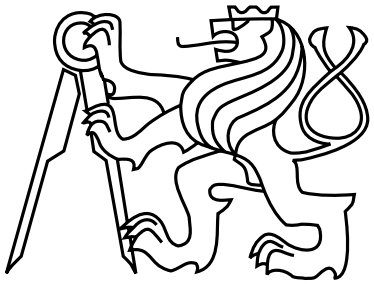
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

Řešení problémů rozvrhování s omezenými zdroji za využití grafických karet

Bc. Tomáš Poledný

Vedoucí práce: Ing. Libor Bukata

Studijní program: Otevřená informatika, Magisterský

Obor: Softwarové inženýrství

8. května 2014

Poděkování

Chtěl bych poděkovat především vedoucímu práce Ing. Liboru Bukatovi za vedení mé diplomové práce, za pomoc při řešení problémů a za poskytnutí serveru pro experimenty. Dále bych chtěl poděkovat mé rodině a přítelkyni Martině za korekturu a podporu při vypracování této práce.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 9. 5. 2014 .....

Abstract

The master thesis deals with the solving NP-hard problem for the Resource Constrained Project Scheduling Problem (RCPSP) by a genetic algorithm. The core of the thesis is a design and its implementation of algorithm using the latest graphical cards. This thesis comes with a way to use several graphical cards at the same time for solving RCPSP by the genetic algorithm. The thesis also includes an implementation of the same algorithm for CPU without using graphical cards. This thesis aims to improve the quality of solving instances of problem RCPSP and the speedup of computing. The implementations were tested on a set of standard problems of RCPSP and compared to other implementations from the literature.

Abstrakt

Tato diplomová práce se zabývá řešením NP-těžkého plánovacího problému Resource Constrained Project Scheduling Problem (RCPSP) pomocí genetického algoritmu. Hlavní částí práce je návrh a implementace algoritmu s využitím nejmodernějších grafických karet. Tato práce přináší způsob, jak lze s využitím více grafických karet naráz řešit problém RCPSP za pomoci genetického algoritmu. Součástí práce je taktéž implementace stejného algoritmu s využitím CPU bez použití grafických karet. Hlavním cílem práce je zvýšení kvality řešení instancí problému RCPSP a zrychlení výpočtu. Implementace byly testovány na standardní množině problémů RCPSP a porovnány s dalšími pracemi.

Obsah

Seznam obrázků

Seznam tabulek

Seznam algoritmů

Kapitola 1

Úvod

V dnešní době se u některých algoritmů naráží na hranice rychlosti výpočtu s využitím CPU. Problém spočívá v tom, že se frekvence CPU nedá neustále zvyšovat, proto se začíná prosazovat více výpočetních jader v rámci jednoho CPU. Díky tomu může CPU zpracovávat více výpočtů současně. Dnešní grafické karty již obsahují mnoho výpočetních jader, které jsou zároveň dostatečně výkonné. Díky jejich zaměření na paralelní zpracování dochází u některých algoritmů při použití grafické karty ke značnému zrychlení oproti paralelní verzi pro CPU.

Tato diplomová práce se zabývá Resource Constrained Project Scheduling Problem (dále jen RCPSP), což je problém plánování aktivit se vzájemnými vztahy na různé typy zdrojů s omezenými kapacitami. RCPSP je NP-těžký problém [2]. Tento problém je velmi známý. Ačkoliv existuje mnoho prací, které se jím zabývají, jen velmi málo z nich řeší problém prostřednictvím grafických karet. Tato práce se pokouší vylepšit dosud známá řešení problému RCPSP pomocí paralelizace s využitím grafických karet. Práce se zaměřuje na využití paralelizace jak na CPU, tak i na GPU a následné porovnání implementací obou verzí. Zároveň se práce zabývá vylepšením řešení problému s využitím více grafických karet pro výpočet naráz. K řešení tohoto problému bude využit genetický algoritmus.

1.1 GPU

V dnešní době, kdy jsou grafické karty velmi výkonné, mnoho výrobců poskytuje framework pro výpočty na těchto kartách. Pro výpočty na grafických kartách se nejvíce hodí algoritmy, které lze z velké části paralelizovat. U takovýchto algoritmů lze dosáhnout mnohonásobného zrychlení ve srovnání se stejnou verzí pro CPU. Je tomu tak dáno z původního účelu grafických karet - zpracování obrazu, kdy je potřeba aplikovat operaci na velké množství dat. Tato práce se zabývá implementací pro CUDA. Důvodů, proč byla zvolena právě CUDA, je několik. CUDA je v současné době velmi používaná a nabízí mnoho příkladů použití. Za CUDA stojí silný hráč na poli grafických karet, tj. NVIDIA. Hlavní výhodou je také dostupnost nástrojů pro vývoj a grafických karet od NVIDIA, které tuto technologii podporují.

1.2 Cíle a požadavky práce

Cílem této práce je navrhnout a poté implementovat algoritmus, který řeší Resource Constrained Project Scheduling Problem (RCPSP) s důrazem kladeným na nalezení co nejlepších řešení problému. Implementace bude provedena za pomoci genetického algoritmu. Hlavním cílem algoritmu bude nalézt co nejlepší řešení problému v omezené době. Algoritmus bude implementován jak pro CPU, tak i pro nejmodernější GPU s využitím platformy CUDA. Dalším úkolem této práce je využít více grafických karet tak, aby bylo dosaženo zlepšení kvality řešení. Implementace bude provedena pomocí programovacího jazyka C++ a CUDA C. Další částí je provedení experimentů s algoritmy a porovnání s podobnými pracemi. Díky tomu, že je problém RCPSP velmi známý a existuje mnoho prací, lze tedy porovnat, jak bude implementace úspěšná. Dalším cílem této práce je vylepšit dosud nejlepší známá řešení z benchmarku PSPLIB [9] pro RCPSP.

1.2.1 Shrnutí cílů a požadavků

- navrhnout algoritmus řešící RCPSP
- implementovat sériovou/paralelní verzi algoritmu pro CPU
- implementovat verzi algoritmu pro CUDA
- důraz kladen na kvalitu řešení
- využít více GPU
- experimenty s implementací a porovnání vůči podobným pracím
- vylepšení řešení z benchmarku PSPLIB pro RCPSP

1.3 Struktura diplomové práce

Celá práce je členěná do 10 kapitol. V kapitole 2 Existující práce jsou uvedeny práce, které se zabývají podobnou tématikou. Kapitola 3 CUDA a GPU architektura je úvod do problematiky obecných výpočtů na grafických kartách a představuje framework CUDA. Popis problému RCPSP a jeho vlastností je uveden v kapitole 4 RCPSP. Kapitola 5 Genetický algoritmus představuje problém genetického algoritmu a jeho paralelní verze. Návrh genetického algoritmu řešícího problém RCPPSP a popis použitých technik je uveden v kapitole 6 Návrh genetického algoritmu řešící RCPSP problém. V kapitole 7 Implementace pro CPU jsou uvedeny detaily implementace algoritmu z předchozí kapitoly pro CPU. Implementace pro GPU a její detaily jsou uvedeny v kapitole 8 Návrh a implementace GPU verze algoritmu. Experimenty s oběma verzemi implementací a jejich porovnání s existujícími pracemi je uvedeno v kapitole 9 Experimenty. V závěru práce v kapitole 10 Závěr je shrnutí celé práce a dosažených výsledků.

Kapitola 2

Existující práce

Tato práce vychází především z několika prací. Článek, který velmi inspiroval návrh a implementaci, je **A Biased Random-Key Genetic Algorithm with Forward-Backward Improvement for the Resource Constrained Project Scheduling Problem** [6] od autorů *J. F. Gonçalves*, *M. G. C. Resende* a *J. J. M. Mendes*. Tato práce popisuje genetický algoritmus s tak zvanými náhodnými čísly, které jsou využity jako chromozomy jedinců. Tento článek vychází z dalšího článku od stejných autorů **A random key genetic algorithm for the resource constrained project scheduling problem** [12]. Tento článek se liší v použití jiné reprezentace chromozomu a zavádí ohodnocovací funkci nazvanou modified makespan. Tyto články byly zvoleny z důvodů dobrých výsledků v benchmarku PSPLIB [9].

Další článek, který se zabývá řešením RCPSP, je **A Decomposition-Based Genetic Algorithm for the Resoure-Constrained Project-Scheduling Problem** [5] od autorů *D. Debels* a *M. Vanhoucke*. Tento článek je také zaměřen na řešení RCPSP pomocí genetického algoritmu, avšak oproti předchozím dvěma článkům zavádí jiný způsob křížení. Dále zavádí způsob, jak rozdělit problém na subproblémy a ty poté vyřešit a opět spojit a tím vytvořit řešení problému. Také tento článek dosáhl jednoho z nejlepších výsledků pro PSPLIB benchmark.

Článek s názvem **New meta-heuristics for the resource-constrained project cheduling problem** [11] od autorů *A. Lim*, *H. Ma*, *B. Rodrigues*, *S.T. Tan* a *F. Xiao* je inspirován pohybem molekul a popisuje mírně odlišný algoritmus od simulovaného žíhání nazvaný A new annealing-like search heuristics. Dále zavádí vylepšení řešení pomocí tzv. skákání a tzv. chození. Skákání je skokové vylepšení řešení. Zavádí také různé metody, jak skákání, resp. chození provést. Také tento článek dosahuje jedněch z nejlepších výsledků v PSPLIB benchmarku.

O pokus s vylepšením genetického algoritmu pomocí proměnné velikosti populace se snaží článek **GAVaPS - a Genetic Algorithm with Varying Population Size** [1] od autorů *J. Arabas*, *Z. Michalewicz* a *J. Mulawka*. Zavádí tzv. lifetime neboli životnost jedince a také nabízí několik možností, jak tento lifetime spočítat.

Článek, který se zabývá implementací genetického algoritmu na GPU pro řešení problému Flow Shop s názvem **Accelerating an Algorithm for Flow Shop Scheduling Problem on the GPU** [19] od autorů *P. Šůcha* a *T. Zajíček*, přináší popis, jak genetický algoritmus implementovat pomocí CUDA.

Jediná práce, která se zabývá přímo RCPSP a využívá GPU, resp. CUDA, se nazývá **GPU algorithms design for Resource Constrained Project Scheduling Problem** [3] od autora *L. Bukaty*. Tato práce však nevyužívá genetický algoritmus, ale algoritmus nazvaný Tabu Search.

Kapitola 3

CUDA a GPU architektura

Grafické karty již neslouží jenom pro zobrazování a práci s grafikou, ale jsou velmi vhodné i pro další výpočty. I proto mnoho výrobců grafických karet nabízí způsob, jak grafické karty využít i jinak. V současné době se pro účely obecných výpočtů na grafických kartách nejvíce využívají dvě technologie a to OpenCl [7] a NVIDIA CUDA [14]. Právě struktura grafických karet a původní účel pro zpracování grafiky umožňuje oproti CPU mnohem lepší paralelizaci. Díky tomu mohou být v určitých případech grafické karty mnohonásobně rychlejší než stejný výpočet na CPU. Srdce grafické karty je GPU (graphics processing unit), což je procesor grafické karty. Na GPU lze provádět většinu výpočtů stejně jako na CPU.

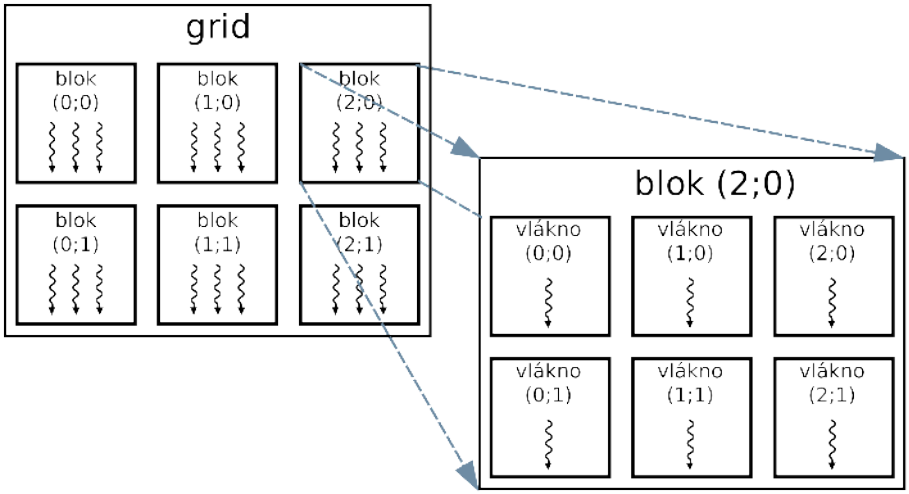
Existují dva modely využití CUDA, resp. obecných výpočtů na grafických kartách. Pokud je celý výpočet prováděn na GPU, nazývá se tento model homogenní. Pokud je na GPU prováděna pouze část výpočtu, například řazení, jedná se o model heterogenní.

3.1 CUDA

CUDA je technologie od firmy NVIDIA [14]. Zkratka CUDA znamená Compute Unified Device Architecture. Je to jak hardwarová platforma, tak i softwarová platforma. CUDA byla představena v roce 2006. Oficiálně jsou podporovány programovací jazyky C/C++ a Fortran. Existuje však mnoho programovacích jazyků, které dokáží CUDU využít, jako jsou např. Java a Python. V případě jazyka C, resp. C++, je program pro CUDA napsán v těchto jazycích s rozšířením CUDA C. CUDA C přidává do C některá další klíčová slova. NVIDIA poskytuje mnoho oblíbených nástrojů pro vývoj na různých platformách. Například debuger cuda-gdb je rozšířením gdb. Pro vývoj jsou připravena rozšíření pro dvě IDE nazvané NVIDIA Nsight pro Visual Studia a pro Eclipse. Následující text se snaží nastínit a přiblížit základní vlastnosti CUDA, další podrobnosti lze nalézt v dokumentaci CUDA [15].

