Výkon GPU hardware

Jiří Filipovič

podzim 2009

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)
- latence 400-600 cyklů

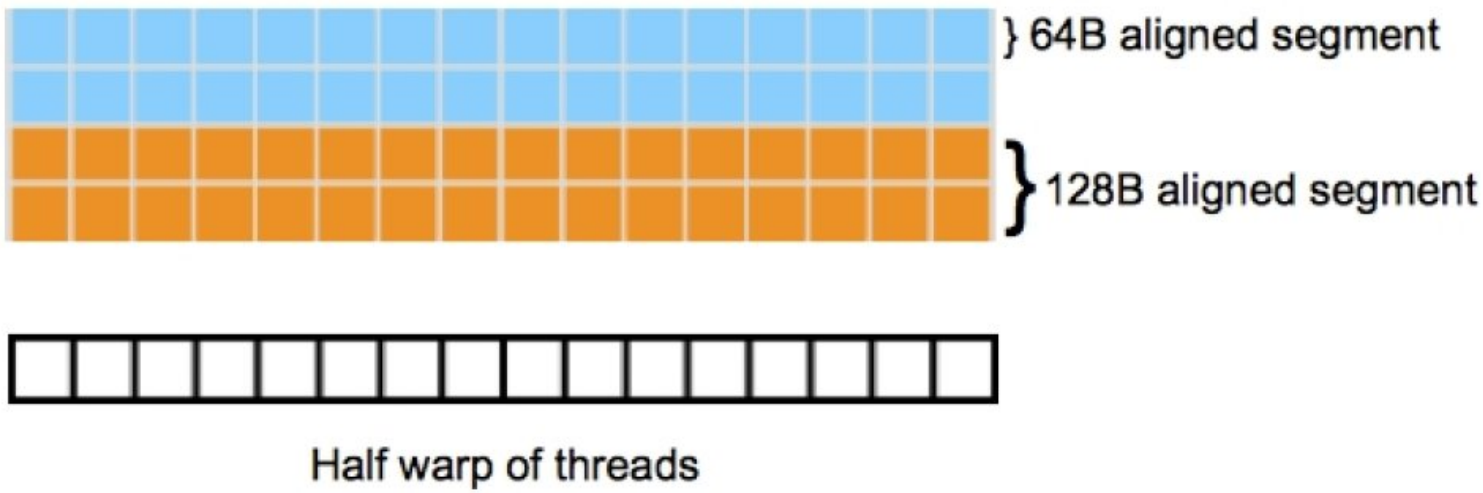
Při špatném vzoru paralelního přístupu do globální paměti snadno výrazně snížíme propustnost

