Compute Unified Device Architecture

Brno, 2013

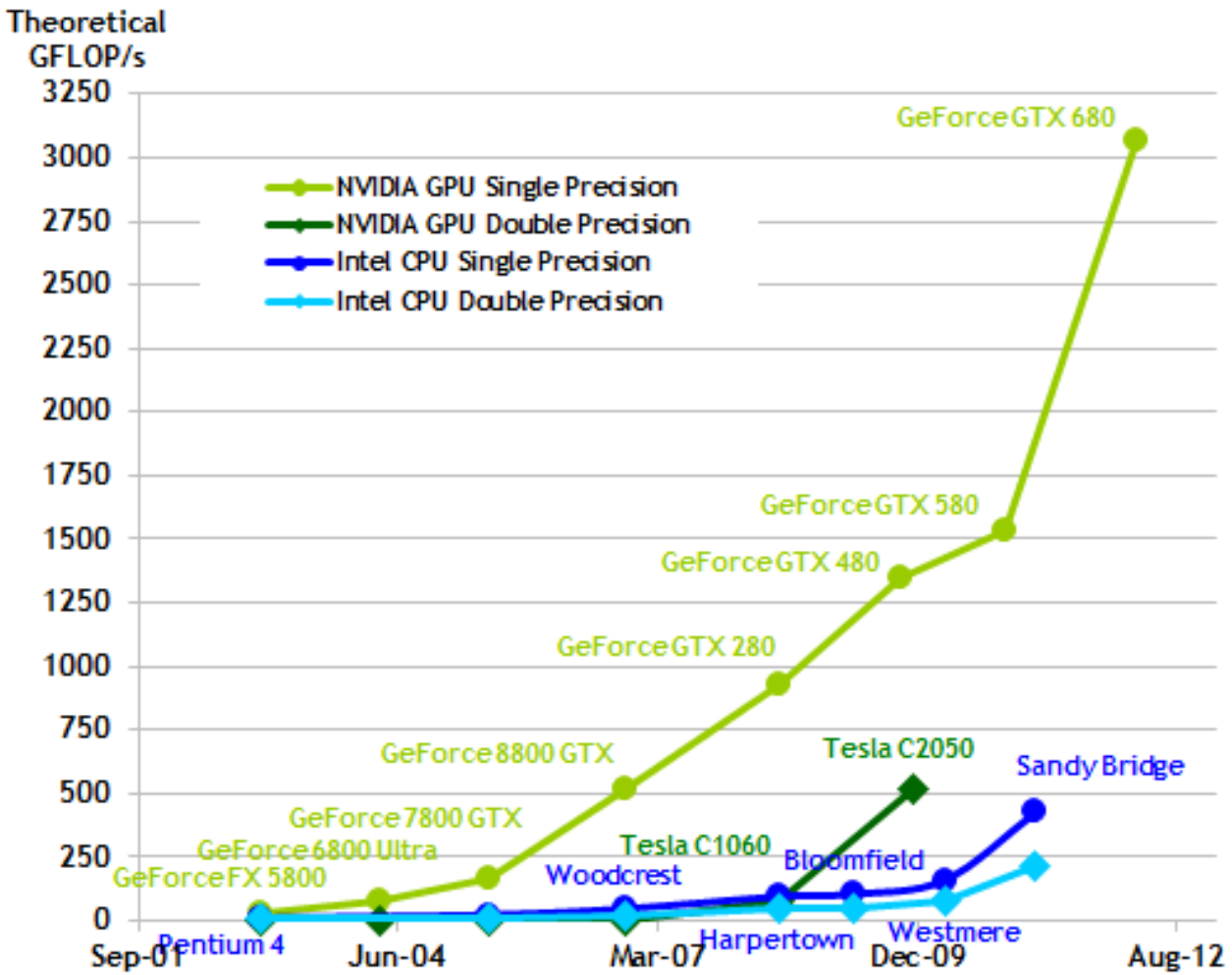
Vysvětlení pojmů

Pojem/zkratka	význam
CPU	procesor
GPU	grafický procesor
MS	multiprocesor - procesory, ze kterých se skládá GPU
kernel	základní funkce provádějící se na GPU
VRAM/DRAM	základní paměť grafické karty
FLOPs	počet operací s desetinnou čárkou za sekundu
host	čip, na kterém je primárně spuštěna aplikace (CPU)
device	zařízení, na kterém běží CUDA (graf. karta)