Zařízení (grafická karta), které vykonává kód CUDA, se nazývá device. Zbytek počítače, resp. CPU a paměť RAM, na kterém je vykonán jiný kód než CUDA, je v rámci CUDA nazýván jako host. Funkce, která se spouští na GPU, se v CUDA nazývá kernel. Tato funkce je v podstatě kód, který vykonává každé vlákno. Tato vlákna se sdružují do bloků a bloky se sdružují do gridu viz 3.1. Rozměry bloku a gridu mohou být až trojrozměrné. Velikost bloku a gridu je omezena. Pro zjištění souřadnice vlákna se využívají speciální struktury *blockIdx* pro zjištění souřadnice bloku a *threadIdx* pro zjištění souřadnice vlákna

v bloku. Tyto struktury mají tři atributy x, y a z, které určují souřadnici v rozměru x, y a z. Pro zjištění počtu bloků se opět používá speciální struktura *gridDim* se stejnými atributy rozměrů jako pro zjištění indexu. Pro zjištění velikosti bloku se používá podobná speciální struktura jako *gridDim* a to *blockDim* se stejnými parametry. Pokud tedy chceme zjistit „globální“ souřadnice vlákna v rozměru x (resp. souřadnice $y = 0$ a $z = 0$), provedeme to pomocí výpočtu: $blockDim.x * blockIdx.x + threadIdx.x$. [15]



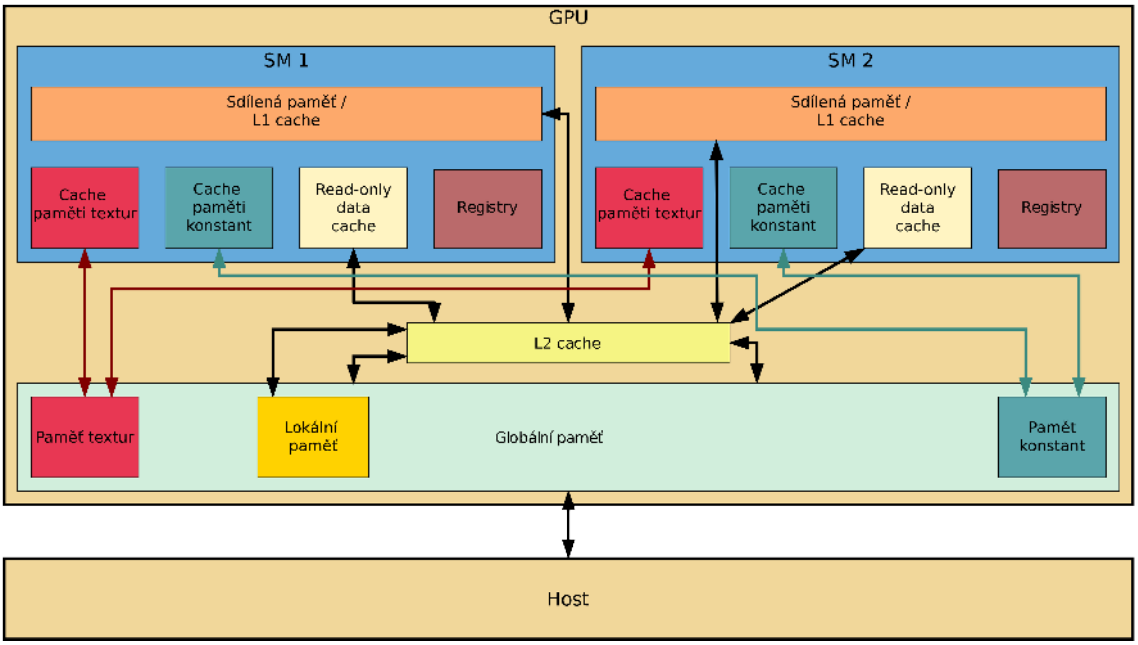
Obrázek 3.1: Seskupení vláken do bloku a gridu

3.1.1 Architektura CUDA

Architektury GPU se od sebe odlišují. To, jaké vlastnosti CUDA dokáže využít, je označováno jako compute capability (např. 1.2, 2.0, 3.5). Různé verze GPU se liší i v počtu a stavbě jednotek a velikostech pamětí. V principu jsou si však podobné. Každé GPU může obsahovat několik streaming multiprocesorů (SM), které jsou základní hardwarovou jednotkou. Multiprocesor se skládá z velkého množství relativně jednoduchých streaming procesorů, které vykonávají vlákna. Na každém multiprocesoru může být spuštěno několik bloků najednou. Blok vláken se rozdělí do tak zvaného warpu. Warp je v současné době (architektura Kepler) 32 vláken. Warp je zpracován multiprocesorem najednou jako SIMD. To znamená, že na každé vlákno ve warpu je použita stejná instrukce. Multiprocesor obsahuje plánovač, který plánuje, jak se budou warpy spouštět. Plánovač nezaručuje pořadí warp, jejich spuštění může být tedy „náhodné“. Přepínáním warp lze zakrýt rychlost přístupu k pomalejší paměti. Je tedy výhodné spouštět na každém multiprocesoru více bloků.

3.1.2 Typy pamětí

V architektuře CUDA je několik druhů pamětí. Paměti se mezi sebou liší svou rychlostí, resp. dobou přístupu, způsobem použití a viditelností v rámci seskupení. Obrázek, jak jsou paměti na grafické kartě umístěny, je vyobrazen na obrázku 3.2.



Obrázek 3.2: Paměťový model CUDA

3.1.2.1 Globální paměť

Globální paměť je paměť na grafické kartě mimo multiprocesor. Tato paměť je relativně veliká - v řádech GB. Přístup k této paměti je ze všech pamětí na grafické kartě nejpomalejší. Pro přístup k této paměti se často používá technika coalescing. Kdy vlákna ve warpu přistupují k části paměti, která je za sebou. Přístup, resp. viditelnost, je odevšud včetně hosta. Nejčastěji se tato paměť používá pro zkopírování potřebných dat pro výpočet z hostu.

Pozn.: Pojem globální paměť má dva významy. První je označení virtuální paměti CUDA. Ve druhém významu je termín globální paměť myšlen jako paměť zařízení (device memory).

3.1.2.2 Registry

Registry jsou umístěny přímo v multiprocesoru a je to nejrychlejší paměť na celé kartě. Data v nich uložená jsou viditelná pouze v rámci vlákna. Počet registrů na vlákno jde nastavit parametrem při kompilaci programu. Počet registrů na multiprocesoru je v řádu tisíců 32 bitových registrů (pro 3.5 je to 64 tisíc).

3.1.2.3 Lokální paměť

Lokální paměť je část globální paměti, má tedy podobné vlastnosti jako tato paměť. Pokud již počet registrů pro vlákno nestačí, ukládají se další lokální data vlákna právě do této paměti. Viditelnost je stejná jako v případě registrů, tedy pouze z vlákna.

3.1.2.4 Sdílená paměť

Sdílená paměť slouží hlavně pro komunikaci v rámci bloku. Je tedy viditelná pouze v rámci bloku. Tato paměť je umístěna v multiprocesoru, tomu také odpovídá rychlost práce s ní. Má nižší latenci a větší šířku pásma než lokální a globální paměti. Velikost sdílené paměti je v řádu kB (pro 3.5 je to 48 kB) pro každý multiprocesor.

3.1.2.5 Paměť textur

Paměť textur, jak již název napovídá, je původně určená pro ukládání textur. Tato paměť je uložena v části globální paměti. Její výhoda oproti globální paměti je speciální způsob cachování. Protože je původně určená pro textury, je cachováno 1D, 2D nebo 3D okolí dle uložených dat. Pokud tedy probíhá čtení naráz z okolí, je vše nacachováno a přístup je velmi rychlý. Pokud tomu tak není, přístup k paměti je náhodný, rychlost se nemusí projevit. Cache této paměti je umístěna na každém multiprocesoru. Viditelná je odevšud včetně hostu. Tato paměť je po naalokování konstantní a neměnná. Její velikost cache na multiprocesoru je několik kB (pro 3.5 je to 12 - 48 kB).

3.1.2.6 Paměť konstant

Paměť konstant je stejně jako paměť textur uložena v globální paměti. Je cachována do cache, která je uložená na multiprocesoru. Jak již název napovídá, je od alokace paměti na hostu neměnná. Do této paměti se ukládají také parametry kernelu. Velikost cache je v řádu jednotek kB (pro 3.5 je to 8 kB).

3.1.2.7 Read-Only Data cache

Read-Only Data cache je paměť, která se vyskytuje až od compute capability 3.5. Tato paměť je umístěna na multiprocesoru, takže přístup k ní je velmi rychlý. Její výhoda oproti paměti konstant je, že nemusí být alokována před spuštěním kernelu, ale data do ní mohou být přiřazeny až za běhu. Kompilátor se sám snaží určit, které proměnné jsou neměnné a budou využívat tuto cache. Využití této cache lze vynutit i v kódu pomocí klíčových slov či speciální funkce. Velikost Read-Only Data cache je v řádu desítek kB (pro 3.5 je to 48 kB).

3.1.2.8 L1 a L2 cache

L1 a L2 cache slouží pro cachování globální paměti, tím pomáhají zrychlit přístup k ní. L1 cache je umístěna v multiprocesoru. Je to stejná část paměti jako sdílená paměť. Rozložení paměti mezi sdílenou a L1 cache lze nastavit - na výběr jsou 16/32/48 kB. Maximální velikost L1 cache je 48 kB. L2 cache je sdílená pro celé GPU.

3.1.3 Synchronizace

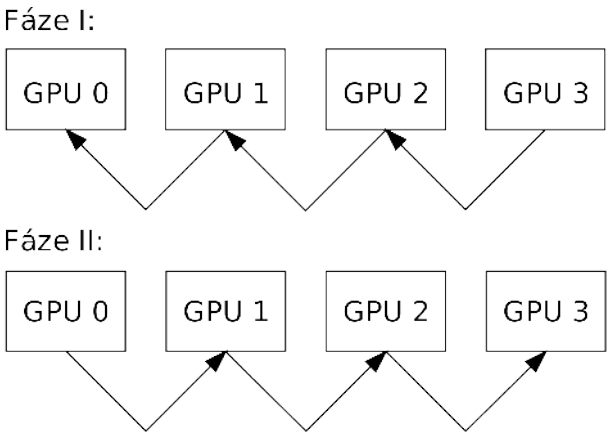
Jako u každého paralelního výpočtu i v CUDA je správná synchronizace nutností. V CUDA lze jednoduše synchronizovat vlákna v rámci bloku pomocí zavolání speciální funkce nazvané *___syncthreads()*. Aby tato synchronizace fungovala, je potřeba, aby tuto funkci

zavolala všechna vlákna v bloku. Je tedy potřeba dbát na to, aby se kód nerozvětvil a *___syncthreads()* nevolala jenom nějaká vlákna. Jelikož se pro čtení z globální paměti využívají L1 a L2 cache, kde mohou být nacachovaná některá data, je v případě, že je potřeba přistupovat ke stejné paměti z více bloků, nutné vynutit zapsání do globální paměti. Toto se provádí pomocí zavolání další speciální funkce *___threadfence()*. Tato funkce vynutí zápis do globální paměti, takže zapíše data z L1, resp. L2 cache. Další možností synchronizace jsou atomické proměnné, díky kterým lze přistupovat k jedné části paměti z více vláken zároveň. Pomocí atomických funkcí lze také v CUDA vytvořit mutex. Protože je volání kernelu asynchronní (nečeká se na jeho dokončení), je ho potřeba také synchronizovat s hostem, resp. počkat na jeho dokončení. Toto se provádí pomocí zavolání funkce *cudaDeviceSynchronize()* z hostu.

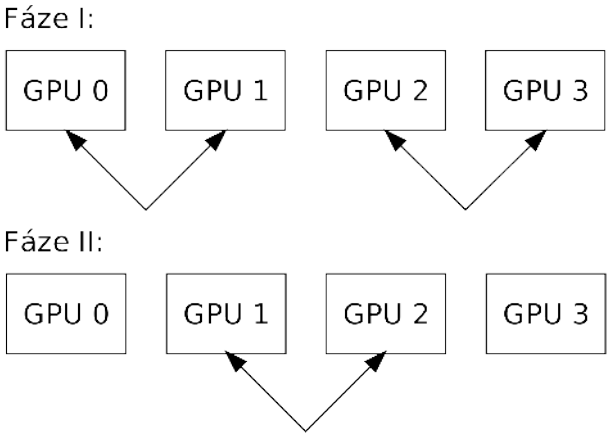
3.1.4 Komunikace více grafických karet

CUDA umožňuje využít více grafických karet, aby se urychlil výpočet. Nastávají zde však další problémy - převážně s výměnou dat a synchronizací mezi GPU. Existuje několik možností, jak problém s výměnnou dat a synchronizací vyřešit. První možností je, že kernely poběží na každém GPU nezávisle na sobě a po nějakém čase se výpočet přeruší. Na hostu se provede potřebná výměna dat mezi GPU a znovu se výpočet spustí. Další možnost přináší využití technologií Unified Virtual Adressing (dále jen UVA) a Peer-to-Peer (dále jen P2P). UVA usnadňuje práci s pamětí. Paměť hosta a více GPU se tváří pro programátora jako jedna paměť. To znamená, že z ukazatele lze poznat, kam ukazatel směřuje (na jaké zařízení nebo hosta). P2P je způsob, jak přistupovat z jednoho GPU do paměti druhého GPU přímo nebo kopírovat mezi GPU přímo bez potřeby využití paměti hosta. Díky tomu je kopírování mezi GPU velmi rychlé a není potřeba definovat, kde je paměť, na kterou ukazatel odkazuje fyzicky přítomna. Aby P2P fungoval, je potřeba, aby bylo před použitím P2P aktivováno pomocí funkce *cudaDeviceEnablePeerAccess()*. Dále je potřeba, aby aplikace byla 64bitová a zařízení byla na stejném řadiči.

Pro výměnu mezi GPU se nejčastěji používají dva modely [13] výměny. Tyto modely zohledňují zapojení grafických karet a snaží se využít co nejrychlejší komunikaci skrz nejrychlejší spojení. První model je tzv. Left-Right, který má dvě fáze. Zařízení si v první fázi mezi sebou vyměňují nejdříve data doleva, v druhé fázi se provádí výměna doprava. Tyto fáze se střídají. Tento model je zobrazen na obrázku 3.3. Druhý model se nazývá párový. Také tento model je rozdělen na dvě fáze. V první fázi mezi sebou komunikuje každé sudé zařízení s nejbližším dalším lichým zařízením (0-1; 2-3; ...). V druhé fázi mezi sebou komunikuje každé liché zařízení s nejbližším dalším sudým zařízení (1-2; 3-4; ...). Tento model je zobrazen na obrázku 3.4.



Obrázek 3.3: Left-Right model komunikace



Obrázek 3.4: Párový model komunikace

Kapitola 4

RCPSP

Resource Constrained Project Scheduling Problem je NP-těžký problém [2]. Je to problém, při němž je třeba rozvrhnout projekt, který se skládá z množiny aktivit $J = \{0, 1, \ldots, n+1\}$ ($n \in \mathbb{Z}^+$) s $n+2$ aktivitami a množiny zdrojů $K = \{1, \ldots, q\}$ ($q \in \mathbb{Z}^+$) s q různými zdroji, které aktivity vyžadují. Zdroje mají omezenou kapacitu R_k (kde $R_k \in \mathbb{Z}^+$, $k \in K$), která je po celou dobu trvání projektu konstantní a v každé časové jednotce se jejich kapacita obnovuje. Každá aktivita má danou dobu trvání aktivity d_j ($d_j \in \mathbb{Z}_0^+$, $j \in J$), která je nepreemtivní (nelze ji přerušit a opětovně naplánovat). Dále má aktivita množinu předchůdců P_j ($j \in J$) a požadavky na zdroje $r_{j,k}$. Aktivita 0 je počáteční aktivita a $n+1$ koncová, tyto aktivity mají nulovou délku ($d_0 = d_{n+1} = 0$) a nemají žádné požadavky na zdroje ($r_{0,k} = r_{n+1,k} = 0; \forall k \in K$), pouze celý rozvrh uzavírají. Vztah aktivit mezi sebou tvoří orientovaný graf $G(J, A)$, kde J je množina aktivit, která tvoří vrcholy grafu, a $A = \{P_0, P_1, \ldots P_{n+1}\}$ je množina předchůdců vrcholů (aktivit), která tvoří orientované hrany. Ukázka této sítě je na obrázku 4.1.

