Výkon GPU hardware

Jiří Filipovič

podzim 2013

Optimalizace přístupu do globální paměti

Rychlost globální paměti se snadno stane bottleneckem

- šířka pásma globální paměti je ve srovnání s aritmetickým výkonem GPU malá (G200 $\geq$ 24 flops/float, G100 $\geq$ 30, GK110 $\geq$ 62)
- latence 400-600 cyklů

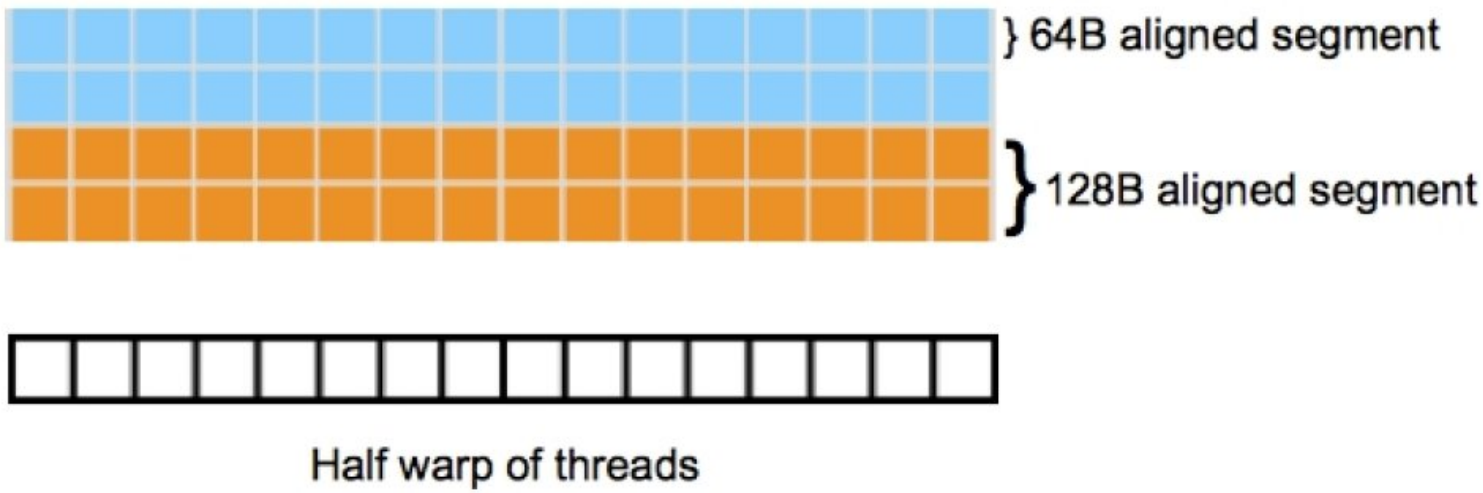
Při špatném vzoru paralelního přístupu do globální paměti snadno výrazně snížíme propustnost

- k paměti je nutno přistupovat spojitě (*coalescing*)
- je vhodné vyhnout se užívání pouze podmnožiny paměťových regionů (*partition camping*)

Spojitý přístup do paměti (c.c. < 2.0)

Rychlost GPU paměti je vykoupena nutností přistupovat k ní po větších blocích

- globální paměť je dělena do 64-bytových segmentů
- ty jsou sdruženy po dvou do 128-bytových segmentů



Spojitý přístup do paměti (c.c. < 2.0)

Polovina warpu může přenášet data pomocí jedné transakce či jedné až dvou transakcí při přenosu 128-bytového slova

- je však zapotřebí využít přenosu velkých slov
- jedna paměťová transakce může přenášet 32-, 64-, nebo 128-bytová slova
- u GPU s c.c. < 1.2
  - blok paměti, ke kterému je přistupováno, musí začínat na adrese dělitelné šestnáctinásobkem velikosti datových elementů
  - k-tý thread musí přistupovat ke k-tému elementu bloku
  - některé thready nemusejí participovat
- v případě, že nejsou tato pravidla dodržena, je pro každý element vyvolána zvláštní paměťová transakce

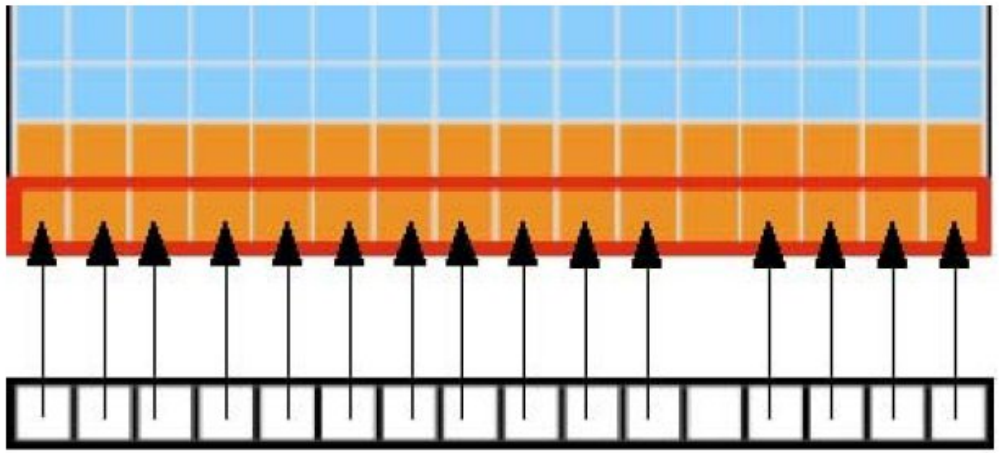
Spojitý přístup do paměti (c.c. < 2.0)

GPU s c.c. ≥ 1.2 jsou méně restriktivní

- přenos je rozdělen do 32-, 64-, nebo 128-bytových transakcí tak, aby byly uspokojeny všechny požadavky co nejnižším počtem transakcí
- pořadí threadů může být vzhledem k přenášeným elementům libovolně permutované

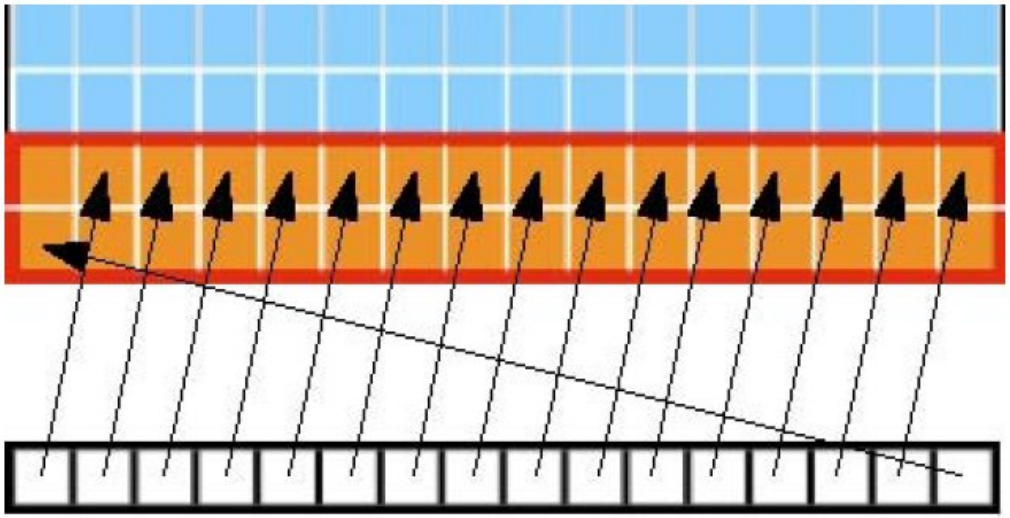
Spojitý přístup do paměti (c.c. < 2.0)

Thready jsou zarovnané, blok elementů souvislý, pořadí není permutované – spojitý přístup na všech GPU.



