Základy obecných výpočtů na GPU

Jiří Filipovič

jaro 2014

Motivace – Moorův zákon

Moorův zákon

Počet tranzistorů na jednom čipu se přibližně každých 18 měsíců zdvojnásobí.

Motivace – Moorův zákon

Moorův zákon

Počet tranzistorů na jednom čipu se přibližně každých 18 měsíců zdvojnásobí.

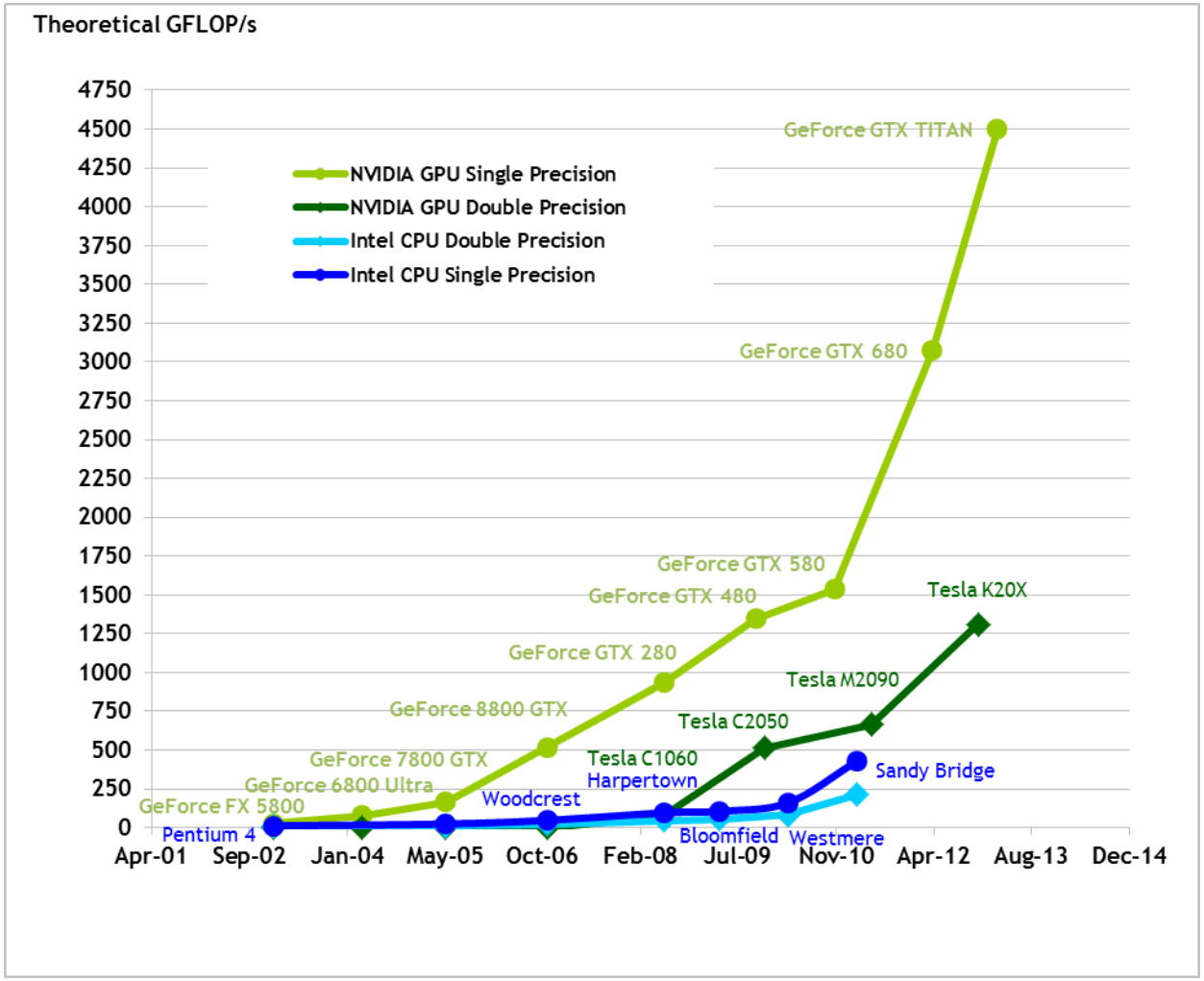
Adekvátní růst výkonu je zajištěn:

- **dříve** zvyšováním frekvence, instrukčním paralelismem, out-of-order spouštěním instrukcí, vyrovnávacími pamětmi atd.
- **dnes** vektorovými instrukcemi, zmnožováním jader

