VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE HEURISTICKÝCH METOD DISKRÉTNÍ OPTIMALIZACE NA GPU

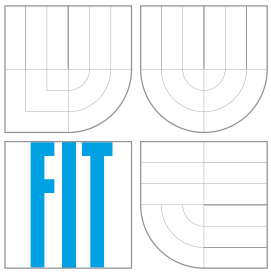
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. VÁCLAV PECHÁČEK

BRNO 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE HEURISTICKÝCH METOD DISKRÉTNÍ OPTIMALIZACE NA GPU

ACCELERATION OF DISCRETE OPTIMIZATION HEURISTICS USING GPU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. VÁCLAV PECHÁČEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. PETR POSPÍCHAL

BRNO 2012

Abstrakt

Práce se zabývá řešením diskrétních optimalizačních úloh. Zaměřuje se na zkrácení doby výpočtu s využitím heuristických metod a paralelismu. Teoretický základ tvoří kombinace algoritmů *ant colony optimization (ACO)* a lokálního prohledávání *k-optimization*. Platformu použitou při implementaci pak představuje technologie *Nvidia CUDA* umožňující efektivní provádění obecných výpočtů na moderních grafických čipech. Návrh využívá případové studie v podobě známého *problému obchodního cestujícího (TSP)*. Řešení je založeno na rozdělení úlohy na podproblémy s pomocí techniky *tour-based partitioning*, paralelním zpracování jednotlivých částí a jejich opětovném spojení. Vytvořený paralelní kód dokáže provádět výpočet více než sedmnáctkrát rychleji než jeho sekvenční verze.

Abstract

Thesis deals with discrete optimization problems. It focusses on faster ways to find good solutions by means of heuristics and parallel processing. Based on *ant colony optimization (ACO)* algorithm coupled with *k-optimization* local search approach, it aims at massively parallel computing on graphics processors provided by Nvidia CUDA platform. Well-known *travelling salesman problem (TSP)* is used as a case study. Solution is based on dividing task into subproblems using *tour-based partitioning*, parallel processing of distinct parts and their consecutive recombination. Provided parallel code can perform computation more than seventeen times faster than the sequential version.

Klíčová slova

akcelerace, diskrétní optimalizace, heuristika, ant colony optimization, ACO, lokální prohledávání, k-optimization, paralelní zpracování, GPU, GPGPU, Nvidia CUDA, problém obchodního cestujícího, TSP, tour-based partitioning

Keywords

acceleration, discrete optimization, heuristic, ant colony optimization, ACO, local search, k-optimization, parallel processing, GPU, GPGPU, Nvidia CUDA, travelling salesman problem, TSP, tour-based partitioning

Citace

Václav Pecháček: Akcelerace heuristických metod diskrétní optimalizace na GPU, diplomová práce, Brno, FIT VUT v Brně, 2012

Akcelerace heuristických metod diskrétní optimalizace na GPU

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Petra Pospíchala.

.....
Václav Pecháček
21. května 2012

Poděkování

Děkuji Ing. Petru Pospíchalovi za vynikající podporu během psaní diplomové práce. Kvalitní rady, rychlé odpovědi při vzájemné komunikaci a v neposlední řadě pomoc se zajišťováním referenčního hardware mi umožnily soustředit se na podstatu projektu. Děkuji také rodičům za podporu při studiu.

Obsah

Kapitola 1

Úvod

Růst výkonu hlavních procesorů počítačových systémů (CPU), který byl již od 70. let minulého století založen především na navyšování kmitočtu hodinového signálu, se v posledním desetiletí výrazně zpomalil. Příčinou byla zejména značná produkce odpadního tepla, kterou s sebou vyšší pracovní frekvence přinášely. Při hledání dalších cest k růstu výpočetního výkonu se mj. dostala do popředí *paralelizace*, která dříve nacházela využití spíše ve vědeckých a dalších speciálních oblastech [15, 17]. Běžné procesory tak dnes nabízejí několik souběžně pracujících jader. V průběhu let se však otevřel prostor také novému typu zařízení, který využívá paralelizaci mnohem masivněji – výpočetních jednotek obsahuje obvykle stovky. Tato zařízení se vyvinula z grafických čipů (GPU) úpravou hardware a zpřístupněním nástrojů pro tvorbu obecně zaměřeného kódu. Na trhu se rychle objevují nové modely, které nabízejí stále lepší parametry a nové možnosti [23, 17]. Výpočetní výkon moderních GPU našel uplatnění v nejrůznějších náročných aplikacích, jako je např. práce s částicovými modely, vyhodnocování magnetické rezonance [17] nebo sekvenování DNA [6].

Tato práce se zaměřuje na využití zmíněných zařízení v teoretičtější oblasti, kterou je diskrétní optimalizace. Přestože i s ní souvisí mnoho reálných úloh, jsou tyto formalizovány a jejich řešení probíhá nad více či méně vzdálenými matematickými reprezentacemi. Hlavním cílem studia jsou dvě heuristické metody – *ant colony optimization (ACO)* a *k-optimization*, přičemž za případovou studii byl zvolen známý *problém obchodního cestujícího (TSP)*. Cílovou platformou je potom Nvidia CUDA, která vedle samotného hardware grafických čipů nabízí také množství podpůrných softwarových nástrojů.

Úvodní kapitoly technické zprávy jsou věnovány teoretickým souvislostem nezbytným pro pochopení navazujících praktických částí práce. Zařazují zkoumané úlohy do kontextu matematické teorie, popisují podstatu heuristických metod a uvádí vlastnosti platformy Nvidia CUDA po hardwarové i softwarové stránce. Shrnují také výhody a nevýhody různých přístupů k paralelizaci. Text těchto kapitol vychází ze semestrálního projektu. Druhá část zprávy pak podrobně rozebírá návrh řešení i způsob jeho implementace. Závěr je věnován analýze dosažených výsledků, zejména pak srovnání rychlosti výpočtu na různých architekturách a efektivitě použitých metod.

Kapitola 2

Diskrétní optimalizace

Tato kapitola představuje diskrétní optimalizaci jako odvětví aplikované matematiky. Prezentuje silnou vazbu studovaných úloh na problémy z reálného prostředí, avšak věnuje se i formálnímu popisu a klasifikaci metod řešení. Druhá polovina kapitoly je zaměřena na problém obchodního cestujícího (TSP) jako zajímavý problém s jednoduchým zadáním, ale značnou výpočetní náročností a praktickými aplikacemi. V závěru je krátce zmíněn aktuální stav výzkumu na poli řešení TSP.

2.1 Motivace

Realita nás každý den staví před mnohá rozhodnutí. Někdy máme možností jen několik nebo si snadno a rychle dokážeme vybrat tu správnou. V jiných případech je však možností mnoho a jejich rozsah je za hranicí toho, co dokážeme intuitivně nebo bez dalších pomůcek uvážit. Představme si například situaci z akademického prostředí, kdy potřebujeme v rámci jediné fakulty naplánovat přednášky nebo zkoušky z desítek předmětů, přičemž máme sadu značně omezujících pravidel. Dvě akce se například nemohou konat zároveň v téže místnosti. Kantoři zase nemohou být na dvou místech najednou. Navíc bychom chtěli, aby se volitelné předměty studentům nepřekrývaly. Zároveň je obvykle k dispozici pouze omezené množství prostor určité velikosti nebo speciálního vybavení. Z ekonomického hlediska je žádoucí, aby malé skupinky studentů nebyly umisťovány do zbytečně velkých poslucháren. A takto bychom mohli pokračovat.

Plánování rozvrhu je jistě složité, ale specifická skupina problémů na nás klade ještě větší nároky – žádá si řešení rychle. Zatímco rozvrh lze vytvářet i několik dnů, v jiných situacích mohou být i minuty příliš velkým měřítkem. Představme si, že v autě na dálnici omylem mineme sjezd. Navigační systém by měl v řádu sekund najít novou cestu, jinak nestihneme využít ani několik sjezdů následujících. Přistává-li na letišti Chicago O'Hare více než 100 letadel za minutu, potřebujeme velmi rychle aktualizovat párování strojů na přistávací dráhy vždy, když se nějaký let zpozdí, nepodaří se manévr nebo dojde k bezpečnostnímu riziku. Podobné úlohy musí řešit spediční společnosti při kombinování malých zásilek do kamionů, MHD při vytváření jízdních řádů s ohledem na návaznost na velkých uzlech, továrny při rozvrhování práce v dílně a mnoho dalších subjektů.

Uvedené problémy mají společný charakter. Při jejich řešení se nabízí velké množství variant spočívajících v *kombinaci* dílčích rozhodnutí – umístění přednášky do posluchárny na daný čas, přiřazení letadla na přistávací dráhu, naložení určitého balíku do konkrétního kamionu. Tato rozhodnutí se označují jako *komponenty*. Vzhledem k vazbě na reálné

prostředí se přitom nemůžeme spokojit s libovolným řešením. Jednak jsou na nás kladena specifická omezení (v krátkodobém horizontu nelze např. přidat přistávací dráhu), jednak se snažíme využívat zdroje co nejefektivněji – vedení univerzity jen těžko kývne na stavbu nové posluchárny, která by byla využita dva dny v týdnu [8, 14].

Řešení úloh tohoto typu proto s ohledem na uvedené souvislosti označujeme jako *kombinatorickou optimalizaci* a studujeme jej v rámci *diskrétní optimalizace*, odvětví aplikované matematiky. Diskrétní charakter souvisí se skutečností, že dílčí rozhodnutí jsou celistvá – neuvažujeme u nich žádnou míru nebo částečné využití nabízených možností. Těžko bychom např. nechali letadlo přistát ze tří čtvrtin na dráhu 27 a ze zbývající čtvrtiny na dráhu 51. Tento přístup přitom nijak nevylučuje speciální případy, diskrétní hodnotou může být i rozhodnutí nechat stroj čekat [8, 9]. V dalších částech této kapitoly uvedu problematiku z matematického úhlu pohledu a zmíním typické úlohy používané při jejím studiu.

2.2 Úlohy a jejich vlastnosti

Diskrétní optimalizační problém (discrete optimization problem, DOP) je takový matematický optimalizační problém, jehož množina řešení je buď konečná, nebo spočetná. Zároveň musí existovat funkce, která každému prvku z této množiny dokáže přiřadit reálné číslo představující jeho ohodnocení. Cílem je najít řešení, jehož ohodnocení je nejmenší možné. Obvykle je tedy DOP definován jako dvojice (S, f), kde S je množina všech řešení a f je hodnoticí funkce představující zobrazení $f : S \mapsto \mathbb{R}$ [15]. Některé zdroje uvádějí ještě třetí složku Ω reprezentující množinu omezení (např. [14]), zatímco jiné označují S zkrátka jako množinu všech *přijatelných* řešení.

Každý prvek $s \in S$ představuje konkrétní řešení skládající se z n dílčích *komponent*, které označujeme c_{ij}. Ty reprezentují přiřazení dostupných diskrétních hodnot $v_i^j \in D_i = \left\{v_i^1, ..., v_i^{|D_i|}\right\}$ příslušným diskrétním proměnným X_i, $i \in \{1, ..., n\}$.

Literatura se rozchází v definici *kombinatorického problému (combinatorial optimization problem, COP)*. Zatímco např. v [15] je tento pojem ztotožňován s DOP, jiní autoři (např. [9]) jej popisují jako podtyp DOP, přičemž jako další větev uvádějí *problémy celočíselného lineárního programování (integer linear programming problems, IP)*. Dodávají však, že obě kategorie mají mnoho společných rysů.

V této práci se dále budu věnovat kategorii COP podle [9]. Půjde tedy o skupinu problémů, u nichž je množina řešení sice konečná, ale velmi rychle roste s velikostí instance. Nejlépe hodnocený prvek je proto obtížné nejen najít, ale také ověřit, že se skutečně o nejlépe hodnocený prvek jedná. Do této oblasti spadají například následující úlohy:

- *travelling salesman problem (TSP)*
- *matching problem (MST)*
- *knapsack problem*

S ohledem na zadání práce zúžím další výklad na problémy, které lze řešit metodou *ant colony optimization (ACO)*. Ty již nebývají v literatuře klasifikovány jako zvláštní skupina, ale můžeme u nich najít některé charakteristické vlastnosti. Nabízí se u nich například poměrně přirozená grafová reprezentace, což umožňuje řešit je mj. technikami pro prohledávání grafů. Dále souvisejí s reálnými ději v oblasti logistiky, plánování či automatizace, což je na jednu stranu činí zajímavými z pohledu praktického využití. Na druhou stranu

z pohledu teoretického patří mezi NP-těžké problémy a jejich časová náročnost tedy roste rychleji než polynomiálně s velikostí instance [14, 15]. Oproti jiným NP-těžkým kombinatorickým úlohám, např. zmíněnému *problému batohu (knapsack problem)*, je navíc není možné efektivně převést na tzv. *binary integer problems (BIPs)*, pro něž jsou známy nejvýkonnější algoritmy přesného řešení [9]. Do této kategorie patří například následující problémy:

- *travelling salesman problem (TSP)*
- *quadrature assignment problem (QAP)*
- *job shop scheduling problem (JSP)*
- *vehicle routing problem (VRP)*

Příslušné aplikace lze nalézt v [25, 16, 11]. Pro realizaci práce jsem si vybral první uvedený, tedy *problém obchodního cestujícího (travelling salesman problem, TSP)*. Jde o známou, avšak dodnes hodně zkoumanou úlohu, která se často používá jako reference při zavádění nových postupů. Zajímavá je nejen svou náročností a zřetelnou vazbou na problémy reálného světa, ale také tím, že další úlohy (JSP, VRP…) lze obvykle na TSP převést [9]. V dalším výkladu se proto na základě uvedených skutečností budu zabývat již pouze problémem obchodního cestujícího.

2.3 Problém obchodního cestujícího (TSP)

V této podkapitole nejdříve popíši TSP jako zvolený ukázkový kombinatorický problém. Ve druhé části pak nastíním možné přístupy k jeho řešení. Metodám ACO a k-optimization, které jsem skutečně použil, je však vyhrazen samostatný prostor v následující kapitole.

2.3.1 Popis problému a jeho vlastností

TSP se v základu odkazuje na skutečné plánování obchodního cestujícího. Ten musí zjednodušeně při vyřizování záležitostí navštívit určitý počet měst (každé z nich právě jednou) a vrátit se domů. Předpokládáme, že na pořadí zastávek nezáleží, ale cesty mezi nimi mohou mít různou cenu [9, 14].

Formálně je TSP definován následovně: máme-li množinu prvků M, pro jejíž každé dva prvky x, y je dáno číslo $d(x, y)$ reprezentující cenu cesty, je naším cílem najít posloupnost prvků $x_1, ..., x_n$, $x_i \in M$, $i \in \{1, ..., n\}$ takovou, že součet řady

$$d(x_1, x_2) + d(x_2, x_3) + ... + d(x_{n-1}, x_n) + d(x_n, x_1) \tag{2.1}$$

je nejmenší možný. Obvykle uvažujeme, že $d(x, y) \geq 0 \ \forall x, y \in M$ a dále že rovnost $d(x, y) = 0$ platí pouze v případě, že x a y jsou totožné. Platí-li navíc $d(x, y) = d(y, x) \ \forall x, y \in M$, jde o tzv. *symetrický TSP*. Nejméně standardní, ale i tak často uváděnou podmínkou je *trojúhelníková nerovnost*, tedy $d(x, y) + d(y, z) \geq d(x, z) \ \forall x, y, z \in M$. Splňují-li hodnoty $d(x, y) \ \forall x, y \in M$ všechny uvedené podmínky, představuje d tzv. *metriku*. O hodnotách potom můžeme přirozeně uvažovat jako o vzdálenostech [19].

Problém obchodního cestujícího nabízí různé varianty zadání i souvisejících omezení. Jednu alternativu jsem naznačil již v předchozím odstavci – jde o tzv. *asymetrický TSP*,

v němž $\exists x, y \in M : d(x, y) \neq d(y, x)$. Mezi další variace patří TSP s opakovanými návštěvami – v takovém případě nevadí, projede-li obchodní cestující některými městy více než jednou. Uvažovat můžeme také situaci, kdy máme k dispozici vícero obchodních cestujících a naším úkolem je zjistit, jaké budou jejich trasy a jaký je jejich optimální počet. Další varianty jsou uvedeny v [9], kde je zároveň řečeno, že tyto úpravy lze obvykle poměrně snadno převést na TSP v základním tvaru. O něco složitější je převod asymetrického TSP na symetrický – i tato konverze je však možná [18].

TSP je typickým příkladem NP-úplného problému. Důkaz lze založit např. na speciálním případu, kterým je hledání hamiltonovské kružnice v daném grafu [9, 19].

2.3.2 Klasifikace metod pro řešení problému

Při volbě konkrétní metody nás bude zajímat především požadovaný poměr mezi kvalitou a rychlostí řešení. Budeme-li totiž požadovat optimální řešení, bude to vzhledem k charakteru NP-těžkých problémů znamenat mnoho výpočetního času i pro menší instance [19]. Někdy však aplikace nalezení skutečně optimálního řešení vyžaduje. V takovém případě použijeme tzv. *metody přesného řešení*, mezi něž patří v cílové oblasti zejména:

- *brute force*
- *branch-and-bound*
- *cutting-plane*

Nejjednodušší přístup spočívá v použití *hrubé síly (brute force)*, tedy prozkoumání všech možností. Vzhledem k tomu, že velikost stavového prostoru narůstá exponenciálně s velikostí instance, je tento přístup časově velmi náročný již pro poměrně malé instance. Efektivnější metodou je např. *branch-and-bound*, která využívá předpočítané nebo za běhu získané limity pro výstup hodnoticí funkce. Tyto limity se označují jako *bounds* a umožňují vynechat vyhodnocování těch částí stavového prostoru, které nemohou optimální řešení obsahovat (*pruning*). Důležitým limitem je zejména tzv. *Held-Karp bound*, který se často používá jako metrika i pro jiné algoritmy včetně heuristických [7]. Třetí zmíněná metoda, *cutting plane*, pak využívá sady *omezení (constraints)* pro ohraničení relevantní části stavového prostoru a nejvíce se tak přibližuje metodám lineárního programování. Existuje také přístup uvedené metody kombinující označovaný jako *branch-and-cut* [9, 15].

Máme však možnost slevit z nároků na kvalitu řešení a získat tzv. *suboptimální řešení* ve výrazně kratším čase. Umožňuje nám to skupina algoritmů označovaných jako *heuristické a aproximační metody*. Těch je k dispozici více, uvedu jich tedy pouze několik:

- *nearest neighbour*
- *k-optimization*
- *ant colony optimization (ACO)*
- *River formation dynamics (RFD)*

První uvedený mechanismus patří mezi nejjednodušší – při přidávání nové komponenty volí nejbližší dosud nenavštívený uzel. Tento nenáročný postup však dává pro některé specifické instance velice špatné výsledky. O něco sofistikovanější je skupina metod označovaných jako *k-optimization*, které se snaží iterativně vylepšovat počáteční řešení (získané

jiným způsobem) s využitím malých změn s lokálním dopadem. Označují se proto jako *metody lokálního prohledávání* [7]. Poslední dva zmíněné postupy, ACO a RFD, patří mezi přírodou inspirované algoritmy, které využívají tzv. *agentů* v roli do určité míry nezávislých heuristických procedur [14, 20]. Heuristickým metodám budu vzhledem k zaměření práce věnovat zvláštní kapitolu 3.

Dosud jsem při hodnocení algoritmů používal spíše neurčité pojmy – zejména při zmínkách o výpočetní náročnosti a velikosti úloh. V tabulce 2.1 proto uvádím konkrétní hodnoty z nedávné historie přesného řešení TSP instancí, kterou sleduje *Georgia Institute Of Technology*. Tato instituce mj. zaštiťuje soutěž s hlavní cenou v hodnotě 1 000 USD pro toho, kdo nalezne přesné řešení TSP instance o velikosti 100 000 uzlů v podobě obrazu Mona Lisa. Výzkumné týmy začaly s řešením již v roce 2009 a přinejmenším do dubna 2012 žádný z nich cenu nezískal [10].

Označení	**Rok**	**Význam dat**	**Procesorový čas a použité CPU**
`d15112`	2001	města v Německu	22,6 let na Compaq EV6 Alpha @ 500 MHz
`sw24978`	2004	města ve Švédsku	88,4 let na Intel Xeon @ 2.8 GHz
`pla85900`	2006	VLSI aplikace	136 let na AMD Opteron 250 @ 2.4 GHz

Tabulka 2.1: Největší instance TSP, jejichž optimální řešení bylo nalezeno do dubna 2012. Číselná část označení vyjadřuje počet uzlů dané instance.

Kapitola 3

Heuristické a aproximační metody

Heuristickým metodám by se měla tato práce, jak už její název napovídá, věnovat především. Konkrétně je v zadání zmíněn algoritmus ant colony optimization (ACO)*, jehož popis tvoří podstatnou část této kapitoly. Jak však uvádějí sami autoři algoritmu ACO v [14], dobrých výsledků dosahuje ACO zejména v kombinaci s algoritmy lokálního prohledávání. Ty jsou proto krátce vysvětleny také, a to v rámci sekce 3.2.*

3.1 Ant colony optimization (ACO)

Tato část je věnována jádru práce, kterým je *algoritmus mravenčí kolonie (ant colony optimization, ACO)*. Nejdříve krátce uvedu přírodní inspiraci, která stála při jeho vzniku. Pokračovat budu formálním modelem algoritmu mravenčí kolonie a jeho vazbou na kategorii úloh zmíněnou v části 2.2. Nakonec popíši různé varianty algoritmu, které se podstupně vyvinuly od jeho objevení v roce 1992.

