Obecné výpočty na GPU v jazyce CUDA

Jiří Filipovič

Obsah přednášky

- motivace
- architektura GPU
- CUDA – programovací model
- jaké algoritmy urychlovat na GPU?
- optimalizace

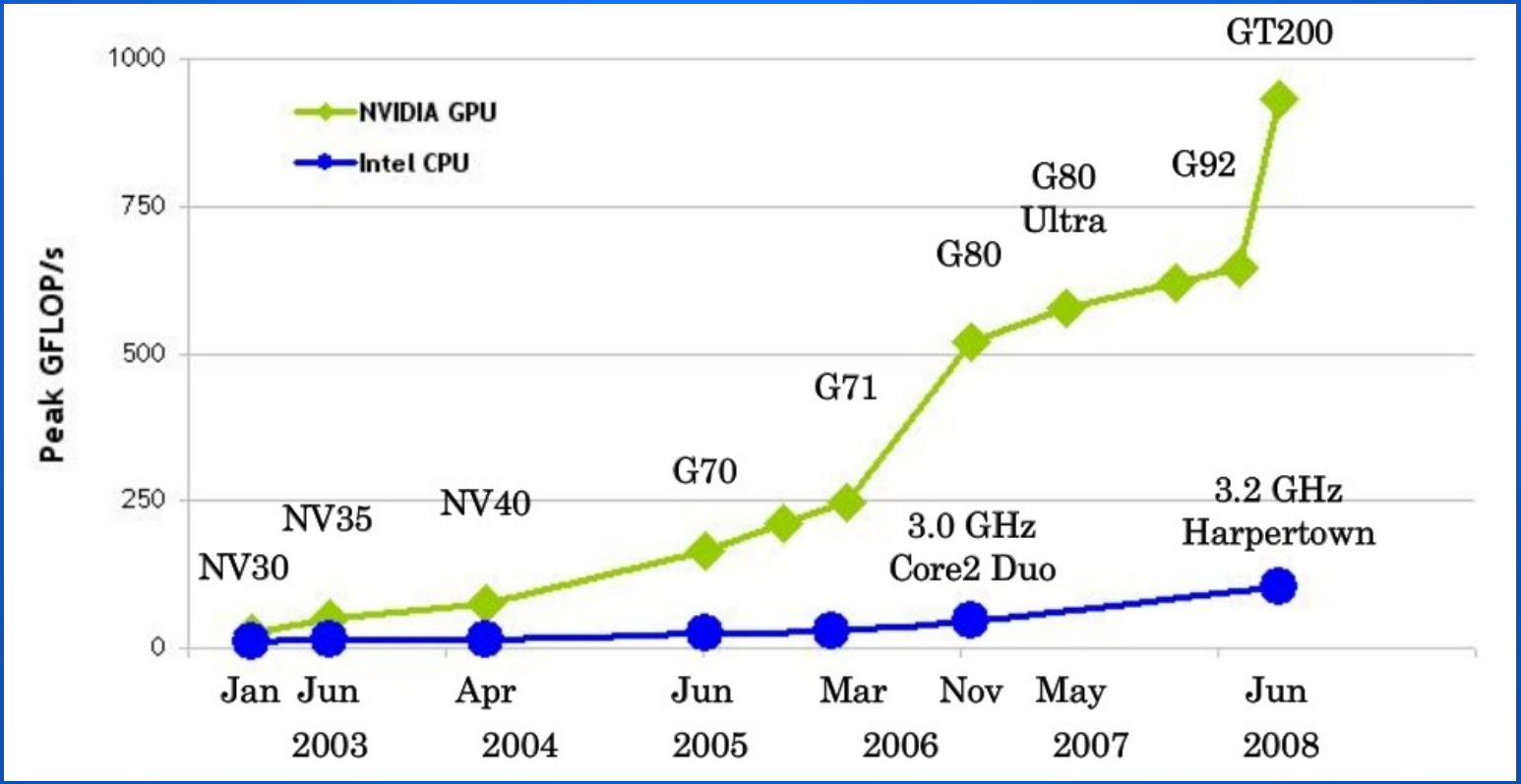
Motivace

- Moorův zákon stále platí pro počet tranzistorů, ne pro kmitočet
  - nové procesory jsou „širší“, ne rychlejší
  - rychlost sekvenčního provádění kódu přestává růst
- ke zrychlení výpočtů je nutná jejich paralelizace
  - na úrovni instrukcí
  - změnou programovacího paradigmatu

