Optimalizace a hodnocení efektivity lineárních kódů

Ivan Šimeček, Pavel Tvrdík

Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University
Karlovo nám. 13
121 35 Prague 2
Czech Republic

This research has been supported by grant IBS 3086102.

Obsah prezentace

(1) Proč optimalizovat?

(2) Architektura procesorů Intel (AMD) a některé nové rysy

(3) SW techniky pro optimalizaci

(4) Pravděpodobnostní model chování skryté paměti

(5) Knihovny pro LA, výhody a nevýhody

(6) Příklady použití jednotlivých optimalizačních postupů

Motivační příklad

Násobení matic (MMM), tzn. $A(n,n) \times B(n,n) = C(n,n)$

- Klasický algoritmus pro násobení (podle definice)
- Pro uložení čísel a výpočet použita dvojitá přesnost (typ *double*).

Pro všechna měření (kromě vyjímky na slajdu 15) byla použita následující konfigurace:

- Intel Pentium 4
  - 2.4GHz, výrobcem udávaná špičková výkonnost procesoru 2.4 Gflops.
  - 512MB hlavní paměť na 400MHz, 8KB skrytá paměť 1. úrovně (L1) a 128KB skrytá paměti 2. úrovně (L2).
- Intel kompilátor (`ICC`) s přepínači pro maximální výkon.

Standardní implementace MMM

```
Algorithm SMMM(in double A, B[1,...,n][1,...,n]; out C[1,...,n][1,...,n])
(* Standardní implementace MMM *)

for i = 1 to n do
    for j = 1 to n do
    begin
        s = 0;
        for k = 1 to n do
            s+ = A[i][k] * B[k][j];
        C[i][j] = s;
    end
```

Požadavky na paměťovou sběrnici

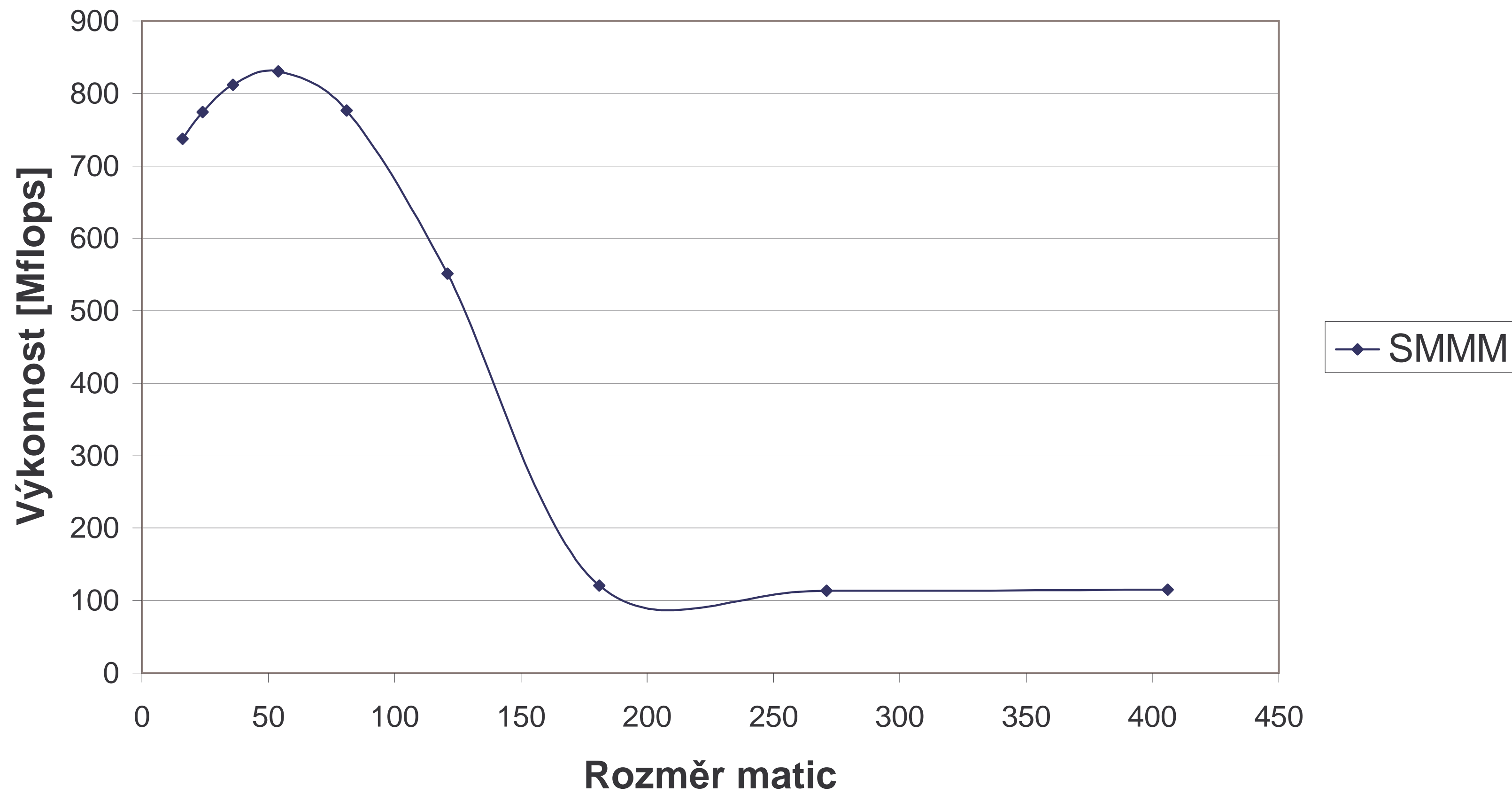
Na 1 prvek výsledné matice je třeba z paměti načíst $2n$ čísel.

Výkonnost

Počet aritmetických FPU operací za jednotkový čas:

$$V = \frac{2 \cdot n^3}{T_{\mathrm{SMMM}}}.$$

Proč optimalizovat ?



Vektorizace výpočtu

Nové procesory obsahují podporu pro vektorové výpočty.

- Intel:
  - od Pentia MMX pro typ *integer*,
  - od Pentia III pro typ *float* (technologie SSE),
  - od Pentia IV pro typ *double* (technologie SSE2).
- AMD:
  - od AMD K6 pro typ *integer*,
  - od AMD K6-2 pro typ *float* (technologie 3D now!),
  - od Athlon pro typ *double*.

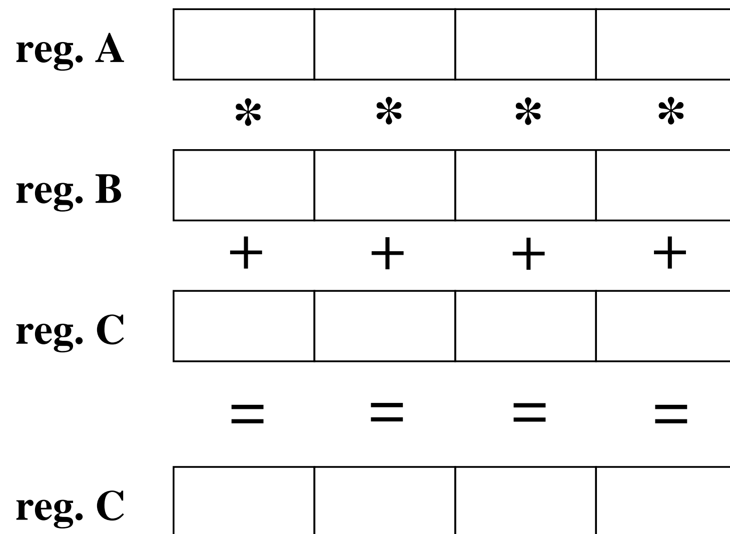
Architektura vektorové jednotky

- miniaturní architektura **SIMD** (Simple Instruction Multiple Data):
  - Procesor obsahuje **vektorové** registry pro krátká pole dat.
  - Jedna operace nad vektorovým registrem =
    paralelní operace nad jednotlivými položkami pole.

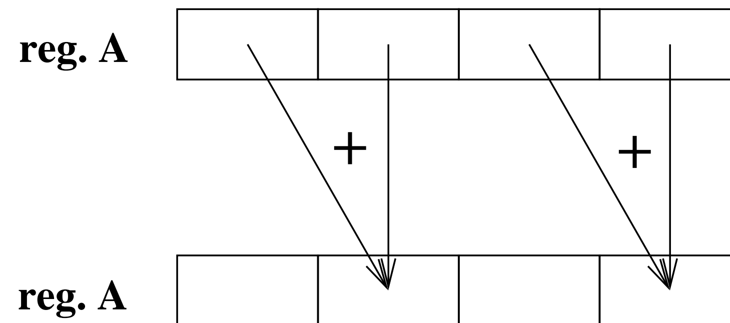
Vektorizace výpočtu (pokr.)

Typy vektorových instrukcí:

- binární aritmetické operace (sčítání, odčítání, násobení) (viz Obr. 1(a)),
- redukce pomocí takových op. (viz Obr. 1(b)),
- binární logické operace (AND, NAND, OR, XOR),
- porovnávací operace, maximum, minimum,
- konverze mezi jednotlivými formáty (*byte*, *word*, *doubleword*, *quadword*),
- posuvné operace.



(a)



(b)

Obr. 1

Vektorizace výpočtu (pokr.)

Požadavky:

(1) Položky musí v paměti sousedit.

(2) Klasický **for** cyklus s jedním výstupním bodem.

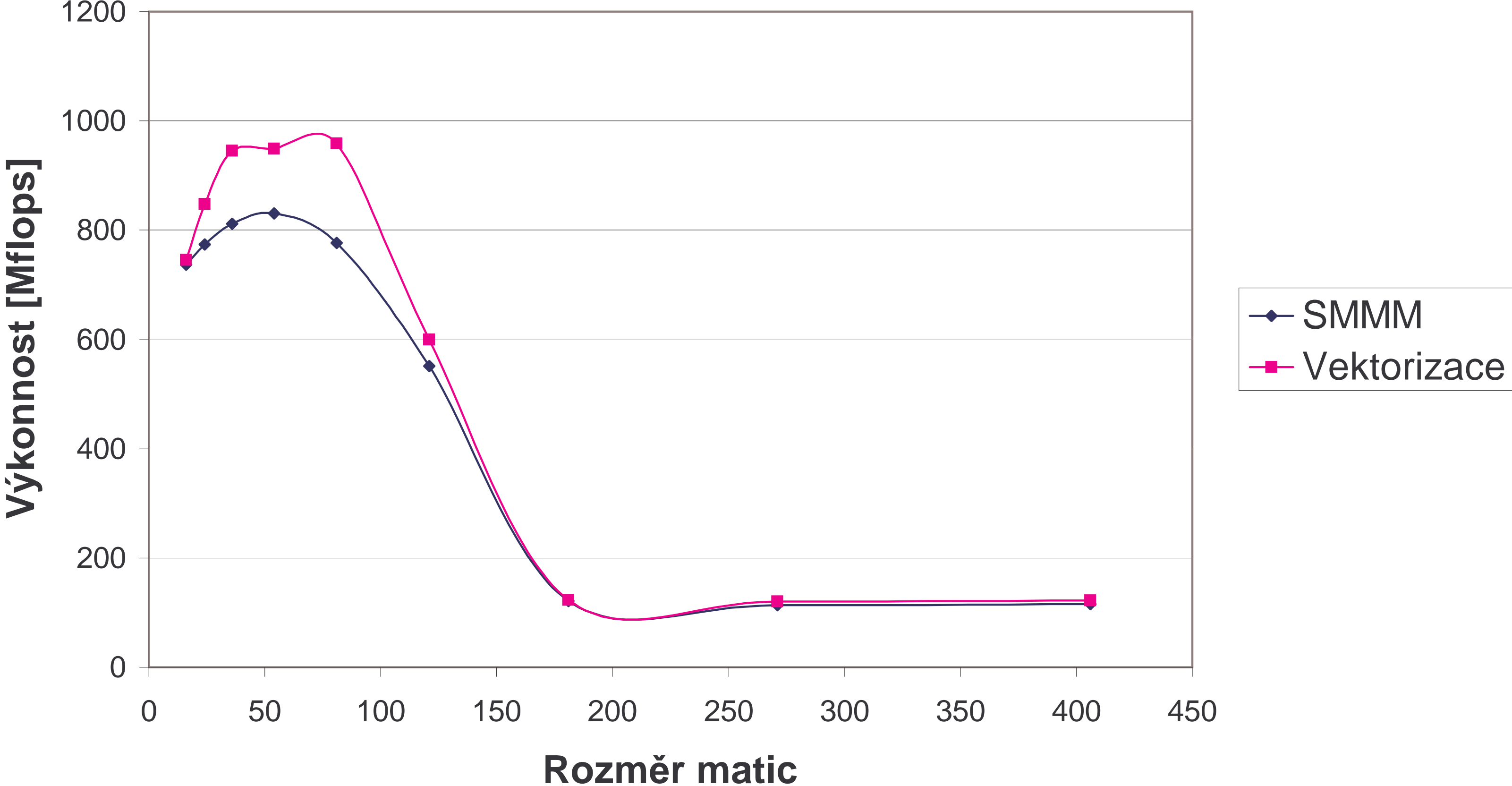
(3) Počet iterací znám před zahájením provádění cyklu.

(4) Položky z aktuální iterace nesmějí být závislé na položkách z předchozích iterací.

Podpora v kompilátorech:

- podpora nových datových typů (`GCC+ICC`),
- automatická vektorizace (`ICC`),
- automatická detekce vhodného procesoru (`ICC`).

Vektorizované MMM



Skrytá paměť (*cache*)

Malá a rychlá paměť pro uchovávání nejčastějších dat, úspěšnost jejíhož použití závisí na naplnění hypotézy výpočetní lokality:

(1) **časová** (*temporal*) = právě používaná data budou pravděpodobně brzy znovu použita;

(2) **prostorová** (*spatial*) = data sousední s právě používanými daty budou pravděpodobně brzy použita.

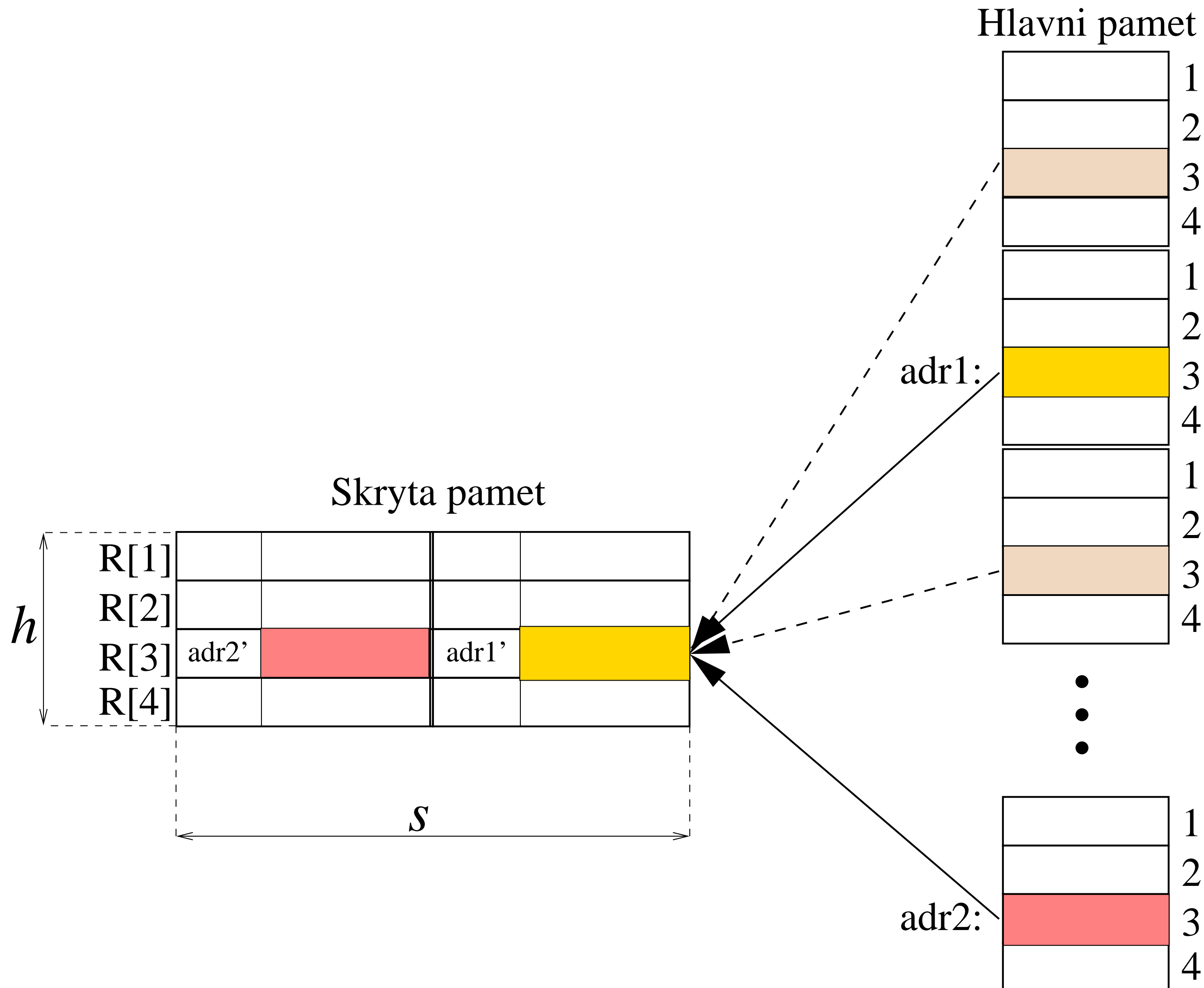
Def.: Skrytá paměť se stupněm asociativity s (s-*way set-associative cache*) = omezená **rozptylovací** (*hash*) tabulka s jednoznačnou **mapovací** fcí f: modulo 2^k.