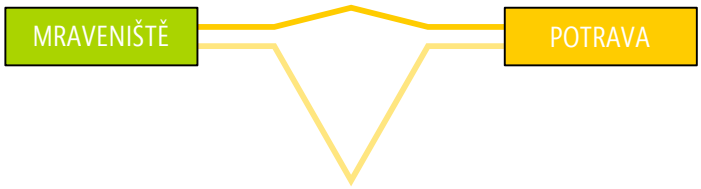
3.1.1 Biologická inspirace

Myšlenka algoritmu ACO vychází z chování mravenců, jejichž kolonie se organizují pomocí nepřímé komunikace založené na *feromonech*. To jsou látky, které mravenci uvolňují během svých cest – zejména za potravou. Jakmile určitý mravenec narazí na zdroj potravy, vrátí se s její částí do mraveniště. Po cestě přitom uvolňuje feromon, který má vliv na chování ostatních mravenců. Ti mají tendenci vybírat si cesty, na nichž je již nějaký feromon uložen (jde o tzv. *pheromone trails*). Sami při jejich následování feromonovou stopu posilují a „lákají“ tak další mravence.

Mohlo by se zdát, že by se celá kolonie mohla po určité době soustřeďovat v místě jediné stopy, která vznikla jako první. Ve hře je však další faktor – *odpařování feromonu*. Chemické látky se po určité době rozptýlí, není-li stopa dále posilována. K tomu může dojít např. v situaci, kdy je zdroj potravy již vyčerpán a mravenci se pro něj přestanou vracet. V tomto bodě si můžeme všimnout, že mravenec nespoléhá při rozhodování pouze na feromonové stopy, ale také na vlastní krátkodobou paměť.

Za průlomový bývá označován tzv. *double-bridge experiment* provedený v roce 1989. V něm připravili výzkumníci mravencům dva mosty na cestě za potravou – jeden kratší a jeden delší, jak je vidět na obrázku 3.1. Díky tomu, že po spodní cestě cestovali mravenci déle, nedokázali na ní obnovovat feromonovou stopu tak rychle, jako mravenci na cestě horní. Kratší cesta tedy přitahovala stále více mravenců, až se po ní začala přemisťovat celá kolonie. Jev, kdy na základě individuálních rozhodnutí mravenců (*mikroskopické hledisko*)

vzniká účelné, zdánlivě inteligentní kolektivní chování celé kolonie (*makroskopické hledisko*) jako v uvedeném příkladě, se nazývá *stigmergie* [14, 13].



Obrázek 3.1: Double-bridge experiment. Kolonie mravenců měla k dispozici dvě cesty mezi mraveništěm a zdrojem potravy. Díky tomu, že na kratší z nich byla feromonová stopa rychleji obnovována, volilo stále více mravenců právě ji. Po určité době od zahájení pokusu pak již horní cestu využívali všichni mravenci v kolonii.

Od realizace *double-bridge experimentu* a jeho variaci se stejně dlouhými mosty v roce 1990 (celá kolonie si přibližně v polovině případů vybrala horní most a v druhé polovině případů most dolní) byl již jen krůček k využití pozorovaného mechanismu pro řešení kombinatorických úloh. Dr. Marco Dorigo ve své disertační práci z roku 1992 navrhnul *ant system (AS)*, první ACO algoritmus, v němž nezávislé stochastické procedury simulují chování mravenců ve snaze nalézt pomocí stigmergie suboptimální, ale poměrně kvalitní řešení vybraných problémů [12].

3.1.2 Teoretický model a jeho aplikace

Matematický popis ACO je založen na tzv. *konstrukčním grafu*. Skládá-li se řešení kombinatorického optimalizačního problému $s \in S$ (definici jsem uvedl v kapitole 2.2) z konečného množství *komponent řešení* označených c_{ij}, kde i je pořadové číslo kroku v rámci řešení a j představuje index zvolené varianty v rámci možností dostupných v daném kroku (D_i), označíme množinu všech takovýchto komponent jako C. Konstrukční graf potom získáme přiřazením vrcholů nebo hran prvkům této množiny $c_{ij} \in C$.

Feromonové stopy zmíněné v předchozí podkapitole jsou reprezentovány hodnotami $\tau_{ij} \in \mathbb{R}$, které jsou přiřazeny každé komponentě c_{ij}. Mravenci samotní jsou potom nahrazeni procedurami, které vytvářejí řešení postupným přidáváním komponent („cestováním“ po konstrukčním grafu), přičemž využívají jak feromonovou stopu zanechanou ostatními mravenci, tak heuristickou informaci závislou na řešeném problému.

Podíváme-li se konkrétně na TSP, zjistíme, že vazba na uvedený mechanismus je poměrně přímočará. Hrany v grafu cest, které může obchodní cestující použít, přiřadíme hranám v konstrukčním grafu. Heuristická informace je v tomto případě dána vzdáleností měst z místa, v němž se mravenec právě nachází a z něhož vybírá další cestu [14, 8].

Podobně jednoduché je i mapování ACO na další úlohy, např. *job shop scheduling problem (JSP)*. Složitější úvahou lze najít propojení také na problém barvení grafu, *vehicle routing problem (VRP)* nebo *quadratic assignment problem (QAP)* [14, 25, 11, 16]. V dalším výkladu se budu zaměřovat na TSP vzhledem k jednoduché vazbě na ACO i skutečnosti, že je použit v praktické části práce.

3.1.3 Algoritmus

Implementace ACO mají obvykle podobnou hlavní kostru, kterou můžeme popsat pseudokódem 3.1. Na nejvyšší úrovni najdeme smyčku `SCHEDULE_ACTIVITIES`. Ta se opakuje, dokud nejsou splněny podmínky pro ukončení algoritmu. Mezi ně obvykle patří množství spotřebovaného procesorového času, počet iterací nebo nalezení řešení nad určitou (předem zadanou) hranici kvality. Může jít ale i o složitější vyhodnocení – běh můžeme ukončit např. v situaci, kdy dlouho nedojde k žádnému zlepšení.

```
ACO_METAHEURISTICS
  SCHEDULE_ACTIVITIES
    ConstructAntSolutions
    UpdatePheromones
    DaemonActions (optional)
  END_SCHEDULE_ACTIVITIES
END_ACO_METAHEURISTICS
```

Kód 3.1: Klíčové části algoritmu ACO.

Nyní podrobněji popíši jednotlivé procedury v rámci smyčky. Klíčovou částí je hned první z nich, tedy `ConstructAntSolutions`. Během ní umělí mravenci hledají svá řešení postupným přidáváním jednotlivých komponent, při jejichž výběru se řídí pravděpodobnostmi vypočítanými podle vzorce závislého na konkrétní variantě algoritmu. Zde uvedu vzorec z prvního algoritmu *ant system (AS)* z roku 1992:

$$p(c_{ij}|s^p) = \frac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{c_{il} \in N(s^p)} \tau_{il}^{\alpha} \cdot \eta_{ij}^{\beta}}, \forall c_{ij} \in N(s^p) \tag{3.1}$$

Vzorec popisuje pravděpodobnost volby komponenty c_{ij} v určitém stavu algoritmu. Ten odpovídá *částečnému řešení* s^p, tedy množině komponent vybraných v předcházejících krocích. Jediným dosud nevysvětleným symbolem v tomto vzorci je η_{ij}. Jedná se o reprezentaci heuristické informace. V případě TSP bychom ji získali jako převrácenou hodnotu ceny cesty, kterou obvykle uvažujeme jako vzdálenost mezi městy (více v kapitole 2.3.1), tedy $\eta_{ij} = 1/d_{ij}$. Poměr mezi feromonovou (paměťovou) a heuristickou složkou pravděpodobnosti je dán jejich vahami v podobě koeficientů α, β.

Jakmile všichni umělí mravenci vytvoří svá řešení, dojde k aktualizaci feromonových stop v rámci procedury `UpdatePheromones`. Každý mravenec uloží na své cestě množství feromonu odpovídající kvalitě jeho řešení – čím lepší řešení, tím více feromonu. Tím je zaručeno, že komponenty kvalitnějších řešení budou v následujícím cyklu `SCHEDULE_ACTIVITIES` zvýhodněny. V případě TSP za kvalitnější řešení považujeme kratší cesty a množství feromonu ukládané každým mravencem můžeme tedy určit na základě vzorce $\Delta\tau_{ij} = Q/L_k$, kde Q je konstanta a L_k je délka cesty představující řešení k-tého mravence.

V rámci procedury `UpdatePheromones` však nedochází pouze k přidávání feromonu, ale i jeho odebírání. Jde o *vypařování*, které je v algoritmu reprezentováno *mírou vypařování*

označenou ρ. Modifikace provedené v rámci procedury `UpdatePheromones` tak můžeme shrnout do vzorce

$$\tau_{ij} \leftarrow (1-\rho) \cdot \tau_{ij} + \sum_k \Delta\tau_{ij}^k, \tag{3.2}$$

kde poslední člen představuje součet příspěvků všech mravenců.

Poslední procedura `DaemonActions` zahrnuje pomocné úkony, které nesouvisejí se základní biologickou inspirací. Jde zejména o centralizované akce, které si žádají širší pohled, často i globální (je třeba pracovat s problémem jako celkem) – odtud také název, který se odkazuje na existenci jakéhosi *démona*, který má k dispozici více informací než jednotliví mravenci. Pseudokód v některých zdrojích obsahuje poznámku o volitelnosti této části. Ve skutečnosti ji prakticky vynechat nemůžeme, protože typickým příkladem využití `DaemonActions` je lokální optimalizace řešení nalezených mravenci. Slovo „lokální“ se však v tomto případě vztahuje ke skutečnosti, že změny jsou prováděny na lokální úrovni; k nalezení vhodných kandidátů je však třeba pohledu globálního. Jak nepřímo vyplývá z [14], mohou být úkony spadající pod `DaemonActions` prováděny i před aktualizací feromonových stop – např. právě u lokální optimalizace. Kvalitu cesty totiž lépe určíme až po optimalizaci, kdy se může ukázat její skrytý potenciál [14, 13, 8].

3.1.4 Varianty ACO

Od uvedení původního algoritmu AS v roce 1992 uplynulo mnoho let, během kterých se objevily různé variace ACO. Mezi nejvýznamnější bývá řazen *ant colony system (ACS)* z roku 1997. M. Dorigo a L. M. Gambardella zde zavedli tzv. *pseudonáhodné pravidlo*, které určuje, že mravenci se s určitou pravděpodobností označenou q_0 nebudou při volbě další komponenty řídit původním vzorcem z AS, ale pokusí se maximalizovat součin $\tau_{ij} \cdot \eta_{ij}^{\beta}$ – upřednostní tedy feromonovou složku na úkor heuristické. Výzkumníci zároveň zavedli mechanismus *lokální úpravy feromonové stopy*, která spočívá ve snížení míry feromonu na hraně vždy, když po ní nějaký mravenec podnikne cestu. Tím se snižuje riziko uváznutí v lokálním extrému a algoritmus pak snáze hledá nová řešení.

Další známou modifikací je *MAX-MIN ant system (MMAS)* z roku 2000, který se od původního AS liší zejména tím, že pouze mravenec s nejlepším řešením upravuje feromonovou stopu. Kromě toho, jak již název napovídá, klade MMAS dolní i horní omezení na množství feromonu τ_{ij} na každé hraně či uzlu konstrukčního grafu. Úrovně těchto omezení se často určují experimentálně, i když autoři MMAS T. Stützle a H. H. Hoos rozvíjeli také analytický přístup k jejich nalezení.

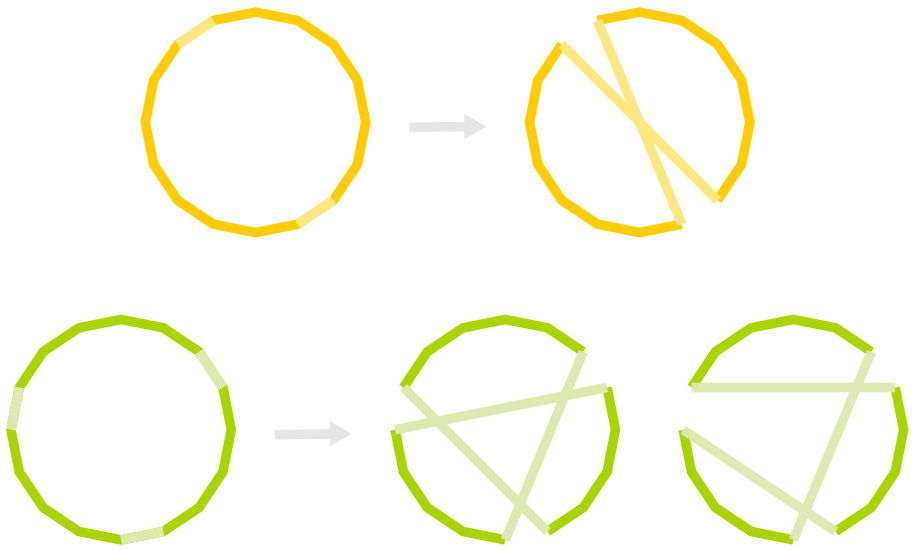
Existuje množství dalších variant. Některé využívají nejlepších cest nalezených v rámci všech iterací (*best-so-far solution*), jiné kombinují vícero nejlepších cest v rámci iterace. Zajímavým přístupem je např. také nahrazení délky cesty jejím pořadovým číslem (*rank*) v seznamu cest uspořádaném podle délk při rozmisťování feromonu (systém *ASrank*). Inovativní prvky lze kombinovat napříč různými ACO algoritmy a výkonnost různých konfigurací může záviset na typu problému i charakteru jednotlivých instancí [14, 13].

3.2 k-optimization

Již na začátku této kapitoly jsem zmínil, že ACO bez podpory algoritmů lokální optimalizace nedokáže příliš kvalitní řešení najít. Souvisí to s faktem, že ACO a metody lokální op-

timalizace se doplňují. Chceme-li totiž provést lokální optimalizaci, musíme mít k dispozici nějaké výchozí řešení, které budeme vylepšovat. Takové počáteční řešení přitom algoritmy ACO dokáží hledat poměrně dobře [14].

V této podkapitole se budu věnovat jednoduchým mechanismům známým pod společným názvem *k-optimization* (někdy zkráceně *k-opt*). Ty patří do skupiny tzv. *exchange heuristics* – algoritmů založených na provádění jednoduchých změn (také *tahů*), které lze opakovaně aplikovat na danou cestu s cílem zlepšit její kvalitu. V případě *k-opt* technik se jedná o odebrání k hran z dané cesty a opětovné přidání těchto hran tak, aby výsledná cesta byla jiná, ale zároveň šlo stále o korektní řešení – tzn. snažíme se vyhnout situaci, kdy by se v důsledku nových propojení rozdělila původní cesta na dvě. V případě běžně používaných hodnot k je platná alternativní konfigurace buď pouze jediná (pro $k = 2$), nebo jsou dvě (pro $k = 3$). Názorně to ukazuje obrázek 3.2.



Obrázek 3.2: Tahy 2-opt (oranžově) a 3-opt (zeleně). Světlejší barvou jsou znázorněny hrany, které byly v rámci tahu odebrány a následně znovu přidány tak, aby spojovaly jiné uzly. U horního případu existuje jediná možnost, u spodního si můžeme vybrat ze dvou variant.

Schémata jsou pouze ilustrativní – pokud bychom uvažovali plošnou vzdálenost vrcholů, nebyly by nové cesty kratší (tedy kvalitnější). V praxi hledáme a provádíme pouze takové tahy, které výslednou cestu zkrátí. Vyhodnocování všech možných tahů s sebou přináší velkou časovou složitost. Efektivnější postup, kdy jsou uvažovány pouze „slibné“ výměny, navrhnuli S. Lin a B. W. Kernighan již v roce 1973. Zároveň prezentovali vlastní algoritmus *Lin-Kernighan heuristic*, který k adaptivně mění při každém tahu [7, 22].

Lokální optimalizace je z hlediska práce velmi zajímavá dobrými předpoklady pro paralelizaci. Ta je dána již skutečností, že provádíme-li optimalizaci cest vytvořených více mravenci, jsou na sobě tyto procesy zcela nezávislé. Mechanismy *k-opt* však nabízí možnosti paralelizovat činnost i v rámci jedné cesty [7].

Kapitola 4

Využití GPU pro obecné výpočty

4.1 Historický vývoj

Výkonné paralelní výpočetní systémy v podobě programovatelných grafických karet mají vývojáři k dispozici již více než pět let. Dříve, než podrobně popíši vlastnosti a možnosti dnešních zařízení, zmíním obecné trendy v historii mikroprocesorů a změnu paradigmatu, která jednočipovým paralelním systémům otevřela dveře k širšímu využití. Dále vysvětlím, jak se z dedikovaných rozšiřujících karet pro podporu vykreslování staly vysoce paralelní obecné výpočetní platformy.

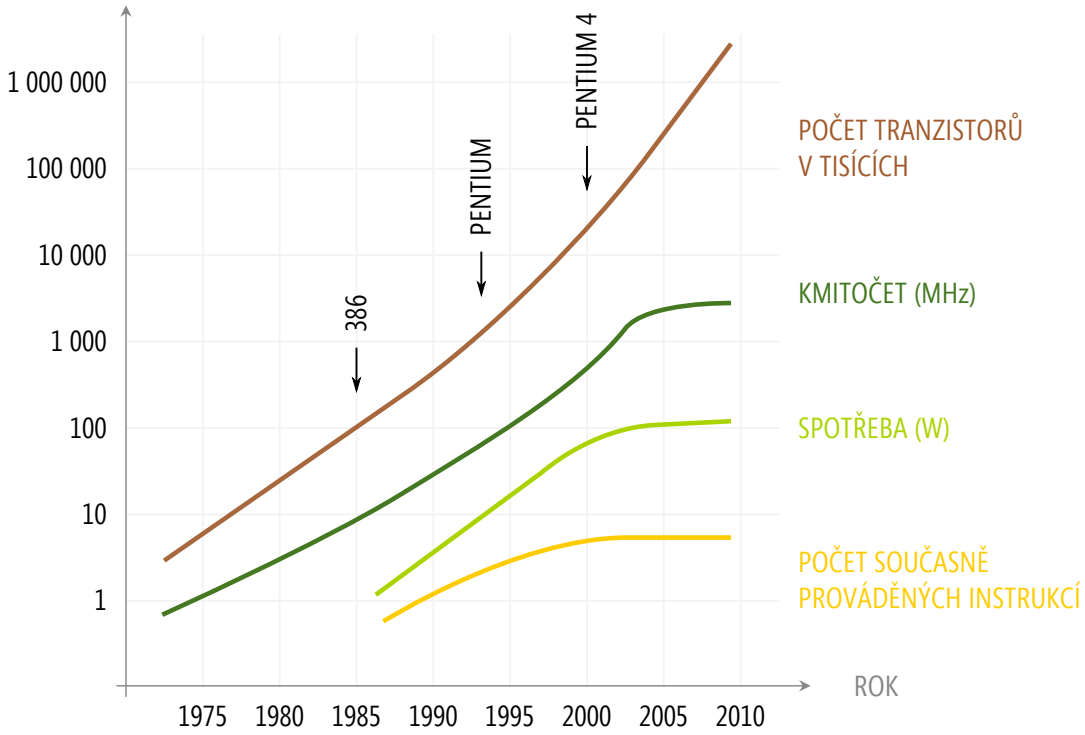
4.1.1 Cesta od jednoprocesorových systémů k paralelním

Vývoj v oblasti mikroprocesorů sledoval přibližně od poloviny sedmdesátých let několik trendů. Rychle se vyvíjející technologie výroby umožnila každoročně přibližně zdvojnásobit počet tranzistorů na běžném čipu. Stabilní růst bylo možné sledovat také v případě frekvence hodinového signálu. V 80. letech se začal výkon zvyšovat i díky řetězení instrukcí, překrývání paměťových operací a dalším technikám z oblasti *instruction level parallelism (ILP)*. Dlouhá léta tak bylo možné psát čím dál tím náročnější programy v očekávaní, že větší objem práce bude zpracován vyšším výkonem nového hardware. Programátoři se nemuseli o techniky zrychlování příliš zajímat, protože ty byly v rukou výrobců hardware.

Situace se však kolem roku 2003 dramaticky změnila. Zatímco vztah pro počet tranzistorů na čipu, který vstoupil ve známost jako *Moorův zákon*, platí dodnes, frekvence hodinového signálu v té době narazila na fyzikální limity používaných polovodičových součástek. Problém představovalo především teplo, které by při vysokých frekvencích čip vyzařoval a které by u běžných pracovních stanic nebylo možné efektivně odvést. Zároveň se zpomalil pokrok v oblasti ILP [17, 23, 24]. Uvedené souvislosti znázorňuje graf na obrázku 4.1.

Výrobci výkonných čipů se tak v posledních deseti letech mnohem více spoléhají na *paralelizaci*. Dostupnou plochu na čipu věnují vícenásobným jednotkám, označovaným jako *jádra (cores)*, které jsou do určité míry samostatnými procesory. Běžné sekvenční programy však obvykle nedokáží vícero jader využít. Na vývojáře software je tak nově kladen požadavek, aby programy více přizpůsobovali konkrétnímu hardware, chtějí-li dosáhnout vyššího výkonu. Tento zvrat popsal Herb Sutter ve svém článku *A fundamental turn toward concurrency in software* [24] již v roce 2004. Klíčem ke zrychlení je *paralelní kód* určený k současnému provádění na několika (případně mnoha) jádrech.

V oblasti návrhu paralelních systémů na jednom čipu můžeme sledovat dvě hlavní větve. Na přístup využívající řádově jednotek jader spoléhá nejen společnost Intel při výrobě



Obrázek 4.1: Trendy v parametrech CPU a jejich změna v posledním desetiletí. Zatímco počet tranzistorů se i nadále zvyšuje, růst ostatních hodnot se prakticky zastavil.

tradičních CPU, ale nově také např. Qualcomm při návrhu *system on chip (SoC)* řešení pro mobilní telefony. Jednotlivá jádra jsou přitom svou složitostí srovnatelná s dřívějšími jednojádrovými procesory. Reprezentanty kategorie označované jako *multicore* jsou např. CPU Intel i3, i5 a i7 postavené na architektuře Sandy Bridge [3].