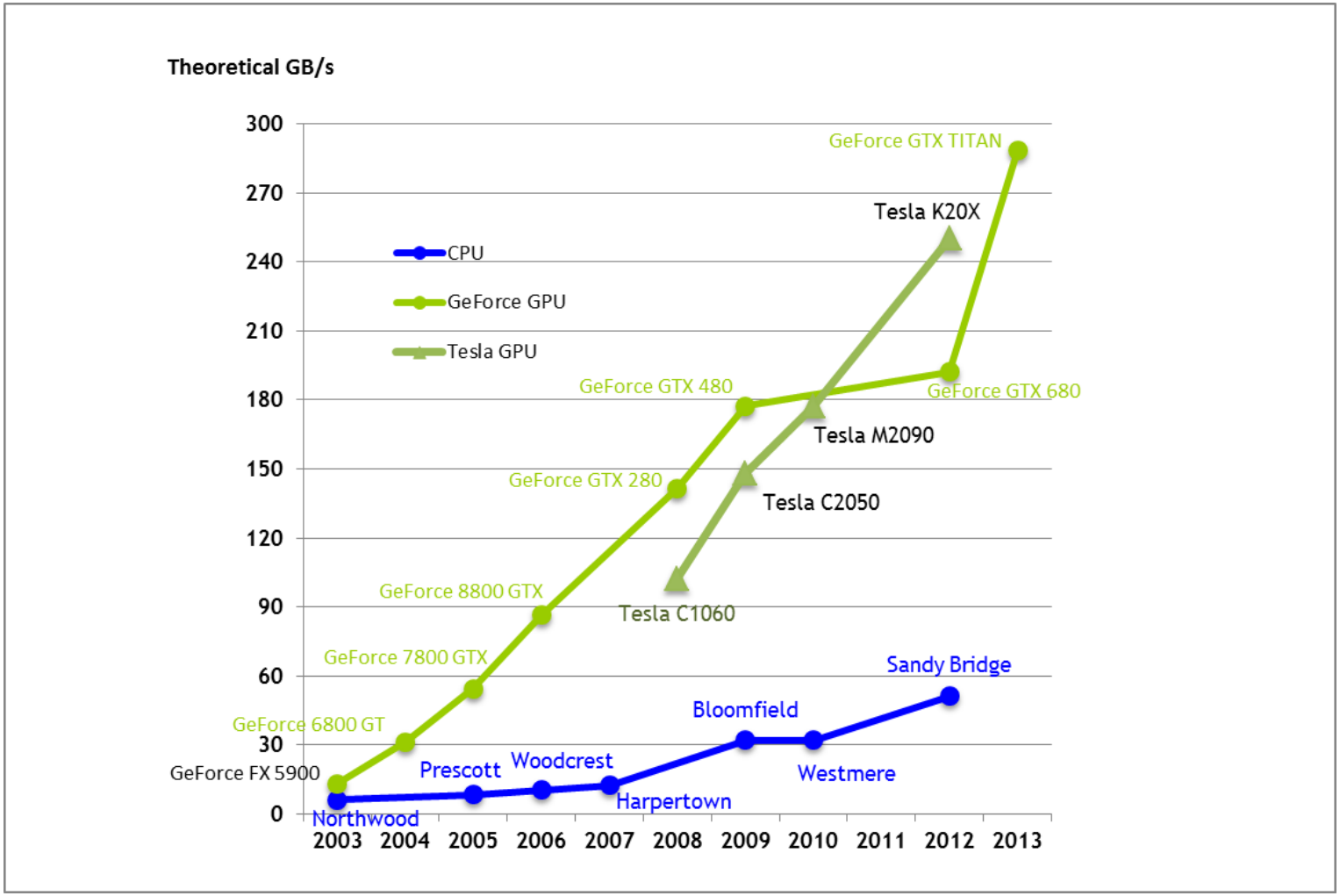
Motivace – grafické výpočty

- datově paralelní
  - provádíme stejné výpočty pro různé vertexy, pixely, ...
  - datově paralelní procesory mají vyšší koncentraci ALU, přináší tak vyšší teoretický výkon
- předdefinované funkce
- programovatelné funkce
  - specifické grafické efekty
  - GPU se stávají stále více programovatelnými
  - díky tomu lze zpracovávat i jiné, než grafické úlohy

Motivace – výkon



Motivace – výkon



Motivace – uplatnění

Využití GPU pro obecné výpočty je dynamicky se rozvíjející oblast s širokou škálou aplikací

Motivace – uplatnění

Využití GPU pro obecné výpočty je dynamicky se rozvíjející oblast s širokou škálou aplikací

- vysoce náročné vědecké výpočty
  - výpočetní chemie
  - fyzikální simulace
  - zpracování obrazů
  - a mnohé další...

Motivace – uplatnění

Využití GPU pro obecné výpočty je dynamicky se rozvíjející oblast s širokou škálou aplikací

- vysoce náročné vědecké výpočty
  - výpočetní chemie
  - fyzikální simulace
  - zpracování obrazů
  - a mnohé další...
- výpočetně náročné aplikace pro domácí uživatele
  - kódování a dekódování multimediálních dat
  - herní fyzika
  - úprava obrázků, 3D rendering
  - atd...

Motivace – obsah přednášky

Pro plné porozumění architektuře, paralelizaci a optimalizaci pro GPU jedna přednáška nestačí

- detailněji v PV197

Přednáška poskytuje základní přehled

- jak vypadá architektura GPU a v čem se liší od klasických CPU
- jaké druhy algoritmů běží na GPU efektivně
- jak se GPU programují

