Semestrální projekt MI-PAP, MI-PRC 2014/2015 Násobení matic

Karel Fiala

České vysoké učení technické v Praze
Fakulta informačních technologií
Thákurova 9, 160 00 Praha 6
Česká republika

13. května 2015

Obsah

1 Definice problému a popis sekvenčního algoritmu

1.1 Definice problému

Násobení matic pomocí klasického algoritmu a rekurzivního Strassenova algoritmu.

1.2 Popis sekvenčního algoritmu a jeho implementace

1.2.1 Klasický multiplikační algoritmus

Násobíme matici B maticí A tak, že výsledný prvek matice C je součtem součinů čísel v řádku matice A s příslušným číslem ve sloupci matici B.

Výpočetní náročnost klasického algoritmu je $O(n^3)$.

Klasický multiplikační algoritmus je implementován třemi vnořenými cykly, kde první dva určují řádek a sloupec výsledné matice a třetí tvoří součet součinů prvků násobených matic.

Algoritmus je časově i paměťové stabilní.

1.2.2 Strassenův rekurzivní algoritmus

Strassenův rekurzivní algoritmus je asymptoticky rychlejší než klasický multiplikační algoritmus. Využívá pouze 7 násobení místo 8 jako je tomu u klasického multiplikačního algoritmu.

Strassenův rekurzivní algoritmus pracuje se složitostí $O(n^{\log_2 7}) \approx O(n^{2.807})$.

V mé implementaci se pro matice řádu 500 a nižší použije klasický multiplikační algoritmus, který je efektivnější.

2 OpenMP

2.1 Popis implementace

2.1.1 Klasický multiplikační algoritmus

V podstatě bez úprav.

Paralelizace spočívá ve vložení následujícího řádku před první cyklus.

```
#pragma omp parallel for shared(C) private(i) schedule(dynamic,10)
```

Dělení práce po 10ti „kusech“ snižuje režii v případě velkých vstupních matic. Jeden řádek matice je zpracován velmi rychle.

2.1.2 Strassenův rekurzivní algoritmus

Úprav v celém programu je opět minimálně.

Místo přímého volání Strassenova rekurzivního algoritmu je zavolán „wrapper“, který se postará o rozdělení dvou vstupních matic na $2 * 16$ podmatic. Jednotlivá vlákna poté volají Strassenův rekurzivní algoritmus na násobení dvou menších matic. Těchto volání je 64, protože každý prvek výsledné matice je složen ze 4 součinů podmatic.

Takto implementovaný algoritmus tedy dokáže využít maximálně 64 procesorových jader.

64 podmatic je poté paralelně redukováno na 32 podmatic a poté opět paralelně redukováno na 16 výsledných podmatic. Tyto matice jsou již částmi chtěného výsledku.

2.2 Použití vektorizace

2.2.1 Klasický multiplikativní algoritmus

```
src/simd_trivial.cpp:7: note: vectorized 0 loops in function.
src/main.cpp:17: note: vectorized 0 loops in function.
src/main.cpp:33: note: vectorized 0 loops in function.
src/main.cpp:50: note: vectorized 0 loops in function.
src/main.cpp:78: note: vectorized 0 loops in function.
src/main.cpp:103: note: vectorized 0 loops in function.
src/main.cpp:149: note: vectorized 0 loops in function.
```

2.2.2 Strassenův rekurzivní algoritmus

```
src/simd_strassen.cpp:353: note: vectorized 1 loops in function.
src/simd_strassen.cpp:347: note: vectorized 1 loops in function.
src/simd_strassen.cpp:367: note: vectorized 4 loops in function.
src/main.cpp:17: note: vectorized 0 loops in function.
src/main.cpp:33: note: vectorized 0 loops in function.
src/main.cpp:50: note: vectorized 0 loops in function.
```

```
src/simd_strassen.cpp:8: note: vectorized 0 loops in function.
src/simd_strassen.cpp:13: note: vectorized 0 loops in function.
src/simd_strassen.cpp:29: note: vectorized 1 loops in function.
src/simd_strassen.cpp:41: note: vectorized 1 loops in function.
src/simd_strassen.cpp:53: note: vectorized 1 loops in function.
src/simd_strassen.cpp:68: note: vectorized 1 loops in function.
src/simd_strassen.cpp:80: note: vectorized 0 loops in function.
src/simd_strassen.cpp:90: note: vectorized 30 loops in function.
src/simd_strassen.cpp:330: note: vectorized 0 loops in function.
src/simd_strassen.cpp:191: note: vectorized 12 loops in function.
src/main.cpp:78: note: vectorized 0 loops in function.
src/main.cpp:103: note: vectorized 0 loops in function.
src/main.cpp:149: note: vectorized 0 loops in function.
src/main.cpp:171: note: vectorized 0 loops in function.
```

2.3 Popis optimalizací pro dosažení lineárního zrychlení

Vše bylo kompilováno s parametrem $-O3$.

2.3.1 Klasický multiplikační algoritmus

Žádné speciální optimalizace.