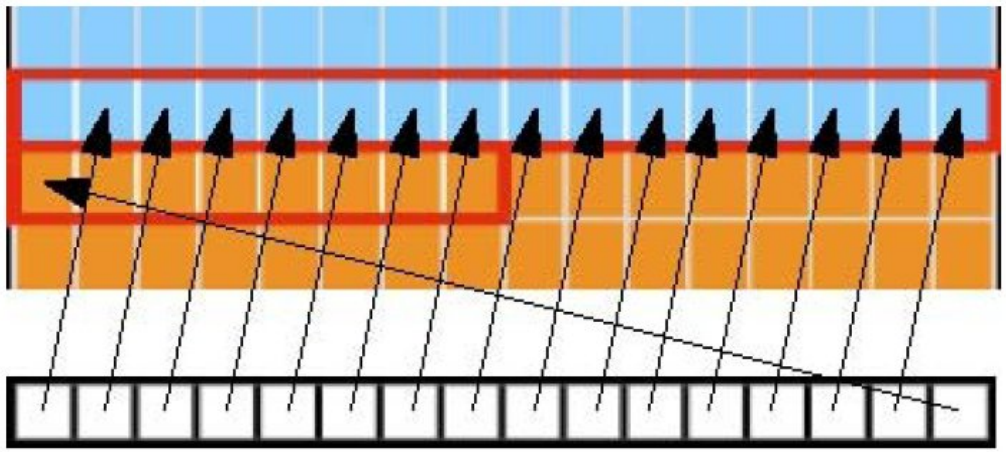
Nezarovnaný přístup do paměti (c.c. < 2.0)

Thready **nejsou** zarovnané, blok elementů souvislý, pořadí není permutované – jedna transakce na GPU s c.c. ≥ 1.2.



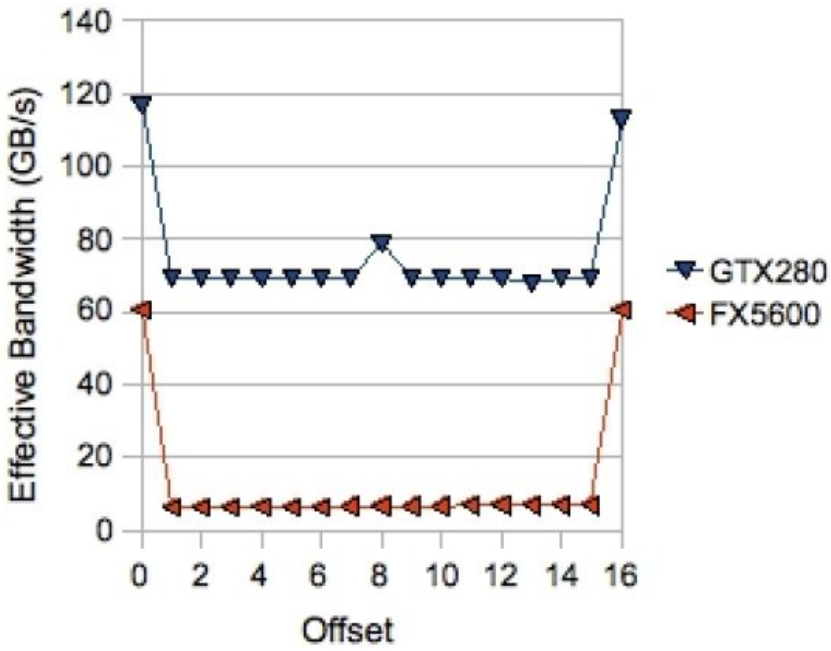
Nezarovnaný přístup do paměti (c.c. < 2.0)

Obdobný případ může vézt k použití dvou transakcí.



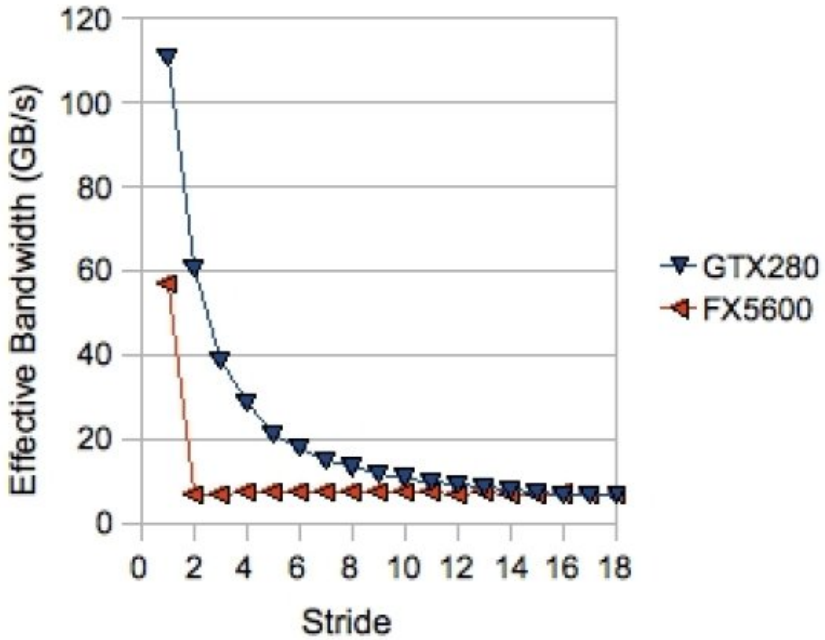
Výkon při nezarovnaném přístupu (c.c. < 2.0)

Starší GPU provádí pro každý element nejmenší možný přenos, tedy 32-bytů, což redukuje výkon na 1/8.
Nové GPU (c.c. ≥ 1.2) provádí dva přenosy.



Výkon při prokládaném přístupu (c.c. < 2.0)

GPU s c.c. ≥ 1.2 mohou přenášet data s menšími ztrátami pro menší mezery mezi elementy, se zvětšováním mezer výkon dramaticky klesá.



