Výkon GPU hardware

Jiří Filipovič

podzim 2010

Paralelismus GPU

Paralelní algoritmy je nutno navrhovat vzhledem k paralelismu, který poskytuje HW

- v případě GPU se jedná o pole SIMT multiprocesorů pracující nad společnou pamětí

Dekompozice pro GPU

- hrubozrnné rozdělení problému na části nevyžadující intenzivní komunikaci/synchronizaci
- jemnozrnné rozdělení blízké vektorizaci (SIMT je ale více flexibilní)

SIMT

Na jednu jednotku spouštějící instrukce připadá několik skalárních procesorů (SP)
G80

- 8 SP na jednotku spouštějící instrukce
- nová instrukce je spuštěna každé 4 cykly
- 32 thredů (tzv. *warp*) musí provádět stejnou instrukci

SIMT

Na jednu jednotku spouštějící instrukce připadá několik skalárních procesorů (SP)
G80

- 8 SP na jednotku spouštějící instrukce
- nová instrukce je spuštěna každé 4 cykly
- 32 thredů (tzv. *warp*) musí provádět stejnou instrukci

A co větvení kódu?

- pokud část threadů ve warpu provádí jinou instrukci, běh se serializuje
- to snižuje výkon, snažíme se divergenci v rámci warpu předejít

SIMT

Na jednu jednotku spouštějící instrukce připadá několik skalárních procesorů (SP)
G80

- 8 SP na jednotku spouštějící instrukce
- nová instrukce je spuštěna každé 4 cykly
- 32 thredů (tzv. *warp*) musí provádět stejnou instrukci

A co větvení kódu?

- pokud část threadů ve warpu provádí jinou instrukci, běh se serializuje
- to snižuje výkon, snažíme se divergenci v rámci warpu předejít

Multiprocesor je tedy MIMD (Multiple-Instruction Multiple-Thread) z programátorského hlediska a SIMT (Single-Instruction Multiple-Thread) z výkonového.

Vlastnosti threadů

Oproti CPU threadům jsou GPU thready velmi „jemné".

- jejich běh může být velmi krátký (desítky instrukcí)
- může (mělo by) jich být velmi mnoho
- nemohou využívat velké množství prostředků

Thready jsou seskupeny v blocích

- ty jsou spouštěny na jednotlivých multiprocesorech
- dostatečný počet bloků je důležitý pro škálovatelnost

Počet threadů a thread bloků na multiprocesor je omezen.

Maskování latence pamětí

Paměti mají latence

- globální paměť má vysokou latenci (stovky cyklů)
- registry a sdílená paměť mají read-after-write latenci

Maskování latence pamětí

Paměti mají latence

- globální paměť má vysokou latenci (stovky cyklů)
- registry a sdílená paměť mají read-after-write latenci

Maskování latence paměti je odlišné, než u CPU

- žádné provádění instrukcí mimo pořadí
- často žádná cache

Maskování latence pamětí

Paměti mají latence

- globální paměť má vysokou latenci (stovky cyklů)
- registry a sdílená paměť mají read-after-write latenci

Maskování latence paměti je odlišné, než u CPU

- žádné provádění instrukcí mimo pořadí
- často žádná cache

Když nějaký warp čeká na data z paměti, je možné spustit jiný

- umožní maskovat latenci paměti
- vyžaduje spuštění *řádově více* vláken, než má GPU jader
- plánování spuštění a přepínání threadů je realizováno přímo v HW bez overheadu

Maskování latence pamětí

Paměti mají latence

- globální paměť má vysokou latenci (stovky cyklů)
- registry a sdílená paměť mají read-after-write latenci

Maskování latence paměti je odlišné, než u CPU

- žádné provádění instrukcí mimo pořadí
- často žádná cache

Když nějaký warp čeká na data z paměti, je možné spustit jiný

- umožní maskovat latenci paměti
- vyžaduje spuštění *řádově více* vláken, než má GPU jader
- plánování spuštění a přepínání threadů je realizováno přímo v HW bez overheadu

Obdobná situace je v případě synchronizace.