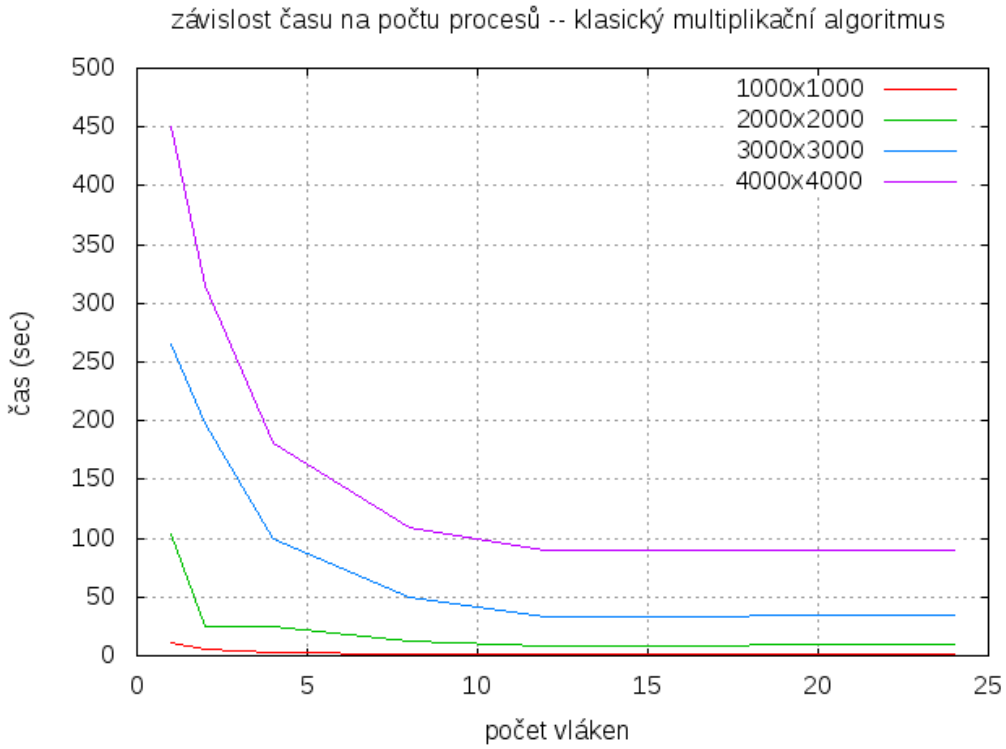
2.3.2 Strassenův rekurzivní algoritmus

Snaha o zmenšení režie při práci s podmaticemi, např. pomocí paralelizace redukce.

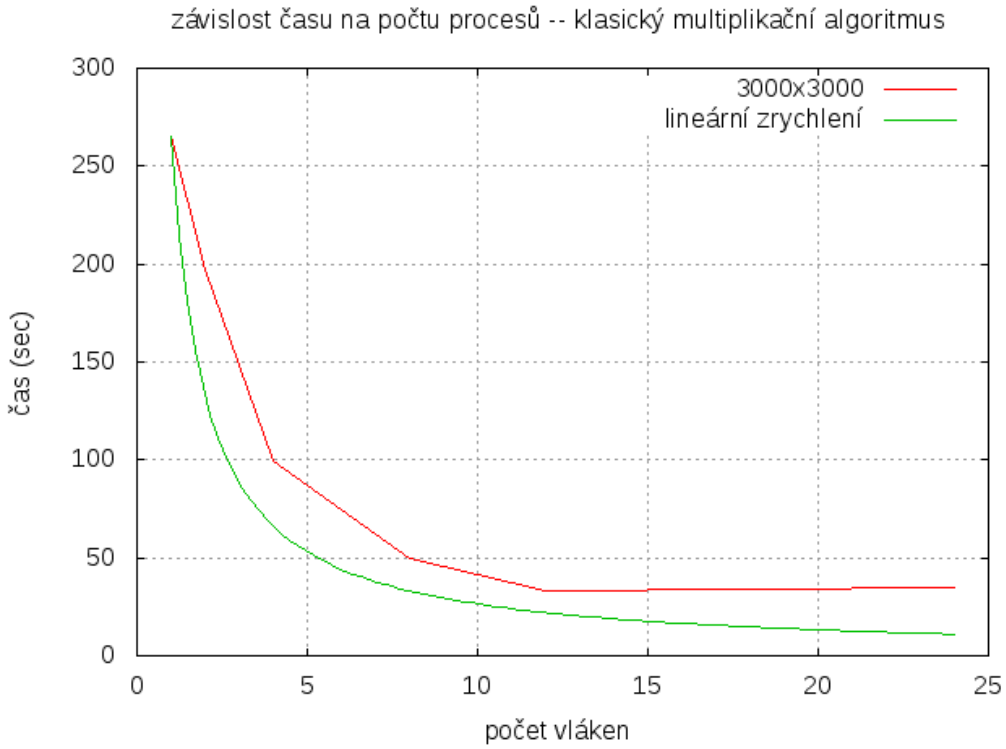
2.4 Naměřené výsledky

2.4.1 Klasický multiplikační algoritmus

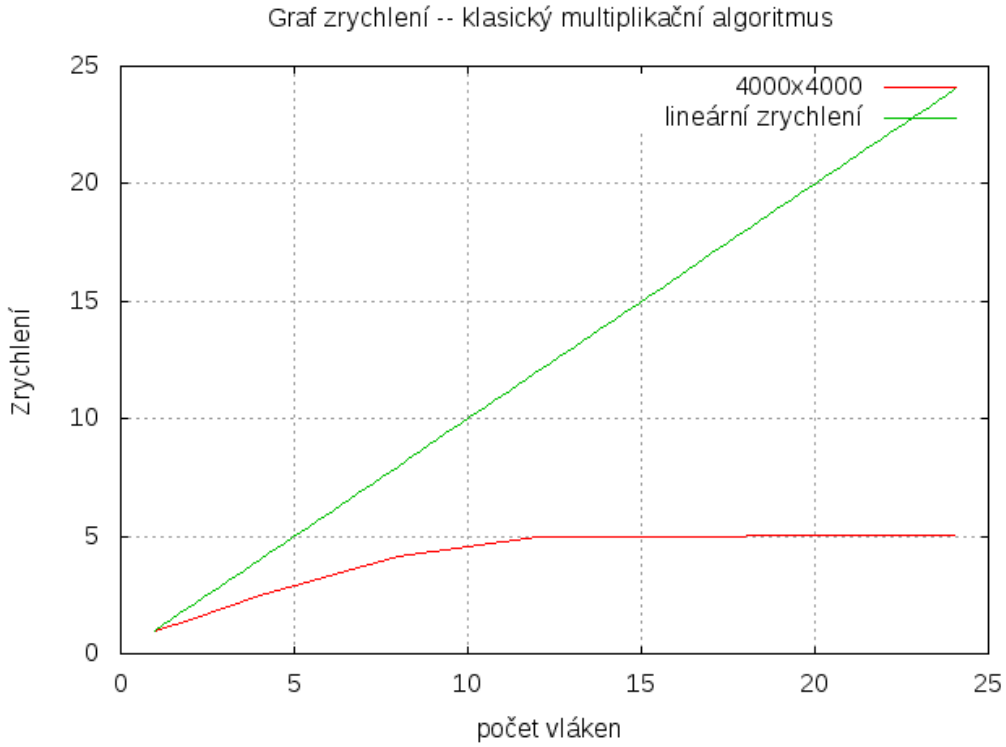
#CPU	1000	2000	3000	4000
1	11.365326	103.732902	264.885942	450.300757
2	5.683157	25.049751	197.348525	314.536432
4	2.847626	25.096038	99.390237	180.675981
8	1.486099	12.946192	49.909194	108.842585
12	1.034171	8.721059	32.952173	90.281103
24	1.031660	9.081376	34.469956	89.329224



