Násobení Čtvercových Matic

Martin Stahl  
NI-MCC

Obsah

[Definice problému 3](#_Toc121077058)

[Sekvenční algoritmus 3](#_Toc121077059)

[Klasické násobení 3](#_Toc121077060)

[Strassenův algoritmus 4](#_Toc121077061)

[Implementace 4](#_Toc121077062)

# Definice problému

V rámci semestrální práce jsem měl za úkol implementovat řešení problému násobení čtvercových matic.

Nechť **A** a **B** jsou čtvercové matice nad okruhem *R*, a nechť **C** je jejich součin, tj.



V mém případě pro implementace algoritmů předpokládáme, že vstupní a výstupní matice jsou řádu 2n. V praxi se také pracuje s maticemi řádu 2n tak, že se zvolené matice, které nejsou řádu 2n doplní nulami tak, aby měly řád 2n.

# Sekvenční algoritmus

Dle zadání jsem implementoval dva algoritmy. Klasické násobení a Strassenův algoritmus.

## Klasické násobení

Klasický naivní sekvenční algoritmus kopíruje standartní postup násobení matic. Toto je naimplementováno pomocí 3 vnořených for-cyklů

for (int i = 0; i < matC.size; i++){  
 for (int j = 0; j < matC.size; j++){  
 sum = 0;  
 for (int k = 0; k < matC.size; k++){  
 sum += matA.getValue(i,k) \* matB.getValue(j, k);  
 }  
 matC.setValue(sum, i, j);  
 }  
}

Toto je možné lehce optimalizovat pro cache tak, že se před for-cykly matice B transponuje a po výpočtu se transponuje zpátky.

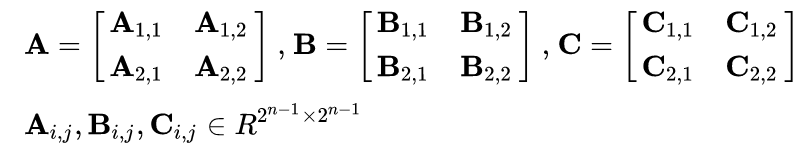
### Cache optimalizovaná implementace

U klasického násobení jsem také naimplementoval jeho cache optimalizovanou verzi využívající techniky ze Short Job 1. (Loop tiling, loop unrolling, etc.)

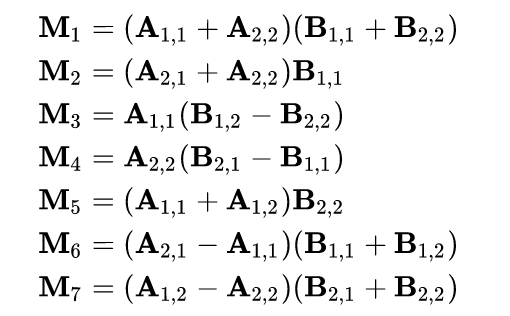
## Strassenův algoritmus

Efektivnější metoda výpočtu násobení matic je **Strassenův algoritmus**.

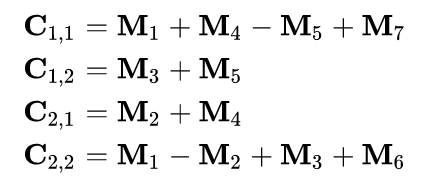
Matice **A, B** a **C** rozdělíme na bloky (podmatice) stejného řádu:



Definujeme-li nové matice:



můžeme bloky Ci,j vyjádřit následujícím způsobem:



Tímto způsobem stačilo pouze 7 maticových součinů velikosti 2n-1. Kdyby se postupovalo naivně bez matic **M**, a bloky **Ai,**j a **Bi,**j se mezi sebou vynásobily a sečetly do bloků **Ci,j** tak bychom potřebovali 8 maticových součinů velikosti 2n-1.

Tento postup se děje rekurzivně až do nějaké zvolené velikosti matice, kde už se provede standartní naivní násobení matic.

