ISBN 978-1-4020-8918-3. Dostupné z: doi:10.1007/978-1-4020-8919-0_38

[17] STORN, Rainer a Kenneth PRICE. Differential Evolution - A Simple and Efficient Adaptive Scheme for Global Optimization Over Continuous Spaces: Technical Report TR-95-012 [online]. Berkeley, CA: *International Computer Science Institute*, 1995 [cit. 2021-08-10]. Dostupné z: https://cse.engineering.nyu.edu/~mleung/CS909/s04/Storn95-012.pdf

[18] STORN, Rainer a Kenneth PRICE. Differential Evolution: A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. *Journal of Global Optimization* [online]. 1997, 11(4), 341-359 [cit. 2021-08-10]. ISSN 0925-5001. Dostupné z: doi:10.1023/A:1008202821328

[19] STORN, R. Differential evolution design of an IIR-filter. In: *Proceedings of IEEE International Conference on Evolutionary Computation* [online]. Nagoya, Japan: IEEE, 1996, s. 268-273 [cit. 2021-08-10]. ISBN 0-7803-2902-3. Dostupné z: doi:10.1109/ICEC.1996.542373

[20] STORN, Rainer a Kenneth PRICE. Minimizing the real functions of the ICEC'96 contest by differential evolution. In: P*roceedings of IEEE International Conference on Evolutionary Computation* [online]. Nagoya, Japan: IEEE, 1996, s. 842-844 [cit. 2021-08-10]. ISBN 0-7803-2902-3. Dostupné z: doi:10.1109/ICEC.1996.542711

[21] TAN, Zhiping a Kangshun LI, 2021. Differential evolution with mixed mutation strategy based on deep reinforcement learning. Applied Soft Computing [online]. 111 [cit. 2024-01-14]. ISSN 15684946. Dostupné z: doi:10.1016/j.asoc.2021.107678

[22] KONONOVA, Anna V., Diederick VERMETTEN, Fabio CARAFFINI, Madalina-A. MITRAN a Daniela ZAHARIE, 2023. The Importance of Being Constrained: Dealing with Infeasible Solutions in Differential Evolution and Beyond. *Evolutionary Computation* [online]. 2023-11-29, 1-46 [cit. 2024-01-15]. ISSN 1530-9304. Dostupné z: doi:10.1162/evco_a_00333

[23] CPPREFERENCE.COM. *std::fmod, std::fmodf, std::fmodl* [online]. [cit. 2024-02-01]. Dostupné z: https://en.cppreference.com/w/cpp/numeric/math/fmod

[24] HAMEED, Asaad Shakir, Burhanuddin MOHD, Ngo HEA a Modhi LAFTA. Improved Discrete Differential Evolution Algorithm in Solving Quadratic Assignment Problem for best Solutions. *International Journal of Advanced Computer Science and Applications* [online]. 2018, 9(12) [cit. 2021-08-10]. ISSN 21565570.
Dostupné z: doi:10.14569/IJACSA.2018.091261

[25] KUSHIDA, Jun-ichi, Kazuhisa OBA, Akira HARA a Tetsuyuki TAKAHAMA. Solving quadratic assignment problems by differential evolution. In: *The 6th International Conference on Soft Computing and Intelligent Systems, and The 13th International Symposium on Advanced Intelligence Systems* [online]. IEEE, 2012, 2012, s. 639-644 [cit. 2021-08-10]. ISBN 978-1-4673-2743-5.
Dostupné z: doi:10.1109/SCIS-ISIS.2012.6505170

[26] DE P. VERONESE, Lucas a Renato A. KROHLING, 2010. Differential evolution algorithm on the GPU with C-CUDA. In: *IEEE Congress on Evolutionary Computation* [online]. IEEE, s. 1-7 [cit. 2023-11-01]. ISBN 978-1-4244-6909-3. Dostupné z: doi:10.1109/CEC.2010.558621

[27] QIN, A. K., Federico RAIMONDO, Florence FORBES a Yew Soon ONG, 2012. An improved CUDA-based implementation of differential evolution on GPU. In: *Proceedings of the 14th annual conference on Genetic and evolutionary computation* [online]. New York, NY, USA: ACM, 2012-07-07, s. 991-998 [cit. 2023-11-01]. ISBN 9781450311779. Dostupné z: doi:10.1145/2330163.2330301

[28] PIOTROWSKI, Adam P., 2017. Review of Differential Evolution population size. *Swarm and Evolutionary Computation* [online]. 32, 1-24 [cit. 2023-11-01]. ISSN 22106502. Dostupné z: doi:10.1016/j.swevo.2016.05.003

[29] CHARILOGIS, Vasileios a Ioannis G. TSOULOS, 2023. A Parallel Implementation of the Differential Evolution Method. Analytics [online]. 2(1), 17-30 [cit. 2022-11-01]. Dostupné z: doi:10.3390/analytics2010002

[30] SKAKOVSKI, Aleksander a Piotr JĘDRZEJOWICZ, 2019. An island-based differential evolution algorithm with the multi-size populations. *Expert Systems with Applications* [online]. (vol. 126), 308-320 [cit. 2023-10-10]. ISSN 09574174. Dostupné z: doi:10.1016/j.eswa.2019.02.027

[31] LIU, J. a J. LAMPINEN, 2005. A Fuzzy Adaptive Differential Evolution Algorithm. *Soft Computing* [online]. 9(6), 448-462 [cit. 2023-10-10]. ISSN 1432-7643. Dostupné z: doi:10.1007/s00500-004-0363-x

[32] MATOUSEK, R., P. OSMERA a J. ROUPEC, 2000. GA with fuzzy inference system. In: *Proceedings of the 2000 Congress on Evolutionary Computation*. CEC00 (Cat. No.00TH8512) [online]. IEEE, s. 646-651 [cit. 2023-10-10]. ISBN 0-7803-6375-2. Dostupné z: doi:10.1109/CEC.2000.870359

[33] PIOTROWSKI, Adam P., 2018. L-SHADE optimization algorithms with population-wide inertia. *Information Sciences* [online]. 468, 117-141 [cit. 2023-11-04]. ISSN 00200255. Dostupné z: doi:10.1016/j.ins.2018.08.030

[34] ZHANG, Sheng Xin, Xin Rou HU a Shao Yong ZHENG, 2024. Differential evolution with evolutionary scale adaptation. *Swarm and Evolutionary Computation* [online]. 85 [cit. 2024-02-07]. ISSN 22106502. Dostupné z: doi:10.1016/j.swevo.2024.101481

