

Optimalizace pro GPU hardware

Jiří Filipovič

jaro 2016

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 threadů (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 warpů 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 threadů (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 warpů předejít

Multiprocessor 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 „lehké“.

- 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 multiprocesech
- dostatečný počet bloků je důležitý pro škálovatelnost

Počet threadů a thread bloků na multiprocessor je omezen.

Maskování latence paměti

Paměti mají latence

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

Maskování latence paměť

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 paměti

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 *řadově více* vláken, než má GPU jader
- plánování spuštění a přepínání threadů je realizováno přímo v HW bez overheadu

Maskování latence paměti

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 *řadově více* vláken, než má GPU jader
- plánování spuštění a přepínání threadů je realizováno přímo v HW bez overheadu

Obdobná situace je v případě synchronizace.

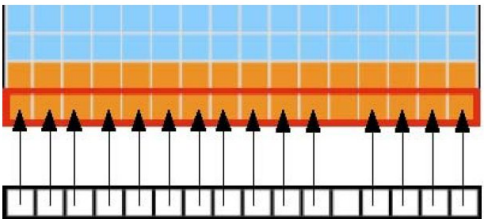
Sdružený přístup do paměti (c.c. < 2.0)

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

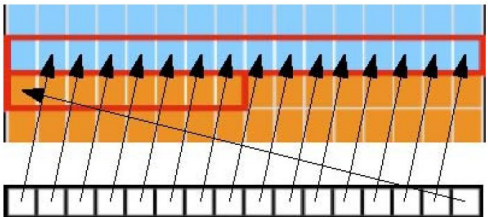
Sdružený přístup do paměti (c.c. < 2.0)

Threads jsou zarovnané, blok elementů souvislý, pořadí není permutované – sdružený přístup na všech GPU.



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

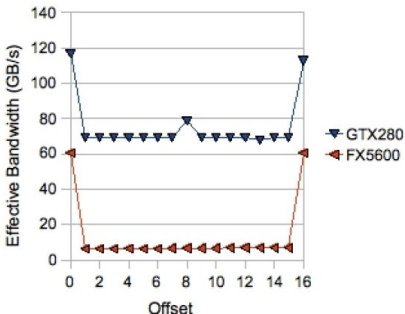
Obdobný případ může vézt k nutnosti použít dvě transakce.



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

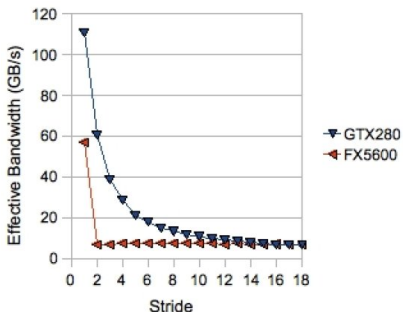
Starší GPU provádí pro každý element nejmenší možný přenos, tedy 32-bytů, což redukuje výkon na 1/8.

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 dramatičticky klesá.

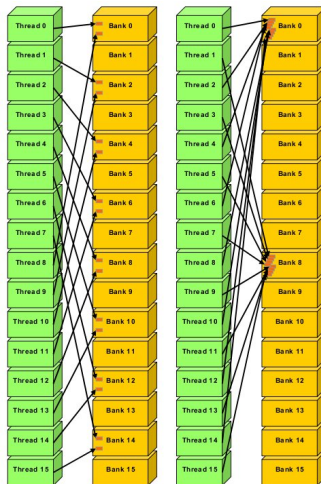


HW organizace sdílené paměti

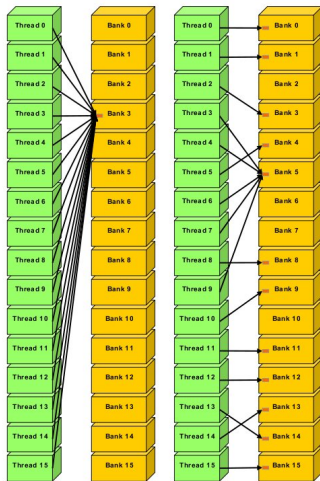
Sdílená paměť je organizována do paměťových bank, ke kterým je možné přistupovat paralelně

- 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

Vícecestné konflikty



Broadcast



Vzory přístupu

Zarovnání není třeba, negeneruje bank conflicts

```
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 přistupující k proměnné násobek 16

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

Komunikace CPU-GPU

Přenosy mezi systémovou a grafickou pamětí

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

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 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 dlaždicích

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

Odstranění prokládání

Matici můžeme zpracovávat po dlaždicích

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

Jak velké dlaždice použít?

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

Dlaždicová 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 dlaždic 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 dlaždic 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

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];
```


Zhodnocení výkonu

Veškeré optimalizace sloužily pouze k lepšímu přizpůsobení-se vlastnostem HW

- přesto jsme dosáhli $17.6\times$ zrychlení
- při formulaci algoritmu je nezbytné věnovat pozornost hardwareovým omezením
- jinak můžeme ztratit výhodu vyššího výkonu GPU...

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 nezatelný
- po odstranění partition campingu by se však již konflikty bank výkonnostně projevíly!
- 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 celých čísel u c.c. 1.x
- 24-bitové násobení celých čísel 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 podmíněné 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*

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 $n \neq 2$ 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.

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];
}

```

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()`
 - musíme sčítat přes `volatile` proměnnou
- podmínka `if(tid < s)` je zde zbytečná (nic neušetří)

Můžeme unrollovat poslední warp. Pozor, jedná se o **porušení obecné korektnosti kódu** (funkční pouze na HW s velikostí warpu dělitelnou 32, což jsou zatím všechna NVIDIA i AMD GPU).

Implementace 5

```
float mySum = 0;

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

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

Ušetříme čas i ostatním warpům (skončí 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 při překladu:

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

Implementace 6

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)
 $\mathcal{O}\left(\frac{n}{p} + \log n\right)$

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

Co to znamená v praxi?

- redukuje 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.

Divergence kódu

Divergence kódu

- serializace, divergují-li thready uvnitř warpu
- nalezení nedivergujícího algoritmu může být snadné
 - redukce
- ale také může prakticky znemožnit akceleraci některých jinak dobře paralelizovatelných algoritmů
 - mnoho nezávislých stavových automatů
 - nutnost zamyslet se nad výrazně odlišným algoritmem pro daný problém

Divergence přístupu do paměti

Divergence přístupu do paměti

- není-li do paměti přístupováno po souvislých blocích v rámci warpu, snižuje se její propustnost
- často velmi těžko překonatelný problém
 - průchod obecného grafu
- může vyžadovat využití odlišných datových struktur
 - práce s řídkými maticemi
- u rigidnějších struktur si lze často pomoci on-chip pamětí
 - transpozice matic