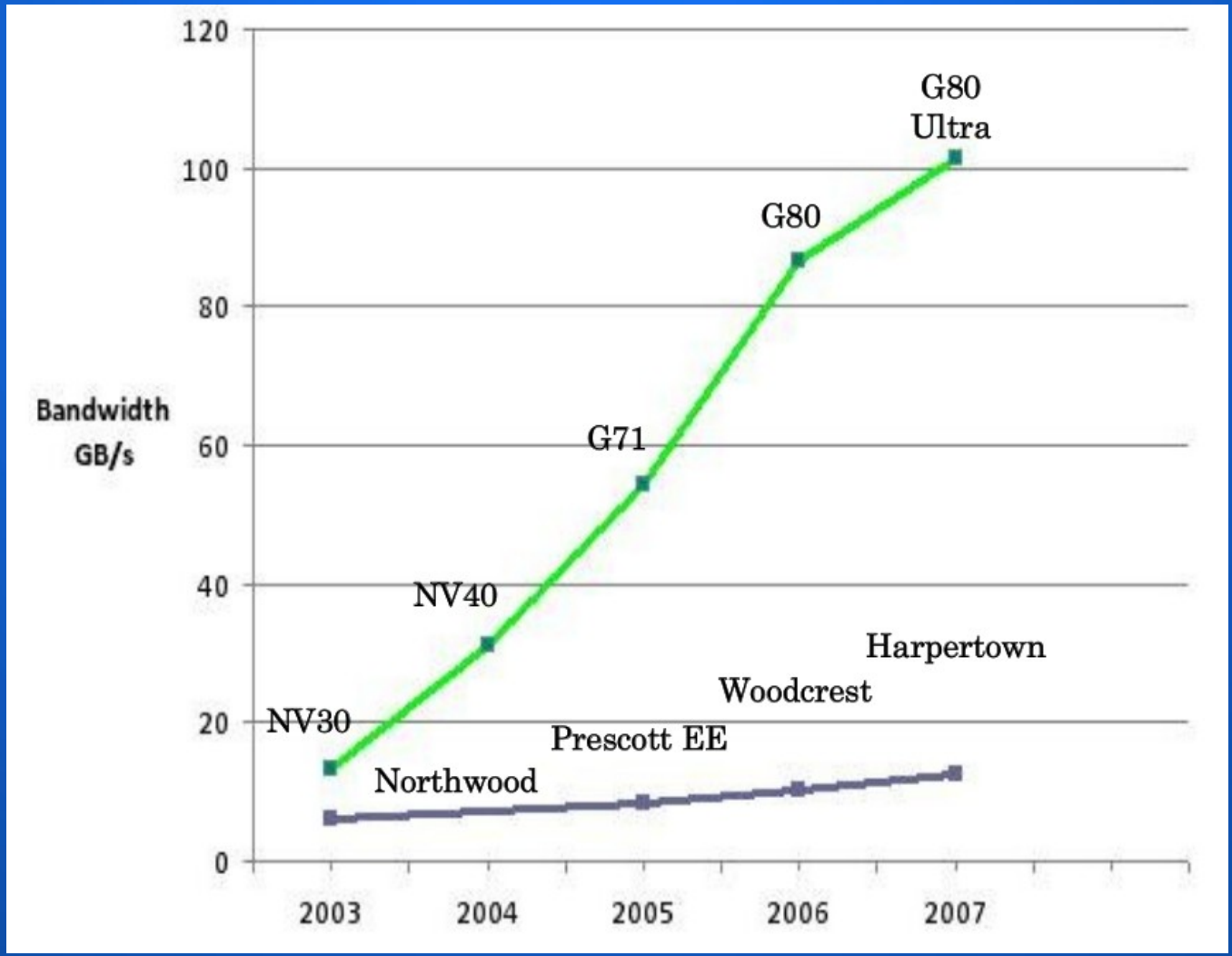
Motivace

- úlohový (task) paralelismus
  - vhodný pro menší počet výkonných procesorových jader
- datový paralelismus
  - jádra mohou být jednodušší, lépe škáluje
- z pohledu programátora
  - rozdílné paradigma znamená rozdílný pohled na návrh algoritmů
  - různé algiritmy poskytují různé možnosti paralelizace

