Homework 6 - matrix_experiment

Jiří Klepl

1 Prolog

V následujících simulacích, čtyř simulovaných a jedné reálné, porovnáváme naivní implementaci (algoritmu "řádek po řádku") a cache-oblivious implementaci transponování čtvercové matice. Uvažovaná cache-oblivious implementace bude varianta algoritmu popisovaného na přednášce používající dvě subrutiny (T a TS). Předpokládáme, že procházení podmatic je děláno v clockwise či anti-clockwise pořadí (a nikoliv v zašroubovávacím).

Sledovaným parametrem je průměrný počet cache-miss na přístup a jeho závislost na velikosti oné matice. Obě implementace provádějí shodné změny na zadané matici, ale v jiném pořadí. Při simulacích je uvažována jednovrstvá cache s LRU strategií.

2 Simulované testy

Jak bylo výše zmíněno, předpokládáme jednovrstvou cache s LRU strategií. Její velikost bude vždy udána parametrem M, velikost paměťových bloků (cache-line) potom parametrem B. Jeden experiment sestává z měření počtu cache-miss nad maticemi s rozměry $N \times N$, kde $N = 2^{\frac{i}{4}}; i \in \{20, 21, \ldots 52\}$, vždy se stejnou cachí.

2.1 Vedlejší efekty na výsledky

Pro všechny simulace platí, že optimalizér kódu nehraje žádnou roli (nemusí totiž tomu tak být u reálných experimentů) a že testovaný kód dokonale popisuje uvažované výpočty.

Na druhou stranu, reálným vliv na výsledky má její příprava. Ta je prováděna metodou "řádek po řádku" a jejím efektem je uložení posledních několika bloků do simulované cache. U velkých matic (ve srovnání k velikosti cache) to nebude mít významný vliv, ale u malých matic to bude potenciálně znamenat snížený počet cache-miss a u analýzy výsledků by toto mělo být zohledněno. Zejména pokud se matice celé vejde do cache, nastanou falešně nulové výsledky.

Výše zmíněný efekt může být v reálných aplikacích využit pro optimalizace, ale stojí na domluvě mezi jednotlivými algoritmy (kde na dané matici začínají a končí) a tedy není předmětem naší analýzy.

Velmi podobný efekt vznikne, když je matice moc malá na to, aby strategie swappování měly dostatečný vliv na počet cache-miss, například, když se celá matice vejde do cache.

2.2 Očekávané výsledky

2.2.1 Odhad naivní implementace

Pro všechny simulace bude očekávaný počet cache-miss na přístup u naivní implementace pracující nad dostatečně velkými maticemi přibližně roven 0.5 (po krátkém vysvětlení uvedeme přesnější odhad). Toto pro nás bude sloužit jako horní limit, kterého lze jistě dosáhnout.

To plyne z toho, že při swapu je vždy jeden prvek pravděpodobně ve stejném bloku jako jeden z naposledy swappovaných prvků ($P_1 = \frac{1}{B}$), zatímco s ním swapovaný (s prohozenými souřadnicemi) je vždy v nenacachovaném bloku ($P_2 = 1$). Obojí dohromady má za výsledek průměrný počet cache-miss na přístup (bez ohledu na zarovnání):

$$P_{max} = \frac{B+1}{2B} = 0.5 + \frac{1}{2B} \approx 0.5$$

2.2.2 Dolní mez cache-miss

Dolní hranice průměrného počtu cache-miss je rovna $\frac{1}{B}$. To jde ukázat tím, že je nutno načíst $\frac{N^2}{B}$ bloků paměti za N^2 přístupů, tedy minimální průměrný počet cache-miss na přístup je $P_{min} = \frac{\frac{N^2}{B}}{N^2} = \frac{1}{B}$. Dosažení tohoto limitu je možné, pokud jsou algoritmem práce na paměťových blocích dokonale spárovány (například, konkrétně v použitém algoritmu, se v nějaké hloubce matice rozdělí na zrcadlové dvojice podmatic (samozřejmě podle diagonály), kdy tyto dvojice dokonale pokryjí cache; lze jednoduše zobecnit).