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)
- latence 400-600 cyklů

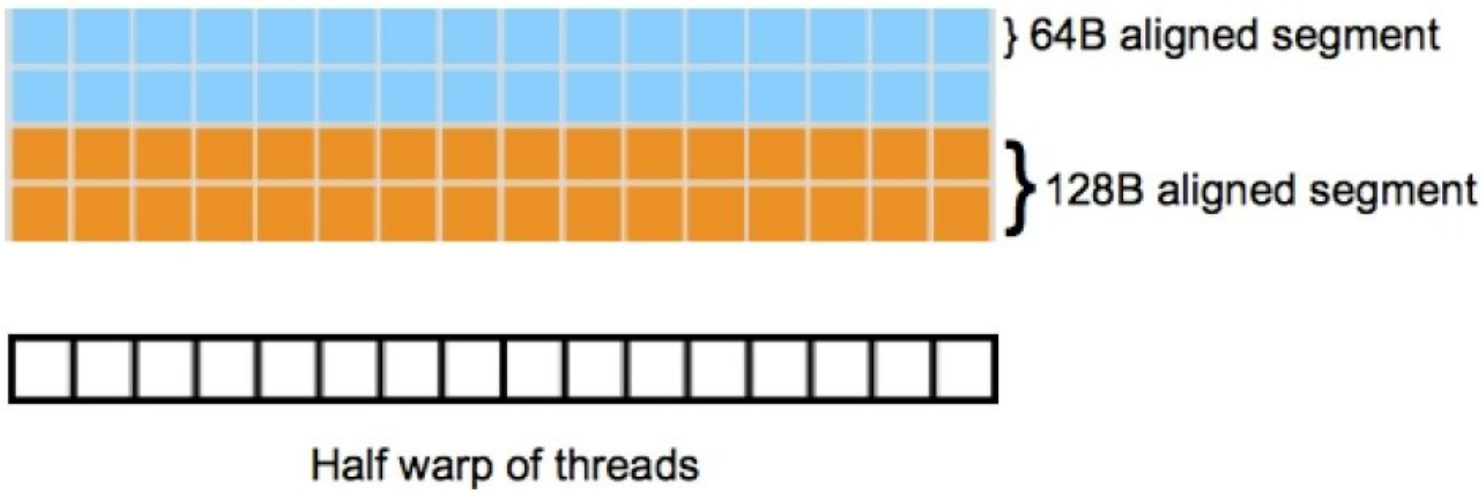
Při špatném vzoru paralelního přístupu do globální paměti snadno výrazně snížíme propustnost

- 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

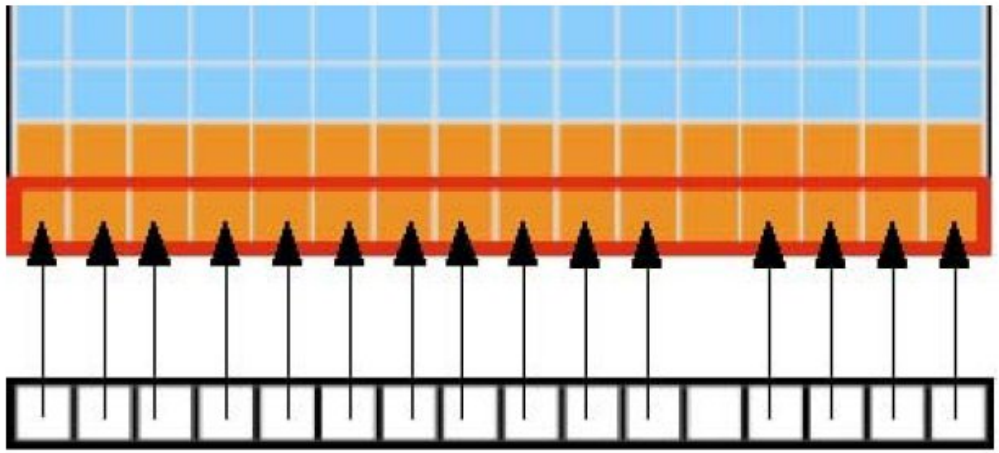
Spojitý přístup do paměti (c.c. < 2.0)