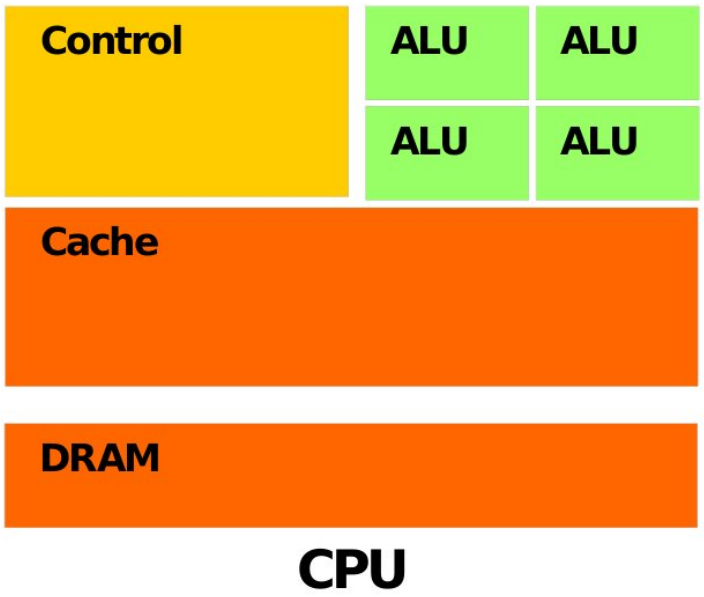
Architektura GPU

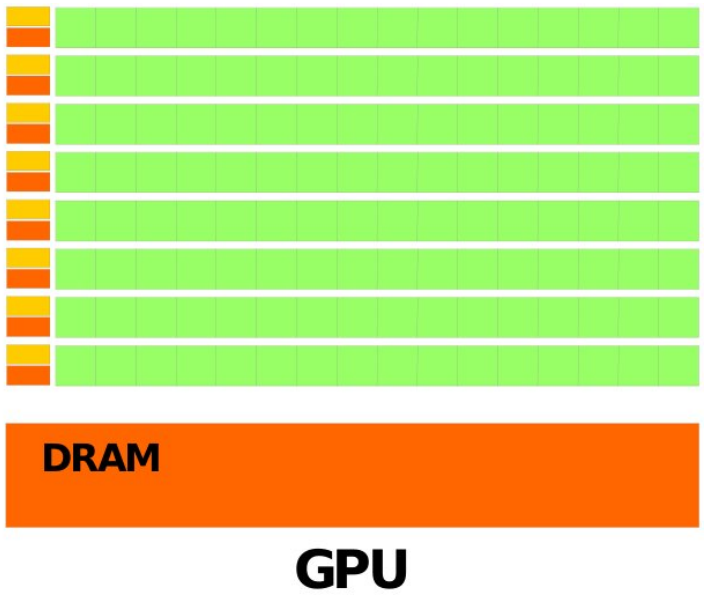
CPU vs. GPU

- jednotky jader vs. desítky multiprocesorů
- out of order vs. in order
- MIMD, SIMD pro krátké vektory vs. SIMT pro dlouhé vektory
- velká cache vs. malá cache, často pouze pro čtení

GPU používá více tranzistorů pro výpočetní jednotky než pro cache a řízení běhu => vyšší výkon, méně univerzální

Architektura GPU





Architektura GPU

V rámci systému:

- koprocesor s dedikovanou pamětí
- asynchronní běh instrukcí
- připojen k systému přes PCI-E

Procesor G80

G80

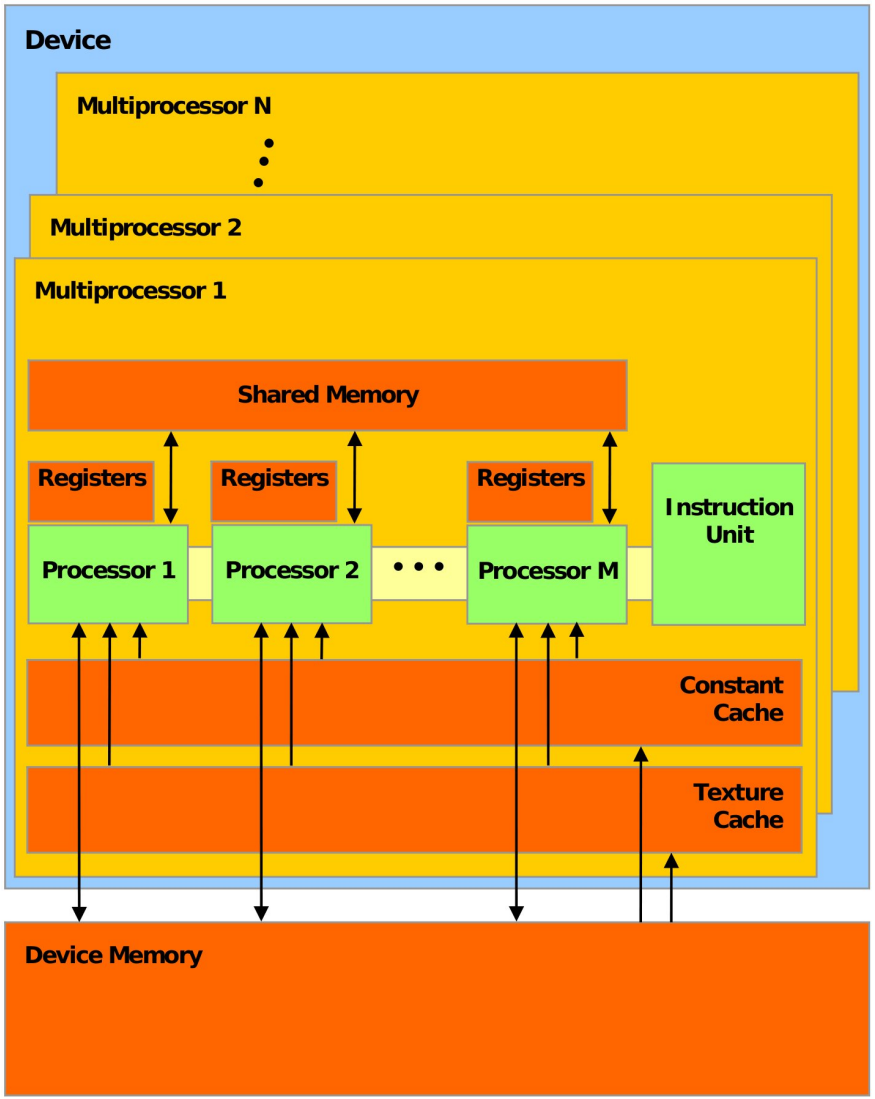
- první CUDA procesor
- obsahuje 16 multiprocesorů
- multiprocesor
  - 8 skalárních procesorů
  - 2 jednotky pro speciální funkce
  - až 768 threadů
    - HW přepínání a plánování threadů
  - thready organizovány po 32 do warpů
    - SIMT
  - nativní synchronizace v rámci multiprocesoru