$$P_{min} = \frac{1}{B}$$

2.2.3 Horní odhad ideálního případu cache-oblivious implementace

Horní odhad průměrného počtu cache-miss v ideálním případě je rovna $\frac{1}{B}$, je-li $2B^2 \leq M$. Tedy, je-li cache po cache-line rozdělitelná na alespoň dva čtverce (tomu pak odpovídají zrcadlové dvojice podmatic, při počtu dělitelným čtyřmi virtuálně zdvojnásobíme cache-line), poté lze dosáhnout postupu popsaného výše. V opačném případě nalezneme mocninu dvou B' takovou, že $2BB' = M$, poté je horním odhadem ideálního případu $\frac{1}{B'} = \frac{2B}{M}$ a to odpovídá strategii, podobné minule diskutované, kdy však toto rozdělení tvoří pouze podmnožina cache o velikosti $2B'^2$, zbytek neužitečný, až na náhodné zlepšení.

$$P_{imax} = \begin{cases} \frac{1}{B}, & \text{if } 2B^2 \leq M \\ \frac{2B}{M}, & \text{otherwise} \end{cases}$$

2.2.4 Horní mez neideálního případu cache-oblivious implementace

Nejprve budeme uvažovat "polo-ideální" případ. U libovolné dvojice podmatic z ideálního případu, kdy matice měly rozměry shodné s paměťovými bloky, když myšleně posuneme bloky paměti přesně o polovinu a rozdělíme zmíněné podmatice, každou na čtyři další, každou teď prochází $\frac{2B'}{4} = \frac{B'}{2}$ unikátních paměťových bloků a danou dvojicí tedy vždy B', tedy stejně, jako tomu bylo u původních podmatic v ideálním případě, tedy na celou dvojici podmatic je potřeba $4B'$ načtení namísto $2B'$. A toto lze dobře rozdělit LRU strategií cache.

Toto lze zobecnit pro libovolné posunutí bloků, kdy sousední dvojicí nových podmatic prochází maximálně $\frac{3B'}{2}$ paměťových bloků, tedy na celou původní podmatici $3B'$ načtení a na dvojici tedy $6B'$ načtení, to by nám dalo teoreticky trojnásobek ideálního odhadu, pokud stále zafunguje LRU strategie. Odpovědí je samozřejmě, že LRU strategie zafunguje, což lze snadno ukázat výčtem možností.

Fyzicky je samozřejmě daná dvojice původních podmatic stále na $4B'$ paměťových blocích. Tedy, vejdou-li se do cache, je horní mez dvojnásobkem meze ideálního případu.

Je-li $2B'B = M$ a $B' < B$, pak bez újmy na obecnosti, za předpokladu, že všechny offsety řádků $\leq \frac{B'}{2}$, je každá druhá původní podmatice celá uvnitř B' paměťových bloků, tedy máme čtyři možnosti, podle toho, jaké z původních podmatic z dané dvojice se vejdou celé do B' bloků (lze si snadno ukázat, že na danou podmatici potřebujeme vždy B' aktivně v cache), které jsou stejně pravděpodobné (u dostatečně velkých matic) a tedy dají nám horní mez $\frac{6B'+2\cdot 2B'+2B'}{3\cdot 2B'} = 2$, dvojnásobek meze ideálního případu.

Je-li $2B^2 = M$, pak nelze bez znalosti konkrétního zarovnání a konfigurace cache vylepšit odhad trojnásobku a existují případy, kdy jej opravdu dosáhneme, není-li deterministicky dána priorita načtení matice (to ale v simulaci dáno je, tedy odhad bude vždy přesahovat).

Vše dohromady tedy:

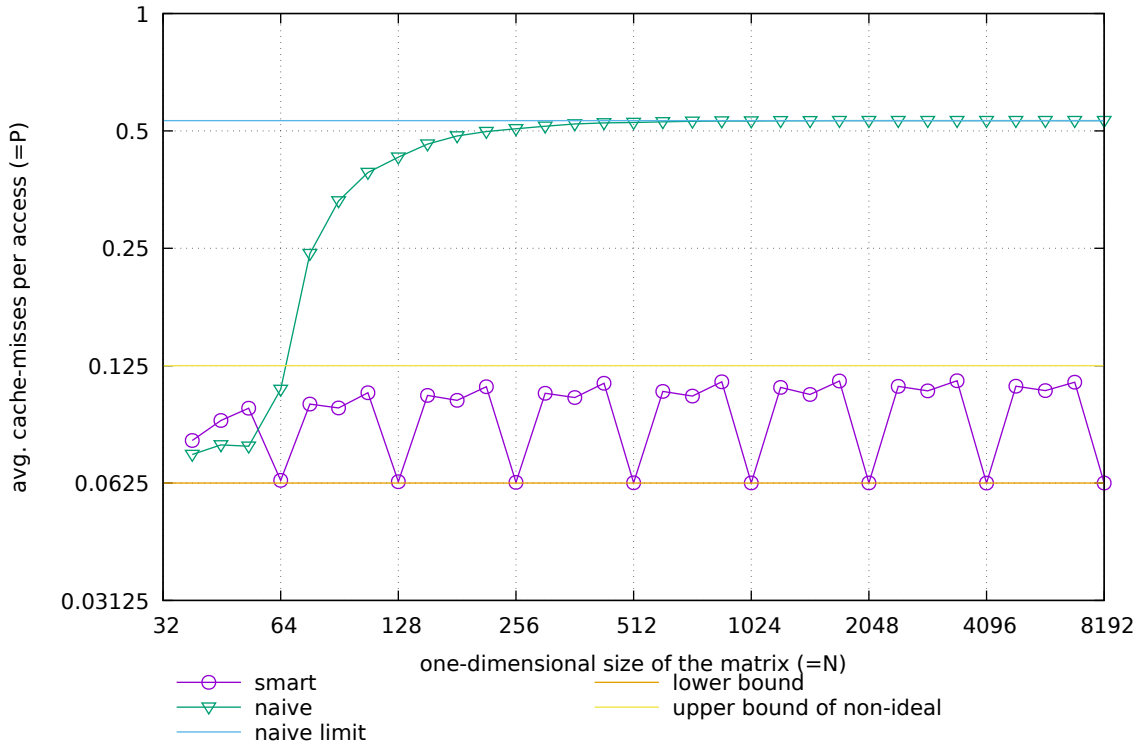
$$P_{nmax} = \begin{cases} 2P_{imax}, & \text{if } 4B^2 \leq M \vee \dfrac{M}{2B} < B \\ 3P_{imax}, & \text{otherwise} \end{cases}$$

2.3 Poznámka ke grafům

Všechny grafy používají logaritmickou škálu o základu 2 pro obě souřadnice. Souřadnice x je v logaritmické škále z důvodu zdůraznění periodicity vzhledem k logaritmu velikosti matice, souřadnice y z důvodu zvýraznění výsledků, neboť celá analýza hovoří o průměrném počtu cache-miss jako o násobcích určitých pravděpodobností a tedy je přirozenější používat škálu, kde násobky tvoří lineární zobrazení svých (taktéž logaritmicky zobrazených) složek.