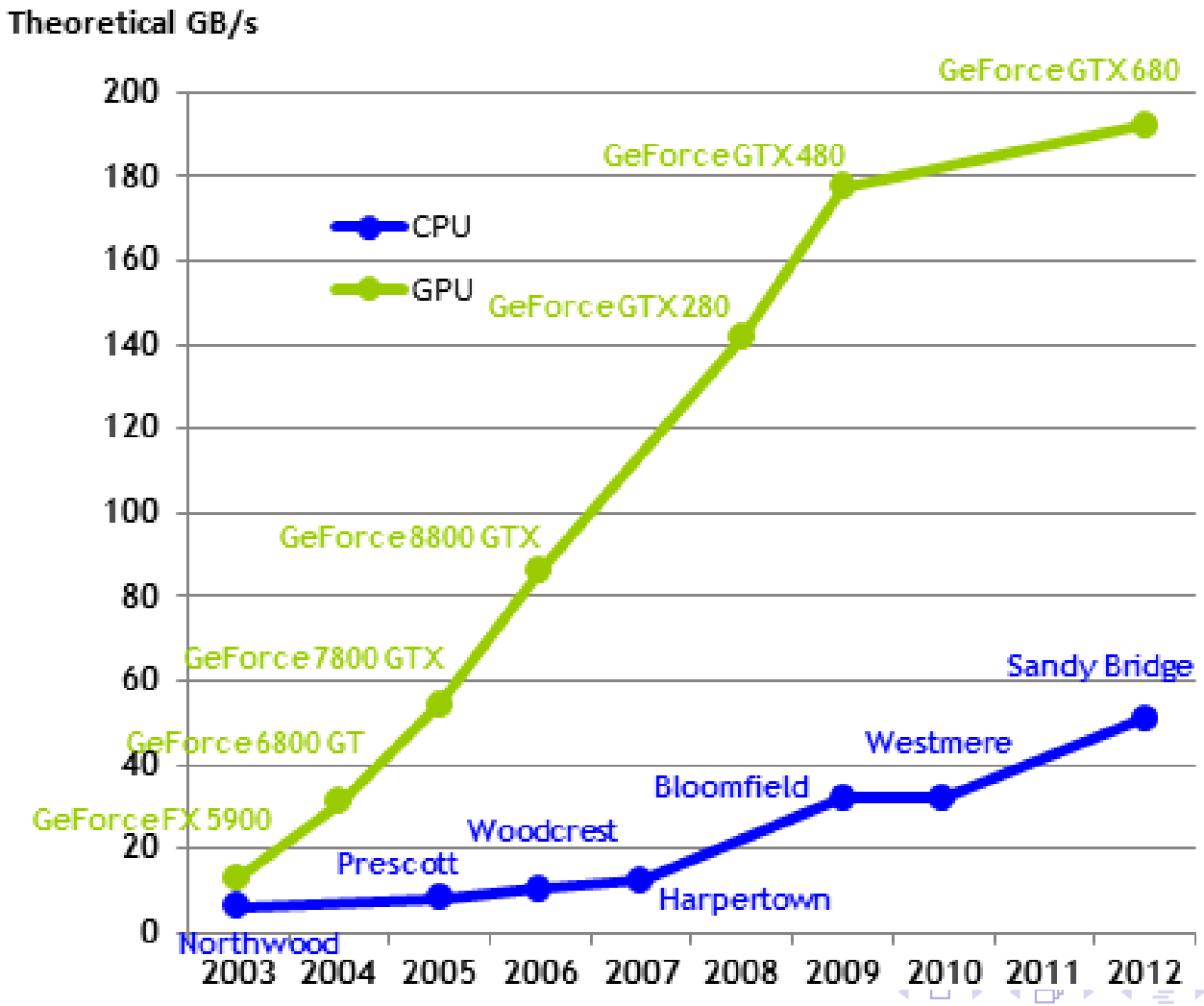
Motivace

- zkratka CUDA
- čipy společnosti nVidia
- technologie od roku 2006; nejstarší technologie pro paralelní výpočty na GPU
- dnes je podporována téměř v každé grafické kartě s čipem společnosti nVidia
- multiplatformní technologie (Linux, Mac OS, Windows)
- podpora jazyků C/Cpp, FORTRAN, Python, Java, Open CL, DirectX Compute

Motivace
Proč grafické karty?



Motivace
Proč grafické karty?



Motivace
Proč grafické karty?

- renderování vektorové grafiky v reálném čase $\Rightarrow$ vysoké nároky na výkon
- architektura GPU se podstatně liší od CPU
- můžeme této architektury využít i pro obecné výpočty
- Některé typy výpočtů nelze na GPU provádět efektivně
- využití:
  - hromadné zpracování dat
  - paralelní výpočty
  - bruteforcing :)

