Fakulta informačních technologií

Paralelní architektury počítačů

technická zpráva

Hledání nejkratších cest v grafu

Autoři:
Vojtěch Myslivec
Zdeněk Nový

14. května 2015

Abstrakt

Účelem této práce je sumarizovat výsledky měření řešení problému hledání nejkratších cest v grafu (NCG). Práce se zaměřuje na řešení problému Dijkstrovým a Floyd-Warshallovým algoritmem a porovnání sekvenční a několika paralelních implementací.

Klíčová slova Dijkstra, Floyd-Warshall, nejkratší cesty, NCG, OpenMP, CUDA

Obsah

1 Úvod **4**

2 Hledání nejkratších cest v grafu **4**
2.1 Definice 4
2.2 Algoritmy 4
2.2.1 Dijkstrův algoritmus 4
2.2.2 Floyd-Warshallův algoritmus 4

3 Sekvenční algoritmus **5**
3.1 Společná implementace 5
3.2 Dijkstrův algoritmus 5
3.2.1 Dijkstrův algoritmus z jednoho zdroje 5
3.2.2 Vektorizace 6
3.3 Floyd-Warshallův algoritmus 8
3.4 Implementace 8
3.4.1 Vektorizace 8

4 Paralelní algoritmus pomocí knihovny OpenMP **9**
4.1 Dijkstrův algoritmus 9
4.1.1 Paměťové struktury 9
4.1.2 Úprava algoritmu 9
4.2 Floyd-Warshallův algoritmus 9
4.2.1 První varianta 9
4.2.2 Druhá varianta 10
4.3 Optimalizace implementace 10
4.3.1 Optimalizace vytváření vláken 10
4.3.2 Odstranění výpočtu předchůdců 11
4.4 Měření 11
4.4.1 Testovací data 11
4.4.2 Výsledky 11
4.4.3 Analýza 14
4.4.4 Optimalizace vláken Floyd-Warshallova algoritmu . . . 15
4.4.5 Optimalizace odstraněním předchůdců obou algoritmů 15
4.4.6 Zhodnocení 15

5 Paralelní algoritmy pomocí technologie CUDA 18
5.1 Dijkstrův algoritmus 18
5.1.1 Paměť 18
5.1.2 Výpočet 19
5.1.3 Efektivita využití akcelerátoru 19
5.2 Floyd-Warshallův algoritmus 21
5.2.1 Základní paralelní algoritmus 21
5.2.2 Optimalizace pro paralelní výpočet 21
5.2.3 Profilace 24
5.2.4 Efektivita využití akcelerátoru 24
5.3 Měření 26
5.3.1 Výsledky 26
5.3.2 Vyhodnocení 29

6 Závěr 29

1 Úvod

Tato práce se zabývá implementací dvou algoritmů hledání nejkratších cest v grafu. Jedná se o implementaci sekvenčním algoritmem, který je poté paralelizován pro procesor a pro grafickou kartu. Pro jednotlivé algoritmy je provedeno měření, které si klade za cíl určit zrychlení paralelních algoritmů proti sekvenčnímu.

2 Hledání nejkratších cest v grafu

2.1 Definice

Hledání nejkratších cest v grafu je NP-úplná grafová úloha, jejímž cílem je nalézt v zadaném grafu nejkratší cesty mezi všemi možnými dvojicemi uzlů A a B [6].

2.2 Algoritmy

2.2.1 Dijkstrův algoritmus

Dijkstrův algoritmus slouží k nalezení všech nejkratších cest ze zadaného uzlu do všech ostatních uzlů grafu. Graf nesmí obsahovat hrany se zápornou délkou [4].

Princip Dijkstrův algoritmus je zobecněné prohledávání grafu do šířky, při kterém se vlna šíří na základě vzdálenosti od zdrojového uzlu. K uchovávání uzlů slouží prioritní fronta, která je řazena podle vzrůstající vzdálenosti od zdroje. V každém kroku algoritmu je vybrán uzel s nejmenší vzdáleností a pro každého souseda je vypočítána jeho vzdálenost od zdrojového uzlu [4].

2.2.2 Floyd-Warshallův algoritmus

Floyd-Warshallův algoritmus slouží k nalezení nejkratších cest mezi všemi dvojicemi uzlů v grafu. Graf může obsahovat hrany – nikoliv však cykly – se zápornou hodnotou délky[1] [5].

[1] Pokud má graf cyklus se zápornou hodnotou délky, postrádá úloha nejkratších cest smysl.

Princip Floyd-Warhsallův algoritmus pracuje s maticí sousednosti, kde hrana je ohodnocena vahou. Na počátku tato matice obsahuje pouze vzdálenosti dvou uzlů, mezi kterými je vedena hrana. V každém kroku je vybrán jeden uzel jako prostředník. Prvek matice sousednosti se přepočítá, pokud je vzdálenost z počátečního do koncového uzlu kratší přes nového prostředníka než bez něj [5].

3 Sekvenční algoritmus

3.1 Společná implementace

Oba algoritmy vycházejí z obecného principu, který je popsán v kapitole 2.2.2. Algoritmy pracují s grafem, který je programu předložen jako soubor, ve kterém je graf ve formě matice sousednosti. Společnou částí je tedy načítání vstupu a jeho kontrola.

3.2 Dijkstrův algoritmus

Protože Dijkstrův algoritmus slouží k hledání nejkratších cest od jednoho zdrojového uzlu, je nutné jej spouštět pro každý uzel grafu. To zajišťuje funkce *dijkstraNtoN*.

3.2.1 Dijkstrův algoritmus z jednoho zdroje

Pro výpočet Dijkstrova algoritmu z jednoho zdrojového uzlu se alokují tři pole o velikosti počtu uzlů. V jednom je uložena vzdálenost daného uzlu od zdrojového, ve druhém předchozí uzel v nalezené nejkratší cestě. Třetí pole určuje, jestli je už uzel uzavřený pro výpočty.

Algoritmus prochází postupně, podle nejmenší vzdálenosti, všechny uzly, které se nacházejí v prioritní frontě. Z daného uzlu vypočítá pro každého svého souseda novou cestu, která by vedla přes uzel samotný a porovná ji s dosavadní vzdáleností souseda. Menší vzdálenost je zapsána do pole vzdáleností a algoritmus pokračuje.

Prioritní fronta Za účelem prioritní fronty byla implementována binární halda, kde složitost výběru minima je logaritmická, oproti nativní implementaci pomocí pole, kde je složitost výběru minima lineární.

Obrázek 1: Úspěšně vektorizovaný cyklus.

```
dijkstra.cpp:38: note: LOOP VECTORIZED.
dijkstra.cpp:28: note: vectorized 1 loops in function.
```

```
for ( unsigned j = 0; j < pocetUzlu; j++ ) {
        vzdalenostM[i][j] = DIJKSTRA_NEKONECNO;
        predchudceM[i][j] = DIJKSTRA_NEDEFINOVANO;
}
```

```
dijkstra.cpp:99: note: not vectorized: number
    of iterations cannot be computed.
dijkstra.cpp:99: note: bad loop form.
dijkstra.cpp:82: note: vectorized 0 loops in function.
```

3.2.2 Vektorizace

Pomocí přepínačů optimalizace *-O3* a podpory vektorových sad *-msse4.2* kompilátoru *gcc* byla zapnuta podpora vektorizace cyklů [9]. Pro záznam o pokusech vektorizace byl použit přepínač *-ftree-vectorizer-verbose=n*, kde za *n* byly dosazeny 1, 3, 5, kde čím vyšší číslo, tím podrobnější informace [9].

Samotný algoritmus je velmi sekvenční. V každém kroku se vybere minimum z prioritní fronty — jinými slovy ze složitější struktury — a přepíší se hodnoty uzlu ve frontě. Tyto operace nelze nijak vektorizovat a v průběhu výpočtu (kromě počáteční inicializace polí) se nevyskytují žádné prvky vedle sebe ani pro čtení ani pro zápis.