2.4 Výsledky

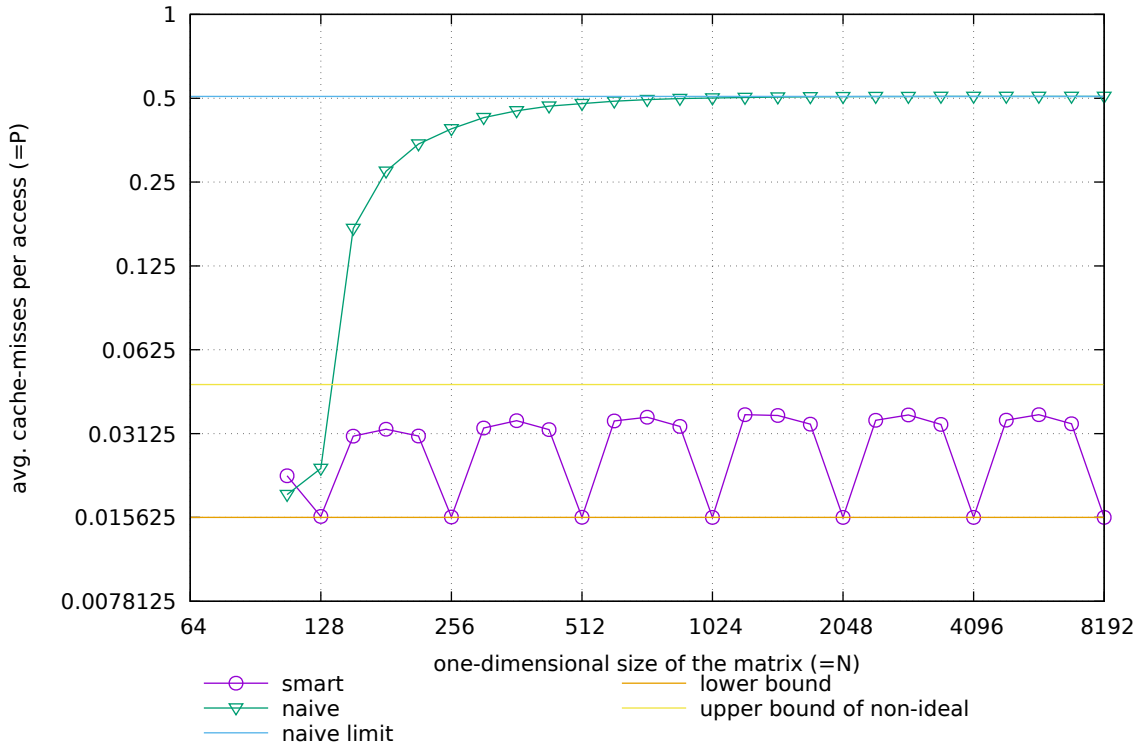
Figure 1: $M = 1024, B = 16$



Experiment zobrazen na figuře 1 dobře reprezentuje případ, kdy $4B^2 \leq M$, tedy z pohledu "chytré" implementace (výše zmiňovaná cache-oblivious) se jedná o nejideálnější konfiguraci cache, kdy v zarovnaných (ideálních) případech dosahuje optimálního počtu cache-miss a v nezarovnaných je nejhůře dvakrát horší. Obě implementace se chovají podle očekávání.

Pro všechny limity a meze vizte část 2.2.

