Obecné výpočty na grafických procesorech

Jiří Filipovič

podzim 2009

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 – 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

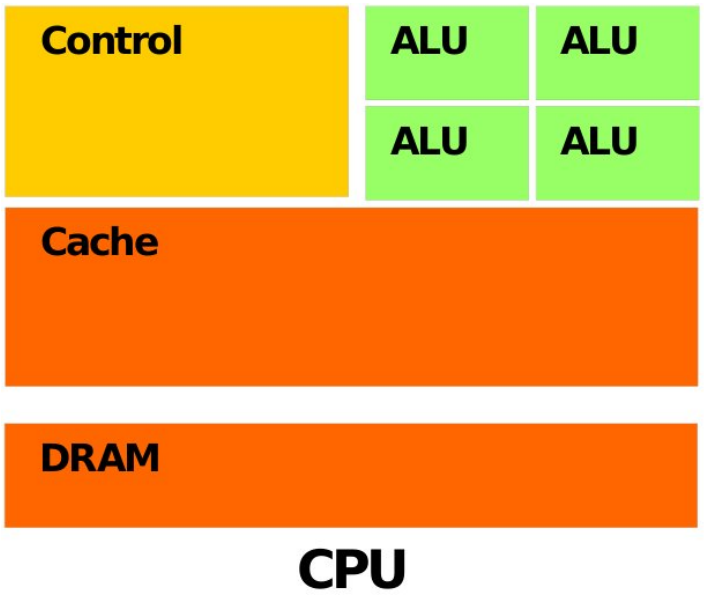
Motivace – druhy paralelismu

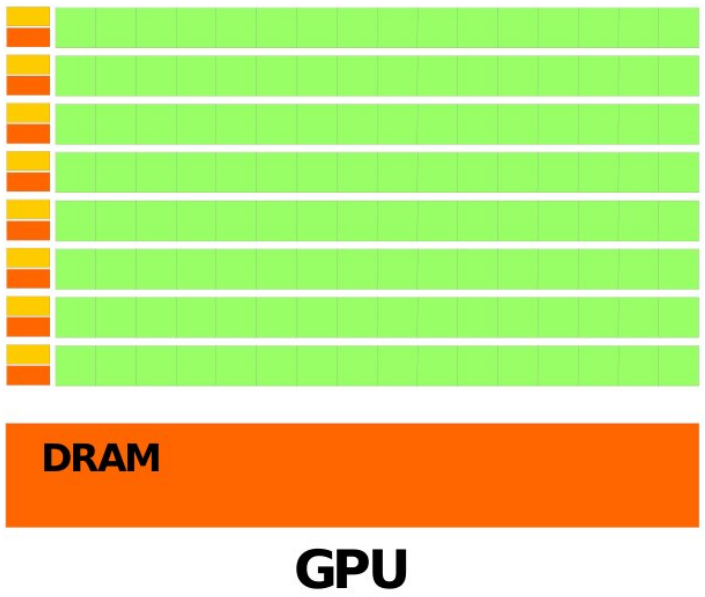
- úlohový paralelismus
  - problém je dekomponován na úlohy, které mohou být prováděny souběžně
  - úlohy jsou zpravidla komplexnější, mohou provádět různou činnost
  - vhodný pro menší počet výkonných jader
  - zpravidla častější (a složitější) synchronizace
- datový paralelismus
  - souběžnost na úrovni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