Původní stav Na obrázcích 1 a 2 je znázorněn příklad jednoho z cyklů, který byl vektorizován a jiný cyklus, který se nepodařilo vektorizovat z důvodu neznámého počtu iterací.

Optimalizace Výpis 3 ukazuje úspěšnou úpravu cyklu, který se díky změně zdrojového kódu podařilo vektorizovat.

Obrázek 2: Cyklus, který se nepodařilo vektorizovat.

```
dijkstra.cpp:99: note: not vectorized: number
    of iterations cannot be computed.
dijkstra.cpp:99: note: bad loop form.
dijkstra.cpp:82: note: vectorized 0 loops in function.

for ( unsigned i = 0 ; i < pocetUzlu ; i++ ) {
        vzdalenostM[idUzlu][i] = vzdalenost[i];
        predchudceM[idUzlu][i] = predchudce[i];
}
```

Obrázek 3: Upravený cyklus, aby mohl být vektorizován.

```
Vectorizing loop at dijkstra.cpp:99
dijkstra.cpp:99: note: LOOP VECTORIZED.
dijkstra.cpp:82: note: vectorized 1 loops in function.

unsigned tmp = pocetUzlu;
for ( unsigned i = 0 ; i < tmp ; i++ ) {
        vzdalenostM[idUzlu][i] = vzdalenost[i];
        predchudceM[idUzlu][i] = predchudce[i];
}
```

Obrázek 4: Úspěšně vektorizovaný cyklus.

```
floydWarshall.cpp:90: note: vectorizing stmts using \
    SLP.BASIC BLOCK VECTORIZED
floydWarshall.cpp:90: note: basic block vectorized using SLP
```

```
for ( unsigned k = 0; k < tmp; k++ ) {
        unsigned i;
        #pragma omp parallel for private(i, novaVzdalenost)
        shared( delkaPredchozi, delkaAktualni,
                predchudcePredchozi, predchudceAktualni)
        for ( i = 0; i < tmp; i++ ) {
                for ( unsigned j = 0; j < tmp; j++ ) {
```

3.3 Floyd-Warshallův algoritmus

Floyd-Warshallův algoritmus obsahuje tři vnořené for cykly a funguje na principu popsaném v 2.2.2. Jako datové struktury používá čtyři matice o velikosti počtu uzlů × počet uzlů. Pro každý uzel si algoritmus udržuje aktuální vzdálenosti ke všem uzlům a navíc vzdálenosti z předchozí iterace. Další dvě matice obsahují předchozí uzel v nalezené cestě.

3.4 Implementace

Aktuální implementace je k nahlédnutí i ke stažení na adrese `https://github.com/VojtechMyslivec/PAP-NCG`. Sekvenční algoritmus se nachází ve složce *01_sekvencni*.

3.4.1 Vektorizace

Algoritmus sice počítá s maticemi, ale během výpočtu se porovnávají čísla, která v dané matici nemusí vůbec sousedit (navíc se porovnávají pouze dvě čísla: jedno na indexu *(i,j)* a druhé, které je součtem dvou čísel na indexech *(i,k)* a *(k,j)*). Z výpisu 4 je patrné, že v programu je vektorizován pouze cyklus pro inicializaci matic.

4 Paralelní algoritmus pomocí knihovny OpenMP

4.1 Dijkstrův algoritmus

Paralelizace algoritmu spočívá v paralelizaci cyklu, který prochází jednotlivé uzly a pro ně řeší problém hledání nejkratší cesty v grafu z jednoho počátečního uzlu. Každé vlákno tedy zpracovává jeden uzel jako počáteční a z něj hledá nejkratší cesty do všech ostatních uzlů.

4.1.1 Paměťové struktury

Každé vlákno dostane ukazatel na strukturu grafu. Protože všechna vlákna používají strukturu grafu pouze ke čtení, nedochází při přístupu k této struktuře k žádným datovým hazardům.

Každé vlákno si vytvoří jeden objekt, ve kterém si alokuje vlastní pole vzdáleností a předchůdců, které používá pro své výpočty. Tyto struktury jsou po ukončení funkce vlákna dealokovány společně s objektem.

4.1.2 Úprava algoritmu

Z důvodu paralelizace algoritmu bylo nutné upravit použitou prioritní frontu. V sekvenčním řešení byla použita implementace pomocí binární haldy 3.2.1. Z důvodu paralelizace výběru minima z fronty je pro paralelní řešení výhodnější použít implementaci polem.

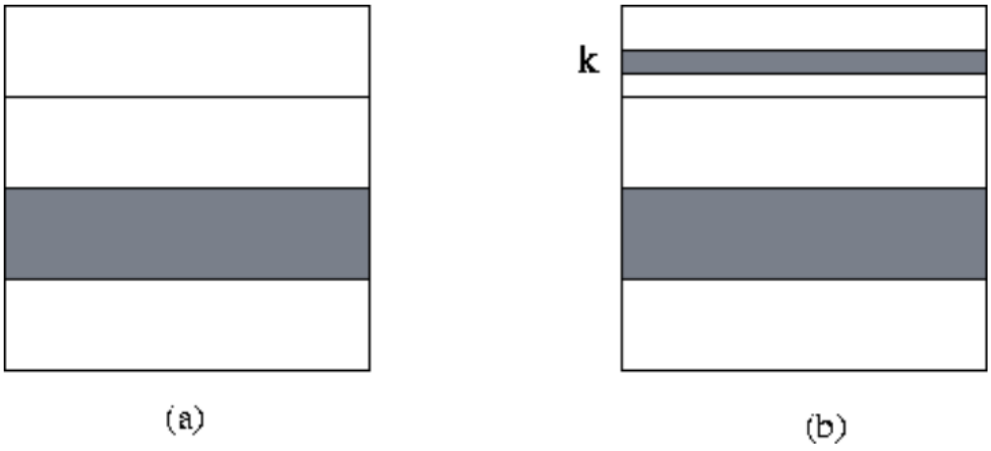
4.2 Floyd-Warshallův algoritmus

Díky třem vnořeným sekvenčního algoritmu existuje několik možností, jak algoritmus paralelizovat.

4.2.1 První varianta

První variantou je algoritmus paralelizovat pouze v jednom cyklu, který definuje, která řádka je právě zpracovávána. Vlákna čtou z matice W_k a zapisují do přidělených řádků v matici W_{k+1}[2]. Přidělené řádky jsou navzájem disjunktní, takže nehrozí konflikt zápisů jednotlivými vlákny (viz obrázek 5).

[2]V každém kroku je potřeba pouze jedna matice z předchozí iterace a jedna matice pro zápis nalezených cest. Na konci každé iterace se pak prohodí matice aktuální s maticí předchozí a mohou se tak nepotřebná starší data přepisovat.



Obrázek 5: Ukázka dat přidělených jednomu vláknu při paralelizaci jednoho cyklu [3].

4.2.2 Druhá varianta

Druhou variantou, jak problém paralelizovat je použít původní variantu a přidat paralelizaci zároveň ve vnitřním cyklu, který prochází jednotlivé sloupce matice. V takovém případě by jednomu vláknu byl přidělen jeden nebo více necelých řádků ohraničených sloupci. Tato varianta se jeví vhodnější pouze při velkém počtu dostupných vláken, proto není v naší implementaci použita.

4.3 Optimalizace implementace

4.3.1 Optimalizace vytváření vláken

Z měření algoritmu Floyd-Warshall v sekci 4.4.2 bylo zjištěno, že implementace v každé iteraci vnějšího cyklu vytváří a spouští nová vlákna, která po skončení vnitřního cyklu ukončí a v další iteraci proces opakuje. Proto byla paralelizace vláken upravena tak, aby na začátku vnějšího cyklu algoritmus vytvořil příslušný počet vláken, kterým pak v jednotlivých iteracích přidělo-val práci. Grafy s touto optimalizací mají tvar **_ v2.png*. Výsledky měření jsou prezentovány v kapitole 4.4.4.