### Implementace

Pro implementaci jsem si nadefinoval následující strukturu, která reprezentuje matici.

typedef struct {  
 MAT\_DATA\_T \* \_\_restrict\_\_ mat; // 1D Buffer for matrix values  
 int size; // Size of current matrix  
 int row\_size; // Size of TOP row matrix, in case this matrix is a submatrix of other matrix  
 MAT\_DATA\_T getValue(int row, int col) const{ return mat[row\*row\_size+col]; }  
 void setValue(MAT\_DATA\_T value, int row, int col){ mat[row\*row\_size+col] = value; }  
} Mat\_t;

Pro tuto strukturu jsem si nadefinoval podpůrné funkce:

allocateMat(Mat\_t \* mat, int size, bool init = false) - Alokace

void freeMat(Mat\_t mat)- Dealokace

void printMat(Mat\_t mat)- Výpis matice na výstup

void randomizeMat(Mat\_t mat) – Randomizace matice

subMatOf(Mat\_t mat, Mat\_t \* sub11, Mat\_t \* sub12, Mat\_t \* sub21, Mat\_t \* sub22, int subSize)- Vytvoření podmatic řádu 2n-1

void compareMat(Mat\_t matA, Mat\_t matB) – Porovnání všech hodnot dvou matic

Strassenův algoritmus jsem implementoval rekurzivně. Do každého volání vstupuje matice A, B a C.

Nejprve se kontroluje podmínka limitu velikosti matice. Toto rozhoduje, zda se bude pokračovat podle Strassenova algoritmu, nebo je velikost matic natolik malá, že se vynásobí klasickým způsobem. Hodnota limitu bude popsána dále ve zprávě.

Pak se vytvoří podmatice **Ai,**j, **Bi,**j, a **Ci,**j pro výpočet.

Dále se alokuje 7 matic M, a 14 matic na mezivýsledky. Všechny jsou stejně velké jako vstupní matice.

Aby se zamezilo opakované alokaci/dealokaci matic M a matic pro mezivýsledky, vytvoří se před během Strassenova algoritmu velký buffer pro hodnoty všech možných matic M a mezivýsledky.

Tento buffer má velikost: **2\*21\*(velikost výstupní matice C / 2)^2** – V každém volání rekurze Strassena je potřeba 21 matic a každé volání má poloviční velikost těchto matic. Tímto způsobem velikost potřebného bufferu je O(2\*21\*(počet prvků matice C))

Následně se spočítají matice M a poté podmatice C.

Během měření výkonosti implementace jsem porovnával implementaci, která využívá klasické násobení v podmínce, a implementaci, která využívá cache optimalizovanou implementaci klasického násobení.

# Paralelní implementace

Paralelní implementaci Strassenova algoritmu jsem provedl dvěma způsoby. První je paralelizace samotných operací násobení, sčítání a odčítání matic, a druhý je paralelizace pomocí datových závislostí.

Zde jsou popsané **rozdíly** oproti sekvenční implementaci.

## Paralelizace operací

Všechny operace, které Strassenův algoritmus využívá (sčítání, odčítání, násobení), jsou implementovány pomocí nějaké formy for-cyklů. Na tyto for cykly jsem aplikoval pragmy z OpenMP. Konkrétně:

#pragma omp parallel for private(sum) schedule(auto) collapse(2)

Která provede collapse na horní dva for-cykly a for-paralelizuje výpočet.

U operace transpozice matice a cache optimalizované násobení jsem využil:

#pragma omp parallel for schedule(auto)

Na vrchní for-cyklus

Touto paralelizací algoritmus postupuje „sekvenčně“ a samotná paralelizace je pouze v samotných operacích. Tudíž není nutné řešit datové závislosti.

## Paralelizace pomocí datových závislostí

U tohoto způsobu jsem využil faktu, že uvnitř volání Strassenova algoritmu se vypočítávají matice M nezávisle na sobě. Tudíž je možné je počítat paralelně. To samé (s omezeními) platí pro finální výpočet podmatic matice C.

K tomuto jsem využil taskový paralelismus a funkci *depend* z OpenMP. Pouzil jsem tedy pragmy:

#pragma omp task depend(in:, out:)

Pro výpočty všech mezivýsledkových matic, matic M a podmatic C jsem nadefinoval závislosti vstupních matic a výstupních matic pro všechny výpočetní operace a nechal jsem OpenMP řídit dataflow paralelního výpočtu podle těchto závislostí.

U tohoto způsobu výpočtu jsem musel zanechat využívání velkého předem alokovaného bufferu pro mezivýsledkové matice a matice M, jelikož zde už se nedá rozumně vypočítat místo, kam v bufferu má každé vlákno ukládat výsledky.

Místo toho jsem zde alokoval/dealokoval tyto matice v každém volání Strassenova algoritmu.

Tato verze využívá v podmínce dolního limitu cache-optimalizované klasické sekvenční násobení.

# Vektorizace

Vektorizace byla využita. Zde jsem vypsal pouze výkonnostně významné vektorizace.