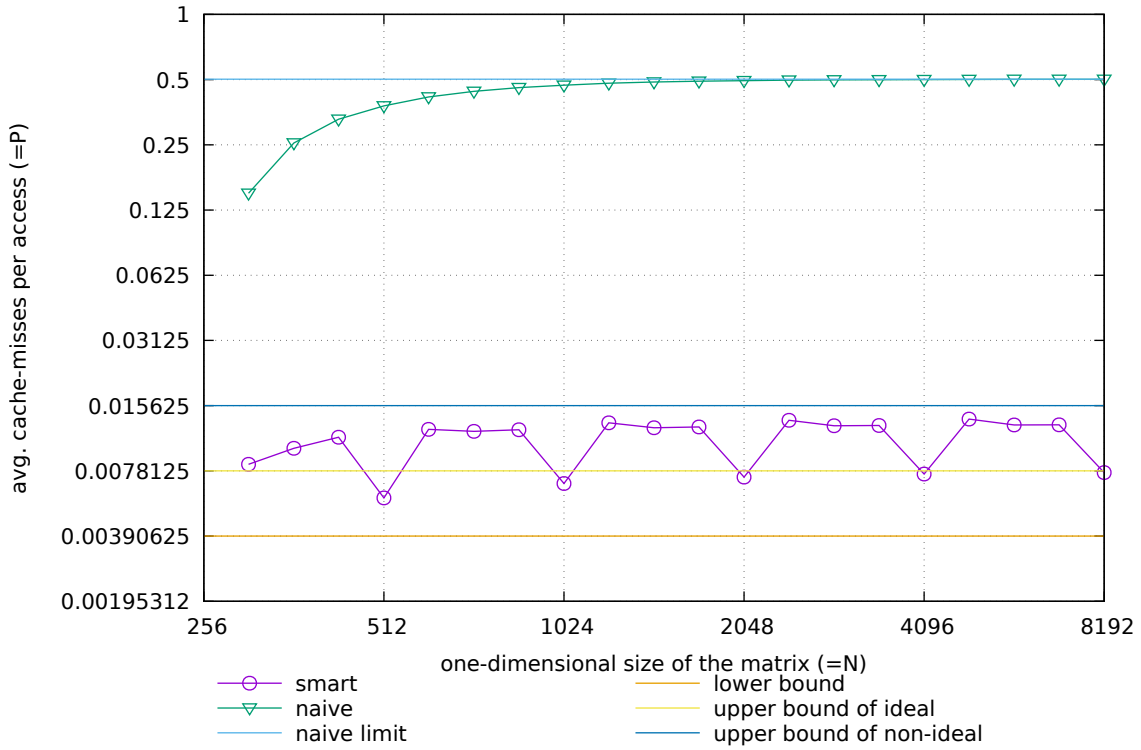
Figure 2: $M = 8192, B = 64$



Experiment zobrazen na figuře 2 dobře zobrazuje případ, kdy je pro chytrou implementaci konfigurace cache ještě stále velice dobrá v ideálních případech a ve srovnání s nimi špatná v nezarovaných, neboť $2B^2 = M$ (vizte část 2.2). Práce na nezarovnané matici může být doprovázen až trojnásobným počtem cache-miss.

Pro všechny limity a meze vizte část 2.2.

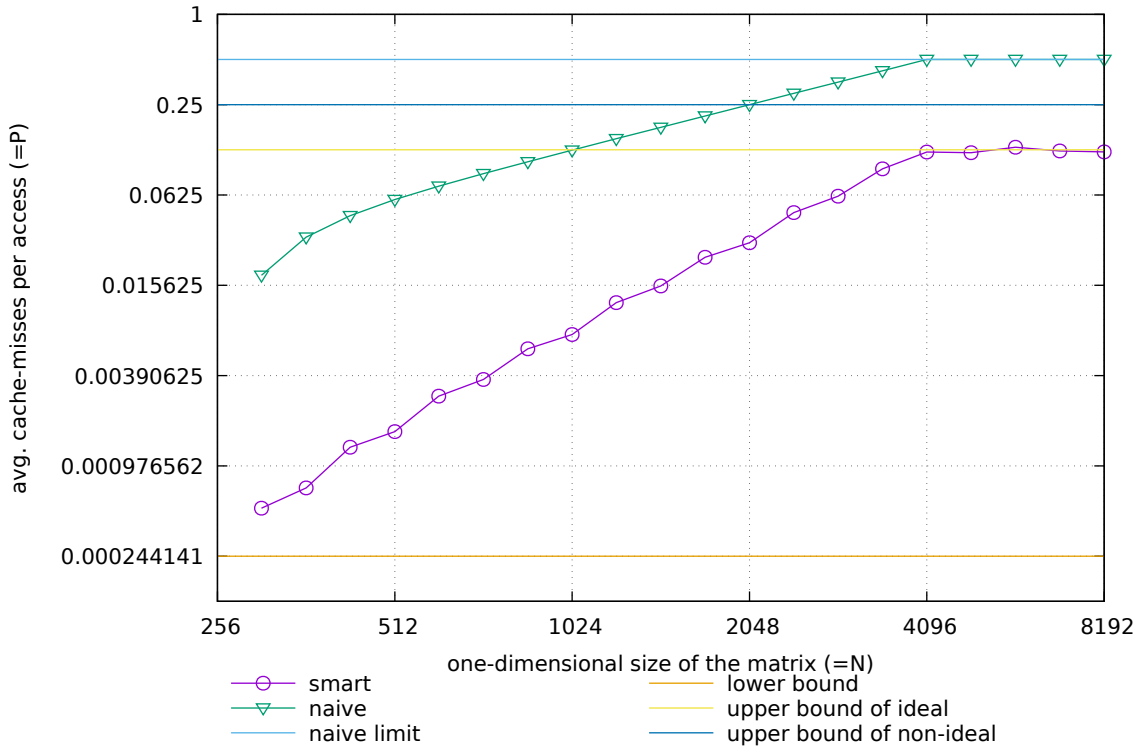
Figure 3: $M = 65536, B = 256$



Experiment zobrazen na figuře 3 dobře zobrazuje případ, kdy je cache vzhledem k velikosti cache line moc malá ($M < 2B^2$), aby cache-oblivious algoritmus dosáhl optimálního počtu cache-miss, ale zato je nejhorší případ jen dvakrát horší, než je dvojnásobek nejlepšího.