Motivace – druhy paralelismu

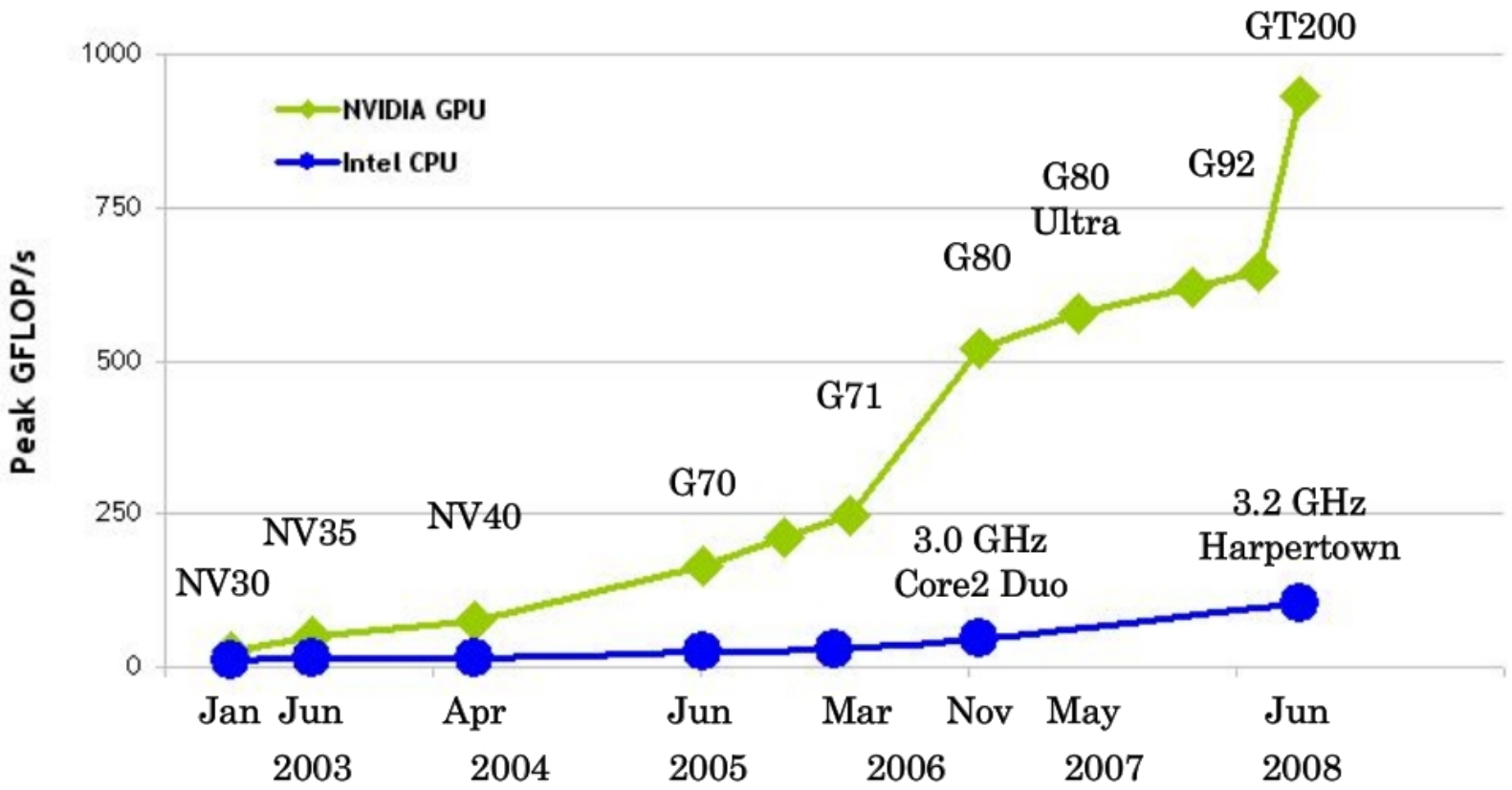
- z pohledu programátora
  - rozdílné paradigma znamená rozdílný pohled na návrh algoritmů
  - některé problémy jsou spíše datově paralelní, některé úlohově
- z pohledu vývojáře hardware
  - procesory pro datově paralelní úlohy mohou být jednodušší
  - při stejném počtu tranzistorů lze dosáhnout vyššího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