MatMul\_simple.h:56:27: optimized: loop vectorized using 32 byte vectors - nejvnitřnější for-cyklus v operaci sčítaní matic

MatMul\_simple.h:70:27: optimized: loop vectorized using 32 byte vectors - nejvnitřnější for-cyklus v operaci odčítaní matic

MatMul\_simple.h:39:31: optimized: loop vectorized using 32 byte vectors – nejvnitřnější for-cyklus v operaci násobení matic

MatMul\_cache.h:59:45: optimized: loop vectorized using 32 byte vectors - nejvnitřnější for-cyklus v cache optimalizované operaci násobení matic

## MatMul\_parallel.h:25:9: optimized: loop vectorized using 16 byte vectors – paralelní operace transpozice matice

MatMul\_parallel.h:44:31: optimized: loop vectorized using 32 byte vectors - nejvnitřnější for-cyklus v paralelní operaci násobení matic

MatMul\_parallel.h:229:49: optimized: loop vectorized using 32 byte vectors - nejvnitřnější for-cyklus v paralelní cache optimalizované operaci násobení matic

# Optimalizace

Zde je shrnutí optimalizací, které jsem využil pro implementaci sekvenčního i paralelního řešení:

* Velký buffer pro matice M a mezivýsledkové matice – během výpočtu se nemusí alokovat/dealokovat.
* V datové struktuře pro matici jsem využil pouze 1D pole pro prvky, a definoval vzorec podle kterého se v poli vyhledávají prvky matice podle řádku/sloupce – optimálnější využívání cache.
* Také jsem zde zvolil ukazatel na matici jako \_\_restrict\_\_ jelikož se v mé implementaci nevyskytuje anti-aliasing – lepší vektorizace
* Využití transpozice matice B při násobení dvou matic. – optimálnější využívání cache.
* Pro cache optimalizované násobení matic: Loop tiling, loop unrolling.
* Využití možnosti collapse(2) při paralelizaci for-cyklů.
* Výběr optimálního dolního limitu pro klasické násobení v Strassenovu algoritmu. (Optimální limit dosažen z měření)

Výsledný program jsem kompiloval s následujícím nastavením:

g++ -Wall -std=c++17 -O3 -fopenmp -ftree-vectorize -mavx -funroll-loops -ffast-math