Architektury pro paralelní počítání se však v posledních letech rozrostly o nový typ zařízení, který se vyvinul z grafických karet. Ta sází na jednodušší jádra, kterých jsou ovšem v rámci systému desítky nebo dokonce stovky. Do větve tzv. *many-core* zařízení patří např. Tesla M2090, jedna z nejvýkonnějších karet (září 2011) společnosti Nvidia. Právě této kategorii, na níž staví i má práce, se převážně věnuje následující kapitola [17].

4.1.2 Změna grafických karet ve výpočetní systémy

Moderní *graphical processing unit (GPU)* představuje masivně paralelní systém postavený na tzv. *many-core* přístupu, který jsem zmínil na konci předcházející podkapitoly. Výpočetním výkonem i paměťovou propustnostní přitom dokáže konkurovat moderním CPU. K tomuto stavu však vedl více než dvě desetiletí trvající vývoj grafických čipů.

Hardwarová podpora vykreslování dvoj- i trojrozměrných prvků začala nacházet širší uplatnění již v 80. letech s příchodem operačních systémů vybavených grafickým uživatelským rozhraním. Vhod přišla také filmovým studiím, vědcům a vojákům při modelování nejrůznějších situací. Od začátku devadesátých let byly na akceleraci trojrozměrného vykreslování kladeny stále náročnější požadavky, které přicházely a rostly společně s herním

průmyslem, zejména pak s akčními hrami založenými na tzv. *first person view*. Rostoucí zájem hráčské obce umožnil společnostem jako 3dfx, Nvidia nebo ATI investovat do vývoje grafických čipů čím dál tím větší prostředky [23].

Dlouhou dobu byla akcelerace založena na dedikovaných logických obvodech, které prováděly příslušné části řetězeného zpracování obrazu (*graphics pipeline*) znázorněného na obrázku 4.2. Nové aplikace si však žádaly stále složitější a rozmanitější úkony, které už výrobci nemohli v rámci pevné logiky nabídnout. Logickým vyústěním byl nápad, aby dva klíčové prvky *graphics pipeline* – tzv. *vertex shader* a *pixel shader* – byly programovatelné. Za přelomový se považuje rok 2001, kdy Microsoft tento záměr začlenil do specifikace DirectX 8.0 a společnost Nvidia jej realizovala na své kartě GeForce 3 [17, 23].

Značný výkon GPU zaujal již v této době vědce zabývající se vysoce náročnými výpočty. Ti našli způsob, jak do podoby grafických dat konvertovat údaje pocházející z jiných oblastí výzkumu. GPU zařízení tedy prováděla běžné vykreslovací operace v rámci možností programovatelných *shader* jednotek. Výstup však nebyl určen k zobrazení, ale ke zpětné konverzi dat na výsledky požadovaných výpočtů. Tento postup byl nejen značně komplikovaný sám o sobě, ale vyžadoval navíc použití speciálních jazyků (*shading languages*), jakými byly např. Cg, HLSL nebo GLSL. Otevřel se tak prostor pro další zdokonalení GPU v oblasti programovatelnosti pro obecné výpočty – *general purpose computation on graphical processing units (GPGPU)*.

Průlomovým se stal rok 2006, kdy Nvidia vydala kartu GeForce 8800. Ta měla na čipu sadu jednotných procesorů, přes něž procházela grafická data opakovaně, přičemž na ně byl pokaždé aplikován jiný program sestavený z univerzální sady instrukcí. Přestože mezi jednotlivými průchody byla někdy použita dedikovaná logika, zastávaly zmíněné procesory funkce všech tří základních typů *shader* jednotek. Takový návrh označujeme jako *unified shader architecture*. Znázorněn je na obrázku 4.2.

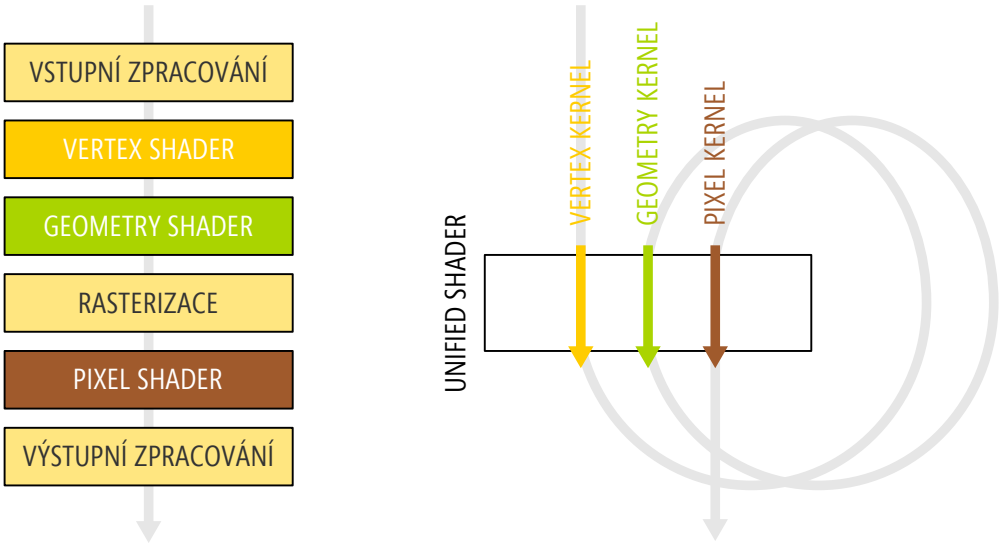
Nvidia šla však v případě GeForce 8800 ještě dál. Doplnila jednotky *unified shader* instrukcemi určenými přímo pro obecné výpočty. Kromě podpory IEEE standardů pro práci s čísly v pohyblivé řádové čárce šlo zejména o instrukce pro neomezený přístup do paměti, a to nejen paměti hlavní, ale i uživatelem řízené vyrovnávací paměti (*cache*) označované jako *shared memory*. Architektura, která dostala jméno *compute unified device architecture (CUDA)*, byla o několik měsíců později doplněna překladačem jazyka *CUDA C*, rozšíření standardního C/C++ umožňujícího provádění obecných výpočtů na GPU. Tyto kroky zpřístupnily GPGPU širokému spektru vývojářů [17, 23].

4.2 Aktuální stav

Dnes (2012) se na trhu nachází velké množství grafických karet, na nichž lze provádět obecné výpočty. Najdeme mezi nimi především produkty společností Nvidia a AMD. V následující podkapitole se zaměřím na obecnou charakteristiku těchto zařízení, provedu srovnání s běžnými CPU a zmíním nástroje pro jejich programování.

4.2.1 Umístění GPU v rámci systému

Grafická jednotka zvládající obecné výpočty je do systému typicky připojena přes rozhraní PCI Express. Přestože jde o velmi rychlé rozhraní a pro mnohé grafické aplikace stačí 4 linky (*PCIe 4x*), využívá většina zařízení spoj o šířce 16 linek. Ten nabízí ve verzi PCI Express 2.0 propustnost 8 GB/s v každém směru. V rámci jednoho systému lze zapojit a používat několik GPGPU zařízení najednou. Může jít např. o kombinaci integrované



Obrázek 4.2: Řetězené zpracování obrazu. Původně procházela data sadou dedikovaných obvodů, mj. třemi různými *shader* jednotkami (vlevo). Společnost Nvidia představila v roce 2006 architekturu využívající jediné univerzální *shader* jednotky, přes kterou prochází data opakovaně. Činnosti náležející dříve logickým obvodům jsou v univerzální jednotce prováděny odpovídajícími programy (*kernels*).

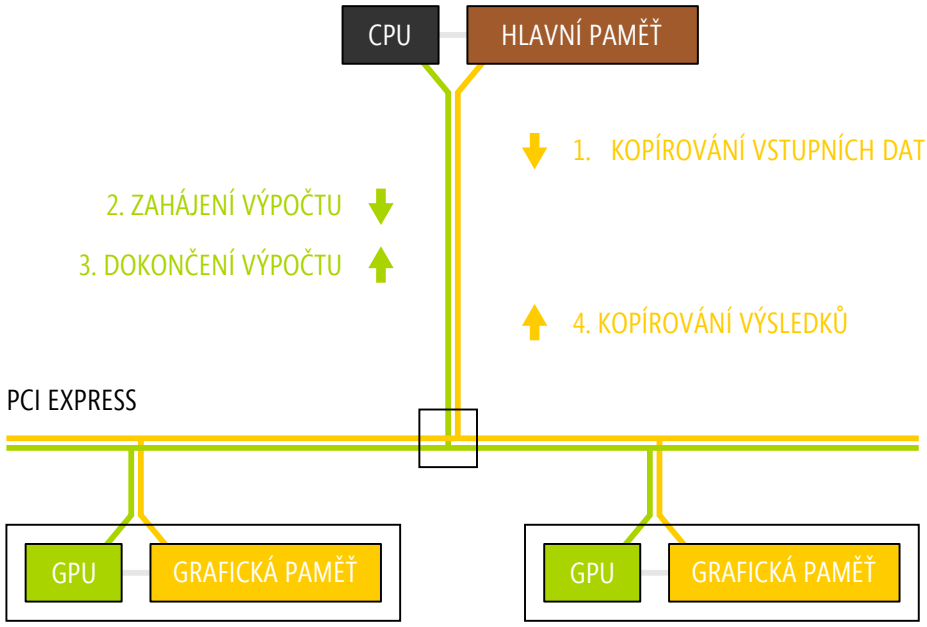
grafické jednotky (*integrated graphics processor, IGP*) a rozšiřující karty nebo dokonce více rozšiřujících karet. V druhém případě se zařízení nemusí spoléhat pouze na PCI Express spoje, protože výrobci nabízejí i proprietární technologie pro přímé propojení karet, jako je *Nvidia SLI* nebo *AMD Crossfire* [23, 6, 4].

Zatímco IGP se vyznačují tím, že využívají přidělenou část hlavní systémové paměti, rozšiřující karty disponují pamětí vlastní. Její velikost se pohybuje od 512 MB u zařízení v ceně kolem 30 USD až po 6 GB u špičkové karty Nvidia Tesla M2090. Technologicky jde o paměť podobnou paměti hlavní, která je optimalizována pro grafické operace a označuje se jako *graphics double data rate memory (GDDR memory)* [23, 6]. Vazby jednotlivých prvků v systému jsou znázorněny na obrázku 4.3.

Základní desky běžných pracovních stanic běžně nabízejí 1-2 sloty vhodné pro rozšiřující grafické karty. V serverových aplikacích lze osadit karet více – např. instalaci osmi zařízení Tesla C2050 ve 4U serveru můžeme zakoupit od společnosti *HPC systems* [5].

Mnohé grafické jednotky mají dnes v oblasti počítání nad čísly s plovoucí řádovou čárkou a paměťové propustnosti lepší parametry než běžné procesory [1]. Jádrem počítačových systémů je však stále CPU, které se proto nezbytně musí na výpočtu na GPGPU podílet. V minimálním případě zkopíruje CPU vstupní data na GPGPU zařízení, spustí kód a po dokončení výpočtu si vyzvedne výsledek. Tento proces je znázorněn na obrázku 4.3.

Vzhledem k tomu, že výpočty na GPGPU zařízení mají určitou režii, rozdělujeme v praxi i užitečnou práci mezi zmíněné prvky. CPU svěříme procedury, které nejsou vhodné pro paralelizaci nebo jsou příliš krátké. GPU pak necháme vykonávat složitější části, u nichž dokážeme využít *many-core* orientaci *zařízení*. Uvedený přístup označujeme jako *heterogenní programování* [1].



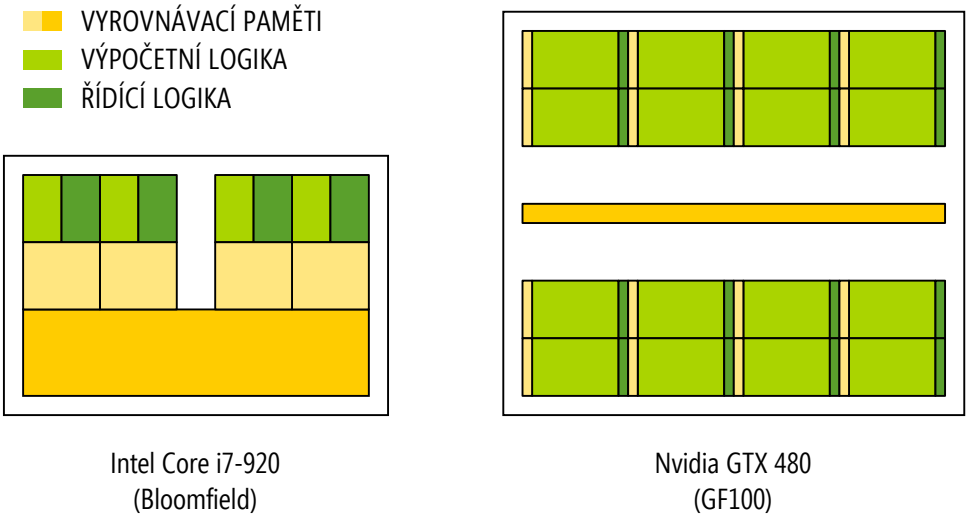
Obrázek 4.3: Integrace GPU do systému. Zeleně jsou znázorněny řídící cesty, oranžově cesty datové – všechny ve skutečnosti vedou po rozhraní PCI Express. Dále jsou uvedeny kroky nezbytné pro provedení výpočtu se šipkami vyjadřujícími směr komunikace.

4.2.2 Architektura současných GPU a srovnání s CPU

Zatímco počet jader u dnešních CPU výjimečně přesahuje desítku i v případě špičkových serverových produktů[1], již základní GPU nabízejí jader řádově stovky. Tento rozdíl je dán několika faktory. Jednak bývá plocha na grafických čipech často větší. Srovnejme např. 263 mm^2 u CPU Intel i7 vyráběného 45nm technologií (731 milionů tranzistorů) s 529 mm^2 u GPU Nvidia GTX 480 vyráběného technologií 40nm (3 miliardy tranzistorů) [2, 6]. Dále jsou významné části procesorových čipů věnovány vyrovnávacím pamětem, zejména L3 cache. Zatímco např. běžné CPU pro pracovní stanice Intel i5 nabízí přinejmenším 3 MB L3 cache, kapacita pamětí umístěných přímo na čipu se i u vyšších řad karet Nvidia GeForce GTX 5xx pohybuje pouze kolem 0,5 MB [3, 6]. Především jsou však jádra grafických karet výrazně jednodušší. Jejich architektura je orientována na počítání nad čísly s plovoucí řádovou čárkou a spoléhá se na sdílenou řídící logiku, která byla navržena s ohledem na přístup *same instruction, multiple thread (SIMT)* [1]. Není zatížena snahou o složité *out-of-order (OOO)* provádění instrukcí jako sofistikovaná jádra moderních procesorů. Uvedené souvislosti jsou znázorněny na obrázku 4.4.

Podíváme-li se na GPU čip podrobněji, uvidíme hierarchickou strukturu podobnou té, která je znázorněna na obrázku 4.5. V tomto případě jde konkrétně o architekturu Nvidia Fermi, ale hlavní rysy mají obecný charakter. Samotná jádra (*streaming processor, SP*) jsou sdružována do bloků označovaných jako *streaming multiprocessor (SM)*. Důležitá je především skutečnost, že v rámci SM je k dispozici uživatelem řízená vyrovnávací paměť, tzv. *shared memory*. Ta zároveň sdílí HW prostředky s vyrovnávací pamětí L1. Nad úrovní

[1] např. 12-jádrová CPU řady Intel Xeon E7



Obrázek 4.4: Rozdíly v rozložení plochy na čipu CPU a GPU. Na čtyřjádrovém CPU architektury Core (nalevo) jsou významně zastoupeny vyrovnávací paměti a řídící logika umožňující provádění instrukcí *out-of-order (OOO)*. Na GPU architektury Fermi (vpravo) naopak můžeme vidět velkou plochu věnovanou výpočetním prvkům. Schémata odpovídají skutečné struktuře i poměrné velikosti uvedených čipů.

SM vidíme společnou řídící logiku, vyrovnávací paměť L2, rozhraní pro přístup k hlavní paměti a pro komunikaci s hostitelským systémem [17, 6].

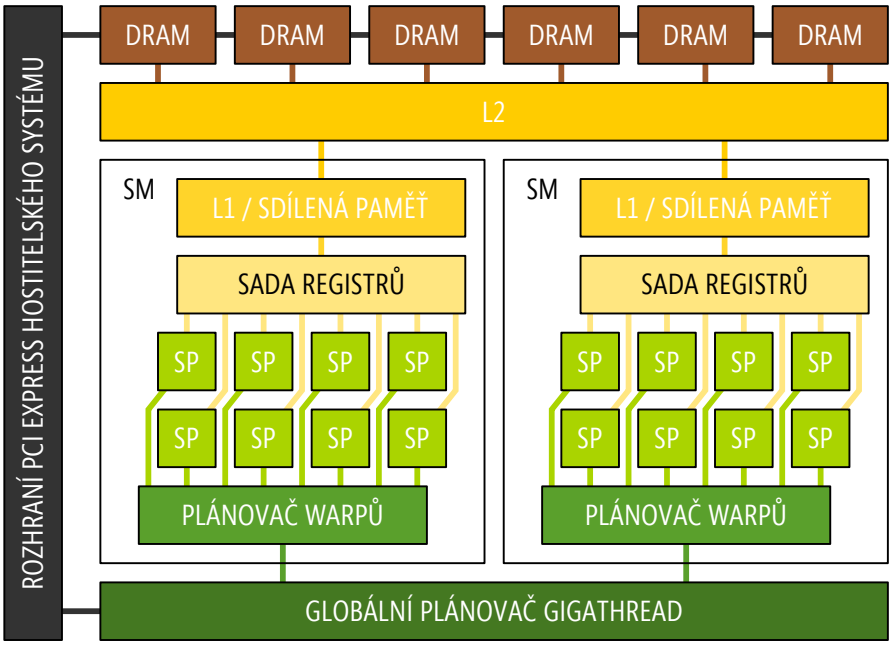
Díky specializaci hardware a své paralelní povaze dosahují *zařízení* velmi vysokých teoretických hodnot v oblasti rychlosti výpočtů i propustnosti paměti. Karta Nvidia GTX 580 například nabízí výkon přesahující 1500 GFLOP v porovnání s přibližně 200 GFLOP u procesorů Intel postavných na architektuře Westmere. Podobný poměr vychází mezi rychlostí 200 GB/s, kterou údajně dokáže GPU komunikovat s hlavní pamětí, a 35 GB/s u zmíněného serverového CPU [1].

Absolutním číslům ve specifikacích GPGPU napomáhá fakt, že jde o výpočty nad čísly s jednoduchou přesností. Provádět výpočty nad čísly s dvojitou přesností totiž grafické jednotky až do roku 2007 nedokázaly vůbec a i dnes je práce nad nimi výrazně pomalejší (přibližně 12x). U CPU takovýto rozdíl není, zpracování 32-bitového i 64-bitového formátu je přibližně stejně rychlé. Pro mnohé aplikace je však základní přesnost postačující [23, 1].

4.2.3 Vývojové nástroje

Programy pro GPGPU lze v současné době vyvíjet na několika sadách nástrojů. V předchozích podkapitolách jsem již zmínil platformu CUDA společnosti Nvidia a jazyk CUDA C. CUDA však není omezena na pouze na něj – nabízí totiž podporu také pro další jazyky, např. Fortran, Python nebo OpenCL [6]. Poslední jmenovaný představuje otevřený standard vyvinutý společností Khronos group a najdeme jej také v produktech dalších výrobců. Staví na něm např. AMD se svou platformou *Accelerated parallel processing (APP)* [4].

K realizaci práce jsem si vybral platformu CUDA zejména proto, že na školním GPU serveru i v laboratořích jsou instalovány karty společnosti Nvidia, u nichž CUDA poskytuje užší propojení na architekturu hardware.



Obrázek 4.5: Architektura Nvidia Fermi. Zelenými odstíny jsou znázorněny prvky řídící cesty, žlutými potom prvky cesty datové. Schéma je pouze ilustrativní – zařízení ve skutečnosti obsahují mnohem více jednotek SP i SM. Počet hnědě vyobrazených řadičů pamětí GDDR5 však realitě odpovídá.

4.3 NVIDIA CUDA

Následující část je věnována platformě Nvidia CUDA jakožto prostředku zvoleném pro realizaci práce. Popíši zejména model řízení vláken a typologii dostupných pamětí. Zmíním také základní informace o aktuální nabídce společnosti Nvidia.

4.3.1 Zařízení a jejich klasifikace

Nvidia nabízí širokou škálu grafických jednotek podporujících CUDA, které ve svých materiálech označuje jako *zařízení (device)*. Mainstream představují karty GeForce určené pro běžnou práci a hraní počítačových her. K dispozici jsou také řady Quadro a NVS mířící na profesionální aplikace. Na špičce stojí produkty Tesla určené pro vysoce náročné výpočty (*high performance computing, HPC*) [6, 1]. Pro srovnání uvádím základní parametry vybraných karet nejnovější architektury *Fermi* v tabulce 4.1.

Architektura CUDA jader se postupně vyvíjí a s ní se mění i vlastnosti jednotlivých čipů. Aby mohli vývojáři snadno zjistit, jaké možnosti konkrétní zařízení nabízí, zavedla Nvidia pojem *compute capability*. Skládá se ze dvou čísel zapisovaných podobně jako např. různé verze software. Nejnovější významnou inovací je již zmíněná architektura Fermi, jejíž čipy mají *compute capability* ve tvaru 2.x. Navýšení posledního čísla označuje menší vylepšení, přičemž k dispozici jsou v tuto chvíli zařízení s označením 2.1 [1].