Motivace – pohled na procesory

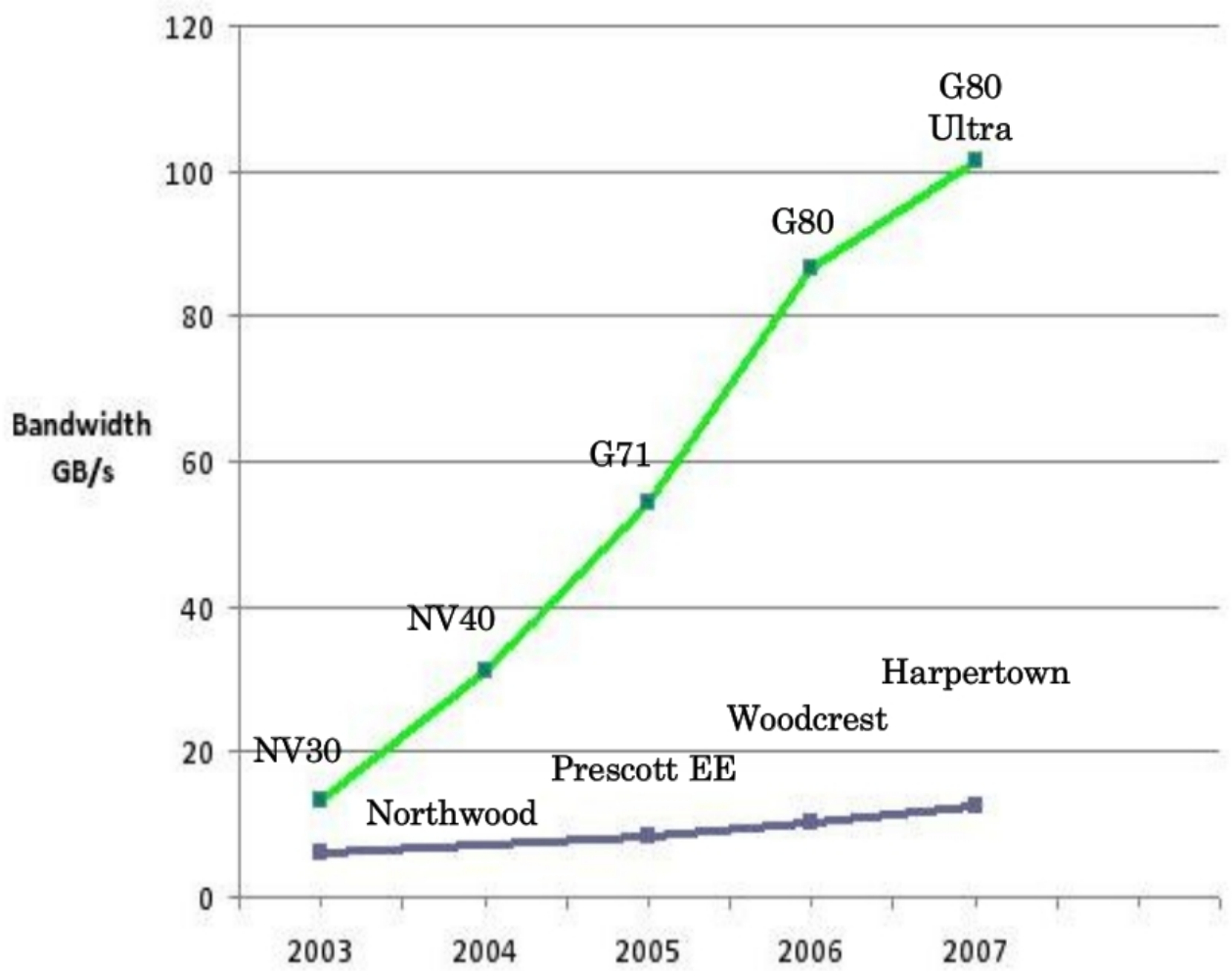




Motivace – výkon



Motivace – výkon



Motivace – shrnutí

- GPU jsou výkonné
  - řádový nárůst výkodu již stoji 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

V rámci systému:

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

Procesor G80

G80

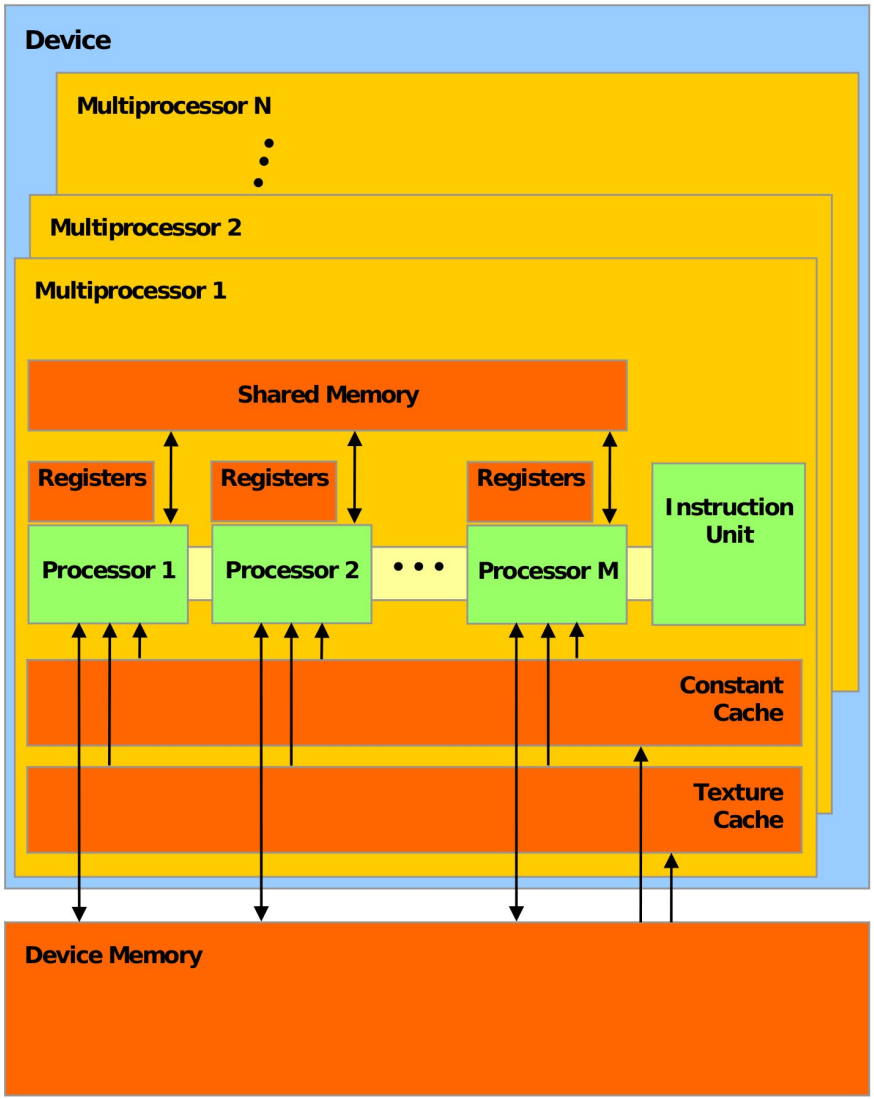
- první CUDA procesor
- obsahuje 16 multiprocesorů
- multiprocesor
  - 8 skalárních procesorů
  - 2 jednotky pro speciální funkce
  - až 768 threadů
    - HW přepínání a plánování threadů
  - thready organizovány po 32 do warpů
    - SIMT
  - nativní synchronizace v rámci multiprocesoru