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

Spojitý přístup do paměti

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

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

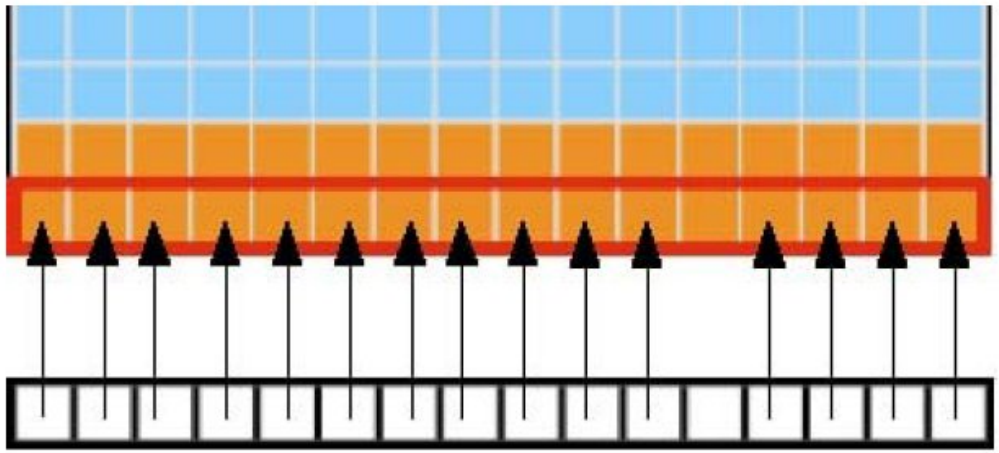
Spojitý přístup do paměti

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

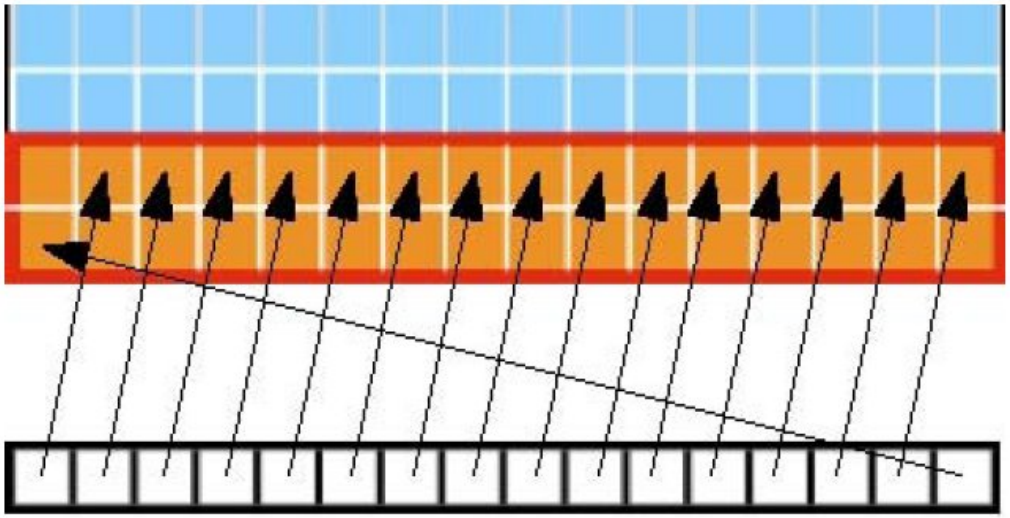
Spojitý přístup do paměti

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



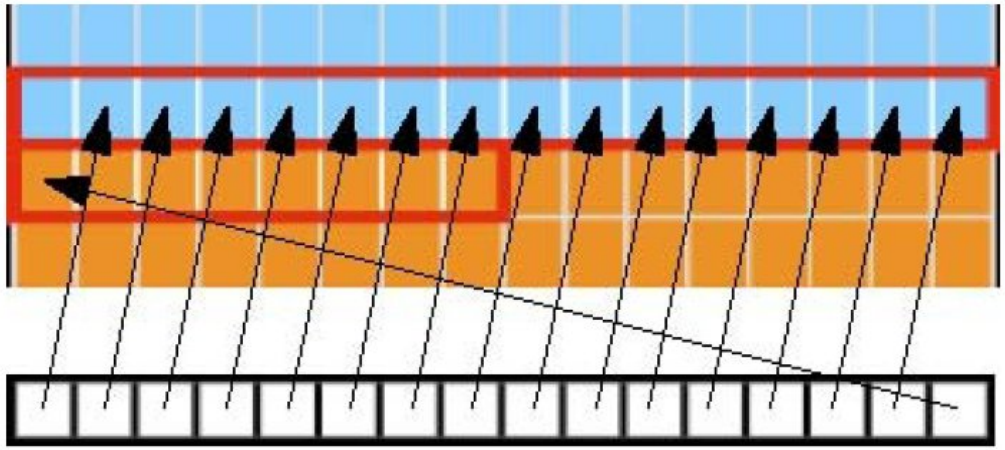
Nezarovnaný přístup do paměti

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



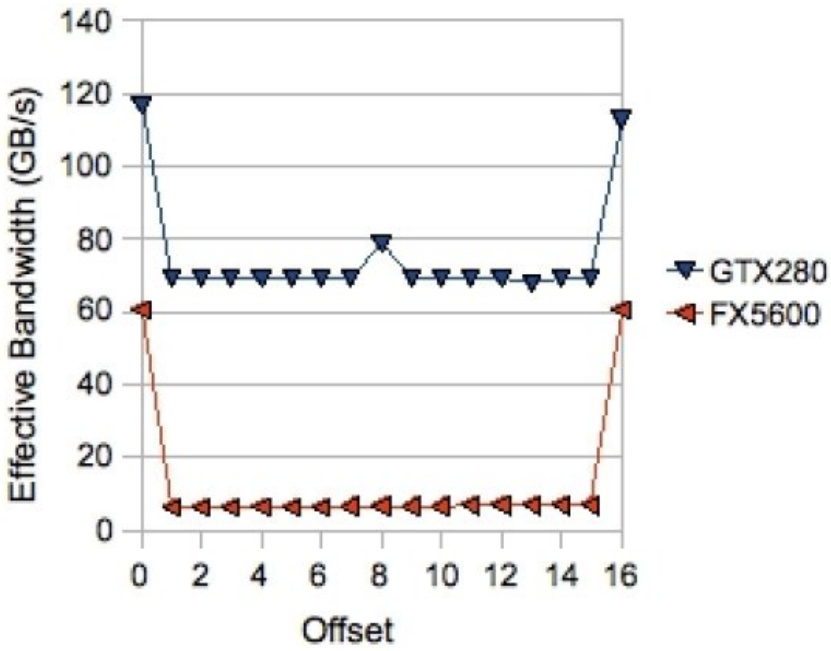
Nezarovnaný přístup do paměti

Obdobný případ může vézt k nutnosti použít dvě transakce.



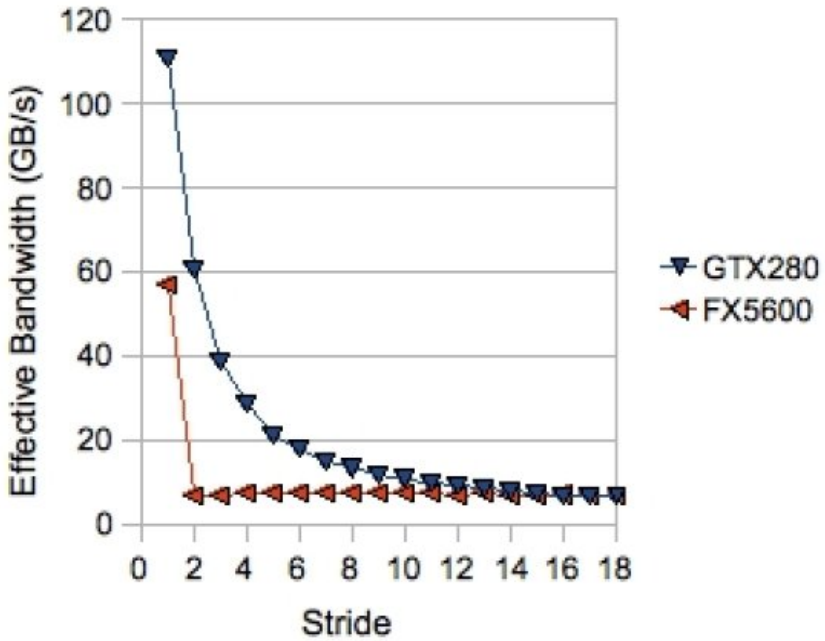
Výkon při nezarovnaném přístupu

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

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



Partition camping

- 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, špatně viditelné při jemnozrnném pohledu

HW organizace sdílené paměti

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

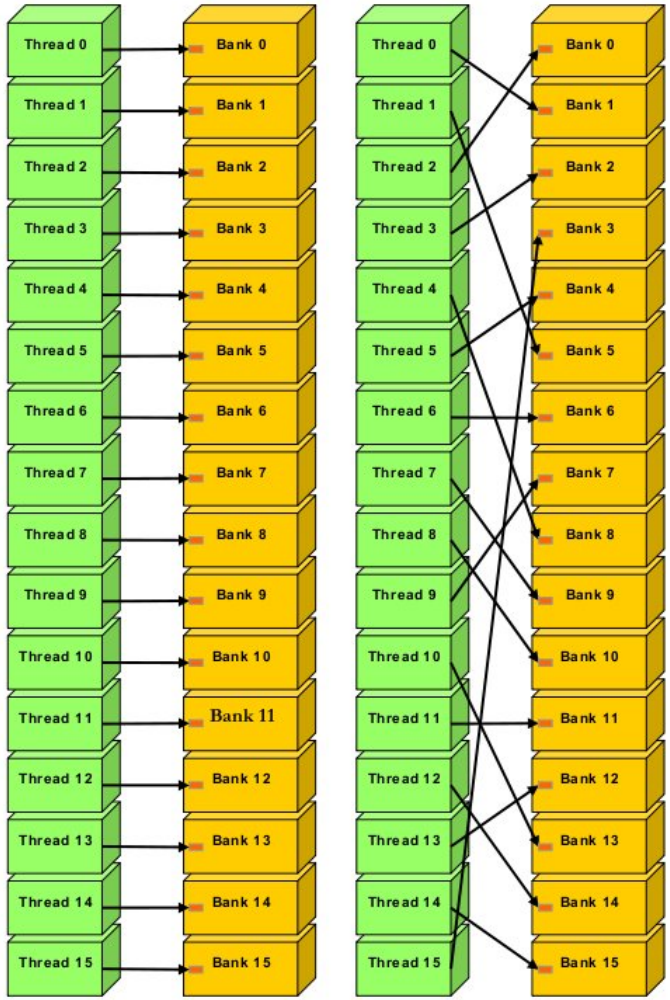
- 16 bank, paměťový prostor mapován prokládaně s odstupem 32 bitů
- pro dosažení plného výkonu paměti musíme přistupovat k datům v rozdílných bankách
- implementován broadcast – pokud všichni přistupují ke stejnému údaji v paměti