Přístup do globální paměti u Fermi (c.c. ≥ 2.0)

Fermi má L1 a L2 cache

- L1: 128 byte na řádek, celkem 16 KB nebo 48 KB na multiprocesor
- L2: 32 byte na řádek, celkem 768 KB na GPU

Jaké to přináší výhody?

- efektivnější programy s nepředvídatelnou datovou lokalitou
- nezarovnaný přístup – v principu žádné spomalení
- prokládaný přístup – data musí být využita dříve, než zmizí z cache, jinak stejný či větší problém jako u c.c. < 2.0 (L1 lze vypnout pro zamezení overfetchingu)

Přístup do globální paměti u Kepler (c.c. ≥ 3.0)

Kepler používá pro obecný přístup pouze L2 cache

- L1: pouze pro lokální paměť, celkem 16 KB, 32 KB nebo 48 KB
- L2: 32 byte na řádek, až 1.5 GB na GPU

Data cache

- sdílená s texturami, podporuje c.c. ≥ 3.5
- pro data pouze ke čtení
- může rozeznat kompilátor, pomůžeme mu pomocí `const __restrict__` či explicitně `__ldg()`

Partition camping

- relevantní pro c.c. 1.x
- procesory založené na G80 mají 6 regionů, G200 mají 8 regionů globální paměti
- paměť je dělena do regionů po 256-bytech
- pro maximální výkon je zapotřebí, aby bylo přistupováno rovnoměrně k jednotlivým regionům
  - mezi jednotlivými bloky
  - ty se zpravidla spouští v uspořádání daném polohou bloku v gridu
- pokud je využívána jen část regionů, nazýváme jev *partition camping*
- obecně ne tak kritické, jako spojitý přístup
- záludnější, závislé na velikosti problému

HW organizace sdílené paměti

Sdílená paměť je organizována do paměťových bank, ke kterým je možné přistupovat paralelně

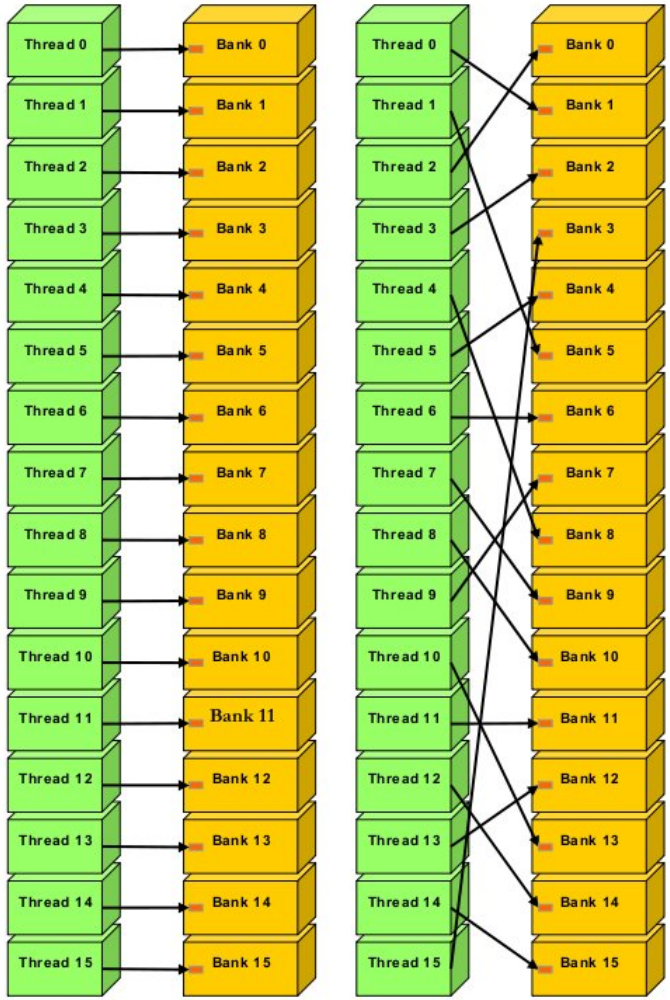
- c.c. 1.x 16 bank, c.c. 2.x a 3.x 32 bank, paměťový prostor mapován prokládaně s odstupem 32 bitů či 64 bitů (c.c. 3.x)
- pro dosažení plného výkonu paměti musíme přistupovat k datům v rozdílných bankách
- implementován broadcast – pokud všichni přistupují ke stejnému údaji v paměti

Konflikty bank

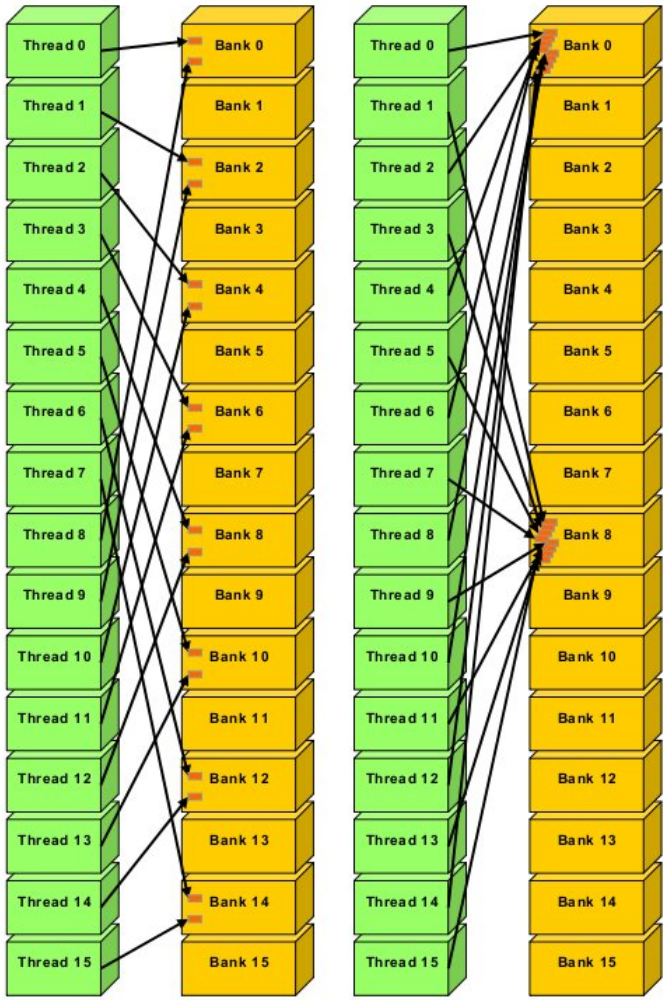
Konflikt bank

- dojde k němu, přistupují-li některé thready v warpu/půlwarpu k datům ve stejné paměťové bance (s vyjímkou, kdy thready přistupují ke stejnému místu v paměti, či přistupují k rozdílným podslovům 64-bitové banky u c.c. 3.0)
- v takovém případě se přístup do paměti serializuje
- spomalení běhu odpovída množství paralelních operací, které musí paměť provést k uspokojení požadavku
  - je rozdíl, přistupuje-li část threadů k různým datům v jedné bance a ke stejným datům v jedné bance