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

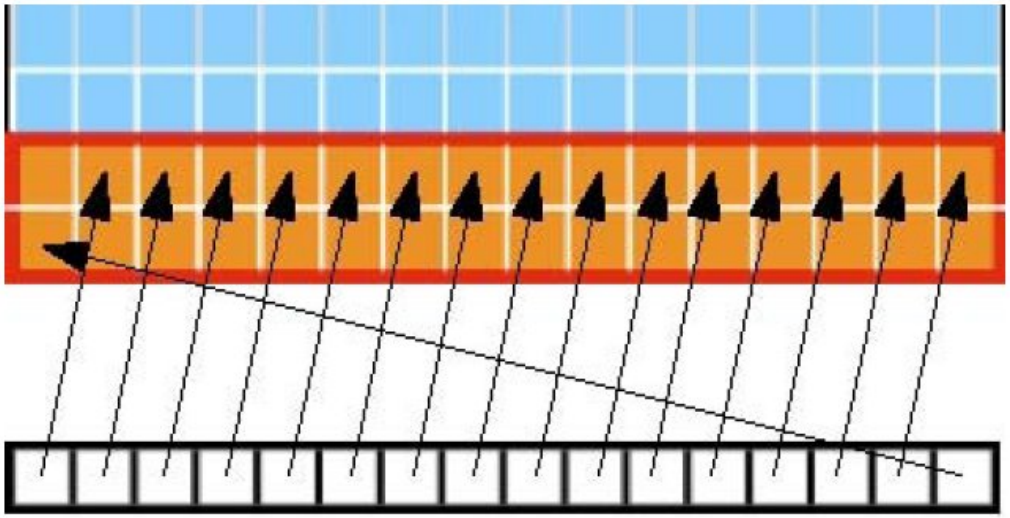
Spojitý přístup do paměti (c.c. < 2.0)

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



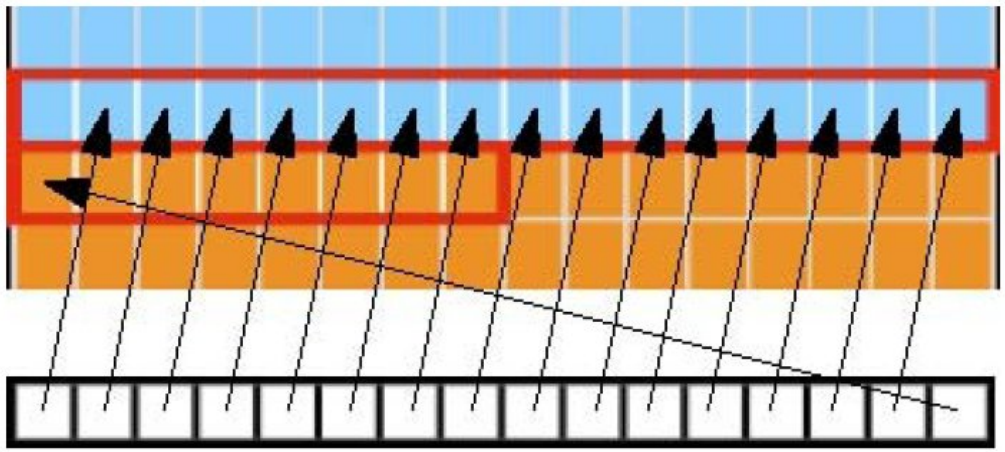
Nezarovnaný přístup do paměti (c.c. < 2.0)