Obrázek 1: Klasický multiplikační algoritmus



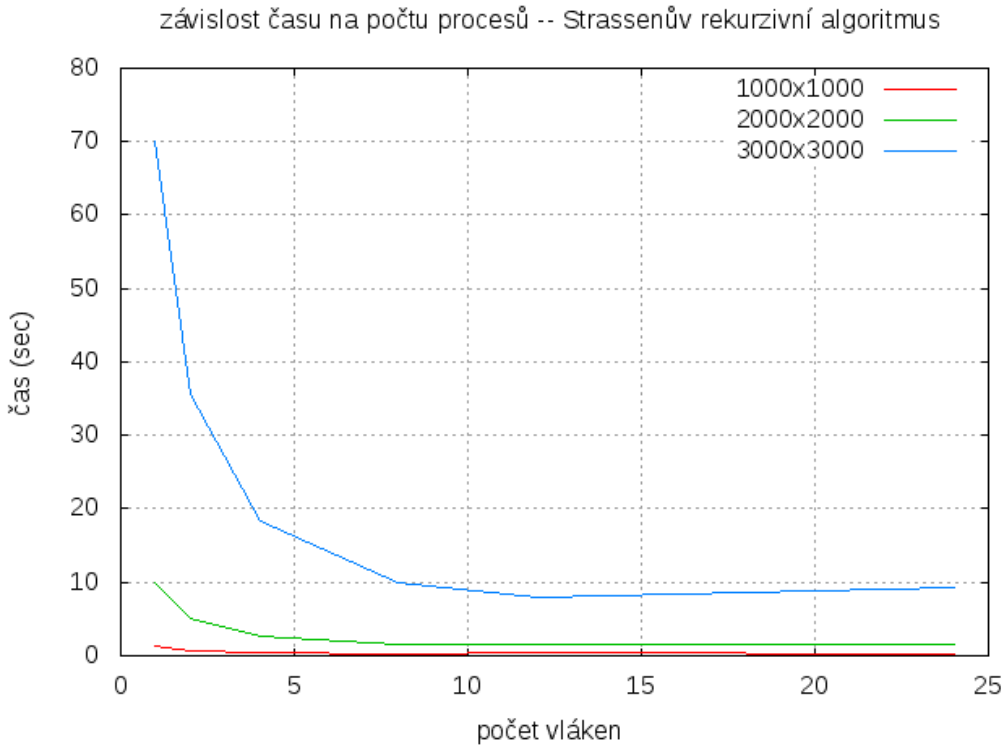
Obrázek 2: Klasický multiplikační algoritmus pro matici 3000x3000



Obrázek 3: Graf zrychlení klasického multiplikačního algoritmu pro matici 4000x4000

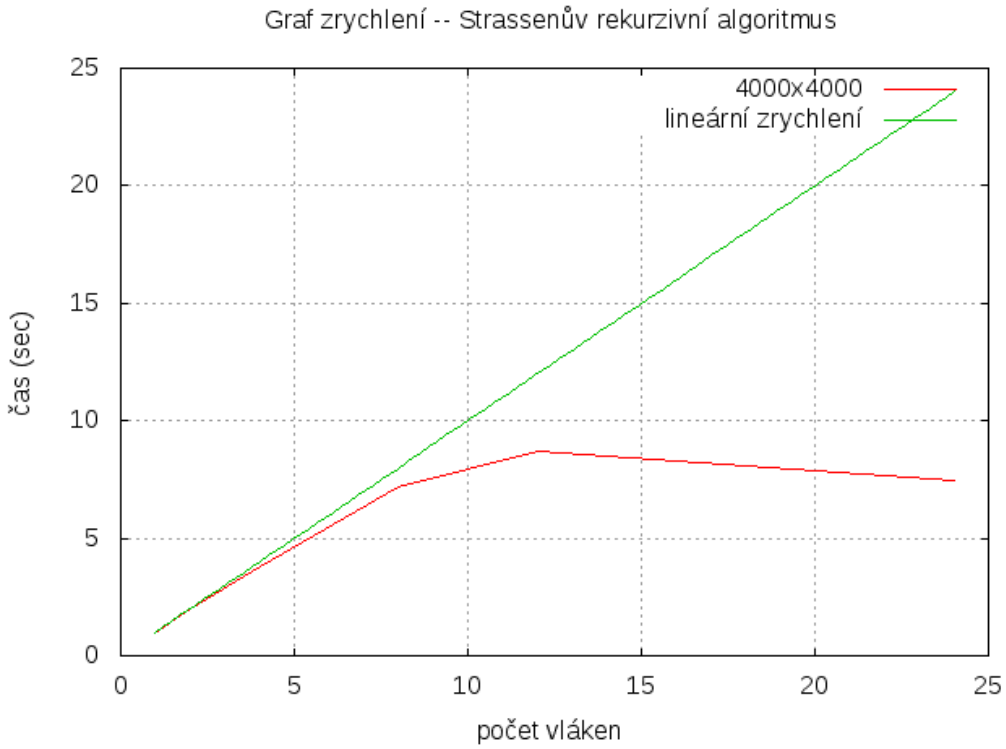
2.4.2 Strassenův rekurzivní algoritmus

#CPU	1000	2000	3000	4000
1	1.387058	9.859318	70.110456	70.087705
2	0.732522	5.096552	35.600932	35.565978
4	0.391072	2.666515	18.332472	18.290037
8	0.297135	1.605526	9.874875	9.713100
12	0.343952	1.643614	7.973451	8.023352
24	0.253875	1.489265	9.384766	9.362794