Zařízení	Počet SM	Počet jader	Velikost paměti
GeForce GTS 450	4	192	1 GB
GeForce GTX 580	16	512	1,5 GB
Quadro 2000	4	192	1 GB
Tesla M2070	14	448	6 GB

Tabulka 4.1: Základní parametry vybraných karet architektury Fermi společnosti Nvidia.

4.3.2 Tok řízení

Heterogenní programování, jak jsem uvedl již dříve, spočívá v rozložení toku řízení mezi CPU a GPU. Ucelená část kódu, která je prováděna na zařízení, se v terminologii CUDA nazývá *kernel*. Definici kernelu ve zdrojovém kódu rozpoznáme od definice běžné funkce podle klíčového slova `__global__` umístěného na začátku hlavičky. Na další netypické prvky narazíme v místě volání kernelu. Vzhledem k SIMT povaze CUDA je třeba definovat míru paralelismu, s jakou má být kernel prováděn. Potřebné doplňující parametry předáme pomocí notace využívající úhlových závorek, která bude popsána dále. Jakmile CPU při provádění programu dorazí k takovému volání, spustí paralelní kód kernelu na zařízení. Poté může program pokračovat sekvenčním kódem pro CPU, ale také dalším voláním kernelu. Uvedený průběh je znázorněn na obrázku 4.6.

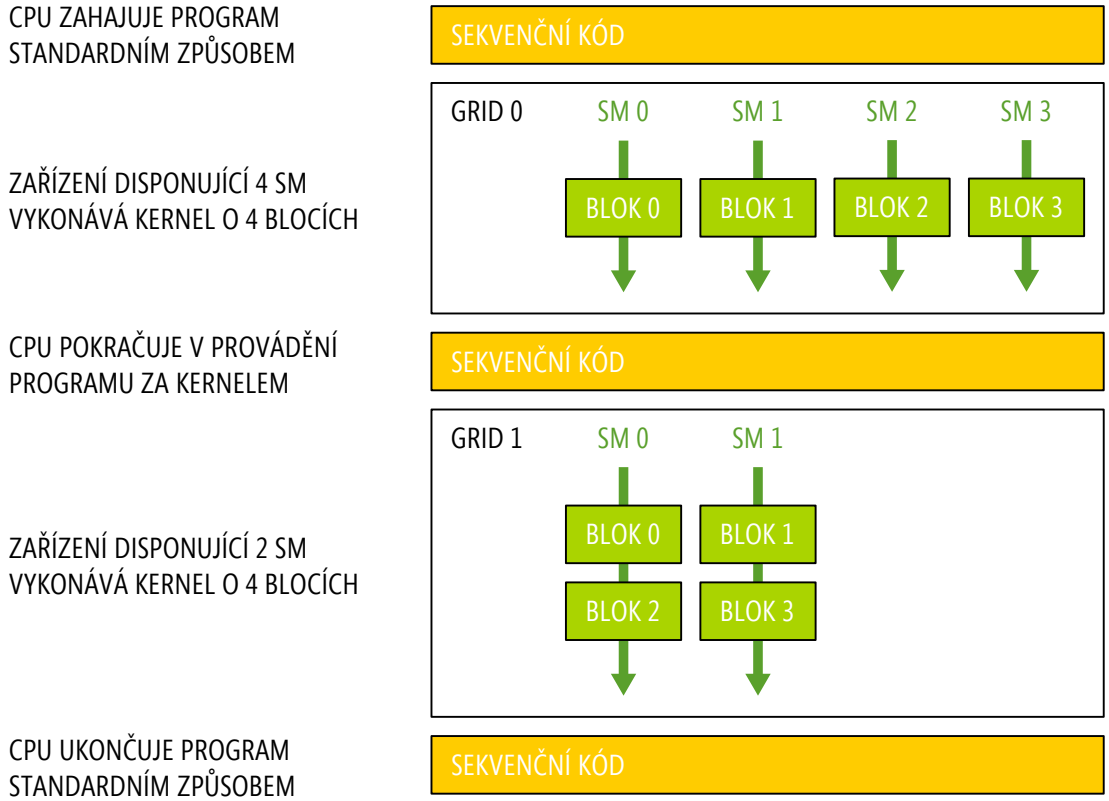
Provádění kernelu na zařízení probíhá na úrovni vláken, která jsou přiřazována na jednotlivé SP neboli CUDA jádra. Pro efektivní řízení programu a snazší identifikaci jsou vlákna sdružována do skupin, tzv. *bloků*. Bloky jsou přiřazovány na jednotlivé SM. Množina všech bloků zapojených do provádění určitého kernelu se pak nazývá *grid*. Na obou úrovních nabízí CUDA 1D, 2D nebo 3D rozložení prvků. Chceme-li např. vytvořit blok se 64 vlákny, můžeme je uspořádat jako 1D sekvenci o délce 64 prvků nebo jako 2D mřížku o rozměrech 8x8 prvků. Slouží k tomu struktura `dim3`, kterou využijeme při zmíněném volání kernelu uvnitř úhlových závorek. Jednoduchou ukázku takového volání uvádím v rámci kódu 4.1.

```
// volání kernelu Add v jednom bloku o N vlákech
Add<<<1, N>>>(A, B, C);
```

Kód 4.1: Volání jednoduchého kernelu.

CUDA dále zavádí pojem *warp*. Jedná se o skupinu vláken, která jsou fyzicky paralelně prováděna na jednom SM. Nvidia doporučuje ve svých materiálech navrhovat velikost bloků v celých násobcích velikosti warpu. Jakmile je totiž blok umístěn na SM, může být rozdělen na střídavě prováděné warpy, z nichž každý zcela využije dostupné SP.

Seskupování vláken umožňuje programům snadno se přizpůsobit hardwarovým zdrojům použitého zařízení. Bloky jsou totiž přiřazovány na SM nezávisle na sobě. Uvažujme např. kernel se čtyřmi bloky. Disponuje-li zařízení pouze dvěma SM, budou každému z nich přiděleny dva bloky. Jakmile však dostaneme k dispozici zařízení o čtyřech SM, získá každý blok vlastní multiprocesor. Tento příklad je znázorněn na obrázku 4.6.



Obrázek 4.6: Heterogenní programování. Kód je prováděn částečně na CPU a částečně na zařízení. V rámci jednoho programu lze využít různá zařízení – CUDA se automaticky přizpůsobí jejich dostupným výpočetním prostředkům.

Přestože kernel představuje samostatnou jednotku provádění na zařízení, nemusí být všechen kód vměstnán přímo do funkce označené klíčovým slovem `__global__`. CUDA totiž nabízí také definici pomocných funkcí určených k provádění na zařízení. Ty se označují klíčovým slovem `__device__` a lze je volat pouze z kernelu nebo jiné `__device__` funkce. Architektura Fermi pak dokonce nabízí jejich rekurzivní volání.

Některá novější zařízení navíc umožňují současné spuštění několika kernelů najednou s pomocí tzv. *streams*. Tím otevírají cestu k dalšímu urychlení a lepšímu využití prostředků dostupných na zařízení [1].

4.3.3 Paměťový model

GPU mají podobně jako CPU k dispozici celou škálu pamětí, u nichž můžeme pozorovat nepřímou úměru mezi velikostí a rychlostí. Jejich hierarchie je přitom úzce svázána s modelem toku řízení, který jsem uvedl v předcházející kapitole.

Jednotlivá vlákna mají vyhrazena registry a dále tzv. *lokální paměť (local memory)*. Na úrovni bloku je k dispozici *sdílená paměť (shared memory)*. Všechna vlákna pak mohou přistupovat ke *globální paměti (global memory)*, *konstantní paměti (constant memory)* a *texturovací paměti (texture memory)*. Poslední tři jmenované jsou k dispozici také hostitelskému systému v rámci příkazů pro kopírování mezi nimi a hlavní pamětí systému [17, 1].

V architektuře CUDA se vyskytují také automaticky řízené vyrovnávací paměti (*cache*) úrovně L1 (každý SM má vlastní) a L2 (všechny SM sdílejí jedinou). L1 přitom pracuje se stejnými hardwarovými prostředky jako sdílená paměť a u novějších zařízení může vývojář dokonce sám nastavit, jaký poměr chce mezi hardwarově řízenou (L1) a uživatelem řízenou (sdílená paměť) vyrovnávací pamětí zvolit.

Zajímavé je, že lokální paměť přiřazená jednotlivým vláknům je ve skutečnosti mapována do globální paměti. Výlučný přístup daného vlákna však umožňuje efektivnější využití předřazených vyrovnávacích pamětí L1 a L2. Z hlediska optimalizace je však nejvýznamnější využití sdílené paměti. Ta je skutečně umístěna na čipu přímo v rámci SM a její přístupová doba může být ve srovnání s hlavní paměti až 40x kratší [1].

Kapitola 5

Paralelizace heuristických metod

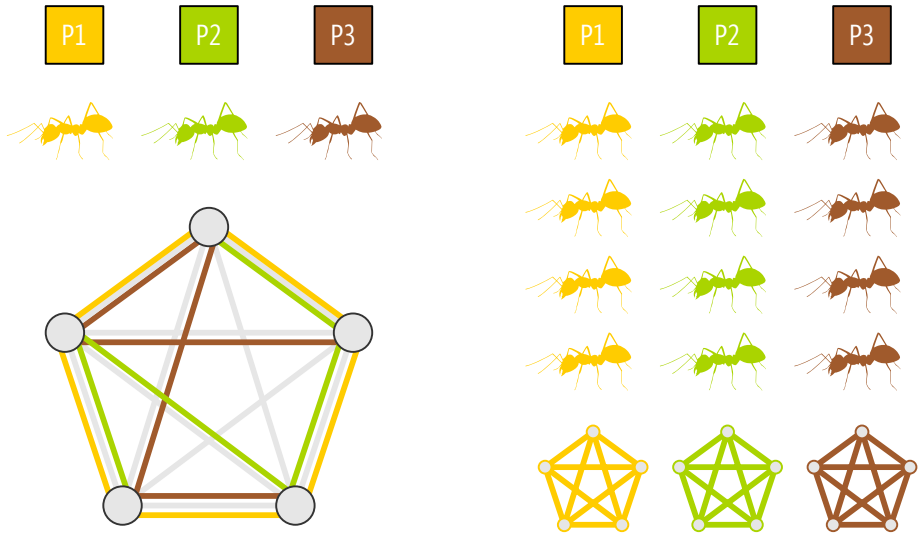
Cílem práce je akcelerace algoritmu ACO s využitím GPU. Nezbytným přípravným krokem k praktické implementaci je proto analýza možností paralelního zpracování a souvisejícího mapování procesů algoritmu ACO na hardware GPGPU zařízení. Právě tomuto úkolu se mohu věnovat nyní – poté, co jsem vysvětlil teoretickou podstatu ACO i metod lokálního prohledávání (kapitola 3) a zároveň jsem popsal hardware i software související s GPGPU (kapitola 4). Na možnosti paralelizace se nyní podívám ze dvou úhlů – populačního a datového. Tyto dva pohledy lze přitom kombinovat i v rámci jedné implementace.

5.1 Populační paralelizace

Charakter ACO umožňuje nahlížet na přístupy k paralelizaci podle toho, jaká část jedinců v určité populaci (tedy mravenců v kolonii) je přiřazována výpočetním jednotkám. Rozlišujeme *jemnozrnou paralelizaci (fine-grained parallelization)*, kdy se každý procesor věnuje výpočtu nad jedním nebo několika málo mravenci, a dále *paralelizaci hrubozrnnou (coarse-grained parallelization)*, kdy jsou jednotlivým procesorům přiřazovány celé kolonie. Oba přístupy, znázorněné na obrázku 5.1, nyní podrobněji popíši.

Jemnozrnná paralelizace představuje poměrně přirozený způsob, jak práci mezi procesory rozložit. Každá výpočetní jednotka se v rámci procedury `ConstructAntSolutions` zabývá hledáním cest pro řádově jednotky mravenců, což jsou nezávislé procesy, neuvažujeme-li variantu ACS, u níž se mění hodnoty feromonů i v průběhu zmíněné procedury. Jakmile jsou cesty nalezeny, mohou na sebe všichni mravenci v kolonii počkat na synchronizační bariéře, aby následně pokračovaly opět nezávislou lokální optimalizací jednotlivých cest. Problém nastává u procedury `UpdatePheromones`, kdy je třeba na každé cestě sečíst přírůstky feromonů ode všech mravenců, kteří po ní prošli. To povede na značnou časovou režii související s R/W konflikty různých druhů.

Hrubozrnná paralelizace pracuje namísto malých skupinek mravenců s celými koloniemi. Ty operují nad stejným konstrukčním grafem, ale jinak jsou na sobě nezávislé. Výrazně je omezena komunikační režie, protože procesory si mezi sebou nemusejí předávat kompletní informace o feromonových stopách a nalezených cestách, ale např. pouze cesty několika nejlepších mravenců nebo *best-so-far* řešení. Nutná dokonce není ani synchronizace po každé iteraci, uvažovat lze např. komunikaci po desítkách či stovkách iterací. T. Stützle dokonce uvedl extrémní variantu, u níž kolonie nekomunikují vůbec – jde prakticky o paralelní běh mnoha ACO algoritmů, mezi jejichž výsledky se nakonec vybere ten nejlepší [14].



Obrázek 5.1: Přístupy k paralelizaci na základě populačního hlediska. V případě *jemnozrnné paralelizace* (nalevo) hledá každý procesor cestu pro jednoho (případně několik málo) mravenců. Feromonové stopy a další informace tak musí procesory sdílet. Při *hrubozrnné paralelizaci* (vpravo) se každý procesor stará o celou kolonii. Sdílet pak stačí výsledky jednotlivých iterací nebo dokonce celých běhů.

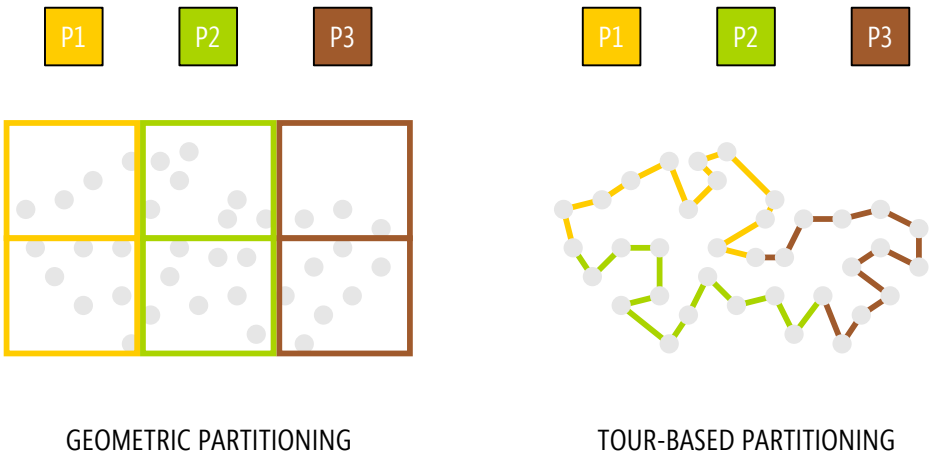
Problémem zdánlivě ideální hrubozrnné paralelizace jsou nároky na paměť. Zatímco konstrukční graf mohou kolonie sdílet, ACO vyžaduje také uchovávání dat, která jsou pro každou kolonii jedinečná – jde především o informace o feromonových stopách. V případě *many-core* architektury, u níž nejsou výjimkou systémy se stovkami výpočetních jader, by přiřazení kolonie každému z nich znamenalo enormní nároky na paměť. U běžných vícejádrových CPU by tento přístup mohl být velmi zajímavý, avšak na ně práce necílí.

5.2 Datová paralelizace

V předcházející kapitole jsem nastínil přístup k paralelizaci, který by se dal označit také jako *procesní*. Nyní popíši způsob, jak uvažovat o rozdělení práce s ohledem na zpracovávaná data. Ta jsou v případě COP řešitelných pomocí ACO (více v kapitole 2.2) tvořena konstrukčním grafem a dále hodnotami feromonových stop, které jsou však jednoznačně přiřazeny prvkům konstrukčního grafu. Nabízí se tedy možnost provést dekompozici konstrukčního grafu i související informace o feromonu na menší části a ty zpracovávat odděleně. Tento přístup je obecnější a umožňuje značnou úroveň paralelizace v oblasti ACO i metod lokálních prohledávání, aniž bychom je museli výrazněji modifikovat. Značnou nevýhodou je skutečnost, že dekompozice konstrukčního grafu a následné spojování s sebou přináší zhoršení kvality výsledného řešení. Než tento jev kvantifikuji, vysvětlím dvě základní metody, které lze při dekompozici použít – jde o tzv. *geometric partitioning* a *tour-based partitioning*. Obě metody jsou znázorněny na obrázku 5.2.

Geometric partitioning představuje v případě TSP poměrně přirozený přístup. Obdélník obsahující všechny uzly grafu rozdělíme rekurzivně na sadu menších obdélníků, které pak přidělíme jednotlivým procesorům. Ty najdou v každém z malých obdélníků cestu spojující všechny uzly, načež jsou tyto cesty pospojovány do cesty vedoucí celým grafem. V důsledku nedokonalého hledání propojek mezi jednotlivými obdélníky dochází ke zhoršení kvality řešení přibližně v rozmezí 3–5 % oproti optimálnímu řešení. Míra tohoto zhoršení klesá, pokud přidělujeme procesorům větší oblasti. V [7] se uvádí, že výrazné zhoršení lze očekávat zejména v případě oblastí obsahujících méně než 1 000 uzlů.

Tour-based partitioning se snaží odstranit nedostatky předchozího přístupu, které jsou způsobeny tím, že města jsou rozdělena do oblastí výhradně na základě svého umístění. Jak již název napovídá, vychází tato metoda z nedokonalé, ale kompletní cesty grafem (tu lze vytvořit pomocí libovolného heuristického algoritmu). Dělení grafu na části probíhá v tomto případě také rekurzivně, ale „podél cesty“. Jednotlivým procesorům jsou tedy přiděleny segmenty původní cesty, přičemž první a poslední uzel v segmentu je dočasně propojen dodatečnou hranou. Po aplikaci libovolných optimalizačních mechanismů na jednotlivé segmenty a odstranění pomocných hran jsou tyto spojeny zpět do kompletní cesty. Tour-based partitioning navíc počítá s tím, že takto vytvořená cesta se dá znovu rozdělit na segmenty, které však budou začínat a končit v jiných uzlech. Díky tomu může dosáhnout velmi zajímavých výsledků – příkladem může být instance o více než 10 milionech uzlů, kterou se podařilo vyřešit s odchylkou 4,3 % od Held-Karp bound.



Obrázek 5.2: Přístupy k paralelizaci na základě datového hlediska. V případě *geometric partitioning* rozdělíme konstrukční graf na pravidelné geometrické útvary, např. obdélníky. Ty pak předáme jednotlivým procesorům k optimalizaci. Máme-li již k dispozici nějakou cestu konstrukčním grafem, můžeme použít *tour-based partitioning*, kdy jednotlivým procesorům přiřadíme určité úseky této cesty.

V roce 1995 byl dále uveden *kombinovaný přístup*, který sice generuje podproblémy na základě geometrického klíče (tedy opět např. obdélníky), ale středy příslušných oblastí volí podél zadané výchozí cesty. Touto technikou se podařilo dosáhnout řešení, jehož odchylka od Held-Karp bound představovala pouhé 1,57 % [7].

Kapitola 6

Návrh řešení

Tato kapitola je věnována návrhu řešení, tedy způsobu implementace algoritmů ACO a k-optimization na platformě CUDA. Identifikuji v ní nejprve paralelní výpočetní jednotky dostupné na GPU čipu, abych následně provedl úvahu nad jejich vazbou na vybrané prvky zmíněných algoritmů. V hlavní části kapitoly pak přednesu kostru, na jejímž základě je vystavěna praktická část práce. Vysvětlím úpravy algoritmu ACO pro jeho efektivnější implementaci na cílové architektuře a nastíním některé problémy vyplývající z nesouladu charakteru řešené úlohy (TSP) a vlastností použité platformy (CUDA).

6.1 Dvouúrovňová povaha paralelizace na CUDA

Paralelizace spočívá v úpravě algoritmů tak, aby mohly běžet současně na několika výpočetních jednotkách. Identifikujme tedy nejprve tyto výpočetní jednotky. Na GPU čipech je nejmenším takovým prvkem *streaming processor (SP)*, který disponuje vlastní sadou registrů, což je však zároveň jediná paměť, k níž má výlučný přístup. Vzhledem ke značně omezené velikosti této sady registrů (nanejvýš v řádu jednotek kB) musíme při snaze o aplikaci standardních paralelních přístupů uvažovat také jednotky větší. V případě platformy CUDA může takovou jednotkou být *streaming multiprocessor (SM)*, který disponuje sdílenou/vyrovnávací pamětí o velikosti řádově desítek kB (více v kapitole 4.3.3). To však není jeho jedinou z tohoto pohledu zajímavou charakteristikou – SM se totiž chová jako celek také např. při přístupu do paměti globální, u níž právě na úrovni SM dochází k prokládání přístupů do paměti s výpočtem.

V případě platformy CUDA bychom mohli najít dokonce i třetí úroveň – samotné *zařízení (device)*. Karet nebo jinak umístěných grafických čipů může být v rámci výpočetního uzlu instalováno několik a CUDA nabízí mj. metody pro vzájemné sdílení paměti. Taková situace však nenastává vždy. SP i SM jsou oproti tomu dostupné na každém zařízení a při návrhu tedy musíme uvažovat paralelizaci přinejmenším na dvou úrovních.