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



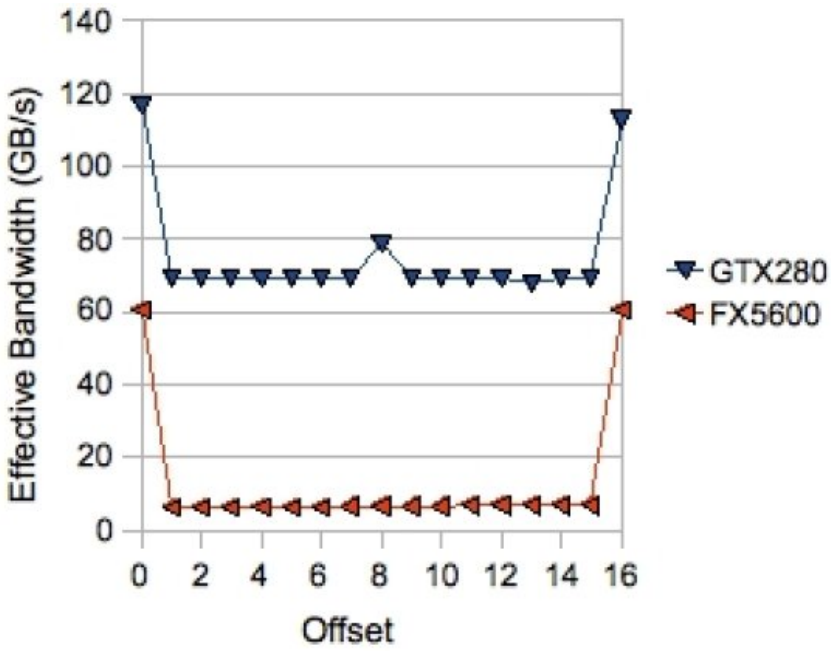
Nezarovnaný přístup do paměti(c.c. < 2.0)

Obdobný případ může vézt k nutnosti použít dvě transakce.



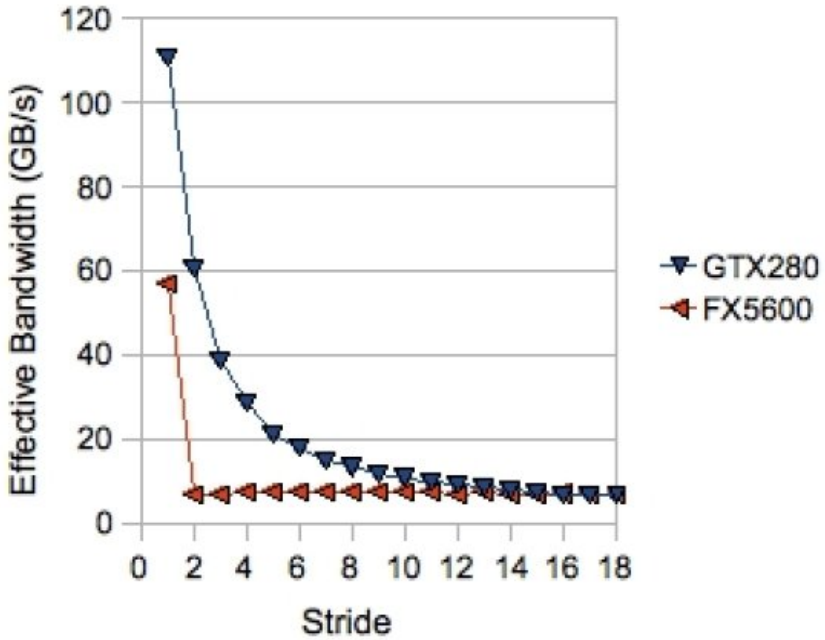
Výkon při nezarovnaném přístupu (c.c. < 2.0)

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: 256 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)