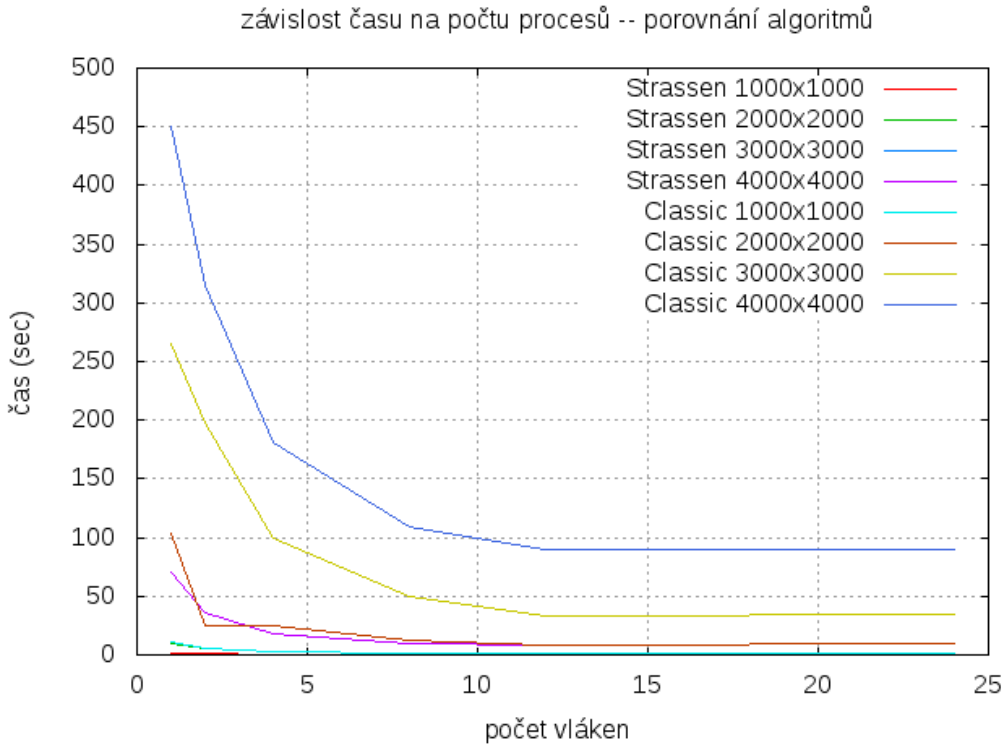
Obrázek 4: Strassenův rekurzivní algoritmus

Obrázek 5: Strassenův rekurzivní algoritmus pro matici 4000x4000

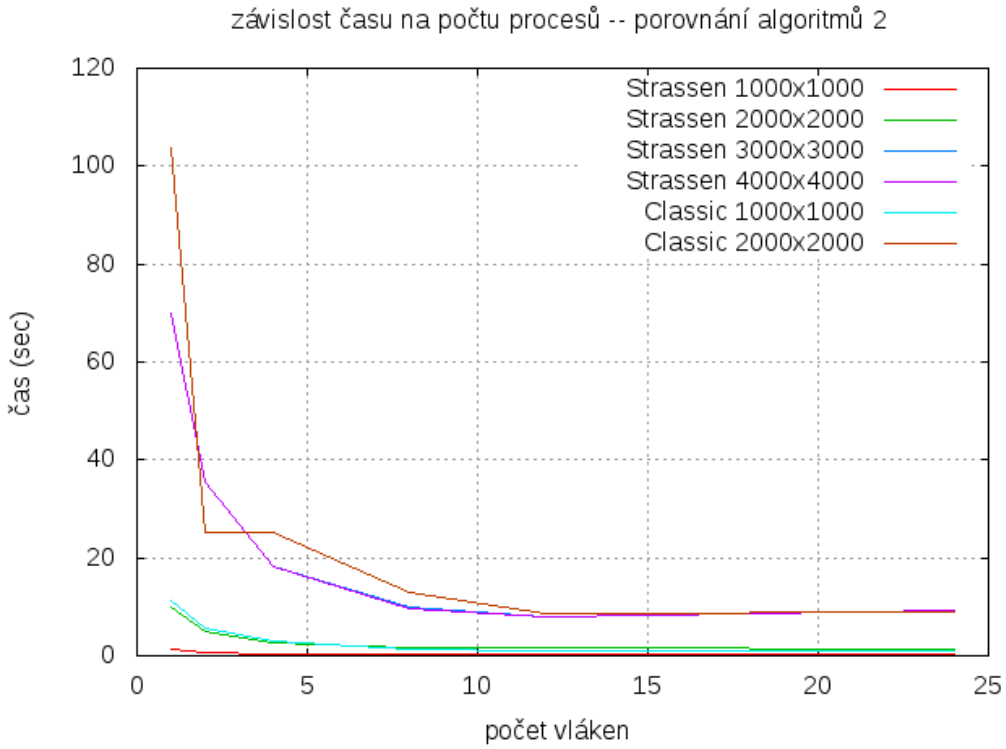


Obrázek 6: Graf zrychlení – Strassenův rekurzivní algoritmus pro matici 4000x4000