Paměťový model G80

Paměťový model

- 8192 registrů sdílených mezi všemi thready multiprocesoru
- 16 KB sdílené paměti
  - lokální v rámci multiprocesoru
  - stejně rychlá jako registry (za dodržení určitých podmínek)
- paměť konstant
  - cacheovaná, pouze pro čtení
- paměť pro textury
  - cacheovaná, 2D prostorová lokalita, pouze pro čtení
- globální paměť
  - pro čtení i zápis, necacheovaná
- přenosy mezi systémovou a grafickou pamětí přes PCI-E

Procesor G80



Další vývoj

Procesory odvozené od G80

- double-precision výpočty
- relaxovány pravidla pro efektivní přístup ke globální paměti
- lepší možnosti synchronizace (atomické operace)

Fermi

- vyšší paralelizace na úrovni multiprocessoru
- konfigurovatelná L1 a sdílená L2 cache
- plochý adresní prostor
- lepší přesnost v plovoucí řádové čárce
- paralelní běh kernelů

Kepler

- vyšší paralelizace na úrovni multiprocessoru
- omezení cacheovaní
- dynamický paralelismus
- efektivní komunikace v rámci warpu

Srovnání teoretické rychlosti GPU a CPU

Teoretická maxima

- GPU má cca 10× rychlejší aritmetiku
- GPU má cca 5× vyšší propustnost paměti
- zajímavé pro mnohé problémy (budu čekat na výsledky simulace měsíc nebo rok? pojede mi video na 3 nebo 30 fps?)

Některé publikace ukazují 100× i 1000× zrychlení

- v pořádku, je-li interpretováno jako zrychlení oproti produkčnímu SW (ten nemusí být perfektně optimalizovaný)
- interpretováno jako srovnání CPU a GPU zpravidla nesmysl

Srovnáváme-li přínos GPU oproti CPU, musíme uvažovat efektivní implementaci pro obě platformy.

Srovnání teoretické rychlosti GPU a CPU

V praxi máme však často sériový CPU kód

- běh v jednom vlákně znamená až 16× zpomalení (16-jádrové CPU)
- absence vektorizace znamená až 4× zpomalení (32-bit operace u SSE instrukcí), 8× u AVX instrukcí

Oproti sériové implementaci tedy můžeme kód paralelizací a vektorizací zrychlit

- 32× pro čtyřjádrové CPU s AVX nebo osmijádrové s SSE

GPU akcelerací pak

- cca 300×

Vektorizace a paralelizace pro CPU je však programátorskou náročností **srovnatelná** s GPU akcelerací.

Teoretické vs. dosažitelné zrychlení

Výkonový odstup GPU může být vyšší

- jednotky pro speciální funkce, operace na texturách
- SIMT pružnější než SIMD
- neduhy SMP (omezení škálování propustnosti paměti, „vytloukání řádků cache")

Stejně jako nižší

- nedostatek paralelismu
- příliš vysoký overhead
- nevhodný algoritmus pro GPU architekturu

Dále se podíváme, jak rozlišit, jestli je nebo naopak není váš algoritmus vhodný pro GPU.

Paralelizace

Sčítání vektorů

- jednoduché datově-paralelní vyjádření
- žádná synchronizace
- potřebujeme velké vektory

Game of Life

- co chceme paralelizovat?

Game of Life – zjištění nového stavu hry

- pro větší herní plochy dostatek paralelismu
- jednoduchá synchronizace

Game of Life – zjištění stavu buňky po n krocích

- inherentně sekvenční? (Game of Life je P-complete, $P \stackrel{?}{=} NC$)
- neznáme paralelní algoritmus

Paralelizace

Redukce

- na první pohled může vypadat sekvenčně
- ve skutečnosti realizovatelná v $\log n$ krocích
- často je třeba nedržet se sekvenční verze a zamyslet se nad paralelizací problému (ne sekvenčního algoritmu)

Paralelizace

Problém nalezení povodňové mapy

- máme výškovou mapu terénu, přítok vody, a chceme zjistit, jaká oblast se zatopí
- sekvenčnost dána rozléváním vody
- je snadné najít úlohově-paralelní algoritmus, datově-paralelní už tak ne
  - periodická aktualizace stavu každého bodu mapy
  - aktualizace omezená jen na hranice vodní plochy (šetří procesory)
  - rozlévání vody zametací přímkou (vhodnější pro GPU, jednodušší synchronizace)
  - hledání souvislých oblastí a jejich spojování (odstraňuje sekvenčnost rozlévání)
  - vždy práce navíc oproti sekvenční/úlohově-paralelní verzi
- úkol PV197 na podzim 2010, výkon odevzdaných implementací se lišil o 4 řády (!)