Konflikty bank

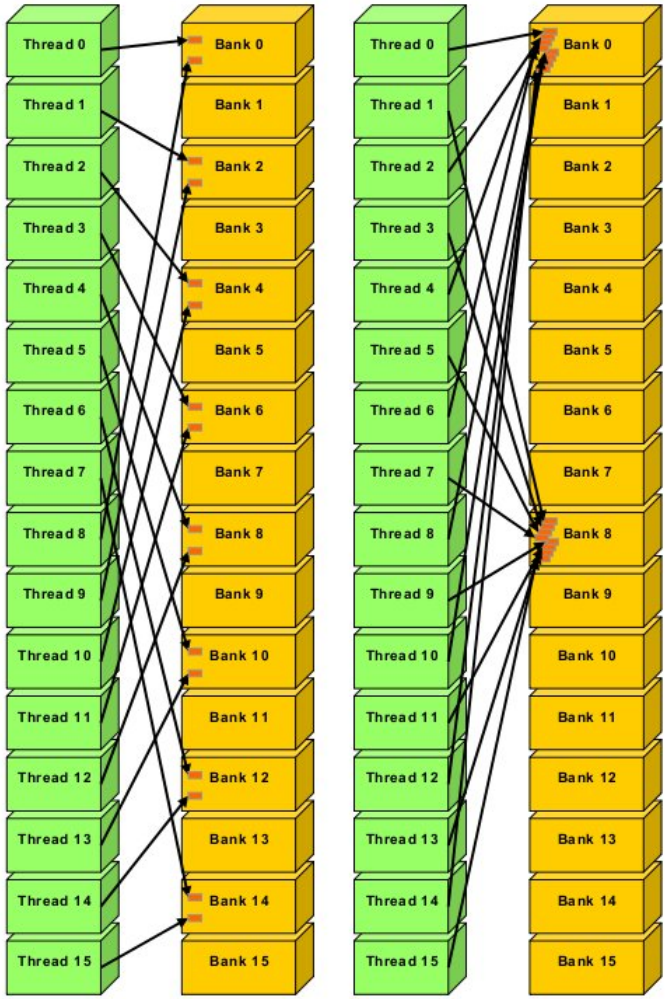
Konflikt bank

- dojde k němu, přistupují-li některé thready v polovině warpu k datům ve stejné paměťové bance (s vyjímkou, kdy všechny thready přistupují ke stejnému místu v paměti)
- v takovém případě se přístup do paměti serializuje
- spomalení běhu odpovída množství paralelních operací, které musí paměť provézt k uspokojení požadavku
  - je rozdíl, přistupuje-li část threadů k různým datům v jedné bance a ke stejným datům v jedné bance

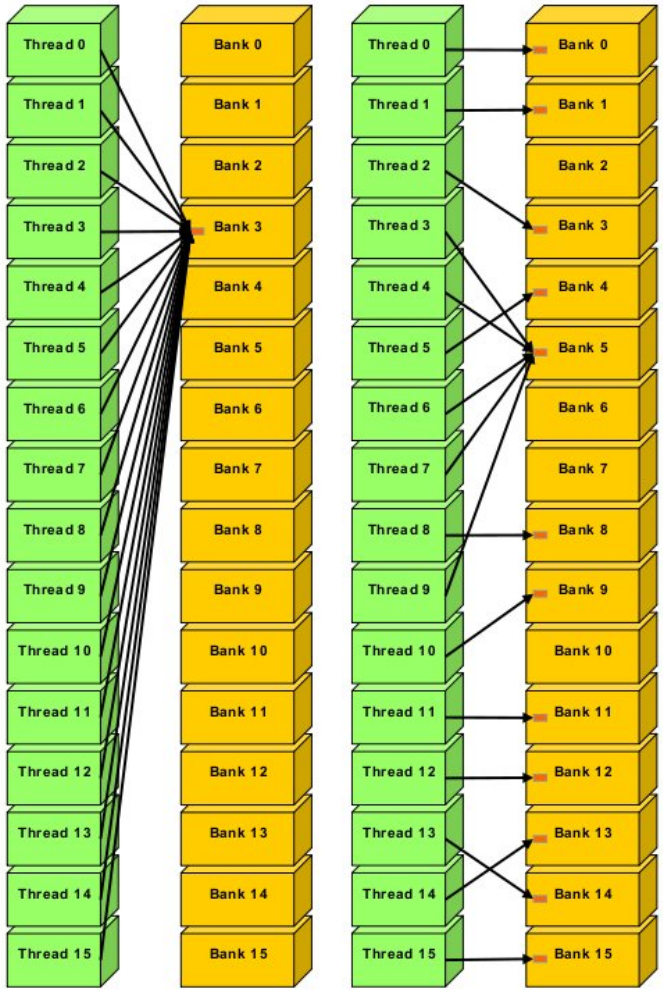
Přístup bez konfliktů



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 bank conflicty

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

Prokládání negeneruje konflikty, je-li c liché

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

Přístup ke stejné proměnné negeneruje konflikty, je-li počet c threadů k proměnné pristupující 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
- 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:

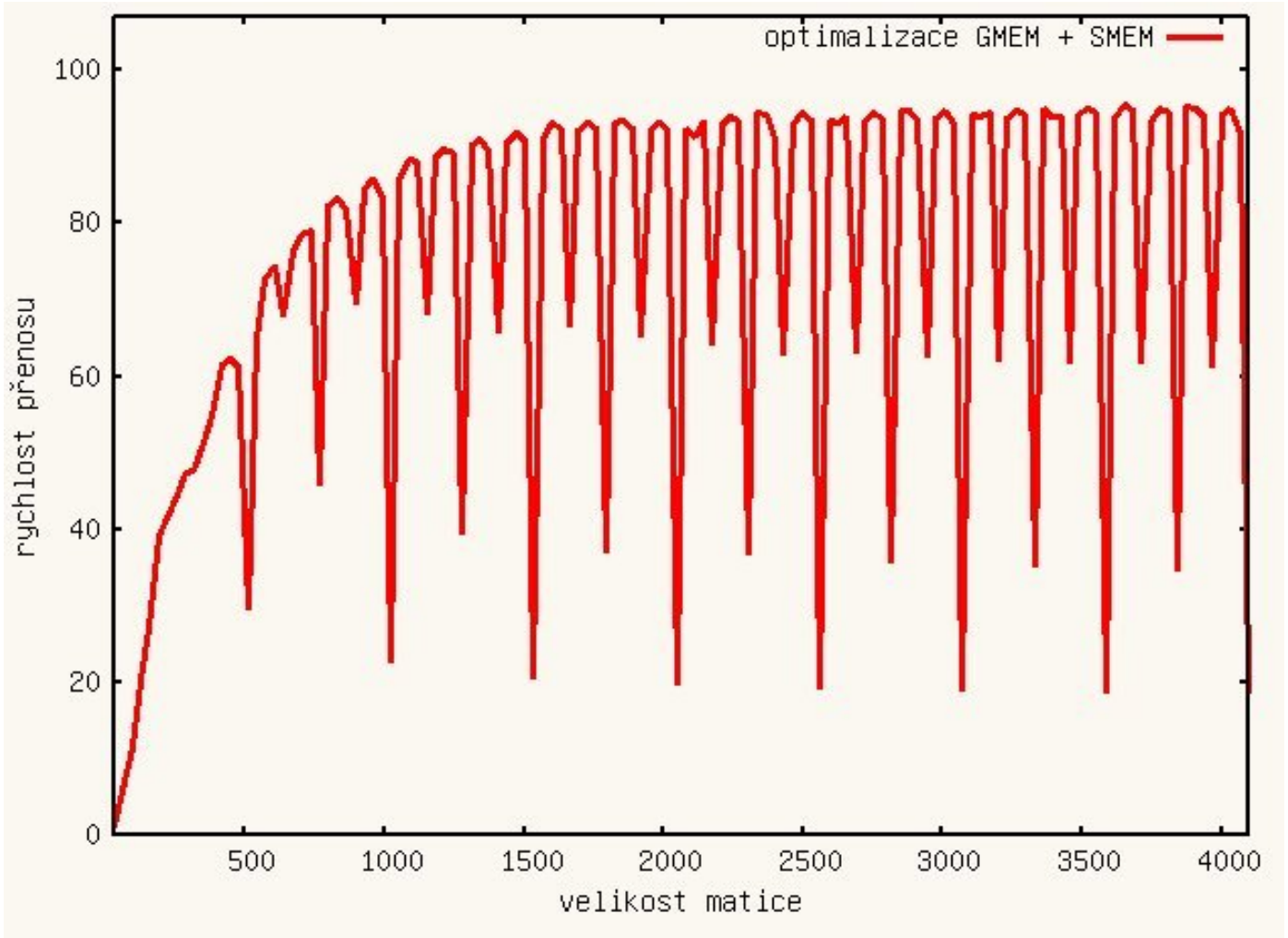
```
__shared__ float tile[TILE_DIM][TILE_DIM + 1];
```

Výkon

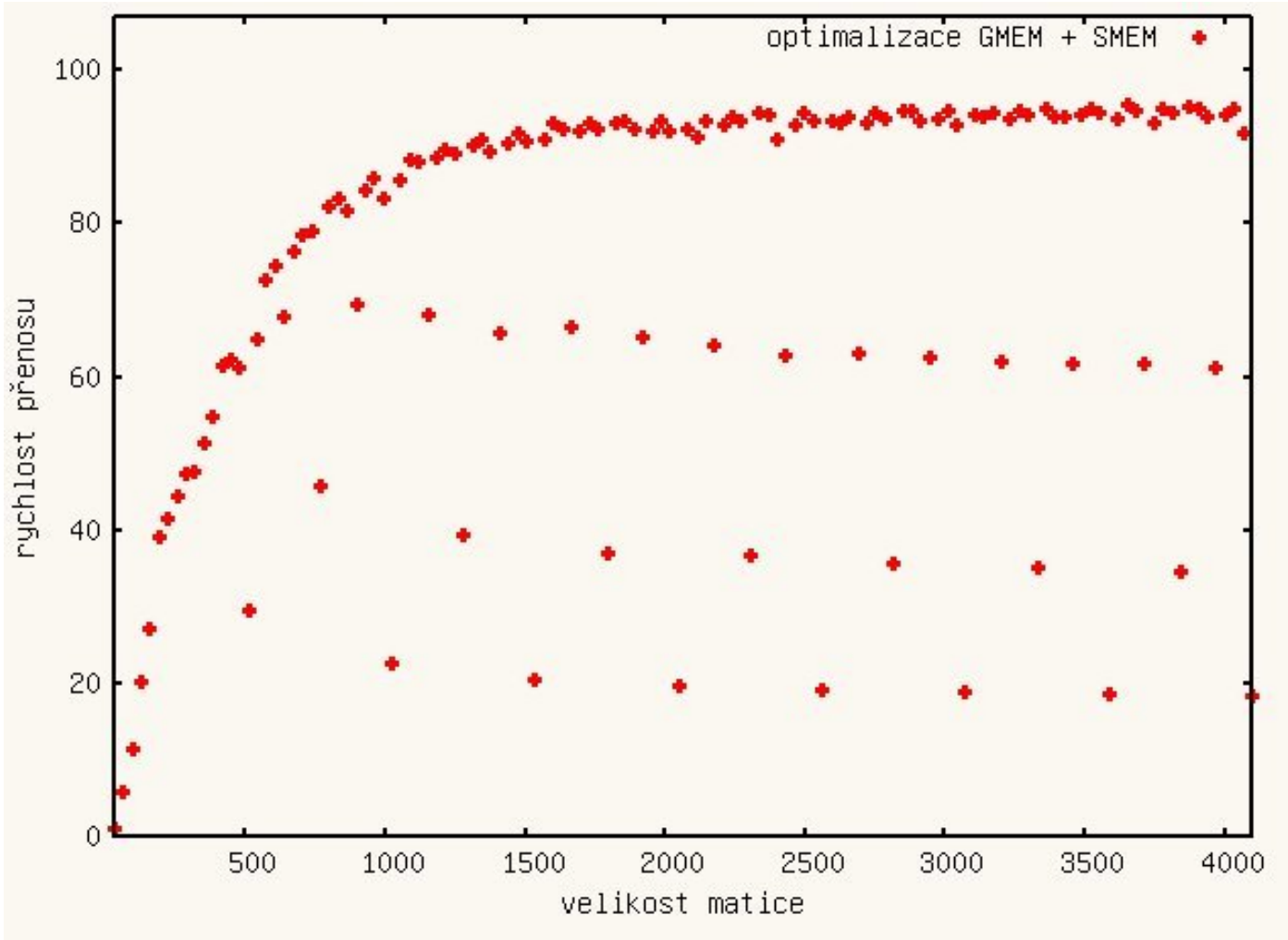
Nyní dosahuje naše implementace výkon 93.4 GB/s.