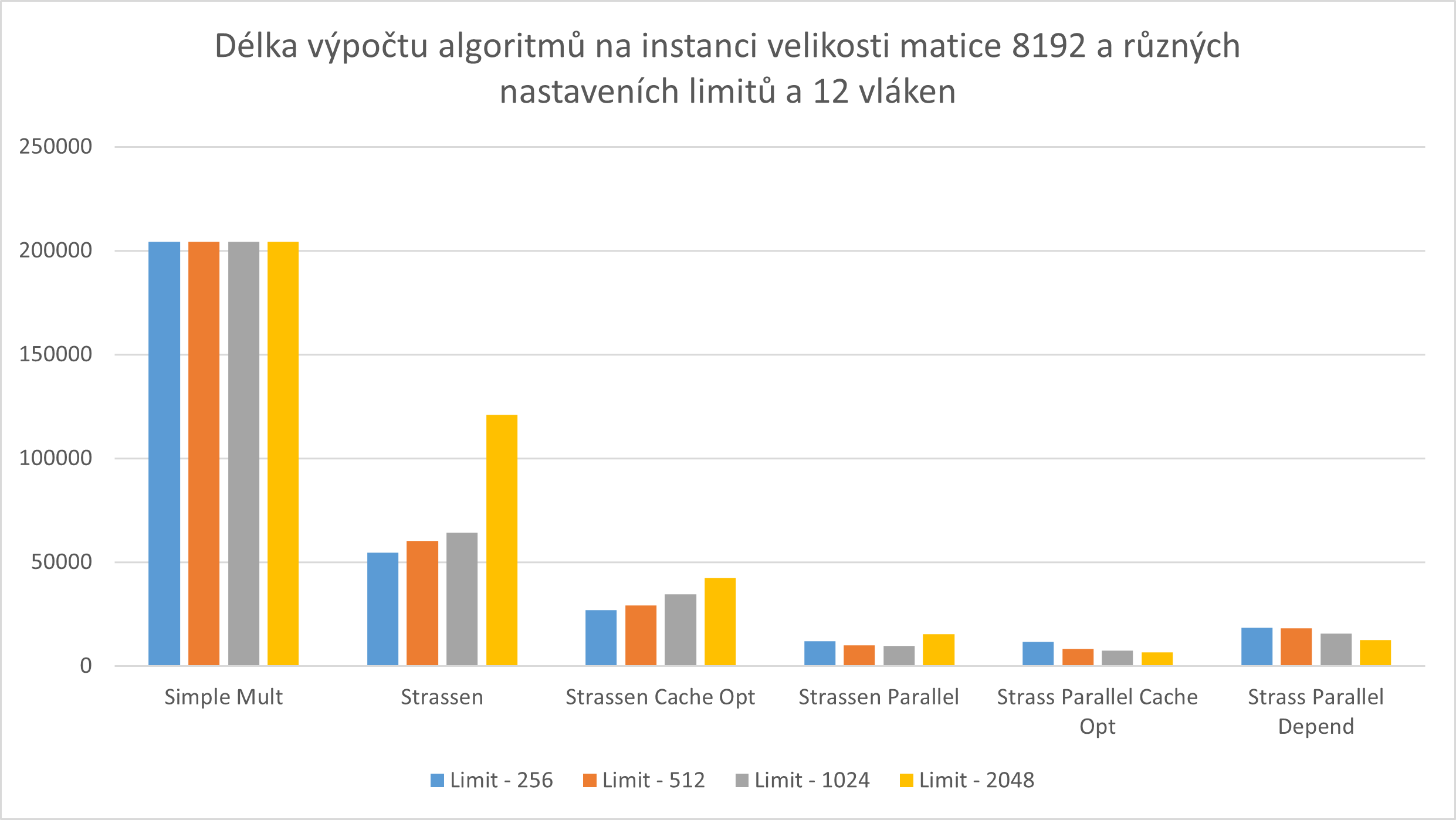
# Výsledky měření

Pro své měření jsem vybral instanci vstupní/výstupní matice řádu 8192. Měřené hodnoty dolního limitu pro Strassenův algoritmus jsem vybral 256, 512, 1024, 2048.

První jsem provedl měření délky výpočtu na 12 vláknech přes všechny hodnoty limitů:

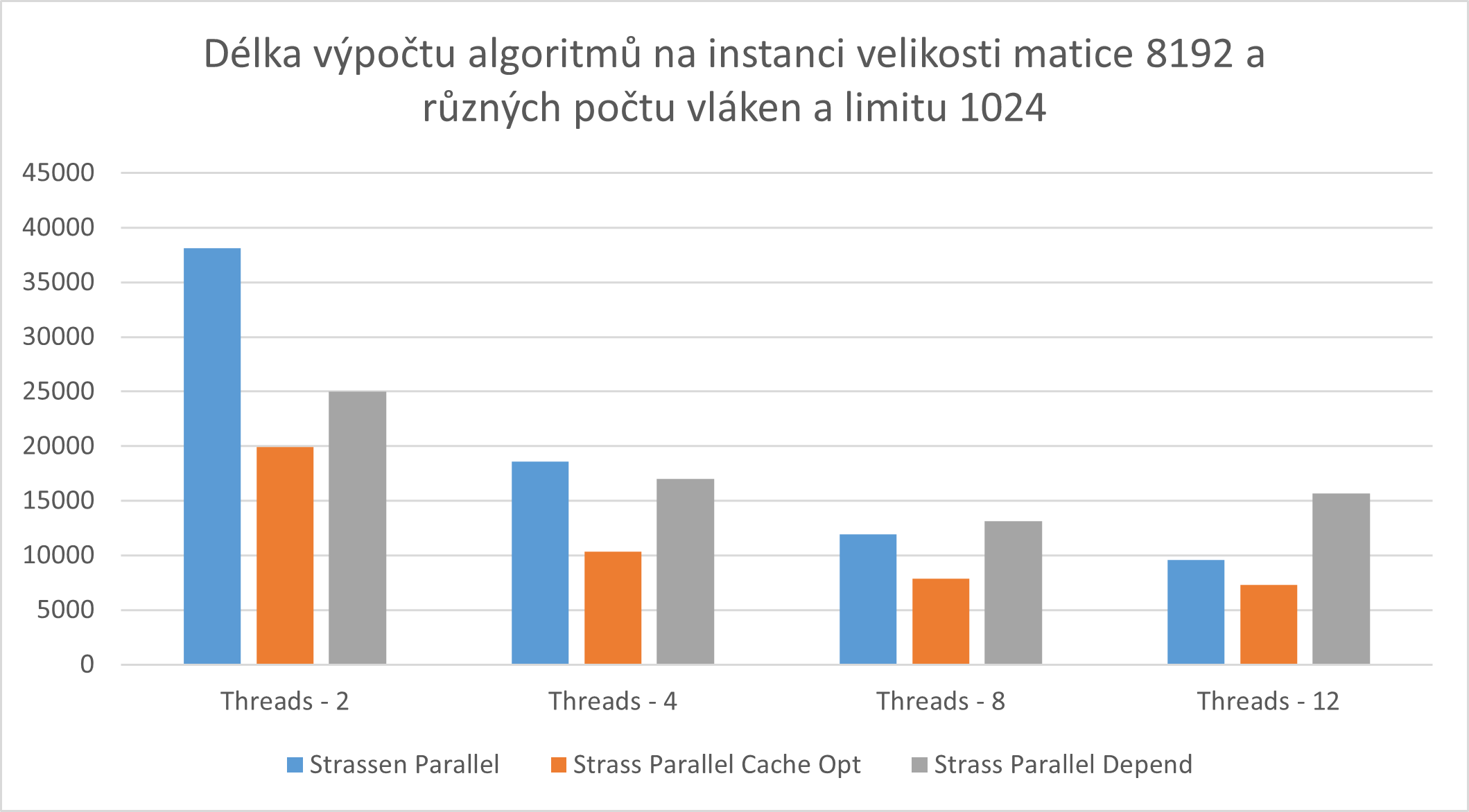
Délka výpočtu je v milisekundách.