Motivace


Peak GFLOP/s
1000
750
500
250
0
NVIDIA GPU
Intel CPU
GT200
G92
G80
Ultra
G80
G71
G70
NV40
NV35
NV30
3.0 GHz
Core2 Duo
3.2 GHz
Harpertown
Jan
Jun
2003
Apr
2004
Jun
2005
Mar
Nov
2006
May
2007
Jun
2008

Motivace


120
100
80
60
40
20
0
Bandwidth
GB/s
2003
2004
2005
2006
2007
G80
Ultra
G80
G71
NV40
NV30
Harpertown
Woodcrest
Prescott EE
Northwood

Motivace -- příklady zrychlení

- faktorizace matic 3-5.5x (LU, Cholesky)
- FFT 10-30x
- generování náhodných čísel 23-59x
- první fáze detekce kolizí 26x
- simulace neuronové sítě 10x
- mapa coulombovského potenciálu 44x
  - ostatní algoritmy pro molekulovou dynamiku 5-23x

Motivace -- shrnutí

- I pro využití moderních procesorů je zapotřebí programovat paralelně
  - paralelní architektura GPU přestává být řádově náročnější
- GPU jsou rozšířené
  - masivní produkce snižuje na cenu
  - spousta uživatelů má na stole superpočítač
- GPU jsou výkonné
  - řádový nárůst výkonu stojí za studium odlišného paradigmatu

Architektura GPU

- CPU
  - jednotky jader
  - out-of-order
  - velká cache
  - složitá logika
  - nízký vektorový paralelismus v rámci jádra
- GPU
  - desítky multiprocesorů
  - in-order SIMT
  - malá cache
  - jednoduchá logika
  - vysoký vektorový paralelismus v rámci multiprocesoru

- GPU používá více tranzistorů pro výpočetní jednotky => vyšší výkon, méně univerzální použití

Architektura G80

- 16 multiprocesorů
- multiprocesor
  - 8 skalárních procesorů (SP)
  - 2 jednotky pro speciální funkce (SFU)
  - až 768 threadů
    - HW přepínání a plánování spouštění
  - thready organizovány po 32 do warpů
    - SIMT (single instruction multiple thread)
  - nativní synchronizace v rámci multiprocesoru