4.3.2 Odstranění výpočtu předchůdců

Protože zadáním této práce nebylo získat matici předchůdců, je v algoritmu zbytečné udržovat informaci o předchůdcích jednotlivých uzlů. Proto byly v rámci další optimalizace odstraněny veškeré výpočty, které se týkaly matice předchůdců. Výsledek byl znovu naměřen a výsledky jsou prezentovány v kapitolách 4.4.5.

4.4 Měření

Na obou algoritmech bylo provedeno měření, které si klade za cíl analyzovat čas, zrychlení a efektivitu použitého paralelního algoritmu. Měření bylo prováděno na hustých grafech, kde při generování grafů byla použita pravděpodobnost 0.5, že mezi dvěmi uzly existuje hrana.

4.4.1 Testovací data

Byly vygenerovány 2 testovací sady dat. Obě sady obsahují 25 grafů, kde pro každé n (počet uzlů) z 1000, 2000, 3000, 4000 a 5000 je vygenerováno 5 náhodných grafů. Jedna sada obsahuje *husté* grafy s pravděpodobností hrany 50%, druhá obsahuje *řídké* grafy s pravděpodobností hrany 1%. Měření probíhalo na serveru `star2.fit.cvut.cz` na stroji *gpu-02* pro počet vláken p 1, 2, 4, 6, 8, 12 a 24.

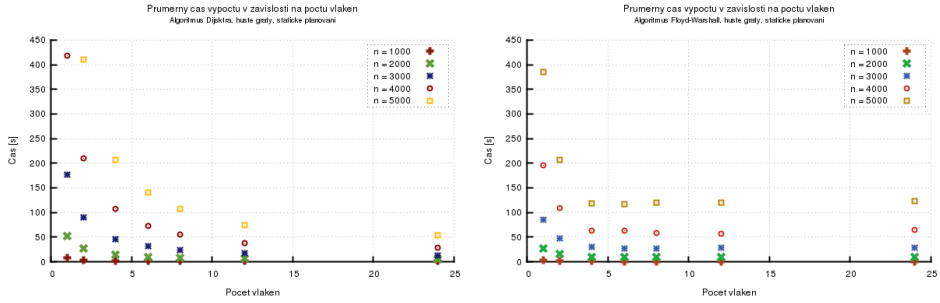
Jako čas výpočtu pro dané n se vypočítal průměr z časů přes všech 5 grafů s odpovídajícím počtem uzlů.

4.4.2 Výsledky

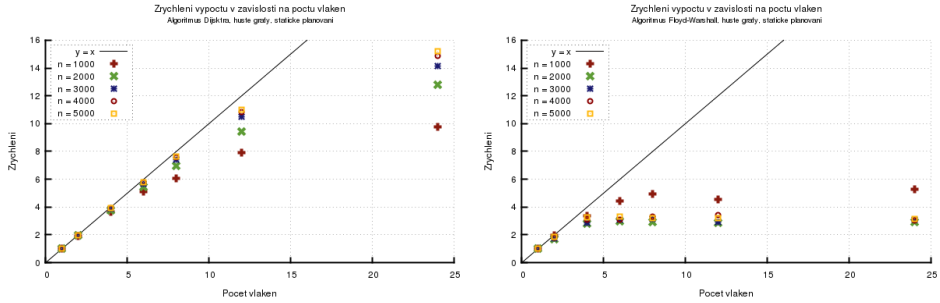
Grafy 6, 7 a 8 ukazují výsledky měření z pohledu času, zrychlení a efektivity.

Poměr rychlosti algoritmů v závislosti na hustotě grafu Graf 11 porovnává výpočetní časy při použití statického plánování v závislosti na hustotě grafu. Z grafů vyplývá, že oba algoritmy jsou téměř nezávislé na hustotě grafu.

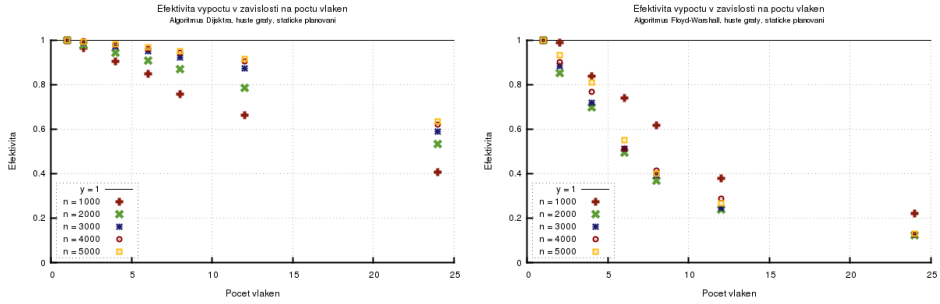
Poměr rychlosti algoritmů v závislosti na plánování Graf 12 zobrazuje poměr výpočetních časů v závislosti na použitém plánování. V porovnání byla použita plánování statické a dynamické. Z grafů je patrné, že



Obrázek 6: Závislost průměrného času výpočtu v závislosti na počtu vláken za použití statického plánování.



Obrázek 7: Závislost zrychlení paralelního algoritmu oproti sekvenčnímu v závislosti na počtu vláken za použití statického plánování.



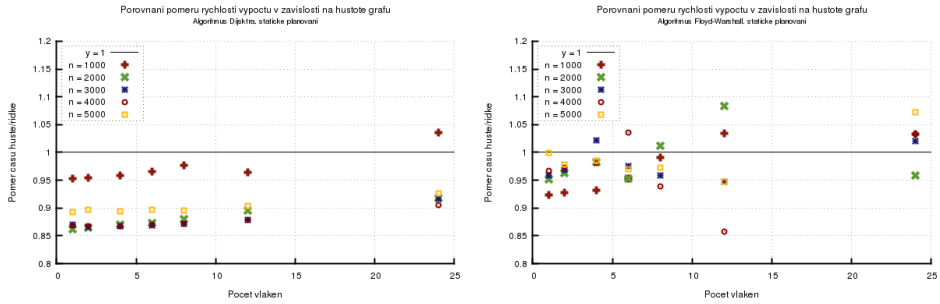
Obrázek 8: Závislost efektivity algoritmu v závislosti na počtu vláken za použití statického plánování.

pocet vlaken	cas	zrychleni	efektivita
1	803.539	1	1
2	400.938	2.00415	1.00207
4	199.811	4.02149	1.00537
6	133.233	6.03109	1.00518
12	67.1254	11.9707	0.99756
8	100.132	8.0248	1.0031
24	46.8958	17.1346	0.71394

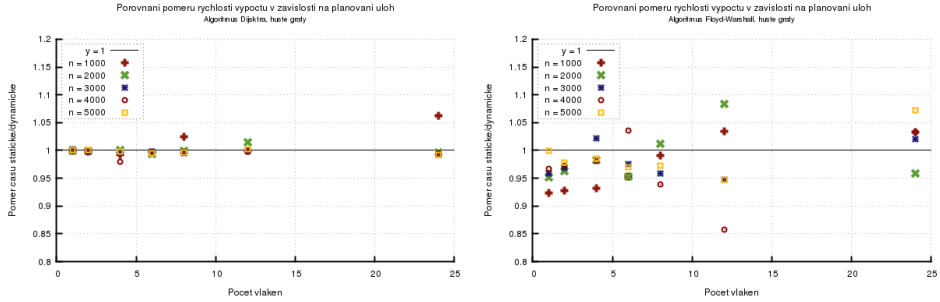
Obrázek 9: Výsledky měření pro Dijkstrův algoritmus pro openMP.

pocet vlaken	cas	zrychleni	efektivita
1	269.878	1	1
2	137.91	1.95691	0.978455
4	72.1159	3.74229	0.935571
6	58.5207	4.61167	0.768612
12	64.9885	4.15271	0.346059
8	57.13	4.72393	0.590492
24	56.5096	4.77579	0.198991

Obrázek 10: Výsledky měření Floyd-Warshallova algoritmu pro openMP.