HW organizace sdílené paměti

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

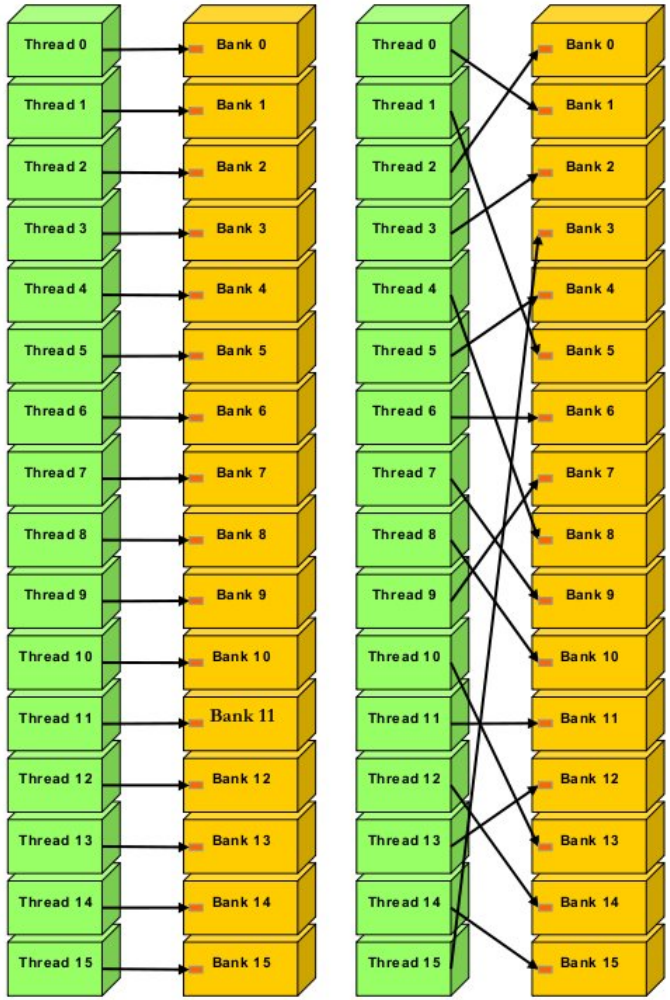
- c.c. 1.x 16 bank, c.c. 2.x 32 bank, paměťový prostor mapován prokládaně s odstupem 32 bitů
- pro dosažení plného výkonu paměti musíme přistupovat k datům v rozdílných bankách
- implementován broadcast – pokud všichni přistupují ke stejnému údaji v paměti

Konflikty bank

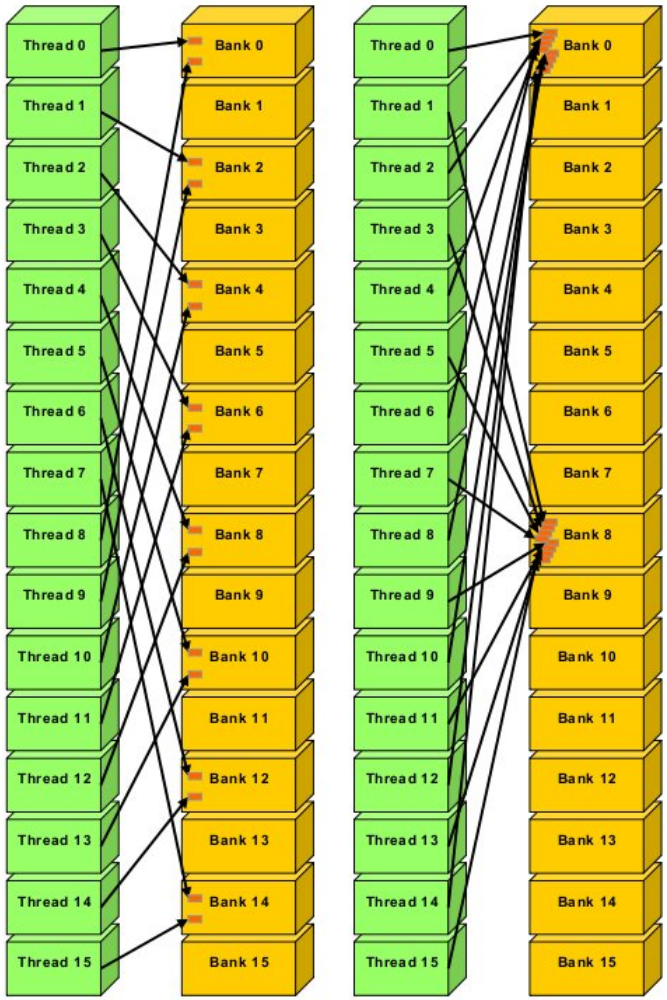
Konflikt bank

- dojde k němu, přistupují-li některé thready v warpu/půlwarpu k datům ve stejné paměťové bance (s vyjímkou, kdy thready přistupují ke stejnému místu v paměti)
- 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