- 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 regularitu

Poklesy výkonu

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

- 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 (co když matici získáváme z host paměti?)
- další (méně nepříjemnou) 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 (co když matici získáváme z host paměti?)
- další (méně nepříjemnou) 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;
```

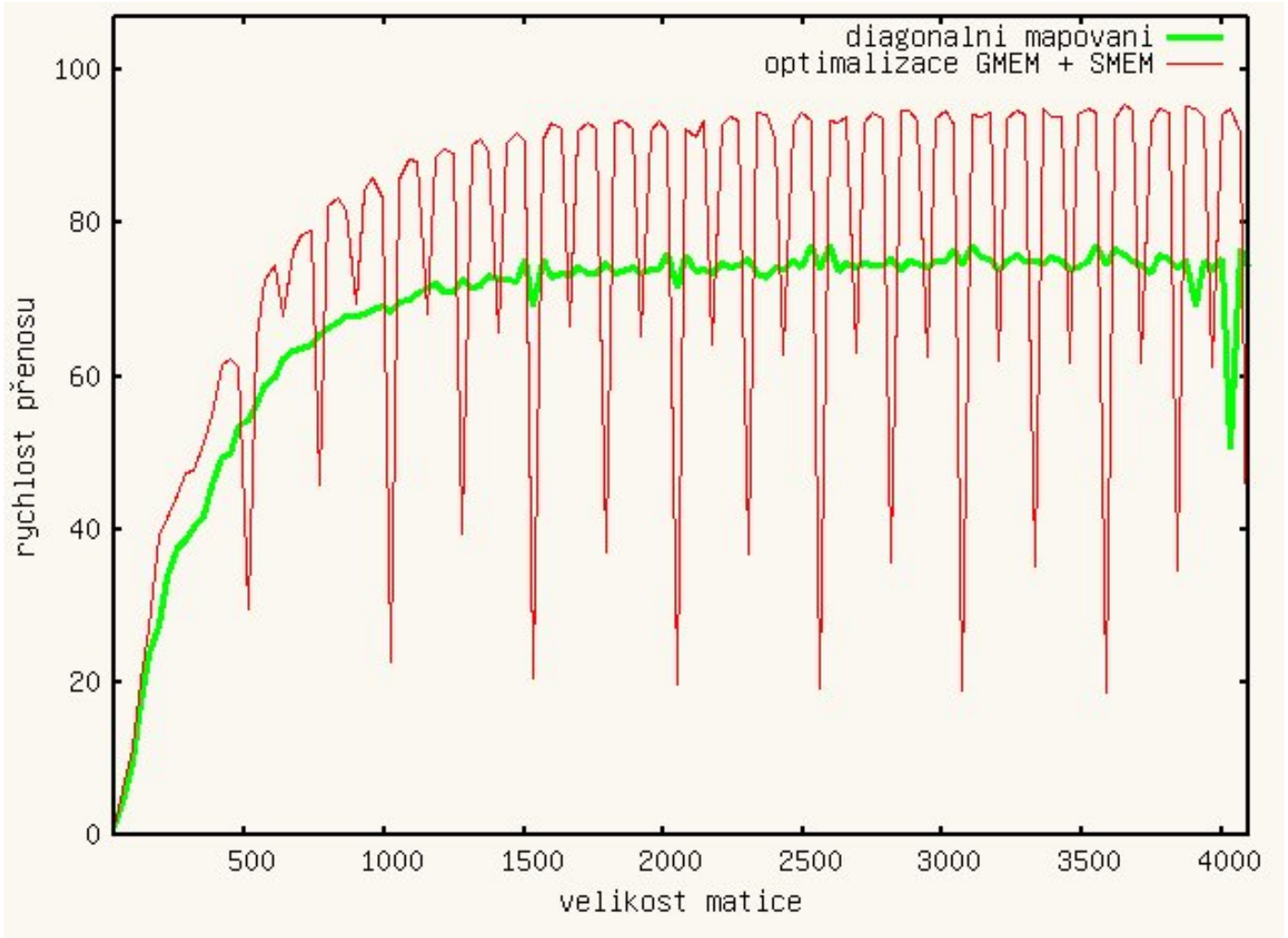
Výkon

Nová implementace podává výkon cca 80 GB/s

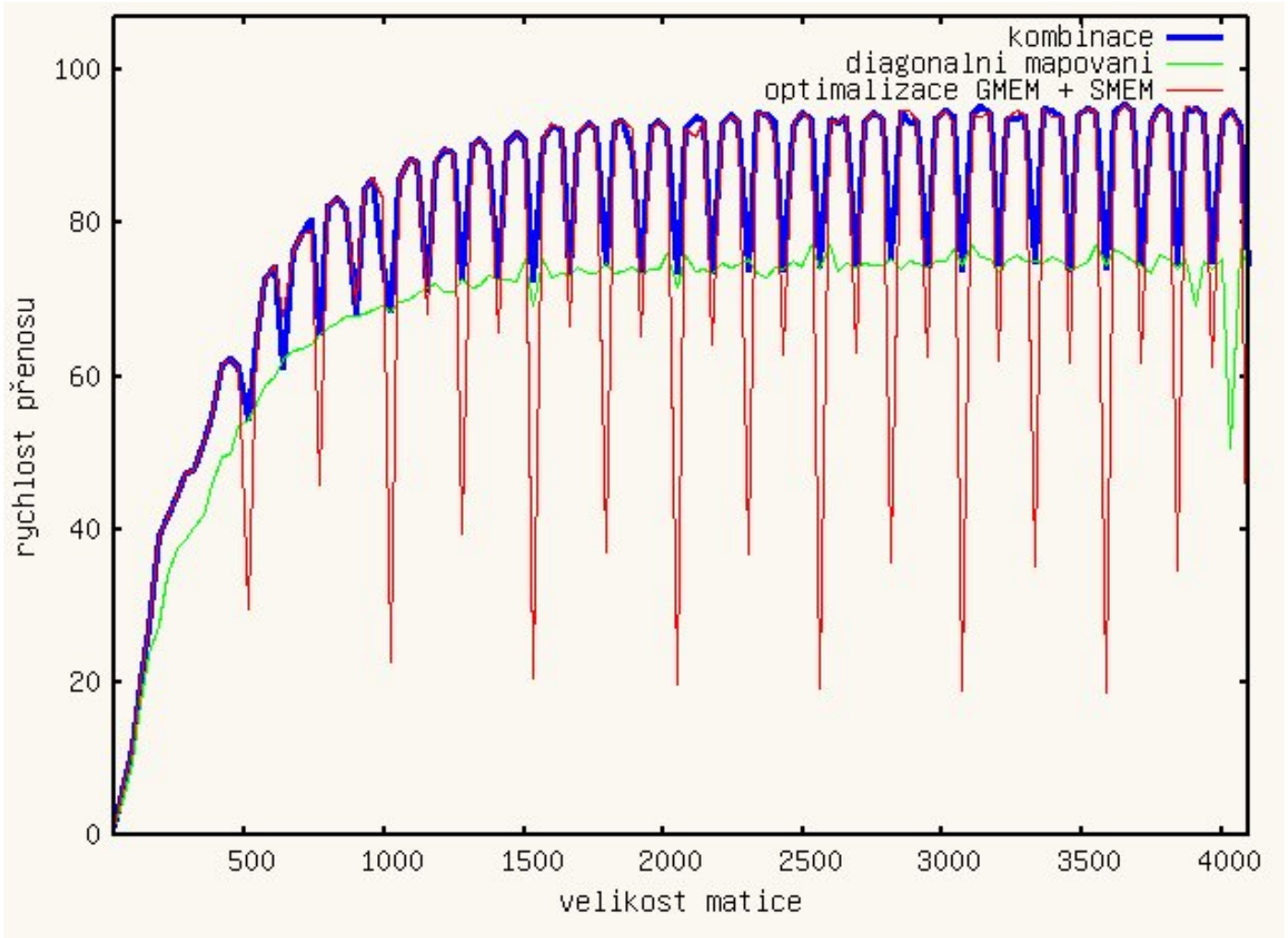
- ten neklesá na datech, kde klesal výkon původní implementace
- pro matice o velikosti nedělitelné 128 je však nižší
  - algoritmus je složitější
- můžeme jej však používat jen u dat, pro které je původní implementace nevýhodná

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 loužily 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, stačilo by napsat dobrý sekvenční algoritmus...

Význam optimalizací

Pozor na význam optimalizací

- pokud bychom si zvolili jako 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

- 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 optimální 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

- sčítání, násobení 4 cykly na warp
- násobení a sčítání může být kombinováno do jedné instrukce MAD
  - nižší přesnost
  - rychlost 4 cykly na warp
  - *__fadd_rn()* a *__fmul_rn()* lze použít pokud nechceme, aby byla v překladu použita instrukce MAD
- převrácená hodnota 16 cyklů na warp
- dělení cca. 26 cyklů na warp
  - rychlejší varianta pomocí *__fdividef(x, y)* 20 cyklů na warp
- reciproká druhá odmocnina 8 cyklů na warp
- druhá odmocnina 16 cyklů na warp

Aritmetické operace

Operace s plovoucí řádovou čárkou

- *__sinf(x)*, *__cosf(x)*, *__expf(x)* 16 cyklů na warp
- přesnější *sinf(x)*, *cosf(x)*, *expf(x)* řádově pomalejší
- operací s různými rychlostmi a přesností je implementováno více, viz CUDA manuál

Celočíselné operace

- sčítání 4 cykly na warp
- násobení 16 cyklů na warp
  - *__mul24(x, y)* a *__umul24(x, y)* 4 cykly na warp
- 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 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 rychlostí 4 instrukce na warp

- porovnávání
- bitové operace
- konverze operandů
- 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 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
  - $a + s[i]$ 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
- nVidia zatím neuvolnila překladač do nativního GPU kódu

Binární soubory lze deassemblovat pomocí nástroje *decuda*

- produkt třetí strany
- nemusí fungovat zcela správně
- přesto dosti užitečný

Naivní implementace