- Celý adresní prostor se dělí do h disjunktních tříd.
- 1 třída se mapuje do 1 **množiny** (*set*) **bloků** (*lines*), značených $R[0, \dots, h-1]$.
- 1 množina $R[i] = s$ bloků, s = **stupeň asociativity**.

Velikost a struktura skryté paměti

Velikost **datové části** skryté paměti se stupněm asociativity s a h množinami bloků o velikosti B_S je

$$C_\mathrm{S} = s \cdot B_\mathrm{S} \cdot h.$$



Popis chování skryté paměti

- Nechť $\mathcal{A}$ je adresní prostor. Pak mapovací fce je zobrazení $f : \mathcal{A} \rightarrow \{0, \ldots, h-1\}$.
- Uvažujme např. čtení z adresy $adr \in \mathcal{A}$.
  - Pokud $adr \in R[f(adr)]$, pak **nalezení** platného bloku ve skryté paměti (*cache hit*) .
  - Pokud $adr \notin R[f(adr)]$, pak **výpadek** bloku ve skryté paměti (*cache miss*) a příslušný blok musí být načten z hlavní paměti.

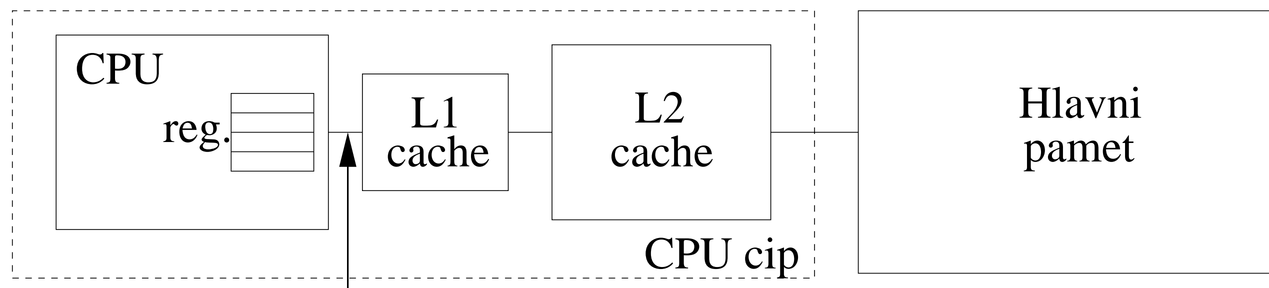
Rozlišujeme 2 typy výpadků ve skryté paměti:

(1) **povinné** (*compulsory*) $=$ první natažení dat z hlavní paměti do prázdných bloků skryté paměti ($|R[f(adr)]| < s$).

(2) **konfliktní** (*thrashing*) $=$ opakované natažení bloku, které byl před tím nahrán do skryté paměti, ale pak "předčasně" z kapacitních důvodů zrušen ($|R[f(adr)]| = s$).

Paměťová hierarchie

- Skrytá paměť 1. úrovně: doba odezvy kolem 1-2 taktů.
- Skrytá paměť 2. úrovně: doba odezvy kolem 2-5 taktů.
- Hlavní paměť: doba odezvy řádově 20-200 taktů $\Longrightarrow$ nejslabší článek řetězu: podstatně nižší průtok = objem načtených dat za jednotku času.

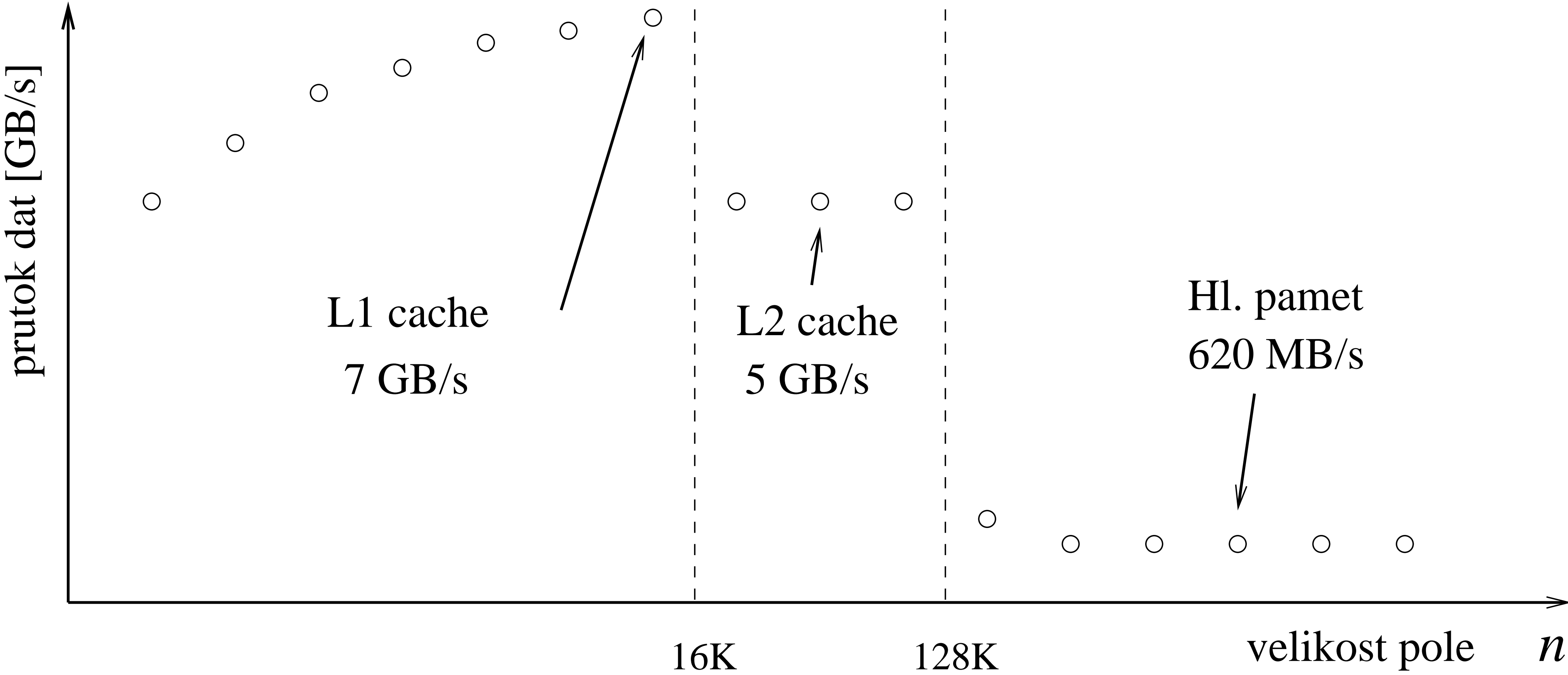


Test průtoku paměťové sběrnice

for $i = 1$ **to** n **do**
    $s = a[i];$
for $i = 1$ **to** n **do**
    $s = a[i];$

Průtok v rámci paměťové hierarchie

Velikost průtoku v závislosti na velikosti pole: na IBM PC Celeron 1GHz, hlavní paměť 256MB @ 100MHz, L2 = 128KB, L1 = 16KB.



Odhad časové složitosti lineárních kódů

- Uvažujme opět IBM PC Celeron 1GHz, hlavní paměť 256MB @ 100MHz, L2 = 128KB, L1 = 16KB.

- Uvažujme výpočet **skalárního** součinu 2 vektorů typu *double* o délce $n = 10^6$.

- Jaká bude doba trvání tohoto výpočtu?

(1) Teoretická špička (maximální výkonnost FPU):
1 takt = 1 flop $\implies$ $T_1 \geq$ 2Mflop / 1Gflops = 2 ms.

(2) Očekávaná výkonnost (70% maximální výkonnosti FPU):
$T_2 \geq$ 2Mflop / (1Gflops * 0,7)= 2,9 ms.

(3) Reálná výkonnost:

- Z hlavní paměti je třeba načíst 16MB dat, všechny mimo skrytou paměť.
- Předpokládáme průtočnost hlavní sběrnice 620MBs^{-1} (viz předchozí test).

Tudíž $T_3 \geq$ 16MB / 620MBs^{-1} = 25,8 ms.

Registry pro sledování četností událostí (*performance counters*) (1)

(1) Implementovány od procesorů:

- Intel: Pentium Pro,
- AMD: Duron.

(2) Více než 70 možných měřitelných událostí,
počet současně měřených událostí je omezen:

- Intel: max. 2.
- AMD: max. 4.

(3) Jsou to 64bitové čítače, jejichž čtení je privilegovaná operace.
Problém u OS Linux:

- patch do jádra,
- speciální C knihovna.

SW techniky a jejich vliv na počet výpadků ve skryté paměti

(1) **Rozvinutí cyklu a sloučení** (*loop unrolling-and-jam*).

(2) **Rozdělení cyklů do bloků** (*loop blocking*).

(3) (**Dynamické**) **obracení směru (smyslu) cyklu** (*(dynamic) loop reversal*).

Rozvinutí cyklu a sloučení (RC&S)

- Cyklus se rozvine do několika nezávislých vláken.
- Po dokončení cyklu se výsledky jednotlivých vláken sloučí.
- Počet vláken se označuje *rank*.

RC&S pro MMM (rank=2)

for $i = 1$ **to** n **do**
  for $j = 1$ **to** n **step** 2 **do**
  begin
    $s1 = 0;$ $s2 = 0;$
    for $k = 1$ **to** n **do**
      $s1 += A[i][k] * B[k][j];$ $s2 += A[i][k] * B[k][j+1];$
    $C[i][j] = s1;$ $C[i][j+1] = s2;$
  end

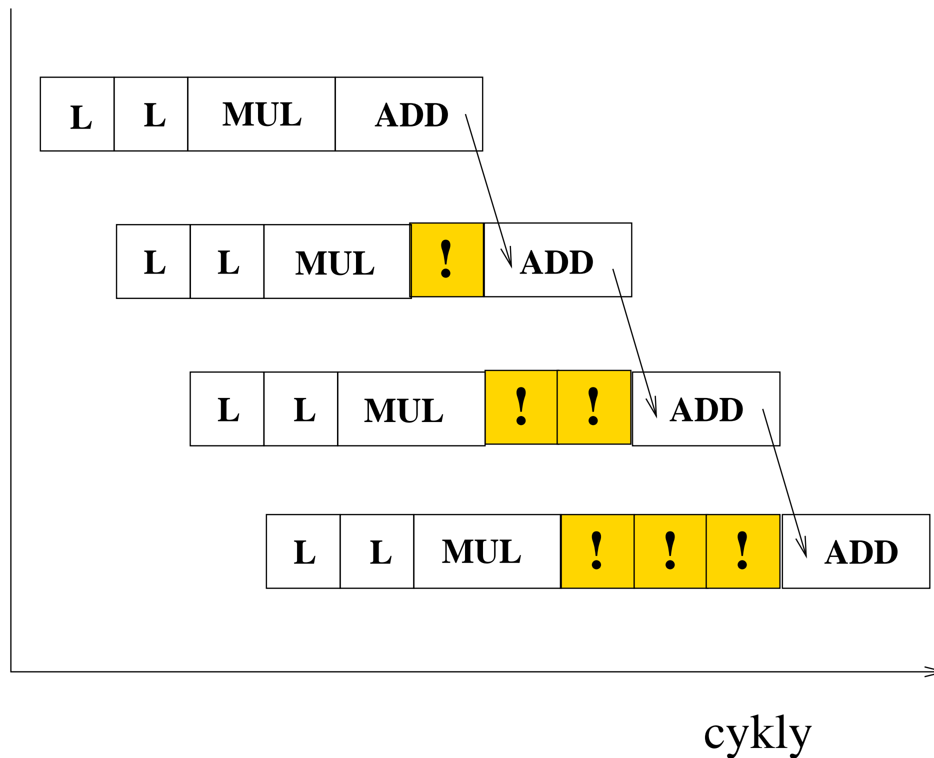
Požadavky na paměťovou sběrnici

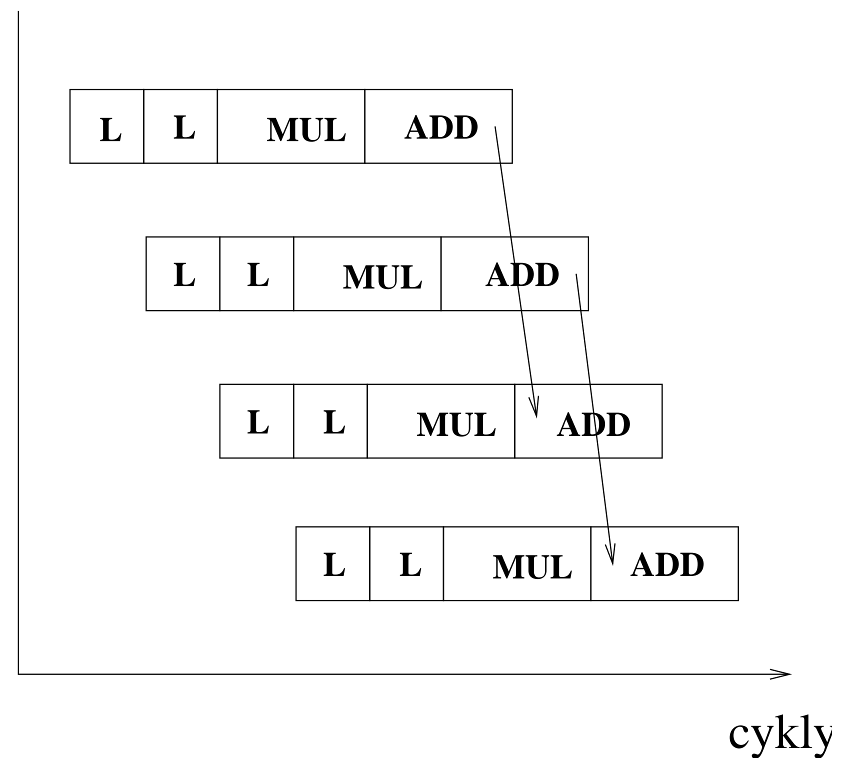
Na 2 prvky výsledné matice je třeba z paměti načíst $3n$ operandů = 25% úspora.