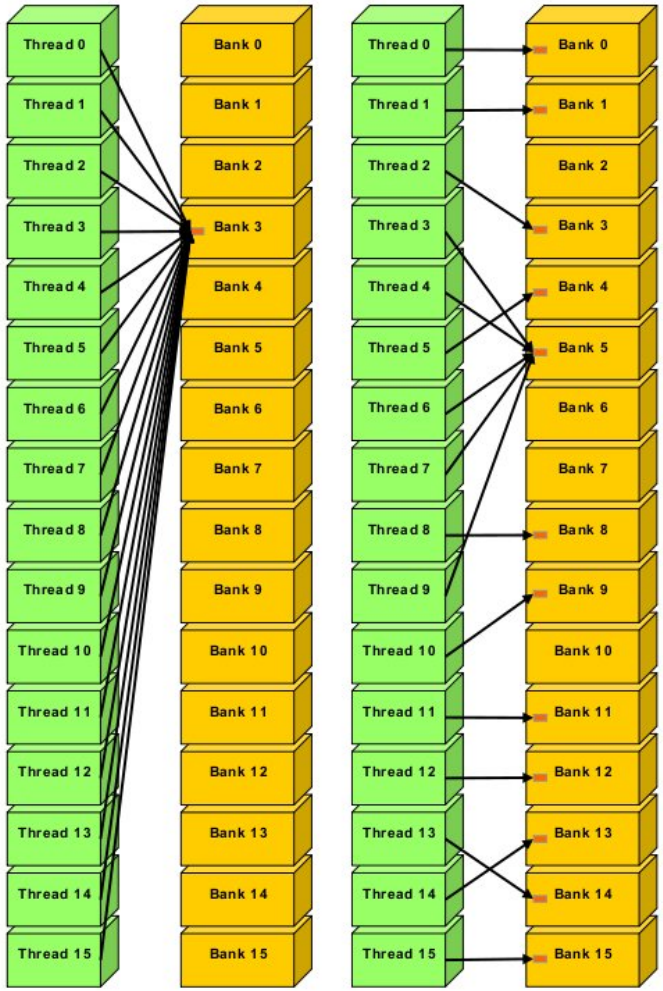
Přístup bez konfliktů



Vícecestné konflikty



Broadcast


Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Thread 12
Thread 13
Thread 14
Thread 15
Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7
Bank 8
Bank 9
Bank 10
Bank 11
Bank 12
Bank 13
Bank 14
Bank 15
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Thread 12
Thread 13
Thread 14
Thread 15
Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7
Bank 8
Bank 9
Bank 10
Bank 11
Bank 12
Bank 13
Bank 14
Bank 15

Vzory přístupu

Zarovnání není třeba, negeneruje konflikty bank

```
int x = s[threadIdx.x + offset];
```

Prokládání negeneruje konflikty, je-li c liché, u 3.x může být $c = 2$ (ne pro 64-bitová čísla)

```
int x = s[threadIdx.x * c];
```

Přístup ke stejné proměnné negeneruje na c.c. 2.x a 3.x konflikty nikdy, na 1,x je-li počet c threadů pristupující k proměnné násobek 16

```
int x = s[threadIdx.x / c];
```

Ostatní paměti

Přenosy mezi systémovou a grafickou pamětí

- je nutné je minimalizovat (často i za cenu neefektivní části výpočtu na GPU)
- mohou být zrychleny pomcí page-locked paměti
- je výhodné přenášet větší kusy současně
- je výhodné překrýt výpočet s přenosem

Texturová paměť

- vhodná k redukci přenosů z globální paměti
- vhodná k zajištění zarovnaného přístupu
- nevhodná, pokud je bottleneck latence
- může zjednodušit adresování či přidat filtraci

Ostatní paměti

Paměť konstant

- rychlá jako registry, pokud čteme tutéž hodnotu
- se čtením různých hodnot lineárně klesá rychlost

Registry

- read-after-write latence, odstíněna pokud na multiprocesoru běží alespoň 192 threadů pro c.c. 1.x a 768 threadů pro c.c. 2.x
- potenciální bank konflikty i v registrech
  - kompilátor se jim snaží zabránit
  - můžeme mu to usnadnit, pokud nastavíme velikost bloku na násobek 64

Transpozice matic

Z teoretického hlediska:

- triviální problém
- triviální paralelizace
- jsme triviálně omezení propustností paměti (neděláme žádné flops)

```
__global__ void mtran(float *odata, float* idata, int n){
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  odata[x*n + y] = idata[y*n + x];
}
```

Výkon

Spustíme-li kód na GeForce GTX 280 s použitím dostatečně velké matice 4000×4000, bude propustnost $5.3\,\mathrm{GB/s}$.
Kde je problém?

Výkon

Spustíme-li kód na GeForce GTX 280 s použitím dostatečně velké matice 4000×4000, bude propustnost 5.3 GB/s.

Kde je problém?
Přístup do odata je prokládaný! Modifikujeme transpozici na kopírování:

```
odata[y*n + x] = idata[y*n + x];
```

a získáme propustnost 112.4 GB/s. Pokud bychom přistupovali s prokládáním i k *idata*, bude výsledná rychlost 2.7 GB/s.

Odstranění prokládání

Matici můžeme zpracovávat po blocích

- načteme po řádcích blok do sdílené paměti
- uložíme do globální paměti jeho transpozici taktéž po řádcích
- díky tomu je jak čtení, tak zápis bez prokládání

Odstranění prokládání

Matici můžeme zpracovávat po blocích

- načteme po řádcích blok do sdílené paměti
- uložíme do globální paměti jeho transpozici taktéž po řádcích
- díky tomu je jak čtení, tak zápis bez prokládání

Jak velké bloky použít?

- budeme uvažovat bloky čtvercové velikosti
- pro zarovnané čtení musí mít řádek bloku velikost dělitelnou 16
- v úvahu připadají bloky 16×16, 32×32 a 48×48 (jsme omezeni velikostí sdílené paměti)
- nejvhodnější velikost určíme experimentálně

Bloková transpozice

```
__global__ void mtran_coalesced(float *odata, float *idata, int n)
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int index_in = x + y*n;
  x = blockIdx.y * TILE_DIM + threadIdx.x;
  y = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = x + y*n;

  for (int i = 0; i < TILE_DIM; i += BLOCK_ROWS)
    tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*n];

  __syncthreads();

  for (int i = 0; i < TILE_DIM; i += BLOCK_ROWS)
    odata[index_out+i*n] = tile[threadIdx.x][threadIdx.y+i];
}
```

Výkon

Nejvyšší výkon byl naměřen při použití bloků velikosti 32×32, velikost thread bloku 32×8, a to 75.1GB/s.