Obrázek 11: Porovnání poměru rychlostí výpočtu v závislosti na hustotě grafu při použití statického plánování.



Obrázek 12: Porovnání poměru rychlostí výpočtu v závislosti na použitém plánování.

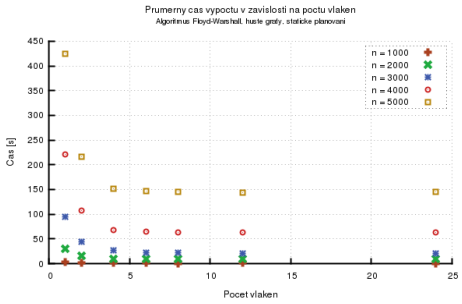
plánování ovlivní čas výpočtu pouze nevýznamně a není tedy podstatné, jaké plánování je v algoritmu použito.

Statické plánování Při statickém plánování se iterace cyklu rovnoměrně rozdělí po blocích mezi všechna vlákna před začátkem provádění cyklu [7].

Dynamické plánování Dynamickým plánováním se rozumí rozdělení jednotlivých iterací vláknům po jedné iteraci. Vždy když vlákno dokončí iteraci, je mu přidělena další iterace [7].

4.4.3 Analýza

Dijkstra Z grafů 6 je vidět klesající dobu výpočtu v závislosti na počtu vláken. Zrychlení, které je zobrazeno v grafu 7 na počátku stoupá téměř



Obrázek 13: Závislost průměrného času výpočtu v závislosti na počtu vláken za použití statického plánování po optimalizaci 4.3.1.

lineárně a teprve pro větší počet vláken se zmírňuje. Toto nelineární zrychlení může způsobit například přístup do paměti, protože se vzrůstajícím počtem vláken klesá relativní velikost cache pro každé z vláken. Z výše uvedeného vyplývá efektivita, která je zobrazena v grafu 8.

Floyd-Warshallův algoritmus U Floyd-Warshallova algoritmu se výpočetní čas pro počty vláken větší než 3 téměř nezkracuje. Zrychlení je tedy patrné pouze při použití 2 případně 4 vláknech. Z výše uvedené plyne, že efektivita paralelního algoritmu velmi rychle klesá.

4.4.4 Optimalizace vláken Floyd-Warshallova algoritmu

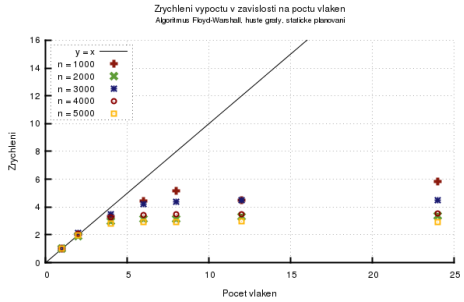
Výsledky měření po implementaci optimalizace 4.3.1 Floyd-Warshallova algoritmu.

4.4.5 Optimalizace odstraněním předchůdců obou algoritmů

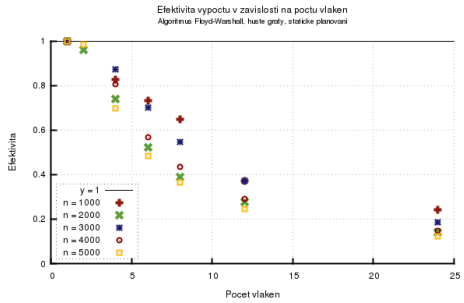
Výsledky měření obou algoritmů po implementaci optimalizace 4.3.2.

4.4.6 Zhodnocení

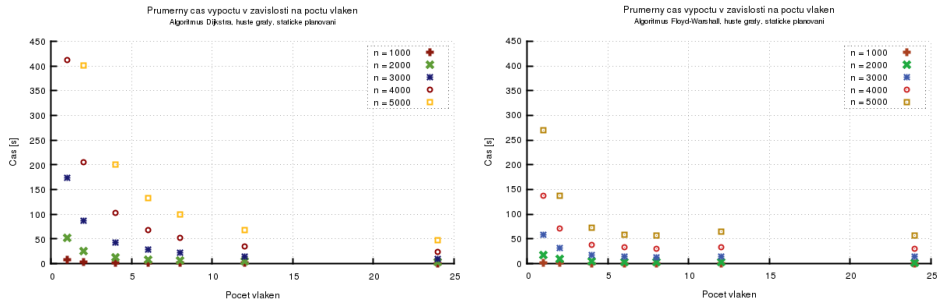
Efektivita Dijkstrova algoritmu s přidáváním vláken pomalu klesá a například pro 24 vláken dosahuje hodnoty 0.5. Naproti tomu u Floyd-Warshallova algoritmu klesá efektivita mnohem rychleji a na hodnotě 0.5 se nachází už pro 6 vláken.



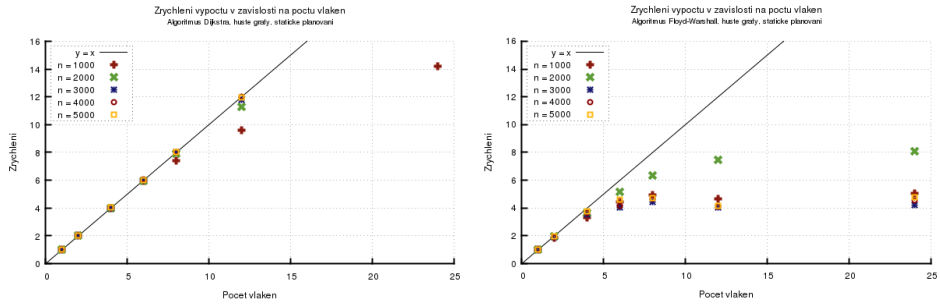
Obrázek 14: Závislost zrychlení paralelního algoritmu oproti sekvenčnímu v závislosti na počtu vláken za použití statického plánovánípo optimalizaci 4.3.1.



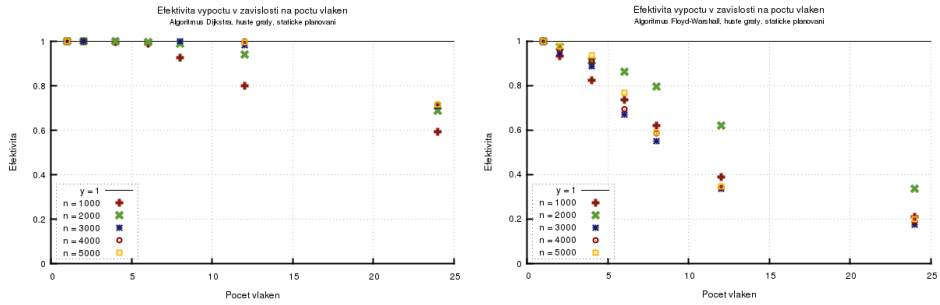
Obrázek 15: Závislost efektivity algoritmu v závislosti na počtu vláken za použití statického plánování po optimalizaci 4.3.1.



Obrázek 16: Závislost průměrného času výpočtu v závislosti na počtu vláken za použití statického plánování po optimalizaci 4.3.2.



Obrázek 17: Závislost zrychlení paralelního algoritmu oproti sekvenčnímu v závislosti na počtu vláken za použití statického plánovánípo optimalizaci 4.3.2.



Obrázek 18: Závislost efektivity algoritmu v závislosti na počtu vláken za použití statického plánování po optimalizaci 4.3.2.

Výrazně lépe ve prospěch Dijkstrova paralelního algoritmu vycházejí i ostatní ukazatele – zrychlení a čas výpočtu.