2.4.3 Porovnání algoritmů

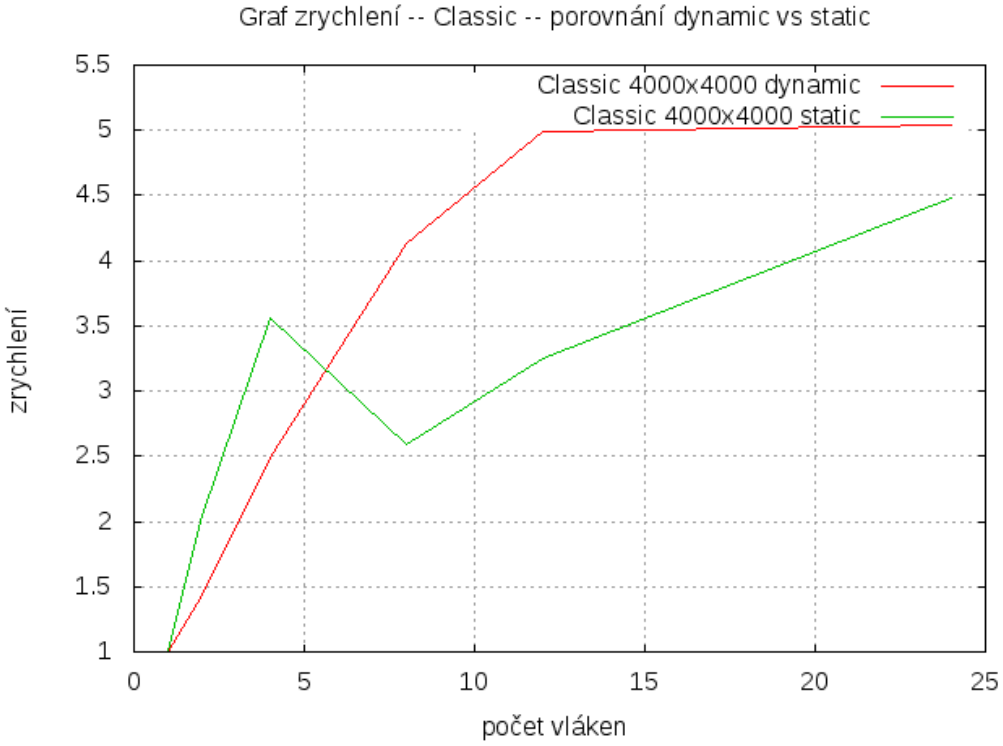


Obrázek 7: Porovnání obou algoritmů

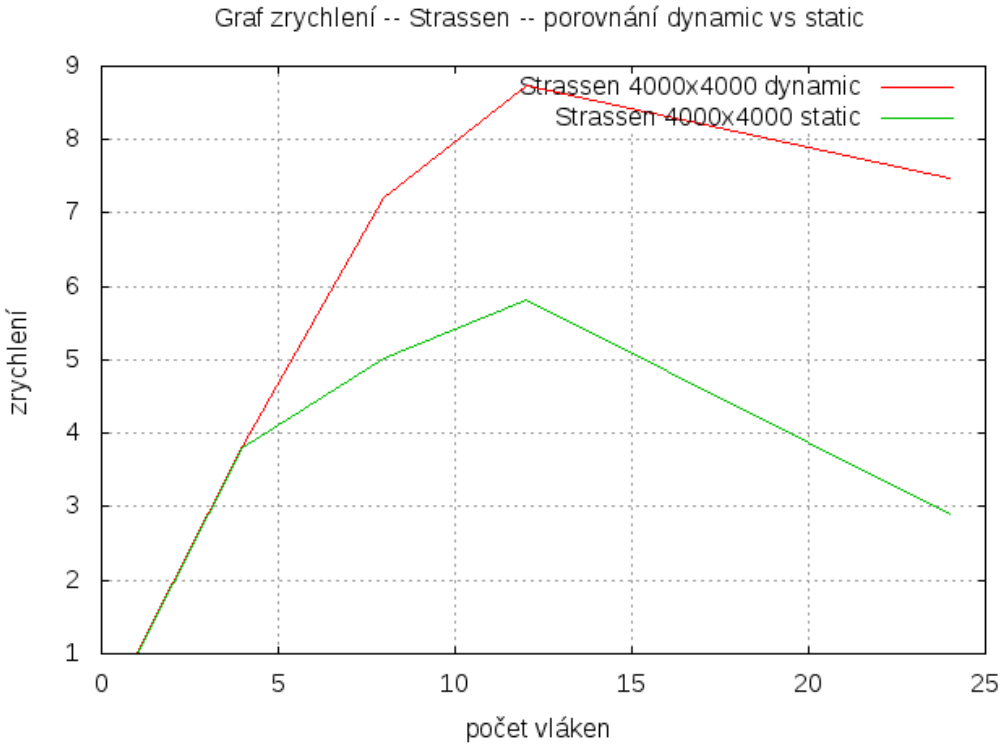


Obrázek 8: Porovnání obou algoritmů 2

2.4.4 Porovnání plánování



Obrázek 9: Porovnání plánování dynamic vs static



Obrázek 10: Porovnání plánování dynamic vs static

2.5 Analýza a hodnocení vlastností dané implementace programu

Byla naměřena relativně velká odchylka měření a podivná nekonzistence v jednotlivých hodnotách.

Zrychlení nad 12 jader není žádné. Předpokládám, že testovací server má 12 fyzických jader (a HyperThreading).

2.5.1 Klasický multiplikační algoritmus

Není dosaženo lineárního zrychlení.

Implementovaný algoritmus je časově i paměťové stabilní a předvídatelný.

2.5.2 Strassenův rekurzivní algoritmus

Je dosaženo téměř lineárního zrychlení, ale jen do 12ti výpočetních jader. Výsledky jsou k mému překvapení lepší než u klasického multiplikačního algoritmu.

Implementovaný algoritmus je paměťové velmi náročný a pro velmi, velmi velké vstupy by byl nepoužitelný. Mnoho času zabere režie s pamětí a mnoho volání.

Nalezené řešení implementovaného algoritmu obsahuje chybu, která způsobuje drobné odchylky od reality. Chyba je stálá a není závislá na velikosti vstupu, času či míře paralelizace.

2.5.3 Dynamické vs statické plánování

Pro klasický multiplikační algoritmus je průběh zrychlení pro dynamické i statické plánování zcela odlišný. Statické plánování zde dosahuje menšího zrychlení, přestože zrychlení roste pro větší množství vláken než u dynamického plánování.

Pro Strassenův rekurzivní algoritmus je průběh zrychlení pro dynamické i statické plánování stejný, avšak dynamické plánování dosahuje většího zrychlení.

3 CUDA

3.1 Popis případných úprav algoritmu a jeho implementace

Základ celého programu zůstal téměř nezměněn. Přidal jsem soubor *simt_trivial.cu*, který obsahuje celou implementaci (včetně měření) klasického multiplikačního algoritmu pomocí CUDA technologie.

Vstupní a výstupní matice jsou v C reprezentovány pomocí 2D pole. Při kopírování matice z paměti procesoru do paměti grafické karty (a naopak) proběhne transformace matice na 1D pole.

Všechny výpočty jsou v celočíselné.

3.2 Popis optimalizací

Snaha o saturaci každého bloku 1024 vlákny, tedy využití všech 32 warpů.