Pro všechny limity a meze vizte část 2.2.

Figure 4: $M = 65536, B = 4096$



Experiment zobrazen na figuře 4 zobrazuje podobný případ, tomu minulému, ale je zde dobře pozorovatelný rozdíl mezi prací na malých a velkých maticích. Tento experiment uvažuje stejně velkou cache jako předešlý experiment zobrazený na figuře 3, ale zde algoritmus u dostatečně velkých matic dosahuje mnohem horších výsledků, neboť konfigurace zcela porušuje předpoklad návrhu algoritmu, že $M \geq 2B^2$. U malých matic samozřejmě dosahuje velice dobrých výsledků (opět ve srovnání s předešlým experimentem), neboť jediným cache-miss je načtena významná část matice do cache.

Pro všechny limity a meze vizte část 2.2.

2.5 Zhodnocení

V částech 2.2 a 2.4 lze dobře vidět vliv zarovnanosti dat, velikosti paměťových bloků (cache-line) a konfigurace cache (pro tento algoritmus nejlépe $M \geq 2B^2$).

Nejlepších výsledků bude dosaženo jedině na zarovnaných datech (či na extrémě malých maticích, kde strategie transponování není rozhodujícím faktorem) a výsledky jsou zlepšovány se zvětšováním velikosti cache-line (díky menšímu potřebnému počtu načtení) za udržení výše zmíněného předpokladu na konfiguraci cache.

3 Reálné testy

Stroj, na kterém měření bylo provedeno má konfiguraci $L1 = 32KB, L2 = 256KB, L3 = 6MB, B = 64B$, $L1$ a $L2$ jsou osmi-cestné a specifické pro dané fyzické jádro, $L3$ poté dvanácticestná a sdílená, vždy set-asociativní. Clock-rate procesoru je v rozmezí $3.2GHz$ až $3.4GHz$) (Haswell i5-4460). Ze simulací bychom tedy očekávali průměrný počet cache-miss $P = \frac{1}{16}$. Použitý compiler byl gcc (verze pro Debian, 10.2.0-16).

3.1 Vedlejší faktory

V těchto testech stojí za zmínku, že neměříme přímo počty cache-miss, ale průměrný čas trávený prací nad jedním z N^2 prvků, tedy mají zde vliv, mimo již zmíněné cache-miss i volání pracujících subrutin (T a TS) a jejich vnitřní logika, zde potom má vliv i predikce skoků (což je tématika, která má u moderních počítačů podobný vliv, jako analýza cache). Dalším vlivem může být funkce optimalizéru, jak již bylo předem řečeno.