6.2 Využití různých přístupů k paralelizaci

V kontextu uvedeném v předcházející sekci se podívejme nejprve na možnosti *populační paralelizace*, kterou jsem popsal v sekci 5.1. I přes dříve uvedené nevýhody tzv. jemnozrnného přístupu (rozlišení na úrovni agentů) musíme tento nějakým způsobem na CUDA zavést, protože jednotkou provádění bude vždy vlákno. To je přiřazeno na SP vybavený pouze registry, které v žádném případě nemohou obsáhnout objem dat potřebný k práci

celé kolonie. Z populačního hlediska tedy nemáme jinou možnost, než každému SP přiřadit práci jednoho či několika mravenců.

Na úrovni SM také nejsou paměti umístěné přímo na čipu příliš velké, ale situace je již výrazně lepší. Navíc máme možnost optimalizovat přístupy do paměti hlavní, která už má obvykle kapacitu poměrně vyhovující. Jednotlivým SM bychom tedy mohli přiřazovat celé kolonie, což odpovídá hrubozrnné paralelizaci. Ta je přitom v literatuře uváděna jako nejslibnější přístup pro paralelizaci ACO. Pokud bychom uvažovali i paralelizaci na úrovni zařízení, byla by tato metoda zřejmě také tou nejvhodnější.

Nabízí se možnost použít také *datovou paralelizaci* nastíněnou v sekci 5.2. Ta spočívá v rozdělení konstrukčního grafu na menší celky, které pak řešíme nezávisle a nalezené dílčí cesty nakonec spojíme. Ani v tomto případě nám paměťové možnosti nedovolí delegovat jednotlivé oblasti na SP. V případě SM je již situace lepší a nabízí se možnost využít sdílenou paměť pro často přistupovaná data, jako jsou např. struktury uchovávající hladiny feromonových stop nebo vzdálenosti mezi městy.

Mohlo by se zdát, že uvedenými způsoby vyřešíme pouze paralelizaci ACO. Jak jsem však uvedl již dříve, na úrovni mravenců mohou pracovat i algoritmy lokálního prohledávání. Ty totiž optimalizují právě cesty vytvořené jednotlivými mravenci, jejichž zpracování je z pohledu těchto algoritmů nezávislé.

6.3 Kostra navrhovaného postupu

Vzhledem k poznatkům uvedeným v předcházejících podkapitolách se nabízí možnost přístupy k paralelizaci kombinovat. Na úrovni SP je nevyhnutelné využití procesní paralelizace, tedy přiřazení jednoho nebo několika mravenců každému jádru. Na úrovni SM však můžeme využít jak paralelizace procesní (vícero kolonií bude řešit stejný problém), tak paralelizace datové (vícero kolonií bude řešit oddělené podproblémy). Vzhledem k paměťovým nárokům algoritmu ACO jsem zvolil druhou variantu. S využitím techniky *tour-based partitioning* (tedy rozdělení „podél cesty“, více v podkapitole 5.2) bude celý problém rozdělen na podproblémy, které bude možné řešit fyzicky paralelně na vícero SM.

Proces výpočtu pak můžeme uvažovat na dvou úrovních. Na vyšší z nich, kterou dále budu označovat jako *makro úroveň*, se budeme zabývat hledáním počátečního řešení celého problému. Rozdělením tohoto řešení na menší celky připravíme vstupní data pro nižší *mikro úroveň*, na které už budeme podproblémy řešit s větším důrazem na kvalitu.

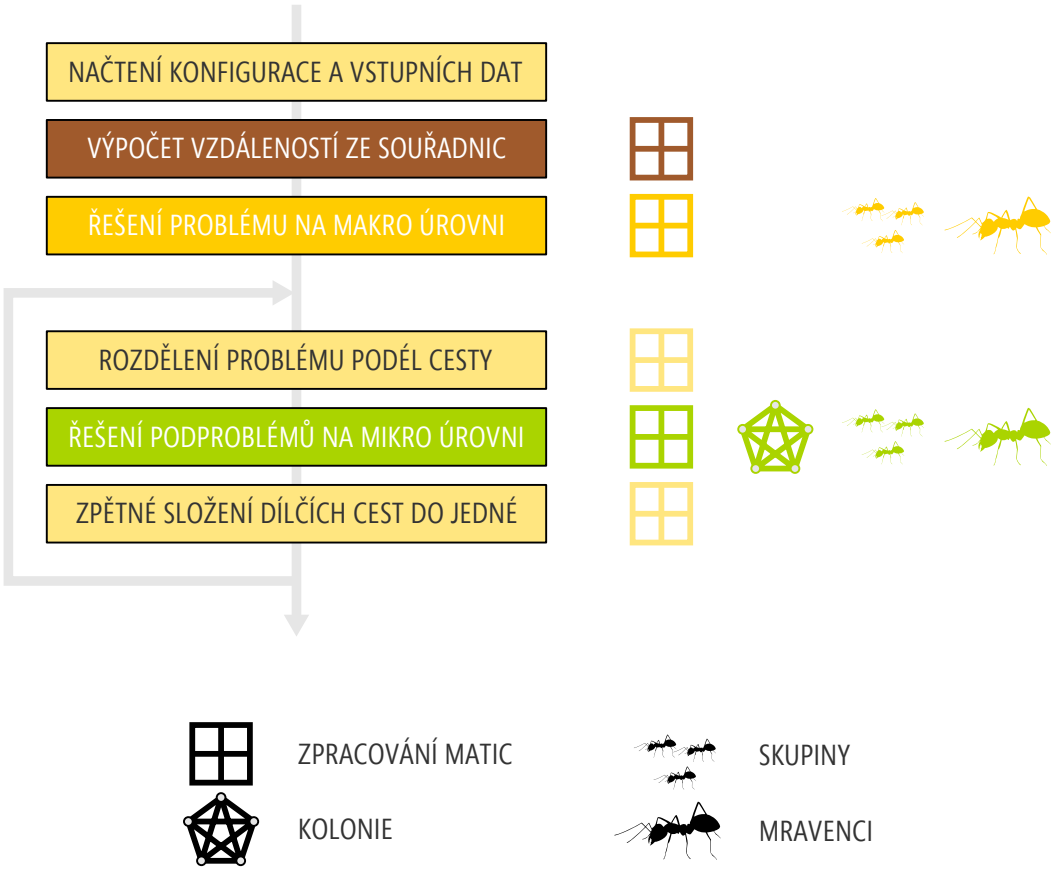
Makro úroveň představuje nezbytný přípravný krok techniky *tour-based partitioning*. Možnosti paralelizace i optimalizace jsou však v jejím případě omezené, a to zejména paměťovými nároky. U větších problémů lze totiž do hlavní paměti nebo paměti zařízení umístit jedinou kolonii – např. na zařízení Nvidia GTX 480 disponujícím 1,5 GB hlavní paměti lze umístit data nanejvýš pro asi 9 000 uzlů. Vzhledem ke značně rozptýleným přístupům do paměti k tak velkému bloku dat zde nenajdeme ani využití pro sdílenou paměť. Pro úsporu výpočetního času tedy použijeme pouze metodu ACO bez podpory lokálního prohledávání.

Mikro úroveň tvoří naopak jádro přístupu k paralelizaci. Při vhodně zvolené velikosti podproblému se nejen otevírá prostor pro využití sdílené paměti, ale také se nabízí možnost uložení dat v hlavní paměti pro velké množství kolonií najednou. Můžeme tedy bez obav použít standardní podobu ACO včetně lokálního prohledávání.

Jak jsem uvedl v kapitole 3, mají ACO i navazující techniky lokálního prohledávaní množství variant, jejichž prvky lze mezi sebou různě kombinovat. Této skutečnosti jsem využil k návrhu vlastní varianty vhodné pro zaměření práce. Základem je *ant system (AS)*,

který minimalizuje požadavky na přístup do paměti a synchronizaci. Rozmisťování feromonu provádí pouze mravenec, který v dané iteraci našel nejkratší cestu, což je prvek převzatý z *ant colony system (ACS)* pro další úsporu paměťových operací. Počet mravenců je dále výrazně vyšší, než je běžné – přibližně třikrát až pětkrát (více v kapitole 7) – přičemž tito začínají své cesty v různých městech. Díky tomu lze zapojit do výpočtu více procesorů a zároveň snížit počet iterací potřebných k dosažení kvalitního řešení.

Pro účely lokálního prohledávání jsem zvolil techniku 2-opt. Také ta má menší nároky na přístup do paměti než její alternativy uvedené v kapitole 3.2, protože v každém kroku uvažuje pouze dva segmenty. Vzhledem k tomu, že pro každý segment je potřeba zjistit, jaká města jsou na jeho okrajích, a dále načíst vzdálenost mezi nimi, znamená vynechání dotazu na třetí segment značnou úsporu. Navíc se snižuje množství přípustných rekombinací a tím i nutnost zjišťování délek alternativních cest.



Obrázek 6.1: Průběh výpočtu se znázorněním možností paralelizace. U každé operace jsou uvedeny symboly odpovídající aplikovaným úrovním paralelního zpracování. Podpůrné procesy obvykle provádějí paralelně pouze výpočty nebo manipulace nad maticemi. Hlavní části výpočtu pak využívají konceptu skupin i jednotlivých mravenců. V případě mikro úrovně lze navíc současně zpracovávat více kolonií. Na makro úrovni by k tomuto přístupu bylo zapotřebí dodatečná CUDA zařízení, která v rámci práce neuvažuji.

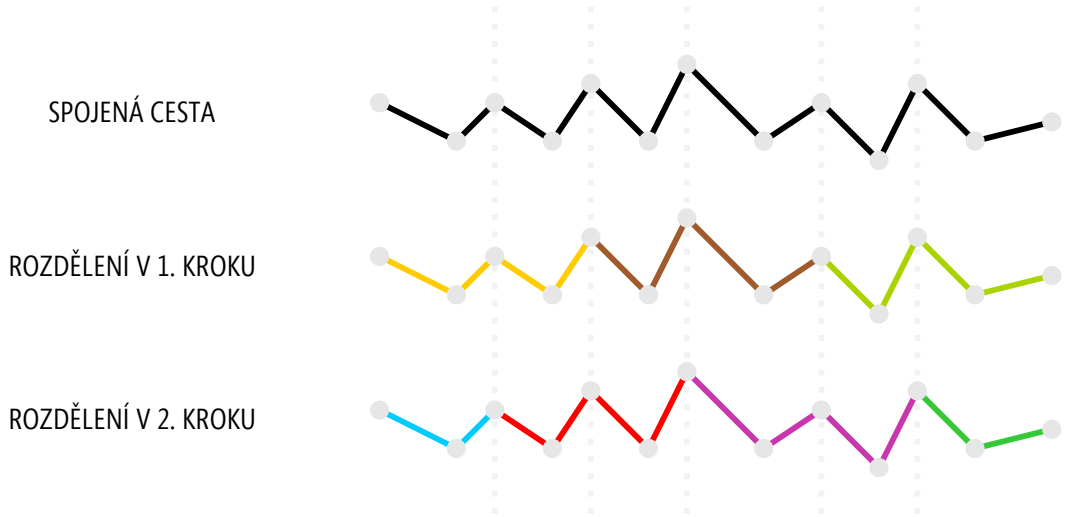
Kromě paralelizace na úrovni celých kolonií a jednotlivých mravenců jsem zavedl ještě jednu další, která pracuje se *skupinami mravenců* (v kódu jsou označovány jako *squads*).

Taková skupina se „pohybuje“ po konstrukčním grafu jednotným způsobem – většinu času se tedy chová jako jediný mravenec. V situacích, kdy je potřeba prohledat určitou množinu a zvolit nejvhodnější prvek, se však mravenci rozdělí a každý z nich prohledává pouze určitý díl. Vedoucí skupiny (v kódu označen jako *squad leader*) potom porovná kandidátní prvky z jednotlivých dílů a vybere ten, který je nejlepší pro celou skupinu. Této techniky se využívá v následujících situacích:

1. výběr města, kterým bude pokračovat cesta v daném kroku ACO
2. výběr výměny, která bude provedena v daném kroku 2-opt

V obou případech se jedná o hledání maxima. U lokálního prohledávání vybíráme dva segmenty, jejichž výměnou dosáhneme nejvýznamnějšího zkrácení cesty. V případě ACO pak vybíráme z množiny přípustných měst takové, které má nejvyšší hodnotu vypočtenou podle vzorce $\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}$. V tomto případě představuje i město, v němž se skupina nachází, a j město z množiny přípustných měst (tedy takových, která nebyla dosud navštívena).

Algoritmus ACO je původně založen na pravděpodobnostním výběru komponent (tedy měst). Tento přístup je však pro cílovou platformu poměrně nevhodný. Zavádí totiž nutnost generování pseudonáhodných čísel, které je značně pomalé, protože obvykle pracuje s celými čísly a operací modulo, což jsou akce překládané na mnoho instrukcí i na moderních zařízeních [1]. Také načítání a ukládání stavu pseudonáhodných generátorů může být při větším počtu vláken náročné. Při shodném počtu iterací navíc dává pravděpodobnostní výběr výrazně horší výsledky. Výsledná implementace jej proto ve standardní podobě nevyužívá. Lze jej však aktivovat vhodným nastavením zvláštní direktivy preprocesoru `useGreedyAggregates` (více v kapitole 7.2.1). Diverzita řešení snížená deterministickým výběrem komponent je do určité míry vyrovnána tím, že cesty se hledají z více počátečních měst, než je běžné (v případě mikro úrovně dokonce ze všech měst).



Obrázek 6.2: Rozdělení úlohy na podproblémy podél cesty v jednotlivých krocích mikro úrovně (tzv. *overlapping steps*). Hraniční body z prvního kroku jsou v druhém kroku umístěny ve středu dílčích cest. Jednotlivé podproblémy jsou odlišeny barvami.

Uvedený návrh nabízí možnosti pro paralelizaci téměř všech částí výpočtu. Zasazení jednotlivých technik do celého procesu znázorňuje obrázek 6.1. Nejdříve budou nastaveny parametry algoritmu a ze souboru budou načtena podkladová data pro řešenou úlohu. Na základě těchto dat pak bude vypočtena matice vzdáleností mezi městy[1]. Poté bude celý problém vyřešen na makro úrovni, čímž vznikne počáteční řešení.

Jakmile bude počáteční řešení k dispozici, bude rozděleno „podél cesty“ na podproblémy shodné velikosti tak, aby tato velikost nepřesáhla předem stanovenou hranici. Získané podproblémy pak bude možné optimalizovat s vysokou mírou paralelního zpracování. Na závěr budou cesty získané řešením podproblémů spojeny zpět do jediné. V tomto bodě by mohl algoritmus skončit. Ve skutečnosti lze zopakovat optimalizaci na mikro úrovni s tím, že cesta bude rozdělena v jiných místech – tak, aby původní „švy“ byly překryty. To umožní zmírnit neblahý vliv rozdělování a opětovného skládání cesty na kvalitu výsledného řešení. Mechanismus, který je v kódu označován jako *overlapping steps*, je znázorněn na obrázku 6.2. Přestože kroků je možné provést více, výsledná implementace ukázala, že výrazný přínos má pouze druhé opakování mikro úrovně, během kterého jsou původní „švy“ přesně uprostřed nově rozdělených cest.

[1] Formát knihovny TSPLIB umožňuje zadat matici vzdáleností také přímo. Implementace však podporuje pouze variantu založenou na souřadnicích ve dvourozměrné euklidovské metrice.

Kapitola 7

Implementace

Návrh uvedený v kapitole 6 jsem realizoval v podobě plnohodnotné implementace. V této části práce podrobně popíši nejen program určený pro platformu CUDA, ale také jeho věrné kopie pro běžné procesory určené pro srovnání výkonnosti. Nejprve krátce zmíním strukturu souborů se zdrojovými kódy. Následně uvedu vlastnosti společné všem vytvořeným aplikacím. Nakonec vysvětlím způsob, jakým jsem provedl akceleraci na GPU a na vícejádrových CPU s využitím knihovny OpenMP.

7.1 Základní struktura

Součástí této práce je úplná implementace navrhovaného řešení pro platformu Nvidia CUDA. Pro vyhodnocení je však třeba, aby bylo možné spustit tentýž nebo velmi podobný výpočet také na jiných architekturách za účelem srovnání. Takovou možnost však stávající nástroje pro GPGPU nenabízejí. Přiložený kód proto zahrnuje několik variant klíčových částí algoritmu, z nichž je možné sestavit celkem tři nezávislé aplikace – první pro platformu CUDA, druhou pro běžný procesor a třetí pro vícejádrový procesor. Poslední jmenovaná aplikace využívá k rozložení práce mezi jednotlivá jádra nástroje OpenMP. Jedná se tedy o paralelizaci na úrovni vláken, jejichž počet je však výrazně menší než v případě technologie CUDA – podobně jako množství výpočetních jader.

Název adresáře	**Význam**
`common`	podpůrný kód využívaný všemi aplikacemi
`cuda`	kód v jazyce CUDA C pro platformu CUDA
`cpu`	kód v jazyce C++ pro běžný procesor
`inputs`	sada instancí problému obchodního cestujícího ve formátu TSPLIB
`omp`	kód v jazyce C++ s podporou OpenMP pro vícejádrový procesor
`vsproject`	projekt pro vývojové prostředí Visual Studio

Tabulka 7.1: Adresářová struktura implementace.

Zdrojové soubory se nacházejí ve společném úložišti, aby bylo možné sdílet některé části kódu a nastavení, což je díky značné kompatibilitě jazyků CUDA C a C++ možné. Podpůrný kód využívaný všemi aplikacemi je umístěn v podadresáři `common`. Soubory specifické pro jednotlivé aplikace jsou pak umístěny v adresářích s odpovídajícím názvem. Adresář

`inputs` obsahuje sadu úloh, které byly použity při testování. Celou adresářovou strukturu shrnuje tabulka 7.1.

Kód lze přeložit a spustit na široké škále operačních systémů. Testován byl na systémech Windows (XP, Vista a 7), Linux (Ubuntu, CentOS), FreeBSD (9.0) a Solaris (5.11)[1]. V prvních dvou rodinách byla ověřena také kompatibilita na 32bit i 64bit verzích. Překlad na systémech Windows může usnadnit projekt pro vývojové prostředí Visual Studio 2010 umístěný v adresáři `vsproject`. Pro systémy vycházející z UNIXu je přiložen `Makefile`.

Požadavky na hardware nelze v případě aplikací `cpu` a `omp` přesněji specifikovat – podle použitého procesoru se bude měnit rychlost výpočtu, od velikosti dostupné paměti se pak bude odvíjet limit počtu uzlů, které může řešený problém obsahovat. Aplikace `cuda` oproti tomu klade specifické nároky na vlastnost *compute capability* použitého zařízení – ta musí mít hodnotu alespoň 2.0, což je splněno od architektury Fermi výše. Příčinou je použití některých atomických instrukcí, které nejsou na dřívějších architekturách dostupné. Dále je třeba, aby sdílená paměť měla dostatečnou velikost – při výchozím nastavení totiž využívá kernel `microFindTour` (více v kapitole 7.3.3) přibližně 46 kB z celkových 48 kB, které architektura Fermi nabízí. Zařízení s *compute capability* menší než 2.0 nabízí pouhých 16 kB sdílené paměti, což by omezovalo maximální velikost podproblému na mikro úrovni na nepoužitelnou hodnotu.

7.2 Společné prvky

Základní podobu řídícího toku programů jsem nastínil již v kapitole 6.3. Nyní popíši vybrané implementační detaily, které jsou společné všem aplikacím. Technikám, které byly použity jen na některých platformách, budu věnovat zvláštní podkapitoly.

7.2.1 Konfigurace

Prvním krokem je zavedení konfigurace. Aplikace pracují s nemalým množstvím parametrů, které lze nastavit podle potřeby. Většina z nich je součástí hlavičkových souborů a měl by je měnit pouze zkušený programátor obeznámený s kódem i cílovou architekturou. Jejich definice je sdílená a nachází se v souboru `common/common-setup.h`. Obecně platí, že hodnoty by nemělo být třeba upravovat, protože byly vhodně zvoleny s ohledem na maximální využití architektury Fermi. Seznam dostupných parametrů uvádí tabulka 7.2.

Parametr `nCitiesInCluster` omezuje rozměr jednotlivých podproblémů, na které je úloha rozdělena před zahájením mikro úrovně. Hodnoty s názvem ve tvaru `*Squad*` souvisejí s konceptem skupin mravenců popsaným v kapitole 6.3. Nastavení `useGreedyAggregates` umožňuje potlačit pravděpodobnostní výběr komponent u aplikace pro platformu CUDA (u zbývajících aplikací je deterministický výběr komponent jediným implementovaným postupem). Parametr `rewindToursImmediately` se týká nutnosti „přetočení“ cest na mikro úrovni, jehož cílem je upravit všechny nalezené cesty tak, aby začínaly ve stejném městě a mohly tak být snadno začleněny zpět do celkového řešení (více v kapitole 7.2.4). Pokud je parameter nastaven, proběhne uvedená modifikace vždy okamžitě po sestavení cesty a algoritmus lokálního prohledávání tak může postupovat obvyklým způsobem s tím, že první a poslední segment cesty musí zůstat zachován. Je-li parametr vynechán, aktivují se zvláštní podmínky v rámci funkce `make2Opt`, které zajistí, aby byly zachovány segmenty, které by po přetočení byly na místě prvního a posledního. Ke skutečnému přetočení pak dojde pouze

[1] Systémy FreeBSD a Solaris nemají podporu pro CUDA, testovány byly pouze aplikace pro procesory.