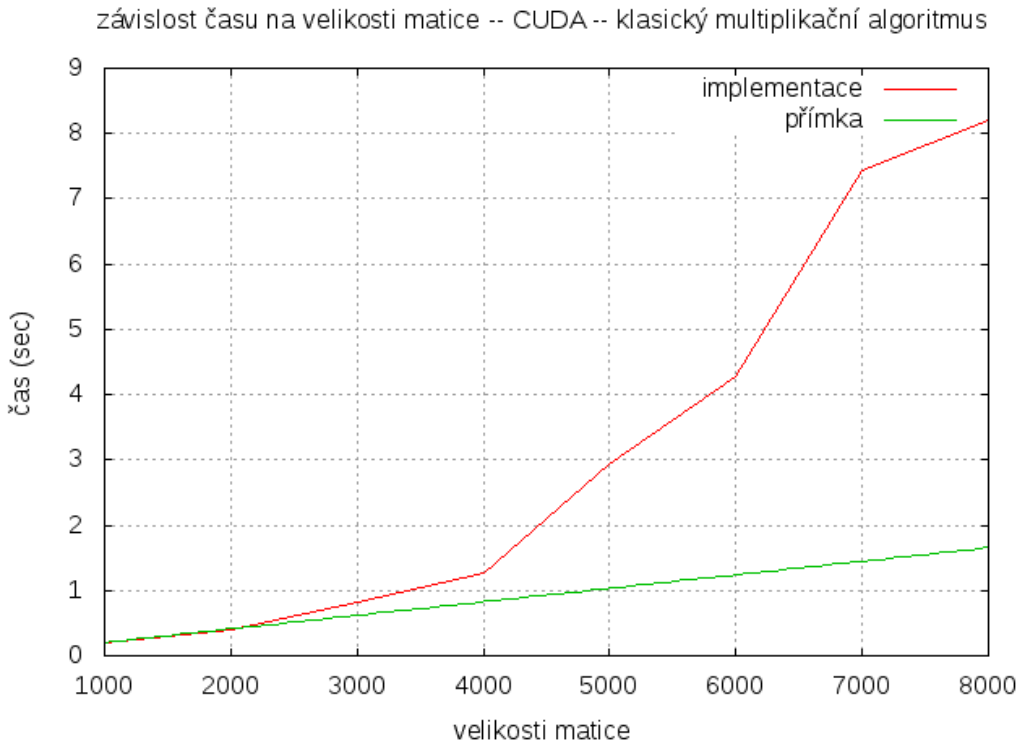
V implementaci jsem nepoužil *loop unrolling*[1].

3.3 Naměřené výsledky

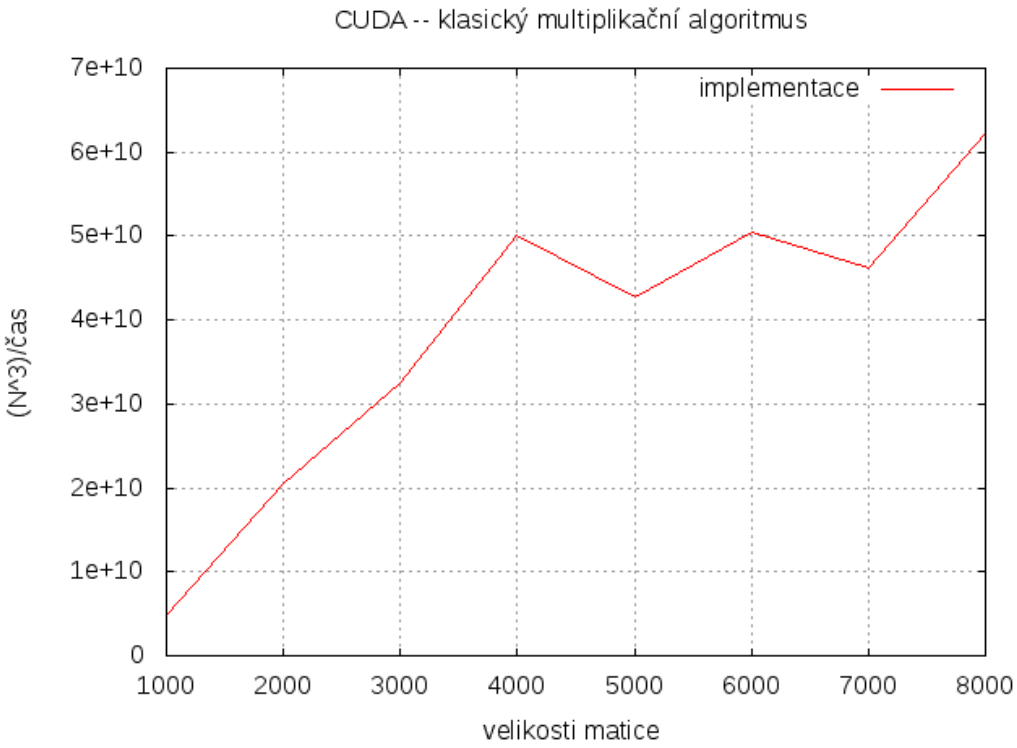
N je velikosti vstupních matic.

N	čas
1000	0.20681
2000	0.39094
3000	0.82996
4000	1.27902
5000	2.92957
6000	4.28045
7000	7.42563
8000	8.21492

[1]Rozbalení cyklů – např. počet průchodů cyklu zmenšíme na polovinu a v každém průchodu uděláme dvě operace najednou. Zmenšíme tak režii samotného cyklu vzhledem k potřebnému výpočtu.