Řešení projektu můžeme zapsat jako vektor $(S_0, S_1, \ldots, S_{n+1})$, kde S_j ($j \in J; S_j \geq 0$) je počáteční čas naplánované aktivity. Řešení lze také zapsat jako vektor konečných časů aktivit $(F_0, F_1, \ldots, F_{n+1})$, kde F_j ($j \in J; F_j \geq 0$) je konečný čas naplánované aktivity, to jest $F_j = S_j + d_j, j \in J$.

To, že je aktivita j naplánována v čase t ($t \in \mathbb{Z}_0^+$), budeme značit jako $A(t)$, kde $A(t) = \{j \in J \mid S_j \leq t < F_j\}$. Aby mohla být aktivita j naplánována, musí být splněny dvě podmínky:

1. Všichni předchůdci aktivity j, tj. množina P_j, již musí být skončeni.
   Tedy musí platit: $F_l \leq S_j; \forall l \in P_j; \forall j, l \in J$

2. Na všech zdrojích, které aktivita požaduje ($r_{j,k}$), musí být místo po celou dobu trvání aktivity, tj. součet aktuálního čerpání zdroje a požadavek aktivity na zdroj $r_{j,k}$ musí být menší než jeho kapacita R_k.
   Musí tedy platit: $\sum_{q \in A(t)} r_{q,k} + r_{j,k} \leq R_k$, kde $j \in J; k \in K; \forall t \in \langle S_j; F_j)$

Úkolem je nalézt co nejkratší rozvrh neboli makespan. Lze tedy říci, že cílem je nalézt takový rozvrh, kde aktivita $n+1$ má minimální konečný čas, tj. hledá se $\min F_{n+1}$. RCPSP je formálně označen jako $m, 1|cpm|C_{max}$.

4.1 Matematický model

$$\min F_{n+1} \tag{4.1}$$

$$F_l \leq S_j \qquad \forall l \in P_j; \forall j, l \in J \tag{4.2}$$

$$\sum_{q \in A(t)} r_{q,k} \leq R_k \qquad \forall k \in K; \forall t \in \mathbb{Z}_0^+ \tag{4.3}$$

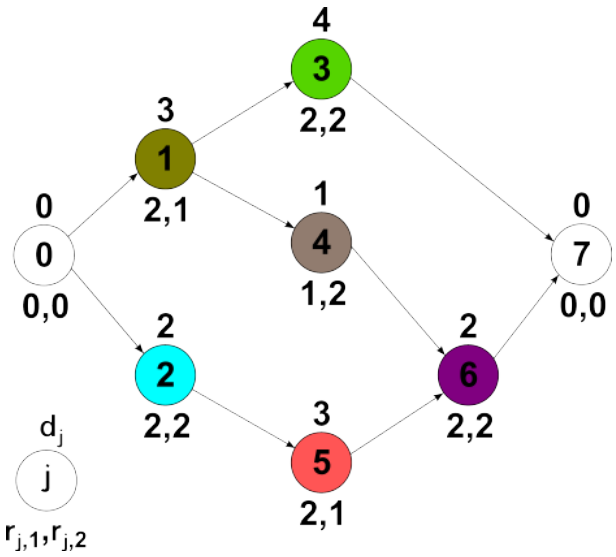
4.2 Další vlastnosti rozvrhu

Každý rozvrh má svou kritickou cestu (CPM). Je to nejkratší cesta v rozvrhu v případě, že se neberou v úvahu kapacity zdrojů, neboli zdroje mají neomezenou kapacitu. Délka kritické cesty říká, že délka optimálního řešení rozvrhu musí být stejná nebo větší než délka kritické cesty ($|CPM| \leq makespan$).

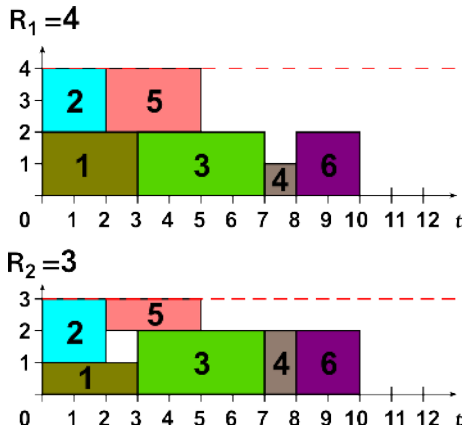
U každé instance lze spočítat také její nejdelší možnou délku výsledného rozvrhu. Jednou z možností, jak nalézt nejdelší délku rozvrhu je sečíst délky trvání všech aktivit, tedy $\sum_J d_j$.

4.3 Ukázka rozvrhu

Na obrázku 4.1 je zobrazena síť projektu. V této síti jsou zaznamenány jak délka trvání aktivity, tak její požadavky na zdroje. Tento projekt má $n = 6$ a množinu aktivit $J = \{0, 1, 2, 3, 4, 5, 6, 7\}$. Dále má projekt dva zdroje $K = \{1, 2\}$. Kapacity zdrojů jsou $R_1 = 4$ a $R_2 = 3$. Jedno z mnoha možných řešení tohoto projektu je zobrazeno na obrázku 4.2. Výsledný vektor tohoto rozvrhu s počátečními časy aktivit (S_j) je $(0, 0, 0, 3, 7, 2, 8, 10)$ nebo vektor s konečnými časy (F_j) je $(0, 3, 2, 7, 8, 5, 10, 10)$. Makespan tohoto řešení je tedy $makespan = S_{n+1} = F_{n+1} = 10$.



Obrázek 4.1: Ukázka sítě projektu



Obrázek 4.2: Možné řešení projektu z obrázku 4.1

Kapitola 5

Genetický algoritmus

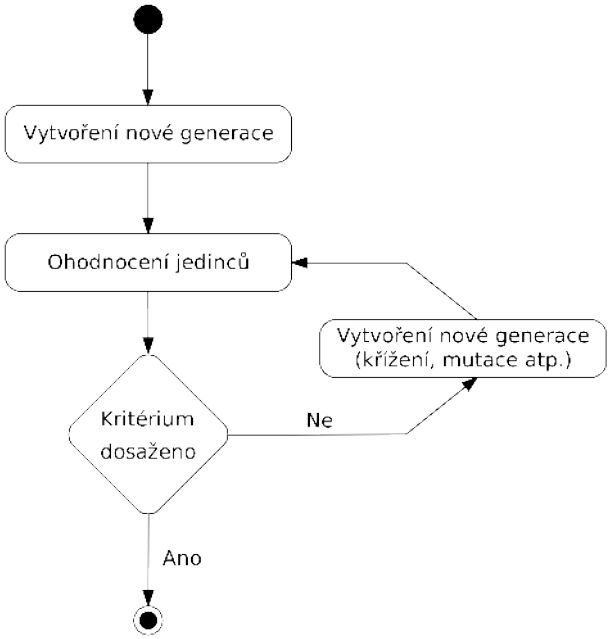
Genetický algoritmus je způsob, který je inspirovaný skutečnou genetikou. Také terminologie odpovídá té, která se používá v genetice. Je dána množina jedinců, která se nazývá generace. Každý jedinec neboli chromozom je reprezentován geny, které ho určují.

Snahou genetického algoritmu je vylepšovat jedince v každé generaci tak, aby bylo po určitém počtu generací získáno co nejlepší řešení. Diagram algoritmu je zobrazen na obrázku 5.1. Sestavení nové generace, resp. vylepšení jedinců, může probíhat několika způsoby. Nejjednodušší způsob je zkopírování nejlepších jedinců do další generace. Další způsob je zkřížení dvou či více jedinců z generace a tím vytvořit nového jedince. Velmi používaným způsobem je také mutace jedince, kdy jedinec zmutuje sám. Aby genetický algoritmus neuvízl v lokálním maximu (resp. minimu), je dobré v každé generaci vygenerovat nějaké nové jedince, čímž se zajistí, že se bude generace dále diverzifikovat.

Každý jedinec genetického algoritmu reprezentuje jedno řešení problému a je tedy nutno tento problém nějakým způsobem převést na geny, které budou reprezentovat jedince. Jsou dva způsoby reprezentace řešení problému. První reprezentace je přímá, kdy každý gen jedince reprezentuje přímo řešení problému. Dalším způsobem je nepřímá reprezentace, kdy se využívá funkce, která dokáže z chromozomu jedince získat.

Důležitou součástí genetického algoritmu je ohodnocovací funkce, která dokáže určit, který jedinec je dobrý a který ne. Tato funkce poskytuje informace nejen o tom, jak je řešení kvalitní, ale i jaké má jedinec předpoklady pro další křížení, resp. mutaci. Z toho vyplývá, že aby mohl být využit genetický algoritmus pro nějaký problém, je potřeba, aby tento problém i jeho řešení bylo možno ohodnotit.

Dobré nastavení genetického algoritmu je pro každý problém trochu jiné. Hlavními faktory nastavení genetického algoritmu jsou velikost populace a počet generací. Malá populace nezajistí dostatečnou diverzifikaci jedinců. Naproti tomu ohodnocování velké populace je velmi náročné a nedokáže zajistit značné zlepšení vůči času běhu. U počtu generací je třeba zjistit vrchol, kdy je pravděpodobnost zlepšení již malá. Tyto dva faktory lze nastavit dle testů a z nich dosažených výsledků. Nastavení faktorů se také odvíjí od toho, zda je cílem nalézt průměrné řešení problému za krátký čas, nebo nalézt co nejlepší řešení a na čase tolik nezáleží.



Obrázek 5.1: Genetický algoritmus

5.1 Paralelní genetický algoritmus

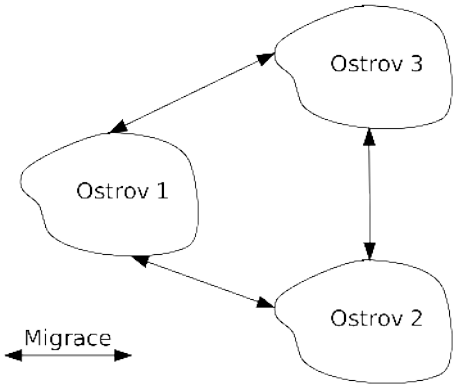
Genetický algoritmus lze paralelizovat různými způsoby. Nejjednodušším způsobem paralelizace je paralelně ohodnocovat jedince ohodnocovací funkcí a paralelně vytvářet novou generaci, tj. křížení, mutace atp. Ohodnocení jednoho nebo skupiny jedinců probíhá v samostatném vlákně.

5.1.1 Ostrovní genetický algoritmus

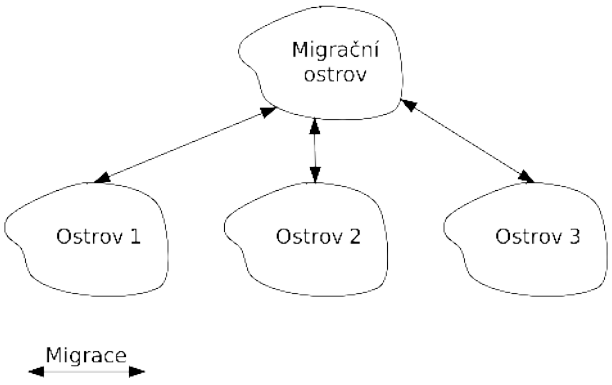
Jednoduchá paralelizace popsaná výše naráží na hranice, že velmi velká populace již nemusí zajistit velké zlepšení. Proto je dobré velkou generaci rozdělit na části, tzv. ostrovy. Tento způsob je inspirován opět genetikou, resp. přírodou. Populace se mohou na ostrově nezávisle na sobě vyvíjet, resp. iterovat. Každý ostrov se tedy může vyvíjet jinak než ostatní ostrovy. Aby se řešení ještě zlepšilo, provádí se jednou za čas tzv. migrace. Migrace může probíhat dvěma způsoby, buď přímo nebo skrz migrační ostrov. Pokud je migrace prováděna přímo, ostrovy si mezi sebou vyměňují jedince přímo. Tento způsob je zobrazen na obrázku 5.2.

Naproti tomu migrace skrz migrační ostrov migruje část populace z ostrova na migrační ostrov. Naopak z migračního ostrova migrují na ostrov jedinci, kteří tam již jsou. Tento způsob je zobrazen na obrázku 5.3.

Ostrovní genetický algoritmus je velmi vhodný pro použití na grafických kartách s CUDA, je patrné, že ostrovy mohou být bloky a vlákna mohou zpracovávat jedince.



Obrázek 5.2: Přímá komunikace ostrovů



Obrázek 5.3: Komunikace pomocí migračního ostrova

Kapitola 6

Návrh genetického algoritmu řešící RCPSP problém

Celý návrh genetického algoritmu pro řešení RCPSP lze rozdělit na dvě části. První část je definování použití genetického algoritmu. Aby genetický algoritmus fungoval, je třeba definovat jedince, kteří budou tvořit populaci. Každý jedinec bude jedno možné řešení instance problému RCPSP. Dále je potřeba definovat křížení a mutaci jedinců v populaci. Nakonec je potřeba navrhnout paralelizaci genetického algoritmu. Druhá část návrhu je získání, zpracování a zlepšení jedince, tedy možné řešení instance problému RCPSP.

Celý algoritmus tedy nejprve vytvoří populaci jedinců. Poté je z každého z chromozomu jedince sestaven rozvrh. Následuje pokus o zlepšení rozvrhu. Po vylepšení rozvrhu je následně přetvořen chromozom tak, aby geny odpovídaly vylepšenému rozvrhu. Tato metoda se nazývá chromozom adjustment. Dále již přichází na řadu proces genetického algoritmu, který vytvoří novou generaci. S novou generací se celý proces opakuje od začátku. Takto se pokračuje, dokud není dosaženo ukončujícího kritéria. Celý proces algoritmu je zobrazen na obrázku 6.1.

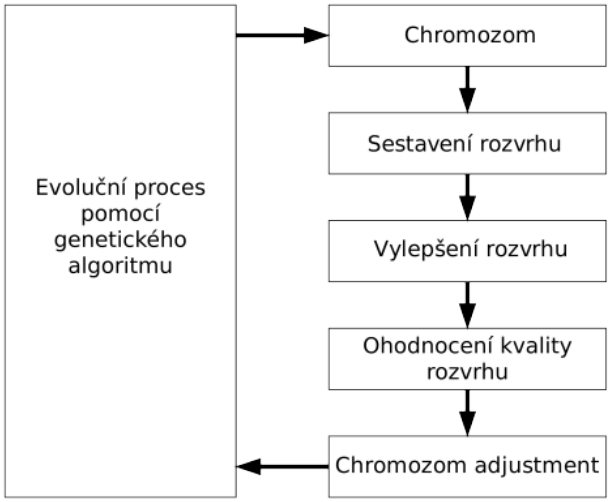
6.1 Reprezentace jedince

Každý jedinec v genetickém algoritmu představuje jedno možné řešení instance problému RCPSP. Reprezentace je přímá, a to pomocí tzv. priorit, které jsou pak použity přímo k sestavení rozvrhu. Každá aktivita tak představuje jeden gen chromozomu. Pokud tedy má instance n aktivit, má chromozom $n + 2$ (n aktivit + počáteční + koncová aktivita) genů.

$$chromozom = \{gen_0,\ gen_1,\ \ldots,\ gen_{n+1}\} \tag{6.1}$$

6.2 Sestavení rozvrhu

Sestavení rozvrhu probíhá ve dvou fázích. V první fázi se sestaví pořadí, v jakém budou aktivity postupně plánovány. Druhá fáze je plánování aktivit.