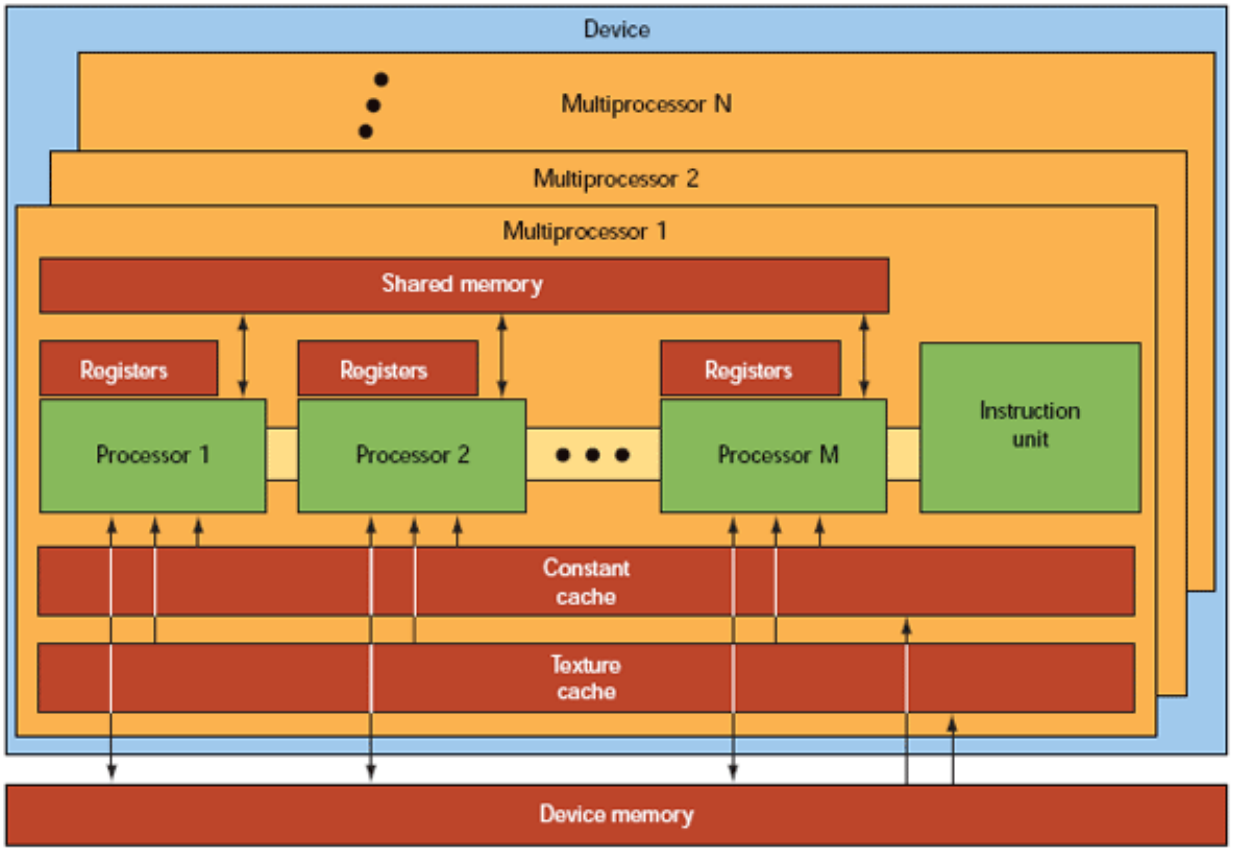
Instalace

- http://docs.nvidia.com/cuda/
- již je 5. verze CUDA
- instalační balík obsahuje vše potřebné (nižší verze obsahovaly více různých souborů)
- je nutné správně nalinkovat CUDA knihovny

Hardwarová Architektura

- GPU je rozděleno na tzv. multiprocesory (dnes Streaming Multiprocessor/CUDA Cores)
- každý MS je samostatná výpočetní jednotka
  - 8 - 192 jader (liší se podle stáří kary)
  - 32bit registry (8K - 64K)
  - sdílenou paměť (16KB - 48KB)
  - konstantní paměť (64KB)
  - texturovací paměť
- GPU komunikuje s CPU pomocí VRAM

Hardwarová Architektura



Základní schéma multiprocesoru

Základní Programové Pojmy

- kernel - funkce definována klíčovým slovem __global__
  - tato funkce je spuštěna *N*-krát
  - zdrojový kód již zpracovává *N* vláken
- každé vlákno má svůj index (threadIdx)
  - tříprvkový vektor[x, y, z] - proměnná typu dim3 (může být nahrazena celočíselnou proměnnou)

Volání Kernelu

```