Efekty techniky RC&S

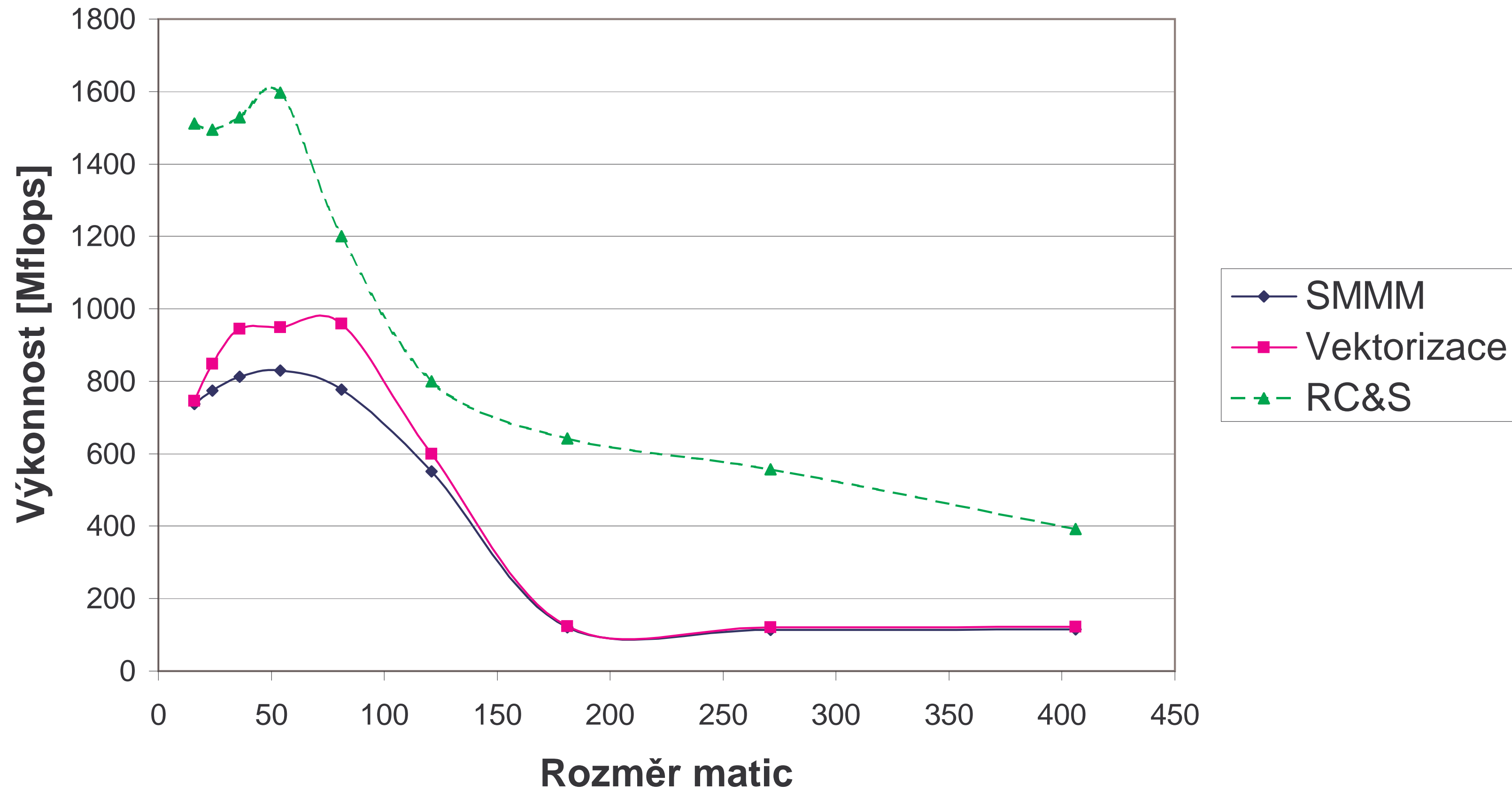
(1) Vícenásobné použití operandu $A[i][k]$.

(2) Pokud předpokládáme, že operace FMUL and FADD jsou 2-úrovňové, pak eliminace **prostojů** (*stalls*).



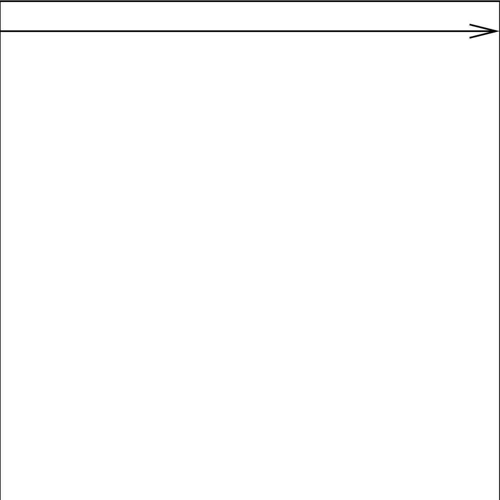


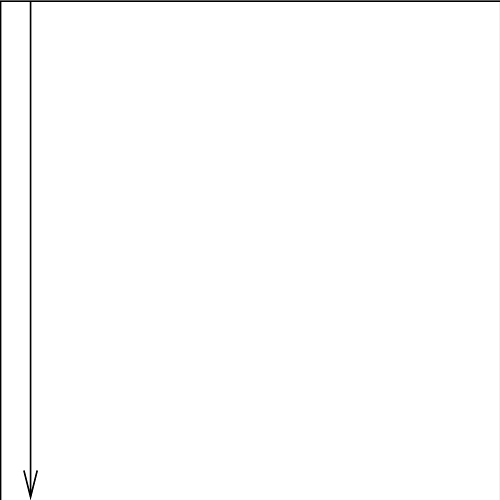
Výkonnost RC&S MMM (rank=4)

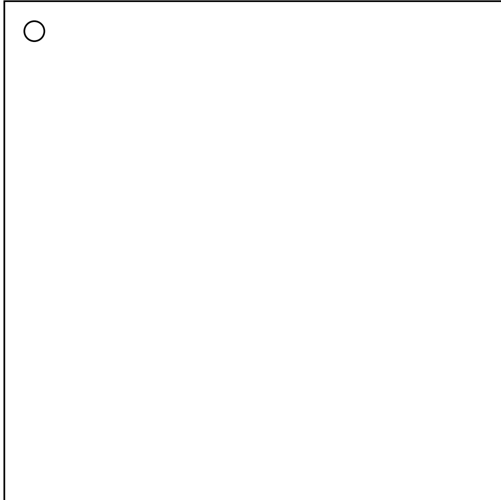


Blokové rozdělení a prolnutí cyklů (BRPC)

Standardní násobení SMMM (1.iterace)



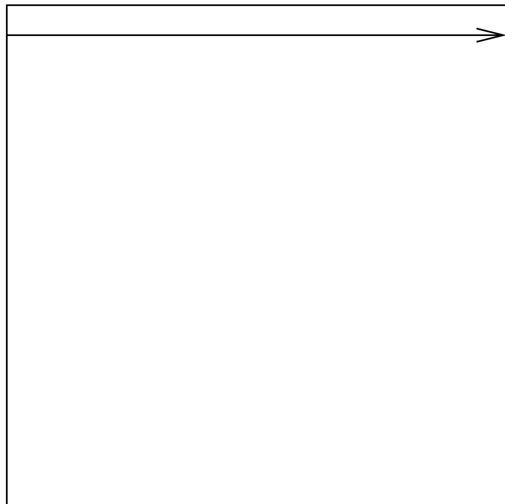
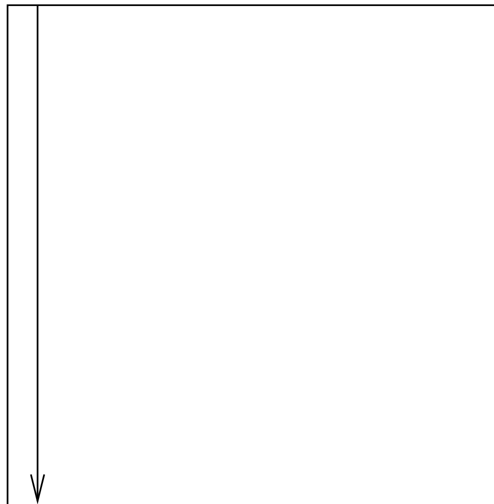
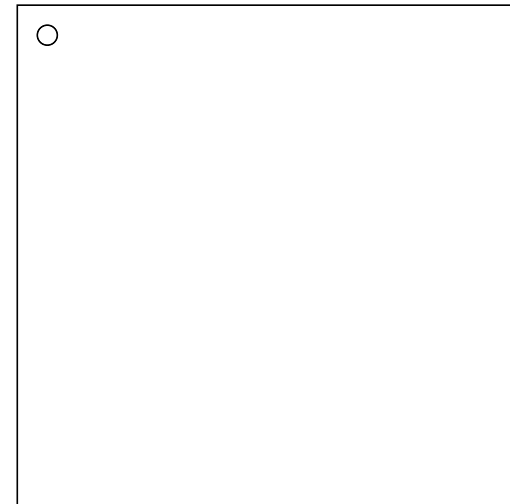




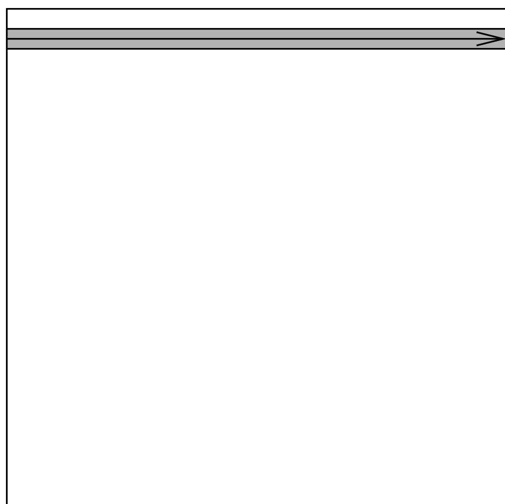
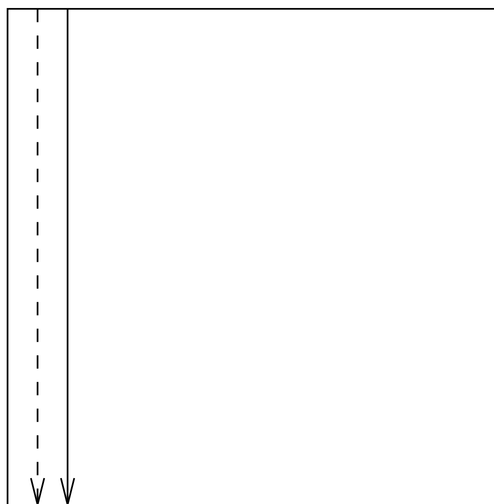
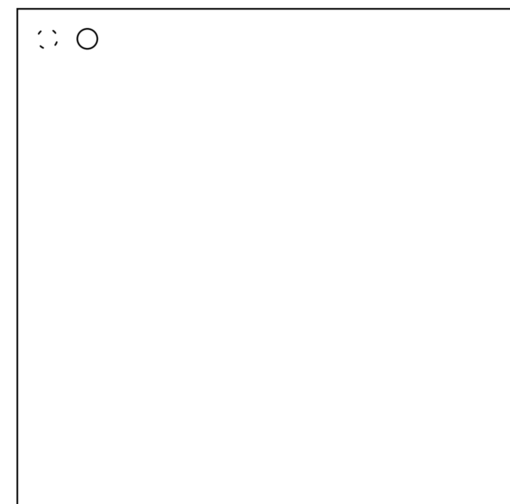
Blokové rozdělení a prolnutí cyklů (BRPC)

Standardní násobení SMMM (1. a 2. iterace)

matice A | matice B | matice C

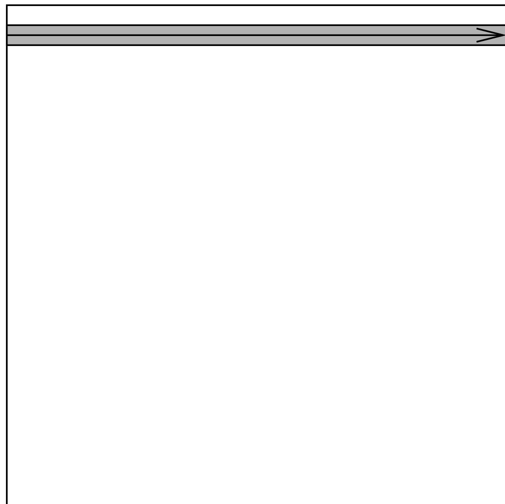




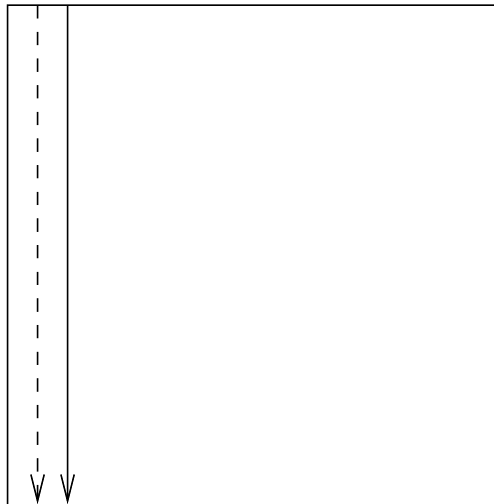
matice A | matice B | matice C

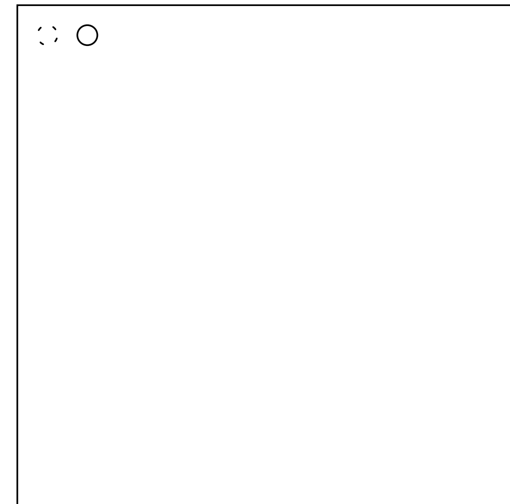




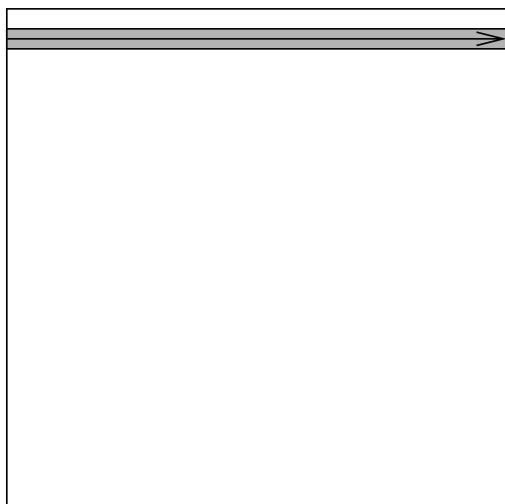
Blokové rozdělení a prolnutí cyklů (BRPC)

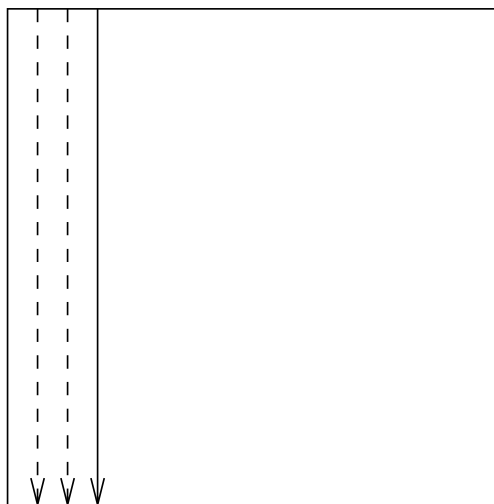
Standardní násobení SMMM (2. a 3. iterace)