Název	Hodnota	Význam
`nCitiesInCluster`	96	maximální velikost podproblému
`nMacroAcoCyclesDefault`	3	počet iterací ACO na makro úrovni
`nMicroAcoCyclesDefault`	15	počet iterací ACO na mikro úrovni
`nLocalSearchCyclesDefault`	96	počet průchodů lokálního prohledávání
`nOverlappingStepsDefault`	2	počet opakování na mikro úrovni
`nMacroSquadsMax`	64	počet skupin na makro úrovni
`nMacroAntsPerSquad`	8	velikost skupiny na makro úrovni
`nMicroAntsPerSquad`	4	velikost skupiny na mikro úrovni
`useGreedyAggregates`	1	deterministický výběr komponent
`rewindToursImmediately`	1	„přetáčení“ cest ihned po ACO

Tabulka 7.2: Přehled parametrů dostupných v podobě direktiv preprocesoru.

u vítězné cesty těsně před jejím začleněním do celkového řešení. Varianta s okamžitým přetáčením (tedy výchozí nastavení) je výrazně rychlejší.

Zbývající čtyři parametry s názvem ve tvaru `*Default` představují pouze výchozí nastavení parametrů, které jsou zamýšleny i pro uživatele aplikace. Upravit toto výchozí nastavení a zadat některé další hodnoty může uživatel jak pomocí parametrů příkazové řádky, tak použitím konfiguračního souboru. Parametry dostupné tímto způsobem jsou uvedeny v tabulce 7.3. První z nich, tedy `source`, udává název souboru, z něhož se mají načíst vstupní data úlohy (více v podkapitole 7.2.2). Podobně parametr `destination` určuje název souboru, do nějž se má zapsat výsledek výpočtu. Nastavením parametru `extendedOutput` na hodnotu `1` pak dojde k aktivaci rozšířeného módu výstupu (více v kapitole 7.2.6).

Parametry `nMacroAcoCycles`, `nMicroAcoCycles` a `nLocalSearchCycles` určují počet iterací příslušných algoritmů. Parametr `nOverlappingSteps` se týká počtu opakování celého výpočtu na mikro úrovni s alternativními hraničními body (více v kapitole 6.3). Další dva parametry, tedy `alpha` a `beta`, odpovídají přímo stejnojmenným parametrům algoritmu ACO (více v kapitole 3.1.3). Parameter `device` pak umožňuje vybrat CUDA zařízení, na němž má výpočet probíhat. Čísla zařízení na daném systému lze zjistit např. pomocí nástroje `deviceQuery` dostupného v rámci balíku Nvidia CUDA SDK.

Všechny parametry uvedené v tabulce 7.3 lze nastavit také pomocí konfiguračního souboru. Jedná se o běžný textový soubor s řádky ve formátu `PARAMETER = VALUE`, kde `PARAMETER` představuje *Označení* parametru z tabulky 7.3 a `VALUE` je vhodné nastavení. Konfigurační soubor lze zavést zadáním přepínače `--cfg FILE`, kde `FILE` je cesta ke konfiguračnímu souboru. Obě metody nastavení lze bez problémů kombinovat. Pro zpracování konfiguračních souborů jsem využil třídu `ConfigFile` vytvořenou Richardem Wagnerem, kterou jsem převzal se všemi náležitostmi a umístil do souborů `common/common-tsplib.cpp` a `common/common-tsplib.h`.

7.2.2 Vstupní data

Implementace algoritmů umožňuje pracovat nad libovolnou instancí TSP, protože využívá úplné matice vzdáleností mezi městy. Pro praktické využití je však vzhledem k velikosti takové matice vhodnější, když je možné ze souboru načíst podkladová data výrazně menší a matici dopočítat až za běhu programu.

Označení	Přepínač	Výchozí hodnota
source	--src FILE	není
destination	--dst FILE	není
nMacroAcoCycles	--mac INT	3
nMicroAcoCycles	--mic INT	15
nLocalSearchCycles	--lsc INT	96
nOverlappingSteps	--os INT	2
alpha	--alpha FLOAT	1.0
beta	--beta FLOAT	2.0
device	--dev INT	0
extendedOutput	--ext INT	0

Tabulka 7.3: Přehled parametrů dostupných v rámci příkazové řádky.

Aplikace umožňují podkladová data načítat z textových souborů ve formátu TSPLIB [21]. Tento formát nabízí několik způsobů zadání úlohy. Obvykle se jedná o seznam souřadnic jednotlivých měst doprovázený definicí metody, která má být použita pro výpočet vzdáleností mezi nimi. Implementace dokáže pracovat pouze se dvěma typy: `EUC_2D`, u nějž se vzdálenost mezi dvěma městy vypočítá podle dvojrozměrné eukleidovské metriky, a `CEIL_2D`, kdy je potřebná hodnota vypočtena stejně, pouze je následně zaokrouhlena na nejbližší větší celé číslo. To je zcela postačující, protože takto je zadána naprostá většina úloh na obou stránkách, z nichž jsem ukázková data čerpal [10, 21].

V případě, že má problém být řešen na platformě CUDA, jsou na zařízení přeneseny pouze souřadnice a tvorba matice probíhá už přímo na zařízení. Tím dochází ke značné časové úspoře zejména u větších instancí, kdy může kompletní matice vzdáleností vyžadovat stovky MB paměti.

7.2.3 Rámce

Matice vzdáleností vytvořená na základě vstupních dat (souřadnic) je jako zadání úlohy zcela postačující. Každý řešený problém (tedy úloha jako celek na makro úrovni i jednotlivé podúlohy na mikro úrovni) však vyžaduje také nemalé množství podpůrných datových struktur. Kromě sady konfiguračních parametrů popsaných v kapitole se jedná zejména o matici feromonových stop a pole pro ukládání nalezených řešení (tedy cest). Pro snazší manipulaci jsou všechny tyto položky sjednoceny do tzv. *rámce*, který je ve zdrojovém kódu reprezentován strukturou `frame_t`. Tato struktura má shodnou podobu na makro i mikro úrovni, což vede na nevyužití některých položek v jednom či druhém případě. Vzhledem k tomu, že naprostá většina potřebné paměti je alokována dynamicky, však nedochází k výraznému plýtvání prostředky (rámec obsahuje pouze ukazatele).

Kromě nezbytných položek zmíněných v předcházejícím odstavci jsem pro urychlení výpočtu umístil do rámce ještě důležitou pomocnou matici, která je v kódu označována jako `aggregates`. Vzhledem k tomu, že u použité varianty ACO (*ant system*) se feromonové stopy nemění v průběhu hledání cest (tedy fáze `constructSolutions`, více v kapitole 3.1.3), lze ohodnocení jednotlivých segmentů na základě vzorce uvedeného v kapitole 6.3 předpočítat na začátku iterace. Složené hodnoty se potom uloží právě do matice `aggregates` a komponenty řešení se ve fázi `constructSolutions` vybírají na základě jejích hodnot.

7.2.4 Vazba mezi úrovněmi řešení

Aplikace připraví nejprve rámec pro algoritmus ACO na makro úrovni (v kódu obvykle označen jako `macroFrame`). Následně zahájí řešení úlohy. Při tvorbě funkce `macroFindTour` jsem se snažil co nejlépe napodobit pseudokód uvedený v kapitole 3.1.3. Najdeme zde tedy volání funkcí zajišťujících nalezení kandidátních řešení (`macroConstructSolutions`), jejich vyhodnocení (`macroPerformDaemonActions`) a odpovídající aktualizaci feromonových stop (`macroUpdatePheromone`). Kromě těchto základních rutin zde najdeme především mechanismy pro výběr nejlepšího řešení v rámci iterace (*iteration best*) i celého algoritmu (*best-so-far*) a zápis odpovídající cesty do rámce pro pozdější použití.

Jakmile nalezne ACO na makro úrovni počáteční řešení, může být zahájena příprava rámců pro mikro úroveň. Nejdříve se vypočte, kolik jich je třeba k pokrytí celé cesty, aniž by velikost jakéhokoliv rámce překročila hranici danou direktivou `nCitiesInCluster` (původ výchozí hodnoty `96` je podrobněji vysvětlen v kapitole 7.3). Následně se do každého rámce zkopíruje standardní konfigurace, zapíše se umístění rámce vzhledem k celé cestě (atribut `offset`) a velikost podproblému (atribut `n` – poslední rámec může obsahovat méně měst). Kromě toho jsou do každého rámce na mikro úrovni umístěny také kopie vzdáleností mezi městy, které se v rámci nacházejí. To umožňuje pracovat s jednotlivými rámci zcela nezávisle a zlepšit paměťovou lokalitu.

V tomto bodě je třeba zmínit, že při přechodu mezi makro a mikro úrovní dochází k jakémusi překladu indexů měst. Na makro úrovni jsou městům postupně přiděleny indexy z rozsahu 0 až $n - 1$, kde n představuje rozměr celé úlohy. Na mikro úrovni se ovšem nepracuje s indexy převzatými „shora“, ale každý rámec má vlastní indexování začínající opět od 0. To umožňuje pracovat s menšími čísly a tedy i menšími datovými typy, čehož se využívá pro efektivnější přístup do sdílené paměti na platformě CUDA (více v kapitole 7.3). Překlad směrem „dolů“ (z makro úrovně na mikro úroveň) probíhá právě při kopírování vzdáleností – ty jsou do matice mikro rámce zapisovány tak, aby odpovídaly sekvenčnímu číslování měst v podúloze.

Připravené mikro rámce se zpracují nezávisle na sobě – podle možností konkrétní architektury tedy i paralelně a v libovolném pořadí. Také na mikro úrovni lze snadno najít korespondenci mezi zdrojovým kódem a teoretickým popisem algoritmu. Hlavní rozdíl oproti makro úrovni spočívá v tom, že v rámci funkce `macroPerformDaemonActions` se spouští lokální prohledávání, tedy algoritmus 2-opt, a to dříve, než dojde k měření délky cesty. Mikro úroveň však klade na řešení jedno specifické omezení – aby bylo možné krátké cesty efektivně spojit zpět do jedné dlouhé, je třeba, aby první a předposlední město nezměnilo pozici. Prakticky se tento požadavek řeší zvláštními podmínkami v algoritmech ACO i k-opt. Aby mohla být zachována diverzita dosažená zahájením cest v různých městech, jsou cesty i na mikro úrovni sestavovány z různých počátečních měst, avšak po dokončení fáze `constructSolutions` jsou „přetočeny“ pomocí funkce `rewindTour` tak, aby začínaly ve „správném“ městě, tedy městě s indexem 0. Alternativní postup, kdy se přetáčí pouze vítězná cesta a podmínky v algoritmech jsou o něco složitější, lze aktivovat potlačením parametru `rewindToursImmediately` (více v kapitole 7.2.1).

Krátké cesty vylepšené na mikro úrovni se následně začlení zpět do celkového řešení na odpovídající místo dané atributem `offset`. Při tom dochází ke zpětnému překladu indexů měst – index na mikro úrovni slouží jako ukazatel do pole reprezentujícího celou cestu před jejím rozdělením v odpovídající části dané opět atributem `offset`. Celá operace je realizována v rámci funkce `transcribeTour`, která výsledek zapisuje do zvláštní struktury `assembledTour`. Pokud má být výpočet na mikro úrovni opakován s alterna-

tivním nastavením hraničních bodů (více v kapitole 6.3), je třeba dosavadní řešení zkopírovat zpět do makro rámce, kde bude rozděleno jiným způsobem. Zajišťuje to funkce `reUseAssembledTour`.

7.2.5 Reprezentace matic

Řešení problému obchodního cestujícího se zapojením ACO využívá rozsáhlé dvourozměrné struktury – jedná se např. o matici vzdáleností nebo matici feromonových stop. Kromě těchto nezbytných položek jsem v implementaci použil dvourozměrnou organizaci i jinde, např. pro uložení kandidátních řešení – každé z nich je samo o sobě polem (odpovídá zvoleným komponentám, tedy městům, v jednotlivých krocích), druhý rozměr pak reprezentuje mravence (případně skupiny mravenců), kterým jednotlivá kandidátní řešení náleží.

Vzhledem k omezeným možnostem kompilátorů nelze na platformě CUDA pracovat s dvourozměrnými poli běžným způsobem. Připravil jsem proto speciální strukturu (v kódu pod názvem `pitchedFloatArray_t`), která sestává z běžného jednorozměrného pole čísel typu `float` (resp. ukazatele na jeho začátek) a doprovodné informace o šířce řádku. Podobné struktury existují také pro datové typy `bool` a `int`. Pomocí maker `pitchedArrayRow` a `pitchedArrayElement` pak lze k prvkům všech takovýchto polí pohodlně přistupovat jako k prvkům polí dvourozměrných.

Platforma CUDA nabízí speciální struktury a funkce s podobným určením. Vlastní implementaci jsem použil proto, aby bylo možné sdílet co nejvíce kódu i s aplikacemi pro běžné procesory, jejichž kompilátory by nedokázaly uvedené speciální prostředky využít. Délka řádku je vždy zarovnána na celočíselný násobek 512 b, což je hodnota doporučená pro architekturu Fermi [1].

7.2.6 Výstupy

Hlavní prostředkem pro prezentaci výsledků je u všech třech aplikací standardní výstup. Množství vypisovaných informací lze upravit pomocí direktiv preprocesoru ve tvaru `print*`. Ty jsou pro CUDA aplikaci umístěny v souboru `cuda/cuda-setup.h`, pro zbývající aplikace pak analogicky. Význam jednotlivých direktiv je obvykle zřejmý již z identifikátoru, podrobnosti lze případně najít v dokumentaci.

Všechny aplikace v každém případě vypíšou tři klíčové informace – dobu výpočtu na makro úrovni, dobu výpočtu na mikro úrovni a délku výsledné cesty. Řádky obsahující tyto údaje jsou pro snazší strojové zpracování výstupů předsazeny symbolem vykřičníku (`!`). Výpis všech ostatních informací je pak vždy zahájen pravou úhlovou závorkou (`>`).

Aplikace určená pro platformu CUDA nabízí navíc zápis výsledků do souboru. Cestu lze nastavit pomocí parametru `destination` (více v kapitole 7.2.1). Standardně je do zvoleného souboru zapsána výsledná cesta ve formátu TSPLIB (jedná se o typ `TOUR`).

V případě požadavku na rozšířený výstup (parametr `extendedOutput`) je do souboru zapsán nejen výsledek, ale také průběh výpočtu. Vzhledem ke složitější struktuře dat je u rozšířeného módu použit formát XML. Kořenový prvek s názvem `solution` obsahuje dvě hlavní větve – větev `history`, která zachycuje průběh výpočtu, a větev `result` zahrnující pouze výsledek. Samotná data jsou pak reprezentována větvemi `tour` složenými z prvků `stop`. Každý prvek `stop` přitom představuje jednu zastávku obchodního cestujícího na dané cestě – atribut `rank` udává pořadí zastávky, zatímco hodnota prvku obsahuje index navštíveného města.

Větev `result` obsahuje pouze výslednou cestu. Větev `history` oproti tomu obsahuje množství cest, které jsou pomocí atributů `type` a `cycle` přiřazeny úrovni a číslu iterace,

v jejímž průběhu byly vytvořeny. Typ `macro` odpovídá makro úrovni, `micro` odpovídá mikro úrovni a hodnota `joint` je použita u cest vzniklých opětovným složením podúloh. Na mikro úrovni je navíc použit atribut `cluster`, který přiřazuje řešení pořadovému číslu podproblému na mikro úrovni.

7.3 Akcelerace pomocí CUDA

V této části kapitoly ukáži, jaké techniky jsem použil pro urychlení výpočtu v případě aplikace určené pro platformu CUDA. Vysvětlím způsob mapování různých částí algoritmů na výpočetní jednotky zařízení a uvedu optimalizace použité při práci s pamětí.

7.3.1 Zpracování vstupních dat

Zpracování vstupních dat je první částí programu, která využívá zařízení ke skutečnému výpočtu – tvorbě matice vzdáleností na základě zadaných souřadnic. Podobně jako v dalších podobných případech je spuštěn jediný blok s nejvyšším možným počtem vláken. V případě, že rozměr problému dostupného počtu vláken ani nedosahuje, je počet vláken redukován právě na hodnotu `n`. Proměnná reprezentující takto stanovený počet vláken je v kódu obvykle označena `nThreads`. V rámci kernelu `computeDistances` pak jednotlivá vlákna počítají prvky matice vzdáleností v odpovídajících sloupcích. Výraznější optimalizací této části jsem se nezabýval, protože čas jejího provádění je i v případě velkých instancí zanedbatelný ve srovnání s ACO a lokálním prohledáváním.

7.3.2 Výpočet na makro úrovni

Tvorbu počátečního řešení pomocí algoritmu ACO zajišťuje kernel `macroFindTour`. V tomto případě se opět spouští jediný blok vzhledem k tomu, že v rámci jednotlivých iterací je třeba synchronizace (výběr nejlepší cesty, aktualizace feromonových stop) a synchronizace mezi bloky je problematická. Bloky totiž mohou být k provádění vystaveny v libovolném pořadí a jediným komunikačním prostředkem mezi nimi je hlavní paměť – přístup zahrnující několik spolupracujících bloků by byl značně neefektivní. Bylo by možné spustit více bloků pracujících nad celými nezávislými koloniemi, avšak tato varianta naráží na paměťové nároky ACO, jak jsem vysvětlil v kapitole 6.3.

Vlákna jsou v kernelu `macroFindTour` uspořádána dvourozměrně – první dimenzi tvoří číslo skupiny mravenců, druhou dimenzi pak pořadové číslo mravence v rámci skupiny. Konkrétní hodnoty byly zvoleny na základě experimentů: ve výchozím nastavení se využívá 64 skupin (parametr `nMacroSquads`) po 8 mravencích (parametr `nMacroAntsPerSquad`). Jedná se o parametry, které je sice možné změnit (jsou umístěny ve zvláštním hlavičkovém souboru jako direktivy preprocesoru, jak jsem uvedl v podkapitole 7.1), ale je třeba s nimi zacházet obezřetně, aby nebyl překročen limit počtu vláken v bloku.

Akcelerace samotného algoritmu ACO je na makro úrovni obtížná. Vzhledem k velikosti datových struktur a špatné lokalitě přístupů do paměti jsem nenašel významnější využití pro sdílenou paměť (nelze odhadnout, jaká data by mohla být potřeba v dalších krocích). Kernel je proto nastaven tak, aby byla většina paměti blízké výpočetním prvkům použita jako L1 cache[2] s tím, že hardware dokáže střídání bloků ve vyrovnávací paměti spravovat lépe. V samotné sdílené paměti (redukované na 16 kB) jsou tak umístěny pouze proměnné využívané pro předávaní dat mezi vlákny.

[2] Zařízení třídy Fermi dokážou této hardwarem řízené vyrovnávací paměti přidělit až 48 kB.

Paralelní zpracování se mi podařilo uplatnit i přesto, že program řeší jedinou instanci v rámci jediného bloku. V první řadě se vlákna podílejí na operacích probíhajících nad celými maticemi – jde o inicializaci feromonových stop, proces jejich vypařování a zejména pak výpočet hodnot, které budou použity při výběru komponent řešení v jednotlivých krocích (v kódu je pole těchto hodnot označeno jako `aggregates`). Všechny tyto operace obaluje makro `processJointly2D`, které skrývá mapování vláken na prvky matice. Přístup je podobný jako v případě výpočtu matice vzdáleností.

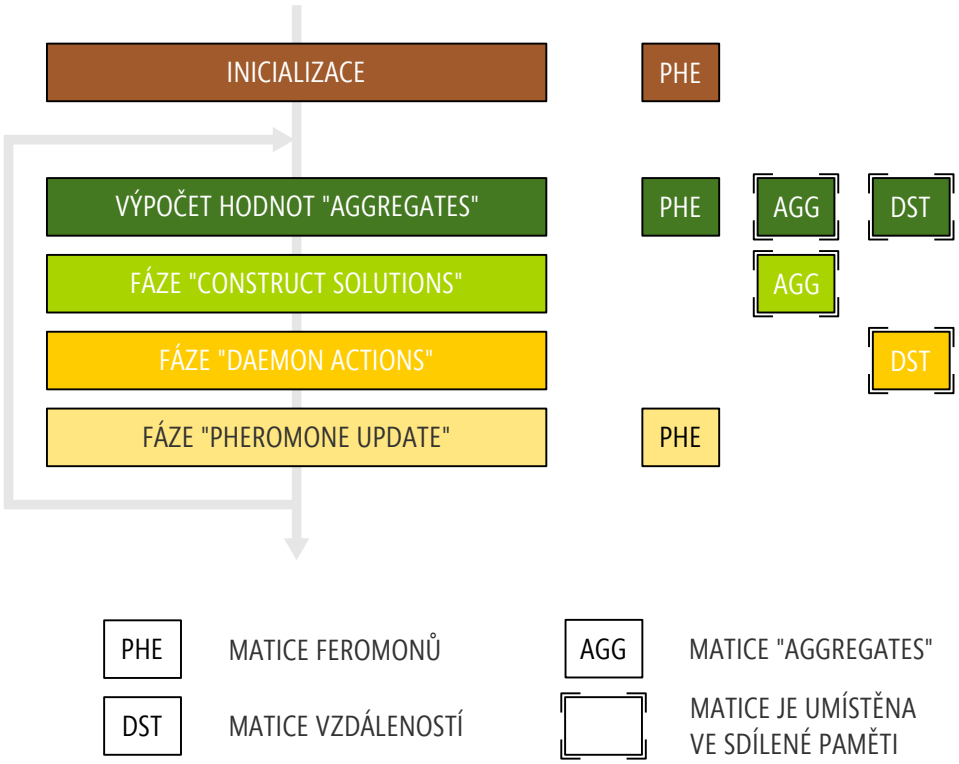
Další využití pro větší počet vláken jsem našel při procesu hledání kandidátních řešení (funkce `macroConstructSolutions`). Jak jsem nastínil již v kapitole 6.3, každá skupina mravenců začíná hledat řešení z jiného města, čímž se dosahuje větší diverzity kandidátních řešení. Vzhledem k tomu, že na makro úrovni je počet měst téměř vždy vyšší než počet skupin, jsou výchozí města volena rovnoměrně v rámci celého problému (u většiny sad vstupních dat tak dochází i ke geometrickému rozptýlení počátků). Jednotliví mravenci ve skupině se uplatní pouze při výběru další komponenty řešení (funkce `macroChooseNextCity`), kdy každý z nich prohledává část přípustné množiny měst. Vedoucí skupiny, k jehož rozpoznání v kódu slouží makro `squadLeader`, pak vyhodnotí nabízené výsledky, vybere ten nejlepší a přidá jej do výsledného řešení.