Výsledky Floyd-Warshallova algoritmu mohou být způsobeny opakovaným vytvářením a rušením vláken. V každém vnějším cyklu se vytvoří daný počet vláken, zpracuje jeden uzel a všechna tato vytvořená vlákna se opět ukončí. Tedy za běhu algoritmu se vytváří a ruší $pocet\_uzlu \times vlaken$, kde parametr *vlaken* je počet najednou vytvářených paralelních vláken.

5 Paralelní algoritmy pomocí technologie CUDA

5.1 Dijkstrův algoritmus

Paralelizace Dijkstrova algoritmu spočívá, stejně jako u technologie openMP 4.1, v paralelizaci cyklu, který prochází jednotlivé uzly a pro ně řeší problém hledání nejkratší cesty v grafu z jednoho počátečního uzlu. Každé vlákno na grafické kartě (GPU) [2] zpracovává jednu instanci s jedním uzlem jako počátečním a z něj hledá nejkratší cesty do všech ostatních uzlů. Na grafické kartě se tedy spouští v součtu N vláken.

Počet vláken Pomocí celočíselného parametru w je možné specifikovat velikost bloku jako počet spouštěných warpů, aby nedocházelo ke zbytečnému snižování efektivity výpočtu[3]. Algoritmus si poté sám dopočte, jaký počet bloků a vláken je potřeba spustit pro daný počet uzlů N.

5.1.1 Paměť

Alokace paměti v programu, který počítá na grafické kartě je rozdílná od klasické alokace v RAM, kterou používá procesor. Pro alokaci paměti na grafické kartě je nutné využívat specializované funkce pro systém CUDA. Pro alokaci pole na grafické kartě je nutné nejprve alokovat pole na příslušnou velikost a poté ještě nakopírovat ukazatel na toto pole do paměti grafické karty. Pro dvojrozměrné pole, které v algoritmu využíváme, je nutné přidat ještě další krok kopírování N ukazatelů a pro každý z nich teprve alokovat příslušné pole.

[3]Řadič grafické karty s technologií CUDA vždy spouští celočíselný počet warpů (1 warp odpovídá 32 vláknům), přičemž nežádoucí vlákna jsou deaktivována. V případě, že je třeba spustit jiný počet vláken, je tak efektivita výkonu grafického akcelerátoru snižována.

Paměťové struktury na CPU V algoritmu je na CPU alokována *matice vzdáleností*, která je použita pro sběr výsledků, kam po skončení algoritmu vlákna z GPU kopírují své výsledky.

Paměťové struktury na GPU Na GPU je alokována a inicializována jedna instance grafu, která je kopií grafu, vytvořeného na CPU. Graf je přístupný všem vláknům a slouží pouze ke čtení, proto stačí pouze jedna instance. Dále je na GPU alokován celý objekt třídy *cDijkstra*, který obsahuje číslo výchozího uzlu, které je pro každé vlákno jiné, počet již uzavřených uzlů, celkový počet uzlů, ukazatel na společný graf a dvě jednorozměrná pole. První je pole *vzdáleností* k ostatním uzlům, které je inicializováno na *nekonečno*. Druhé pole *uzavřený* definuje již uzavřené uzly, ke kterým již byla nalezena nejkratší možná cesta.

5.1.2 Výpočet

Samotný výpočet již zajišťuje samotná třída *cDijkstra*, respektive její metoda *devSpustVypocet()*, která je spuštěna v každém vlákně. Protože vlákno již má všechny potřebné paměťové struktury, může počítat nejkratší cesty v grafu z daného výchozího uzlu, aniž by bylo nutné synchronizovat se s ostatními vlákny.

Kopírování a výpis výsledků Po skončení výpočtu na všech vláknech na GPU následuje kopírování výsledků zpět do paměti RAM tak, aby data byla přístupná z CPU. CPU pak pouze vypíše připravenou matici nejkratších vzdáleností v grafu.

5.1.3 Efektivita využití akcelerátoru

Následující výpočet efektivnosti využití *gpu* byl spočítán pro graf o 5000 uzlech a 1 warp (32 vláken v bloku). Měření bylo prováděno na grafické kartě *GeForce GTX 780 Ti* (*CUDA CC 3.0*), pro kterou platí hardwarové omezení dle tabulky 1.

Spouštěný počet bloků ($\#SB$) se dopočítává tak, aby celkový počet vláken byl roven alespoň N. Pro 32 vláken (počet warpů $\#Wb = 1$) v bloku je tedy $\#SB = 157$.

Vlastnost	Označení	Hodnota
Počet SM	$\#SM$	15
Vlastnosti na 1 SM:		
Počet bloků	$BlMax$	16
Počet warpů	$WarpMax$	64
Počet registrů	$RegMax$	64 k
Velikost sdílené paměti	$ShMax$	48 kB

Tabulka 1: Hardwarové vlastnosti karty GeForce GTX 780 Ti [8]

Celkově se tedy spustí $\#SB.\#Wb.32$ vláken, ale počet vláken provádějících užitečný výpočet je právě N. Efektivita pro počet užitečných vláken je tedy

$$E_1 = \frac{N}{\#SB.\#Wb.32} = 99.5\%$$

Každé vlákno zabírá *18* registrů ($\#reg$) a není využita žádná sdílená paměť. Pro rezidentní počet bloků na jednom SM ($\#B$) platí následující nerovnice:

$$\#B \leq BlMax \Rightarrow \#B \leq 16$$

$$\#B.\#Wb \leq WarpMax \Rightarrow \#B \leq 64$$

$$\#B.\#V.\#reg \leq RegMax \Rightarrow \#B \leq 113.7$$

Z nerovnic tedy vyplývá, že počet rezidentních bloků na jednom SM je 16. Celkový počet rezidentních bloků $\#RB = \#B * \#SM = 240$.

Počet iterací[4] je $\#it = \lceil \frac{\#SB}{\#RB} \rceil = 1$. Efektivita využití SM je tedy

$$E_2 = \frac{\#SB}{\#it.\#RB} = 65.4\%$$

Počet rezidentních warpů na jednom SM je $\#W = \#B.\#Wb = 16$. Což nemusí být dostatečný počet pro prokládání warpů, aby se schovala latence instrukcí.

Celková efektivita je

$$E = E_1.E_2 = 65.1\%$$

[4]V případě, že výpočet všech bloků trvá stejně dlouho. To v tomto případě platí, protože je algoritmus datově necitlivý.

5.2 Floyd-Warshallův algoritmus

V rámci této semestrální práce byly implementovány dvě verze paralelního zpracování Floyd-Warshallova algoritmu. První verze je základní algoritmus, kde je pro každý výpočet relaxace každé hrany[5] voláno jedno vlákno. Dále bylo implementováno optimálnější paralelní řešení Floyd-Warshallova algoritmu – zpracování po dlaždicích, včetně efektivního použití sdílené paměti. Měření bylo provedeno u obou verzí a je shrnuto v sekci 5.3.

5.2.1 Základní paralelní algoritmus

Základní naivní paralelní řešení Floyd-Warshallova algoritmu spočívá v N-krát paralelním spuštění N^2 vláken. Každé vlákno vždy dostane jednu dvojici indexů i a j a má za úkol relaxovat tuto hranu i, j. Pro zadaný počet vláken v bloku roste počet bloků *kvadraticky* s počtem uzlů N, nicméně toto řešení strádá na propustnosti sběrnice ke globální paměti. V každém kroku algoritmu k se snaží N^2 vláken přistoupit ke třem – v drtivé většině případů – různým prvkům matice. I kdyby byly všechny přístupy do paměti *združené*, sběrnice je velmi úzké hrdlo pro tuto verzi algoritmu.