__global__ void Soucet(int *A, int *B, int *C)
{
    int idx = threadIdx.x;  \\ index vlakna
    C[idx] = A[idx] + B[idx];
}

int main()
{
    ...
    \\ soucet N prvku matic A a B
    Soucet<<<1; N>>>(A_device, B_device, C_device);
    ...
}
```

Volání Kernelu

- syntaxe: <<<1, N>>> - „Execution Configuration“
  - první argument, počet bloků v mřížce
  - druhý argument, počet vláken v bloku
  - obě dvě proměnné nabývají tří hodnot - pro každou dimenzi jinou velikost
- při volání kernelu určujeme topologii kernelu
- implicitně je každá hodnota souřadnic bloků i mřížky nastavena na hodnotu 1
- <<<200, 256>>> - 200 bloků, každý o 256 vláknech

Volání Kernelu

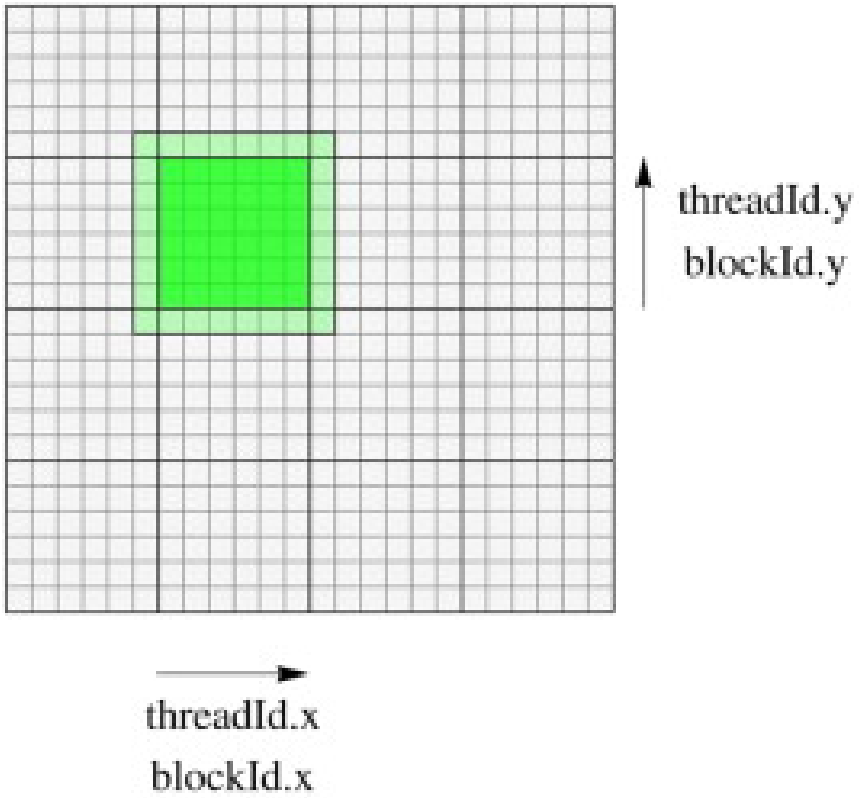
```
__global__ void Soucet(int A[N][M], int B[N][M], int C[N][M])
{
    int idx = threadIdx.x;
    int idy = threadIdx.y;
    C[idx][idy] = A[idx][idy] + B[idx][idy];
}

int main()
{
    ...
    dim3 vlakna(N, M);
    Soucet<<<1; vlakna>>>(A_device, B_device, C_device);
    ...
}
```