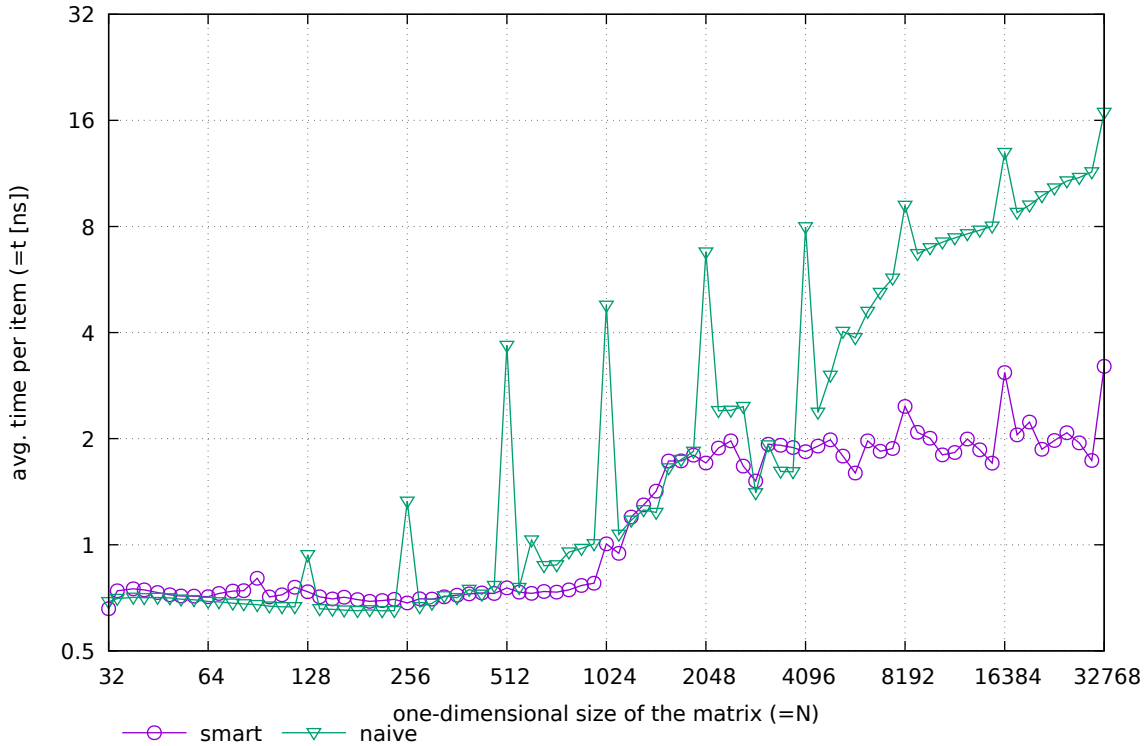
Proto je zde používán upravený algoritmus s téměř inlinovaným kódem, pokud dokáže podmatice pohodlně zarovnat na mocniny dvou (jejich abstrakce, ne data), v takových případech pak téměř úplně vymizí všechny vlivy ze strany logiky a predikce skoků, neboť takto upravený kód je z pohledu procesoru mnohem jednodušší.

Dále je použita úprava algoritmu, že na dostatečně malé matici (s jedním z rozměrů maximálně 4) algoritmus přejde na "naivní" zpracování dané podmatice - to pomáhá zejména na malých rozměrech.

Testy byly upraveny, aby používaly přesnější měření času.