Výkon

Nejvyšší výkon byl naměřen při použití bloků velikosti 32×32, velikost thread bloku 32×8, a to 75.1GB/s.

- to je výrazně lepší výsledek, nicméně stále nedosahujeme rychlosti pouhého kopírování

Výkon

Nejvyšší výkon byl naměřen při použití bloků velikosti 32×32, velikost thread bloku 32×8, a to 75.1GB/s.

- to je výrazně lepší výsledek, nicméně stále nedosahujeme rychlosti pouhého kopírování
- kernel je však složitější, obsahuje synchronizaci
  - je nutno ověřit, jestli jsme narazili na maximum, nebo je ještě někde problém

Výkon

Nejvyšší výkon byl naměřen při použití bloků velikosti 32×32, velikost thread bloku 32×8, a to 75.1GB/s.

- to je výrazně lepší výsledek, nicméně stále nedosahujeme rychlosti pouhého kopírování
- kernel je však složitější, obsahuje synchronizaci
  - je nutno ověřit, jestli jsme narazili na maximum, nebo je ještě někde problém
- pokud v rámci bloků pouze kopírujeme, dosáhneme výkonu 94.9GB/s
  - něco ještě není optimální

Sdílená paměť

Při čtení globální paměti zapisujeme do sdílené paměti po řádcích.

```
tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*n];
```

Sdílená paměť

Při čtení globální paměti zapisujeme do sdílené paměti po řádcích.

```
tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*n];
```

Při zápisu do globální paměti čteme ze sdílené po sloupcích.

```
odata[index_out+i*n] = tile[threadIdx.x][threadIdx.y+i];
```

To je čtení s prokládáním, které je násobkem 16, celý sloupec je tedy v jedné bance, vzniká 16-cestný bank conflict.

Sdílená paměť

Při čtení globální paměti zapisujeme do sdílené paměti po řádcích.

```
tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*n];
```

Při zápisu do globální paměti čteme ze sdílené po sloupcích.

```
odata[index_out+i*n] = tile[threadIdx.x][threadIdx.y+i];
```

To je čtení s prokládáním, které je násobkem 16, celý sloupec je tedy v jedné bance, vzniká 16-cestný bank conflict.
Řešením je padding:

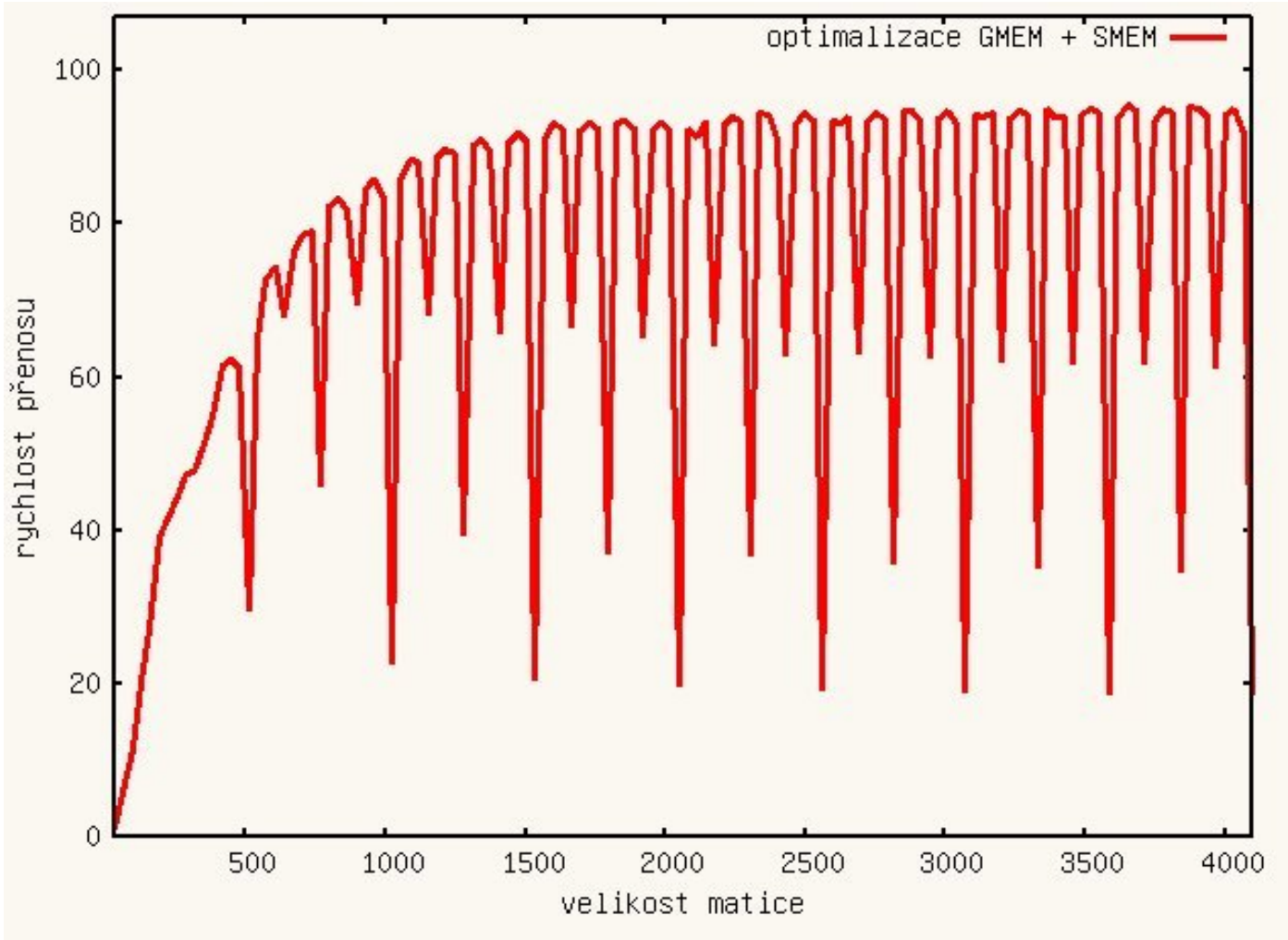
```
__shared__ float tile[TILE_DIM][TILE_DIM + 1];
```

