

Obecné výpočty na GPU

Jiří Filipovič

jaro 2010

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 paměťmi atd.
- **dnes** vektorovými instrukcemi, zmnožováním jader

Motivace – změna paradigmatu

Důsledky Moorova zákona:

- **dříve:** rychlost zpracování programového vlákna procesorem se každých 18 měsíců zdvojnásobí
 - změny ovlivňují především návrh kompilátoru, aplikační programátor se jimi nemusí zabývat
- **dnes:** rychlost zpracování **dostatečného počtu** programových vláken se každých 18 měsíců zdvojnásobí
 - pro využití výkonu dnešních procesorů je zapotřebí paralelizovat algoritmy
 - paralelizace vyžaduje nalezení souběžnosti v řešeném problému, což je (stále) úkol pro programátora, nikoliv kompilátor

Motivace – druhy paralelismu

- úlohový paralelismus
 - problém je dekomponován na úlohy, které mohou být prováděny souběžně
 - úlohy jsou zpravidla komplexnější, mohou provádět různou činnost
 - vhodný pro menší počet výkonných jader
 - zpravidla častější (a složitější) synchronizace
- datový paralelismus
 - souběžnost na úrovni datových struktur
 - zpravidla prováděna stejná operace nad mnoha prvky datové struktury
 - jemnější paralelismus umožňuje konstrukci jednodušších procesorů

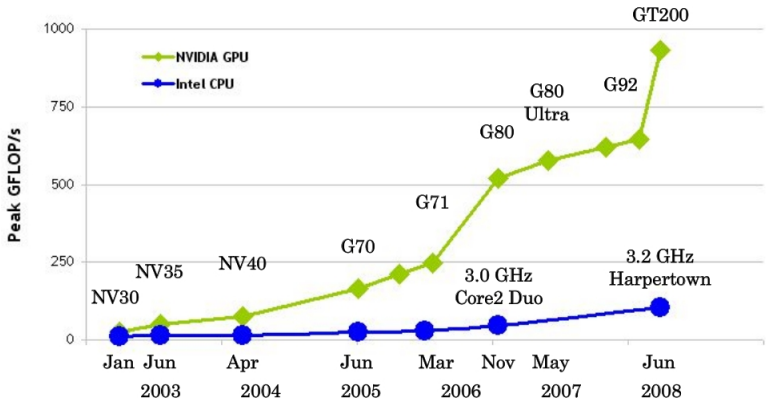
Motivace – druhy paralelismu

- z pohledu programátora
 - rozdílné paradigma znamená rozdílný pohled na návrh algoritmů
 - některé problémy jsou spíše datově paralelní, některé úlohově
- z pohledu vývojáře hardware
 - procesory pro datově paralelní úlohy mohou být **jednodušší**
 - při stejném počtu tranzistorů lze dosáhnout **vyššího aritmetického výkonu**
 - jednodušší vzory přístupu do paměti umožňují konstrukci HW s **vysokou paměťovou propustností**

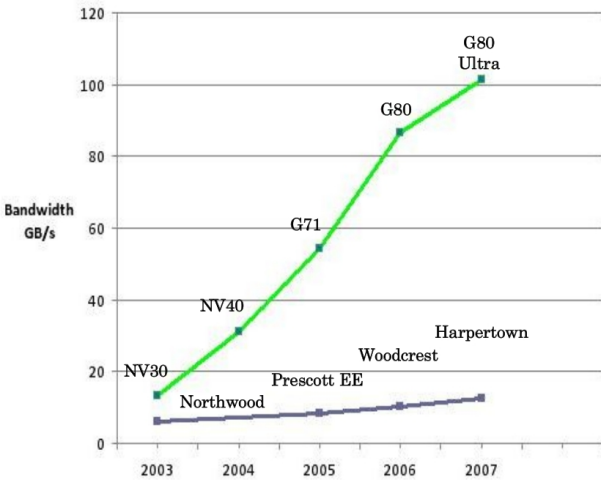
Motivace – grafické výpočty

- datově paralelní
 - provádíme stejné výpočty pro různé vertexy, pixely, ...
- 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 – shrnutí

- GPU jsou výkonné
 - řádový nárůst výkonu již stojí za studium nového programovacího modelu
- pro plné využití moderních GPU i CPU je třeba programovat paralelně
 - paralelní architektura GPU přestává být řádově náročnější
- GPU jsou široce rozšířené
 - jsou levné
 - spousta uživatelů má na stole superpočítač

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...