```
__global__ void mmul(float *A, float *B, float *C, int n){
  int x = blockIdx.x*blockDim.x + threadIdx.x;
  int y = blockIdx.y*blockDim.y + threadIdx.y;

  float tmp = 0;
  for (int k = 0; k < n; k++)
    tmp += A[y*n+k] * B[k*n+x];

  C[y*n + x] = tmp;
}
```

Co jsme se naučili

Naivní implementace algoritmu

- každý thread zpracovává odděleně jeden element výsledné matice
- omezena propustností paměti
- teoretické maximum jsme určili jako 66.8 GFlops
- výkon velmi závislý na uspořádání threadů – bloky 128×1 dávají výkon 36.6 GFlops, bloky 1×128 3.9 GFlops

Co jsme se naučili

Naivní implementace algoritmu

- každý thread zpracovává odděleně jeden element výsledné matice
- omezena propustností paměti
- teoretické maximum jsme určili jako 66.8 GFlops
- výkon velmi závislý na uspořádání threadů – bloky 128×1 dávají výkon 36.6 GFlops, bloky 1×128 3.9 GFlops

Nyní rozumíme rozdílným výsledkům

- teoretického maxima nelze docílit – z paměti GPU přenášíme po nejméně 32-bytových částech, musíme tedy přenést více dat, než je nutné
- je-li 128 threadů v bloku zarovnáno ve smyslu osy x, je přenos dat neprokládaný, v opačném případě je prokládaný

Co jsme se naučili

Navrhli jsme blokovou implementaci

- každý blok threadů načítá bloky matic A a B do sdílené paměti, znovuužívá data ke snížení omezení přenosovou rychlostí globální paměti
- teoretické maximum 568 GFlops, dosáhli jsme 198 GFlops

S novými znalostmi můžeme jeho implementaci přehodnotit...

Násobení po blocích

```
__global__ void mmul(float *A, float *B, float *C, int n){
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  __shared__ float As[BLOCK][BLOCK];
  __shared__ float Bs[BLOCK][BLOCK];

  float Csub = 0.0f;
  for (int b = 0; b < n/BLOCK; b++){
    As[ty][tx] = A[(ty + by*BLOCK)*n + b*BLOCK+tx];
    Bs[ty][tx] = B[(ty + b*BLOCK)*n + bx*BLOCK+tx];
    __syncthreads();

    for (int k = 0; k < BLOCK; k++)
      Csub += As[ty][k]*Bs[k][tx];
    __syncthreads();
  }

  C[(ty + by*BLOCK)*n + bx*BLOCK+tx] = Csub;
}
```

Hříchy implementace

```
As[ty][tx] = A[(ty + by*BLOCK)*n + b*BLOCK+tx];
Bs[ty][tx] = B[(ty + b*BLOCK)*n + bx*BLOCK+tx];
...
C[(ty + by*BLOCK)*n + bx*BLOCK+tx] = Csub;
```

Přístup do globální paměťi se zdá být v pořádku.