Pro konkrétní implementaci v této práci byla zvolena velikost bloku jako volitelný parametr programu. Tato velikost bloku se zadává jako celočíselný počet warpů (w) a je popsána v kapitole 5.1. Počet bloků se pak dopočítá tak, aby bylo celkově spuštěno minimálně N^2 vláken. Tedy počet bloků $B = \lceil \frac{N^2}{w*32} \rceil$.

5.2.2 Optimalizace pro paralelní výpočet

Optimalizace pro paralelní výpočet na akcelerátoru CUDA byla zpracována dle [1]. Prvním krokem bylo rozdělit výpočet celé matice najednou do výpočtu po (pevně velkých) dlaždicích. Touto optimalizací se zvyšuje prostorová lokalita dat a zvyšuje se tak *cache-hit* pravděpodobnost. V tomto paralelním zpracování se každý blok vláken stará pouze o jednu dlaždici a je tak možné využít sdílenou paměť, kterou mohou společně využít všechna vlákna v jednom bloku.

[5] Relaxace hrany i, j přes k je operace, kde se do matice délky cest uloží menší ze dvou vzdáleností – buď původní cesta z i do j nebo cesta přes k, tedy součet vzdáleností z i do k a z k do j.

Dlaždicové zpracování V tomto způsobu výpočtu je daná matice délek cest rozdělena do pevně velkých dlaždic. Iterování v průběhu algoritmu je pak upraveno podle klasické metody *proložení cyklů* (*loop tiling*) – tedy kde $k1$ iteruje přes počet dlaždic a $k2$ přes velikost dlaždice. Výpočet každé iterace $k1$ je pak dále rozdělen do 3 sekcí – výpočet pro *nezávislé dlaždice*, pro *jedno-závislé dlaždice* a *dvou-závislé dlaždice*.

1. **Výpočet pro nezávislé dlaždice**
   Jedná se o výpočet dlaždice ležící na hlavní diagonále matice. Pro každé $k1$ i $k2$ je mezilehlý uzel k uvnitř této dlaždice a výpočet relaxace všech hran je tedy velmi lokalizovaný. Všechny přístupy do paměti jsou pouze v rámci této jedné dlaždice.

2. **Výpočet pro jedno-závislé dlaždice**
   Tyto dlaždice vždy leží ve stejném řádku (resp. sloupci)[6] jako dlaždice z bodu 1. Pro tyto dlaždice platí, že hrana do

   mezilehlého uzlu k leží v matici z bodu 1. a hrana z uzlu k v jedno-závislé dlaždici samotné (resp. pro sloupce je to přesně naopak). Výpočet je tak velmi lokalizovaný, neboť každý blok čte pouze z dlaždice samotné a z matice z bodu 1.

3. **Výpočet pro dvou-závislé dlaždice**
   Jedná se o výpočet pro všechny ostatní dlaždice nezpracované v předešlých krocích. Pro všechny tyto dlaždice platí, že hodnota hran do i z mezilehlého uzlu k leží v dlaždicích vypočítaných v kroku 2.. Přesněji potřebné hodnoty pro výpočet jedné *dvou-závislé* dlaždice leží v dlaždici samotné a právě ve dvou dlaždicích spočítaných v kroku 2. První, z *hlavního sloupce* je ve stejném řádku a druhá, z *hlavního řádku*, je ve stejném sloupci. Opět je tak výpočet velmi lokalizovaný – pro výpočet jedné dlaždice se opakuje přístup ke stejným hodnotám na malém prostoru v paměti.

Použití sdílené paměti a registrů V případě, že se jeden blok vždy stará právě o jednu dlaždici, je možné iterování přes $k2$ řešit uvnitř kernelu. Pak v případě výpočtu *dvou-závislých* dlaždic každý blok pracuje s daty právě ze tří dlaždic:

[6] Dále nazýváno jako *hlavní řádek* (resp. *sloupec*) dlaždic.

- dlaždice samotné, kde se zapisují nově nalezené hodnoty
- jedna dlaždice z *hlavního sloupce* a jedna dlaždice z *hlavního řádku*, ze kterých se čtou hodnoty pro mezilehlý uzel k

Pro jednu konkrétní hranu i, j a velikost dlaždice s se uvnitř kernelu provádí s čtení a s zápisů prvku i, j matice $(M_{i,j})$. Dále se čtou opakovaně hodnoty z dlaždice z *hlavního sloupce* resp. *řádku* – pro všech s hran z jednoho řádku se čte daná hodnota pro index $k2$ z dlaždice z *hlavního sloupce* a stejně tak pro všech s hran z jednoho sloupce se čte daná hodnota pro index $k2$ z dlaždice z *hlavního řádku*. V průběhu iterování $k2$ se tak čte každá hodnota z dlaždice z *hlavního sloupce* i *řádku* a s-krát.

Jelikož o výpočet pro jednu dlaždici se stará právě jeden blok vláken, je možné využít sdílenou paměť a uštřit tak čas kvůli opakovanému čtení stejný položek z matice. Na začátku kernelu se načtou 2 dlaždice z *hlavního řádku* a sloupce do sdílené paměti a v průběhu výpočtu se pak používají data z této sdílené paměti. Jelikož se tato data používají jen pro čtení, není třeba je zpět zapisovat do hlavní paměti.

Pro optimalizaci algoritmu byly implementovány různé kernely pro pevný počet warpů. Díky tomu je možné dlaždici samotnou – která je kernelem zpracovávána – uložit do registrů. Každé vlákno si na začátku kernelu uloží do vlastních registrů hodnoty $M_{i,j}$, které má na starost a výpočet provádí v registrech. Před ukončením kernelu každé vlákno uloží výsledná data z registrů uloží zpět do hlavní paměti.

Kvůli úpravě algoritmu pro použití registrů jsou implementovány kernely pouze pro počet warpů rovný mocninám dvou.

Fázové zpracování Jak již bylo řečeno v 5.2.2, tak pro výpočet *dvouzávislých* dlaždic se opakovaně čte jedna hodnota z dlaždic z *hlavního řádku* resp. *sloupce*. Konkrétně v iteraci danou hodnotou k se čte sloupec matice $M_{i,k}$ a řádek matice $M_{k,j}$. Tedy při omezení na dlaždicové zpracování se pro výpočet celé dlaždice v iteraci k čte pouze jeden sloupec z dlaždice z *hlavního sloupce* a jeden řádek z dlaždice z *hlavního řádku*. V průběhu iterování přes $k2$ není tedy třeba si udržovat ve sdílené paměti celé dlaždice z *hlavního řádku* resp. sloupce.

Tato skutečnost se dá využít pro fázové načítání dat do sdílené paměti, čímž se radikálně ušetří požadavky na velikost sdílené paměti pro jeden blok. Místo toho, aby se data z dlaždic načetla do sdílené paměti na začátku ker-

```
nvcc -c -D CACHE -D MERENI -D SHARED -arch=sm_30 -rdc=true \
  -O3 -Xptxas -O3 -Xptxas -v -o floydWarshall.o floydWarshall.cu
ptxas info : 0 bytes gmem
...
ptxas info : Compiling entry function \
  '_Z31kernelProDvouZavisleDlazdice128PPjjj' for 'sm_30'
ptxas info : Function properties for \
  _Z31kernelProDvouZavisleDlazdice128PPjjj
  0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info : Used 25 registers, 256 bytes smem, 48 bytes cmem[0]
...
```

Obrázek 19: Výpis z příkazu *make* při použití *Xptxas verbose* módu.

nelu, se načítá potřebný sloupec (resp. řádek) do sdílené paměti v každé iteraci přes $k2$. Pro správný průběh výpočtu a načítání dat v průběhu iterování je potřeba vlákna synchronizovat bariérou v každé iteraci.

5.2.3 Profilace

Výstup 19 ukazuje informace z verbose režimu příkazu *nvcc*. Výstup je zkrácen a jsou ponechány pouze nejdůležitější informace pro kernel pro dvou–závislé dlaždice 3. Každé vlákno z tohoto kernelu používá 25 registrů a každý blok vláken používá 256 B sdílené paměti. Tyto údaje jsou použity pro výpočet efektivity v kapitole 5.2.4.