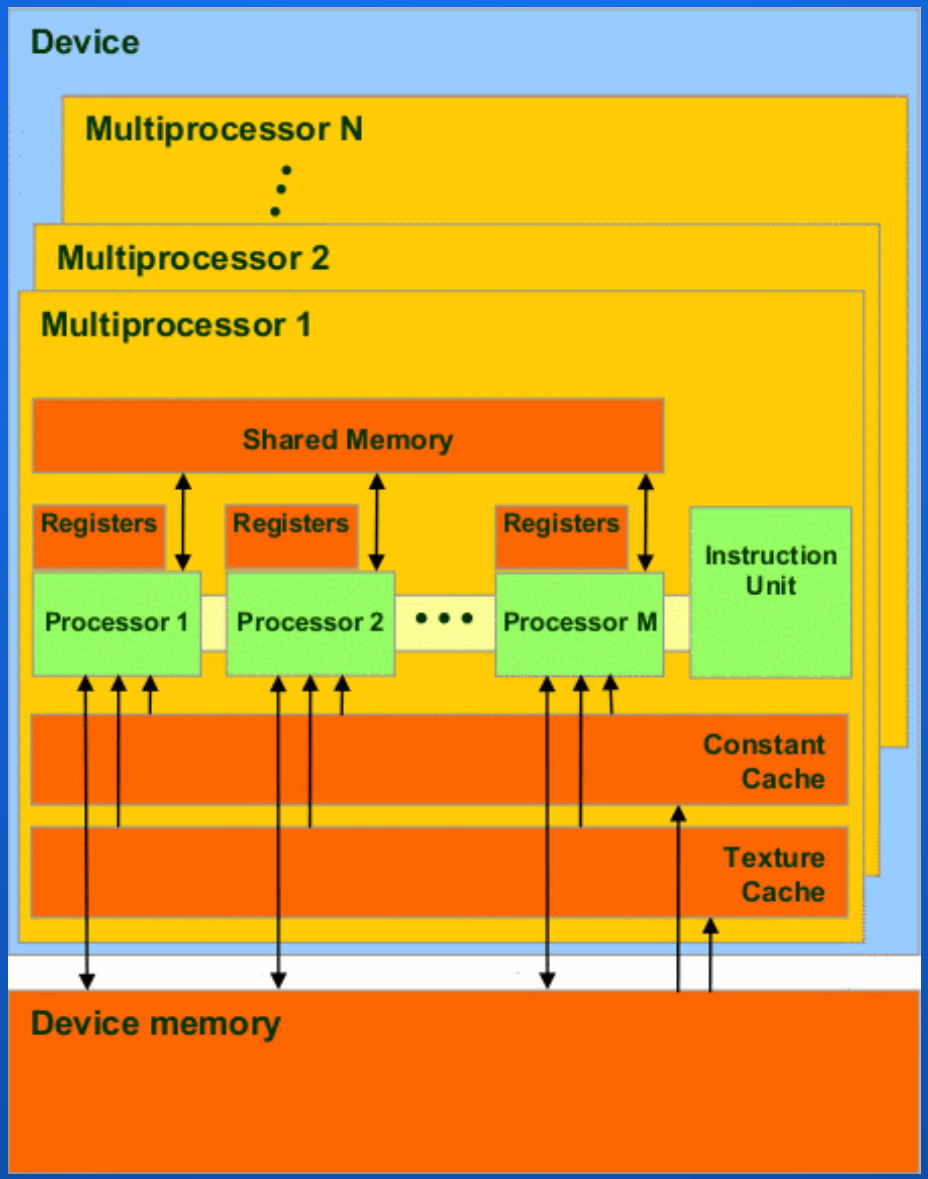
Paměťový model G80

- 8192 registrů sdílených mezi všemi thready v multiprocesoru
- 16KB sdílené paměti
  - lokální pro každý multiprocesor
  - stejně rychlá jako registry
    - pokud nedojde k bank konfliktům
- paměť konstant
  - cacheována
  - pouze pro čtení

Paměťový model G80

- paměť pro textury
  - cacheována, 2D prostorová lokalita
  - pouze pro čtení
- globální paměť
  - pro čtení i zápis, necacheovaná
  - 86 GB/s při zarovnaném přístupu
- přenosy mezi systémovou a grafickou RAM
  - omezeny rychlostí PCI-E (3.2 GB/s pro 1.0)