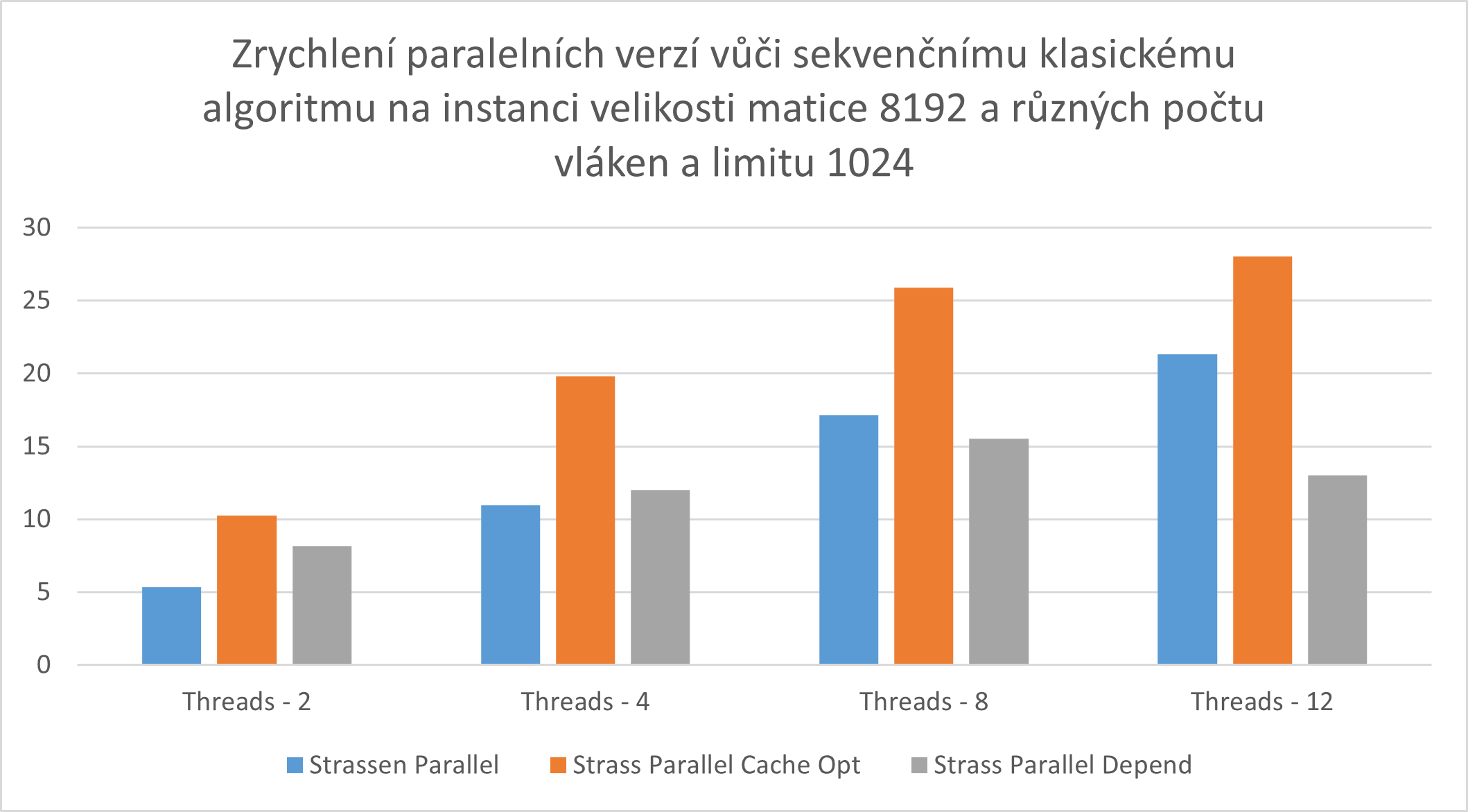
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Limit - 256** | **Limit - 512** | **Limit - 1024** | **Limit - 2048** |
| **Simple Mult** | **204128** | **204128** | **204128** | **204128** |
| **Strassen** | **54714** | **60219** | **64112** | **121078** |
| **Strassen Cache Opt** | **26888** | **29054** | **34538** | **42549** |
| **Strassen Parallel** | **11895** | **9958** | **9581** | **15404** |
| **Strass Parallel Cache Opt** | **11727** | **8177** | **7278** | **6706** |
| **Strass Parallel Depend** | **18339** | **18015** | **15696** | **12627** |