Obrázek 20 ukazuje, že třetina času algoritmu se počítá ve funkci *kernelProRadky* a dalších téměř 30% ve funkci *kernelProSloupce*. Obě tyto funkce představují výpočet jedno–závislých dlaždic 2, kdy se vypočítává celkem x dlaždic (jeden sloupec nebo řádek dlaždic). Funkce *kernelProDvouZavisleDlazdice*, která je nejvíce optimalizována, trvá přibližně čtvrtinu času algoritmu, ačkoli se provádí pro všechny dlaždice, tedy x^2 dlaždic. Vše napovídá tomu, že funkce *kernelProDvouZavisleDlazdice* je velice dobře optimalizována.

5.2.4 Efektivita využití akcelerátoru

Pro výpočet efektivnosti pro algoritmu Floyd-Warshalla byly použity stejné podmínky jako pro Dijkstrův algoritmus 5.1.3. Jedná se tedy opět o graf

```
======== Profiling result:
Time(%)    Time    Calls    Avg    Min    Max    Name
35.50%  32.382ms  1024  31.622us  30.561us  32.736us
    kernelProRadky(unsigned int**, unsigned int, ...)
28.88%  26.345ms  1024  25.727us  25.057us  27.169us
    kernelProSloupce(unsigned int**, unsigned int, ...)
24.76%  22.584ms    32  705.76us  699.56us  720.91us
    kernelProDvouZavisleDlazdice128(unsigned int**, ...)
5.41%  4.9374ms   2025  2.4380us  1.9520us  4.0000us
    [CUDA memcpy DtoH]
2.74%  2.5038ms   2048  1.2220us     832ns  1.6640us
    [CUDA memcpy HtoD]
2.71%  2.4743ms   1024  2.4160us  2.3360us  3.6800us
    kernelProNezavisleDlazdice(unsigned int**, ...)
```

Obrázek 20: Výpis z příkazu *nvprof* pro Floyd-Warshallův algoritmus při použití *cache* a sdílené paměti.

o 5000 uzlech, ale tentokrát pro 4 warpy (128 vláken v bloku), protože tato hodnota vyšla z měření jako nejlepší dle času výpočtu.

Spouštěný počet bloků ($\#SB$) odpovídá počtu dlaždic matice vzdáleností. Pro 5000 uzlů a velikost dlaždice 32 je to $\lceil \frac{5000}{32} \rceil^2 = 24649$. Počet vláken $\#V$ je tedy 128 (4 warpy).

V každém bloku se vždy spouští přesně 4 warpy ($\#Wb$), ale celkově se matice 5000×5000 musí zarovnat na celočíselný násobek dlaždic, tedy na 5024 a je tak třeba provádět výpočet i pro neexistující prvky. Celkový počet prvků navíc je tedy 5024^2, kde užitečných počet prvků je 5000^2. Efektivita pro počet užitečných vláken je tedy

$$E_1 = \frac{5000^2}{5024^2} = 99.05\%$$

Dle **??** zabírá každé vlákno *25* registrů ($\#reg$) a každý blok *256 B* sdílené paměti $\#ShM$. Pro rezidentní počet bloků na jednom SM ($\#B$) platí následující nerovnice:

$$\#B \leq BlMax \Rightarrow \#B \leq 16$$

$$\#B.\#Wb \leq WarpMax \Rightarrow \#B \leq 16$$

$$\#B.\#V.\#reg \leq RegMax \Rightarrow \#B \leq 20.48$$

$$\#B.\#ShM \leq ShMax \Rightarrow \#B \leq 192$$

Z nerovnic tedy vyplývá, že počet rezidentních bloků na jednom *SM* je 16. Celkový počet rezidentních bloků $\#RB = \#B * \#SM = 240$.

Počet iterací[7] je $\#it = \lceil \frac{\#SB}{\#RB} \rceil = 103$. Efektivita využití *SM* je tedy

$$E_2 = \frac{\#SB}{\#it.\#RB} = 99.7\%$$

Počet rezidentních warpů na jednom *SM* je $\#W = \#B.\#Wb = 64$. To je více než nejdelší latence instrukcí a proto efektivita prokládání warpů je

$$E_3 = 100\%$$

Celková efektivita je

$$E = E_1.E_2.E_3 = 98.8\%$$

5.3 Měření

Pro měření byla použita stejná vstupní data jako pro měření řešení pomocí technologie *OpenMP*. Měření bylo provedeno na *gpu-02*, konkrétně na kartě *GeForce GTX 780 Ti*. Výsledky byly zpracovány a zprůměrovány přes počet uzlů a bylo vypočteno zrychlení oproti sekvenční verzi (algoritmus pro *openMP* s jedním vláknem).

Výsledky porovnání dob běhů jednotlivých implementací jsou zahrnuty v grafech 23, 24 a 25. Zrychlení je zobrazeno na grafech 26, 27. Pro porovnání jsou dvojice grafů vždy ve stejném měřítku.

5.3.1 Výsledky

Tabulky 21 a 22 zobrazují naměřené časy výpočtů jednotlivých algoritmů v závislosti na počtu spuštěných bloků a vláken.

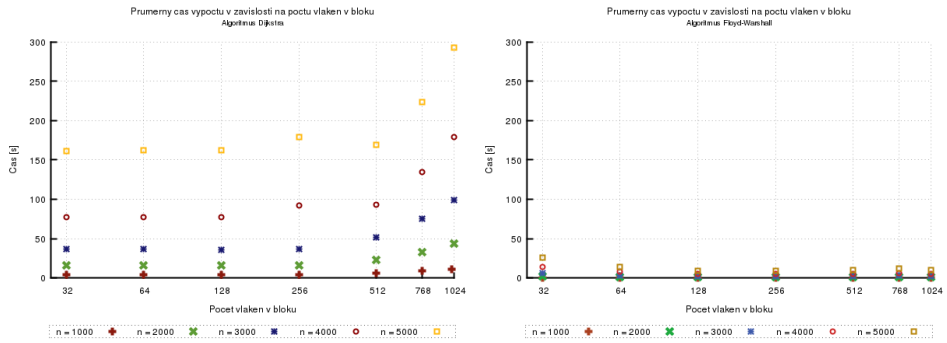
[7] V případě, že výpočet všech bloků trvá stejně dlouho. To v tomto případě platí, protože je algoritmus datově necitlivý.

pocet bloku	vlaken v bloku	cas vypoctu	cas celkovy	zrychleni
79	64	161.39	162.715	4.93833
10	512	168.132	169.44	4.74233
7	768	222.098	223.41	3.59671
20	256	177.866	179.173	4.47
157	32	159.831	161.159	4.98599
40	128	161.095	162.402	4.94783
5	1024	291.225	292.53	2.74686

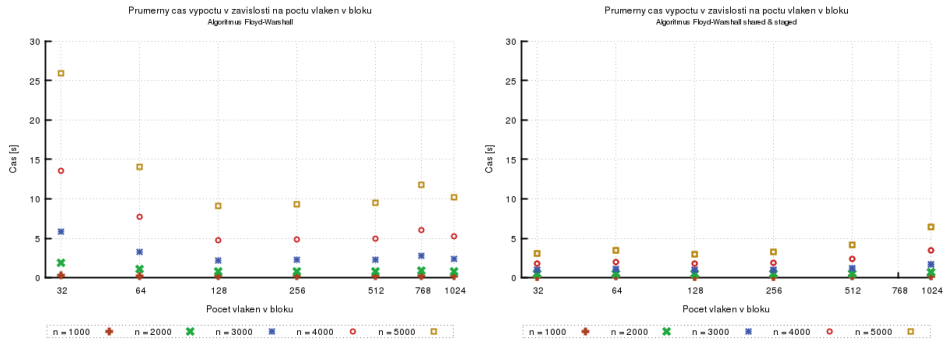
Obrázek 21: Výsledky měření pro Dijkstrův algoritmus.