[35] AHMAD, Mohamad Faiz, Nor Ashidi Mat ISA, Wei Hong LIM a Koon Meng ANG, 2022. Differential evolution: A recent review based on state-of-the-art works. Alexandria Engineering Journal [online]. 61(5), 3831-3872 [cit. 2024-02-07]. ISSN 11100168. Dostupné z: doi:10.1016/j.aej.2021.09.013

[36] BAO, Wenzheng a Bin YANG, 2024. Protein acetylation sites with complex-valued polynomial model. *Frontiers of Computer Science* [online]. 18(3) [cit. 2024-02-10]. ISSN 2095-2228. Dostupné z: doi:10.1007/s11704-023-2640-9

[37] HU, Siyang, Sofiane BOUHEDMA, Arwed SCHÜTZ, Simon STINDT, Dennis HOHLFELD a Tamara BECHTOLD. Design optimization of multi-resonant piezoelectric energy harvesters. *Microelectronics Reliability* [online]. 120 [cit. 2021-10-15]. ISSN 00262714. Dostupné z: doi:10.1016/j.microrel.2021.114114

[38] POHLHEIM, Hartmut. GEATbx: The Genetic and Evolutionary Algorithm Toolbox for Matlab [online]. 2007-03 [cit. 2021-11-08]. Dostupné z: http://www.geatbx.com/

[39] PRICE, KV, NH AWAD, MZ ALI a PN SUGANTHAN. Problem definitions and evaluation criteria for the 100-digit challenge special session and competition on single objective numerical optimization. Technical Report. Nanyang Technological University, 2018.

[40] SURJANOVIC, Sonja a Derek BINGHAM. Virtual Library of Simulation Experiments: Test Functions and Datasets [online]. 2013 [cit. 2021-11-08]. Dostupné z: https://www.sfu.ca/~ssurjano

[41] ABDEL-BASSET, Mohamed, Gunasekaran MANOGARAN, Heba RASHAD a Abdel Nasser H. ZAIED. A comprehensive review of quadratic assignment problem: variants, hybrids and applications. Journal of Ambient Intelligence and Humanized Computing [online]. 20 June 2018, 2018 [cit. 2021-08-10]. ISSN 1868-5137. Dostupné z: doi:10.1007/s12652-018-0917-x

[42] ÇELA, Eranda. The Quadratic Assignment Problem: Theory and Algorithms. 1. Springer Science & Business Media, 2013. ISBN 978-1-4419-4786-4. eISBN 978-1-4757-2787. Dostupné z: doi:10.1007/978-1-4757-2787-6

[43] BURKARD, Rainer E, Stefan E KARISCH a Franz RENDL. QAPLIB–a quadratic assignment problem library. *Journal of Global optimization*. Springer, 1997, 10(4), 391-403. ISSN 1573-2916. Dostupné z: doi:10.1023/A:1008293323270,
(QAPLIB stránka přesunuta na: https://coral.ise.lehigh.edu/data-sets/qaplib/)

[44] MISEVIČIUS, Alfonsas, 2003. A Modified Simulated Annealing Algorithm for the Quadratic Assignment Problem. *Informatica* [online]. 2003-1-1, 14(4), 497-514 [cit. 2023-11-05]. ISSN 0868-4952. Dostupné z: doi:10.15388/Informatica.2003.037

[45] FLEURENT, Charles a Jacques FERLAND, 1994. Genetic hybrids for the quadratic assignment problem. In: PARDALOS, Panos a Henry WOLKOWICZ, ed. Quadratic Assignment and Related Problems [online]. Providence, Rhode Island: American Mathematical Society, 1994-8-17, s. 173-187 [cit. 2023-11-05]. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. ISBN 9780821866078. Dostupné z: doi:10.1090/dimacs/016/08

[46] STÜTZLE, Thomas a Holger H. HOOS, 2000. MAX–MIN Ant System. *Future Generation Computer Systems* [online]. 16(8), 889-914 [cit. 2023-11-05]. ISSN 0167739X. Dostupné z: doi:10.1016/S0167-739X(00)00043-1

[47] TAILLARD, E., 1991. Robust taboo search for the quadratic assignment problem. *Parallel Computing* [online]. 17(4-5), 443-455 [cit. 2024-02-26]. ISSN 01678191. Dostupné z: doi:10.1016/S0167-8191(05)80147-4

[48] CLAUSEN, Jens a Michael PERREGAARD, 1997. Solving Large Quadratic Assignment Problems in Parallel. *Computational Optimization and Applications* [online]. 8(2), 111-127 [cit. 2023-11-05]. ISSN 09266003. Dostupné z: doi:10.1023/A:1008696503659

[49] HOOS, Holger H a Thomas STÜTZLE. SATLIB - The Satisfiability Library [online]. Canada: Computer Science Department of the University of British Columbia in Vancouver, 2000 [cit. 2021-11-02].
Dostupné z: http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

[50] Public cloud resources for research. *Indiana University: Knowledge Base* [online]. 2021-09-22 [cit. 2021-10-02]. Dostupné z: https://kb.iu.edu/d/baom

[51] ANDERSON, David P., 2021. Globally Scheduling Volunteer Computing. *Future Internet* [online]. 13(9) [cit. 2024-02-07]. ISSN 1999-5903. Dostupné z: doi:10.3390/fi13090229

[52] ZIMMERMAN, Maxwell I., Justin R. PORTER, Michael D. WARD, et al.
SARS-CoV-2 Simulations Go Exascale to Capture Spike Opening and Reveal Cryptic Pockets Across the Proteome. *BioRxiv*. 2020.
Dostupné z: doi:10.1101/2020.06.27.175430

[53] SHAH, Agam. Proposed RISC-V vector instructions crank up computing power on small devices: When you need to do audio, voice or image processing at the network edge or on a battery budget. *The Register* [online]. 8 Oct 2021 [cit. 2021-11-10]. Dostupné z: https://www.theregister.com/2021/10/08/riscv_vector_instructions/

[54] CARBONNEAUX, Quentin. X86 Instruction Set Reference: Repeat String Operation Prefix. *c9x.me* [online]. [cit. 2021-11-11].
Dostupné z: https://c9x.me/x86/html/file_module_x86_id_279.html

[55] GREENGARD, Samuel. Will RISC-V revolutionize computing? *Communications of the ACM*. ACM New York, NY, USA, 2020, 63(5), 30-32. ISSN 0001-0782.
Dostupné z: doi:10.1145/3386377