Z těchto měření jsem vybral „optimální“ limit 1024 a využil ho k měření napříč různých počtů využitých vláken. Počty vláken jsem vybral 2, 4, 8, 12.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Threads - 2** | **Threads - 4** | **Threads - 8** | **Threads - 12** |
| **Strassen Parallel** | **38119** | **18604** | **11923** | **9581** |
| **Strass Parallel Cache Opt** | **19925** | **10314** | **7882** | **7278** |
| **Strass Parallel Depend** | **25008** | **17009** | **13140** | **15696** |

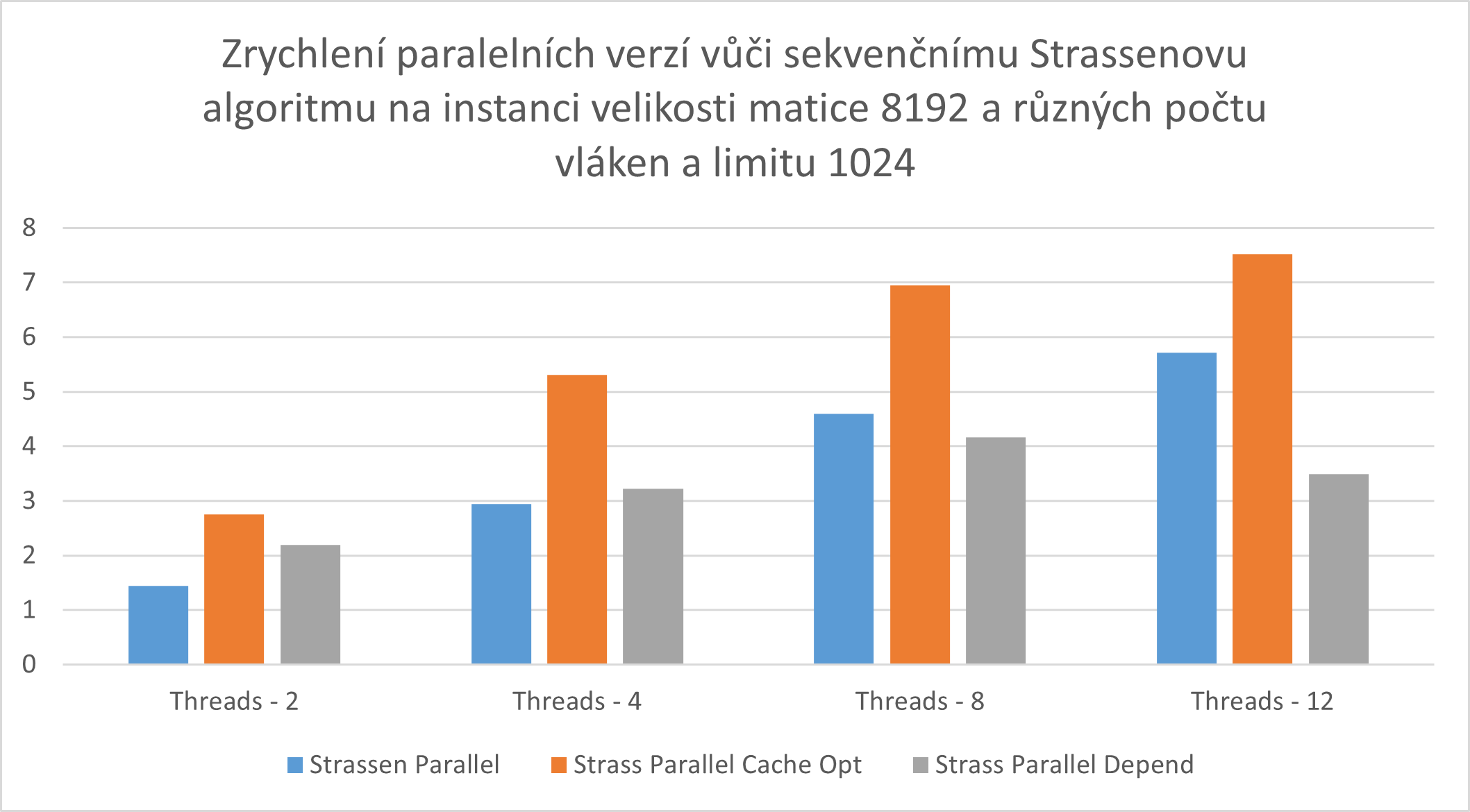