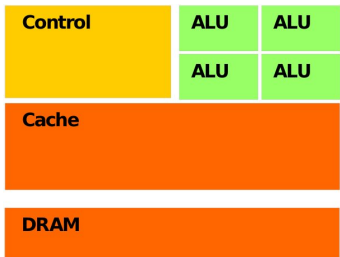
Architektura GPU

CPU vs. GPU

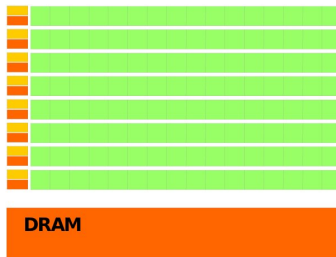
- jednotky jader vs. **desítky multiprocesorů**
- out of order vs. **in order**
- MIMD, SIMD pro krátké vektory vs. **SIMT pro dlouhé vektory**
- velká cache vs. **malá cache pouze pro čtení**

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

Architektura GPU



CPU



GPU

Architektura GPU

V rámci systému:

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

Processor G80

G80

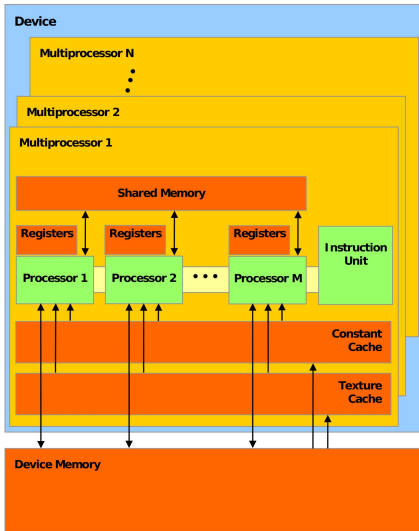
- první CUDA procesor
- obsahuje 16 multiprocessorů
- multiprocessor
 - 8 skalárních procesorů
 - 2 jednotky pro speciální funkce
 - až 768 threadů
 - HW přepínání a plánování threadů
 - thready organizovány po 32 do warpů
 - SIMT
 - nativní synchronizace v rámci multiprocessoru

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

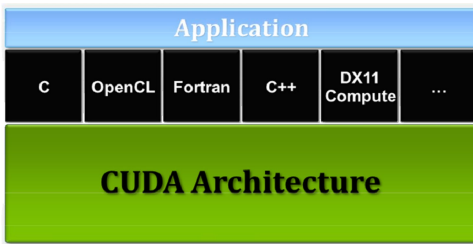
Processor G80



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

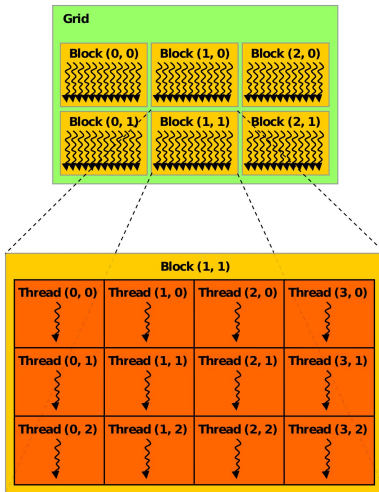
- explicitně oddělen host (CPU) a device (GPU) kód
- hierarchie vláken
- hierarchie pamětí
- synchronizační mechanismy
- API