[56] EHRETT, Pete, Todd AUSTIN a Valeria BERTACCO. Chopin: *Composing Cost-Effective Custom Chips with Algorithmic Chiplets.*
In: *2021 IEEE 39th International Conference on Computer Design (ICCD)*. 2021, s. 395-399. ISSN 2576-6996. Dostupné z: doi:10.1109/ICCD53106.2021.00069

[57] ARGONNE NATIONAL LABORATORY, 2019. *Argonne National Laboratory Deploys Cerebras CS-1, the World's Fastest Artificial Intelligence Computer* [online]. [cit. 2024-02-05]. Dostupné z: https://www.anl.gov/article/argonne-national-laboratory-deploys-cerebras-cs1-the-worlds-fastest-artificial-intelligence-computer

[58] SUBBIAH, Vishal, 2023. Argonne National Laboratory Brings Cerebras Computational Tools to the Forefront of Artificial Intelligence Research. Dostupné také z: https://8968533.fs1.hubspotusercontent-na1.net/hubfs/8968533/Case%20Studies/Cerebras-ANL-Case-Study.pdf

[59] PHARR, Matt a Randima FERNANDO. GPU Gems 2: Programming Techniques For High-Performance Graphics And General-Purpose Computation. 2. Boston: Addison-Wesley, 2005. ISBN 0-321-33559-7.
Dostupné také z: https://developer.nvidia.com/gpugems/gpugems2/inside-back-cover

[60] FANG, Jianbin, Chun HUANG, Tao TANG a Zheng WANG. Parallel programming models for heterogeneous many-cores: a comprehensive survey. *CCF Transactions on High Performance Computing* [online]. 2020, 2(4), 382-400 [cit. 2021-10-02]. ISSN 2524-4922. Dostupné z: doi:10.1007/s42514-020-00039-4

[61] NINOMIYA, Kai, Brandon JONES a Jim BLANDY, 2024. *WebGPU: W3C Working Draft* [online]. [cit. 2024-02-06]. Dostupné z: https://www.w3.org/TR/webgpu/

[62] KILGARIFF, Emmett, Henry MORETON, Nick STAM a Brandon BELL. NVIDIA Turing Architecture In-Depth. DEVELOPER BLOG [online]. NVIDIA, 2018, Sep 14, 2018 [cit. 2021-10-15].
Dostupné z: https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/

[63] AMD. *ROCm: Compatibility matrices* [online]. [cit. 2024-02-05]. Dostupné z: https://rocm.docs.amd.com/projects/radeon/en/latest/docs/compatibility.html

[64] WONG, Adrian. AMD CDNA Architecture: Tech Highlights!. TechARP [online]. [cit. 2021-10-08].
Dostupné z: https://www.techarp.com/computer/amd-cdna-architecture/

[65] CUDA Toolkit Documentation: CUDA C++ Programming Guide [online]. [cit. 2021-10-05]. Dostupné z: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[66] LIU, Rui, Lin FU, Bruno DE MAN a Hengyong YU. GPU-Based Branchless Distance-Driven Projection and Backprojection. *IEEE Transactions on Computational Imaging* [online]. 2017, 3(4), 617-632 [cit. 2022-01-21]. ISSN 2333-9403. Dostupné z: doi:10.1109/TCI.2017.2675705

[67] UZUN, Alp Bintuğ. Branchless programming. GitHub: alpbintug [online]. Jul 24, 2020 [cit. 2021-10-15]. Dostupné z: https://github.com/alpbintug/Branchless-programming

[68] DOBROVSKÝ, Ladislav. Clamp with if statements generate branchless x86-64 assembly. *CompilerExplorer* [online]. [cit. 2021-10-15]. Dostupné z: https://godbolt.org/z/MGf3Gsv93

[69] SHOSHANY, Barak, 2021. A C++17 Thread Pool for High-Performance Scientific Computing. *ArXiv e-prints* [online]. Dostupné z: doi:10.48550/arXiv.2105.00613

[70] CHAKRABARTI, Gautam, Vinod GROVER, Bastiaan AARTS, et al., 2012. CUDA: Compiling and optimizing for a GPU platform. *Procedia Computer Science* [online]. 9, 1910-1919 [cit. 2024-02-03]. ISSN 18770509. Dostupné z: doi:10.1016/j.procs.2012.04.209

[71] KRETZ, Matthias. s*td::experimental::simd: portable, zero-overhead C++ types for explicitly data-parallel programming* [online]. [cit. 2024-02-05]. Dostupné z: https://github.com/VcDevel/std-simd

[72] CPPREFERENCE.COM. Execution policy [online]. [cit. 2024-02-05]. Dostupné z: https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag

[73] NVIDIA, 2023. *cuRAND Library: Programming Guide* [online]. [cit. 2024-02-09]. Dostupné z: https://docs.nvidia.com/cuda/pdf/CURAND_Library.pdf

[74] OLSEN, David, Graham LOPEZ a Bryce Adelstein LELBACH. Accelerating Standard C++ with GPUs Using stdpar. NVIDIA. NVIDIA Developer Technical Blog [online]. [cit. 2024-02-01]. Dostupné z: https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar/

[75] LIN, Wei-Chen, Simon MCINTOSH-SMITH a Tom DEAKIN, 2024. Preliminary report: Initial evaluation of StdPar implementations on AMD GPUs for HPC. *ArXiv e-prints* [online]. [cit. 2024-02-01]. Dostupné z: doi:10.48550/arxiv.2401.02680

[76] VOICU, Alex. *[RFC] Adding C++ Parallel Algorithm Offload Support To Clang & LLVM* [online]. AMD. [cit. 2024-02-01]. Dostupné z: https://discourse.llvm.org/t/rfc-adding-c-parallel-algorithm-offload-support-to-clang-llvm/72159

[77] STOTKO, Patrick, 2019. Stdgpu: Efficient STL-like Data Structures on the GPU. *ArXiv e-prints* [online]. [cit. 2024-02-01]. Dostupné z: doi:10.48550/arXiv.1908.05936

[78] CCCL DEVELOPMENT TEAM, 2023. *CCCL: CUDA C++ Core Libraries* [online]. [cit. 2024-02-01]. Dostupné z: https://github.com/NVIDIA/cccl

[79] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 2018. *ISO/IEC TS 19570:2018: Programming Languages - Technical Specification for C++ Extensions for Parallelism*. Geneva.