```
Csub += As[ty][k]*Bs[k][tx];
```

Přístup do sdílené také

- má-li blok threadů velikost ve smyslu osy x násobek velikosti warpu, dochází u proměnné *As* k broadcastu
- proměnná *Bs* je čtena v souvislých řádcích, přístup tedy negeneruje konflikty bank

Teoretické maximum

Lze určit přesněji teoretické omezení výkonu?

- maximum jsme určili podle výkonu GPU v MAD instrukcích (622 GFlops)
- nyní víme, že MAD instrukce pracující s operandem ve sdílené paměti pracují rychlostí 6 taktů na warp
- nově lze tedy teoretické maximum určit jako 415 GFlops
- stále jsme od něj však daleko

Ztráty výkonu

Co nás vzdaluje od maxima?

- overhead spuštění kernelu a spouštění threadů
  - z principu se mu nevyhneme, počet threadů lze redukovat
- operace ,,režije"
  - pointerová aritmetika, cykly
  - lze redukovat
- synchronizace
  - může a nemusí být problém
- load/store ve výpočtu
  - pracujeme se dvěma operandy na jednu MAD instrukci
  - je tedy zapotřebí jeden load na jednu MAD

Počítáme-li výkonový strop pro kombinaci load + MAD s operandem ve sdílené paměti, dostaneme se k omezení 244 GFlops.

- od toho již nejsou naměřené výsledky příliš vzdáleny

Nalezení lepší implementace

Lze počet load instrukcí omezit?

Nalezení lepší implementace

Lze počet load instrukcí omezit?

- data ve sdílené paměti snižují přenosy z paměti globální

Nalezení lepší implementace

Lze počet load instrukcí omezit?

- data ve sdílené paměti snižují přenosy z paměti globální
- můžeme snížit přenosy ze sdílené paměti pomocí dat v registrech?

Nalezení lepší implementace

Lze počet load instrukcí omezit?

- data ve sdílené paměti snižují přenosy z paměti globální
- můžeme snížit přenosy ze sdílené paměti pomocí dat v registrech?
- můžeme – stačí nechat pracovat méně threadů nad více daty

Nalezení lepší implementace

Lze počet load instrukcí omezit?

- data ve sdílené paměti snižují přenosy z paměti globální
- můžeme snížit přenosy ze sdílené paměti pomocí dat v registrech?
- můžeme – stačí nechat pracovat méně threadů nad více daty

Blok o velikosti $m \times n$ threadů necháme pracovat s daty o velikosti $m \times m$, kde $m = n \cdot k; k \in N$.

- větší bloky potenciálně nevýhodné kvůli synchronizaci
- menší bloky potenciálně nevýhodné kvůli overheadu daném pointerovou aritmetikou
- experimentálně najdeme vhodnou velikost bloku

Nalezení lepší implementace

Nejlepší výsledky dosaženy pro bloky velikosti 32×32, na kterých pracuje 32×16 threadů.

- půl loadu na jednu MAD instrukci dává teoretické omezení 311 GFlops
- naměřili jsme 235.4 GFlops
- něco je ještě špatně

Deassembling kódu

Zaměříme se na vnitřní smyčku

```
Csub1 += As[ty][k]*Bs[k][tx];
Csub2 += As[ty+16][k]*Bs[k][tx];
```

```
...
mov.b32 $r0, s[$ofs4+0x0000]
add.b32 $ofs4, $ofs2, 0x00000180
mad.rn.f32 $r7, s[$ofs1+0x0008], $r0, $r7
mad.rn.f32 $r8, s[$ofs3+0x0008], $r0, $r8
...
```

Deassembling kódu

Zaměříme se na vnitřní smyčku

```
Csub1 += As[ty][k]*Bs[k][tx];
Csub2 += As[ty+16][k]*Bs[k][tx];
```

```
...
mov.b32 $r0, s[$ofs4+0x0000]
add.b32 $ofs4, $ofs2, 0x00000180
mad.rn.f32 $r7, s[$ofs1+0x0008], $r0, $r7
mad.rn.f32 $r8, s[$ofs3+0x0008], $r0, $r8
...
```

Kompilátor dokázal převést adresaci přes *k* na konstantní offsety pouze u proměnné *As*

- k *Bs* je přistupováno prokládaně
- znamená to jednu add instrukci navíc

Odstranění add instrukce

Do pole *Bs* můžeme ukládat transponovaná data, pak vypadá kód vnitřní smyčky takto

```
Csub1 += As[ty][k]*Bs[tx][k];
Csub2 += As[ty+16][k]*Bs[tx][k];
```

Odstranění add instrukce

Do pole *Bs* můžeme ukládat transponovaná data, pak vypadá kód vnitřní smyčky takto

```
Csub1 += As[ty][k]*Bs[tx][k];
Csub2 += As[ty+16][k]*Bs[tx][k];
```

Ve výsledném assembleru již instrukce add chybí

```
...
mov.b32 $r0, s[$ofs4+0x0008]
mad.rn.f32 $r6, s[$ofs3+0x0034], $r0, $r6
mad.rn.f32 $r8, s[$ofs1+0x0008], $r0, $r8
...
```

Odstranění add instrukce

Do pole *Bs* můžeme ukládat transponovaná data, pak vypadá kód vnitřní smyčky takto

```
Csub1 += As[ty][k]*Bs[tx][k];
Csub2 += As[ty+16][k]*Bs[tx][k];
```

Ve výsledném assembleru již instrukce add chybí

```
...
mov.b32 $r0, s[$ofs4+0x0008]
mad.rn.f32 $r6, s[$ofs3+0x0034], $r0, $r6
mad.rn.f32 $r8, s[$ofs1+0x0008], $r0, $r8
...
```

Nový problém – konflikty bank sdílené paměti

Odstranění add instrukce

Do pole *Bs* můžeme ukládat transponovaná data, pak vypadá kód vnitřní smyčky takto

```
Csub1 += As[ty][k]*Bs[tx][k];
Csub2 += As[ty+16][k]*Bs[tx][k];
```

Ve výsledném assembleru již instrukce add chybí

```
...
mov.b32 $r0, s[$ofs4+0x0008]
mad.rn.f32 $r6, s[$ofs3+0x0034], $r0, $r6
mad.rn.f32 $r8, s[$ofs1+0x0008], $r0, $r8
...
```

Nový problém – konflikty bank sdílené paměti

- vyřeší padding

Odstranění add instrukce

Do pole *Bs* můžeme ukládat transponovaná data, pak vypadá kód vnitřní smyčky takto

```
Csub1 += As[ty][k]*Bs[tx][k];
Csub2 += As[ty+16][k]*Bs[tx][k];
```

Ve výsledném assembleru již instrukce add chybí