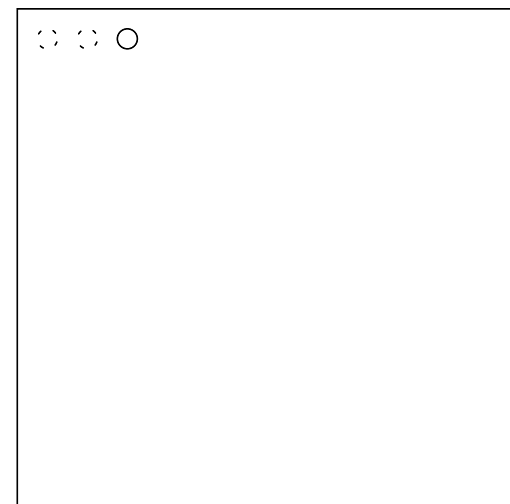








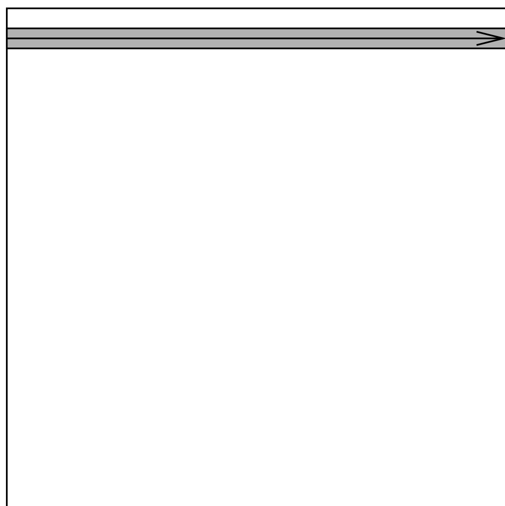
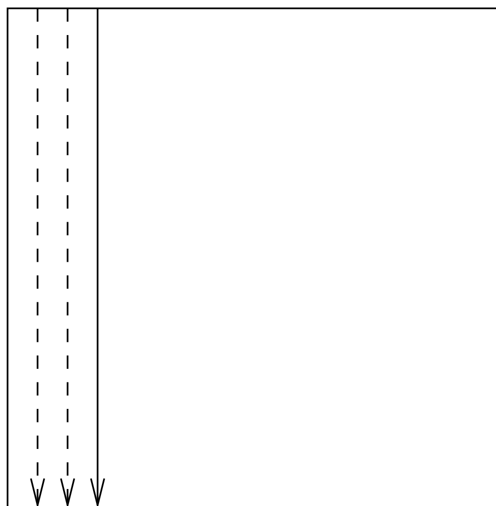
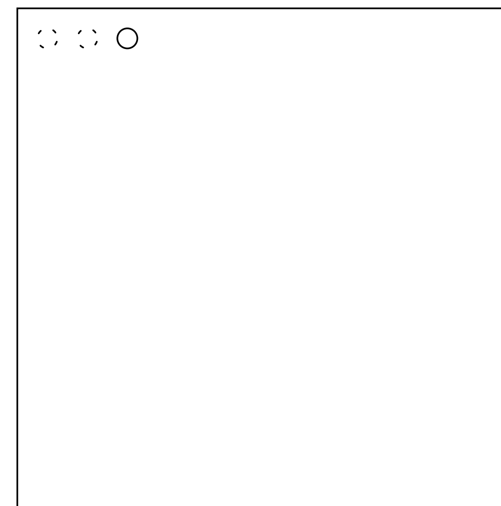




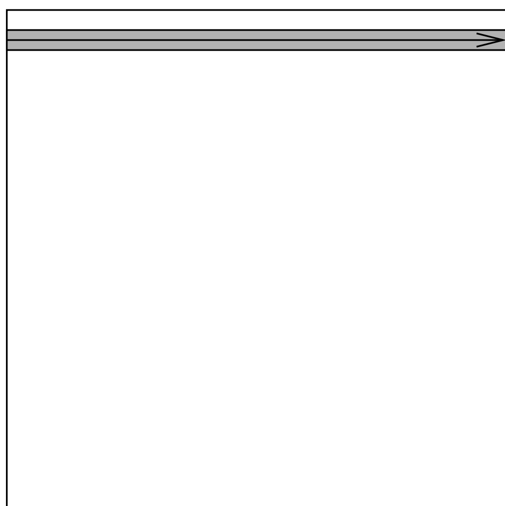
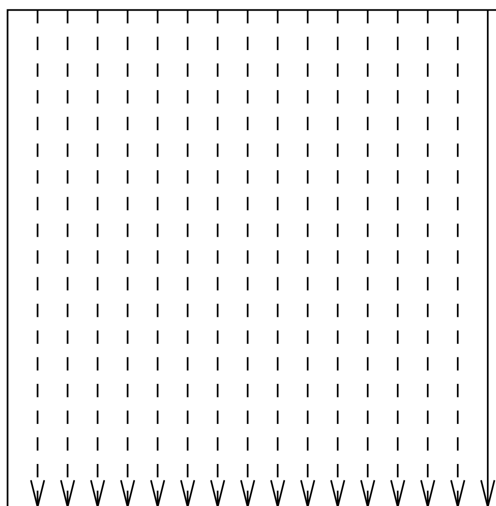
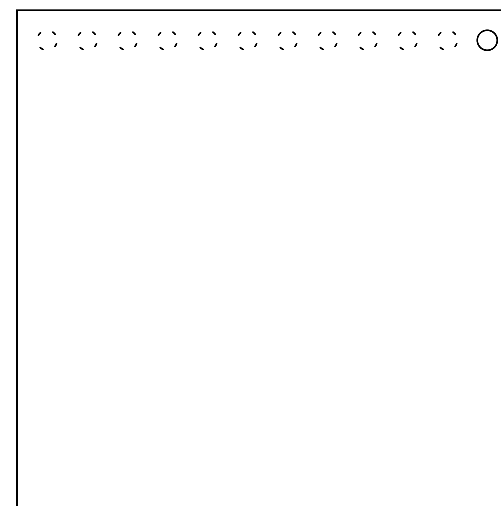
Blokové rozdělení a prolnutí cyklů (BRPC)

Standardní násobení SMMM (3. a $(n-1)$. iterace)

matice A matice B matice C

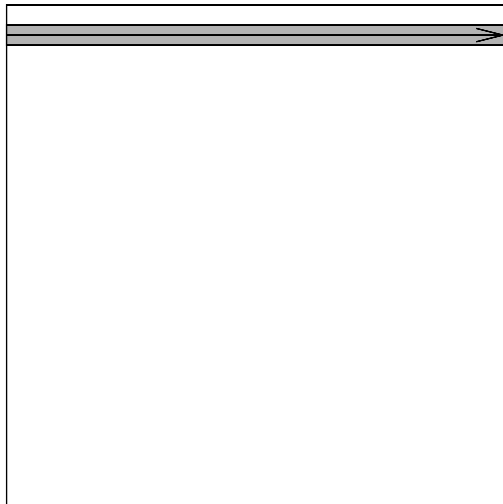




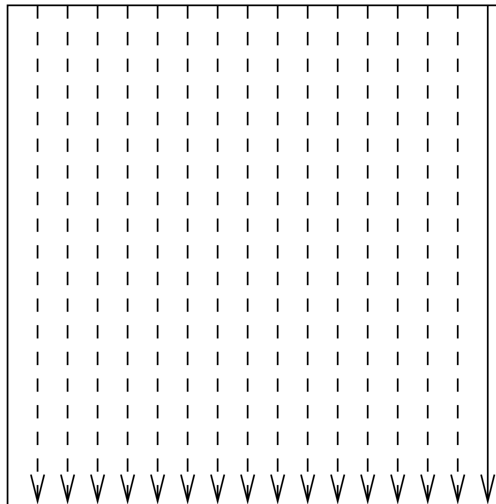
matice A matice B matice C

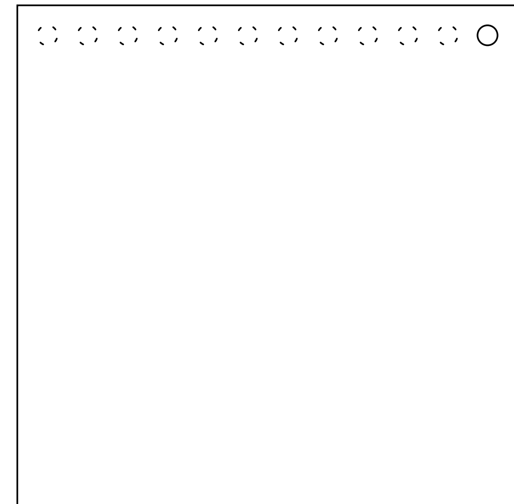




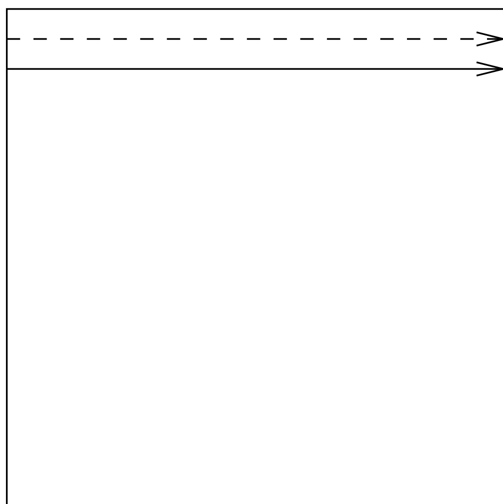
Blokové rozdělení a prolnutí cyklů (BRPC)

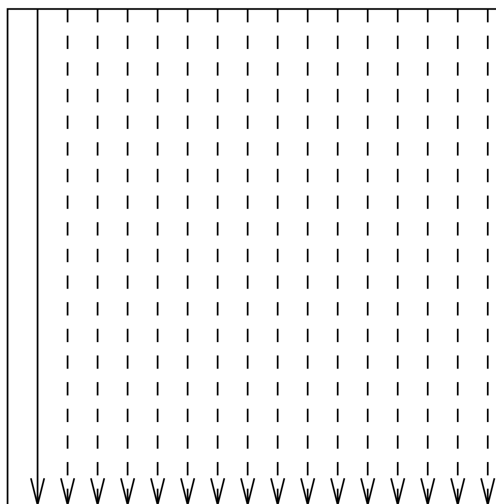
Standardní násobení SMMM ($(n-1)$. a n. iterace)

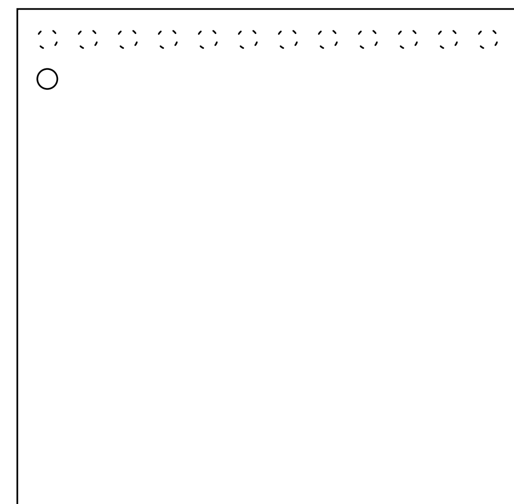






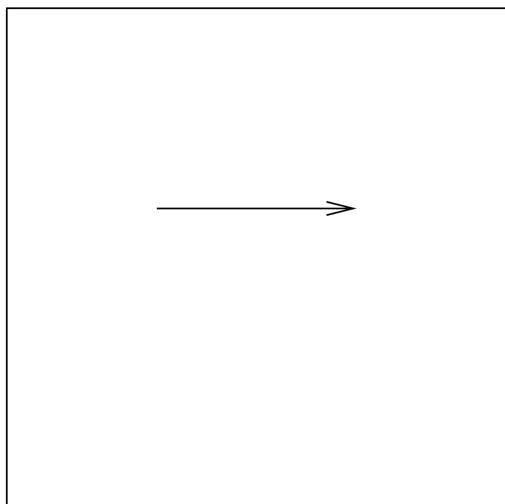


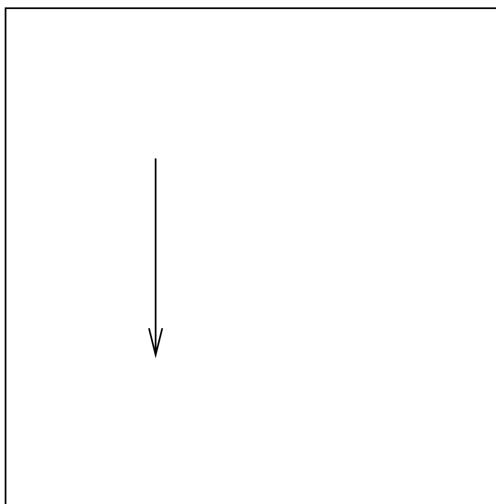


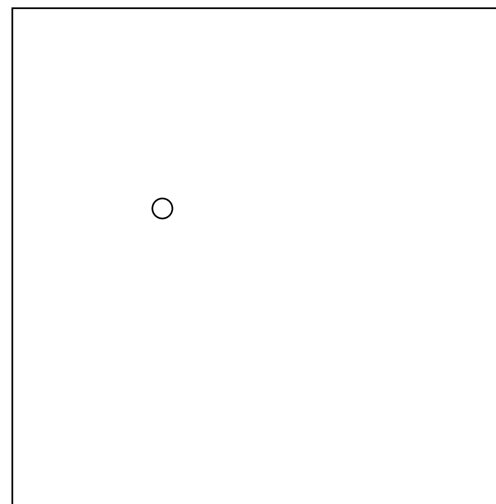


Blokové rozdělení a prolnutí cyklů (BRPC)

Násobení MMM s BRPC (1. iterace)

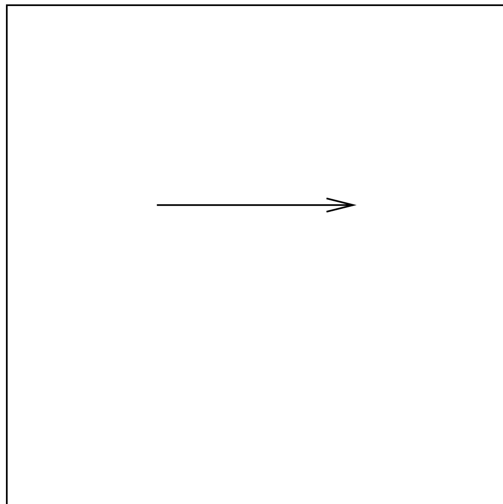


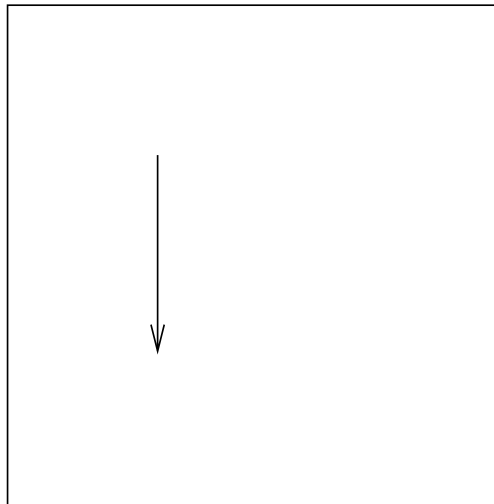


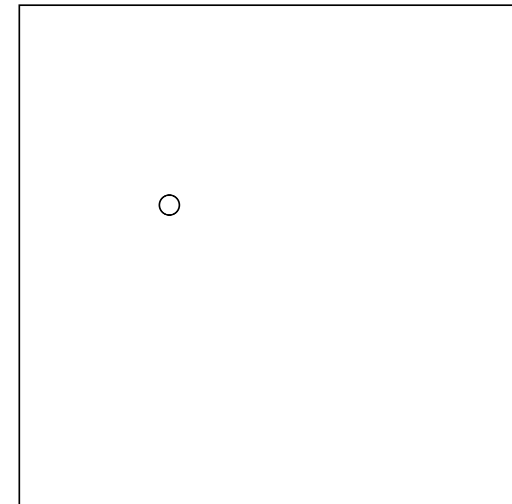


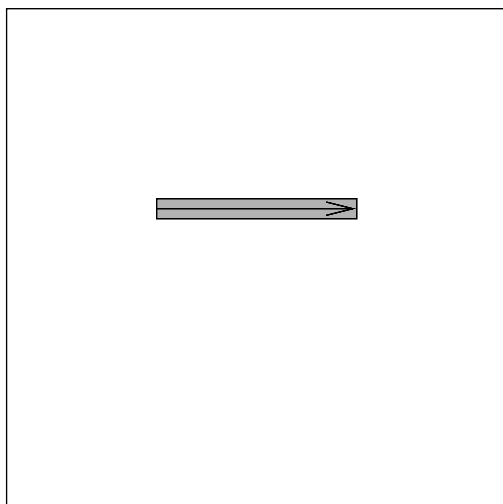
Blokové rozdělení a prolnutí cyklů (BRPC)