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, nepravidelné datové struktury
  - nutnost zamyslet se nad výrazně odlišným algoritmem pro daný problém

Divergence přístupu do paměti

Divergence přístupu do paměti

- není-li do paměti přistupováno po souvislých blocích v rámci warpu, snižuje se její propustnost
- často složitě 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

Latence GPU

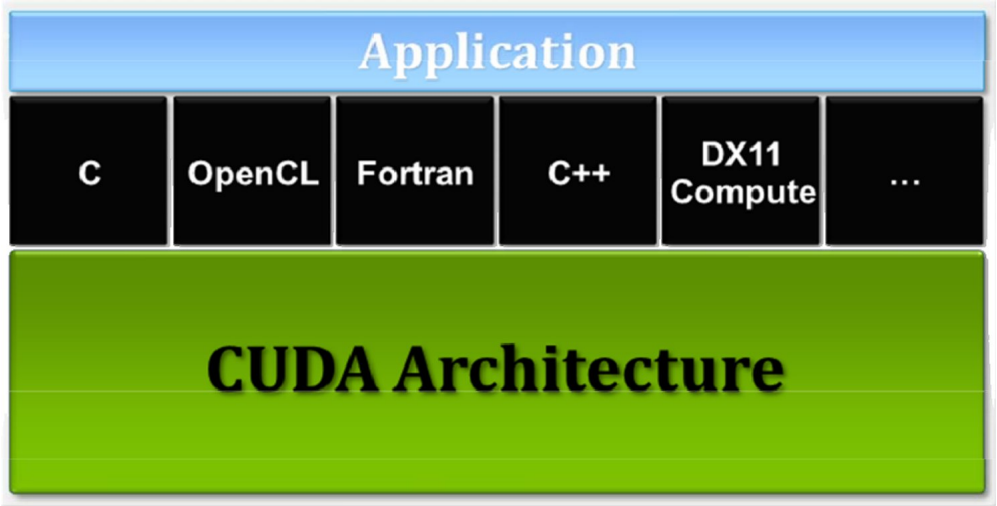
GPU je dnes často propojena se zbytkem systému přes PCI-E

- kopírování vstupů/výstupů je relativně pomalé
- akcelerovaný algoritmus musí provádět dostatečné množství aritmetiky na přenášená data
  - násobení matic je vhodné ($\mathcal{O}(n^3)$ operací na $\mathcal{O}(n^2)$ dat)
  - sčítání vhodné není ($\mathcal{O}(n^2)$ operací na $\mathcal{O}(n^2)$ dat), může být však součástí většího problému

CUDA

CUDA (Compute Unified Device Architecture)

- architektura pro paralelní výpočty vyvinutá firmou NVIDIA
- poskytuje nový programovací model, který umožňuje efektivní implementaci obecných výpočtů na GPU
- je možné použít ji s více programovacími jazyky



C for CUDA

C for CUDA přináší rozšíření jazyka C pro paralelní výpočty

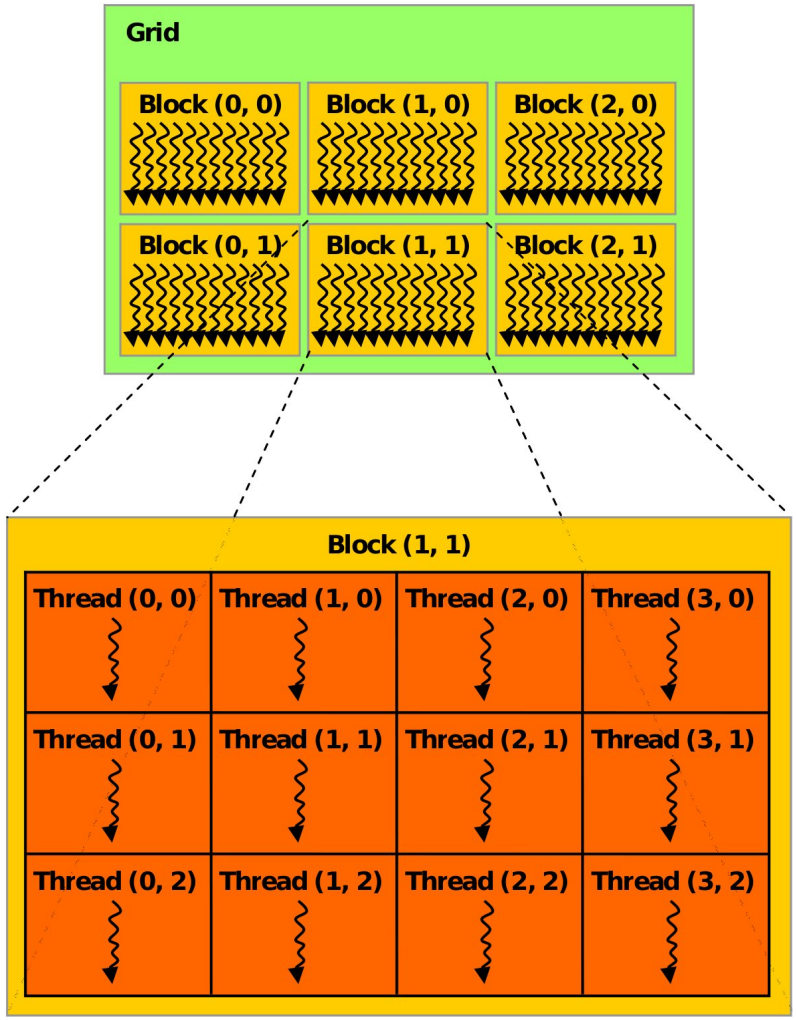
- explicitně oddělen host (CPU) a device (GPU) kód
- hierarchie vláken
- hierarchie pamětí
- synchronizační mechanismy
- API