Obrázek 11: Klasický multiplikativní algoritmus – CUDA implementace



Obrázek 12: Klasický multiplikativní algoritmus – CUDA implementace

3.4 Analýza a hodnocení vlastností dané implementace programu

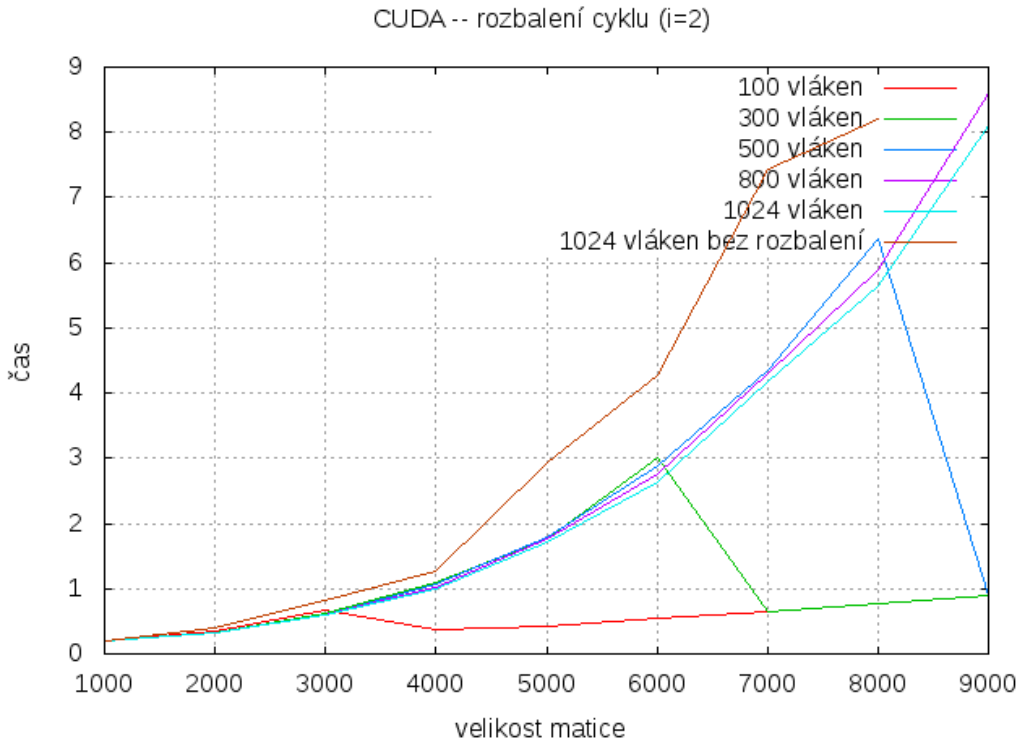
Tato implementace algoritmu je násobně rychlejší než předchozí implementace na CPU. Zrychlení je velké.

Algoritmus není efektivní ohledně přístupu do paměti. Zde by pomohlo například použití *loop unrolling* nebo algoritmus přepracovat tak, aby uměl vhodně využít sdílenou paměť.

4 CUDA podruhé

4.1 Klasický multiplikativní alg. – rozbalení cyklu

Jednou z možných optimalizací je rozbalení cyklů a tím snížení režie. Na následujícím grafu je vidět snížení výpočetního času o $\approx 25\%$ při rozbalení cyklu pro $i = 2$.



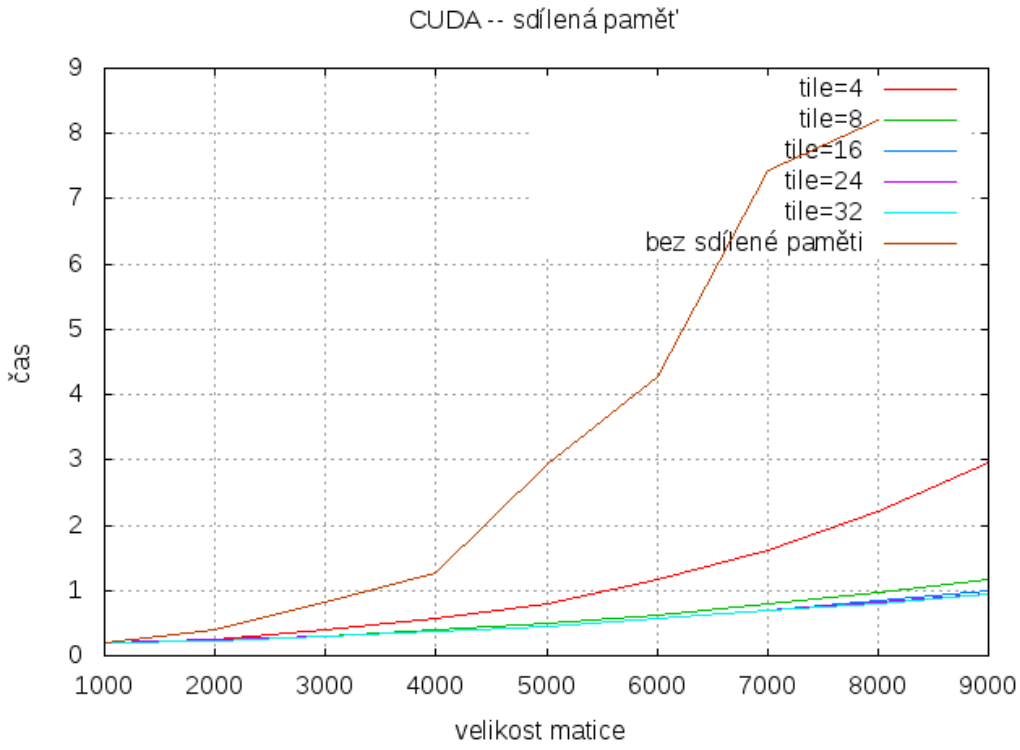
Obrázek 13: Rozbalení cyklu pro $i = 2$

4.2 Klasický multiplikativní alg. – použití sdílené paměti

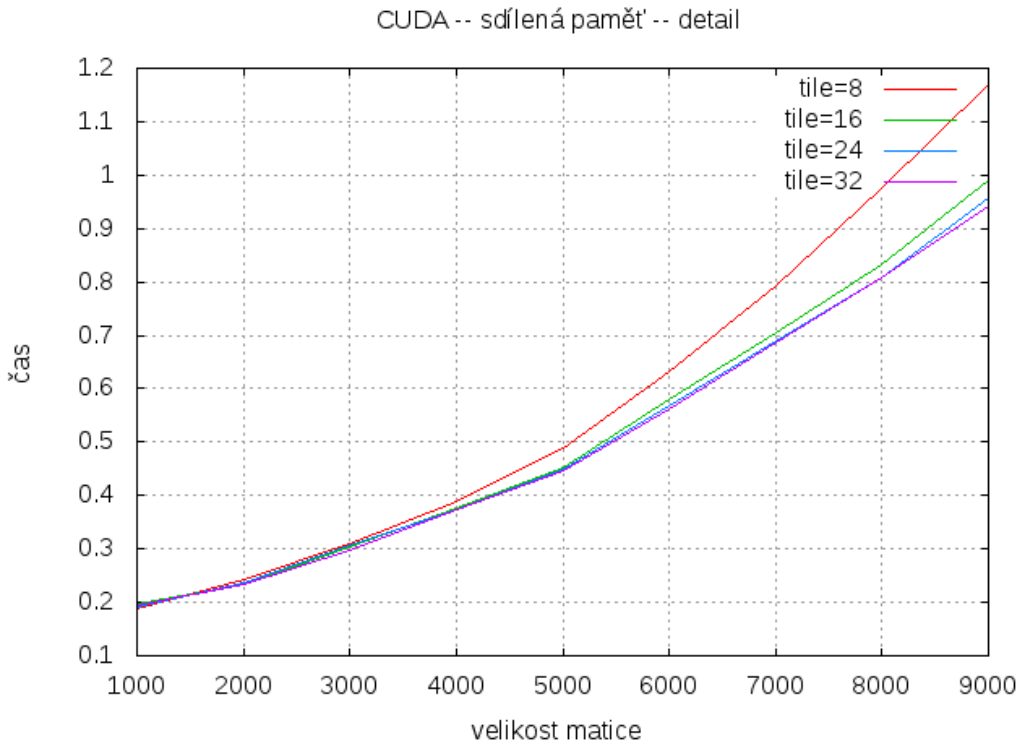
Další z možných optimalizací je použití sdílené paměti a tím snížení latence, která vzniká při přistupu do globální paměti. Na následujícím grafu je vidět snížení výpočetního času na $\approx 1/4$.