Obrázek 6.1: Diagram algoritmu

6.2.1 Activity list

Aby mohl být z chromozomu sestaven rozvrh, je potřeba nejprve sestavit tzv. activity list (AL). Tento list je seřazení aktivit v pořadí, v jakém budou naplánovány. Pořadí aktivit je určeno dvěma pravidly. První pravidlo, které určuje pořadí, je, že všichni předchůdci aktivity již musí být v activity listu. Druhým pravidlem řazení je právě priorita, kterou je gen chromozomu. Pseudokód sestavení activity listu je uveden v Algoritmus 1.

Algoritmus 1 Sestavení activity listu

Input: P - pole s množinami předchůdců aktivit
Input: J - set aktivit
Input: C - pole s prioritami aktivit (chromozom)
Output: AL - activity list

1: $AL \leftarrow \{\}$ //aktivity list
2: $freeActivities \leftarrow \{0\}$ //set aktivit k přidání do AL
3: **while** not empty *freeActivities* **do**
4: $i \leftarrow aktivita \in freeActivities\, s \max C_{aktivita}$
5: přidat i do AL
6: odeber i z *freeActivities*
7: **for all** $ac \in J$ AND $ac \notin freeActivities$ AND $ac \notin AL$ **do**
8: **if** $\forall aktivity \in P_{ac}$ již jsou v AL **then**
9: přidej ac do *freeActivities*
10: **end if**
11: **end for**
12: **end while**

6.2.2 SGS

Pro sestavení rozvrhu z activity listu se používá algoritmus SGS (Schedule generation schemes). Tento algoritmus existuje ve dvou verzích. U první verze nazvané paralelní SGS je dokázáno [8], že v některých případech nedokáže nalézt optimální řešení, je tedy pro sestavování rozvrhů nevhodná. I proto je zvolena druhá verze nazvaná serial SGS.

Jak již název Schedule generation schemes napovídá, postupně se za sebou všechny aktivity naplánují. Pořadí aktivit, v jakém jsou naplánovány, je určeno podle activity listu. Sestavení rozvrhu funguje na principu generací (g), kde se v každé generaci vybere a naplánuje jedna úloha. Pro sestavení rozvrhu je tedy třeba stejný počet generací jako počet aktivit. V každé generaci je potřeba vzít další aktivitu j z activity listu. Protože pořadí activity listu respektuje precedence aktivity, nemůže nastat problém s tím, že bude plánována aktivita, u které ještě nejsou naplánovaní její předchůdci. Následuje spočítání nejdřívějšího konečného času aktivity j označeného EF_j. Nejdřívější konečný čas se spočítá tak, že se berou v úvahu pouze předci a zanedbává se potřeba zdrojů. To lze provést jednoduchým zjištěním, který předek má největší konečný čas F_i, a přičtením doby trvání d_j plánované aktivity j k tomuto času. Nakonec se určí konečný čas F_j úlohy j a to tak, že se v intervalu $\langle EF_j - d_j, LS - d_j \rangle$, kde LS je největší možná délka rozvrhu spočítaná jako součet všech délek aktivit, nalezne nejnižší čas, od něhož lze aktivitu umístit na zdroje, které vyžaduje. Časová složitost algoritmu uvedeného v Algoritmus 2 je $O(Kn^2)$, kde K je počet zdrojů a n počet aktivit.

Algoritmus 2 Sestavení rozvrhu serial SGS

Input: P - pole s množinami předchůdců aktivit
Input: J - set aktivit
Input: AL - activity list
Output: F - pole s konečnými časy naplánovaných aktivit

1: $F_0 \leftarrow 0$ //Naplánování počáteční aktivity
2: **for** $(g = 1; g < n + 2; ++g)$ **do**
3: $j \leftarrow AL_g$ //Získání další aktivit v pořadí z activity listu
// Spočítej nejdřívější možný konečný čas j v závislosti pouze na precedencí
4: $EF_j \leftarrow maxF_i \mid i \in P_j$
// Spočítej nejdřívější možný konečný čas j v závislosti precedencí a kapacitou zdrojů
5: $F_j \leftarrow \min t \in \langle EF_j - d_j, LS - d_j \rangle + d_j$, kde musí být volno pro aktivitu j na všech zdrojích, které potřebuje v intervalu $\langle t, t + d_j \rangle$
6: **end for**

6.3 Vylepšení rozvrhu

Vylepšení rozvrhu probíhá třemi různými způsoby. Způsob použití těchto technik se liší. První způsob, nazvaný Forward-Backward Improvement, je použit vždy po sestavení rozvrhu. Druhý způsob, Hidden order jumping, využívá historie již sestavených rozvrhů, resp. vztahů mezi aktivitami. Tento způsob se používá na náhodně vybrané jedince. Třetí způsob, nazvaný Walking, se používá pouze u náhodně vybraných nejlepších jedinců.

6.3.1 Forward-Backward Improvement

Forward-Backward Improvement(dále jen FBI) je známý způsob, jak snadno vylepšit rozvrh [10]. Jak již název napovídá, má tato metoda dvě fáze - dopřednou a zpětnou. Aby tuto techniku bylo možno použít, je nejdříve potřeba sestavit rozvrh. Na takto sestavený rozvrh již je možno aplikovat backward fázi algoritmu.

Backward fáze je plánování rozvrhu v opačném směru od ukončující aktivity. Backward fáze se provede tak, že se otočí hrany v projektové síti, tedy z předchůdců se stanou následovníci. Jako priority pro aktivity se použijí konečné časy aktivit F_j v původním rozvrhu. Z priorit se sestaví activity list a poté následuje SGS s projektovou sítí s otočenými hranami. SGS začíná v koncovém čase původního rozvrhu a od tohoto času se délka trvání aktivit d_j odečítá. Tím se zajistí, že aktivity budou „zarovnány“ doprava k makespanu původního rozvrhu.

Po backward fázi následuje druhá fáze forward. V této fázi se použije stejná projektová síť jako u výchozího rozvrhu a začíná se také plánovat od počáteční aktivity. Tato fáze je totožná s technikou sestavení původního rozvrhu s tím rozdílem, že jako priority pro activity list slouží počáteční čas aktivit z rozvrhu sestaveného backward fází. Nyní se může opět opakovat backward fáze s prioritami aktivit z konečných časů aktivit z předcházející forward fáze. Střídání fází může probíhat tak dlouho, dokud buď nenastane maximální počet fází, nebo se rozvrh nepřestane zlepšovat. Toto přeuspořádávání aktivit pomocí FBI může délku rozvrhu zmenšit avšak nemůže se stát, že se délka rozvrhu zvětší.

6.3.2 Hidden order jumping

Hidden order jumping je metoda, která využívá historii získanou z předchozího sestavení rozvrhů [11]. Pokaždé, když se sestavuje rozvrh, je do matice historie zapsáno, zda je aktivita i v activity listu před aktivitou j. Pokud je aktivita i v AL před j, přičítá se k $X_{i,j}$ jedna. Jestliže tomu tak není, nepřičítá se nic.

Metoda Hidden order jumping je vylepšená metoda sestavení AL, kde se místo priorit aktivit využívá právě matice historie $X_{i,j}$ vztahů aktivit. Začíná se stejně, tj. vybere se počáteční aktivita a ta se umístí na první místo AL. Poté se do množiny aktivit U, které je možno zařadit do AL, přidají ostatní aktivity, které mají již všechny své předchůdce v AL a sami v AL nejsou. Rozdíl je ve výběru aktivity, která bude zařazena do AL. Místo použití priorit aktivit se používá váha aktivity w_j vůči ostatním aktivitám, které jsou v množině U. Váha w_j se spočítá jako $w_j = \prod_{i \in (U-j)} X_{j,i}/s$, kde s je počet AL použitých k získání historie. Aktivita je poté náhodně vybrána proporcionálně vůči váze w_j. Pseudokód Hidden order jumping je zapsán v Algoritmus 3.

6.3.3 Walking

Walking [11] je metoda, která se snaží vylepšit pouze nejlepší jedince v populaci. Snahou této metody je přeuspořádat aktivity v AL a ověřit, zda přesunutí aktivity vedlo ke zlepšení řešení. To se provádí tak, že se N_{walk}-krát vybere náhodně aktivita j, která bude posunuta náhodně v AL více dopředu, maximálně však za posledního svého předchůdce $p \in P_j$. Tímto posunem se pravidla AL neporuší. Poté se sestaví rozvrh z modifikovaného AL. Pokud se

Algoritmus 3 Hidden order jumping

Input: P - pole s množinami předchůdců aktivit
Input: J - set aktivit
Input: X - matice s historií vztahů
Output: AL - activity list

1: $AL \leftarrow \{\}$ //aktivity list
2: $U \leftarrow \{0\}$ //set aktivit k přidání do AL
3: **while** not empty U **do**
4: $w \leftarrow \{\}$ //list s váhami aktivit vůči množině U
5: **for all do**$j \in U$
6: $w_j \leftarrow \prod_{i \in (U-j)} X_{j,i}/s$
7: **end for**
8: $i \leftarrow$ náhodná aktivita proporcionálně vůči váze w_i
9: přidat i do AL
10: odeber i z U
11: **for all** $ac \in J$ AND $ac \notin U$ AND $ac \notin AL$ **do**
12: **if** $\forall aktivity \in P_{ac}$ již jsou v AL **then**
13: přidej ac do U
14: **end if**
15: **end for**
16: **end while**

výsledný rozvrh zlepšil, pokračuje se dále. Pokud nastalo zhoršení, vrátí se aktivita na své původní místo v AL. Pseudokód je uveden v Algoritmus 4.

6.4 Chromosome adjustment

Již vylepšený rozvrh nemusí odpovídat chromozomu, ze kterého byl původní rozvrh sestaven. Pokud by chromozom neodpovídal rozvrhu po zlepšení, není zaručeno, že se z původního chromozomu povede sestavit stejný vylepšený rozvrh. Proto se používá metoda nazvaná Chromosome adjustment [12]. Chromosome adjustment se provádí tak, že se v prvním kroku seřadí priority z původního chromozomu od nejvyšší priority po nejnižší. Poté se získá activity list z vylepšeného rozvrhu. Poté se postupně přiřazují priority od nejvyšší po nejnižší a ty se zapisují na místo indexu aktuální aktivity z AL. Pseudokód Chromosome adjustment je uveden v Algoritmus 5.

6.5 Genetický algoritmus

Základ celého návrhu je tzv. random-key genetický algoritmus. Každý jedinec představuje jedno možné řešení instance problému RCPSP. Tento algoritmus lze ve stručnosti popsat následovně:

1. Vytvoření populace tvořené chromozomy, které představují jedince, resp. řešení

Algoritmus 4 Walking

Input: P - pole s množinami předchůdců aktivit
Input: J - set aktivit
Input: AL - activity list
Output: AL - activity list

1: $best \leftarrow SGS(AL)$ //Počáteční makespan, funkce SGS vrací makespan
2: **for** $(n = 1; n < N_{walk}; ++n)$ **do**
3: $k \leftarrow$ random $aktivita \in J$
4: $p \leftarrow$ poslední $aktivita \in P_k$ v AL
5: $end \leftarrow$ pozice k v $AL - pozice\ p$ v AL
6: $move \leftarrow random(1; end)$
7: posuň k v AL doleva o $move$
8: **if** $best <= SGS(AL)$ **then**
9: posuň k v AL doprava o $move$
10: **end if**
11: **end for**

Algoritmus 5 Chromosome adjustment

Input: Chromosome - chromozom
Input: AL - activity list posledního vylepšeného rozvrhu
Output: AdjustedChromosome - chromozome

1: $sortedPriority \leftarrow$ seřaď $Chromosome$ od nejvyšší po nejnižší priority
2: **while** not empty AL **do**
3: $j \leftarrow$ odeber další aktivitu z AL
4: $p \leftarrow$ odeber další prioritu ze $sortedPriority$
5: $Chromosome_j \leftarrow p$
6: **end while**

2. Vyhodnocení populace
3. Z předchozí populace vytvořit novou populaci a opět opakovat 2. krok, dokud nenastane ukončující podmínka

Random-key genetický algoritmus má výhodu v tom, že reprezentace jedinců je pouze list čísel. Vytvoření počáteční populace může tedy proběhnout pomocí vygenerování náhodných čísel. Díky tomu lze na začátku získat dostatečně různorodé jedince.

6.5.1 Ohodnocovací funkce

Jako nejlepší ohodnocovací funkce genetického algoritmu se jeví makespan výsledného rozvrhu jedince. Další možností je využití tzv. modified makespan [12]. Modified makespan dokáže k makespan přidat hodnotu v intervalu $(0; 1)$ a tím od sebe rozlišit jedince, kteří mají stejný makespan. Modified makespan zavádí funkci vzdálenosti $dist(j)$ aktivity j k poslední aktivitě $n + 1$. Tato vzdálenost je určena jako počet aktivit na cestě od aktivity j k poslední aktivitě $n + 1$, včetně úlohy j a bez úlohy $n + 1$. Na cestě jsou vybírány vždy úlohy s nejmenší vzdáleností. Hodnoty funkce $dist$ jsou neměnné, lze je tedy spočítat pouze jednou (na začátku běhu celého algoritmu). Výpočet této modified makespan se vzdáleností L, což je maximální vzdálenost aktivit využitých k výpočtu, se provádí dle následujícího vzorce [12]:

$$modified\,makespan = F_{n+1} + \frac{\sum_{d=1}^{L} \sum_{i|dist(i)=d} F_i}{\sum_{d=1}^{L} \sum_{i|dist(i)=d} F_{n+1}} \tag{6.2}$$

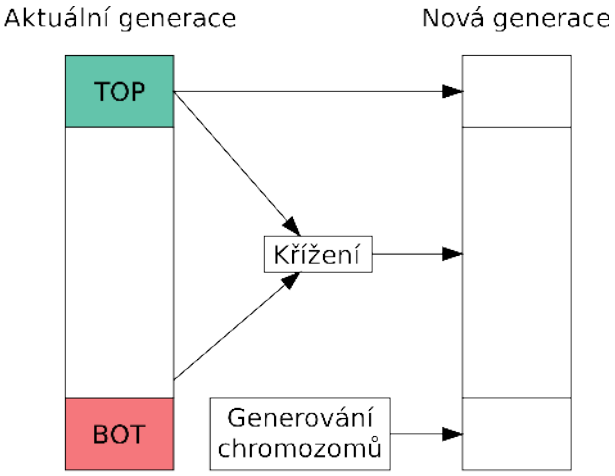
Druhou možností, jak od sebe rozlišit dva jedince se stejným makespanem, je jejich index, resp. pořadí v populaci.