Hierarchie vláken

Hierarchie vláken

- vlákna jsou organizována do bloků
- bloky tvoří mřížku
- problém je dekomponován na podproblémy, které mohou být prováděny nezávisle paralelně (bloky)
- jednotlivé podproblémy jsou rozděleny do malých částí, které mohou být prováděny kooperativně paralelně (thready)
- dobře škáluje

Hierarchie vláken

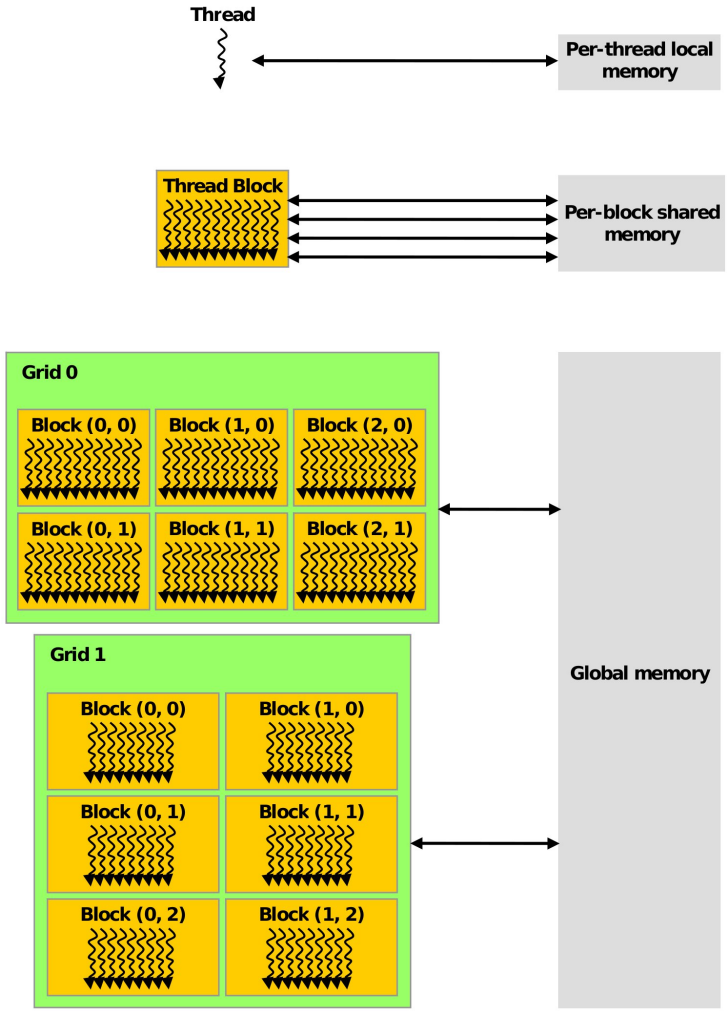


Hierarchie pamětí

Více druhů pamětí

- rozdílná viditelnost
- rozdílný čas života
- rozdílné rychlosti a chování
- přináší dobrou škálovatelnost

Hierarchie pamětí



Příklad – součet vektorů

Chceme sečíst vektory a a b a výsledek uložit do vektoru c.

Příklad – součet vektorů

Chceme sečíst vektory a a b a výsledek uložit do vektoru c.
Je třeba najít v problému paralelismus.

Příklad – součet vektorů

Chceme sečíst vektory *a* a *b* a výsledek uložit do vektoru *c*.
Je třeba najít v problému paralelismus.
Sériový součet vektorů:

```
for (int i = 0; i < N; i++)
  c[i] = a[i] + b[i];
```

Příklad – součet vektorů

Chceme sečíst vektory *a* a *b* a výsledek uložit do vektoru *c*.
Je třeba najít v problému paralelismus.
Sériový součet vektorů:

```
for (int i = 0; i < N; i++)
  c[i] = a[i] + b[i];
```

Jednotlivé iterace cyklu jsou na sobě nezávislé – lze je paralelizovat, škáluje s velikostí vektoru.

Příklad – součet vektorů

Chceme sečíst vektory *a* a *b* a výsledek uložit do vektoru *c*.
Je třeba najít v problému paralelismus.
Sériový součet vektorů:

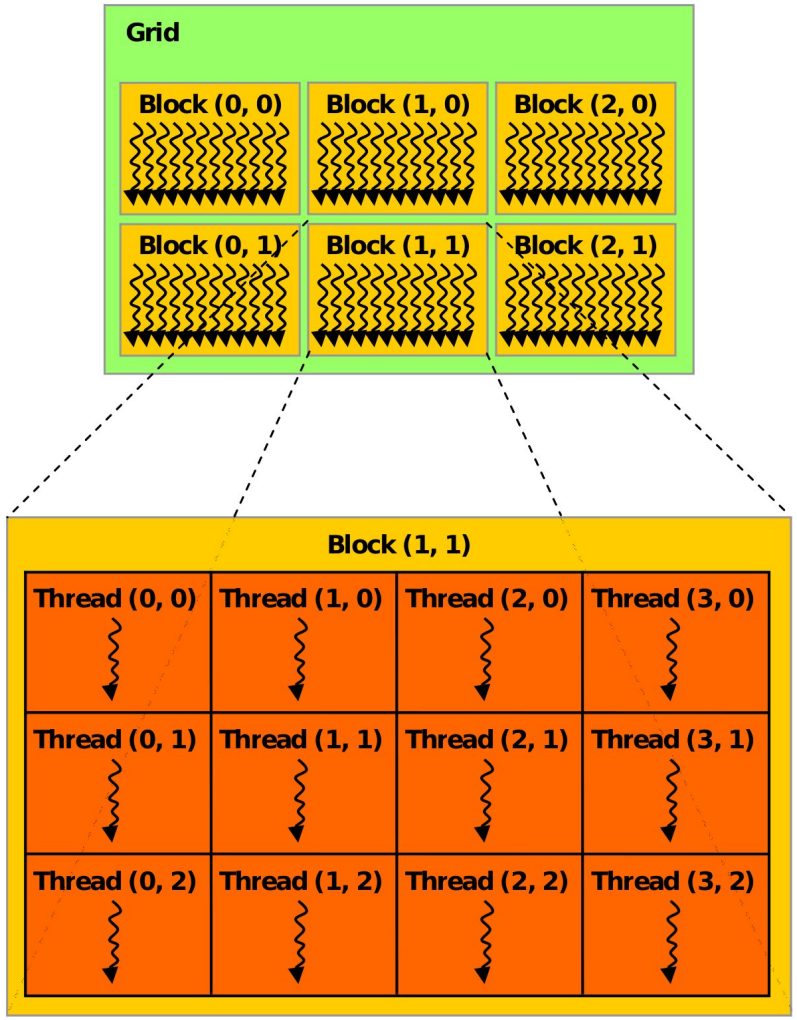
```
for (int i = 0; i < N; i++)
  c[i] = a[i] + b[i];
```

Jednotlivé iterace cyklu jsou na sobě nezávislé – lze je paralelizovat, škáluje s velikostí vektoru.
i-tý thread sečte i-té složky vektorů:

```
c[i] = a[i] + b[i];
```

Jak zjistíme, kolikátý jsme thread?

Hierarchie vláken



Identifikace vlákna a bloku

C for CUDA obsahuje zabudované proměnné:

- **threadIdx.**{**x, y, z**} udává pozici threadu v rámci bloku
- **blockDim.**{**x, y, z**} udává velikost bloku
- **blockIdx.**{**x, y, z**} udává pozici bloku v rámci mřížky (z je vždy 1)
- **gridDim.**{**x, y, z**} udává velikost mřížky (z je vždy 1)

Příklad – součet vektorů

Vypočítáme tedy globální pozici threadu (mřížka i bloky jsou jednorozměrné):

Příklad – součet vektorů

Vypočítáme tedy globální pozici threadu (mřížka i bloky jsou jednorozměrné):

```
int i = blockIdx.x*blockDim.x + threadIdx.x;
```

Příklad – součet vektorů

Vypočítáme tedy globální pozici threadu (mřížka i bloky jsou jednorozměrné):

```
int i = blockIdx.x*blockDim.x + threadIdx.x;
```

Celá funkce pro paralelní součet vektorů:

```
__global__ void addvec(float *a, float *b, float *c){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}
```

Příklad – součet vektorů

Vypočítáme tedy globální pozici threadu (mřížka i bloky jsou jednorozměrné):

```
int i = blockIdx.x*blockDim.x + threadIdx.x;
```

Celá funkce pro paralelní součet vektorů:

```
__global__ void addvec(float *a, float *b, float *c){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}
```

Funkce definuje tzv. kernel, při volání určíme, kolik threadů a v jakém uspořádání bude spuštěno.

Kvantifikátory typů funkcí

Syntaxe C je rozšířena o kvantifikátory, určující, kde se bude kód provádět a odkud půjde volat:

- __**device**__ funkce je spouštěna na device (GPU), lze volat jen z device kódu
- __**global**__ funkce je spouštěna na device, lze volat jen z host (CPU) kódu
- __**host**__ funkce je spouštěna na host, lze ji volat jen z host
- kvantifikátory __host__ a __device__ lze kombinovat, funkce je pak kompilována pro obojí

Ke kompletnímu výpočtu je třeba:

Ke kompletnímu výpočtu je třeba:

- alokovat paměť pro vektory, naplnit je daty

Ke kompletnímu výpočtu je třeba:

- alokovat paměť pro vektory, naplnit je daty
- **alokovat paměť na GPU**

Ke kompletnímu výpočtu je třeba:

- alokovat paměť pro vektory, naplnit je daty
- **alokovat paměť na GPU**
- **zkopírovat vektory *a* a *b* na GPU**

Ke kompletnímu výpočtu je třeba:

- alokovat paměť pro vektory, naplnit je daty
- **alokovat paměť na GPU**
- **zkopírovat vektory *a* a *b* na GPU**
- **spočítat vektorový součet na GPU**

Ke kompletnímu výpočtu je třeba:

- alokovat paměť pro vektory, naplnit je daty
- **alokovat paměť na GPU**
- **zkopírovat vektory** a **a** b **na GPU**
- **spočítat vektorový součet na GPU**
- **uložit výsledek z GPU paměti do** c

Ke kompletnímu výpočtu je třeba:

- alokovat paměť pro vektory, naplnit je daty
- **alokovat paměť na GPU**
- **zkopírovat vektory *a* a *b* na GPU**
- **spočítat vektorový součet na GPU**
- **uložit výsledek z GPU paměti do *c***
- použít výsledek v *c* :-)

Příklad – součet vektorů

CPU kód naplní *a* a *b*, vypíše *c*:

```
#include <stdio.h>
#define N 64
int main(){
  float a[N], b[N], c[N];
  for (int i = 0; i < N; i++)
    a[i] = b[i] = i;

// zde bude kód provádějící výpočet na GPU

  for (int i = 0; i < N; i++)
    printf("%f, ", c[i]);
  return 0;
}
```

Správa GPU paměti

Paměť je třeba dynamicky alokovat.

```
cudaMalloc(void** devPtr, size_t count);
```

Alokuje paměť velikosti *count*, nastaví na ni ukazatel *devPtr*. Uvolnění paměti:

```
cudaFree(void* devPtr);
```

Kopírování paměti:

```
cudaMemcpy(void* dst, const void* src, size_t count,
  enum cudaMemcpyKind kind);
```

Kopíruje *count* byte z *src* do *dst*, *kind* určuje, o jaký směr kopírování se jedná (např. *cudaMemcpyHostToDevice*, nebo *cudaMemcpyDeviceToHost*).