[80] KRETZ, Matthias, 2023. *std::simd: merge data-parallel types from the Parallelism TS 2* [online]. [cit. 2024-02-01]. Dostupné z: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p1928r8.pdf

[81] CPPREFERENCE.COM. *SIMD library* [online]. [cit. 2024-02-01]. Dostupné z: https://en.cppreference.com/w/cpp/experimental/simd

[82] INTEL. *Intel® Implicit SPMD Program Compiler: An open-source compiler for high-performance SIMD programming on the CPU and GPU* [online]. [cit. 2024-02-05]. Dostupné z: https://ispc.github.io

[83] DU BOIS, Marissa, Pete BRUBAKER a Dominic MILANO. *Single Instruction Multiple Data Made Easy with Intel® Implicit SPMD Program Compiler* [online]. [cit. 2024-02-05]. Dostupné z: https://www.intel.com/content/www/us/en/developer/articles/technical/simd-made-easy-with-intel-ispc.html

[84] PHARR, Matt. The story of ispc. *Matt Pharr's blog* [online]. 2018 [cit. 2021-09-10]. Dostupné z: https://pharr.org/matt/blog/2018/04/18/ispc-origins

[85] ROUS, Jeff, 2019. *Unreal Engine* New Chaos Physics System Screams With In-Depth Intel CPU Optimizations* [online]. [cit. 2024-02-05]. Dostupné z: https://www.intel.com/content/www/us/en/developer/articles/technical/unreal-engines-new-chaos-physics-system-screams-with-in-depth-intel-cpu-optimizations.html

[86] THE MATHWORKS, INC. *Unreal Engine Scenario Simulation* [online]. [cit. 2024-02-02]. Dostupné z: https://www.mathworks.com/help/driving/unreal-engine-scenario-simulation.htm

[87] RIGO, Armin a Maciej FIJALKOWSKI. *CFFI documentation* [online]. [cit. 2024-02-01]. Dostupné z: https://cffi.readthedocs.io/

[88] WENZEL, Jakob. *Pybind11 Documentation* [online]. Jul 17, 2023 [cit. 2024-02-01]. Dostupné z: https://pybind11.readthedocs.io/_/downloads/en/stable/pdf/

9 PŘEHLED PUBLIKACÍ

[89] MATOUSEK, Radomil, Ladislav DOBROVSKY a Jakub KUDELA. How to start a heuristic? Utilizing lower bounds for solving the quadratic assignment problem. International Journal of Industrial Engineering Computations [online]. 2022, 13(2), 151-164. ISSN 1923-2934. Dostupné z: doi:10.5267/j.ijiec.2021.12.003

[90] MATOUSEK, Radomil, Ladislav DOBROVSKY a Jakub KUDELA. The Quadratic Assignment Problem: Metaheuristic Optimization Using HC12 Algorithm. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. New York, NY, USA: ACM, 2019, 2019-07-13, s. 153-154. ISBN 978-1-4503-6748-6. Dostupné z: doi:10.1145/3319619.3322088

[91] MATOUŠEK, Radomil, Tomáš HŮLKA, Ladislav DOBROVSKÝ a Jakub KŮDELA. Sum Epsilon-Tube Error Fitness Function Design for GP Symbolic Regression: Preliminary Study. In: *International Conference on Control, Artificial Intelligence, Robotics & Optimization (ICCAIRO)*. Athens, Greece: IEEE, 2019, 2019, s. 78-83. ISBN 978-1-7281-3572-4. Dostupné z: doi:10.1109/ICCAIRO47923.2019.00021

[92] HŮLKA, Tomáš, Radomil MATOUŠEK, Ladislav DOBROVSKÝ, Monika DOSOUDILOVÁ a Lars NOLLE. Optimization of Snake-like Robot Locomotion Using GA: Serpenoid Design. *MENDEL*. 2020, 26(1), 1-6. ISSN 2571-3701. Dostupné z: doi:10.13164/mendel.2020.1.001

[93] MIŠKAŘÍK, Kamil, Radomil MATOUŠEK, Ladislav DOBROVSKÝ a George HANNA. About controller design by means of grammatical evolution and bees algorithm. In: *MENDEL 2015: 21 st International Conference on Soft Computing*. Brno, 2015, s. 65-70. ISSN 1803-3814.

10 SEZNAMY ZK., POJMŮ, OBRÁZKŮ A TABULEK

ABI	Application Binary Interface – rozhraní binárních modulů, (dynamických) knihoven, v C++ závisí na kompilátoru a jeho verzi
AdaptiveCpp	implementace SYCL, dříve pod názvy hipSYCL a openSYCL
ALU	Arithmetic Logic Unit - aritmetickologická jednotka
AMD	Advanced Mirco Devices - společnost navrhující procesory
API	Application Programming Interface - rozhraní pro programování aplikací (případně komunikaci s nimi), v C++ datové typy a deklarace funkcí
ARM	Advanced RISC Machines (původně Acorn RISC Machine)
AVX	Advanced Vector Extensions
branch predictor	technika pro předpovídání výsledků instrukcí skoku
C++	obecný programovací jazyk
C++ AMP	C++ Accelerated Massive Parallelism
CEC	IEEE Congress on Evolutionary Computation
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture – výpočty na GPU NVIDIA
fat binary	spustitelný soubor, obsahující strojový kód pro více HW architektur
FMA	Fused Multiply Add: sdružené násobení a sečtení s jedním zaokrouhlením
GCC	GNU Compiler Collection *nebo* GNU C Compiler
GPU	Graphics Processing Unit
HCC	již nevyvýjený kompilátor pro GPU společnosti AMD
HW	hardware
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Standard Organization
ISPC	(Intel) Implicit SPMD Program Compiler
kernel	jádro; na GPU je to program, který se spusti v mnoha instancích; v HC12 algoritmu je současným řešením, které se modifikuje bitovými maskami
LLVM IR	LLVM intermediate reprezentation (LLVM není zkratka)
MMX	rozšíření instrukční sady x86; neoficiálně nazýváno MultiMedia eXtension
NP-úplný	úplný nedeterministicky polynomiální problém
NVCC	NVIDIA CUDA Compiler Driver - rozděluje kód pro CPU a GPU
NumPy	Numerical Python – lineární algebra a základní výpočty pro Python
OpenCL	Open Computing Language
pipelining	technika řetězení dílčích operací instrukcí, zvyšuje propustnost HW
PTX	Parallel Thread Execution
PRNG	Pseudo-Random Number Generator (generátor pseudonáhodných čísel)
QAP	Quadratic Assignment Problem - problém kvadratického přiřazení
QAPLIB	knihovna instancí QAP úloh
RISC-V	Reduced Instruction Set Computer – 5. verze
ROCm	Radeon Open Compute platform - software pro výpočty na AMD GPU
SDK	Software Developer Kit - sada nástrojů pro vývoj určitého software
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread (termín společnosti NVIDIA)
SM	Streaming Multiprocessor – větší jádra GPU NVIDIA
SPIR-V	Standard Portable Intermediate Representation - Vukan
SPMD	Single Program Multiple Data
SSE	Streaming SIMD Extensions
SYCL	původně "SYstem-wide Compute Language", nyní je samotné názvem
TSMC	Taiwan Semiconductor Manufacturing Company Limited
thread	vlákno programu