```
...
mov.b32 $r0, s[$ofs4+0x0008]
mad.rn.f32 $r6, s[$ofs3+0x0034], $r0, $r6
mad.rn.f32 $r8, s[$ofs1+0x0008], $r0, $r8
...
```

Nový problém – konflikty bank sdílené paměti

- vyřeší padding

Výsledná rychlost: 276.2 GFlops.

Lze matice násobit ještě rychleji?

Naměřený výkon je již poměrně blízký teoretickému maximu

- rozdíl je dán spouštěním kernelu/threadů, synchronizací a pointerovou aritmetikou
- chceme-li dosáhnout vyšší rychlosti, je třeba přehodnotit algoritmus

Lze matice násobit ještě rychleji?

Naměřený výkon je již poměrně blízký teoretickému maximu

- rozdíl je dán spouštěním kernelu/threadů, synchronizací a pointerovou aritmetikou
- chceme-li dosáhnout vyšší rychlosti, je třeba přehodnotit algoritmus

Zásadním problémem je, že spolu násobíme dvě matice ve sdílené paměti

- nutnost provádět load instrukce spolu s MAD instrukcemi

Lze matice násobit ještě rychleji?

Naměřený výkon je již poměrně blízký teoretickému maximu

- rozdíl je dán spouštěním kernelu/threadů, synchronizací a pointerovou aritmetikou
- chceme-li dosáhnout vyšší rychlosti, je třeba přehodnotit algoritmus

Zásadním problémem je, že spolu násobíme dvě matice ve sdílené paměti

- nutnost provádět load instrukce spolu s MAD instrukcemi

Můžeme mít ve sdílené paměti jen jeden blok?

Přehodnocený blokový přístup

Namísto čtvercových bloků v matici C můžeme použít obdélníkové

Přehodnocený blokový přístup

Namísto čtvercových bloků v matici C můžeme použít obdélníkové

- provádíme iterativně rank-1 update bloků v C ze sloupce matice A a řádku matice B

Přehodnocený blokový přístup

Namísto čtvercových bloků v matici C můžeme použít obdélníkové

- provádíme iterativně rank-1 update bloků v C ze sloupce matice A a řádku matice B
- sloupce je nutno číst se sdílené paměti

Přehodnocený blokový přístup

Namísto čtvercových bloků v matici C můžeme použít obdélníkové

- provádíme iterativně rank-1 update bloků v C ze sloupce matice A a řádku matice B
- sloupce je nutno číst se sdílené paměti
- řádky můžeme načítat postupně, lze tedy použít data v registrech

Přehodnocený blokový přístup

Namísto čtvercových bloků v matici C můžeme použít obdélníkové

- provádíme iterativně rank-1 update bloků v C ze sloupce matice A a řádku matice B
- sloupce je nutno číst se sdílené paměti
- řádky můžeme načítat postupně, lze tedy použít data v registrech
- výsledný blok může být uložen v registrech

Přehodnocený blokový přístup

Namísto čtvercových bloků v matici C můžeme použít obdélníkové

- provádíme iterativně rank-1 update bloků v C ze sloupce matice A a řádku matice B
- sloupce je nutno číst se sdílené paměti
- řádky můžeme načítat postupně, lze tedy použít data v registrech
- výsledný blok může být uložen v registrech
- pracujeme tedy pouze s jedním operandem ve sdílené paměti, není nutný load

Přehodnocený blokový přístup

Namísto čtvercových bloků v matici C můžeme použít obdélníkové

- provádíme iterativně rank-1 update bloků v C ze sloupce matice A a řádku matice B
- sloupce je nutno číst se sdílené paměti
- řádky můžeme načítat postupně, lze tedy použít data v registrech
- výsledný blok může být uložen v registrech
- pracujeme tedy pouze s jedním operandem ve sdílené paměti, není nutný load
- není nutná aritmetika uprostřed smyčky (viz předchozí optimalizace)

Přehodnocený blokový přístup

Namísto čtvercových bloků v matici C můžeme použít obdélníkové

- provádíme iterativně rank-1 update bloků v C ze sloupce matice A a řádku matice B
- sloupce je nutno číst se sdílené paměti
- řádky můžeme načítat postupně, lze tedy použít data v registrech
- výsledný blok může být uložen v registrech
- pracujeme tedy pouze s jedním operandem ve sdílené paměti, není nutný load
- není nutná aritmetika uprostřed smyčky (viz předchozí optimalizace)
- teoretické maximum výkonu je tak omezeno rychlostí instrukce MAD pracující se sdílenou pamětí na cca 415 GFlops

Implementace

Nejvyšší rychlosti bylo dosaženo s konfigurací

- matice A zpracovávána po blocích 16×16, uložených ve sdílené paměti
- matice B zpracovávána po blocích 64×1, uložených v registrech
- bloky matice C mají tedy rozměr 64×16, jsou uloženy v registrech

Implementace

Nejvyšší rychlosti bylo dosaženo s konfigurací

- matice A zpracovávána po blocích 16×16, uložených ve sdílené paměti
- matice B zpracovávána po blocích 64×1, uložených v registrech
- bloky matice C mají tedy rozměr 64×16, jsou uloženy v registrech

Dosažená rychlost této implementace 375 GFlops.

Shrnutí

Implementace	rychlost	rel. Δ	abs. Δ
Naivní implementace, thready 1×128	3.9 GFlops		
Naivní implementace	36.6 GFlops	9.4×	9.4×
Blokový přístup	198 GFlops	5.4×	51×
Bloky 32×16 pracující s daty 32×16	235 GFlops	1.19×	60×
Odstranění ADD instrukce	276 GFlops	1.17×	71×
Jen jeden blok ve sdílené paměti	375 GFlops	1.36×	96×

Shrnutí

Implementace	rychlost	rel. Δ	abs. Δ
Naivní implementace, thready 1×128	3.9 GFlops		
Naivní implementace	36.6 GFlops	9.4×	9.4×
Blokový přístup	198 GFlops	5.4×	51×
Bloky 32×16 pracující s daty 32×16	235 GFlops	1.19×	60×
Odstranění ADD instrukce	276 GFlops	1.17×	71×
Jen jeden blok ve sdílené paměti	375 GFlops	1.36×	96×

- Nejzásadnější je redukce poměru aritmetických operací k paměťovým přenosům a základní optimalizace přístupu do paměti.

Shrnutí

Implementace	rychlost	rel. Δ	abs. Δ
Naivní implementace, thready 1×128	3.9 GFlops		
Naivní implementace	36.6 GFlops	9.4×	9.4×
Blokový přístup	198 GFlops	5.4×	51×
Bloky 32×16 pracující s daty 32×16	235 GFlops	1.19×	60×
Odstranění ADD instrukce	276 GFlops	1.17×	71×
Jen jeden blok ve sdílené paměti	375 GFlops	1.36×	96×

- Nejzásadnější je redukce poměru aritmetických operací k paměťovým přenosům a základní optimalizace přístupu do paměti.
- Optimalizace na úrovni instrukcí je relativně náročná, avšak pro kritické kódy může přinést relativně významné zrychlení.