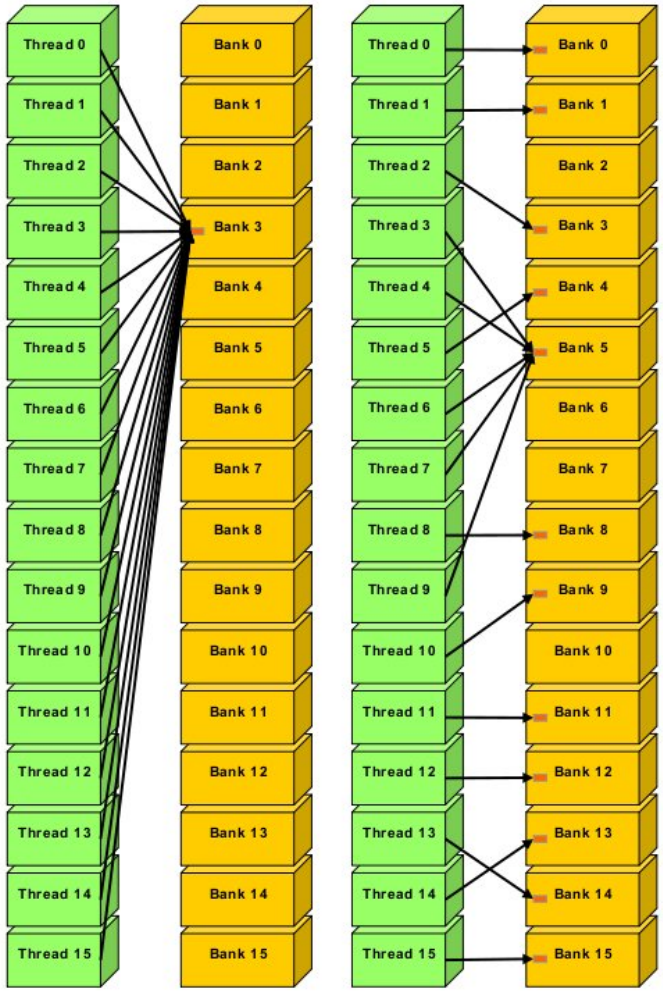
Přístup bez konfliktů



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 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 na c.c. 2.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 (tzv. partition camping, není problém u Fermi)

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, stačilo by napsat dobrý CPU algoritmus...

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 partition 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

Rychlost provádění instrukcí

Některé instrukce jsou v porovnání s ostatními pomalejší, než u procesoru

- celočíselné dělení a modulo
- 32-bitové násobení u c.c. 1.x
- 24-bitové násobení u c.c. 2.x

Některé jsou naopak rychlejší

- méně přesné verze prováděné na SFU
- *__sinf(x)*, *__cosf(x)*, *__expf(x)*, *__sincosf(x)*, *__rsqrtf(x)* aj.

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*

Součet prvků vektoru

Pro vektor v o n prvcích chceme spočítat $x = \sum_{i=1}^{n} v_i$.

Součet prvků vektoru

Pro vektor v o n prvcích chceme spočítat $x = \sum_{i=1}^{n} v_i$.
Zápis v jazyce C

```
int x = 0;
for (int i = 0; i < n; i++)
  x += v[i];
```

Jednotlivé iterace cyklu jsou na sobě závislé.

Součet prvků vektoru

Pro vektor v o n prvcích chceme spočítat $x = \sum_{i=1}^{n} v_i$.
Zápis v jazyce C

```
int x = 0;
for (int i = 0; i < n; i++)
  x += v[i];
```

Jednotlivé iterace cyklu jsou na sobě závislé.

- nemůžeme udělat všechnu práci paralelně
- sčítání je však (alespoň teoreticky :-)) asocitativní
- není tedy nutno počítat sekvenčně

Paralelní algoritmus

Představený sekvenční algoritmus provádí pro 8 prvků výpočet:

$$((((((v_1 + v_2) + v_3) + v_4) + v_5) + v_6) + v_7) + v_8$$

Paralelní algoritmus

Představený sekvenční algoritmus provádí pro 8 prvků výpočet:

$$((((((v_1 + v_2) + v_3) + v_4) + v_5) + v_6) + v_7) + v_8$$

Sčítání je asociativní... spřeházejme tedy závorky:
$((v_1 + v_2) + (v_3 + v_4)) + ((v_5 + v_6) + (v_7 + v_8))$

Paralelní algoritmus

Představený sekvenční algoritmus provádí pro 8 prvků výpočet:

$$((((((v_1 + v_2) + v_3) + v_4) + v_5) + v_6) + v_7) + v_8$$

Sčítání je asociativní... spřeházejme tedy závorky:
$((v_1 + v_2) + (v_3 + v_4)) + ((v_5 + v_6) + (v_7 + v_8))$
Nyní můžeme pracovat paralelně

- v prvním kroku provedeme 4 sčítání
- ve druhém dvě
- ve třetím jedno

Celkově stejné množství práce ($n - 1$ sčítání), ale v $\log_2 n$ paralelních krocích!