Příklad – součet vektorů

Alokujeme paměť a přeneseme data:

```
float *d_a, *d_b, *d_c;
cudaMalloc((void**)&d_a, N*sizeof(*d_a));
cudaMalloc((void**)&d_b, N*sizeof(*d_b));
cudaMalloc((void**)&d_c, N*sizeof(*d_c));

cudaMemcpy(d_a, a, N*sizeof(*d_a), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, N*sizeof(*d_b), cudaMemcpyHostToDevice);

// zde bude spuštěn kernel

cudaMemcpy(c, d_c, N*sizeof(*c), cudaMemcpyDeviceToHost);

cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
```

Příklad – součet vektorů

Spuštění kernelu:

- kernel voláme jako funkci, mezi její jméno a argumenty vkládáme do trojitých špičatých závorek velikost mřížky a bloku
- potřebujeme znát velikost bloků a jejich počet
- použijeme 1D blok i mřížku, blok bude pevné velikosti
- velikost mřížky vypočteme tak, aby byl vyřešen celý problém násobení vektorů

Pro vektory velikosti dělitelné 32:

```
#define BLOCK 32
addvec<<<N/BLOCK, BLOCK>>>(d_a, d_b, d_c);
```

Jak řešit problém pro obecnou velikost vektoru?

Příklad – součet vektorů

Upravíme kód kernelu:

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

A zavoláme kernel s dostatečným počtem vláken:

```
addvec<<<N/BLOCK + 1, BLOCK>>>(d_a, d_b, d_c, N);
```

Příklad – spuštění

Nyní už zbývá jen kompilace :-).

```
nvcc -I/usr/local/cuda/include -L/usr/local/cuda/lib -lcudart \
  -o vecadd vecadd.cu
```

Kde s CUDA pracovat?

- vlastní stroj: stáhněte a nainstalujte CUDA toolkit a SDK z developer.nvidia.com
- windowsí stanice v učebnách (titan)
- ke vzdálené práci s high-end GPU: na přání

Paměti lokální v rámci threadu

Registry

- nejrychlejší paměť, přímo využitelná v instrukcích
- lokální proměnné v kernelu i proměnné nutné pro mezivýsledky jsou automaticky v registrech
  - pokud je dostatek registrů
  - pokud dokáže kompilátor určit statickou indexaci polí
- mají životnost threadu (warpu)

Paměti lokální v rámci threadu

Registry

- nejrychlejší paměť, přímo využitelná v instrukcích
- lokální proměnné v kernelu i proměnné nutné pro mezivýsledky jsou automaticky v registrech
  - pokud je dostatek registrů
  - pokud dokáže kompilátor určit statickou indexaci polí
- mají životnost threadu (warpu)

Lokální paměť

- co se nevejde do registrů, jde do lokální paměti
- ta je fyzicky uložena v DRAM, je tudíž pomalá a má dlouhou latenci
- má životnost threadu (warpu)

Paměť lokální v rámci bloku

Sdílená paměť

- organizována do bank umožňujících paralelní přístup
- u rodiny G80 rychlá jako registry
  - nedojde-li ke konfliktům paměťových bank
  - instrukce umí využít jen jeden operand ve sdílené paměti (jinak je třeba explicitní load/store)
- u novějších GPU ve srovnání s registry pomalejší
- v C for CUDA deklarujeme pomocí *__shared__*
- má životnost bloku

Paměť lokální pro GPU

Globální paměť

- řádově nižší přenosová rychlost než u sdílené paměti
- latence ve stovkách GPU cyklů
- pro dosažení optimálního výkonu je třeba paměť adresovat sdruženě
- má životnost aplikace
- u Fermi L1 cache (128 byte na řádek) a L2 cache (32 byte na řádek)

Lze dynamicky alokovat pomocí *cudaMalloc*, či staticky pomocí deklarace *__device__*

Ostatní paměti

- paměť konstant
- texturová paměť
- systémová paměť

Synchronizace v rámci bloku

- nativní bariérová synchronizace
  - musí do ní vstoupit všechny thready (pozor na podmínky!)
  - pouze jedna instrukce, velmi rychlá, pokud neredukuje paralelismus
  - v C for CUDA volání **__syncthreads()**
  - Fermi rozšíření: count, and, or
- atomické operace nad sdílenou pamětí

Synchronizace bloků

Mezi bloky

- globální paměť viditelná pro všechny bloky
- slabá nativní podpora synchronizace
  - žádná globální bariéra uvnitř kernelu
  - globální bariéru lze implementovat voláním kernelu (jiné řešení dosti trikové)
  - slabé možnosti globální synchronizace znesnadňují programování, ale umožňují velmi dobrou škálovatelnost hardware
- u novějších GPU *atomické operace* nad globální pamětí

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 sdružené 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...