3.2 Výsledky

Figure 5: Real experiment

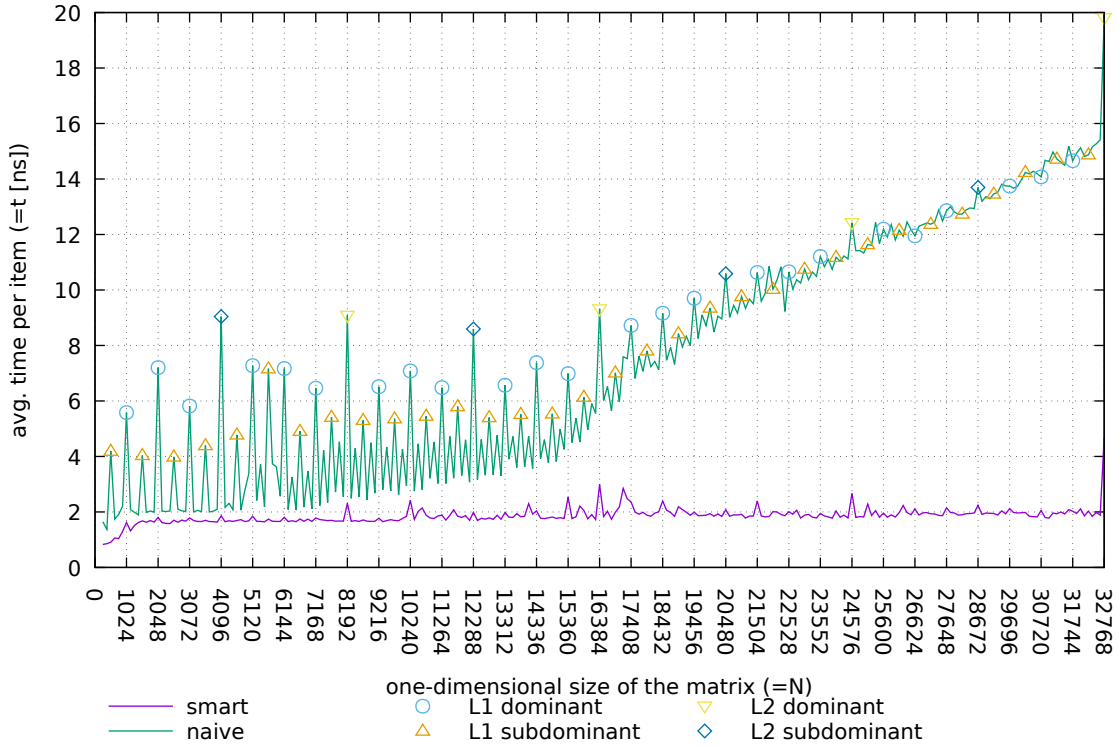


Na figuře 5 je zobrazen experiment s časy transponování matic od $N = 32$ do $N = 32768$, kde $N = 2^{\frac{k}{8}}; k = 40..120$. Jednotlivé hodnoty rozměrů matic sledují exponenciální řadu, tak je přirozené graf znázorňovat v logaritmické škále (u osy y taktéž pro zvýraznění násobností zrychlení a zpomalení).

Tvar grafu chytré implementace od $N = 32$ do $N = 512$ napovídá, že se matici daří přednačíst do $L2$ a pracuje pak pohodlně s cachí $L1$, někdy od $N = 1024$ se nejspíš přestává celá vejít do $L3$ a ustálí se práce implementace až na $N = 2048$ a dál. Graf nijak nenaznačuje jiné významné zdroje šumu.

Tvar naivní implementace, kde jsou velké výkyvy vysvětlíme za pomoci nového experimentu, který vyšetřuje vliv n-cestnosti cache.

Figure 6: Cache experiment



Grafy experimentu zobrazeny na figuře 6 ukazují zaznamenané hodnoty stejné, jako v minulém experimentu, ale od $N = 256$ do $N = 32768$, kde $N = 128k; k = 2..256$, tento graf není v logaritmické škále pro zvýraznění absolutních rozdílů.

Na naivní implementaci lze snadno pozorovat dobře předvídatelné různě velké výkyvy, předvídatelnost postupně klesá.

Tyto výkyvy jsou pravděpodobně způsobeny n-cestnostmi $L1$, $L2$ a $L3$ cache. Ty totiž způsobují, že v periodě $p = \frac{M}{n \cdot sizeof(unsigned)}$ prvky matice sdílejí stejnou množinu (kde M je velikost dané cache a n (n-cestnost) je počet cache-line velikosti B ve množině, velikost množiny je vždy $n \cdot B$, počet množin tedy $\frac{M}{n \cdot B}$ a prvků v cache-line je $\frac{B}{sizeof(unsigned)}$; $sizeof(unsigned) = 4, B = 2^6$ pro procesor používaný v experimentu).

Dané periody jsou $p_{L1} = \frac{32KB}{8 \cdot 4B} = 1024$, $p_{L_2} = \frac{256KB}{8 \cdot 4B} = 8192$ a $p_{L3} = \frac{6MB}{12 \cdot 4B} = 2^{17}$.

Nyní se konečně dostáváme k samotným výkyvům. Ty největší nastávají právě, je-li N dělitelné nějakou takovou periodou, v grafu zobrazeno pro $L1$ a $L2$ jako "dominantní". To vzniká, jak již bylo naznačeno, tím, že prvky matice v dané cache po sloupcích vždy sdílejí stejnou množinu v cache. Je zde pak další důležitý efekt působícím u moderních

procesorů, tedy s pipelining, kdy procesor nafetchuje data, než je vyhodnotí. Na těchto sloupcích tedy lze nafetchovat jen n prvků (n-cestnost), a tedy se procesor musí více zpomalit.

Na násobcích tento efekt funguje stejně a na $N = \frac{p}{k}$ pak přibližně v síle k^{-1}, lze nafetchovat kn prvků.

S postupně rostoucím N přestává být efekt p_{L1} dostatečně výrazný a poté i p_{L2}, to je nejspíše způsobeno tím, že hlavní vliv na tvar grafu přejímá $L3$ a latence v RAM, obojí sdílené a vypadá velice náchylné na šum.