thread pool	skupina spuštěných vláken, které mají postupně provádět výpočetní úlohy
vyhození výjimky	používají se též termíny: vyvolání, způsobení, vyvhrnutí; (`throw`, `raise`)
W3C	The World Wide Web Consortium
WG21	ISO/IEC JTC1 (Joint Technical Committee 1) / SC22 (Subcommittee 22) / WG21 (Working Group 21) - komise standardizace jazyka C++
x86	instrukční sada procesorů firmy Intel a licenčních partnerů

Seznam obrázků

Seznam tabulek

11 PŘÍLOHY

Příloha A: Optimalizační funkce s reálnými parametry

F1: De Jong 1, Sphere function

$$f_2(\boldsymbol{x})=\sum_{i=1}^{D} i\,x_i^2 \qquad x_i\in\langle -5{,}12\,;5{,}12\rangle \tag{21}$$

F2: Axis parallel hyper-ellipsoid function

$$f_2(\boldsymbol{x})=\sum_{i=1}^{D} i\,x_i^2 \qquad x_i\in\langle -5{,}12\,;5{,}12\rangle \tag{22}$$

F3: Rotated hyper-ellipsoid function

$$f_3(\boldsymbol{x})=\sum_{i=1}^{D}\sum_{j=1}^{i} x_j^2 \qquad x_i\in\langle -65{,}536\,;65{,}536\rangle \tag{23}$$

F4: Moved axis parallel hyper-ellipsoid function

$$f_4(\boldsymbol{x})=\sum_{i=1}^{D} 5\,i\,x_i^2 \qquad x_i\in\langle -5{,}12\,;5{,}12\rangle \tag{24}$$

F5: Rosenbrock's valley, De Jong's function 2, Banana function

$$f_5(\boldsymbol{x})=\sum_{i=1}^{D-1} 100\left(x_{i+1}-x_i\right)^2+\left(1-x_i\right)^2 \qquad x_i\in\langle -2{,}048\,;2{,}048\rangle \tag{25}$$

F6: Rastrigin's function

$$f_6(\boldsymbol{x})=10\,D\sum_{i=1}^{D} x_i^2-10\cos\left(2\,\pi\,x_i\right) \qquad x_i\in\langle -5{,}12\,;5{,}12\rangle \tag{26}$$

F7: Schwefel's function 7

$$f_7(\boldsymbol{x})=\sum_{i=1}^{D} -x_i\sin\sqrt{|x_i|} \qquad x_i\in\langle -500\,;500\rangle \tag{27}$$

F8: Griewangk's function 8

$$f_8(\boldsymbol{x})=\sum_{i=1}^{D}\frac{x_i^2}{4000}-\prod_{i=1}^{D}\frac{x_i^2}{\sqrt{i}} \qquad x_i\in\langle -600\,;600\rangle \tag{28}$$

F9: Sum of different power function 9

$$f_9(\boldsymbol{x})=\sum_{i=1}^{D}|x_i|^{i+1} \qquad x_i\in\langle -1;1\rangle \tag{29}$$

F10: Ackley's Path function 10

$$f_{10}(\boldsymbol{x})=-20\exp\left(0{,}2\sqrt{\frac{1}{D}\sum_{i=1}^{D}x_i^2}\right)-\exp\left(\frac{1}{D}\sum_{i=1}^{D}\cos(2\pi x_i)\right)+20+e \qquad x_i\in\langle -32{,}768;32{,}768\rangle \tag{30}$$

F11: Langermann's function 11 (rozšířená pro N-D)

$$f_{11}(\boldsymbol{x})=\sum_{i=1}^{m}c_i\exp\left(-\frac{1}{\pi}\sum_{j=1}^{d}(x_j-A_{i\bmod(j,2)})^2\right)\cos\left(\pi\sum_{j=1}^{d}(x_j-A_{i\bmod(j,2)})^2\right) \qquad m=5 \qquad c=(1,2,3,5,2,3) \qquad A=\begin{pmatrix}3&5\\5&2\\2&1\\1&4\\7&9\end{pmatrix} \qquad x_i\in\langle 0;10\rangle \tag{31}$$

F12: Michalewicz's function 12

$$f_{12}(\boldsymbol{x})=-\sum_{i=1}^{D}\sin x_i\cdot\sin^{2m}\left(\frac{i\,x_i^2}{\pi}\right) \qquad m=10 \qquad x_i\in\langle 0;\pi\rangle \tag{32}$$

CEC04 je shodná s F6.

CEC05 je shodná s F8.

CEC06: Weierstrass's function

$$f_{CEC\,06}(\boldsymbol{x})=\sum_{i=1}^{D}\left(\sum_{k=0}^{k_{max}}a^k\cos\left(2\pi b^k(x_i+0{,}5)\right)\right)-D\sum_{k=0}^{k_{max}}a^k\cos(\pi b^k) \qquad a=0{,}5 \qquad b=3 \qquad k_{max}=20 \qquad x_i\in\langle -0{,}5;0{,}5\rangle \tag{33}$$

CEC07: Modified Schwefel's function 7

Funkce mod zde odpovídá v Matlabu funkci *rem*, v NumPy a CUDA C funkci *fmod*. Použití *mod* v Matlabu i NumPy vede k nesprávnému výsledku.