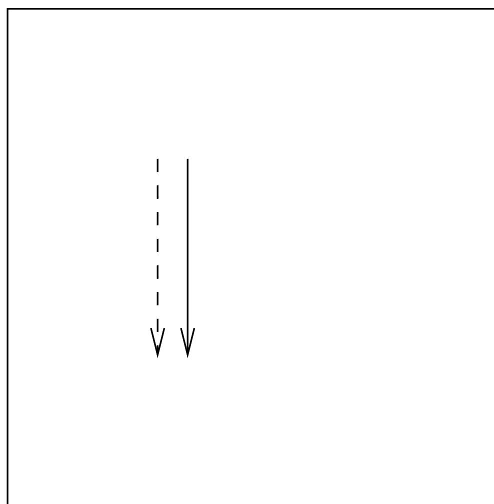
Násobení MMM s BRPC (1. a 2. iterace)

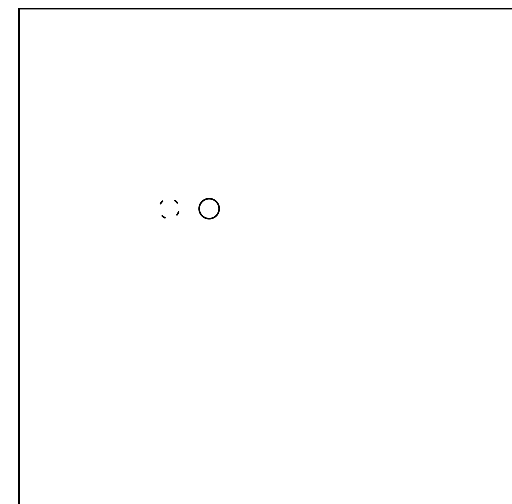








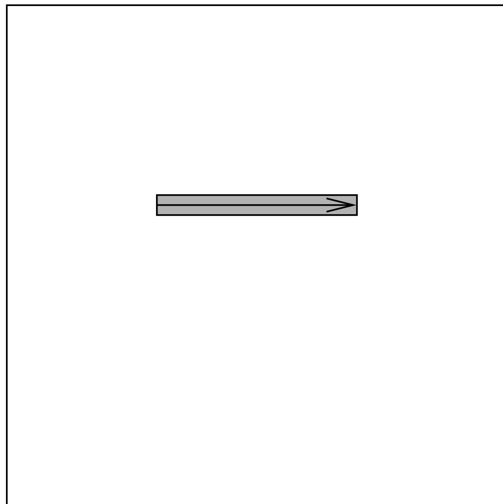


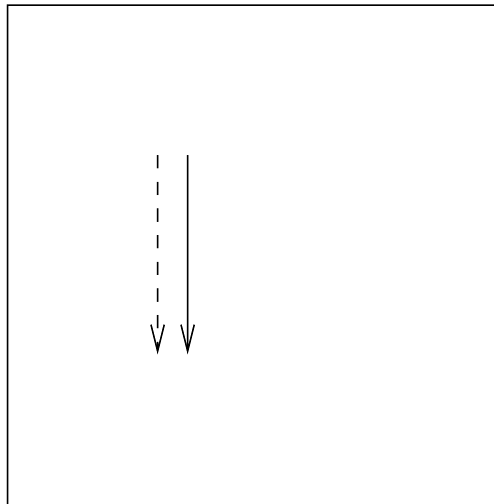


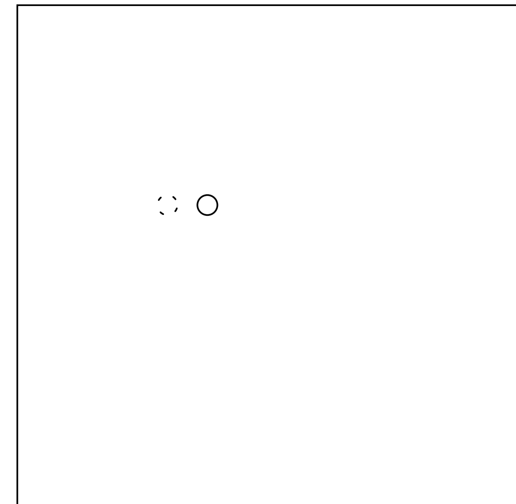
Blokové rozdělení a prolnutí cyklů (BRPC)

Násobení MMM s BRPC (2. a 3. iterace)

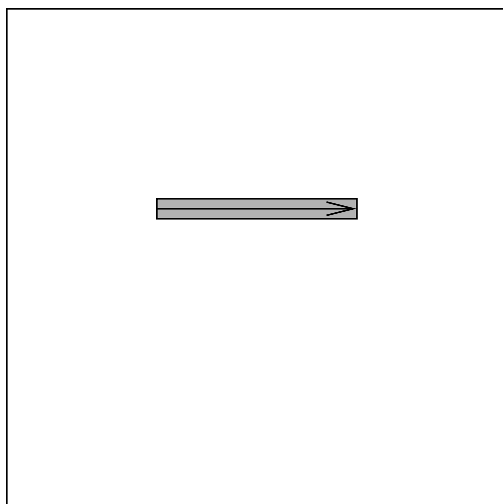
matice A | matice B | matice C

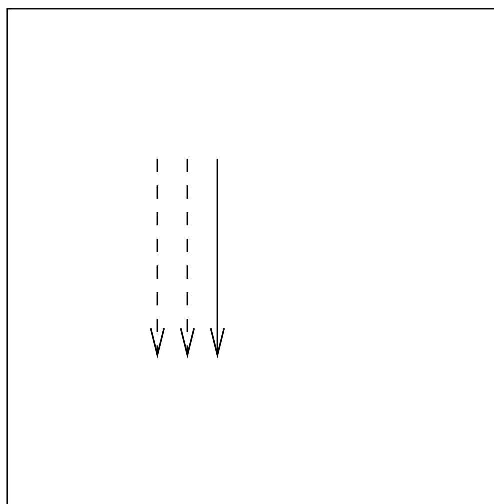


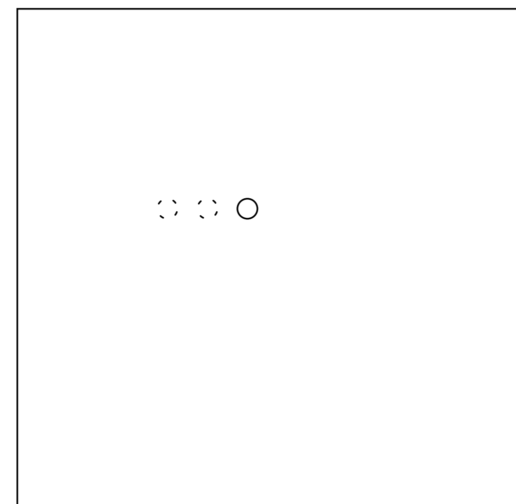




matice A | matice B | matice C



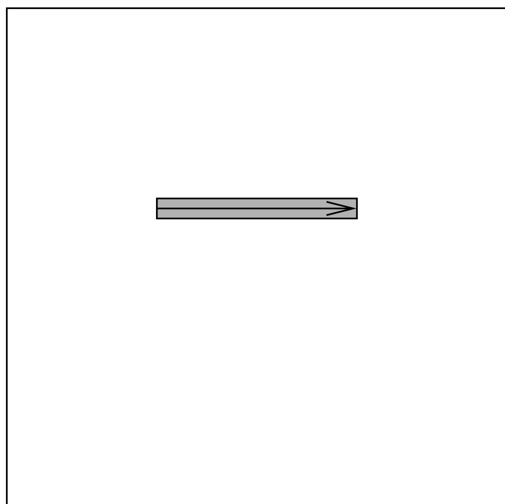
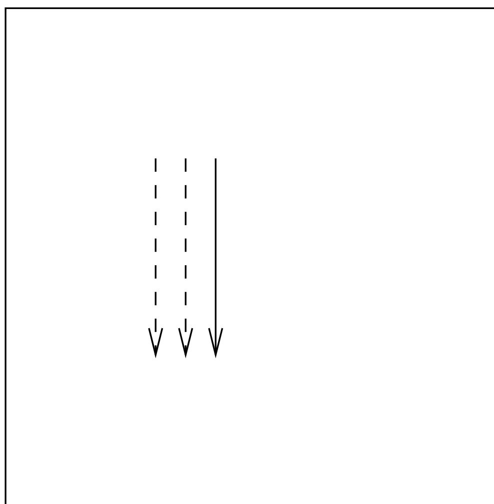
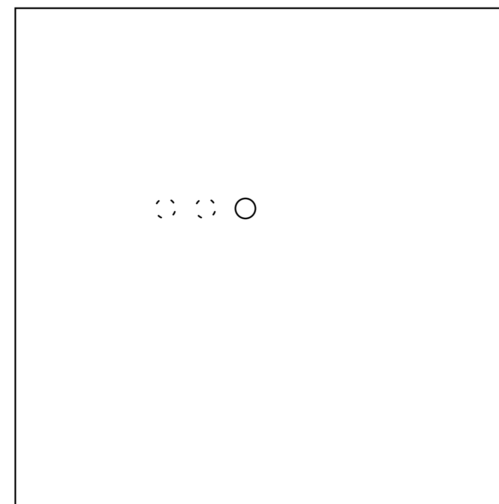




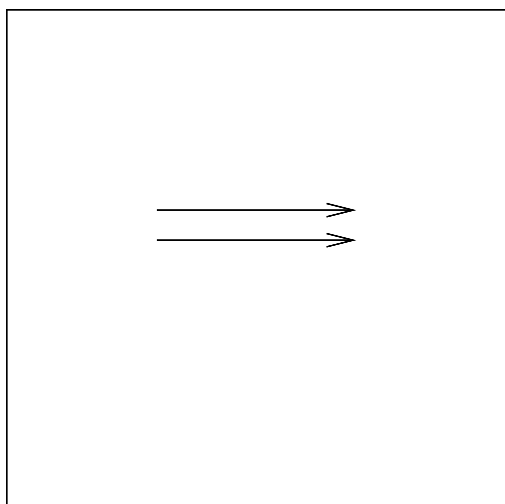
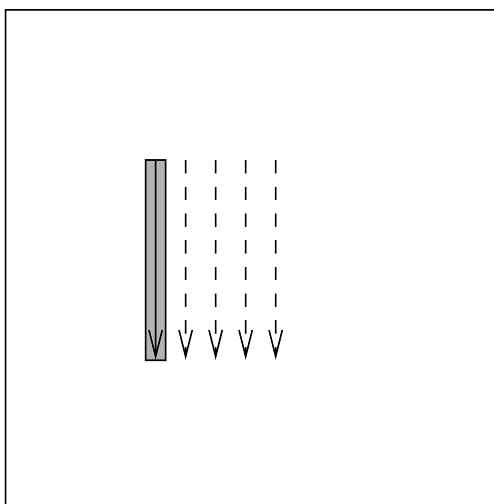
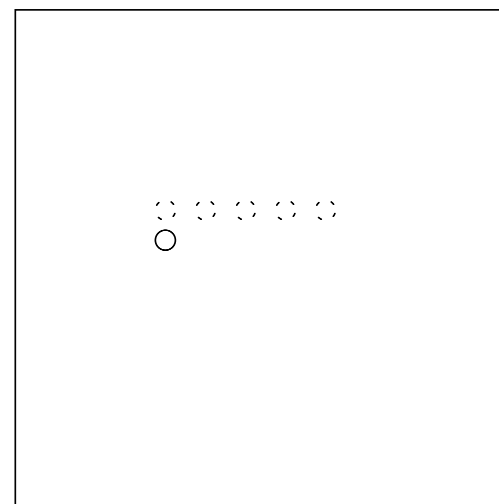
Blokové rozdělení a prolnutí cyklů (BRPC)

Násobení MMM s BRPC (3. a $blok$. iterace)

matice A | matice B | matice C

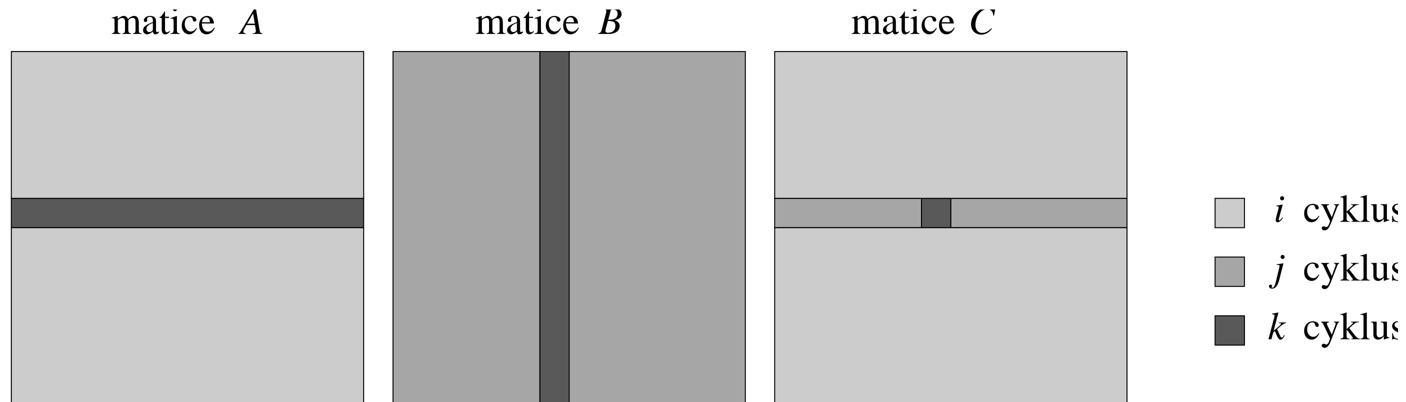




matice A | matice B | matice C

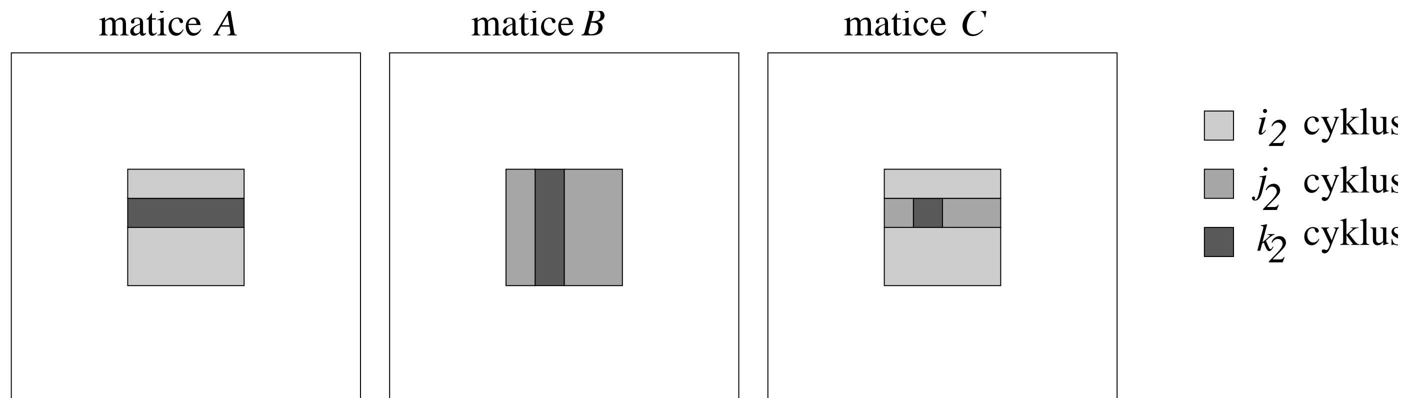




Blokové rozdělení a prolnutí cyklů (BRPC)

Standardní násobení SMMM: pro výpočet řádky C se musí načíst řádek A a celá B.