Hierarchie vláken

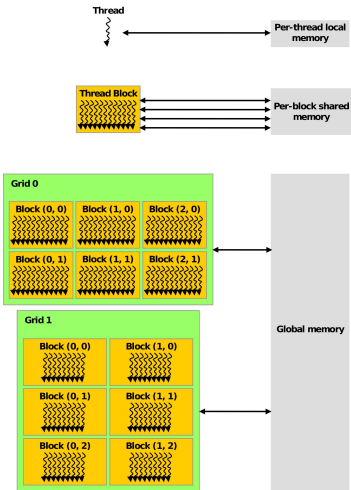
Hierarchie vláken

- 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 paměť



SIMT

Multiprocesor má jen jednu jednotku pro spouštějící instrukce

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

SIMT

Multiprocesor má jen jednu jednotku pro spouštějící instrukce

- všech 8 SP musí provádět stejnou instrukci
- 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

Multiprocesor má jen jednu jednotku pro spouštějící instrukce

- všech 8 SP musí provádět stejnou instrukci
- 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

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 lehké (lightweight).

- 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 paměť

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
- u většiny pamětí 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
- u většiny pamětí 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 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
- u většiny pamětí 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.

Paměťová hierarchie viditelná pro programátora

Dekompozice pro GPU

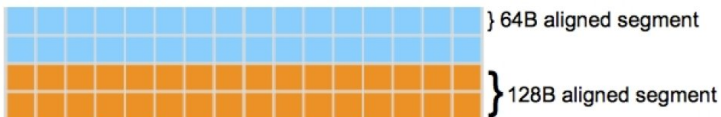
- hrubozrnné rozdělení problému na části nevyžadující intenzivní komunikaci/synchronizaci
- jemnozrnné rozdělení blízké vektorizaci (SIMT je ale více flexibilní)

Bloky mohou využívat sdílenou paměť jako cache.

Spojité přístupu do paměti

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



Half warp of threads

Spojité přístupy do paměti

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

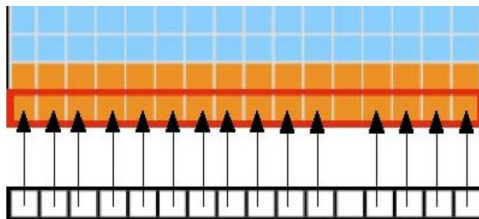
Spojité přístupy do paměti

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

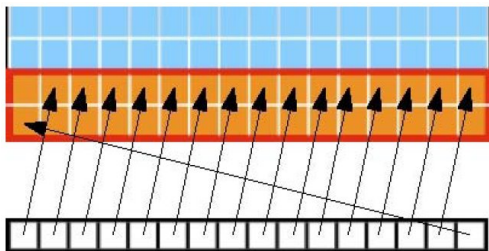
Spojité přístupy do paměti

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



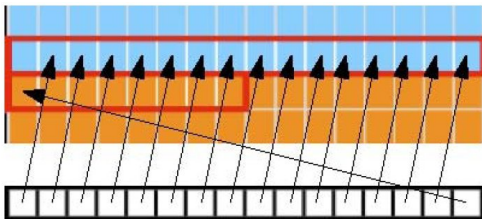
Nezarovnaný přístup do paměti

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



Nezarovnaný přístup do paměti

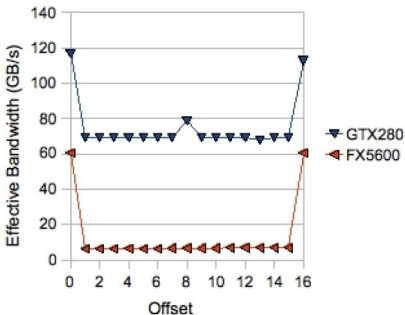
Obdobný případ může vézt k nutnosti použít dvě transakce.



Výkon při nezarovnaném přístupu

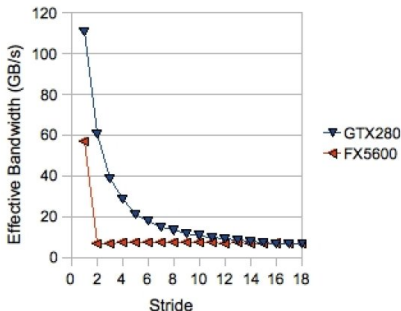
Starší GPU provádí pro každý element nejmenší možný přenos, tedy 32-bytů, což redukuje výkon na 1/8.

Nové GPU (c.c. ≥ 1.2) provádí dva přenosy.