Paralelní algoritmus

Našli jsme vhodný paralelní algoritmus

- provádí stejné množství operací jako sériová verze
- při dostatku procesorů je proveden v logaritmickém čase

Sčítáme výsledky předešlých součtů

- předešlé součty provádělo více threadů
- vyžaduje globální bariéru

Naivní přístup

Nejjednodušší schéma algoritmu:

- kernel pro sudá $i < n$ provede *v[i] += v[i+1]*
- opakujeme pro $n /= 2$ dokud $n > 1$

Omezení výkonu

- $2n$ čtení z globální paměti
- n zápisů do globální paměti
- $\log_2 n$ volání kernelu

Na jednu aritmetickou operaci připadají 3 paměťové přenosy, navíc je nepříjemný overhead spouštění kernelu.

Využití rychlejší paměti

V rámci volání kernelu můžeme posčítat více, než jen dvojice

- každý blok bx načte m prvků do sdílené paměti
- provede redukci (ve sdílené paměti v $\log_2 m$ krocích)
- uloží pouze jedno číslo odpovídající $\sum_{i=m\cdot bx}^{m\cdot bx+m} v_i$

Výhodnější z hlediska paměťových přenosů i spouštění kernelů

- $n + \frac{n}{m} + \frac{n}{m^2} + .. + \frac{n}{m^{\log_m n}} = (n-1)\frac{m}{m-1}$
- přibližně $n + \frac{n}{m}$ čtení, $\frac{n}{m}$ zápisů
- $\log_m n$ spuštění kernelu

Implementace 1

```
__global__ void reduce1(int *v){
  extern __shared__ int sv[];

  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sv[tid] = v[i];
  __syncthreads();

  for(unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0)
      sv[tid] += sv[tid + s];
    __syncthreads();
  }

  if (tid == 0)
    v[blockIdx.x] = sv[0];
}
```

Výkon

Vysoká úroveň divergence

- první iteraci pracuje každý 2. thread
- druhou iteraci pracuje každý 4. thread
- třetí iteraci pracuje každý 8 thread
- atd.

Přenos (GTX 280) 3.77 GB/s, 0.94 MElem/s.

Implementace 2

Nahradíme indexaci ve for cyklu

```
for (unsigned int s = 1; s < blockDim.x; s *= 2) {
  int index = 2 * s * tid;
  if (index < blockDim.x)
    sv[index] += sv[index + s];
  __syncthreads();
}
```

Přenos 8.33 GB/s, 2.08 MElem/s.

Řeší divergenci, generuje konflikty bank.

Implementace 3

Tak ještě jinak...

```
for (unsigned int s = blockDim.x/2; s > 0; s >>= 1) {
  if (tid < s)
    sv[tid] += sv[tid + s];
  __syncthreads();
}
```

Žádná divergence ani konflikty.

Přenos 16.34 GB/s, 4.08 MElem/s.

Polovina threadů nic nepočítá...

Implementace 4

První sčítání provedeme již během načítání.

```
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sv[tid] = v[i] + v[i+blockDim.x];
```

Přenos 27.16 GB/s, 6.79 MElem/s.
Data zřejmě čteme optimálně, stále je zde však výkonová rezerva – zaměřme se na instrukce.

Implementace 5

V jednotlivých krocích redukce ubývá aktivních threadů

- nakonec bude pracovat pouze jeden warp
- ten je však synchronizován implicitně, můžeme tedy odebrat *__syncthreads()*
- podmínka $if\,(tid < s)$ je zde zbytečná (nic neušetří)

Unrollujme tedy poslední warp...