Násobení MMM s BRPC: pro výpočet příspěvků několika řádků C se načtený blok B použije vícenásobně.



Výpočet optimální velikosti bloku

Pro dobrou výkonnost kódu je třeba, aby skrytá paměť dokázala uchovat submatice $A'(blok, blok)$, $B'(blok, blok)$ a $C'(blok, blok)$.

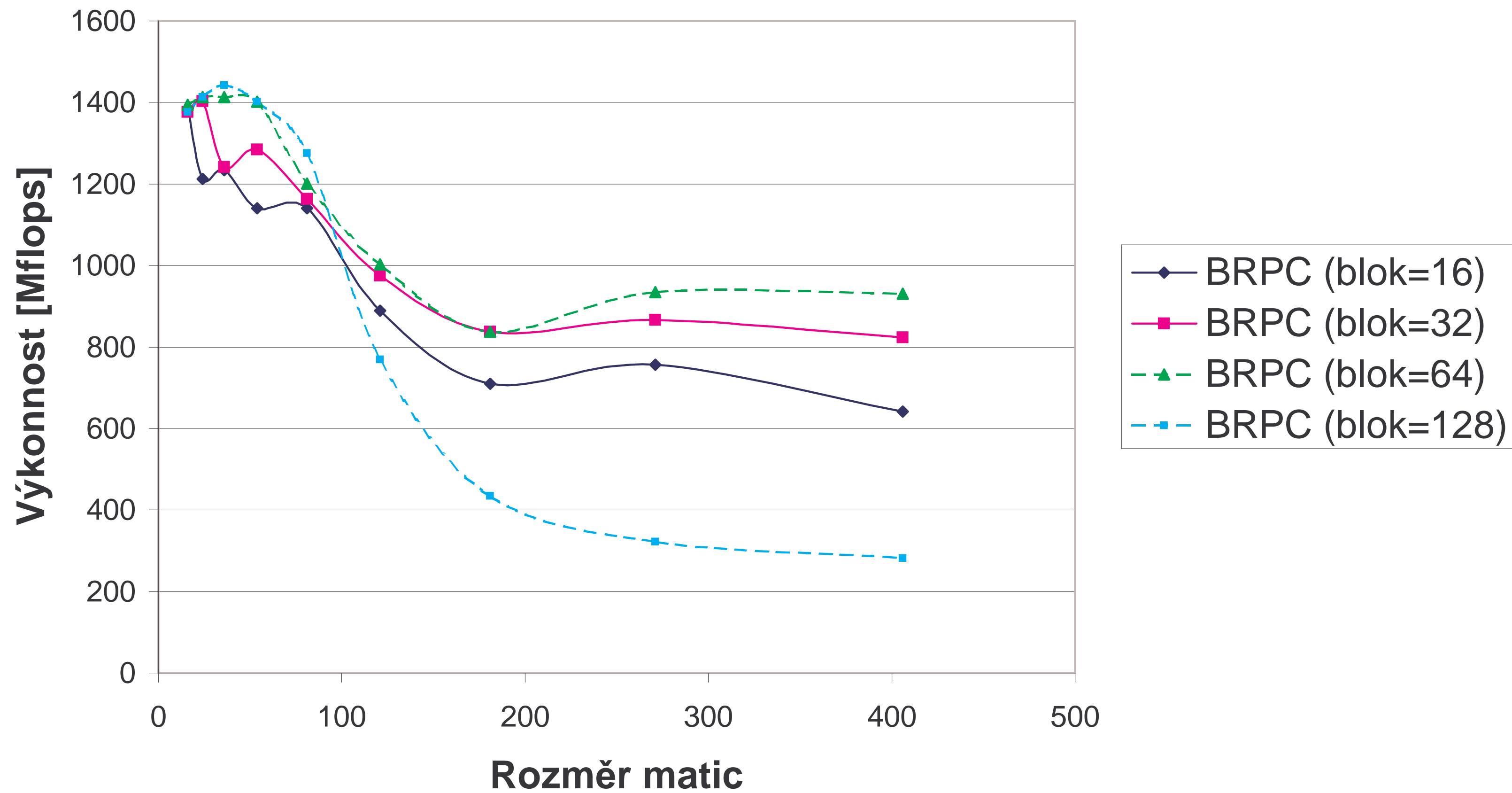
Proto musí být velikost skryté paměti větší než velikost těchto submatic.

$$C_\mathrm{S} > 3 \cdot blok^2 \cdot (\text{velikost elementu}).$$

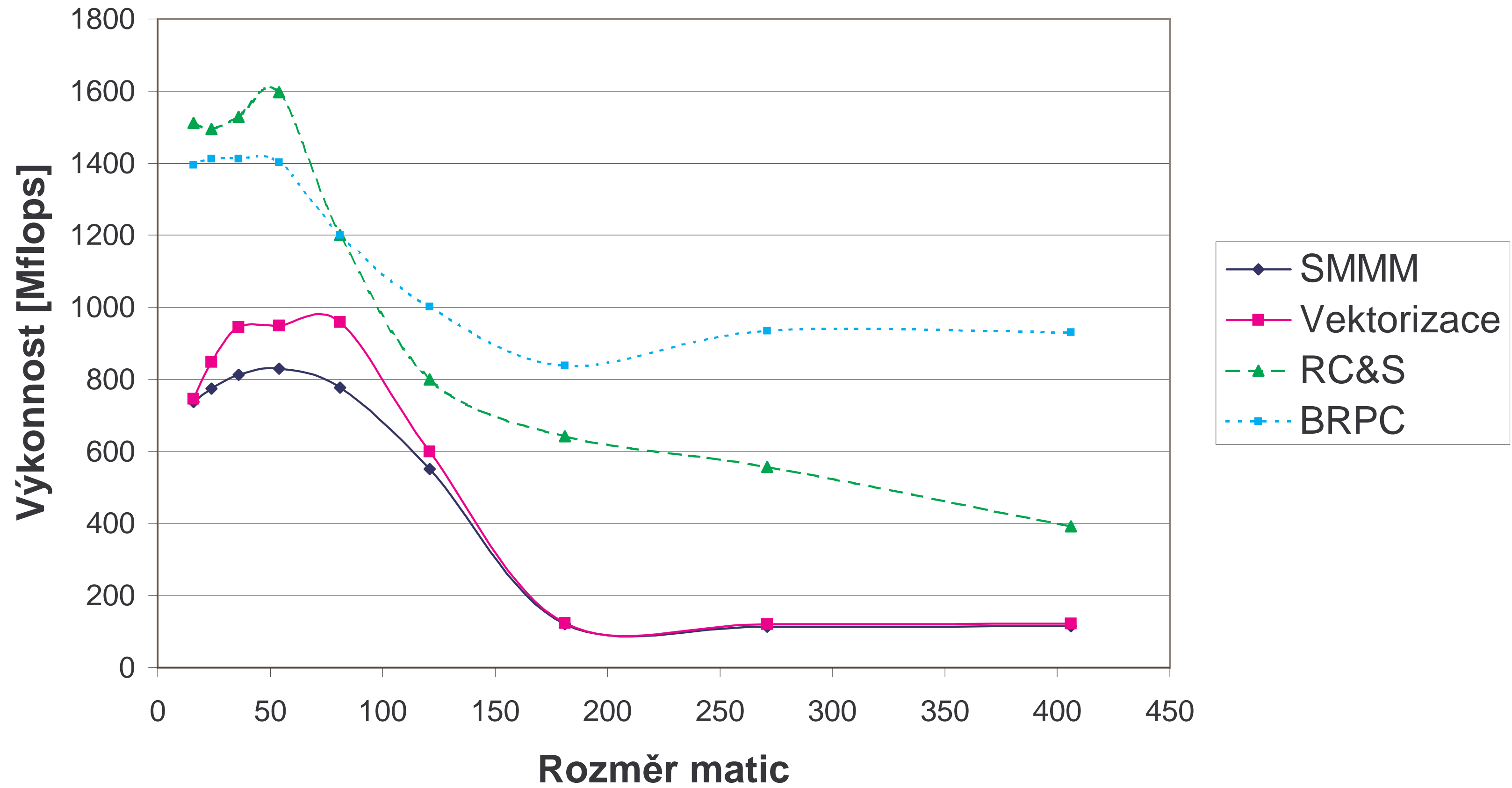
V našem případě

$$blok < 74.$$

Výkonnost MMM s BRPC



Výkonnost MMM s BRPC



Obrácení smyslu cyklu (OSC)

Klasický výpočet normy vektoru

Algorithm $\textsc{Norm}$(**in** *double* $a[1,\ldots,16]$; **out** a)

(1) $s = 0.0;$
(2) **for** $i = 1$ **to** 16 **do**
(3) $\quad s+ = a[i] * a[i];$
(4) $vel = \sqrt{s};$
(5) **for** $i = 1$ **to** 16 **do**
(6) $\quad a[i]/ = vel;$

Plnění skryté paměti s $h = 4$ a $s = 2$ v jednotlivých iteracích.

1)

a[1]	

Klasický výpočet normy vektoru

Algorithm $\textsc{Norm}$(**in** *double* $a[1,\ldots,16]$; **out** a)

(1) $s = 0.0;$
(2) **for** $i = 1$ **to** 16 **do**
(3) $\quad s+ = a[i] * a[i];$
(4) $vel = \sqrt{s};$
(5) **for** $i = 1$ **to** 16 **do**
(6) $\quad a[i]/ = vel;$

Plnění skryté paměti s $h = 4$ a $s = 2$ v jednotlivých iteracích.

1)

a[1]	

4)

a[1]	
a[2]	
a[3]	
a[4]	

Obrácení smyslu cyklu (OSC)

Klasický výpočet normy vektoru

Algorithm $\textsc{Norm}$(**in** *double* $a[1,\ldots,16]$; **out** a)

(1) $s = 0.0;$
(2) **for** $i = 1$ **to** 16 **do**
(3) $\quad s+ = a[i] * a[i];$
(4) $vel = \sqrt{s};$
(5) **for** $i = 1$ **to** 16 **do**
(6) $\quad a[i]/ = vel;$

Plnění skryté paměti s $h = 4$ a $s = 2$ v jednotlivých iteracích.

1)

a[1]	

4)

a[1]	
a[2]	
a[3]	
a[4]	

5)

a[1]	a[5]
a[2]	
a[3]	
a[4]	

Klasický výpočet normy vektoru

Algorithm NORM(**in** *double* $a[1,\ldots,16]$; **out** a)

(1) $s = 0.0;$
(2) **for** $i = 1$ **to** 16 **do**
(3) $\quad s+ = a[i] * a[i];$
(4) $vel = \sqrt{s};$
(5) **for** $i = 1$ **to** 16 **do**
(6) $\quad a[i]/ = vel;$

Plnění skryté paměti s $h = 4$ a $s = 2$ v jednotlivých iteracích.

1)

a[1]	

4)

a[1]	
a[2]	
a[3]	
a[4]	

5)

a[1]	a[5]
a[2]	
a[3]	
a[4]	

9)

a[9]	a[5]
a[2]	a[6]
a[3]	a[7]
a[4]	a[8]

Obrácení smyslu cyklu (OSC)

Klasický výpočet normy vektoru

Algorithm $\textsc{Norm}$(**in** *double* $a[1,\ldots,16]$; **out** a)

(1) $s = 0.0;$
(2) **for** $i = 1$ **to** 16 **do**
(3) $\quad s+ = a[i] * a[i];$
(4) $vel = \sqrt{s};$
(5) **for** $i = 1$ **to** 16 **do**
(6) $\quad a[i]/ = vel;$

Plnění skryté paměti s $h = 4$ a $s = 2$ v jednotlivých iteracích.

1)

a[1]	

4)

a[1]	
a[2]	
a[3]	
a[4]	

5)

a[1]	a[5]
a[2]	
a[3]	
a[4]	

9)

a[9]	a[5]
a[2]	a[6]
a[3]	a[7]
a[4]	a[8]

16)

a[9]	a[10]
a[11]	a[12]
a[13]	a[14]
a[15]	a[16]

První **for** cyklus vyvolá 16 povinných výpadků ve skryté paměti.

Obrácení smyslu cyklu (OSC)

Klasický výpočet normy vektoru

Algorithm $\textsc{Norm}$(**in** *double* $a[1,\ldots,16]$; **out** a)

(1) $s = 0.0;$
(2) **for** $i = 1$ **to** 16 **do**
(3) $\quad s += a[i] * a[i];$
(4) $vel = \sqrt{s};$
(5) **for** $i = 1$ **to** 16 **do**
(6) $\quad a[i] /= vel;$

Plnění skryté paměti s $h = 4$ a $s = 2$ v jednotlivých iteracích.

17)

a[1]	a[10]
a[11]	a[12]
a[13]	a[14]
a[14]	a[16]

Obrácení smyslu cyklu (OSC)

Klasický výpočet normy vektoru

Algorithm NORM(**in** *double* $a[1, \ldots, 16]$; **out** a)

(1) $s = 0.0;$
(2) **for** $i = 1$ **to** 16 **do**
(3) $\quad s += a[i] * a[i];$
(4) $vel = \sqrt{s};$
(5) **for** $i = 1$ **to** 16 **do**
(6) $\quad a[i] /= vel;$

Plnění skryté paměti s $h = 4$ a $s = 2$ v jednotlivých iteracích.

17)

a[1]	a[10]
a[11]	a[12]
a[13]	a[14]
a[14]	a[16]

32)

a[9]	a[10]
a[11]	a[12]
a[13]	a[14]
a[14]	a[16]

Druhý **for** cyklus vyvolá 16 konfliktních výpadků ve skryté paměti.

Výpočet normy vektoru s OSC

Algorithm NORMOSC(**in** *double* $a[1, \ldots, 16]$; **out** a)

(1) $s = 0.0;$
(2) **for** $i = 1$ **to** 16 **do**
(3) $\quad s+ = a[i] * a[i];$
(4) $vel = \sqrt{s};$
(5) **for** $i = 16$ **downto** 1 **do**
(6) $\quad a[i]/ = vel;$

Plnění skryté paměti s $h = 4$ a $s = 2$ v jednotlivých iteracích.

1)

a[1]	

Výpočet normy vektoru s OSC

Algorithm NORMOSC(**in** *double* $a[1, \ldots, 16]$; **out** a)

(1) $s = 0.0;$
(2) **for** $i = 1$ **to** 16 **do**
(3) $\quad s+ = a[i] * a[i];$
(4) $vel = \sqrt{s};$
(5) **for** $i = 16$ **downto** 1 **do**
(6) $\quad a[i]/ = vel;$

Plnění skryté paměti s $h = 4$ a $s = 2$ v jednotlivých iteracích.

1)

a[1]	

4)

a[1]	
a[2]	
a[3]	
a[4]	

Výpočet normy vektoru s OSC

Algorithm NormOSC(**in** *double* $a[1,\ldots,16]$; **out** a)

(1) $s = 0.0;$
(2) **for** $i = 1$ **to** 16 **do**
(3) $\quad s+ = a[i] * a[i];$
(4) $vel = \sqrt{s};$
(5) **for** $i = 16$ **downto** 1 **do**
(6) $\quad a[i]/ = vel;$

Plnění skryté paměti s $h = 4$ a $s = 2$ v jednotlivých iteracích.

1)

a[1]	

4)

a[1]	
a[2]	
a[3]	
a[4]	

5)

a[1]	a[5]
a[2]	
a[3]	
a[4]	

Výpočet normy vektoru s OSC

Algorithm NormOSC(**in** *double* $a[1,\ldots,16]$; **out** a)

(1) $s = 0.0;$
(2) **for** $i = 1$ **to** 16 **do**
(3) $\quad s+ = a[i] * a[i];$
(4) $vel = \sqrt{s};$
(5) **for** $i = 16$ **downto** 1 **do**
(6) $\quad a[i]/ = vel;$

Plnění skryté paměti s $h = 4$ a $s = 2$ v jednotlivých iteracích.