Architektura



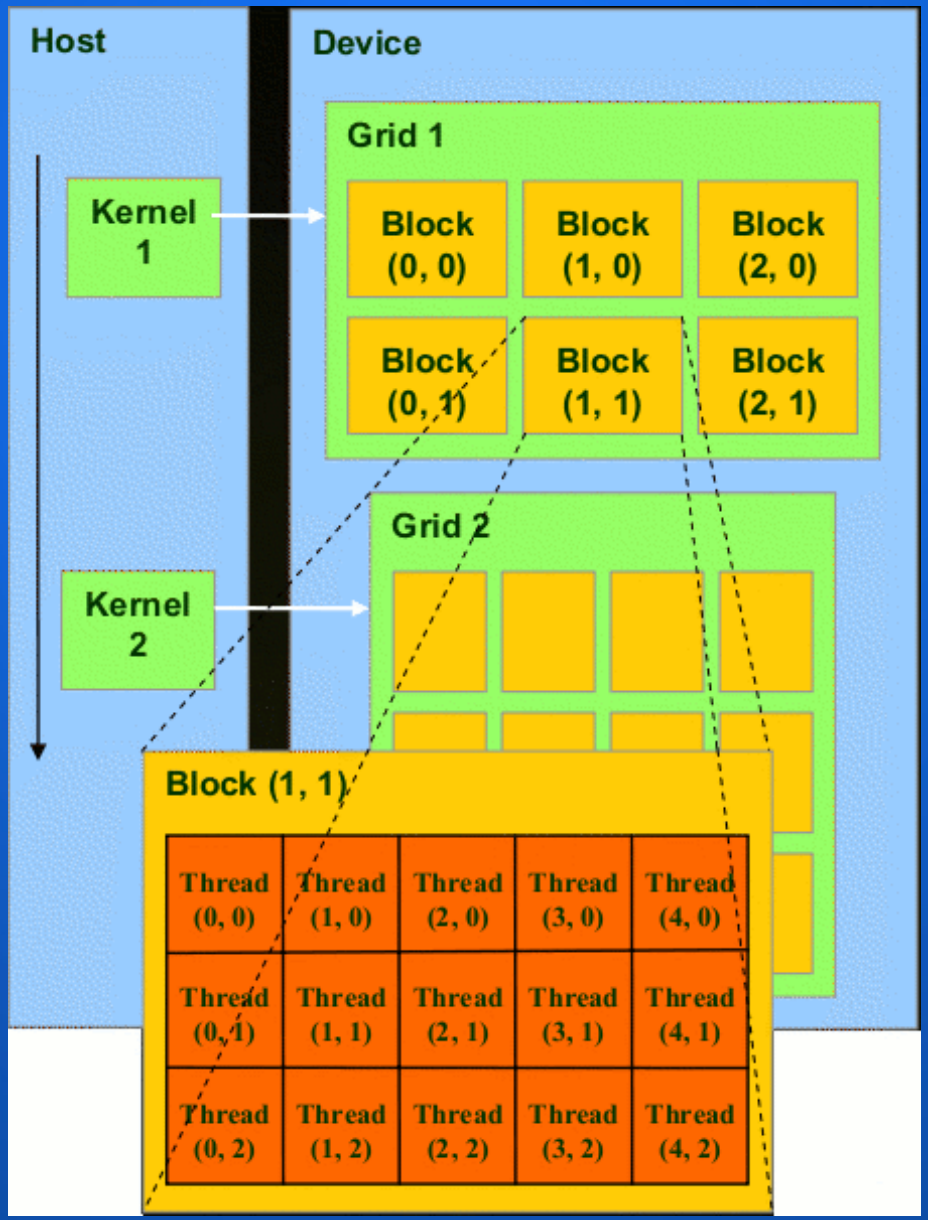
CUDA

- programovací rozhraní pro obecné výpočty na nVidia GPU
- odvozeno z jazyka C
  - jen velmi malé změny
  - samotný jazyk snadný k naučení se
- zapsaný GPU kód představuje jeden thread
- významný pokrok oproti „hackování grafického API“

CUDA – programovací model

- explicitně oddělen *host* (CPU) a *device* (GPU) kód
- GPU kód spouštěný z CPU se nazývá *kernel*
- kernel je tvořen mřížkou (1-2D) thread bloků (1-3D)
  - všechny thready z thread bloku běží na jednom multiprocesoru
  - HW plánuje spouštění thread bloků na multiprocesorech

Organizace threadů



Součet vektorů

```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

saxpy_serial(n, 2.0, x, y);
```