6.6 Rozdělení populace

Před vytvářením nové generace je potřeba celou populaci současné generace seřadit podle ohodnocení. Dle parametrů genetického algoritmu se stanoví, kolik jedinců z populace bude patřit do TOP, tj. nejlepší jedinci, a kolik jedinců bude náležet BOT, tj. nejhorší jedinci. Takto seřazená a rozdělená populace slouží jako základ nové generace. Jedinci z TOP budou do nové generace zkopírováni. Jedinci z BOT budou mutovat. Jedinci, kteří nejsou zařazeni v TOP ani BOT, budou sloužit jako základ pro křížení. Obrázek vytvoření nové generace je zobrazen na obrázku 6.2.

6.6.1 Křížení

Křížení dvou jedinců probíhá pomocí náhodných čísel. Křížení je prováděno spojením dvou jedinců (rodičů). První rodič je jeden z nejlepších jedinců v populaci (z TOP) a druhý rodič je náhodný jedinec z celé populace. Křížení je realizováno tak, že se pro každý gen



Obrázek 6.2: Vytváření nové generace

vygeneruje náhodné číslo v intervalu $(0; 1)$, pokud toto číslo bude menší nebo rovno než stanovená hranice, bude gen potomka na i-tém místě stejný jako gen na i-tém místě prvního rodiče (z TOP), pokud bude číslo větší, bude použit i-tý gen druhého rodiče. Ukázka křížení je znázorněna v tabulce 6.1.

Gen	1	2	3
První rodič (z TOP)	**0,3**	**0,8**	**0,5**
Druhý rodič (z celé populace)	**0,1**	**0,2**	**0,7**
Náhodné číslo	**0,8**	**0,1**	**0,5**
Porovnání s hranicí 0,7	>	<	<
Nový chromozom po křížení	**0,1**	**0,8**	**0,5**

Tabulka 6.1: Křížení jedinců

6.6.2 Mutace

Mutace jedinců může probíhat několika způsoby. První způsob je vygenerování náhodných čísel, které vytvoří nového jedince stejně jako na začátku celého genetického algoritmu. Druhý způsob je inspirován křížením, kdy se vygeneruje nový jedinec, jenž se poté zkříží s jedincem, který bude mutovat. Obě tato řešení si jsou velmi podobná. Pokud se vytvoří nový jedinec, je pravděpodobné, že v další generaci bude zkřížen s jiným jedincem.

6.6.3 Ukončující kritéria

Ukončující podmínka genetického algoritmu řešícího problém RCPSP je předem daný počet generací, získaných z experimentů s implementací. Další ukončující kritérium je dosažení kritické cesty, protože po dosažení kritické cesty již nemůže být nalezen jedinec s lepším makespan.

Kapitola 7

Implementace pro CPU

Implementace pro CPU je provedena pomocí jazyka C++. Tato implementace je multiplatformní (testováno MS Windows a GNU/Linux). Implementace využívá kód z diplomové práce L. Bukaty [3]. Z tohoto kódu jsou použity funkce načtení instance, spočítání lower bound a vypsání výsledku. Implementace odpovídá návrhu algoritmu popsaného v předchozí kapitole 6. Při implementaci bylo použito rozšíření jazyka C++ C++11.

7.1 Generování náhodných čísel

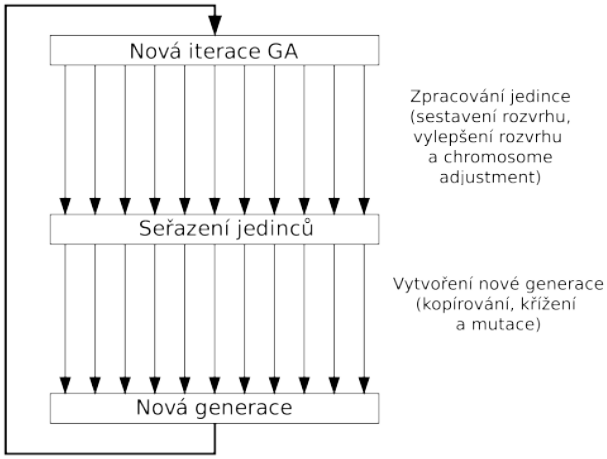
Generování náhodných čísel probíhá pomocí standardní knihovny z rozšíření jazyka C++ C++11. Jako generátor byl zvolen default_random_engine. Implementace tohoto generátoru je zvolena knihovnou. Jako semínko tohoto generátoru je zvolen aktuální čas. Pro distribuci čísel z generátoru jsou použity uniformní distribuce.

7.2 Implementace paralelizace

Paralelizace je provedena za pomoci OpenMP [16]. OpenMP je specifikace pro překladače a knihovna pro vícevláknové programování. Podpora OpenMP je v současné době v překladačích GCC a Visual C++, plánovaná je i podpora v LLVM [17]. K použití OpenMP se používají tzv. direktiva. Díky tomu lze implementovat paralelní verzi, která může fungovat jako sériová v případě, že překladač nepodporuje OpenMP. OpenMP direktiva začínají klíčovými slovy *pragma omp*. Klíčové slovo *pragma* značí, že pokud není direktivum *omp* v překladači implementováno, může ho ignorovat. Díky tomu lze snadno paralelizovat for cyklus tím, že se před něj napíše $\#pragma\ omp\ parallel\ for$. Překladač poté provede celý for cyklus paralelně tak, že všechny iterace se spouští naráz. Synchronizace těchto vláken bude provedena na konci celého cyklu. Pokud je iterací cyklu příliš mnoho, není vytvořeno nové vlákno pro každou iteraci. Je vytvořeno pouze několik vláken, která jsou znovu použita. OpenMP také nabízí možnost, jak jednoduše pracovat se stejnou proměnnou ve více vláknech najednou. Více podrobností lze nalézt ve specifikaci OpenMP [18].

7.3 Paralelizace

Genetický algoritmus lze snadno zparalelizovat. Nejsnadněji to lze provést tak, že se výpočet každého jedince provede v samostatném vláknu. Každé vlákno tedy sestaví rozvrh, poté se ho pokusí vylepšit, následně rozvrh ohodnotí a nakonec se provede chromosome adjustment. Po této fázi je nutné synchronizovat celý výpočet a seřadit jedince podle ohodnocovací funkce. Řazení jedinců také může proběhnout pomocí některého ze známých paralelních řadících algoritmů (např. bitonic sort). Po seřazení jedinců přichází na řadu vytvoření nové generace. To lze taktéž provést paralelně, tj. každé vlákno vytvoří jedince pro další generaci buď kopírováním, křížením nebo mutací jedinců z aktuální generace. Celý tento proces je zobrazen na obrázku 7.1.



Obrázek 7.1: Paralelizace algoritmu

7.4 Sériová verze

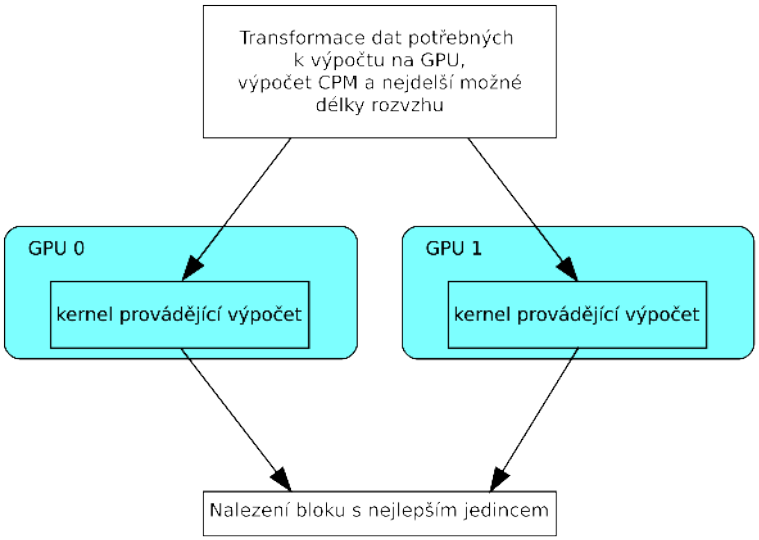
Sériová verze implementace je díky implementaci paralelizace pomocí OpenMP provedena zkompilováním kódu bez podpory OpenMP. Díky tomu proběhnou všechny části, které jinak probíhají paralelně, sériově.

Kapitola 8

Návrh a implementace GPU verze algoritmu

GPU verze vychází z CPU verze a je implementována pomocí CUDA. Implementace pro GPU, stejně jako implementace pro CPU, využívá část kódu z diplomové práce L. Bukaty [3]. Implementace je určena jak pro výpočet za pomoci více grafických karet, tak i pro výpočet na jedné grafické kartě. Z technických důvodů, pokud má být použita P2P komunikace mezi kartami, lze implementaci přeložit pouze na 64bitovém operačním systému MS Windows a GNU/Linux. Implementace je navržena pro architekturu Kepler a novější, tedy compute capability 3.0 a výše.

Výpočet na grafické kartě je homogenní a provádí ho jeden kernel. Na hostu jsou spočítány pouze délka kritické cesty, nejdelší možná délka rozvrhu a po výpočtu kernelu nalezení bloku s nejlepším jedincem. Diagram s postupem je zobrazen na obrázku 8.1.



Obrázek 8.1: Model výpočtu na GPU

8.1 Rozdíl oproti CPU

Oproti CPU bylo potřeba implementaci zjednodušit a odebrat některá vylepšení, která nelze dostatečně paralelizovat nebo která odporují návrhu CUDA. Proto ve srovnání s CPU verzí nevyužívá GPU implementace metody vylepšení rozvrhu Hidden order jumping a Walking. Aby byl kód dostatečně výkonný, je potřeba dodržet několik věcí. Nejdůležitější je využití správného druhu paměti pro data, která se pro tuto paměť hodí. Mělo by se co nejméně náhodně přistupovat ke globální paměti. Pokud se ke globální paměti přistupuje, je vhodné využívat metody coalescing. Dále je třeba zajistit dostatečné obsazení SM bloky a tím zakrýt přístupy k pomalejší globální paměti. Poté je potřeba zamezit přílišnému větvení kódu (převážně v rámci warpu). Velkým problémem v CUDA je řazení. Pro každé řazení je použit bitonic sort [4], který je nejpoužívanějším paralelním řadícím algoritmem pro CUDA.

Další nutností bylo použití jiné datové struktury než haldy, která je použita k vytvoření activity listu. To je provedeno tak, že se prvky v „haldě“ neřadí při přidání prvku, ale při získání nejlepšího prvku se projde celá „halda“ a nalezne se nejlepší prvek. Díky tomu je výrazně sníženo větvení i „náhodný“ přístup, který by probíhal při použití klasické haldy. Zároveň se tím však zhorší časová složitost „haldy“. Pro získání prvku je $O(n)$ a pro přidání prvku $O(1)$, kde n je počet prvků v haldě.

8.2 Paralelní model

Stejně jako u CPU je sestavení, vylepšení a ohodnocení jedince prováděno pomocí jednoho vlákna. Blok CUDA tedy reprezentuje jeden ostrov v paralelním genetickém algoritmu, resp. populaci tohoto ostrova. Protože není pro implementaci nikterak výhodné uspořádat vlákna, resp. bloky, do více rozměrů, je použit pouze jeden rozměr. Implementace je navržena pro spouštění 512 vláken v bloku a spuštění 4 bloků na jeden SM. Počet vláken a bloků lze samozřejmě nastavit.

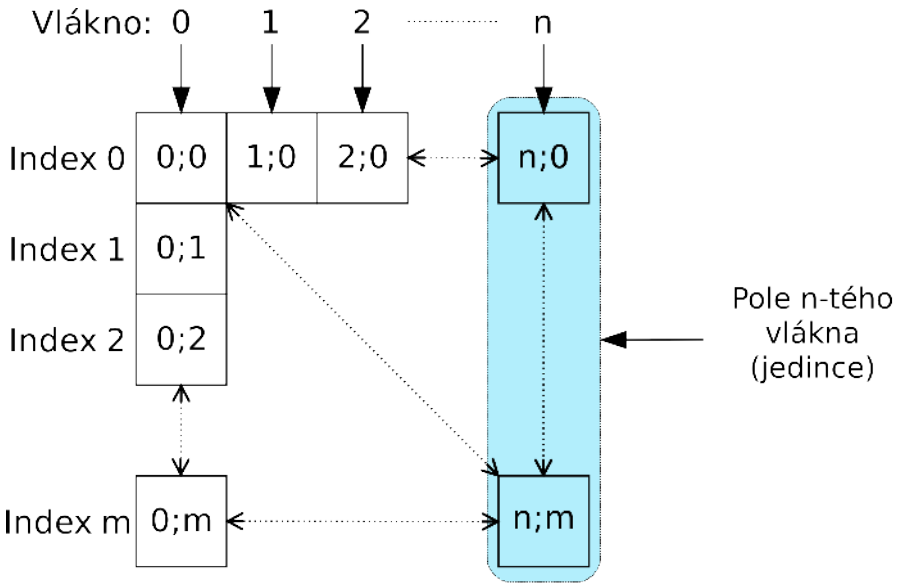
8.3 Datový model

Nejdůležitější částí implementace je využití správné paměti. Správné uspořádání a využití paměti může výpočet výrazně zrychlit. V případě nevhodného využití paměti může být výpočet výrazně zpomalen.

8.3.1 Globální paměť

Všechna pole, která jsou uložena v globální paměti CUDA, jsou vytvářena pomocí funkce cudaMallocPitch, tím je zaručeno, že budou pole správně zarovnána a k přístupu bude využit coalescing. Takovéto pole lze pomocí výpočtu indexu použít jako dvourozměrné, kde šířka pole je stejně velká jako počet vláken. Uspořádání těchto polí je zobrazeno na obrázku 8.2.

V globální paměti jsou takto uložena pole pro zaznamenání aktuálního obsazení zdrojů, konečné časy aktivit, priority, resp. chromozomy jedinců, activity list a pole s prvky haldy. Tato pole jsou příliš velká na to, aby se pro každé vlákno vešla do sdílené paměti.



Obrázek 8.2: Uspořádání polí v globální paměti

8.3.2 Sdílená paměť

Jelikož je sdílená paměť sdílená pro všechna vlákna, která jsou spuštěna na multiprocesoru, je potřeba, aby data ve sdílené paměti zabírala co nejméně místa, aby bylo možno spustit více bloků na SM a tím zakrýt přístup k pomalejší globální paměti. Je proto použit trik, kdy je stejná část paměti sdílena pro více druhů polí, která nejsou ve stejný čas využívána zároveň. Protože je velikost buňky sdílené paměti 4 B, musí být všechna pole uložená ve sdílené paměti zarovnána na 4 B. Následující data (data jsou seřazena stejně jako ve sdílené paměti):

Délka trvání aktivit, která je pro všechna data stejná, má velikost je 1 B pro každou aktivitu. Její celková velikost pro blok je tedy n bytů, kde n je počet aktivit. Tato část paměti je zarovnána na 4 B, její velikost je tedy $n + 4 - (n \mod 4)$.