7.3.3 Výpočet na mikro úrovni

Řešení podproblémů získaných rozdělením počáteční cesty je pro platformu CUDA příznivnější zejména díky rozměru jednotlivých úloh omezených parametrem `nCitiesInCluster`. Také tento parametr je určen pouze programátorům a nastavuje se proto pouze pomocí stejnojmenné direktivy preprocesoru. Výchozí hodnotu `96` jsem zvolil s ohledem na možnosti architektury Fermi – je to největší možný rozměr problému, pro nějž lze do sdílené paměti umístit alespoň jednu matici o rozměru $n \times n$, přičemž tato má dále parametry vhodné pro zarovnaný přístup (v případě architektury Fermi by měla mít rozměry v násobcích 32).

Vzhledem k tomu, že matice uvedených vlastností se do sdílené paměti vejde pouze jedna, využívá se v průběhu algoritmu pro různé účely a je proto označena obecně jako `sharedArray`. Rámec na mikro úrovni tedy využívá matice vzdáleností, feromonových stop i předpočítaných hodnot pro výběr další komponenty (první dvě jsou navíc trvale uloženy i v hlavní paměti), ale podle potřeby tyto matice dočasně kopíruje do prostoru `sharedArray` a nad ním provádí výpočet. Přehled matic využívaných v různých fázích výpočtu na mikro úrovni podává obrázek 7.1 – dvojitě jsou orámovány ty, které se v dané chvíli nacházejí ve sdílené paměti. Jsou-li takto orámovány v jednom kroku dvě matice, znamená to, že na základě dat jedné z nich se vytvářejí data druhé, přičemž čtení i zápis probíhá nad sdílenou pamětí. Kromě matice `sharedArray` jsou do sdílené paměti i na mikro úrovni umístěny proměnné pro předávání hodnot v rámci bloku. Díky omezenému rozsahu indexů měst, který umožňuje reprezentaci na pouhých 8 bitech, se do sdílené paměti navíc vejdou také kandidátní řešení nalezená jednotlivými skupinami mravenců (`squadTours`).

Počet vláken na mikro úrovni vždy přesně odpovídá rozměru problému. Operace nad celými maticemi sice stále probíhají podobně jako na makro úrovni, avšak při hledání kandidátního řešení je již možné použít každé město jako počáteční. Zcela novým prvkem je potom lokální prohledávání v podobě algoritmu 2-opt. Každá skupina mravenců se zabývá optimalizací svého kandidátního řešení, přičemž jednotliví mravenci se podílejí na hledání nejlepší výměny v daném kroku (každý prohledává určitou část možných segmentů). Během celého procesu jsou dále aktualizovány tzv. *don't look bits*. Ty uchovávají informaci o segmentech, pro které se již nepodařilo najít další vylepšení, a brání tak jejich zbytečnému



Obrázek 7.1: Matice využívané při řešení podproblémů na mikro úrovni a jejich zapojení v různých částech výpočtu. Dvojité orámování značí umístění příslušné struktury do víceúčelového prostoru ve sdílené paměti. Jsou-li takto označeny dvě matice ve stejném kroku, představuje jedna z nich zdroj a druhá cíl příslušné operace.

opětovnému vyhodnocování. Pokoušel jsem se rozdělit práci mezi jednotlivé mravence také při hledání další komponenty (podobně jako na makro úrovni), ale zjistil jsem, že při práci nad sdílenou pamětí způsobuje tato technika zpomalení výpočtu.

Přestože paralelní zpracování se mi podařilo zavést i v rámci řešení jedné podúlohy (na úrovni vláken), hlavním přínosem mikro úrovně je možnost paralelního řešení většího množství podúloh, které probíhá na úrovni bloků – každý mikro rámec je zastoupen jedním blokem. Takový blok je opět organizován jako dvourozměrný, přičemž první dimenze odpovídá `n` skupinám mravenců (jedna pro každé počáteční město) a druhá dimenze jednotlivým mravencům ve skupině, jejichž počet závisí na parametru `nMicroAntsPerSquad` s výchozím nastavením na hodnotu 4. Obvykle má tedy blok 384 vláken. Díky značné redukci objemu dat vyžadovaného jednotlivými mikro rámci (obvykle pouze přibližně 50 kB v hlavní paměti) je i v případě velkých instancí možné připravit všechny mikro rámce v hlavní paměti zároveň a následně spustit kernel `microFindTour` s plným počtem bloků – tedy jedním pro každý mikro rámec.

7.4 Akcelerace pomocí OpenMP

Aplikace pro vícejádrové procesory využívající paralelizace s podporou OpenMP je pouze doprovodným nástrojem pro lepší vyhodnocení rychlosti hlavní části implementace. Díky značnému rozpracování návrhu paralelního zpracování a souběžného kódu pro platformu CUDA však bylo možné využít technologii OpenMP poměrně efektivně. Sekce vybrané pro paralelizaci jsou označeny direktivami `#pragma omp`. Žádná z nich však neupravuje použitý počet vláken – během celého programu se tedy používá implicitní nastavení cílového systému. To obvykle odpovídá počtu dostupných výpočetních jader, avšak lze jej změnit také např. pomocí proměnné prostředí `OMP_NUM_THREADS`.

Všechny případy použití OpenMP spočívají v paralelizaci smyček. Na makro úrovni byla takto nastavena smyčka pro výpočet hodnot využívaných při konstrukci cest (`aggregates`) a dále smyčka zahrnující hledání řešení z různých počátečních měst. Druhá jmenovaná, která napodobuje použití většího množství skupin mravenců, využívá navíc direktivy `#pragma omp critical` pro aktualizaci informace o nejlepší nalezené cestě. Na mikro úrovni byla paralelizována smyčka pro řešení jednotlivých podproblémů, která zase odpovídá spouštění většího množství bloků v CUDA implementaci. Další části kódu již paralelizovány být příliš nemohly, protože OpenMP zatím nemá dostatečnou podporu pro vnořené paralelní sekce – zápis kódu je sice možný po syntaktické stránce, ale kompilátor obvykle použije ve vnořeném bloku jen jediné vlákno.

Kapitola 8

Měření a vyhodnocení výsledků

Cílem diplomové práce byl návrh postupu, který by urychlil řešení kombinatorických úloh na GPU, a jeho následná implementace. Nyní rozeberu výsledky rozsáhlých měření, která jsem nad implementací popsané v kapitole 7 provedl. Zabýval jsem se zejména srovnáním doby řešení různých instancí TSP na GPU a CPU. Tato měření představují první významnou část kapitoly. Dále jsem se rozhodl analyzovat přínos jednotlivých technik, které jsem při akceleraci na platformě CUDA použil. Zjištěné informace uvedu v sekci 8.3. Věnoval jsem se také kvalitě výsledného řešení, přestože ta nebyla nebyla hlavním cílem práce. Vazbu této veličiny na nastavení uživatelských parametrů popíši v poslední podkapitole.

8.1 Cíle a metodika

Měření jsem prováděl na operačním systému Linux. Pro zachycení času výpočtu jsem použil vlastní třídu `LinuxTimer` založenou na systémové funkci `gettimeofday`. Alternativou pro systém Windows je třída `WindowsTimer` postavená na nástroji `QueryPerformanceTimer`. Obě uvedené třídy jsou součástí implementace. Údajem použitým pro vyhodnocení kvality řešení je jednoduše délka nalezené cesty, respektive její odchylka od optima – u všech testovaných úloh je totiž známé. Výjimkou jsou dvě instance, u nichž je však k dispozici alespoň cesta s garancí velikosti odchylky menší než 0,2% od optima.

Vyhodnocení jsem provedl odděleně pro makro a mikro úroveň. Výsledky tak poskytují jasnější přehled o efektivitě různých částí implementace a umožňují posoudit výhodnost kombinovaných přístupů. Výpočet na jednotlivých úrovních je totiž téměř úplně oddělený (předává se mezi nimi pouze řešení v podobě jedné cesty) a nabízí se tak možnost provádět je na různých architekturách, případně nahradit jednu z nich zcela jiným algoritmem. Kopírování souřadnic do paměti zařízení, výpočet vzdáleností, manipulace s výsledným řešením a další režijní procesy trvají zcela zanedbatelnou dobu ve srovnání s hlavními částmi algoritmů a jejich vynechání tak nemá prakticky žádný dopad na výsledky měření.

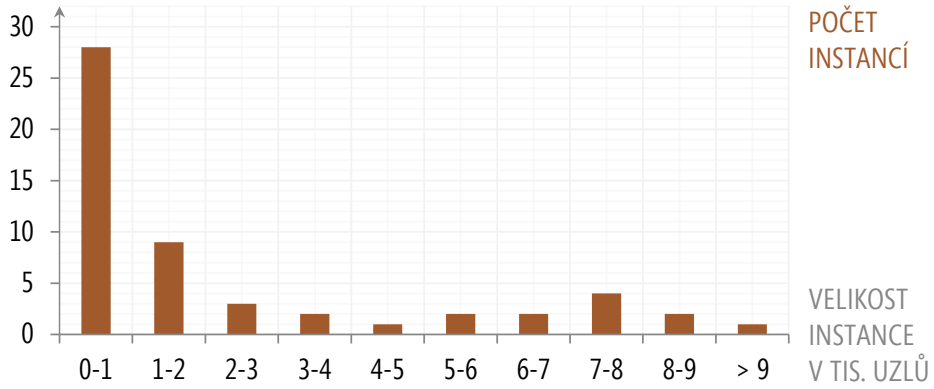
8.2 Srovnání rychlosti výpočtu na různých architekturách

8.2.1 Nastavení testů

Cílem práce byla akcelerace výpočtu. Dosažené zrychlení, které uvedu v této podkapitole, proto představuje nejdůležitější výsledek měření. Vzhledem k deterministickému charakteru výpočtu jsem neprováděl opakované běhy aplikací s toutéž konfigurací. Nalezená řešení se přesto někdy mohou na různých platformách drobně lišit – příčinou je použitý mechanismus

hledání maxima, který může při paralelním zpracování v závislosti na řazení vláken vybrat odlišné komponenty řešení, pokud mají stejné ohodnocení. Tím je ovlivněn navazující výpočet i celkový výsledek. Odchylka je však v takových případech minimální.

Testy jsem provedl na rozmanité škále celkem 54 konkrétních úloh, tedy instancí problému obchodního cestujícího. V sadě se nacházejí především úlohy z knihovny TSPLIB (původ dat je zde rozmanitý), úlohy založené na optimalizaci propojů na velkých čipech (VLSI) a dále úlohy odpovídající skutečnému cestování po velkých městech ve vybraných státech. Přibližně polovina použitých úloh má méně než tisíc uzlů, zbývající část potom docela rovnoměrně pokrývá rozsah od jednoho tisíce do deseti tisíc uzlů. Rozložení je znázorněno histogramem na obrázku 8.1. Největší řešená instance představující cestování po Argentině (`ar9152`) má 9152 uzlů. Limitujícím faktorem byla kapacita zařízení – uvedená instance téměř dokonale vyplňuje 1,5 GB hlavní paměti na Nvidia GTX 480, nejvýkonnějším CUDA zařízení, které jsem měl k dispozici. Na slabší kartě Nvidia GTS 450 a také na serveru Merlin pak velikost dostupné paměti znemožnila i řešení dalších problémů – přibližně od pěti tisíc uzlů výše.



Obrázek 8.1: Rozložení testovaných úloh podle jejich velikosti. Výrazněji jsou zastoupeny „malé“ instance do jednoho tisíce uzlů, které vykazují velkou variabilitu v charakteru dat.

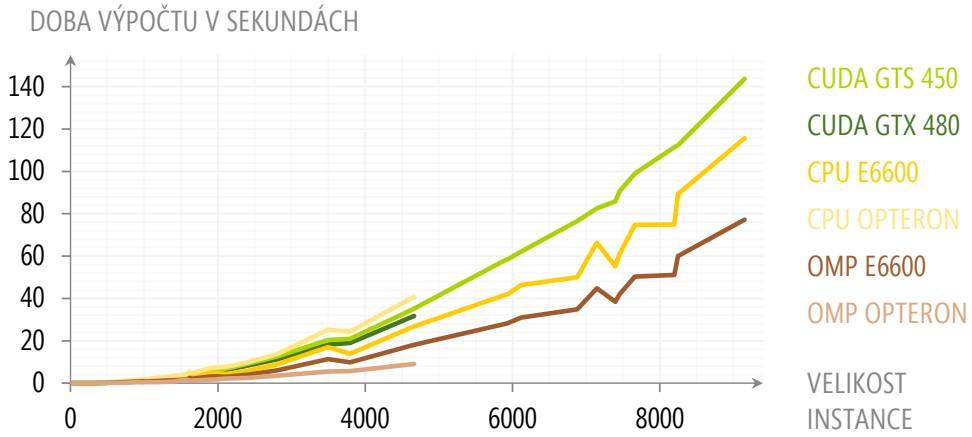
Implementace zahrnuje tři samostatné aplikace, jak jsem vysvětlil již dříve – program pro standardní procesor (`cpu`), program pro vícejádrový procesor (`omp`) a heterogenní program využívající zařízení CUDA (`cuda`). Pro každou z těchto větví jsem vyzkoušel dva konkrétní prvky hardware. V případě CUDA aplikace se jednalo o low-end kartu Nvidia GTS 450 a high-end kartu Nvidia GTX 480. U procesorových variant jsem využil desktopového procesoru Intel Pentium E6600 se dvěma jádry na frekvenci 3,06 GHz (kódové označení Wolfdale) a dvojici serverových procesorů AMD Opteron 2387 se čtyřmi jádry na frekvenci 2,8 GHz (tato dvojice byla použita současně, výpočet tedy probíhal paralelně na 8 jádrech). Pro srovnání je vhodná zejména dvojice Intel Pentium E6600 a Nvidia GTX 480, protože oba tyto výpočetní systémy patří ve své oblasti k vyšší třídě a byly uvedeny do prodeje v průběhu roku 2010 – jedná se tedy i o poměrně stejně staré produkty.

Testy, na nichž jsou založena data prezentovaná dále, lze zopakovat s využitím skriptu `testbench.sh`. Uživatelské parametry (tedy zejména počty cyklů u jednotlivých algoritmů) nebyly při měření doby výpočtu nijak upravovány – časy uvedené dále tedy odpovídají výchozímu nastavení těchto parametrů.

8.2.2 Dosažené výsledky

Příprava počátečního řešení na makro úrovni nenabízí příliš mnoho příležitostí k paralelizaci. Tuto skutečnost jsem podrobně vysvětlil již v kapitole 7.3.2 – zde pouze připomenu, že vzhledem k různým omezením je v této části programu spouštěn na zařízeních CUDA pouze jediný blok. Zařízení jsou pak značně nevyužitá – např. testovaná karta Nvidia GTX 480 v tomto případě používá jediný SM z 15 a v jeho rámci pak spouští pouze polovinu vláken, kterou by mohla. Hrubý odhad využití výpočetních prvků se tak pohybuje kolem 3 %. Povaha přístupů do paměti dále neumožňuje efektivně využít uživatelem řízené vyrovnávací paměti v podobě sdílené paměti na úrovni SM.

Ve světle uvedených skutečností není příliš překvapivé, že procesory vybavené výrazně většími vyrovnávacími paměťmi, pokročilou logikou pro zřetězené zpracování a dalšími optimalizacemi na makro úrovni platformu CUDA poráží. Uvážíme-li instance s více než jedním tisícem uzlů, pak je procesor E6600 průměrně o třetinu rychlejší než karta GTX 480. V případě použití OpenMP je pak tentýž procesor rychlejší dokonce dvakrát. Situaci pro všechny testované hardwarové konfigurace vykresluje graf 8.2.



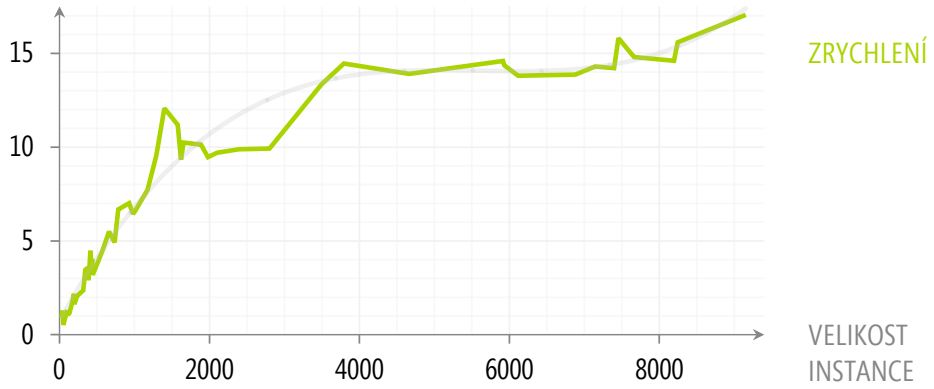
Obrázek 8.2: Doba sestavování počátečního řešení na makro úrovni. Vzhledem k sekvenční povaze výpočtu a špatné paměťové lokalitě jsou běžné procesory v této části rychlejší.

Platforma CUDA tedy vůbec není vhodná pro řešení velkých instancí v celku pomocí ACO. Pro praktické použití se nabízí zejména dva alternativní přístupy – buď přenechat práci v této části výpočtu běžnému procesoru (nejlépe s využitím OpenMP), nebo zvolit pro nalezení počátečního řešení úplně jiný algoritmus, než je ACO. Jediným potenciálním přínosem implementace makro úrovně na CUDA je úspora času hlavního procesoru a kapacity hlavní paměti v případě, že na cílovém systému jsou tyto prvky zatíženy, zatímco grafická karta nebo jiné zařízení CUDA „zahálí“. Zejména na běžných počítačích je obvykle nemožné přidat do systému plnohodnotný procesor, zato vsazení rozšiřující karty je bezproblémové. Vzhledem k nepříznivým výsledkům se makro úrovni z hlediska doby trvání výpočtu již nebudu dále věnovat.

Zpracování podúloh na mikro úrovni je již od fáze návrhu jádrem paralelizace a implementace na platformě CUDA je proto v této části mnohem efektivnější. Například karta GTX 480 bude z pohledu výpočetních jednotek vytížena minimálně na 25 % již při 15

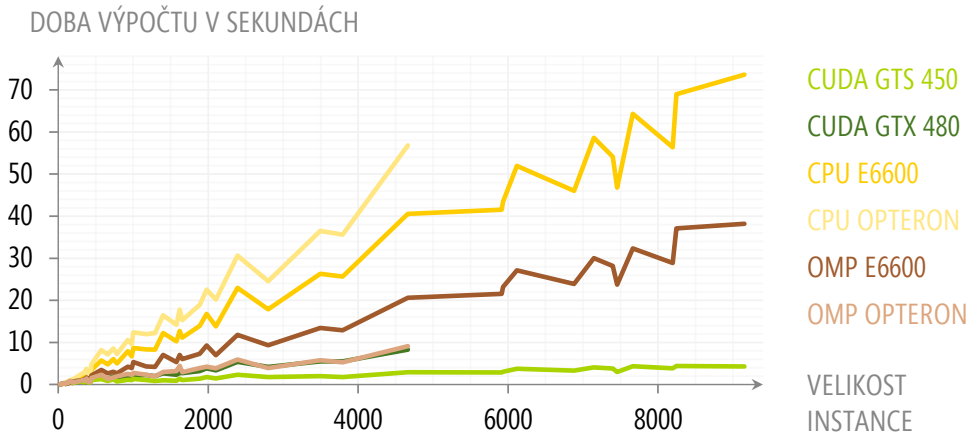
podproblémech – každý z nich zaměstná jeden multiprocesor 384 vlákny, což je přesně čtvrtina limitu architektury Fermi (1536 vláken). Kromě toho je na mikro úrovni intenzivně zapojena sdílená paměť a přístupy do hlavní paměti jsou výrazně optimalizovány (více v kapitole 7.3.3), což výpočet značně zrychluje. Právě sdílená paměť je úzkým místem, které brání umístění většího počtu podproblémů (tedy bloků) na jeden SM. Pokud bychom sdílenou paměť nepoužili, bylo by možné na každý SM umístit dokonce 4 bloky najednou a instance s více než 5 400 uzly by tak zařízení využily z pohledu výpočetních prostředků na 100 %. Výpočet by však byl pomalejší (více v kapitole 8.3). Kapacita sdílené paměti je jediným blokem využita přibližně na 95%.

Výsledky měření potvrzují, že potenciál platformy CUDA se v této části podařilo využít. S rostoucí velikosti instance významně stoupá také zrychlení dosažitelné na CUDA zařízení. Zatímco instance o celkové velikosti kolem 500 uzlů dokáže GTX 480 řešit na mikro úrovni pouze přibližně čtyřikrát rychleji než E6600, úlohy nad 3 000 uzlů jsou optimalizovány přibližně čtrnáctkrát rychleji a v případě největší použité instance `ar9152` dosahuje zrychlení dokonce hodnoty 17,1. Tento trend je zachycen na obrázku 8.3. Průměrné zrychlení u instancí s více než jedním tisícem uzlů dosahuje hodnoty 12,3.