C++

```
__global__ void saxpy_parallel(int n, float a, float *x,
    float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

CUDA

Překažky vývoje pro GPU

- nutnost vysoké paralelizace
  - tisíce threadů namísto jednotek/desítek u SMP
- nemožnost efektivní spolupráce všech threadů
- běh algoritmu i přístup do paměti musí být velmi „pravidelný“
- PCI-E může být snadno bottleneck

Jaké výpočty urychlovat?

- velké instance problémů
- kritické pro výkon aplikace
- možná dekompozice do tisíců threadů
- žádná nebo málo častá globální synchronizace
- vektorový charakter výpočtů
- vysoký podíl aritmetických operací, nebo závislost na propustnosti paměti

Mapování funkce

- stejná funkce aplikovaná na mnoho prvků
  - silová pole
  - n-body problém
- relativně snadná implementace, vysoký výkon
- často využití SFU
  - GPU překonává CPU více, než lze předpokládat podle aritmetické síly (colombovský potenciál)

Paralelní redukce

- asociativní funkce aplikovaná postupně na všechny prvky vektoru
- sériová i paralelní verze celkem O(n) kroků
- z hlediska optimalizace už není zcela triviální
  - nepravidelný běh
  - nutné synchronizace
  - složitější přístup do paměti

Lineární algebra

- Basic Linear Algebra Subprograms
  - široce používané v HPC
  - mnoho výrobců HW má vlastní BLAS implementace (nVIDIA -- CUBLAS)
- BLAS1 -- operace vektor-vektor, O(n)
- BLAS2 -- operace vektor-matice, $O(n^2)$
- BLAS3 – operace matice-matice, $O(n^3)$

Lineární algebra

- BLAS1 a BLAS2 funkce
  - omezené propustností paměti
  - vhodné, pokud jich provádíme více po sobě, jinak omezeny rychlosti čtení z RAM a přenosu po PCI-E
- BLAS3 funkce
  - $O(n^3)$ aritmetických operací, $O(n^3/k)$ operací s globální pamětí
  - využijí aritmetickou sílu GPU

Mnoho dalších uplatnění

- třídění, vyhledávání (zpravidla těží z propustnosti paměti)
- zpracování signálů
  - fyzikální děje
  - filtrování obrazu
  - komprese obrázků, audia, videa
- řešení diferenciálních rovnic

Optimalizace v CUDA

- zaměření optimalizace
  - skrývání latence globální paměti
  - redistribuce práce v threadech a blocích threadů
  - paralelismus v rámci threadu
  - optimalizace sekvenčního kódu
  - optimalizace využití zdrojů
- teoretické maximum GPU je dobrým měřítkem, kolik je třeba ještě optimalizovat

Skrývání latence globální paměti

- GMEM má latenci 400-600 cyklů
- GPU zpracovává instrukce in-order
- pokud thready čekají na globální paměť, HW plánovač spustí warp, který může něco dělat
- k odstínění latence je zapotřebí
  - mít dostatečný počet threadů současně běžících na multiprocesoru
  - každý thread musí provádět dostatečné množství práce mezi synchronizacemi

Redistribuce práce

- jeden thread může dělat více práce, je-li část operací společná
  - pozor na redukci paralelismu!
- thready v rámci bloku mohou využívat sdílenou paměť
  - společné načtení dat, využívaných i ostatními thready
  - redukuje paměťové přenosy, ale omezuje volnost plánování warpů (hodně threadů se synchronizuje po operacích v globální paměti)

Paralelismus v rámci threadu

- GPU není ANDES, pokud ale po načtení následují instrukce nepracující s načítaným operandem, jsou prováděny
- lze využít prefetching
  - redukuje latenci
  - zvyšuje nároky na lokální paměťové zdroje a tím potenciálně omezuje paralelismus

Optimalizace sekvenčního kódu

- obdobné jako u CPU
  - přejmenování proměnných
  - eliminace subvýrazů
  - vyjmutí invariantu z cyklů
  - rozvinutí cyklů
    - ne nutně vnitřních
  - ...

Optimalizace využití zdrojů

- optimalizace zrychlující běh jednoho threadu mohou snižovat celkový výkon, pokud spotřebují více zdrojů
- změny nejsou spojité
  - 256 threadů v bloku, 16 registrů na thread = 512 threadů na multiprocesor, zvýšení nároků o 1 registr sníží počet threadů na polovinu
- náročný vícerozměrný problém
  - prostor pro autotuning

HW omezení -- globální paměť

- zarovnaný přístup (zrychlení až 10x)
  - polovina warpu (16 threadů) čte synchronně
    - k-tý thread čte k-tou pozici nebo nic
    - startovací adresa dělitelná 16 x délka čtení
    - instrukce čte 32, 64, nebo 128 bitů
  - novější GPU mají méně omezení
- skrývání latence (zrychlení až 2,5x)
  - více warpů na multiprocesor

HW omezení -- sdílená paměť

- nutno předejít bank konfliktům
  - 16 paměťových bank
  - pokud čte více threadů z jedné banky, serializují se
    - výjimka – pokud čte celý half-warp stejnou adresu, provede se broadcast
- bez bank konfliktů 2 hodinové tiky na přenos 32 bitů pro každou banku
  - stejně rychlé jako registry

HW omezení -- ostatní paměti

- texturová paměť
  - optimalizace na 2D prostorovou lokalitu
- paměť konstant
  - podobný přístup jako u CPU
- přenos mezi pamětí systému a GPU
  - přenášet co nejméně
  - musíme-li přenášet, pak co největší bloky současně

HW omezení -- výpočet

- velikost thread bloku násobek velikosti warpu
  - předejde plýtvání paralelismem v kartě
- obcházení některých aritmetických operací
  - pomalá některá celočíselná aritmetika, velmi pomalé modulo
- odchylky od IEEE-754
  - zpravidla méně přesné
  - SFU výrazně rychlejší na úkor přesnosti

Otázky?

Děkuji za pozornost