$$f_{CEC\,07}(\boldsymbol{x}) = 418{,}9829\,D - \sum_{i=1}^{D} g(z_i)$$

$$z_i = x_i + 420{,}9687462275036$$

$$g(z_i) = \begin{cases} z_i \sin\sqrt{|z_i|} & pro\ |z_i| \le 500 \\ (500 - \mathrm{mod}(z_i, 500))\sin\sqrt{|500 - \mathrm{mod}(z_i, 500)|} & pro\ z_i > 500 \\ (\mathrm{mod}(|z_i|, 500) - 500)\sin\sqrt{|\mathrm{mod}(z_i, 500) - 500|} & pro\ z_i < -500 \end{cases} \tag{34}$$

$$x_i \in \langle -500\,;\,500 \rangle$$

CEC08: Expanded Schaffer's F6 Function

$$f_{CEC\,08}(\boldsymbol{x}) = g(x_1, x_D) + \sum_{i=1}^{D-1} g(x_i, x_{i+1})$$

$$g(x, y) = \frac{1}{2} + \frac{\sin^2\sqrt{x^2 + y^2} - \frac{1}{2}}{1 + \frac{x^2 + y^2}{1000}} \tag{35}$$

$$x_i \in \langle -100\,;\,100 \rangle$$

CEC09: Happy Cat function

$$f_{CEC\,09}(\boldsymbol{x}) = \sum_{i=1}^{D} \sqrt[4]{|x_i^2 - D|} + \frac{0{,}5\sum_{i=1}^{D} x_i^2 + \sum_{i=1}^{D} x_i}{D} \tag{36}$$

$$x_i \in \langle -5\,;\,5 \rangle$$

CEC10 je shodná s F10 s odlišným rozsahem parametrů $x_i \in$ <-100; 100>.

Příloha B: Implementace funkce CEC07

Python + NumPy, vektorizace přes řádky

```
from numpy import abs, sum, sqrt, sin, zeros, ndarray, fmod

def CEC07(X: ndarray) → ndarray:
    Z = 420.9687462275036 + X
    C1 = abs(Z) ⩽ 500
    C2 = Z > 500
    C3 = Z < -500
    D = X.shape[1]
    gZ = zeros(X.shape)
    gZ[C1] = Z[C1] * sin(sqrt(abs(Z[C1])))
    gZ[C2] = ((500 - fmod(Z[C2], 500))
              * sin(sqrt(abs(500 - fmod(Z[C2], 500))))
              - (Z[C2]-500)**2 / (10000*D))
    gZ[C3] = ((fmod(abs(Z[C3]), 500) - 500)
              * sin(sqrt(abs((fmod(Z[C3], 500)-500))))
              - (Z[C3]+500)**2 / (10000*D))
    return 418.9829*D - sum(gZ, 1)
```

Matlab, vektorizace přes sloupce

```
function Y = CEC07(X)
    Z = 420.9687462275036 + X;
    C1 = abs(Z) ⩽ 500;
    C2 = Z > 500;
    C3 = Z < -500;
    D = size(X, 1);
    gZ = zeros(size(X));
    gZ(C1) = Z(C1) .* sin(sqrt(abs(Z(C1))));
    gZ(C2) = (500 - rem(Z(C2), 500)) ...
                .* sin(sqrt(abs(500 - rem(Z(C2), 500)))) ...
                - (Z(C2)-500).^2 / (10000*D);
    gZ(C3) = (rem(abs(Z(C3)), 500)-500) ...
                .* sin( sqrt( abs(rem(Z(C3), 500) -500 )) ) ...
                - (Z(C3)+500).^2/(10000*D);
    Y = 418.9829*D - sum(gZ, 1);
end
```

CUDA C, uložení paměti odpovídá HC12

```
__global__ void CEC07_kernel(
    REAL *R,
    size_t R_pitch,
    REAL *F,
    unsigned nParam,
    unsigned rows)
{
  unsigned idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx<rows)
  {
    REAL s=REAL(0);
    for(unsigned par=0; par<nParam; par++) // suma
    {
      REAL *col=(REAL*)((char*)R + par * R_pitch);
      REAL r=col[idx];
      REAL z = 420.9687462275036 + r;
      REAL gZ = -100000;
      if(fabs(z) <= 500) {
        gZ = z * sin(sqrt(fabs(z)));
      } else if(z > 500) {
        REAL zm500 = 500 - fmod(z, REAL(500));
        gZ = zm500 * sin(sqrt(fabs(zm500)))
            - (z-500)*(z-500) / REAL(10000*nParam);

      } else { // z < -500
        REAL a = fmod(fabs(z), REAL(500)) - 500;
        REAL b = sin( sqrt( fabs( fmod(z, REAL(500)) - 500 ) ) );
        REAL c = (z+500)*(z+500) / REAL(10000*nParam);
        gZ = a * b - c;
      }

      s += gZ;
    }

    F[idx] = 418.9829*nParam - s;
  }
}
```

Následné spuštění CUDA kernelu (tpb: počet vláken na blok):

```
unsigned blocks = rows / tpb + (rows % tpb ? 1 : 0);
CEC07_kernel<<< blocks, tpb >>>(R, R_pitch, F, nParam, rows);
```