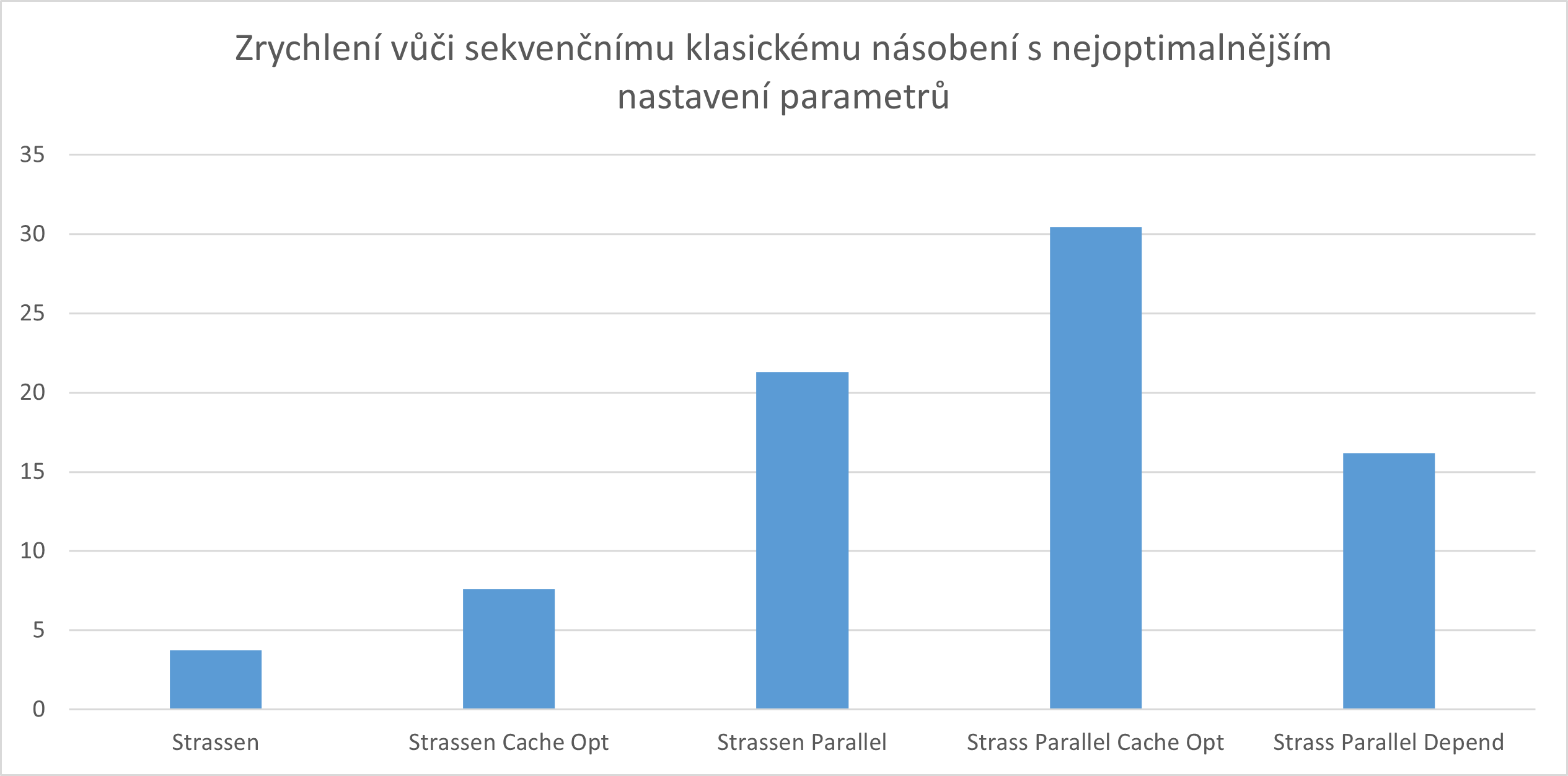
Z těchto výsledků jsem vypočítal zrychlení vůči klasickému sekvenčnímu násobení.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Threads - 2** | **Threads - 4** | **Threads - 8** | **Threads - 12** |
| **Strassen Parallel** | **5,355019806** | **10,97226403** | **17,12052336** | **21,30550047** |
| **Strass Parallel Cache Opt** | **10,24481807** | **19,79135156** | **25,89799543** | **28,04726573** |
| **Strass Parallel Depend** | **8,162507997** | **12,00117585** | **15,5348554** | **13,00509684** |