1)

a[1]	

4)

a[1]	
a[2]	
a[3]	
a[4]	

5)

a[1]	a[5]
a[2]	
a[3]	
a[4]	

9)

a[9]	a[5]
a[2]	a[6]
a[3]	a[7]
a[4]	a[8]

Výpočet normy vektoru s OSC

Algorithm $\textsc{NormOSC}$(**in** *double* $a[1,\ldots,16]$; **out** a)

(1) $s = 0.0;$
(2) **for** $i = 1$ **to** 16 **do**
(3) $\quad s+ = a[i] * a[i];$
(4) $vel = \sqrt{s};$
(5) **for** $i = 16$ **downto** 1 **do**
(6) $\quad a[i]/ = vel;$

Plnění skryté paměti s $h = 4$ a $s = 2$ v jednotlivých iteracích.

1)

a[1]	

4)

a[1]	
a[2]	
a[3]	
a[4]	

5)

a[1]	a[5]
a[2]	
a[3]	
a[4]	

9)

a[9]	a[5]
a[2]	a[6]
a[3]	a[7]
a[4]	a[8]

16)

a[9]	a[10]
a[11]	a[12]
a[13]	a[14]
a[15]	a[16]

První **for** cyklus vyvolá 16 povinných výpadků ve skryté paměti.

Výpočet normy vektoru s OSC

Algorithm NormOSC(**in** *double* $a[1,\ldots,16]$; **out** a)

(1) $s = 0.0;$
(2) **for** $i = 1$ **to** 16 **do**
(3) $\quad s+ = a[i] * a[i];$
(4) $vel = \sqrt{s};$
(5) **for** $i = 16$ **downto** 1 **do**
(6) $\quad a[i]/ = vel;$

Plnění skryté paměti s $h = 4$ a $s = 2$ v jednotlivých iteracích.

16)

a[9]	a[10]
a[11]	a[12]
a[13]	a[14]
a[15]	a[16]

Výpočet normy vektoru s OSC

Algorithm $\textsc{NormOSC}$(**in** *double* $a[1,\ldots,16]$; **out** a)

(1) $s = 0.0;$
(2) **for** $i = 1$ **to** 16 **do**
(3) $\quad s+ = a[i] * a[i];$
(4) $vel = \sqrt{s};$
(5) **for** $i = 16$ **downto** 1 **do**
(6) $\quad a[i]/ = vel;$

Plnění skryté paměti s $h = 4$ a $s = 2$ v jednotlivých iteracích.

16)

a[9]	a[10]
a[11]	a[12]
a[13]	a[14]
a[15]	a[16]

17)

a[9]	a[10]
a[11]	a[12]
a[13]	a[14]
a[15]	a[16]

Výpočet normy vektoru s OSC

Algorithm NORMOSC(**in** *double* $a[1,\dots,16]$; **out** a)

(1) $s = 0.0;$
(2) **for** $i = 1$ **to** 16 **do**
(3) $\quad s += a[i] * a[i];$
(4) $vel = \sqrt{s};$
(5) **for** $i = 16$ **downto** 1 **do**
(6) $\quad a[i] /= vel;$

Plnění skryté paměti s $h = 4$ a $s = 2$ v jednotlivých iteracích.

16)

a[9]	a[10]
a[11]	a[12]
a[13]	a[14]
a[15]	a[16]

17)

a[9]	a[10]
a[11]	a[12]
a[13]	a[14]
a[15]	a[16]

24)

a[9]	a[10]
a[11]	a[12]
a[13]	a[14]
a[15]	a[16]

Výpočet normy vektoru s OSC

Algorithm NormOSC(**in** *double* $a[1,\ldots,16]$; **out** a)

(1) $s = 0.0;$
(2) **for** $i = 1$ **to** 16 **do**
(3) $\quad s+ = a[i] * a[i];$
(4) $vel = \sqrt{s};$
(5) **for** $i = 16$ **downto** 1 **do**
(6) $\quad a[i]/ = vel;$

Plnění skryté paměti s $h = 4$ a $s = 2$ v jednotlivých iteracích.

16)

a[9]	a[10]
a[11]	a[12]
a[13]	a[14]
a[15]	a[16]

17)

a[9]	a[10]
a[11]	a[12]
a[13]	a[14]
a[15]	a[16]

24)

a[9]	a[10]
a[11]	a[12]
a[13]	a[14]
a[15]	a[16]

25)

a[9]	a[10]
a[11]	a[12]
a[13]	a[14]
a[8]	a[16]

Výpočet normy vektoru s OSC

Algorithm NormOSC(**in** *double* $a[1,\ldots,16]$; **out** a)

(1) $s = 0.0;$
(2) **for** $i = 1$ **to** 16 **do**
(3) $\quad s+ = a[i] * a[i];$
(4) $vel = \sqrt{s};$
(5) **for** $i = 16$ **downto** 1 **do**
(6) $\quad a[i]/ = vel;$

Plnění skryté paměti s $h = 4$ a $s = 2$ v jednotlivých iteracích.

16)

a[9]	a[10]
a[11]	a[12]
a[13]	a[14]
a[15]	a[16]

17)

a[9]	a[10]
a[11]	a[12]
a[13]	a[14]
a[15]	a[16]

24)

a[9]	a[10]
a[11]	a[12]
a[13]	a[14]
a[15]	a[16]

25)

a[9]	a[10]
a[11]	a[12]
a[13]	a[14]
a[8]	a[16]

32)

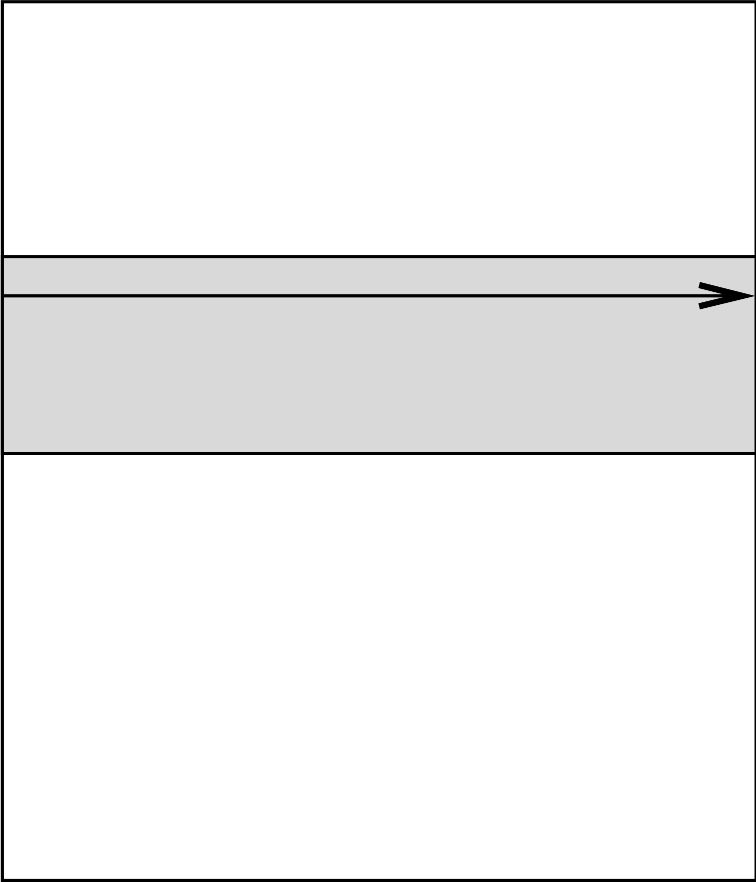
a[5]	a[1]
a[6]	a[2]
a[7]	a[3]
a[8]	a[4]

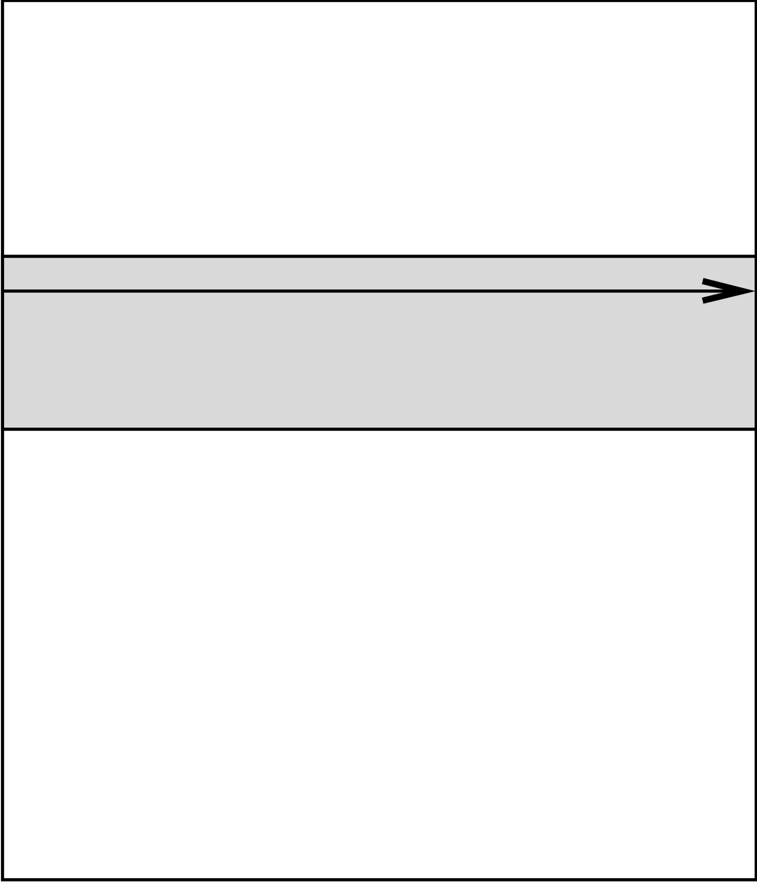
Druhý **for** cyklus vyvolá 8 nalezení a 8 konfliktních výpadků ve skryté paměti.

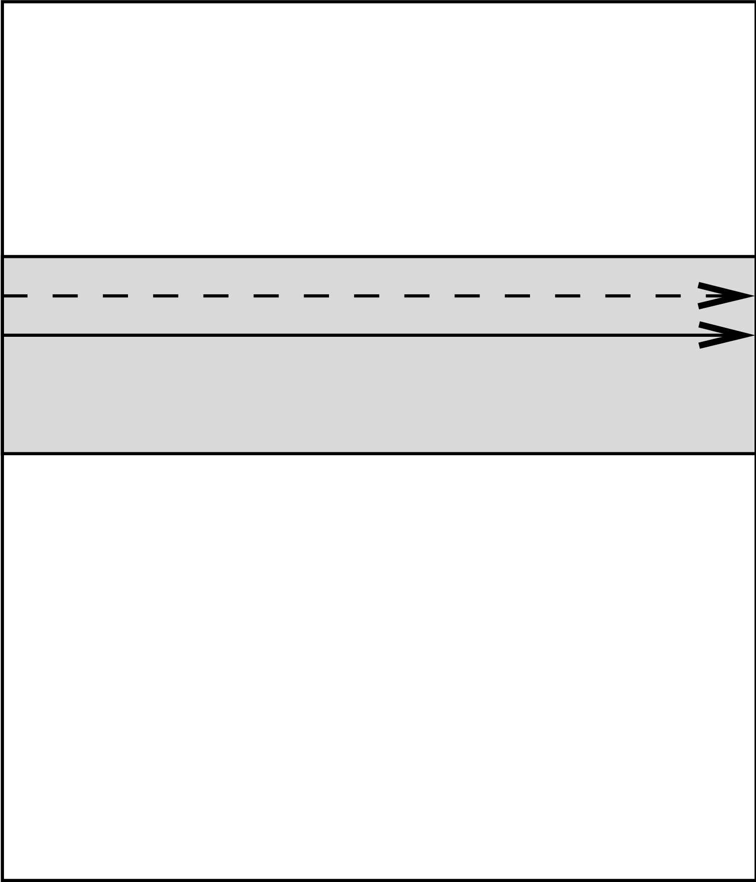
Dynamické OSC

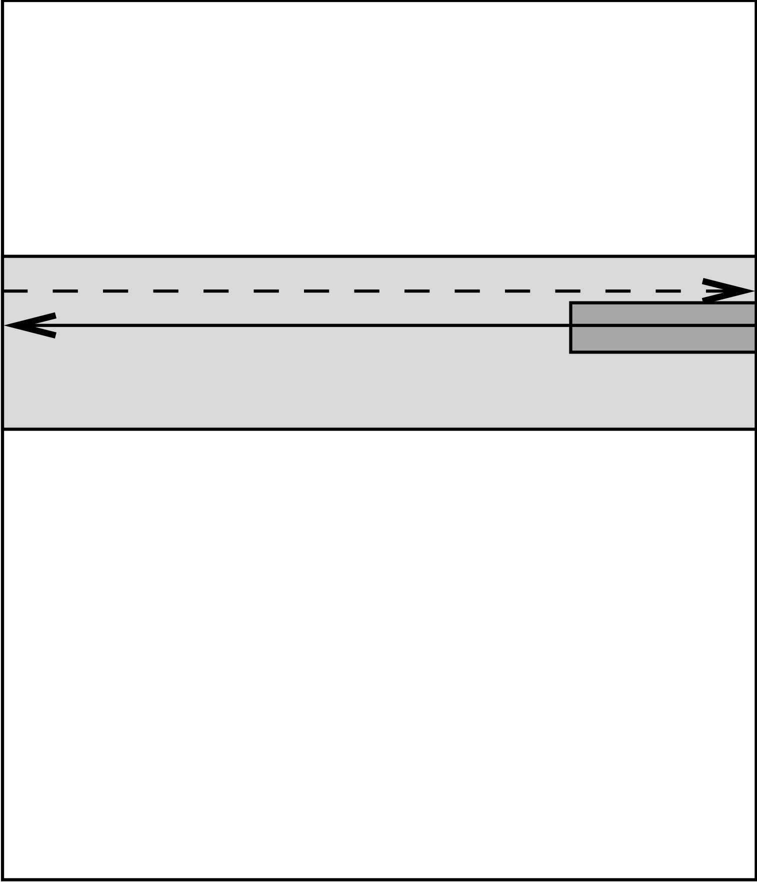
- Technika použitelná pro 2-úrovňový cyklus.
- Provádí OSC vnitřního v závislosti na paritě řídící proměnné vnějšího cyklu.