CUDA C pro více populační DE

```
template<typename FP, typename FPf>
__global__ void fitness_CEC07_kernel(
    PitchedPtrGPU<FP> X, PitchedPtrGPU<FP> Y,
    PitchedPtrGPU<FP> Yt, // transformed Y, can be nullptr
    PitchedPtrGPU<FPf> Fx, PitchedPtrGPU<FPf> Fy,
    uint32_t n_populations, uint32_t n_members, uint32_t n_dims)
{
  DEMULTIPOP_FITNESS_m_p // init makro

  if(m < n_members && p < n_populations) {

    FPf s=FPf(0);

    for(int n=0; n<n_dims; n++) { // sum

      int Y_col = m;
      int Y_row = n_populations * n + p;
      FP r = Yt.p==nullptr ? Y(Y_col, Y_row) : Yt(Y_col, Y_row);
      FPf z = 420.9687462275036 + r;
      FPf gZ = -100000;

      if(fabs(z) ⩽ 500) {
        gZ = z * sin(sqrt(fabs(z)));

      } else if(z > 500) {

        FPf zm500 = 500 - fmod(z, Fpf(500));
        gZ = zm500 * sin(sqrt(fabs(zm500)))
             - (z-500)*(z-500) / FPf(10000*n_dims);

      } else if(z < -500) {

        FPf a = fmod(fabs(z), FPf(500)) − 500;
        FPf b = sin( sqrt( fabs( fmod(z, FPf(500)) - 500 ) ) );
        FPf c = (z+500)*(z+500) / FPf(10000*n_dims);
        gZ = a * b - c;

      }

      s += gZ;

    }

    FPf f = 418.9829*n_dims - s;

    DEMULTIPOP_FITNESS_COPY_BETTER_Y2X // makro selekce

  }
}
```

```
// CECfitness<FP, FPf> přidává metody get_func_num a call_transformed
template<typename FP, typename FPf>
struct FitnessCEC07: CECfitness<FP, FPf>
{
    dim3 tpb;
    FitnessCEC07(): tpb{16, 16}
    {}

    int get_func_num() final  // pro transformaci
    {   return 7;   }

    dim3 grid(uint32_t n_populations, uint32_t n_members)
    {
        return dim3{
            unsigned(split_to_blocks(n_members, tpb.x)),
            unsigned(split_to_blocks(n_populations, tpb.y)),
        };
    }

    void operator()(
        PitchedPtrGPU<FP> X, PitchedPtrGPU<FP> Y,
        PitchedPtrGPU<FPf> Fx, PitchedPtrGPU<FPf> Fy,
        uint32_t n_populations, uint32_t n_members, uint32_t n_dims
        ) final
    {
        PitchedPtrGPU<FP> Yt_empty{nullptr, 0};
        fitness_CEC07_kernel<FP, FPf> <<< grid(), tpb >>> (
            X, Y, Yt_empty,
            Fx, Fy,
            n_populations, n_members, n_dims
        );
        cudaLastErrorThrow();
    }

    void call_transformed( // použito s FitnessCECTransform wrapperem
        PitchedPtrGPU<FP> X, PitchedPtrGPU<FP> Y, PitchedPtrGPU<FP> Yt,
        PitchedPtrGPU<FPf> Fx, PitchedPtrGPU<FPf> Fy,
        uint32_t n_populations, uint32_t n_members, uint32_t n_dims
    ) final
    {
        fitness_CEC07_kernel<FP, FPf>
          <<< grid(n_populations, n_members), tpb >>> (
            X, Y, Yt,
            Fx, Fy,
            n_populations, n_members, n_dims
        );
        cudaLastErrorThrow();
    }

    Interval<FP> bounds(uint32_t dim)
    {
        return {-1000, 1000};
    }

    virtual ~FitnessCEC07()
    {}

};
```

Příloha C: GPU akcelerace výpočtu SAT problému – MCSX a DXAC

```
_global__ void SAT_MCSX_phase1(
    CLAUSE_IDX_TYPE *clauses, uint32_t *clauses_offsets,
    uint32_t *clauses_lengths, uint32_t n_clauses,
    X_TYPE *x_packed,
    uint32_t *n_satisfied)
{
    unsigned idx = blockIdx.x * blockDim.x + threadIdx.x;

    if(idx < n_clauses) {

        uint32_t offset = clauses_offsets[idx];
        uint32_t clause_length = clauses_lengths[idx];
        uint32_t n_satisfied_local = 0;

        for(uint32_t ci=0; ci<clause_length; ci++) {

            auto bidx = clauses[offset+ci];

            // NOTE: compiler should optimize /% to proper & with shifts
            auto xidx = (abs(bidx)-1) / (sizeof(X_TYPE)*8);
            auto xbshift = (abs(bidx)-1) % (sizeof(X_TYPE)*8);
            uint32_t b = 0 ≠ (x_packed[xidx] & (1 << xbshift));
            uint32_t result = (bidx > 0 && b) || (bidx < 0 && !b);

            n_satisfied_local += result;
        }

        n_satisfied[idx] = n_satisfied_local;
    }
}

__global__ void SAT_MCSX_phase2(
    uint32_t *n_satisfied, uint32_t n_clauses,
    uint32_t *clause_satisfied)
{
    unsigned idx = blockIdx.x * blockDim.x + threadIdx.x;

    if(idx == 0) {

        uint32_t clause_satisfied_local = 1;
        for(uint32_t ci=0; ci<n_clauses; ci++) {

            clause_satisfied_local =
                clause_satisfied_local && n_satisfied[ci];
        }

        *clause_satisfied = clause_satisfied_local;
    }
}
```

```
__global__ void SAT_DXAC(
    CLAUSE_IDX_TYPE *clauses, uint32_t *clauses_offsets,
    uint32_t *clauses_lengths, uint32_t n_clauses,
    X_TYPE *x_packed_T, size_t x_pitch, uint32_t x_rows,
    uint32_t *n_satisfied, size_t n_satisfied_pitch,
    uint8_t *is_satisfied)
{
   unsigned idx = blockIdx.x * blockDim.x + threadIdx.x;

   if(idx < x_rows) {

      uint8_t is_satisfied_local = 1;

      for(uint32_t clausule_i=0; clausule_i<n_clauses; clausule_i++) {

         uint32_t offset = clauses_offsets[clausule_i];
         uint32_t clause_length = clauses_lengths[clausule_i];
         uint32_t n_satisfied_local = 0;

         for(uint32_t ci=0; ci<clause_length; ci++) {

            CLAUSE_IDX_TYPE bidx = clauses[offset+ci];

          // NOTE: compiler should optimize /% to proper & with shifts
            auto xidx = (abs(bidx)-1) / (sizeof(X_TYPE)*8);
            auto xbshift = (abs(bidx)-1) % (sizeof(X_TYPE)*8);
           X_TYPE *x_packed_T_column=
                (X_TYPE*)((char*)x_packed_T + xidx * x_pitch);

            X_TYPE x_cell=x_packed_T_column[idx];

            uint32_t b = 0 =/= (x_cell & (1 << xbshift));
            uint32_t result = (bidx > 0 && b) || (bidx < 0 && !b);

            n_satisfied_local += result;
         }

         is_satisfied_local = is_satisfied_local && n_satisfied_local;

         uint32_t *n_satisfied_column = (uint32_t*)((char*)n_satisfied
                     + clausule_i * n_satisfied_pitch);

         n_satisfied_column[idx] = n_satisfied_local;

      }

      is_satisfied[idx] = is_satisfied_local;
   }
}
```