To velmi předčilo mé očekávání. Zároveň je vidět, že při použití sdílené paměti mají naměřené výsledky menší odchylku měření – „čáry v grafu jsou plynulejší“.

Na dalším obrázku je pak vidět detailnější rozdíl mezi „velikosti sdíleného okénka“ pro velikosti 8, 16, 24 a 32.



Obrázek 14: Použití sdílené paměti



Obrázek 15: Použití sdílené paměti – detail

4.3 Strassenův algoritmus v CUDA

Implementovaný algoritmus bych nazval „pseudo-Strassenův algoritmus“. Poprvé rozdělí matice po vzoru Strassena a na GPU je vypočte s pomocí klasického multiplikativní algoritmu.

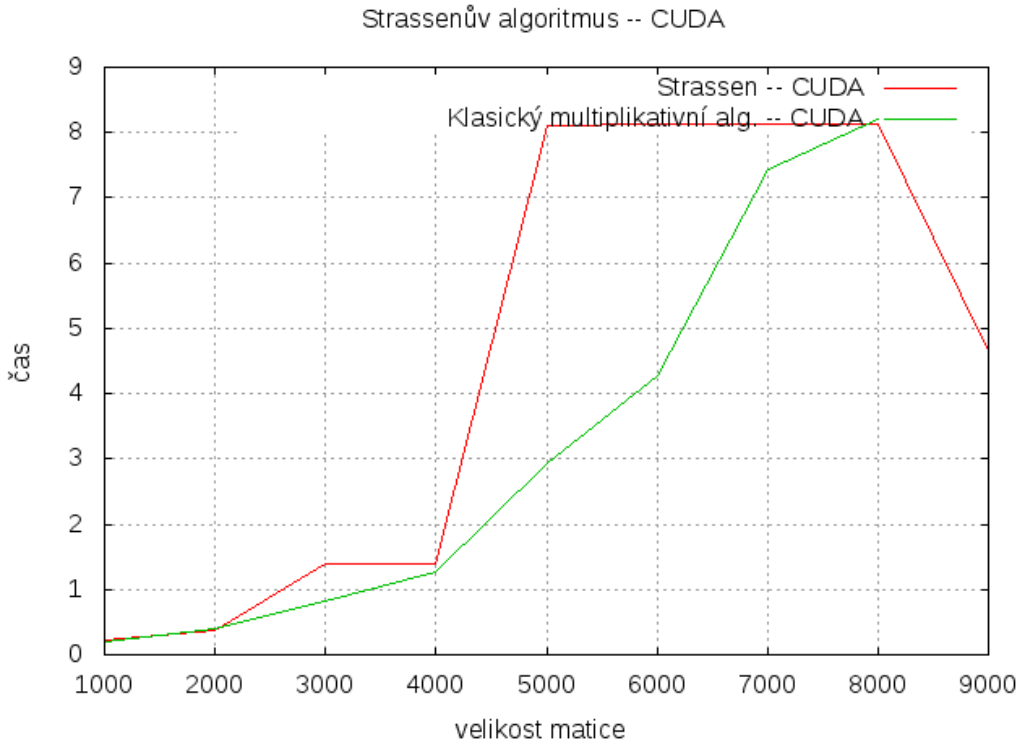
Snažil jsem se většinu operací realizovat na GPU – násobení (využil jsem již implementovaný klasický multiplikativní algoritmus), sčítání a odčítání matic.

4.3.1 Popis činnosti implementace

- dvě vstupní matice na CPU rozdělí na čtvrtiny
  – tedy má $4 + 4$ podmatice
- tyto podmatice nahraje do paměti GPU a na CPU je smaže
- inicializuje a alokuje prostor pro pomocné matice výpočtu
  – jejich 17
- na GPU naplní pomocné matice pomocí *add_gpu()* a *sub_gpu()*
- tyto pomocné matice pronásobí po vzoru Strassena, ale již klasickým multiplikativním algoritmem *mul_gpu()*
- částečné výsledky opět pomocí *add_gpu()* a *sub_gpu()* sečte/odečte
- čtyři výsledné podmatice nahraje zpět k CPU a spojí do jedné výstupní matice

4.3.2 Výkonost

Algoritmus nevyužívá sdílenou paměť ani rozbalení cyklů. Také se zcela jistě projevuje zdržení při nevhodném volání kernelu (celkem 25-krát) a následná bariéra *cudaThreadSynchronize()*.



Obrázek 16: Porovnání implementovaných algoritmů v CUDA

5 Závěr

5.1 Porovnání výkonnosti všech tří verzí

Dle očekávání je nejrychlejší řešení pomocí CUDA technologie. Dále pak Strassenův algoritmus na CPU, který je ale v mé implementaci paměťově velmi náročný a pro velmi, velmi velké vstupy by byl nepoužitelný. Klasický multiplikační algoritmus na CPU podává nejslabší výkon, ale je velmi dobře předpovídatelný a paměťově nenáročný. Jeho implementace by mohla běžet i na velmi slabém stroji.