Výkon

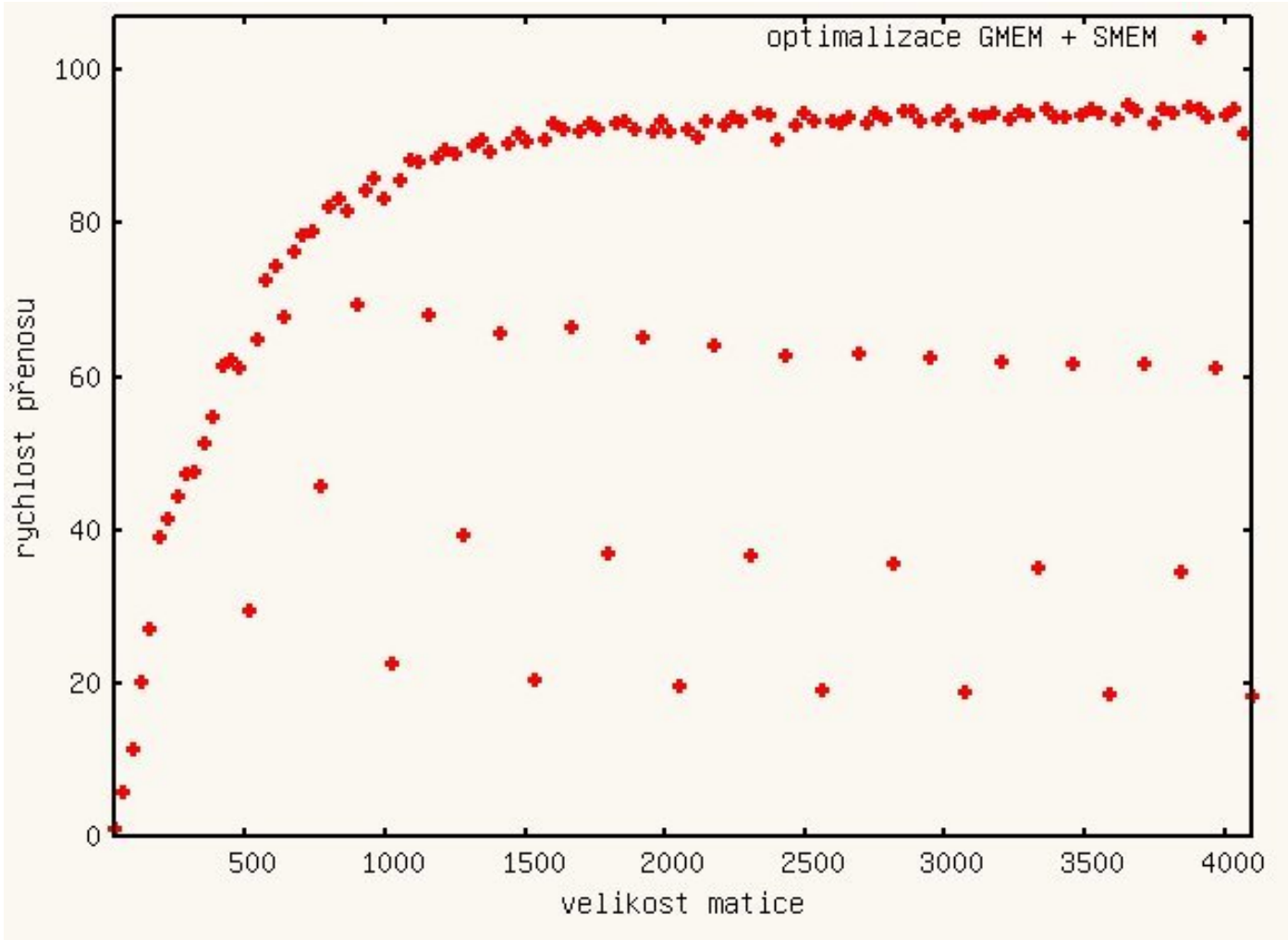
Nyní dosahuje naše implementace výkon 93.4 GB/s.

- obdobný výsledek, jako při pouhém kopírování
- zdá se, že výrazněji lepšího výsledku již pro danou matici nedosáhneme
- pozor na různou velikost vstupních dat (viz. partition camping)

Výkon


optimalizace GMEM + SMEM
rychlost přenosu
velikost matice
0
20
40
60
80
100
500
1000
1500
2000
2500
3000
3500
4000

Výkon


optimalizace GMEM + SMEM
rychlost přenosu
velikost matice
0
20
40
60
80
100
500
1000
1500
2000
2500
3000
3500
4000

Poklesy výkonu

Pro některé velikosti problému výkon klesá, v tomto chování lze nalézt pravidla

Poklesy výkonu

Pro některé velikosti problému výkon klesá, v tomto chování lze nalézt pravidla

- u matic o velikosti dělitelné 512 dosahujeme pouze cca 19 GB/s
- pro zbývající o velikosti dělitelné 256 cca 35 GB/s
- pro zbývající o velikosti dělitelné 128 cca 62 GB/s

Poklesy výkonu

Jeden region paměti má šířku 2 bloků (256 byte / 4 byte na float, 32 floatů v bloku). Podíváme-li se na umístění bloků vzhledem k velikosti matice, zjistíme, že

- při velikosti dělitelné 512 jsou bloky ve sloupcích ve stejném regionu
- při velikosti dělitelé 256 je každý sloupec nejvýše ve dvou regionech
- při velikosti dělitelné 128 je každý sloupec nejvýše ve čtyřech regionech

Dochází tedy k partition campingu!

Jak odstraníme partition camping

Můžeme doplnit ,,slepá data" a vyhnout se tak nevhodným velikostem matic

- to komplikuje práci s algoritmem
- další nevýhodou jsou větší paměťové nároky

Jak odstraníme partition camping

Můžeme doplnit ,,slepá data" a vyhnout se tak nevhodným velikostem matic

- to komplikuje práci s algoritmem
- další nevýhodou jsou větší paměťové nároky

Můžeme změnit mapování id thread bloků na bloky v matici

- diagonální mapování zajistí přístup do rozdílných regionů

```
int blockIdx_y = blockIdx.x;
int blockIdx_x = (blockIdx.x+blockIdx.y) % gridDim.x;
```