Implementace 5

```
for (unsigned int s = blockDim.x/2; s > 32; s >>= 1){
  if (tid < s)
    sv[tid] += sv[tid + s];
  __syncthreads();
}

if (tid < 32){
  sv[tid] += sv[tid + 32];
  sv[tid] += sv[tid + 16];
  sv[tid] += sv[tid + 8];
  sv[tid] += sv[tid + 4];
  sv[tid] += sv[tid + 2];
  sv[tid] += sv[tid + 1];
}
```

Ušetříme čas i ostatním waprům (zkončí dříve s for cyklem).
Přenos 37.68 GB/s, 9.42 MElem/s.

Implementace 6

Jak je to s rozvinutím for cyklu?
Známe-li počet iterací, můžeme cyklus rozvinout

- počet iterací je závislý na velikosti bloku

Můžeme být obecní?

- algoritmus pracuje s bloky o velikosti 2^n
- velikost bloku je shora omezena
- známe-li při kompilaci velikost bloku, můžeme použít šablonu

```
template <unsigned int blockSize>
__global__ void reduce6(int *v)
```

Implementace 6

Podmínky s *blockSize* se vyhodnotí již pří překladu:

```
if (blockSize >= 512){
  if (tid < 256)
    sv[tid] += sv[tid + 256];
  __syncthreads();
}
if (blockSize >= 256){
  if (tid < 128)
    sv[tid] += sv[tid + 128];
  __syncthreads();
}
if (blockSize >= 128){
  if (tid < 64)
    sv[tid] += sv[tid + 64];
  __syncthreads();
}
```

Implementace 6

```
if (tid < 32){
  if (blockSize >= 64) sv[tid] += sv[tid + 32];
  if (blockSize >= 32) sv[tid] += sv[tid + 16];
  if (blockSize >= 16) sv[tid] += sv[tid + 8];
  if (blockSize >= 8) sv[tid] += sv[tid + 4];
  if (blockSize >= 4) sv[tid] += sv[tid + 2];
  if (blockSize >= 2) sv[tid] += sv[tid + 1];
}
```

Spuštění kernelu:

```
reduce6<block><<<grid, block, mem>>>(d_v);
```

Přenos 50.64 GB/s, 12.66 MElem/s.

Implementace 7

Můžeme algoritmus ještě vylepšit?
Vraťme se zpět ke složitosti:

- celkem $\log n$ kroků
- celkem $n - 1$ sčítání
- časová složitost pro p threadů běžících paralelně (p procesorů) $\mathcal{O}(\frac{n}{p} + \log n)$

Cena paralelního výpočtu

- definována jako počet procesorů krát časová složitost
- přidělíme-li každému datovému elementu jeden thread, lze uvažovat $p = n$
- pak je cena $\mathcal{O}(n \cdot \log n)$
- není efektivní

Implementace 7

Snížení ceny

- použijeme $\mathcal{O}(\frac{n}{\log n})$ threadů
- každý thread provede $\mathcal{O}(\log n)$ sekvenčních kroků
- následně se provede $\mathcal{O}(\log n)$ paralelních kroků
- časová složitost zůstane
- cena se sníží na $\mathcal{O}(n)$

Co to znamená v praxi?

- redukujeme práci spojenou s vytvářením threadu a pointerovou aritmetikou
- to přináší výhodu v momentě, kdy máme výrazně více threadů, než je třeba k saturaci GPU
- navíc snižujeme overhead spouštění kernelů

Implementace 7

Modifikujeme načítání do sdílené paměti

```
unsigned int gridSize = blockSize*2*gridDim.x;
sv[tid] = 0;

while(i < n){
  sv[tid] += v[i] + v[i+blockSize];
  i += gridSize;
}
__syncthreads();
```

Přenos 77.21 GB/s, 19.3 MElem/s.

Implementace 7

Modifikujeme načítání do sdílené paměti

```
unsigned int gridSize = blockSize*2*gridDim.x;
sv[tid] = 0;

while(i < n){
  sv[tid] += v[i] + v[i+blockSize];
  i += gridSize;
}
__syncthreads();
```

Přenos 77.21 GB/s, 19.3 MElem/s.

Jednotlivé implementace jsou k nalezení v CUDA SDK.