pocet bloku	vlaken v bloku	cas vypoctu	cas celkovy	zrychleni
24649	1024	5.97198	6.39453	42.2045
24649	256	2.87073	3.29842	81.8204
24649	32	2.682	3.09462	87.2087
24649	128	2.35	3.01689	89.4558
24649	512	3.75256	4.16878	64.7378
24649	64	3.05214	3.46458	77.8962

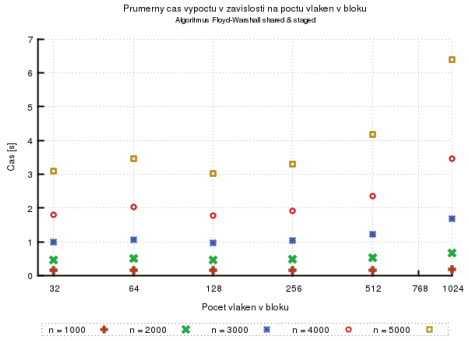
Obrázek 22: Výsledky měření druhé verze Floyd-Warshallova algoritmu.



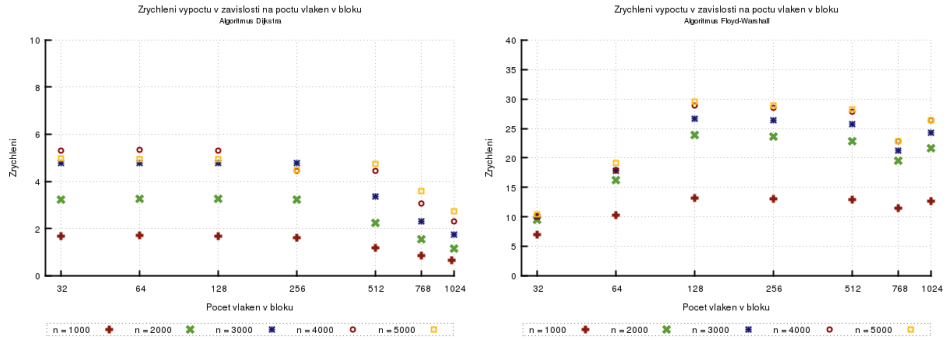
Obrázek 23: Srovnání dob běhu Dijkstrova a základního Floyd-Warshallova algoritmu.



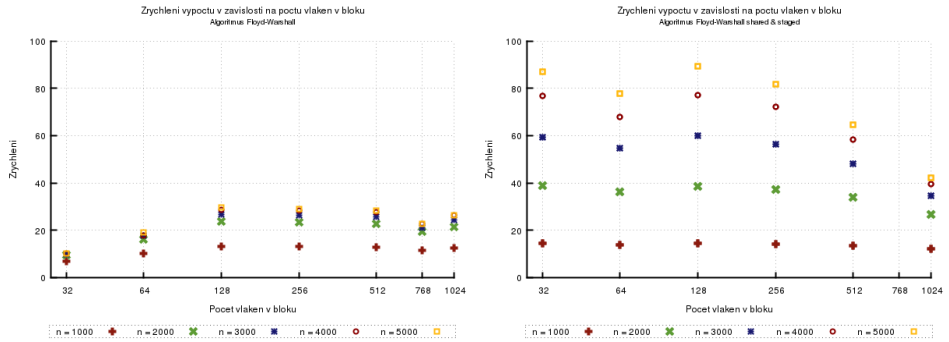
Obrázek 24: Srovnání dob běhu základního a *shared & staged* Floyd-Warshallova algoritmu.



Obrázek 25: Graf doby běhu optimalizovaného *shared & staged* Floyd-Warshallova algoritmu (již není s čím porovnávat).



Obrázek 26: Srovnání dob běhu Dijkstrova a základního Floyd-Warshallova algoritmu.



Obrázek 27: Srovnání dob běhu základního a *shared & staged* Floyd-Warshallova algoritmu.

5.3.2 Vyhodnocení

Graf 23 potvrzuje, že algoritmus Dijkstra je špatně paralelizovatelný pro mnohovláknovou implementaci pomocí technologie *CUDA*. I naivní implementace Floyd-Warshallova algoritmu (kde se jedno vlákno stará o jeden prvek v matici) je řádově rychlejší.

Z grafu 24 je patrné, že optimalizovaná verze Floyd-Warshallova algoritmu zrychluje výpočet průměrně 2–3 krát. Společně s výstupy z profilace (5.2.3) je vidět, že se efektivně využívá sdílená paměť a registry na *multiprocesorech*.

6 Závěr

Maximální sedmnáctinásobné zrychlení bylo dosaženo, při použití technologie *openMP*, pro dvacet čtyři vláken. Implementace Dijkstrova algoritmu na technologii *CUDA* dosahuje maximálně přibližně *pětinásobného* zrychlení proti sekvenčnímu řešení. Při masivní paralelizaci (počet vláken rovný N) je sběrnice do globální paměti úzkým hrdlem a proto nebylo dosaženo výrazně velkého zrychlení než pětinásobného.

OpenMP dosahuje při použití šesti vláken přibližně pětinásobného zrychlení. Pro větší počet použitých vláken se již čas běhu nesnižuje. Naproti tomu optimalizovaný Floyd-Warshallův algoritmus na technologii *CUDA* dosahuje oproti sekvenční verzi až *devadesátinásobného* zrychlení. Tohoto zrychlení bylo dosaženo při použití asymptoticky N^2 počtu vláken. Efektivita paralelního výpočtu pomocí *openMP* strádá hlavně na tom, že nebyl použit blokový

algoritmus, který byl implementován pro řešení technologií *CUDA* 5.2.2.

V další práci je možné v optimalizovaném Floyd-Warshallově algoritmu optimalizovat pomocí sdílené paměti a registrů funkce *kernelProRadky* a *kernelProSloupce* podobným způsobem, jako byla optimalizována funkce *kernelProDvouZavisleDlazdice*.

Reference

[1] Ben Lund, J. W. S.: A Multi-Stage CUDA Kernel for Floyd-Warshall. 2010, Cincinnati (Ohio, United States): University of Cincinnati. [cit. 2015-05-13].
URL http://arxiv.org/pdf/1001.4108.pdf

[2] Falcus, D.: Graphics Processing Unit (GPU). [cit. 2015-05-11].
URL http://www.nvidia.com/object/gpu.html

[3] Foster, I.: Case Study: Shortest-Path Algorithms. 1995, [cit. 2015-04-13].
URL http://www.mcs.anl.gov/~itf/dbpp/text/node35.html

[4] Mička, P.: Dijkstrův algoritmus. [cit. 2015-04-08].
URL http://www.algoritmy.net/article/5108/Dijkstruv-algoritmus

[5] Mička, P.: Floyd-Warshallův algoritmus. [cit. 2015-04-08].
URL http://www.algoritmy.net/article/5207/Floyd-Warshalluv-algoritmus

[6] Mička, P.: Problém nejkratší cesty. [cit. 2015-04-08].
URL http://www.algoritmy.net/article/36597/Nejkratsi-cesta

[7] Šimeček, I.: Technologie OpenMP. 2014, [cit. 2015-04-14].
URL https://edux.fit.cvut.cz/courses/MI-PAP/_media/lectures/omp.pdf

[8] Šimeček, I.: CUDA Compute capabilities. 2015, [cit. 2015-05-13].
URL https://edux.fit.cvut.cz/courses/MI-PRC/_media/lectures/cc.pdf

[9] Šimeček, I.; Šoch, M.: Použití vektorizace v C/C++. 2015, [cit. 2015-04-13].
URL https://edux.fit.cvut.cz/courses/MI-PAP/_media/lectures/vektorizace.pdf