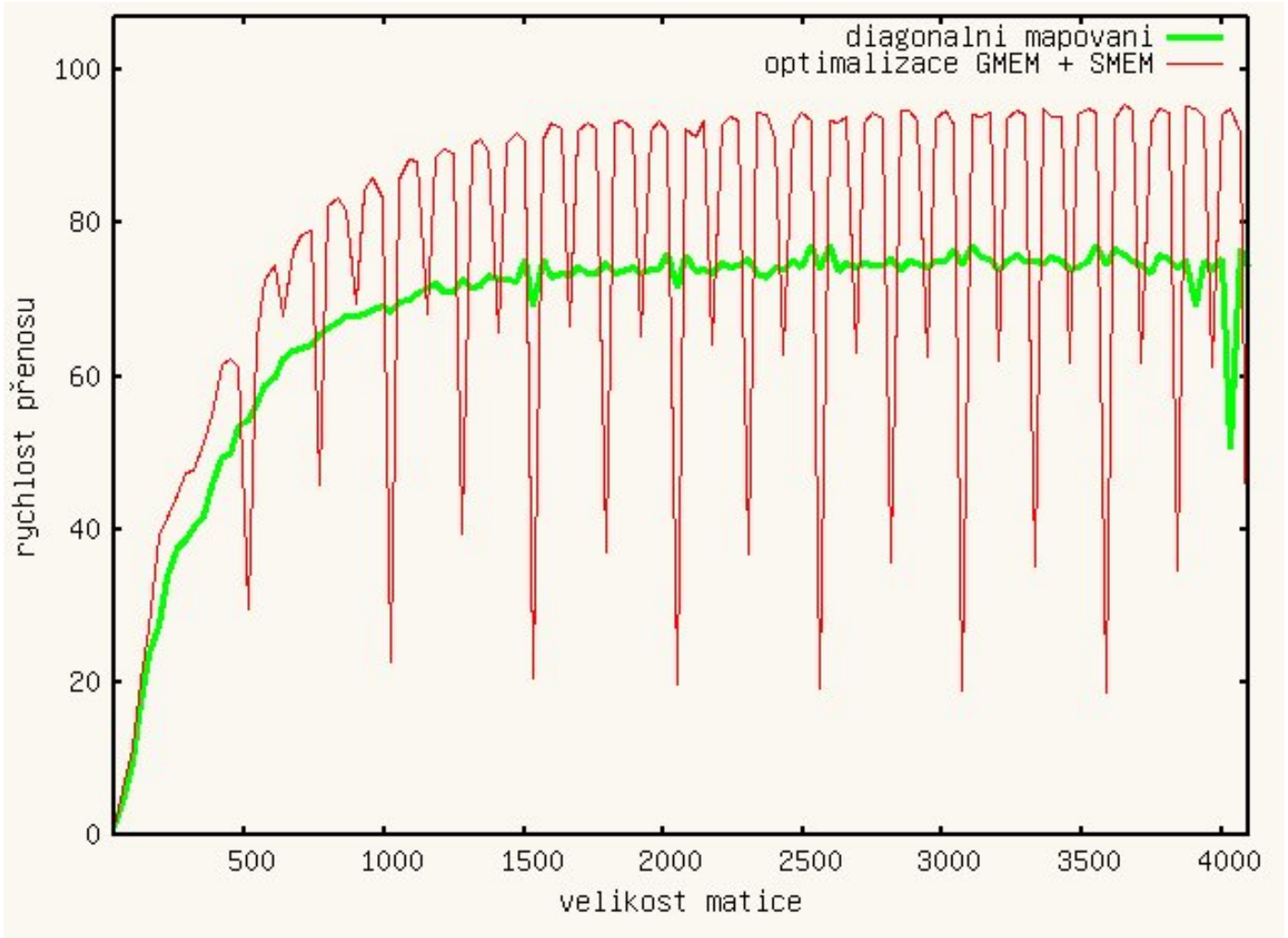
Výkon

Nová implementace podává výkon cca 80 GB/s

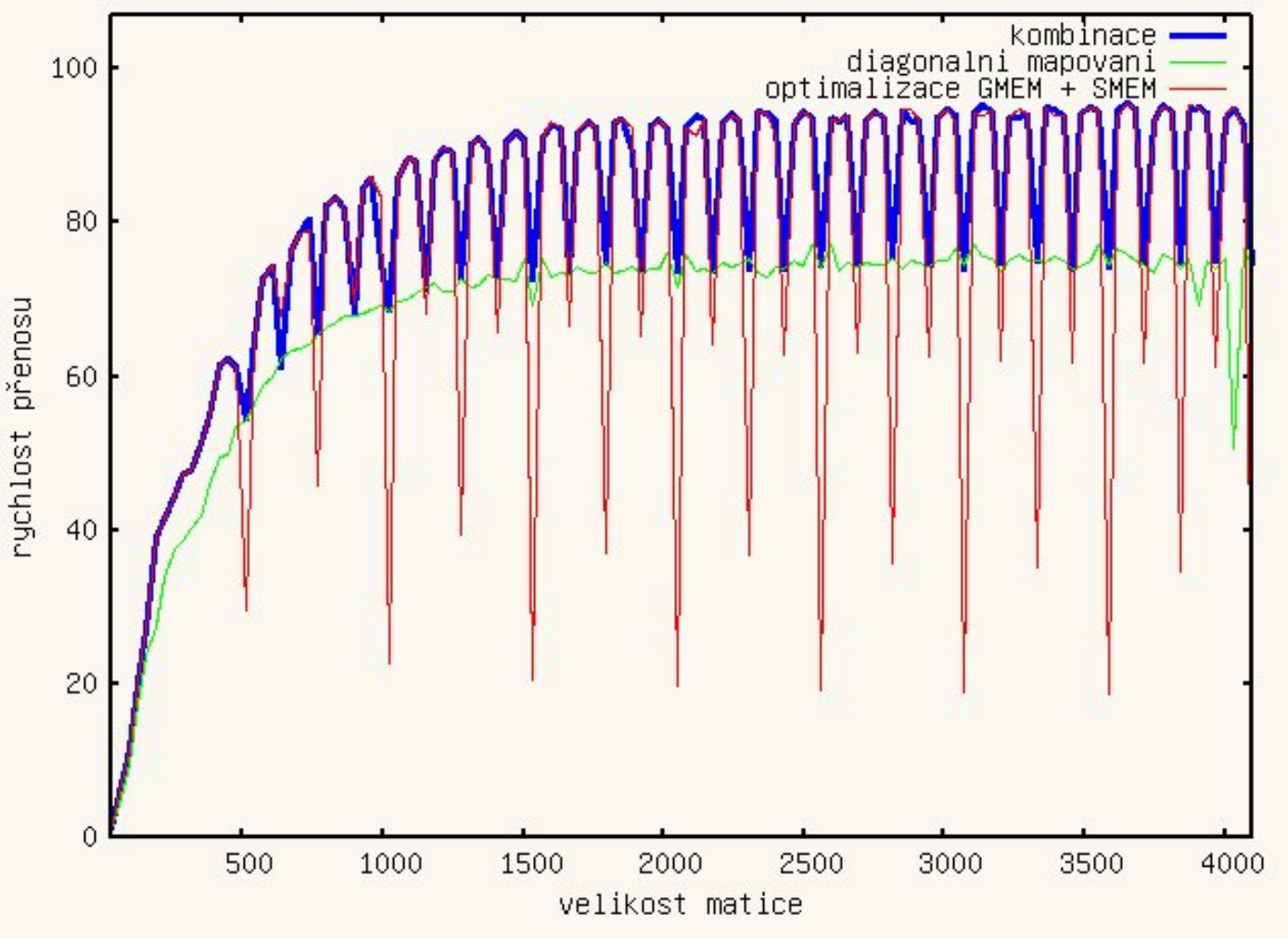
- ten neklesá na datech, kde klesal výkon původní implementace
- pro matice o velikosti nedělitelné 128 je však nižší
  - algoritmus je složitější
- můžeme jej však používat jen u dat, pro které je původní implementace nevýhodná

Pro daný problém nemusí existovat (a spravidla neexistuje) ideální algoritmus pro celý rozsah (či typ) vstupních dat, je vhodné řádně benchmarkovat (ne každý potenciální problém odhalíme pohledem na kód).