Kapacita zdrojů, která má velikost 2 B pro každý zdroj. Její celková velikost je $2 * K$ bytů, kde K je počet zdrojů. Tato část paměti je zarovnána na 4 B, její velikost je tedy $2 * [K + (K \mod 2)]$ B.

Následují tři proměnné, jejichž účel je: určení začátku čtení migrace v bloku, určení začátku čtení migrace v zařízení a proměnná určující, zda je již v bloku dosaženo kritické cesty. Každá z těchto tří proměnných má velikost 4 B, dohromady tedy zabírají 12 B.

Další část paměti zabírají proměnné, které určují počet prvků v haldě s velikostí 2 B pro každé vlákno. Její velikost je tedy $2*t$ B, kde t je počet vláken v bloku. Tato část paměti je sdílena s polem, ve kterém jsou uložena pořadí seřazených jedinců. Pro každé vlákno je velikost 2 B, tedy celková velikost na blok je $2*t$ B. Dále je tato část paměti sdílená s polem, které se využívá pro seřazení aktivit, kde každá aktivita zabírá velikost 2 B. Její velikost je $2 * n$B, kde n je počet aktivit. Tato část paměti je veliká jako velikost největšího sdíleného pole, tedy $2*\max(n, t)$ B. Tato část paměti je zarovnána na 4 B. Celková velikost lze spočítat jako $2 * [\max(n, t) + (\max(n, t) \mod 2)]$ B.

Po předchozí části paměti následuje část, kde jsou uložena pole, která určují, zda je aktivita již naplánována. Každé vlákno má své pole. Každý bit tohoto pole představuje jednu aktivitu (zda je, či není naplánovaná). Proto je jeho velikost dostatečně malá na to, aby mohlo být umístěno ve sdílené paměti. Jeho velikost je $n/8$ B pro každé vlákno. Část pro každé vlákno musí být zarovnána na 4 B. Velikost pro každé vlákno po zarovnání bude $n/8+4-[(n/8) \mod 4]$ B. Celková velikost pro blok je tedy $\{n/8+4-[(n/8) \mod 4]\}*t$ B. Stejná část paměti je zároveň sdílena s polem pro uložení makespanu každého vlákna, který má velikost 4 B. Její minimální velikost musí být tedy $4 * t$ B pro celý blok. Dále je tato část paměti sdílena s polem, ve kterém se provádí řazení aktivit a konečných časů. Velikost každého z polí je následující mocnina 2 po počtu aktivit. To lze spočítat jako $4 * 2^{\lceil \log_2 n \rceil}$. Proto musí být minimální velikost této paměti alespoň $2 * 2^{\lceil \log_2 n \rceil}$ B. Také tato část paměti je zarovnána na 4 B. Celková část této paměti je tedy $\max(\{n/8 + 4 - [(n/8) \mod 4]\} * t, 4 * t, 4 * 2^{\lceil \log_2 n \rceil})$ B.

Celková velikost potřebné sdílené paměti pro jeden blok lze vypočítat dle následujícího vzorce:
$\{n+4-(n \mod 4)\}+\{2*[K+(K \mod 2)]\}+12+\{2*[\max(n,t)+(\max(n,t) \mod 2)]\}+$
$\{\max(\{n/8+4-[(n/8) \mod 4]\}*t, 4*t, 4*2^{\lceil \log_2 n \rceil})\}$ B

Pro výpočet instance se 122 aktivitami a 4 zdroji, který bude využívat 512 vláken na blok, je potřeba 9 360 B. Jestliže je velikost sdílené paměti na multiprocesoru 48 KB, lze na jednom SM spustit až 5 bloků.

8.3.3 Ostatní paměti

Projektová síť, tedy předchůdci (resp. následovníci) aktivit a požadavky aktivit na zdroje, je uložena v paměti textur. Požadavky aktivit na zdroje jsou v paměti uloženy za sebou v 1D poli následovně $\{r_{0,0}; r_{0,1}; \ldots; r_{0,k}; r_{1,0}; \ldots r_{n,q}\}$, kde n je počet aktivit a q počet zdrojů. V případě sítě s předchůdci jsou použita dvě pole. V jednom poli jsou za sebou uloženi vždy předchůdci první aktivity poté předchůdci druhé aktivity atd. Druhé pole jsou indexy, kde v prvním poli začínají předchůdci aktivity. Pokud je potřeba získat předchůdce aktivity j, je nejprve potřeba získat z druhého pole počet předchůdců aktivit s indexem menším než j, toto číslo je uloženo v poli na indexu j, a počet aktivit, které se získá jako $p[j+1] - p[j]$.

Další podrobnosti o instanci, jako počet aktivit, počet zdrojů atd. jsou kernelu předávány jako parametr. Jsou tedy uloženy v konstantní, resp. read-only cache, paměti.

8.4 Chromosome adjustment

V metodě Chromosome adjustment bylo třeba vyřešit problém s řazením. Seřazení priority a aktivit dle konečných časů pro každého jedince jedním vláknem je velké zpomalení celého výpočtu. Proto se používá chromosome adjustment postupně od prvního až po posledního jedince. Nejprve se načtou do sdílené paměti potřebná pole (priority a konečné časy aktivit), tato pole se postupně seřadí a nakonec se provede uložení priority do chromozomu do globální paměti. Poté přichází na řadu další jedinec. Toto řešení je velmi rychlé díky paralelnímu řadícímu algoritmu bitonic sort. Pseudokód tohoto postupu je uveden v Algoritmus 6.

Algoritmus 6 Chromosome adjustment GPU verze

Input: PR - pole polí priorit aktivit v globální paměti
Input: FT - pole polí konečných časů aktivit v globální paměti
Input: Individuals - množina indexů jedinců
Output: AdjustedChromosme - pole polí chromozomů v globální paměti

1: $sharedPriority$ //pole ve sdílené paměti
2: $sharedFinishTime$ //pole ve sdílené paměti
3: **for all** $i \in Individuals$ **do**
4: $sharedPriority \leftarrow PR[i]$ //načti do sdílené paměti priority jedince i
5: $sharedFinishTime \leftarrow FT[i]$ //načti do sdíl. paměti konečné časy aktivit jedince i
6: $bitonic\_sort(sharedPriority)$ //paralelně seřaď priority
7: $bitonic\_sort(sharedFinishTime)$ //paralelně seřaď konečné časy aktivit
8: //chromosme_adjustment - funkce, která provede chromosome adjustment a uložení do globální paměti
9: $AdjustedChromosme[i] \leftarrow chromosme\_adjustment(sharedPriority, sharedFinishTime)$
10: **end for**

8.5 Komunikace mezi bloky

Celá implementace je postavena na modelu ostrovního genetického algoritmu. Každý blok je jeden ostrov, na kterém se vyvíjí jedinci. Po určitém počtu generací v bloku se provádí migrace. Tato migrace probíhá na samostatné místo v globální paměti. Jedná se tedy o model komunikace pomocí migračního ostrova. Paměť je členěna jako 2D pole. V tomto poli jsou uloženy chromozomy jedinců, kteří migrují. Jeden jeho rozměr je počet jedinců M, kteří migrují z ostrova i, druhý rozměr je počet aktivit n (chromozom jedince). Pole má tedy velikost $M * gridDim.x \times n$. Každý jedinec na migračním ostrově má zároveň proměnnou, kterou lze uzamknout pomocí atomických operací. Zámky jsou v paměti reprezentovány jako pole o velikosti $M * gridDim.x$. Pokud se zapisuje, resp. čte, z migračního ostrova a je zámek uzamčen (jiný blok zapisuje nebo čte požadovaného jedince), je migrace odložena a provedena v další generaci.

Na migrační ostrov vždy migrují nejlepší jedinci v celé populaci v rámci bloku. Toto provádí prvních M vláken v bloku. Každé vlákno zapisuje na ostrov jednoho jedince (resp. jeho chromozom). Každý blok má předem vyhrazen svůj prostor v paměti, kam své jedince zapisuje.

Získávání jedinců z migračního ostrova probíhá tak, že jedno vlákno náhodně určí index z jedince ($z < M * gridDim.x$), od kterého vlákna budou číst jedince z migračního ostrova. Získávání jedinců provádějí poslední vlákna v bloku a získanými jedinci nahrazují mutaci, resp. generování jedinců pro novou generaci. Stejně jako při zápisu na migrační ostrov je při získávání jedinců získáno právě M jedinců. Pokud nastane, že by vlákno překročilo velikost migračního pole, tj. index jedince pro získání je $l > M * gridDim.x$, je index jedince, který bude získán, vypočítán jako: $l \mod M * gridDim.z$. Začíná se tedy s jedinci od začátku pole.

8.6 Spolupráce více karet

Spolupráce mezi více kartami probíhá obdobně jako komunikace mezi bloky. Komunikace probíhá pouze mezi kartami, které mají mezi sebou povoleno P2P. Díky tomu je možno z jedné karty přistupovat do globální paměti druhé karty. Jako model komunikace mezi kartami je zvolen model Left-Right, tj. nejprve komunikují s kartou vlevo a poté s kartou vpravo. Každá karta má svou část paměti, kam ostatní karty ukládají své nejlepší jedince. Tato část paměti má stejný model jako část pro komunikaci mezi bloky, tedy 2D pole s chromozomy a zámky těchto chromozomů. Celý proces lze rozdělit na dvě části - zápis jedinců a čtení jedinců.

Zápis jedinců provádí pouze první blok na GPU a to místo jedné iterace genetického algoritmu. Vlákna zapisují nejlepší jedince z každého bloku. Nejlepší jedinci jsou získáni z migračního ostrova určeného pro komunikaci mezi bloky. Zápis je prováděn pomocí P2P. Každá karta má tedy ukazatel na levou a pravou kartu na pole, kam budou chromozomy zapsány. Zápis probíhá vždy po předem daném počtu iterací genetického algoritmu.

Získání jedinců, které zapsala jiná karta, probíhá stejně jako získání jedinců z migračního ostrova použitého mezi bloky. První vlákno vygeneruje index, odkud se jedinci zapsaní jinou kartou budou číst. Poslední vlákna v bloku po předem daném počtu iterací provedou získání jedinců. Získání jedinců z pole probíhá bez komunikace s jinými kartami, proto ho provádí každý blok.

8.7 Generování náhodných čísel

Pro generování náhodných čísel je použita knihovna cuRAND, která je součástí CUDA Toolkit. Tato knihovna je optimalizovaná pro použití na GPU. Každé vlákno má svůj vlastní generátor. Jako semínko generátoru je použit aktuální čas plus globální index vlákna ($blockDim.x * blockIdx.x + threadIdx.x$). Pro distribuci čísel je použito modulo.

Kapitola 9

Experimenty

9.1 Ohodnocení kvality

Existuje několik možností, jak lze porovnat kvalitu řešení. Mohou to být:

- Průměrná vzdálenost od CPM (Avg. CPM dist.) je nejčastější způsob, jak zjistit kvalitu výsledných řešení. Je to průměr procentuálních vzdáleností od kritické cesty. Použití této jednotky je nejlepší ukazatel, protože délka kritické cesty je pro každou instanci problému neměnná. Čím nižší je tím lepší.
- Speedup neboli zrychlení (S_p), které určuje, jaké je zrychlení jedné verze oproti jiné verzi implementace, resp. nastavení se stejným počtem SGS. Slouží například pro srovnání, jaké je zrychlení paralelní verze implementace ve srovnání s verzí sériovou. Čím vyšší je tím lepší.
- Počet ohodnocených SGS za sekundu (SGS/s) říká, kolik rozvrhů je implementace schopna sestavit pomocí SGS za jednu sekundu. Čím vyšší je tím lepší.
- Dalším hodnotícím kritériem je průměrný čas ohodnocení jedné instance (Avg. čas).

9.2 Testovací prostředí

Pro testování byla zvolena dvě testovací prostředí. První je velmi výkonný server, který obsahuje čtyři nejmodernější grafické karty. Druhé testovací prostředí je běžný notebook s běžnou grafickou kartou. Jejich parametry jsou následující:

9.2.1 Server

- Procesor: 2x Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz s 6 jádry a s Hyper-threading
- Paměť: 64 GB
- GPU: 4x NVIDIA Tesla K20c 13x SM

- Operační systém: GNU/Linux Gentoo 64 bit a jádro 3.12.15
- Verze CUDA: 5.5
- Verze GCC: 4.7.3

9.2.2 Notebook

- Procesor: Intel(R) Core(TM) i7-4702MQ CPU @ 2.20GHz s 4 jádry a s Hyper-threading
- Paměť: 8 GB
- GPU: NVIDIA GeForce GT 750M
- Operační systém: GNU/Linux Kubuntu 14.04 64 bit a jádro 3.13.0
- Verze CUDA: 5.5
- Verze GCC: 4.8.2

9.2.3 Kompilace

Obě implementace jsou zkompilovány pomocí GCC, resp. pomocí NVCC. Obě prostředí využívají kód se stejnými optimalizačními flagy pro kompilaci.

Pro CPU verzi to jsou:

- GCC: -march=native -O3 -pipe -funsafe-math-optimizations -fopenmp

Pro GPU verzi je kód zkompilován s:

- NVCC pro server: -arch=compute_35 -code=sm_35 --maxrregcount=32
- NVCC pro notebook: -arch=compute_30 -code=sm_30

9.2.4 Výchozí nastavení CPU implementace

Výchozí nastavení je stanoveno dle experimentů s implementací tak, aby bylo dosaženo co nejlepšího výsledku. Pokud není uvedeno jinak, je výchozí nastavení implementace pro CPU v experimentech uvedené v tabulce 9.1.

9.2.5 Výchozí nastavení GPU implementace

Výchozí nastavení je stanoveno dle experimentů s implementací tak, aby bylo dosaženo co nejlepšího výsledku. Pokud není určeno jinak, je výchozí nastavení implementace pro GPU v experimentech uvedené v tabulce 9.2.