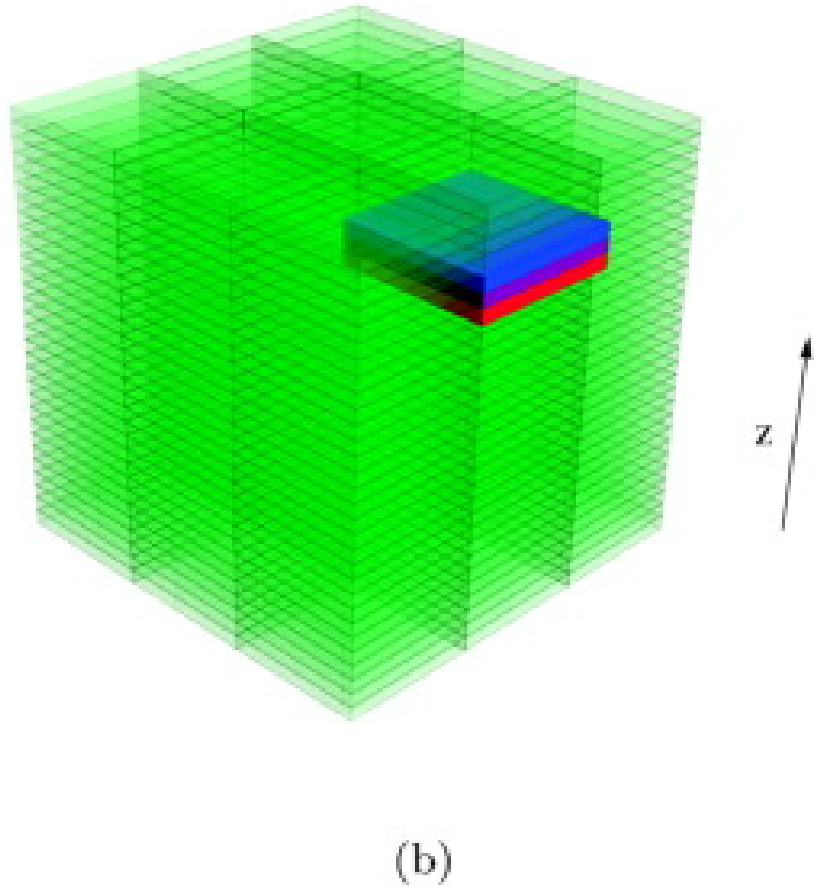
Užití Indexu Vláken

- index vlákna platí pouze v rámci bloku
- každý blok má vlastní ID v mřížce blockIdx[x, y, z]
- k vypočítání indexu potřebujeme velikost bloku blockDim[x, y, z]
- pokud chceme spustit na jedno pole více bloků s vlákny, každé vlákno si musí spočítat index v poli, se kterým bude pracovat
- idx = blockIdx.x * blockDim.x + threadIdx.x;