Příloha D: Simulace indexování matic X a Y pro DE

```
import sys
from itertools import product, starmap
from collections import namedtuple
from functools import reduce
import operator
import numpy as np

dim3 = namedtuple('dim3', ('x', 'y', 'z'))

def dim3_idx(dims: dim3, d_idx: dim3):
    return d_idx.z*dims.y + d_idx.y*dims.x + d_idx.x

tile_size = 8
n_pops = 20
n_members = 100
n_dims = 10

gridDim = dim3(
    x=n_pops // tile_size + 1 if n_pops % tile_size else 0,
    y=n_members // tile_size + 1 if n_members % tile_size else 0,
    z=n_dims
)

blockDim = dim3(tile_size, tile_size, 1)  # threads per block

# __shared__ memory
__shared__buffer = np.zeros((tile_size, tile_size, 3), np.uint32)

value = np.zeros((1, 1, 3))

n_threads = reduce(operator.mul, blockDim)
thread_contexts = [{} for _ in range(n_threads)]

X = np.zeros((n_members * n_dims, n_pops, 3), np.uint32)
X[:, :, 0] = 1 + np.tile(np.arange(n_pops).reshape((1, n_pops)),
                         (n_members*n_dims, 1))

X[:, :, 1] = 1 + np.tile(np.arange(n_members).reshape((n_members, 1)),
                         (n_dims, n_pops))
X[:, :, 2] = 1 + np.tile(np.repeat(np.arange(n_dims), n_members).reshape(
                         (-1, 1)), (1, n_pops))

Y = np.zeros((n_pops * n_dims, n_members, 3), np.uint32)

print(f'grid definition: {gridDim=}, {blockDim=}')

def kernel_PART01_X_to_shared(blockIdx, threadIdx):
    # from the point of view of X
    p = blockIdx.x * blockDim.x + threadIdx.x  # population number
    m = blockIdx.y * blockDim.y + threadIdx.y  # member number
    n = blockIdx.z                             # dimension number
    if m < n_members and p < n_pops:
        X_row, X_col = n * n_members + m, p
        # print(f'  {threadIdx=} read X row={X_row} col={X_col}')
        __shared__buffer[threadIdx.y, threadIdx.x, :] = \
            X[X_row, X_col, :]
```

```
def kernel_PART02_shared_to_Y(blockIdx, threadIdx):
    n = blockIdx.z                          # dimension number
    p_blk = blockIdx.x * blockDim.x
    m_blk = blockIdx.y * blockDim.y

    Y_col = m_blk + threadIdx.x
    Y_row = p_blk + n_pops * n + threadIdx.y

    if Y_col < n_members and p_blk + threadIdx.y < n_pops:
        Y[Y_row, Y_col, :] = \
            __shared__buffer[threadIdx.x, threadIdx.y, :]

def print_pmn_matrix(M, file=sys.stdout):
    for i in range(M.shape[0]):
        for j in range(M.shape[1]):
            print(M[i, j, 0], M[i, j, 1], M[i, j, 2],
                  sep=',', end=';', file=file)

        print(file=file)

def reversed_dim3(*v):
    return dim3(*reversed(v))

# "grid launch"
for blockIdx in starmap(reversed_dim3,
                        product(*map(range, reversed(gridDim)))):

    __shared__buffer[:] = 0

    print(f'———— {blockIdx=} ——————————————————')

    for threadIdx in starmap(reversed_dim3,
                             product(*map(range, reversed(blockDim)))):

        tcontext = thread_contexts[dim3_idx(blockDim, threadIdx)]
        tcontext.clear()
        kernel_PART01_X_to_shared(blockIdx, threadIdx, tcontext)

    print(f'__shared__ buffer:')
    print_pmn_matrix(__shared__buffer)

    # __sync_threads();
    print(' **********  __sync_threads(); ***************')

    for threadIdx in starmap(reversed_dim3,
                             product(*map(range, reversed(blockDim)))):

        tcontext = thread_contexts[dim3_idx(blockDim, threadIdx)]
        kernel_PART02_shared_to_Y(blockIdx, threadIdx, tcontext)

with open('tiled_trials_overlap_Y.csv', 'w') as f:
    print_pmn_matrix(Y, file=f)
```

Příloha E: Generování SAT pro testování prototypu

```
import numpy as np
from random import seed, randint, sample
from functools import reduce


x_type = np.uint32
x_type_bytes = x_type().nbytes
x_type_bits = x_type_bytes * 8

def get_bit(x, bit):
    idx, bshift = divmod(bit, x_type_bits)
    return x[idx] & (1 << bshift)


def is_satisfied(clause, x):
    n_satisfied = 0

    for bidx in clause:
        b = get_bit(x, abs(bidx)-1)
        isneg = bidx < 0
        pos_true = int(not isneg and b≠0)
        neg_true = int(isneg and b═0)
        n_satisfied += pos_true or neg_true

    return n_satisfied


if __name__ == '__main__':
    # semínko PRNG → reprodukovatelnost výsledků
    seed(321)
    n_clauses = 5
    n_variables = 20
    n_tests = 40

    clauses = [[] for _ in range(n_clauses)]
    variables = list(range(1, n_variables+1))

    print('SAT clauses:')

    for clause in clauses:
        clause_length = randint(2, n_variables)
        clause.extend(sample(variables, clause_length))

        for vi in range(clause_length):
            if randint(0, 1) == 1:
                clause[vi] = -clause[vi]

        print(*[{0:'~', 1:''}[x>0]+'x'+
              str(abs(x)) for x in clause], sep=' v ')

    print('Random evaluations:')
```

```
for xi in range(n_tests):
    x_vec = [randint(0, 1) for _ in range(n_variables)]

    pars, leftbits = divmod(len(x_vec), x_type_bits)
    if leftbits > 0:
        pars += 1
    x = np.zeros(pars, dtype=x_type)
    for bidx, value in enumerate(x_vec):
        if value:
            idx, bshift = divmod(bidx, x_type_bits)
            x[idx] |= 1 << bshift

    clauses_satisfied = tuple(is_satisfied(clause, x)
                              for clause in clauses)

    print(x_vec,
          x,
          clauses_satisfied,
          all(clauses_satisfied),
          sep=' → ')
```

Příloha F: Použité softwarové nástroje

Pro vypracování této práce byly používány různé softwarové nástroje a jejich neúplný výčet je zde určitým doporučením pro jejich vyzkoušení:

- Kancelářský balík: Libre Office (Writer, Calc, Math, Draw)
- Grafické editory: InkScape, GIMP
- Kompilátory: LLVM/clang, GCC, NVCC
- IDE (integrované vývojové prostředí):
  - JetBrains PyCharm Community Edition (Python)
  - Vistual Studio Code (C++, CUDA a ISPC pluginy)
- Python a knihovny: Matplotlib, Seaborn, NumPy, pandas
- Jupyter Notebook
- Operační systém: Linux Mint
- Klasický nástroj řešení závislostí: GNU Make
- Systém pro zprávu verzí zdrojového kódu: git
- PDF prohlížeče: Xreader, Evince
- Formátování citací z RIS nebo přímo doi podle normy ČSN: CitacePRO
- Konverze citací z BibTeX do RIS: https://www.bibtex.com/c/bibtex-to-ris-converter/