Počet generací GA	250
Velikost populace (násobek počtu aktivit)	10
Počet jedinců z populace v TOP	10 %
Počet jedinců z populace v BOT	20 %
Pravděpodobnost křížení	70 %
Počet FBI	1
Počet N_{walk}	100
Počet jedinců pro Walking	10
Počet jedinců pro Hidden order jumping	2

Tabulka 9.1: Výchozí nastavení CPU implementace

Počet bloků na SM	4
Počet vláken v bloku	512
Počet generací GA	250
Velikost populace (pro jedem blok)	512
Počet jedinců z populace v TOP	10 %
Počet jedinců z populace v BOT	20 %
Pravděpodobnost křížení	70 %
Počet FBI	1
Počet migrujících jedinců (mezi bloky)	20
Migrace po počtu iterací GA (mezi bloky)	20
Počet migrujících jedinců (mezi zařízeními)	25
Migrace po počtu iterací GA (mezi zařízeními)	25

Tabulka 9.2: Výchozí nastavení GPU implementace

9.3 PSPLIB

PSPLIB (Project Scheduling Problem Library) [9] je knihovna pro generování nejen instancí RCPSP a množina instancí pro benchmark. Množiny instancí jsou čtyři J30, J60, J90, J120. První tři obsahují 480 instancí a čtvrtá obsahuje 600 rozdílných instancí problému RCPSP. Množiny se liší v maximálním počtu aktivit 30, 60, 90 a 120. Všechny množiny mají maximálně 4 zdroje. Většina prací uvádí průměrnou vzdálenost od CPM za určitý počet SGS (maximální počet SGS je většinou 50 000 nebo 500 000).

Toto srovnání není pro GPU implementaci příliš vhodné. GPU implementace je zaměřená na velkou paralelizaci. V případě, že má karta 13 SM, kde na každém SM jsou spuštěny 4 bloky s 512 vlákny a je provedena jedna iterace FBI, je počet SGS v jedné generaci 79 872. V případě, že jsou použity čtyři karty, je to až 319 488 SGS. Pro úplnost jsou zde však tyto výsledky také uvedeny pro každou instanci ve srovnání s obdobnými pracemi. Srovnání probíhá se čtyřmi nastaveními implementací. Jsou to GPU implementace využívající všechny 4 grafické karty na serveru, GPU implementace využívající jednu grafickou kartu na serveru, GPU implementace využívající jednu grafickou kartu na notebooku a CPU implementace na serveru. Testy nejsou provedeny s benchmarkem J30, protože počet aktivit v benchmarku

je velmi malý a z výsledků nelze říci, jak je implementace kvalitní. Většina prací neuvádí výsledky benchmarku J90. Nejlepší výsledek, tj. nejkratší průměrná vzdálenost od CPM, je vždy označen tučně.

9.3.1 J60

Srovnání prací s maximálním množstvím SGS 500 000 pro benchmark J60 je uvedeno v tabulce 9.3.

Práce	**Avg. CPM dist [%]**	**Avg. čas [s]**	**Generací GA**	**Poznámky**
GPU 4x Tesla K20c	12,01	0,1	1	52 bloků na kartu
GPU 1x Tesla K20c	11,15	0,3	6	52 bloků
GPU GeForce GT 750M	10,72	0,94	27	13 bloků
CPU server	10,73	0,81	200	
Gonçalves [6]	**10,49**	52,5	308	
Lim [11]	10,51	7,40		
Debels [5]	10,68	1,106		50 000 SGS
Mendes [12]	10,67	20,11	250	63 546 SGS

Tabulka 9.3: Srovnání prací J60 max 500 000 SGS

9.3.2 J90

Výsledky množiny instancí J90 nejsou ve srovnávaných článcích uvedeny. Pro úplnost je zde však i tato instance uvedena. Srovnání prací s maximálním množstvím SGS 500 000 pro benchmark J120 je uvedeno v tabulce 9.4.

Práce	**Avg. CPM dist [%]**	**Avg. čas [s]**	**Generací GA**	**Poznámky**
GPU 4x Tesla K20c	11,94	0,16	1	52 bloků na kartu
GPU 1x Tesla K20c	10,96	0,43	6	52 bloků
GPU GeForce GT 750M	10,41	1,23	27	13 bloků
CPU server	**9,93**	0,83	149	
Debels [5]	-	1,815	-	Avg. CPM není uvedeno

Tabulka 9.4: Srovnání prací J90 max 500 000 SGS

9.3.3 J120

Srovnání prací s maximálním množstvím SGS 500 000 pro benchmark J120 je uvedeno v tabulce 9.5.

Práce	**Avg. CPM dist [%]**	**Avg. čas [s]**	**Generací GA**	**Poznámky**
GPU 4x Tesla K20c	37,20	0,27	1	52 bloků na kartu
GPU 1x Tesla K20c	34,37	1,21	6	52 bloků
GPU GeForce GT 750M	32,94	3,75	27	13 bloků
CPU server	31,08	2,31	120	
Gonçalves [6]	30,8	180	154	
Lim [11]	**29,91**	30,65		
Debels [5]	30,69	2,992		50 000 SGS
Mendes [12]	31,20	112,46	250	127 341 SGS

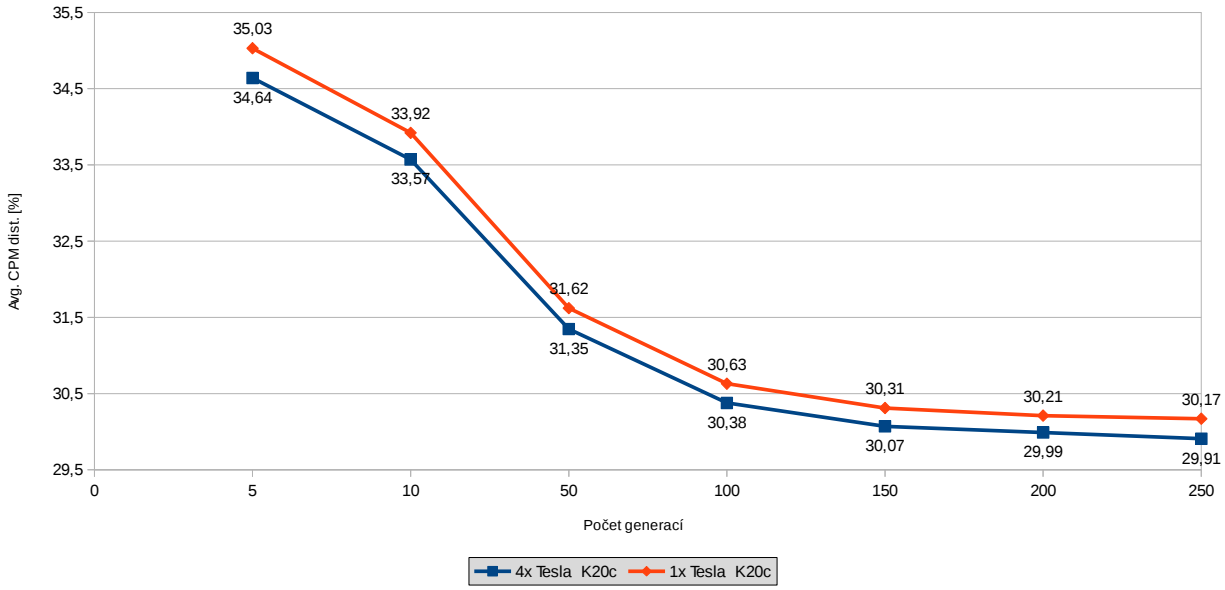
Tabulka 9.5: Srovnání prací J120 max 500 000 SGS

Další srovnání pro instance benchmarku J120 je bez omezení počtu SGS. Jediným omezením u implementací této práce je počet iterací genetického algoritmu - 250 iterací. Výsledky jsou uvedeny v tabulce 9.6. V tabulce je také uveden nejlepší dosažený výsledek další práce, která se zabývá řešením problému RCPSP na GPU [3]. GPU implementace, která využívá 4 grafické karty Tesla K20c, dokázala nalézt nejmenší průměrnou vzdálenost od CPM. Jediných srovnatelných výsledků dosahuje práce Lim [11]. I další implementace této práce dosáhly velmi dobrých výsledků. Ve srovnání s pracemi, které mají omezení na 500 000 SGS, dosahují lepších výsledků pouze dvě práce. Díky využití 4 grafických karet bylo dosaženo průměrně 1 584 766 SGS/s. S tímto výkonem lze ohodnotit všech 600 instancí benchmarku J120 za 190 vteřin, s omezením 500 000 SGS na instanci.

Práce	**Avg. CPM dist [%]**	**Avg. čas [s]**	**Velikost populace**	**SGS/s**	**Poznámky**
GPU 4x Tesla K20c	**29,908**	35,3	106 496	**1 584 766**	52 bloků na kartu
GPU 1x Tesla K20c	30,171	35,17	26624	398 340	52 bloků
GPU GeForce GT 750M	30,739	33,25	6 144	65 072	13 bloků
CPU server	30,670	4,8	1200	153 996	
Lim [11]	29,91	30,65			
Bukata [3]	37,11				GPU Tabu Search

Tabulka 9.6: Srovnání prací J120 bez omezení počtu SGS

Důkaz, že navržený genetický algoritmus s počtem generací konverguje ke kratší průměrné vzdálenosti od CPM, je uveden v grafu na obrázku 9.1. Jsou zde uvedeny výpočty na GPU a to v testovacím prostředí serveru s grafickými kartami Tesla K20c. V grafu jsou zaneseny výsledky pro jednu grafickou kartu a pro využití všech 4 grafických karet na serveru. Všechna nastavení jsou výchozí s výjimkou proměnného počtu generací. Z grafu lze vyčíst, že počet karet zlepšuje výslednou průměrnou vzdálenost od CPM za stejný počet generací a to v průměru o 0,28 %. Především rozdíl 0,26 % při 250 generacích je v souvislosti benchmarkem J120 opravdu velké zlepšení.

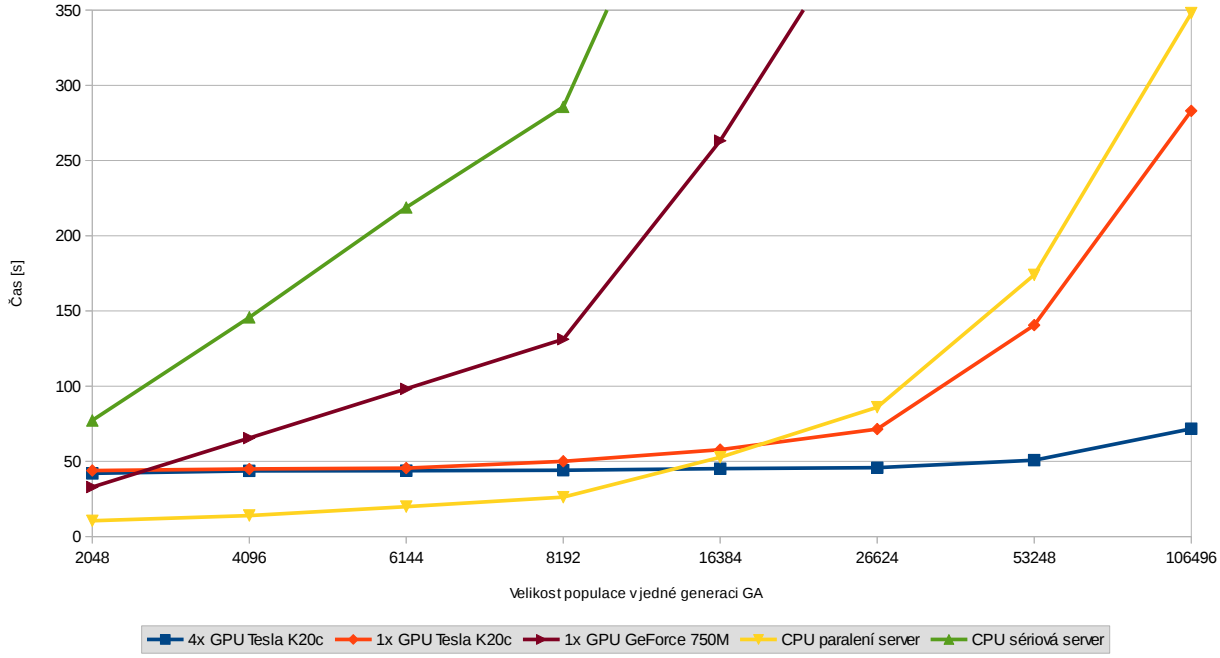


Obrázek 9.1: Konvergence GA v závislosti na počtu generací

9.4 Rychlost výpočtu

K měření rychlosti výpočtu a zrychlení byla použita instance j12036_10 benchmarku J120. Tato instance byla zvolena díky tomu, že rozdíl nejlepšího nalezeného makespanu této práce (219) od nejlepšího nalezeného optimálního řešení (216) a lower bound (198) je značný. Srovnání je provedeno tak, že všechny implementace mají vždy stejnou velikost populace GA. Zvolené velikosti odpovídají násobkům 4, aby je bylo možné aplikovat i na měření na 4 grafických kartách a dále různým počtem bloků na SM. Naměřené výsledky jsou zapsány v tabulce 9.7 a zaneseny do grafu na obrázku 9.2.

Z naměřených výsledků vyplývá, že se GPU implementace ve srovnání s paralelní implementací pro CPU výrazně vyplácí, když je populace větší nebo rovna 26 624 jedincům. S využitím více grafických karet, konkrétně čtyř, je výhodné využít GPU implementaci již při velikosti populace 16 384 jedinců. Dále je prokázáno, že běžné grafické karty jako je GeForce GT 750M nejsou pro náročné výpočty příliš vhodné. Zrychlení mezi sériovou a paralelní verzí CPU implementace je 10 násobné při využití největší měřené velikosti populace. Zrychlení GPU implementace s využitím 4 grafických karet je oproti sériové verzi CPU implementace dokonce více než 51 násobné, ve srovnání s paralelní verzí více než 4,8 násobné s použitím největší měřené velikosti populace. Toto zrychlení ukazuje, že navržený algoritmus je velmi vhodný pro paralelizaci. Z měření vyplývá, že přidáním více grafických karet lze dosáhnout mnohanásobného zrychlení.



Obrázek 9.2: Graf času [s] výpočtu instance j12036_10 s 250 generacemi GA

Velikost populace:	2048	4096	6144	8192	16384	26624	53248	106496
GPU 4x Tesla K20c	42,0	43,7	43,8	44,1	**45,1**	**45,8**	**50,8**	**71,7**
GPU 1x Tesla K20c	43,9	44,9	45,5	50,1	57,8	71,5	140,6	283,1
GPU GeForce GT 750M	32,9	65,5	98,2	131,2	263,2	427,2	853,9	1711,8
CPU paralelní server	**10,5**	**14**	**19,9**	**26,2**	52,7	86	174	348
CPU sériová server	77,2	145,7	218,8	285,7	571,1	925,7	1837,9	3687,9

Tabulka 9.7: Čas [s] výpočtu instance j12036_10 s 250 generacemi GA

9.5 Shrnutí experimentů

Při experimentech byla nejlepší průměrná vzdálenost od CPM u benchmarku J120 29,908 % při 250 generacích GA a s využitím 4 grafických karet. V tomto měření bylo nalezeno 369 z 600 řešeních, které mají stejný makespan jako dosud nejlepší známá řešení a byly nalezeny tři makespany lepší než dosud známé u dosud nejlepších řešení. V porovnání s průměrnou vzdáleností od CPM, všech dosud nejlepších vzdáleností od CPM, která je 29,18 %, je dosažený výsledek na velmi dobré úrovni. Při experimentech s J120 na instanci bylo nalezeno 10 zlepšení dosud nejlepších nalezených vzdáleností od CPM. U všech deseti instancí byl makespan zmenšen o jedna. Tyto instance a jejich nalezené makespany jsou uvedeny v tabulce 9.8.