Obrázek 8.3: Zrychlení při optimalizaci podproblémů na mikro úrovni dosažené na zařízení Nvidia GTX 480 ve srovnání s procesorem Intel Pentium E6600. Naměřená data (zeleně) jsou aproximována polynomem třetího řádu (šedě).

Měření jsem opět provedl i na dalších HW/SW konfiguracích. Všechny získané hodnoty jsou znázorněny v grafu na obrázku 8.4. Kromě zřetelného odstupu ve výkonu CUDA varianty od aplikací prováděných sekvenčně na běžných procesorech jsou v obrázku vidět dvě spojitosti, které považuji za zajímavé. Za prvé se zřetelněji projevil význam OpenMP paralelizace na procesorech Opteron. Na mikro úrovni je vidět, že dosažené zrychlení je vzhledem k použitému počtu jader velmi dobré a aplikace využívající OpenMP je tedy plnohodnotným „soupeřem“ a zároveň relevantním měřítkem v testech. Za druhé je pozitivní fakt, že i s takto výkonnou OpenMP implementací běžící na osmi jádrech serverových procesorů snese srovnání jedna z nejlevnějších GPGPU karet na trhu v roce 2011, GTS 450 (dosažené časy jsou přibližně stejné). Zařízení vyšší třídy GTX 480 i proti zmíněné OpenMP konfiguraci dosahuje zrychlení, i když už jen přibližně trojnásobného.



Obrázek 8.4: Doba optimalizace podproblémů na mikro úrovni. Návrh orientovaný na paralelizaci umožnil dosáhnout na platformě CUDA výrazného zrychlení ve srovnání s běžnými procesory. Osm jader na dvou serverových procesorech AMD Opteron 2387 dosahuje přibližně stejných časů jako jeden z nejlevnějších GPU čipů na kartě Nvidia GTS 450.

8.3 Přínos použitých akceleračních technik

Celkové zrychlení uvedené v předcházející podkapitole vzniká společným působením mnoha faktorů. Nyní se podívám, jakou měrou některé z nich k akceleraci přispívají, i když je zřejmé, že mezi použitými postupy jsou určité závislosti. Pokusím se postihnout všechny významné techniky s výjimkou hledání cest z více počátečních měst, protože tento prvek má kromě rychlosti značný vliv také na kvalitu řešení. Navíc je součástí návrhu již od základu a jeho testování by si tak vyžádalo výrazné úpravy zdrojového kódu.

Zatímco význam zvýšeného počtu celých skupin mravenců tedy vyhodnocovat nebudu, na přínos konceptu skupin z pohledu jednotlivých mravenců se nyní podívám. Pro účely získání dat jsem dočasně nastavil parametr `nMacroAntsPerSquad`, resp. `nMicroAntsPerSquad`, na hodnotu `1`, což odpovídá situaci, kdy by koncept skupin nebyl do implementace zaveden. Následně jsem srovnal dobu výpočtu na makro i mikro úrovni s případem, kdy by byly uvedené parametry nastaveny na standardní hodnoty, tedy `8`, resp. `4`. Zrychlení dosažené aplikací konceptu skupin v závislosti na velikosti instance je zachyceno na grafu na obrázku 8.5. Zatímco na makro úrovni je zrychlení poměrně stabilní s průměrnou hodnotou kolem 3,5, na mikro úrovni se více mění v závislosti na konkrétní úloze, přičemž průměr je v tomto případě přibližně 1,4.

Koncept skupin je jediným faktorem zmíněným v této podkapitole, který má pozitivní vliv na dobu výpočtu na makro úrovni. Zde je vidět, že přestože běžné procesory jsou na makro úrovni v každém případě úspěšnější než platforma CUDA, jejich odstup je stále poměrně přijatelný v porovnání se situací, kdy by koncept skupin nebyl u CUDA použit. Na mikro úrovni je již význam skupin menší – faktory zmíněné dále v této podkapitole jsou mnohem důležitější.

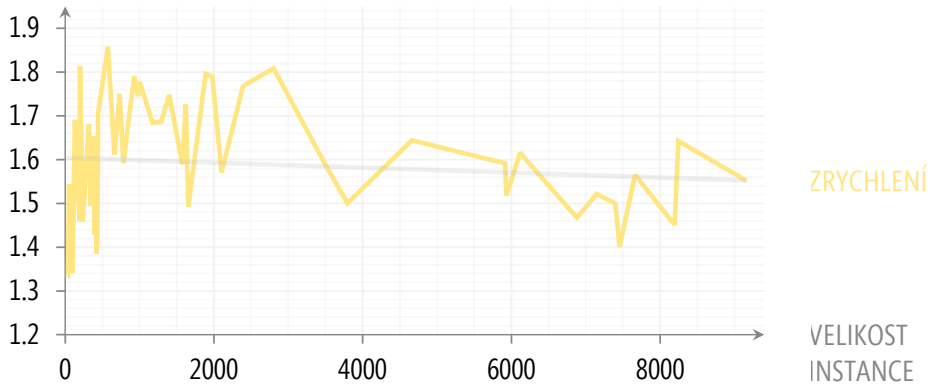
Jak jsem uvedl již v kapitole 7.3.2, prostředky nejrychlejší vyrovnávací paměti na CUDA zařízení jsou na makro úrovni řízeny hardwarem (jsou využity jako *L1 cache*). Mikro rámce jsou oproti tomu navrženy tak, aby byla zmíněná paměť co nejfektivněji řízena uživatelsky

Obrázek 8.5: Zrychlení dosažené zapojením konceptu skupin. V horní části je znázorněna situace na makro úrovni, kde zařízení u menších instancí dokáže využít hardwarově řízenou L1 cache. Ve spodní části je vidět stabilnější charakter zrychlení na mikro úrovni.

(tedy jako *sdílená paměť*). Zavedení univerzální matice, do níž se v průběhu algoritmu dynamicky nahrávají nejvíce potřebná data, přináší průměrně 1,6násobné zrychlení.

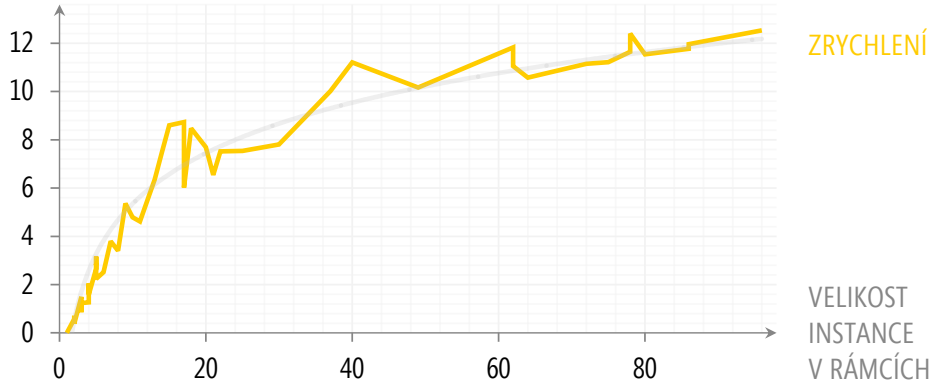
Zde je třeba zmínit, že samotný přístup do sdílené paměti je mnohem výhodnější, než by uvedenému zrychlení odpovídalo. U varianty bez využití sdílené paměti totiž hraje roli fakt, že na každý SM pak lze umístit více bloků. Velikost sdílené paměti je omezující hodnotou, která u základní varianty limituje počet bloků umístitelných na každý SM na jediný. Na naměřených hodnotách znázorněných v grafu na obrázku 8.6 je však vidět, že na všech úlohách v testovací sadě je výhodnější použít postup založený na sdílené paměti.

Nejvýznamnější metodou akcelerace ACO na paralelních zařízeních je již podle literatury [14] rozdělení úlohy na podproblémy, jejich nezávislé řešení a následné spojení dílčích výsledků do jediné cesty. Tento postup se ukázal být skutečně klíčovým i v praktické části práce. Výslednou verzi implementace jsem srovnával s variantou, kdy jsou jednotlivé mikro rámce sice řešeny na zařízení, avšak sekvenčně. Toho jsem dosáhl pomocí cyklu, který postupně volá funkci `microFindTour` pro jednotlivé rámce a po každém takovém volání si vyžádá synchronizaci CUDA zařízení (jinak by byly bloky spouštěny asynchronně a tím do určité míry i paralelně).



Obrázek 8.6: Zrychlení dosažené umístěním datových struktur do sdílené paměti. Význam tohoto faktoru je částečně potlačen tím, že nároky na sdílenou paměť při jejím zapojení snižují počet bloků, které je možné umístit na každý SM.

Oproti předcházejícím technikám můžeme na grafu dosaženého zrychlení 8.7 pozorovat značnou závislost na celkové velikosti instance s rostoucí tendencí. Pro lepší představu nejsou na ose x vyneseny přímo rozměry úloh, ale počty řešených rámců. Ty získáme poděleníím rozměru problému parametrem `nCitiesInCluster`, který byl nastaven na výchozí hodnotu `96`. Nejmenší instancí z testovací sady, která umožňuje nasadit na každý SM karty GTX 480 alespoň jeden blok, je `fl1400` s 8,6násobným zrychlením. Největší úlohu `ar9152` lze pak na mikro úrovni řešit díky paralelnímu zpracování rámců až 12,5krát rychleji. To je vzhledem k použitému počtu výpočetních prvků (tedy SM, kterých bylo na zařízení k dispozici 15) poměrně dobrý výsledek.

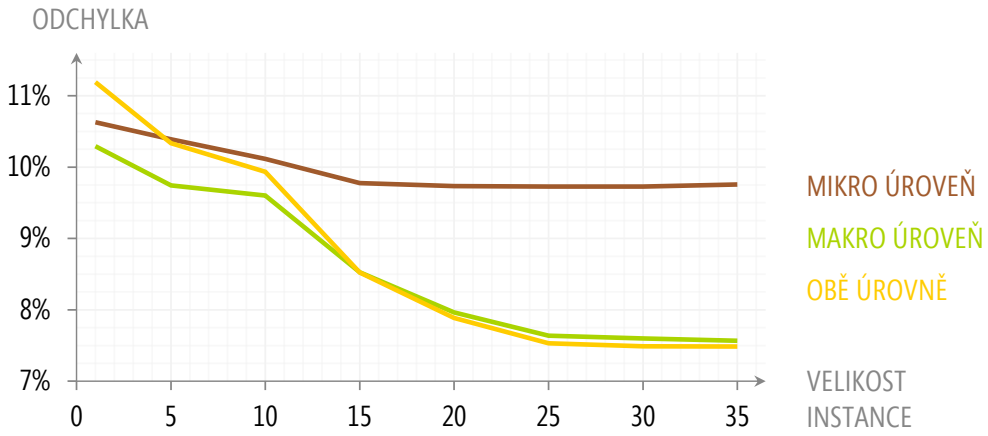


Obrázek 8.7: Zrychlení dosažené současným řešením vícero podproblémů na mikro úrovni. Každý podproblém přitom odpovídá jednomu rámci, jejichž celkový počet u různých instancí je vynesen na ose x. Naměřená data jsou aproximována logaritmickou křivkou.

8.4 Vliv parametrů na kvalitu řešení

Vzhledem k charakteru použitých algoritmů, které hledají pouze „dobré“ řešení, nikoliv však řešení optimální, není doba běhu jediným výsledkem celého procesu. Zajímavé mohou být také změny ve kvalitě nalezené cesty, jimž budu věnovat tuto podkapitolu. Analýza bude podstatně méně rozsáhlá, neboť odchylky od optima, kterých algoritmy dosahují při různých konfiguracích, se tolik neliší. Budu se spíše snažit ukázat, že modifikace vysvětlené v kapitole 6.3 nedegradují implementované ACO na *greedy search* nebo podobný algoritmus využívající pouze údajů založených na vzdálenostech, ale že je stále zapojena i paměťová složka heuristická informace v podobě feromonových stop.

Provedl jsem rozsáhlá měření, během nichž jsem každou úlohu z testovací sady spustil ve 24 různých konfiguracích, u nichž byly hodnoty z množiny $\{1, 5, 10, 15, 20, 25, 30, 35\}$ nastaveny nejdříve u parametru `nMacroAcoCycles`, potom u parametru `nMicroAcoCycles` a nakonec u obou zároveň. Vliv na kvalitu řešení jsem následně vypočítal jako průměr odchylek od optimálních cest u všech instancí při daném nastavení. Závislost vypočtených hodnot na použitých parametrech zachycuje graf 8.8.



Obrázek 8.8: Závislost kvality řešení na počtu iterací jednotlivých algoritmů. Na ose y je vynesena průměrná odchylka délky nalezené cesty od známého optima. V grafu je zachycena významná vazba celkového výsledku na kvalitu počátečního řešení.

Vidíme, že navyšování počtu cyklů na mikro úrovni nad 15 nemá valný význam, což je způsobeno zejména tím, že jednak se hledá řešení z každého města, druhak je aplikováno lokální prohledávání nad celým rozsahem problému. Na makro úrovni naopak dochází k poměrně významnému poklesu odchylky ještě při navýšení počtu cyklů z 20 na 25. V důsledku chybějícího lokálního prohledávání a deterministického výběru další komponenty je feromonová stopa jediným zdrojem zlepšení a každý další cyklus její tvorby může přinést významný pokrok. Výsledky dosažené nastavováním obou zmíněných parametrů najednou kopírují u vyšších hodnot průběh získaný nastavením `nMicroAcoCycles` na výchozí hodnotu `15` a rostoucí hodnoty `nMicroAcoCycles`. Můžeme tedy říct, že kvalita počátečního řešení má na kvalitu celkového řešení výrazný vliv a nahrazení ACO jiným algoritmem na makro úrovni by tak mohlo být zajímavou alternativou k vytvořené implementaci, pokud by zvolený algoritmus měl lepší předpoklady pro paralelní řešení úlohy v celku.

Kromě uvedených parametrů mohou uživatelé snadno nastavit také hodnoty `alpha` a `beta` udávající relativní význam jednotlivých složek heuristické informace u ACO. Vytvořil jsem tedy obdobný test jako v případě počtu cyklů, při němž byla každá instance spuštěna v celkem 63 konfiguracích. Zde se mi však nepodařilo dosáhnout žádných výsledků, které by stály za zmínku. Rozdíly byly naprosto minimální a napříč různými instancemi nebylo možné pozorovat žádný trend. V průměru se tedy úlohy téměř ve všech konfiguracích řešily velmi podobně jako v případě výchozího nastavení `alpha` na `1.0` a `beta` na `2.0`.

Kapitola 9

Závěr

Diplomovou práci jsem věnoval akceleraci postupů pro řešení kombinatorických úloh s využitím GPU. Zabýval jsem se zejména *algoritmem mravenčí kolonie (ACO)*, jak bylo požadováno v zadání, avšak soustředil jsem se také na techniku lokálního prohledávání *2-opt* sloužící jakožto nezbytný doplněk uvedené heuristické metody. V rámci práce jsem provedl teoretický rozbor problému, navrhnul jsem postup pro paralelizaci i akceleraci výpočtu a následně jsem provedl implementaci na platformě Nvidia CUDA. Použil jsem při tom známý *problém obchodního cestujícího (TSP)* jako případovou studii.

Podařilo se mi využít několika odlišných technik vedoucích ke zrychlení výpočtu. Zavedl jsem paralelní zpracování na úrovni jednotlivých jader (*streaming processor, SP*) i na úrovni větších celků (*streaming multiprocessor, SM*). Optimalizoval jsem přístupy do hlavní paměti CUDA zařízení a intenzivně jsem zapojil uživatelem řízenou *sdílenou paměť* umístěnou na čipu. Dále jsem upravil vybrané části obou heuristických algoritmů tak, aby byly vhodnější pro paralelní provádění. I přes značnou složitost řešeného problému se mi nakonec podařilo dosáhnout lineární závislosti doby výpočtu na velikosti problému.

Součástí práce je nejen paralelní kód pro GPU, ale také sekvenční kód pro běžné procesory a dále paralelní kód pro vícejádrové procesory využívající nástroje OpenMP. Použité algoritmy i základní nastavení jsou ve všech případech stejné. Programy jsem vytvořil za účelem přesného vyhodnocení výkonnosti implementace na platformě CUDA.

Rozsáhlá sada úloh použitá při testování zahrnovala problémy o velikosti od jednoho do deseti tisíc uzlů. Ačkoliv povaha řešené kombinatorické úlohy příliš neodpovídá obvyklému využití architektury GPU v oblasti obecných výpočtů, podařilo se mi ve srovnání s běžnými i vícejádrovými procesory dosáhnout významného zrychlení. Například podproblémy tvořící největší použitou instanci o 9 152 uzlech dokázala grafická karta Nvidia GTX 480 řešit více než 17krát rychleji než srovnatelný procesor Intel Pentium E6600 provádějící sekvenční kód na frekvenci 3,06 GHz. Doba výpočtu na CUDA zařízení byla výrazně kratší také ve srovnání s OpenMP verzí. Výsledky práce dokládají, že schopnost moderních grafických čipů provádět obecné výpočty může být významným přínosem i v oblasti diskrétních problémů, které nejsou pro GPGPU typickou doménou.

Vytvořené řešení představuje spolehlivý základ, který otevírá možnosti pro další zlepšení v podobě optimalizace dílčích procesů. Nejvíce prostoru se nachází v oblasti dělení úlohy na podproblémy, která trpí vysokými paměťovými nároky ACO. Další příležitost by mohla spočívat v zavedení pokročilejšího algoritmu lokálního prohledávání, jakým je např. *Lin-Kernighan heuristic*.

Literatura

[1] *NVIDIA CUDA C programming guide*. NVIDIA Corporation, 2011, verze 4.0.

[2] *Transistor count* [online]. `http://en.wikipedia.org/wiki/Transistor_count`, 25.10.2011 [cit. 2011-10-07].

[3] *Intel homesite* [online]. `http://www.intel.com/`, c2011 [cit. 2011-10-07].

[4] *AMD homesite* [online]. `http://www.amd.com/`, c2011 [cit. 2011-10-30].

[5] *HPC systems homesite* [online]. `http://www.hpcsystems.com/`, c2011 [cit. 2011-10-30].

[6] *NVIDIA homesite* [online]. `http://www.nvidia.com/`, c2011 [cit. 2011-10-30].

[7] Aarts, E.; Lenstra, J.: *The TSP: A Case Study In Local Optimization*. John Wiley & Sons, Inc., 1997, iSBN 978-0-691-11522-1.

[8] Alba, E.: *Parallel metaheuristics: A new class of algorithms*. John Wiley & Sons, Inc., 2005, iSBN 978-0-471-67806-9.

[9] Chen, D.; Batson, R.; Dang, Y.: *Applied integer programming*. John Wiley & Sons, Inc., 2010, iSBN 978-0-470-37306-4.

[10] Cook, W.: *The travelling salesman problem* [online]. `http://www.tsp.gatech.edu/`, c2011 [cit. 2011-09-29].

[11] Doerner, K.; Hartl, R.; Reimann, M.: *A hybrid ACO algorithm for the full truckload transportation problem*. 2001.

[12] Dorigo, M.: *Ant colony optimization* [online]. `http://www.aco-metaheuristic.org/`, 10.3.2009 [cit. 2011-09-23].

[13] Dorigo, M.: *Ant colony optimization* [online]. `http://www.scholarpedia.org/article/Ant_colony_optimization`, 28.3.2007 [cit. 2011-11-15].

[14] Dorigo, M.; Stützle, T.: *Ant colony optimization*. MIT press, 2004, iSBN: 978-0-262-04219-2.

[15] Grama, A.; Gupta, A.; Karypis, G.; aj.: *Introduction to parallel computing*. Addison-Wesley, 2003, iSBN 978-0201648652.

[16] Huang, R.-H.; Yang, C.-L.: Ant colony system for job shop scheduling with time windows. *The International Journal of Advanced Manufacturing Technology*, 2008, ISSN 0268-3768.

[17] Kirk, D.; Hwu, W.: *Programming massively parallel processors: A hands-on approach*. Morgan Kaufmann Publishers, 2010, iSBN 978-0-12-381472-2.

[18] Kumar, R.; Li, H.: *On asymmetric TSP: transformation to symmetric TSP and performance bound*. 2007.

[19] Kučera, L.: *Kombinatorické algoritmy*. Státní nakladatelství technické literatury, 1989.

[20] Rabanal, P.; Rodríguez, I.; Rubio, F.: Using river formation dynamics to design heuristic algorithms. In *Unconventional computation*, Springer Berlin / Heidelberg, 2007, ISBN 978-3-540-73553-3.

[21] Reinelt, G.: *TSPLIB* [online]. `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`, 6.8.2008 [cit. 2012-04-05].

[22] Rossi, F.; van Beek, P.; Walsh, T.: *Handbook of constraint programming*. Elsevier, 2006, iSBN 978-0-444-52726-4.

[23] Sanders, J.; Kandrot, E.: *CUDA by example*. Addison-Wesley, 2011, iSBN 978-0-13-138768-3.

[24] Sutter, H.: *The free lunch is over: A fundamental turn toward concurrency in software* [online]. `http://www.gotw.ca/publications/concurrency-ddj.htm`, c2009 [cit. 2011-11-15].

[25] Tsutsui, S.; Fujimoto, N.: ACO with tabu search on a GPU for solving QAPs using move-cost adjusted thread assignment. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, ACM, 2011, ISBN 978-1-4503-0557-0.

Seznam příloh

Příloha A

Obsah CD

app	zdrojové soubory všech vytvořených programů
dox	programová dokumentace ve formátu HTML
tex	zdrojové soubory technické zprávy
dt-xpecha02.pdf	technická zpráva ve formátu PDF
dt-xpecha02-print.pdf	technická zpráva ve formátu PDF pro tisk
readme	nápověda k obsahu CD