Také jsem porovnal zrychlení vůči sekvenčnímu Strassenovu algoritmu bez cache optimalizace.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Threads - 2** | **Threads - 4** | **Threads - 8** | **Threads - 12** |
| **Strassen Parallel** | **1,435347202** | **2,940980434** | **4,588945735** | **5,710677382** |
| **Strass Parallel Cache Opt** | **2,745997491** | **5,304828389** | **6,941639178** | **7,51772465** |
| **Strass Parallel Depend** | **2,187859885** | **3,216767594** | **4,163926941** | **3,485856269** |



  
Z těchto měření jsem vybral neoptimálnější nastavení parametrů (dolní limit) a porovnal jejich zrychlení vůči klasickému sekvenčnímu násobení

|  |  |
| --- | --- |
|  | **Speedup** |
| **Strassen** | **3,730818438** |
| **Strassen Cache Opt** | **7,591788158** |
| **Strassen Parallel** | **21,30550047** |
| **Strass Parallel Cache Opt** | **30,43960632** |
| **Strass Parallel Depend** | **16,16599351** |

# Analýza naměřených výsledků

Nejprve se zaměřím na porovnání sekvenčních verzí. Z výsledku je patrné, že samotný Strassenův algoritmus dává poměrně velké zrychlení i bez paralelizace vůči klasickému násobení. Zrychlení je zde 3,7x.

Zrychlení je ještě vetší s cache-optimalizovaným násobením. Tato optimalizuje zdvojnásobuje zrychlení Strassenova algoritmu vůči klasickému násobení. Zrychlení je až 7,5x

Z výsledků měření napříč limity je patrná vlastnost Strassenova algoritmu. A to, že v každé rekurzi se ušetří jedno maticové násobení. Tudíž čím vyšší dolní limit tím menší tento efekt je.

Toto však neplatí pro paralelní verze. U těch z výsledků vyplívá že vyšší limit Strassenova algoritmu je optimálnější. Předpokládám, že je to hlavně kvůli menšímu počtu synchronizace a vytváření/zanikání vláken.

Paralelní verze bez cache-optimalizací má vůči klasickému násobení zrychlení pro 12 vláken 21,3x. S grafu je patrné, že s vyšším počtem vláken, se zvyšuje zrychlení, každopádně od 8 vláken více další zrychlení není tak velké.

Při porovnání vůči sekvenčnímu Strassenovu algoritmu získáváme zrychlení na 12 vláknech 5,7x.

Tyto vlastnosti platí i pro cache-optimalizovanou paralelní verzi. Ta je ale obecně ještě rychlejší než paralelní verze bez optimalizací. Vůči klasickému násobení získáváme zrychlení 30x a vůči Strassenovu algoritmu 7,5x.

Paralelní verze, která využívá datové závislosti, byla obecně pomalejší než předchozí paralelní verze. Zde nejvyšší naměřené zrychlení bylo 16,1x. Toto není překvapivé, vzhledem k tomu, že v této paralelizaci silně záleží na tom, kolik task instancí je dostupných v daný moment pro výpočet. Je pravděpodobné, že během výpočtu nastávají situace, kde některá vlákna nic nepočítají, protože čekají.

U této verze se také projevilo, že více vláken než 8 už způsobuje zpomalení.

# Závěr

V této semestrální práci jsem implementoval paralelizované algoritmy pro výpočet násobení čtvercových matic. Pomocí optimalizací a paralelizace na 12 vláknech jsem dosáhl zrychlení až 30x vůči naivnímu klasickému sekvenčnímu násobení. Samotná paralelizace vůči Strassenovu algoritmu dosáhla 7,5x zrychlení na 12 vláknech.

Ze všech měřených verzí byla nejrychlejší verze, která paralelizuje samotné operace násobení, sčítání a odčítání matic a zároveň operace násobení je cache-optimalizovaná.

# Literatura

https://en.wikipedia.org/wiki/Strassen\_algorithm