Volání Kernelu

```
__global__ void Soucet(int A[N][M], int B[N][M], int C[N][M])
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;
    C[idx][idy] = A[idx][idy] + B[idx][idy];
}

int main()
{
    ...
    dim3 vlaknaNaBlok(16, 16);          //256
    dim3 pocetBloku(N / vlaknaNaBlok.x, M / vlaknaNaBlok.y);
    Soucet<<<pocetBloku, vlaknaNaBlok>>>(A_device, B_device,
     C_device);
    ...
}
```


threadId.y
blockId.y
threadId.x
blockId.x
(a)


z
(b)

Základy Práce s Pamětí

- rozlišujeme RAM a VRAM
- rozšířená syntaxe
- standardní alokace paměti GPU
  - cudaMalloc((void**) &p_device, size);
  - p_device musí být pointer
- data je nutné na GPU překopírovat
  - cudaMemcpy(p_device, p_host, size, cudaMemcpyHostToDevice);
  - cudaMemcpy(p_host, p_device, size, cudaMemcpyDeviceToHost);
- uvolnění paměti
  - cudaFree(p_device);
- hodnota - cudaError_t

Hello CUDA!

- prográmek :)

Bloky, Mřížky, Vlákna

- různá GPU mají různý počet MP
- není nutno rekompilovat program kvůli různým GPU
- sdílená paměť, synchronizace vláken v rámci jednoho bloku

Paměti Podrobněji

Typ paměti	Přístup	Umístění	Operace	Keš.
Registry	Jedno vlákno	Čip	čtení/zápis	Ne
Lokální	Jedno vlákno	DRAM	čtení/zápis	Ne
Sdílená	Vlákna v rámci bloku	Čip	čtení/zápis	-
Globální	Všechna vlákna + CPU	DRAM	čtení/zápis	Ne
Texturovací	Všechna vlákna + CPU	DRAM	čtení/zápis	Ano
Konstantní	Všechna vlákna + CPU	DRAM	GPU jen čtení	Ano

Sdílená Paměť

- globální paměť je příliš pomalá, sdílená paměť je mnohonásobně rychlejší
- vlákna v jednom bloku mohou spolupracovat, jsou propojena sdílenou pamětí
- žádná jiná vlákna nemají do sdílené paměti přístup
- po zániku bloku zaniká i alokovaná sdílená paměť
- užití: pole s častějším využitím v rámci jednoho bloku
- KONFLIKTY BANK - načítání hodnot, které ještě neexistují

Alokace Sdílené Paměti

- v Execution Configuration - třetí argument

```
__global__ kernel()
{
    extern __shared__ int SHARED_MEMORY[];
}

int main()
{
    ...
    kernel<<<bloky, vlakna, velikost_sdilene_pameti_v_bytech>>>();
    ...
}
```

Alokace Sdílené Paměti

- v kernelu

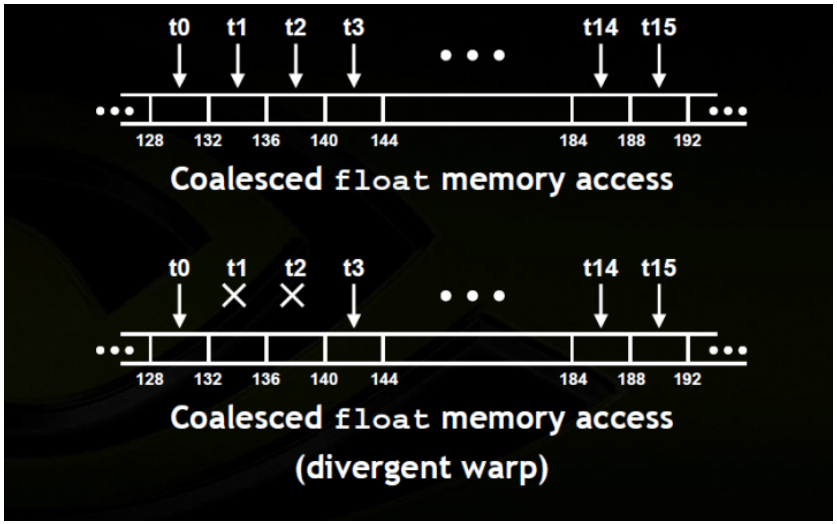
```
__global__ kernel()
{
    __shared__ int SHARED_MEMORY[velikost_v_bytech];
}

int main()
{
    ...
    kernel<<<bloky, vlakna>>>();
    ...
}
```