Výkon


diagonalni mapovani
optimalizace GMEM + SMEM
rychlost přenosu
velikost matice
0
20
40
60
80
100
500
1000
1500
2000
2500
3000
3500
4000

Výkon


kombinace
diagonalni mapovani
optimalizace GMEM + SMEM
rychlost přenosu
velikost matice
0
20
40
60
80
100
500
1000
1500
2000
2500
3000
3500
4000

Zhodnocení výkonu

Veškeré optimalizace sloužily pouze k lepšímu přizpůsobení-se vlastnostem HW

- přesto jsme dosáhli 17.6× zrychlení
- při formulaci algoritmu je nezbytné věnovat pozornost hardwareovým omezením
- jinak bychom se nemuseli vývojem pro GPU vůbec zabývat...

Význam optimalizací

Pozor na význam optimalizací

- pokud bychom si zvolili testovací matice velikosti 4096×4096 namísto 4000×4000, byl by efekt odstranění konfliktů ve sdílené paměti po odstranění prokládaného přístupu prakticky neznatelný
- po odstranění memory campingu by se však již konflikty bank výkonnostně projevily!
- je dobré postupovat od obecně zásadnějších optimalizací k těm méně zásadním
- nevede-li některá (prokazatelně korektní :-)) optimalizace ke zvýšení výkonu, je třeba prověřit, co algoritmus brzdí

Provádění instrukcí

Provádění instrukcí na multiprocesoru (c.c. 1.x)

- zde je 8 SP jader a 2 SFU jádra
- nedojde-li k překryvu SP a SFU provádění instrukcí, může multiprocesor dokončit až 8 instrukcí na takt
  - jeden warp je tedy proveden za 4 nebo více taktů
- některé instrukce jsou výrazně pomalejší
- znalost doby provádění instrukcí nám pomůže psát efektivní kód

Operace v pohyblivé řádové čárce

GPU je primárně grafický HW

- v grafických operacích pracujeme zpravidla s čísly s plovoucí řádovou čárkou
- GPU je schopno provádět je velmi rychle
- novější GPU (compute capability ≥ 1.3) dokážou pracovat i v double-precision, starší pouze v single-precision
- některé aritmetické funkce jsou používány v grafických výpočtech velmi často
  - GPU je implementuje v hardware
  - HW implementace poskytuje méně přesné výsledky (pro spoustu aplikací není problém)
  - rozlišeno pomocí prefixu „__"

Aritmetické operace

Operace s plovoucí řádovou čárkou (propustnost na MP)

- sčítání, násobení 8 (1.x), 32 (2.0), 48 (2.1), 192 (3.x)
- násobení a sčítání může být u c.c. 1.x kombinováno do jedné instrukce MAD
  - nižší přesnost
  - rychlost 1 cyklus na SP
  - *__fadd_rn()* a *__fmul_rn()* lze použít pokud nechceme, aby byla v překladu použita instrukce MAD
- MAD je nahrazeno FMAD u c.c. 2.x (shodná rychlost, vyšší přesnost)
- 64-bitové verze 1/8 (1.3), 1/2 (2.0), 1/12 (2.1), 1/24 (3.0), 1/3 (3.5)
- převrácená hodnota 2 (1.x), 4 (2.0), 8 (2.1), 32 (3.x)
- dělení relativně pomalé (u c.c. 1.x v průměru cca 1.23)
  - rychlejší varianta pomocí *__fdividef(x, y)* 1.6 (c.c. 1.x)
- reciproká druhá odmocnina 2 (1.x), 4 (2.0) a 8 (2.1), 32 (3.x)

Aritmetické operace

Operace s plovoucí řádovou čárkou

- *__sinf(x)*, *__cosf(x)*, *__expf(x)* 2 (c.c. 1.x), 4 (c.c. 2.0), 8 (c.c. 1.2), 32 (3.x)
- přesnější *sinf(x)*, *cosf(x)*, *expf(x)* řádově pomalejší
- operací s různými rychlostmi a přesností je implementováno více, viz CUDA manuál

Celočíselné operace

- sčítání jako u plovoucí řádové čárky (160 u c.c. 3.0)
- násobení u c.c. 1.x 2 instrukce na MP
  - *__mul24(x, y)* a *__umul24(x, y)* 8 instrukcí
- násobení u c.c. 2.x 16, u c.c. 3.x 32 instrukcí na MP, 24-bitová verze naopak pomalá
- dělení a modulo velmi pomalé, pokud je n mocnina 2, můžeme využít
  - i/n je ekvivalentní $i >> log2(n)$
  - $i\%n$ je ekvivalentní $i\&(n-1)$

Smyčky

Malé cykly mají značný overhead

- je třeba provádět skoky
- je třeba vyhodnocovat podmínky
- je třeba updatovat kontrolní proměnnou
- podstatnou část instrukcí může tvořit pointerová aritmetika

To lze řešit rozvinutím (*unrolling*)

- částečně je schopen dělat kompilátor
- můžeme mu pomoci ručním unrollingem, nebo pomocí direktivy *#pragma unroll*

Ostatní operace

Další běžné instrukce jsou prováděny základní rychlostí (tj. odpovídající počtu SP)

- porovnávání
- základní bitové operace (ne posuvy)
- instrukce přistupující do paměti (s omezeními popsanými výše a s omezením latence a šířky pásma)
  - jako offset mohou použít hodnotu v registu + konstantu
- synchronizace (pokud ovšem nečekáme :-))

Pozor na sdílenou paměť

Pokud nedojde ke konfliktům bank, je sdílená paměť rychlá téměř jako registry.
Ale pozor

- instrukce dokáží pracovat pouze s jedním operandem ve sdílené paměti
- použijeme-li v rámci jedné instrukce více operandů ve sdílené paměti, je třeba explicitní load/store
- instrukce MAD běží pomaleji (c.c. 1.x)
  - $a + s[i]$ 4 cykly na warp
  - $a + a * s[i]$ 5 cyklů na warp
  - $a + b * s[i]$ 6 cyklů na warp
- tyto detaily již nejsou nVidií publikovány (zjištěno měřením)
- může se výrazně měnit s dalšími generacemi GPU, užitečné pro opravdu výkonově kritický kód

Překlad C for CUDA

Device kódy lze přeložit do PTX assembleru a binárních souborů

- PTX je mezikód, neodpovídá přímo instrukcím prováděným na GPU
  - snáze se čte
  - hůře se zjišťuje, co se přesně na GPU děje

Binární soubory lze deassemblovat pomocí nástroje *cuobjdump*

- pro GT200 a novější
- pro starší procesory *decuda* (produkt třetí strany, nemusí fungovat zcela správně)