Dynamické OSC

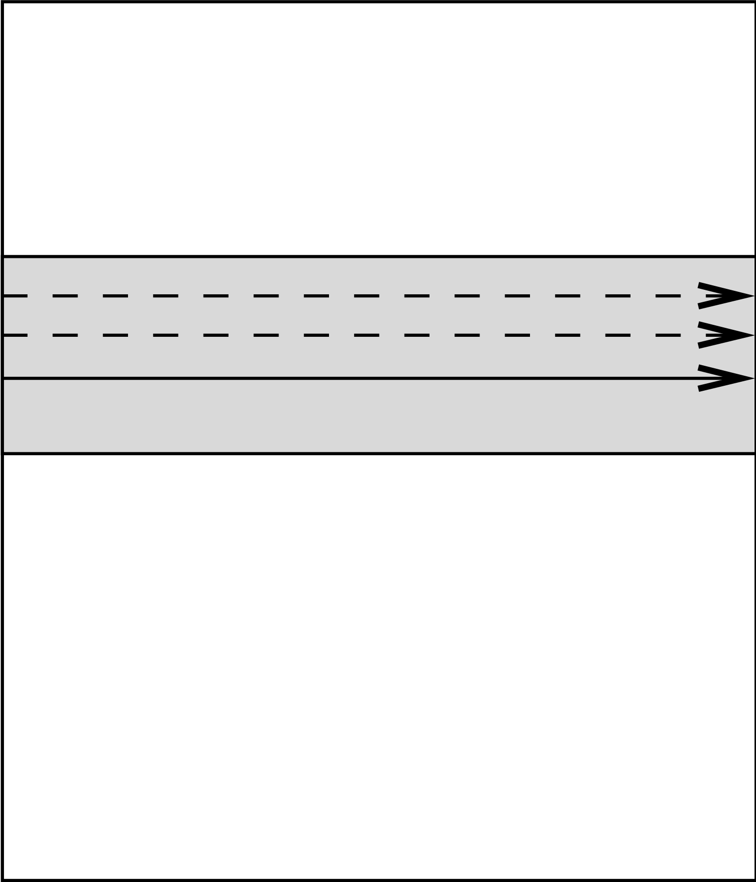


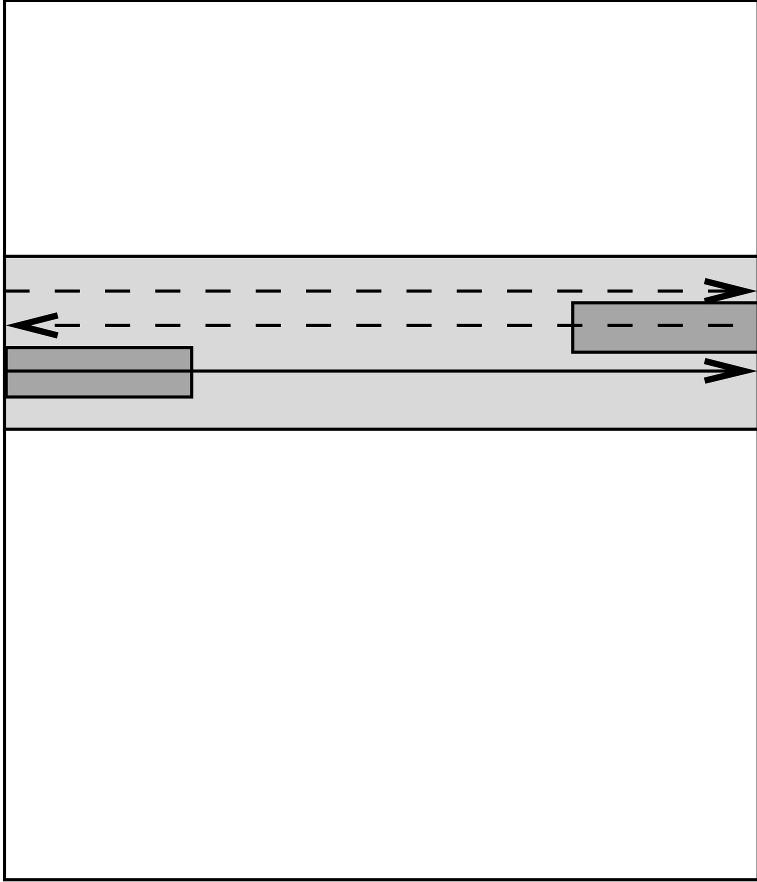




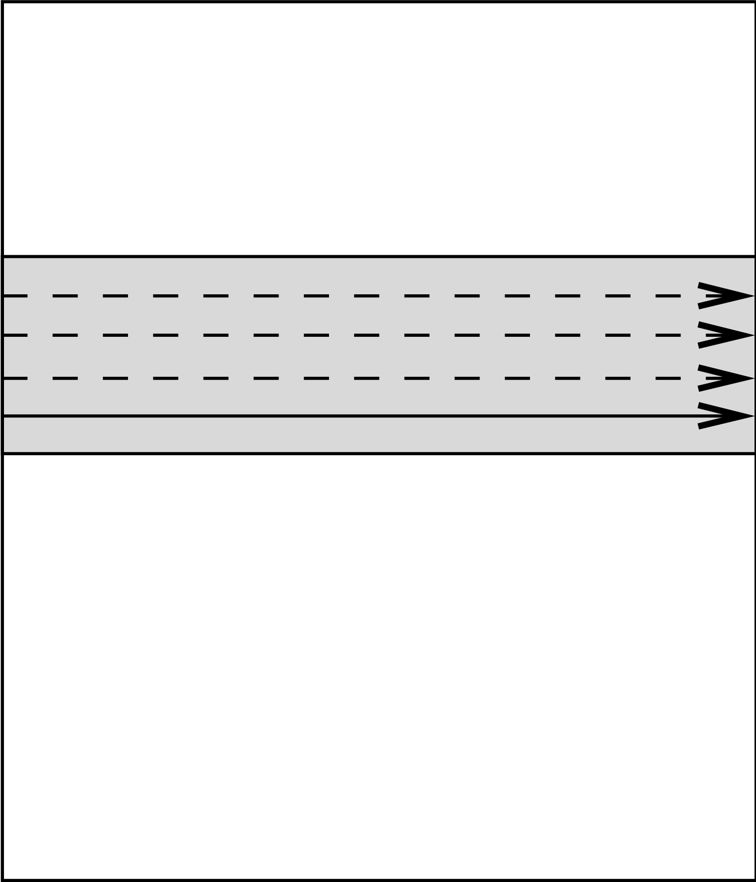


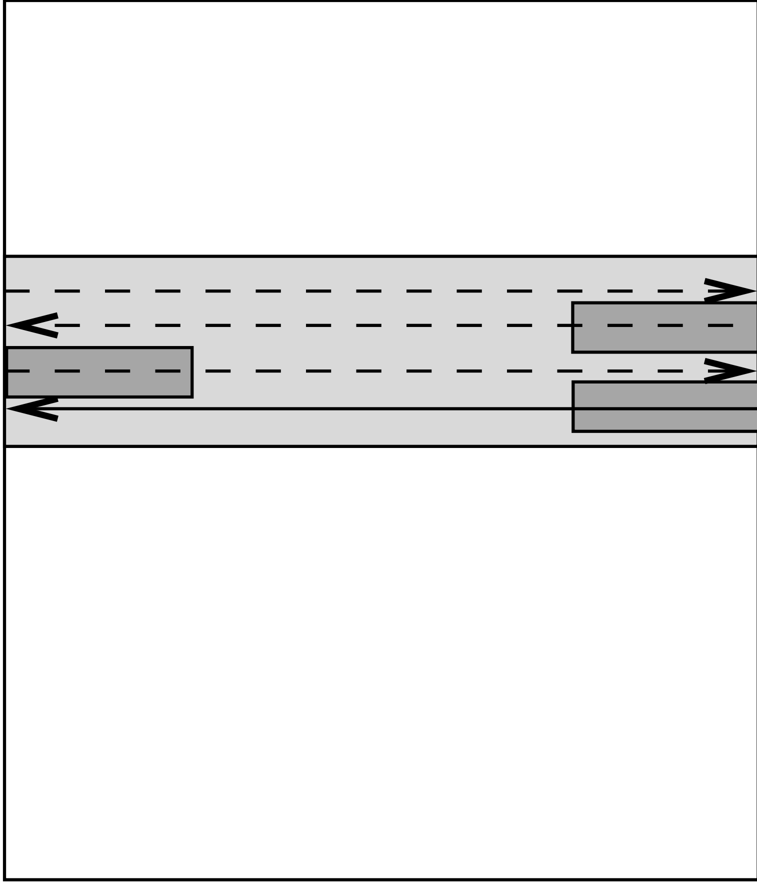
Dynamické OSC





Dynamické OSC





Pravděpodobnostní model pro chování skryté paměti

- Je třeba rozumně zvolit zjednodušující předpoklady.
- Umožňuje předpovídat počty výpadků ve skryté paměti během provádění algoritmu
  $\Longrightarrow$ lze použít pro výpočet parametrů optimální transformace kódů.
- Existují dva základní typy:

(1) pravděpodobnostní model,
(2) model využívající tzv. **přístupový interval** (*reuse distance*).

Model využívající přístupový interval (1)

Definice:

- **Posloupnost paměťových referencí** P: $P[t] = adr \iff$ paměťová transakce číslo t přistupovala na adresu adr.
- **Přístupový interval** $RD(t)$ = počet **různých** paměťových referencí mezi dvěma následnými použitími adresy $P[t]$.
  čili:
  je-li $P[t] = adr$ a $\delta > 0$ je minimální přirozené číslo takové, že $P[t+\delta] = adr$,
  pak $RD(t) = |\{P[t+1], \ldots, P[t+\delta-1]\}|$.

Pokud $RD(t) > C_{\mathrm{S}}$, pak je blok na adrese $P[t] = adr$ (který byl do skryté paměti nahrán v čase t) odstraněn před svým dalším znovupoužitím
$\Longrightarrow$ vznikne konfliktní výpadek ve skryté paměti při čtení či zápisu.

Omezení

- Přístupový interval podle definice je funkcí času, ale v praxi počítáme s jeho průměrnou hodnotou.
- Tento model nerespektuje mapovací funkci skryté paměti, počítá s plně asociativní skrytou pamětí ($h = 1$).

Knihovny pro LA

(1) Dostupné na každé počítačové platformě:

- volně šiřitelné (BLAS, LAPACK, ScaLAPACK),
- firemní (IBM ESSL, Intel MKL, Sun Performance Library).

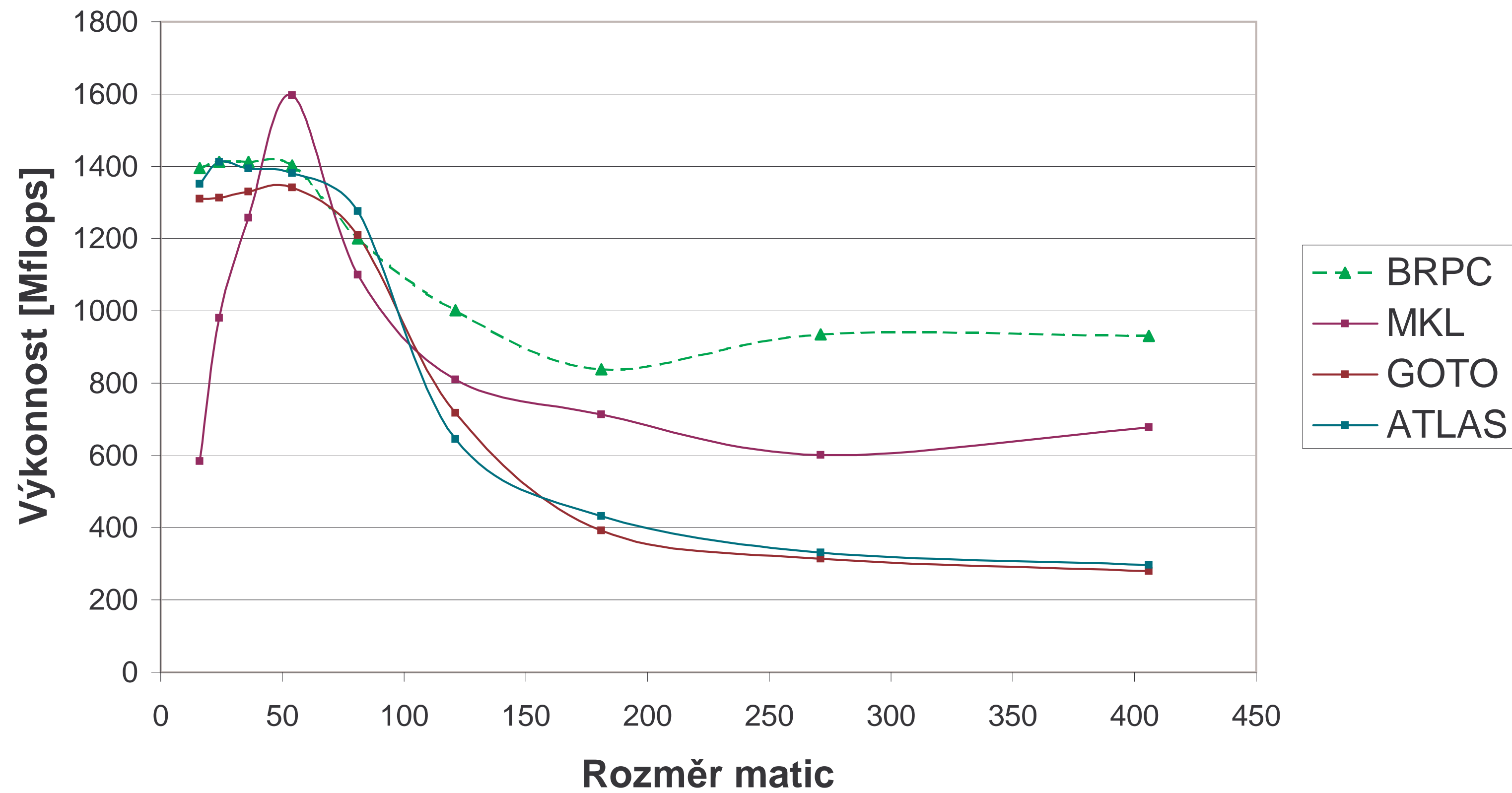
(2) Optimalizované pro architekturu procesoru.

(3) Mají zabudované techniky pro zvýšení přesnosti (např. pivotizace).

(4) Verze pro

- sekvenční výpočet,
- výpočet nad sdílenou pamětí,
- výpočet nad distribuovanou pamětí,
- výpočet na gridu.

Výkonnost MMM v knihovnách.



Možné nevýhody knihoven pro LA

- Nedostatečný počet poskytovaných metod a algoritmů.
- Nedostatečný počet podporovaných formátů.
- Neexistuje možnost ovlivnit způsob řešení uvnitř jednoho algoritmu.
- Poskytovaná přesnost a stabilita nemusí být dostatečná.
- Způsob řešení problémů může být příliš obecný.
- Kombinace jednotlivých výpočtů nemusí být paměťově optimální.
  Např., výpočet $\vec{a} = A\vec{x}$ a $\vec{b} = A^T\vec{x}$.

Vyhodnocení výsledků (1)

Násobení řídké matice vektorem

Formát komprimovaných řádků (CSR)

- Vektorizace nejde provést kvůli nepřímému adresování.
- Rozvinutí cyklu a sloučení vede ke zvýšení výkonnosti o 10% díky lepšímu využití vnitřní kaskády FPU jednotky.
- Rozdělení cyklů do bloků nemá smysl, protože není co znovupoužívat.
- Dynamické obracení směru cyklu nemá smysl, protože počet nenulových prvků v jednom řádku matice je příliš malý.

Jiné formáty uložení řídké matice

- Lze vymýšlet lepší formáty uložení řídké matice.
- Např. kombinovaný formát slučující výhody CSR a diagonálního uložení.
  - Adaptivní algoritmus pro $SpM \times V$.
  - Umožňuje efektivnější uložení řídkých matic a použití vektorových instrukcí.
  - Vede k dalšímu zvýšení výkonnosti o 15% v závislosti na velikosti matice.

Vyhodnocení výsledků (2)

Metoda CG

- Násobení $SpM \times V$ uvnitř iterací zabírá cca 70% celkového času.
- Jeho vylepšení vede ke zvýšení výkonnosti o 7% .
- Vektorizace lineárních operací s vektory vede ke zvýšení výkonnosti o 6% .
- Rozdělení cyklů do bloků nemá smysl.
- Obrácení směru cyklu u jedné operace s vektorem vede ke zvýšení výkonnosti o < 1%
- Datové propojení operací $A\vec{x} = \vec{b}$ a $\vec{b}\vec{x} = \alpha$ vede ke zvýšení výkonnosti o 6% .

Vyhodnocení výsledků (3)

Metoda BiCG

- Dvě násobení $SpM \times V$ uvnitř iterací zabírají cca 70% celkového času.
- Jejich vylepšení vede ke zvýšení výkonnosti o 7% .
- Vektorizace lineárních operací s vektory vede ke zvýšení výkonnosti o 6% .
- Rozdělení cyklů do bloků nemá smysl.
- Obracení směru cyklu u dvou operací s vektory vede ke zvýšení výkonnosti o < 1%
- Datové propojení operací $A\vec{x} = \vec{b}$ a $A^T\vec{x} = \vec{c}$ vede ke zvýšení výkonnosti o 30% , protože matice A se čte pouze jednou.

.

Vyhodnocení výsledků (4)

Choleskyho faktorizace

- Vektorizace vede ke zvýšení výkonnosti o 50%
- Rozvinutí cyklu a sloučení vede ke zvýšení výkonnosti o 30% díky lepšímu využití vnitřní kaskády FPU jednotky a díky znovupoužití některých operandů.
- Rozdělení cyklů do bloků vede ke zvýšení výkonnosti až o 90% pro velké rozměry matic.
- Dynamické obracení směru cyklu vede ke zvýšení výkonnosti o cca 3% v závislosti na velikosti matice.

Choleskyho faktorizace pro pásové matice

Použití rekurzivního algoritmu vede ke zvýšení výkonnosti o 20-200% v závislosti na velikosti matice díky odstranění globálních referencí.