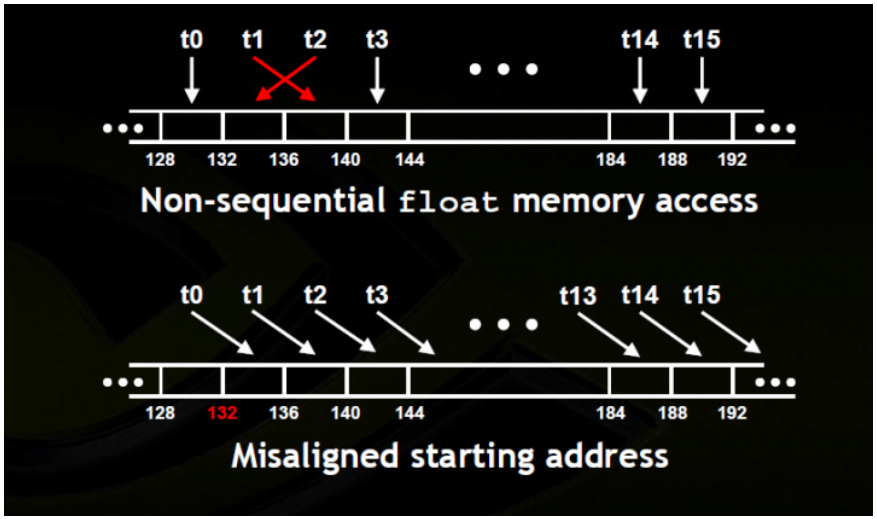
Sdružený Přístup Do Paměti
Coalesced Memory Access

- globální paměť je veliká, ale pomalá a nekešovaná
- je zde technika pro využívání paralelních činností vláken
- každá instrukce je zpracovávána ve warpech; 32 vláken
- vlákna ve warpu nesmí divergovat! $\Rightarrow$ zvýšení rychlosti
- dělení na Half-Warpy (16 vláken)
- pro sdružený přístup platí tři pravidla:
  - velikost proměnných musí být 4, 8, 16 bytů
  - vlákna k proměnným přistupují sekvenčně (N-té vlákno přistupuje k N-tému prvku)
  - proměnné musí být ve stejném paměťovém segmentu, adresa prvního prvku je zarovnána k 16ti násobku velikosti proměnné

Sdružený Přístup Do Paměti
Colesced Memory Access

- výrazné snížení latencí paměti
- proměnné je třeba upravit podle zmíněných velikostí





Pole v CUDA

- optimální jsou jednorozměrná pole
- 2D a 3D pole nelze v CUDA použít
- cudaMallocPitch() a cudaMalloc3D()
- (cudaMemcpy2D() a cudaMemcpy3D())
- funkce nutné kvůli načítání bloků dat
- malloc() vs cudaMallocHost();

Proudy
Streams

- chceme více paralelizovat!
- chceme zaměstnávat CPU i GPU zároveň
- proud - posloupnost příkazů, které se provádějí v jedné frontě
- můžeme vytvořit více front
- cíl:
  - CPU před-zpracovává data a spouští kernely
  - FSB kopíruje data z/na GPU
  - GPU zpracovává data

Proudy
Streams

- rozdělit problém na menší části
- každou část provádět postupně
- viz nVidia prezentace a příklad ...

Synchronizace

- __syncthreads()
- cudaThreadSynchronize()
- cudaDeviceSynchronize()
- cudaStreamSynchronize()
- cudaStreamWaitEvent()
- cudaStraeamQuery()
- cudaEventSynchronize()
- cudaEventQuery()