Instance	**Dosud nejlepší makespan**	**Nově nalezený makespan**
j1207_10	118	117
j1208_3	96	95
j1209_4	87	86
j12026_9	172	171
j12027_4	108	107
j12027_7	125	124
j12034_3	102	101
j12034_9	95	94
j12046_9	166	165
j12048_5	111	110

Tabulka 9.8: Vylepšené instance J120

Kapitola 10

Závěr

Všechny stanovené cíle uvedené v úvodu byly splněny. Povedlo se úspěšně navrhnout a implementovat CPU a GPU verzi algoritmu, která řeší problém RCPSP. Obě implementace dosahují výborných výsledků jak v kvalitě výsledných řešení, tak i v rychlosti. Implementace jsou schopné nalézt velmi dobrou průměrnou vzdálenost od CPM. Ve srovnání s dalšími pracemi zabývajícími se stejným problémem patří dosažené výsledky, zjištěné během experimentů, k nejlepším. Toto je velký úspěch, protože problém RCPSP je velmi známý a existuje mnoho prací, které se řešením tohoto problému zabývají.

Bylo prokázáno, že genetický algoritmus lze velmi dobře paralelizovat za pomoci CUDA. Velkým pokrokem je úspěšná implementace algoritmu SGS pro CUDA, který ze svého návrhu není příliš vhodný pro výpočty pomocí CUDA. Ačkoliv rozdíl v rychlosti výpočtu pomocí grafických karet není ve srovnání s výpočtem pomocí implementace paralelní verze pouze pro CPU nikterak výrazný, je prokázáno, že implementace pro GPU je možná a rychlejší než CPU verze. Zároveň poukazuje vhodnost využití grafických karet s pomocí CUDA pro řešení dalších problémů v oblasti kombinatorické optimalizace. Výsledná implementace s využitím více grafických karet představuje obrovský výpočetní výkon pro řešení problému RCPSP. Právě propojení více grafických karet s využitím technologií P2P a UVA umožnilo dosáhnout takto dobrých výsledků. Zároveň tato diplomová práce představuje návod, jak použít více grafických karet současně pro výpočet za pomoci CUDA.

Dále bylo zjištěno, že průměrná grafická karta NVIDIA GeForce GT 750M není příliš výkonná a nehodí se pro využití řešení problému RCPSP s využitím dané implementace. Oproti této kartě je grafická karta NVIDIA Tesla K20c několikanásobně výkonnější.

GPU implementace nabízí prostor pro budoucí zlepšení, především v oblasti využití některého algoritmu pro vylepšení rozvrhu. Další možné zlepšení výsledků lze očekávat s využitím více než čtyř grafických karet.

Největším úspěchem této práce je nalezení deseti vylepšení instancí benchmarku PSPLIB u množiny J120. Tyto instance byly zaslány emailem do databáze nejlepších řešení PSPLIB [9].

Literatura

[1] Arabas J., Michalewicz Z. a Mulawka J. GAVaPS-a genetic algorithm with varying population size. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, s. 73–78 vol.1, Jun 1994.

[2] Blazewicz J. and Lenstra J.K. a Rinnooy Kan A.H.G. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*. 1983, 5, 1, s. 11 – 24.

[3] Bukata L. GPU algorithms design for Resource Constrained Project Scheduling Problem. Master's thesis, Czech Technical University in Prague, May 2012.

[4] Christopher T. W. *Bitonic Sort* [online]. [cit. 6. 5. 2014]. `http://www.tools-of-computing.com/tc/CS/Sorts/bitonic_sort.htm`.

[5] Debels D. a Vanhoucke M. A Decomposition-Based Genetic Algorithm for the Resource-Constrained Project-Scheduling Problem. *OPERATIONS RESEARCH*. 2007, 55, 3, s. 457–469.

[6] GonçalvesJ. F., Resende M. G. C. a Mendes J. J. M. A Biased Random-Key Genetic Algorithm with Forward-Backward Improvement for the Resource Constrained Project Scheduling Problem. *Journal of Heuristics*. 2011, 17, 5, s. 467–486.

[7] Khronos Group. *OpenCl* [online]. [cit. 6. 5. 2014]. `https://www.khronos.org/opencl/`.

[8] Kolisch R. Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation. *European Journal of Operational Research*. 1996, 90, s. 320 – 333.

[9] Kolisch R. a Sprecher A. *PSPLIB benchmark* [online]. [cit. 28. 4. 2014]. `http://www.om-db.wi.tum.de/psplib/`.

[10] Li K.Y. a Willis R.J. An iterative scheduling technique for resource-constrained project scheduling. *European Journal of Operational Research*. 1992, 56, 3, s. 370 – 379.

[11] Lim A., Ma H., Rodrigues B., Tan S.T. a Xiao F. New meta-heuristics for the resource-constrained project scheduling problem. *Flexible Services and Manufacturing Journal*. 2013, 25, 1-2, s. 48–73.

[12] Mendes J. J. M., Gonçalves J. F. a Resende M. G. C. A random key genetic algorithm for the resource constrained project scheduling problem. *Computers & Operations Research*. 2009, 36, 1, s. 92 – 109.

[13] Micikevicius P. NVIDIA. *Multi-GPU Programming* [online]. 2011. [cit. 28. 4. 2014]. http://www.nvidia.com/docs/IO/116711/sc11-multi-gpu.pdf.

[14] NVIDIA. *CUDA* [online]. [cit. 28. 4. 2014]. http://www.nvidia.com/object/cuda_home_new.html.

[15] NVIDIA. *CUDA Toolkit Documentation* [online]. [cit. 28. 4. 2014]. http://docs.nvidia.com/cuda/index.html.

[16] OpenMP. *The OpenMP® API specification for parallel programming* [online]. [cit. 2. 5. 2014]. http://openmp.org.

[17] OpenMP. *OpenMP® Support for the OpenMP language* [online]. [cit. 2. 5. 2014]. http://openmp.llvm.org/.

[18] OpenMP. *OpenMP Specifications* [online]. [cit. 2. 5. 2014]. http://openmp.org/wp/openmp-specifications/.

[19] Šůcha P. a Zajíček T. Accelerating an Algorithm for Flow Shop Scheduling Problem on the GPU. 2012.

Příloha A

Seznam použitých zkratek

AL Activity list

CPM Critical Path Method (délka kritické cesty)

CPU Central Procesor Unit

CUDA Compute Unified Device Architecture

FBI Forward-Backward Improvement

GA Genetický algoritmus

GPU Graphics Processing Unit

IDE Integrated Development Environment

P2P Peer-to-Peer

RCPSP Resource Constrained Project Scheduling Problem

SGS Schedule Generation Schemes

SM Streaming Multiprocesor

UVA Unified Virtual Addressing

Příloha B

Instalační a uživatelská příručka

Zdrojové kódy je možné získat z přiloženého CD. Instalace je popsána pro operační systém GNU/Linux distribuce Kubuntu 14.04, ale měla by jít aplikovat na většinu moderních distribucí operačního systému GNU/Linux. Pro spuštění na MS Windows je potřeba zdrojové kódy naimportovat a zkompilovat pomocí Visual Studia, potřebný projekt přiložen není.

B.1 Kompilace

Pro zkompilování zdrojových kódů je potřeba:

- Operační systém GNU/Linux (testováno Kubuntu 14.04)
- NVIDIA CUDA toolkit >=5.5
- GCC >=4.7
- OpenMP
- CMake >= 2.8
- CUDA compute capability: >=3.0

Pozn.: Je pravděpodobné, že budou dostačovat i nižší verze nástrojů, avšak to není zaručeno, neboť to nebylo testováno.

Implementace pro CPU využívá pro sestavení Makefile. Kompilace probíhá za pomoci následujících příkazů (z adresáře, kde jsou zdrojové kódy umístěny):

```
make #přeloží zdrojové kódy
```

Pokud při překladu není vypsána žádná chyba, přeložení proběhlo správně a v adresáři vznikne spustitelný soubor s názvem **RCPSP**.

Implementace pro GPU využívá pro sestavení CMake. Program CMake dokáže vygenerovat Makefile, který lze poté zkompilovat. Kompilace probíhá za pomocí těchto příkazů (z adresáře, kde jsou zdrojové kódy umístěny):

```
mkdir build #vytvoří adresář build
cd build #vstoupíte do adresáře build
cmake .. #spustíte cmake o úroveň výše než je adresář build
make #přeloží zdrojové kódy
```

Pokud není v průběhu kompilace vypsána žádná chyba, v adresáři build vznikne spustitelný soubor s názvem **rcpsp_gen_gpu**. Výsledný zdrojový kód je určen pro grafické karty s compute capability 3.5. Pokud má být zkompilován pro karty s jinou compute capability, je potřeba změnit compute capability v souboru CMakeList.txt. Konkrétně řádek:

```
set(CUDA_NVCC_FLAGS ${CUDA_NVCC_FLAGS} -arch=compute_35 -code=sm_35)
```

a změnit číslo 35, které značí compute capability 3.5, na compute capability grafické karty, např. pro compute capability 3.0 to bude číslo 30 a řádek bude vypadat následovně:

```
set(CUDA_NVCC_FLAGS ${CUDA_NVCC_FLAGS} -arch=compute_30 -code=sm_30)
```

B.2 Spuštění a popis parametrů CPU implementace

Spuštění CPU implementace algoritmu s výchozím nastavením probíhá pomocí příkazu:

```
./RCPSP -if instance_RCPSP
```

Kde instance_RCPSP je umístění instance RCPSP ve formátu PSPLIB s příponou sm. Lze použít také wildcard (např. *) pro spuštění více instancí. Např. textitJ120/j120*.sm bude řešit všechny instance začínající na j120 a příponou .sm v adresáři J120. Při spuštění lze nastavit několik argumentů, které jsou popsány v tabulce B.1. Většina argumentů je stejná jako argumenty pro GPU implementaci. Změnu výchozích hodnot a další nastavení lze provést ve zdrojovém kódu v souboru **DefaultConfigureRCPSP.h**.

B.3 Spuštění a popis parametrů GPU implementace

Spuštění s výchozím nastavením se provádí pomocí příkazu:

```
./rcpsp_gen_gpu -if instance_RCPSP
```

Kde instance_RCPSP je umístění instance RCPSP ve formátu PSPLIB s příponou sm. Lze použít také wildcard (např. *) pro spuštění více instancí. Např. textitJ120/j120*.sm bude řešit všechny instance začínající na j120 a příponou .sm v adresáři J120. Při spuštění lze nastavit několik parametrů, které jsou popsány v tabulce B.2. Většina parametrů je stejná jako parametry pro CPU implementaci. Změnu výchozích hodnot a další nastavení lze provést ve zdrojovém kódu v souboru **DefaultConfigureRCPSP.h**.

Parametr	**Zkratka**	**Argument**
Popis		
--input-files	-if	FILE1 FILE2 ... FILEX
instance RCPSP ve formátu PSPLIB		
--help	-h	
vypsání nápovědy		
--number-of-generations	-nog	celé číslo
počet generací GA		
--number-of-forward-backward-iterations	-fbi	celé číslo
počet FBI		
--percent-top-of-population	-top	desetinné číslo (0-1)
velikost TOP v procentech z populace		
--percent-bot-of-population	-bot	desetinné číslo (0-1)
velikost BOT v procentech z populace		
--crossover-probability	-cprob	desetinné číslo (0-1)
pravděpodobnost křížení v procentech		
--multiple-of-individuals-in-population	-moin	celé číslo
velikost populace jako násobek počtu aktivit		
--modified-makespan-distance	-mmd	celé číslo
modified makespan vzdálenost L		
--compute-top	-ctop	
sestavovat rozvrh pokaždé pro TOP jedince		
--nwalk	-nwalk	celé číslo
počet N_{walk}		
--number-of-walking	-now	celé číslo
počet provedení walking		
--number-of-hidden-order-jump	-nohoj	celé číslo
počet provedení hidden order jumping		
--write-makespan-graph	-wmg	
zápis nejlepších nalezených makespan v generacích do CVS souboru		
--write-result-file	-wrf	
zápis řešení do binárního souboru		
--one-line-output	-olo	
pouze jednořádkový výpis řešení		

Tabulka B.1: Nastavitelné parametry CPU implementace

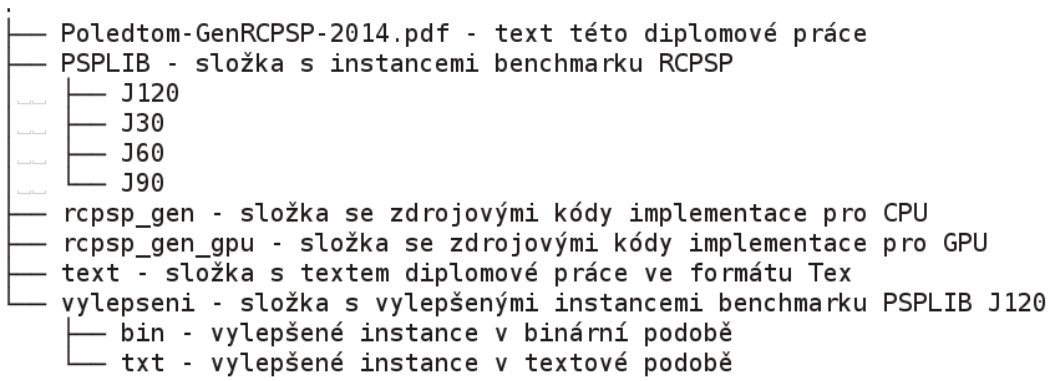
Parametr	**Zkratka**	**Argument**
Popis		
--input-files	-if	FILE1 FILE2 ... FILEX
instance RCPSP ve formátu PSPLIB		
--help	-h	
vypsání nápovědy		
--number-of-generations	-nog	celé číslo
počet generací GA		
--number-of-forward-backward-iterations	-fbi	celé číslo
počet FBI		
--percent-top-of-population	-top	desetinné číslo (0-1)
velikost TOP v procentech z populace		
--percent-bot-of-population	-bot	desetinné číslo (0-1)
velikost BOT v procentech z populace		
--crossover-probability	-cprob	desetinné číslo (0-1)
pravděpodobnost křížení v procentech		
--max-cards-use	-maxcards	celé číslo
maximální počet použitých grafických karet k výpočtu		
--max-number-of-blocks	-maxblocks	celé číslo
počet bloků na jednu grafickou kartu		
--number-of-blocks-per-multiprocessor	-blocks	celé číslo
počet bloků na multiprocesor		
--write-result-file	-wrf	
zápis řešení do binárního souboru		
--one-line-output	-olo	
pouze jednořádkový výpis řešení		

Tabulka B.2: Nastavitelné parametry GPU implementace

Příloha C

Obsah přiloženého CD

Na obrázku C.1 se nachází obsah přiloženého CD včetně popisu významu složek a souborů.



Obrázek C.1: Obsah přiloženého CD