Výkon při prokládaném přístupu

GPU s c.c. ≥ 1.2 mohou přenášet data s menšími ztrátami pro menší mezery mezi elementy, se zvětšováním mezer výkon dramaticky klesá.



HW organizace sdílené paměti

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

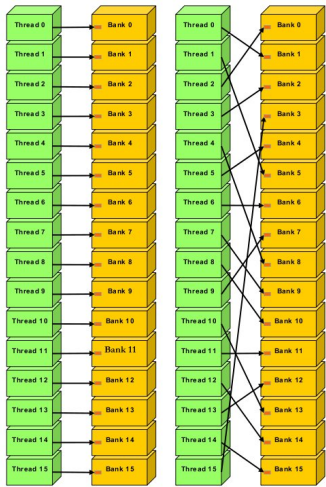
- 16 bank, paměťový prostor mapován prokládaně s odstupem 32 bitů
- pro dosažení plného výkonu paměti musíme přistupovat k datům v rozdílných bankách
- implementován broadcast – pokud všichni přistupují ke stejnému údaji v paměti

Konflikty bank

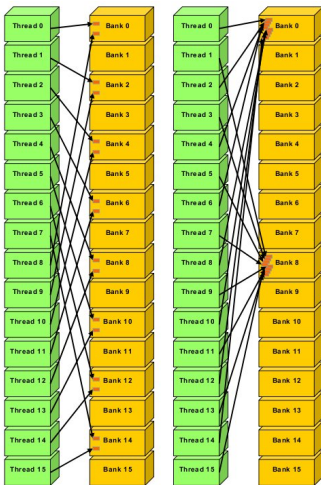
Konflikt bank

- dojde k němu, přistupují-li některé thready v polovině warpu k datům ve stejné paměťové bance (s výjimkou, kdy všechny thready přistupují ke stejnému místu v paměti)
- v takovém případě se přístup do paměti serializuje
- spomalení běhu odpovídá 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

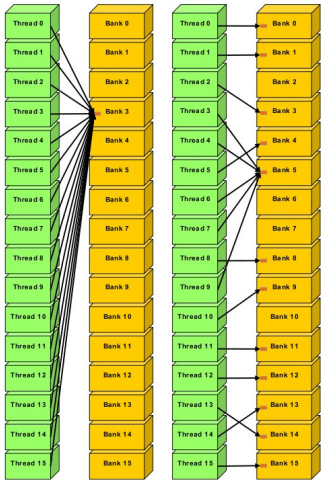
Přístup bez konfliktů



Vícecestné konflikty



Broadcast



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:

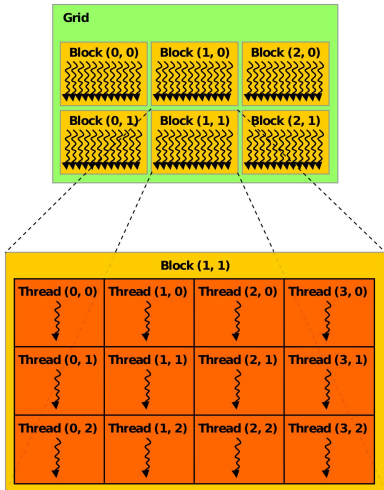
```
for (int i = 0; i < N; i++)  
    c[i] = a[i] + b[i];
```

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.

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 dostatek paměťových operací na vstupních datech

Zhodnocení

GPU představují výkonné, běžně dostupné akcelerátory

- řádový posun výkonu je dostatečně zajímavý
- lze předpokládat, že se jejich náskok bude dále zvyšovat
- jsou levné a rozšířené

Programovací model je složitý, ale zvládnutelný

- z programování CPU nejsme zvyklí na tak restriktivní výkonová omezení
- použitelnost programovacího modelu dokazují úspěšné aplikace z mnoha oblastí
- stejně se paralelní programování musíme naučit :-)
- otevřené pole jak pro vývoj aplikací, tak pro výzkum

Pokud vás téma zaujalo...

Kam dál na FI?

- PV197 GPU Programming
- napište na fila@mail.muni.cz