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

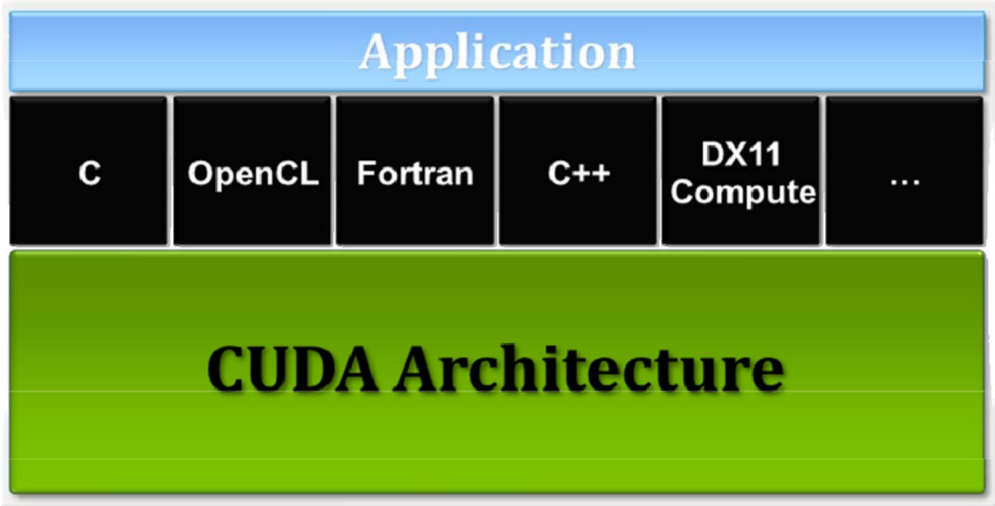
Procesor G80



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

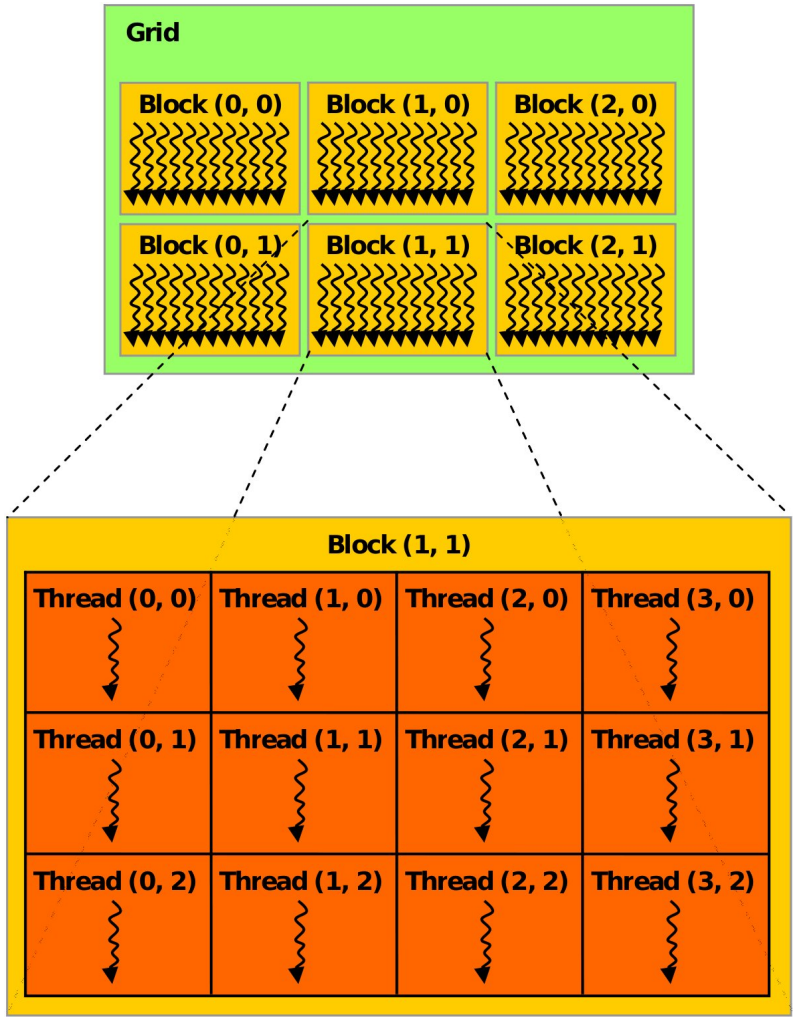
- explicitně oddělen host (CPU) a device (GPU) kód
- hierarchie vláken
- hierarchie pamětí
- synchronizační mechanismy
- API

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



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:

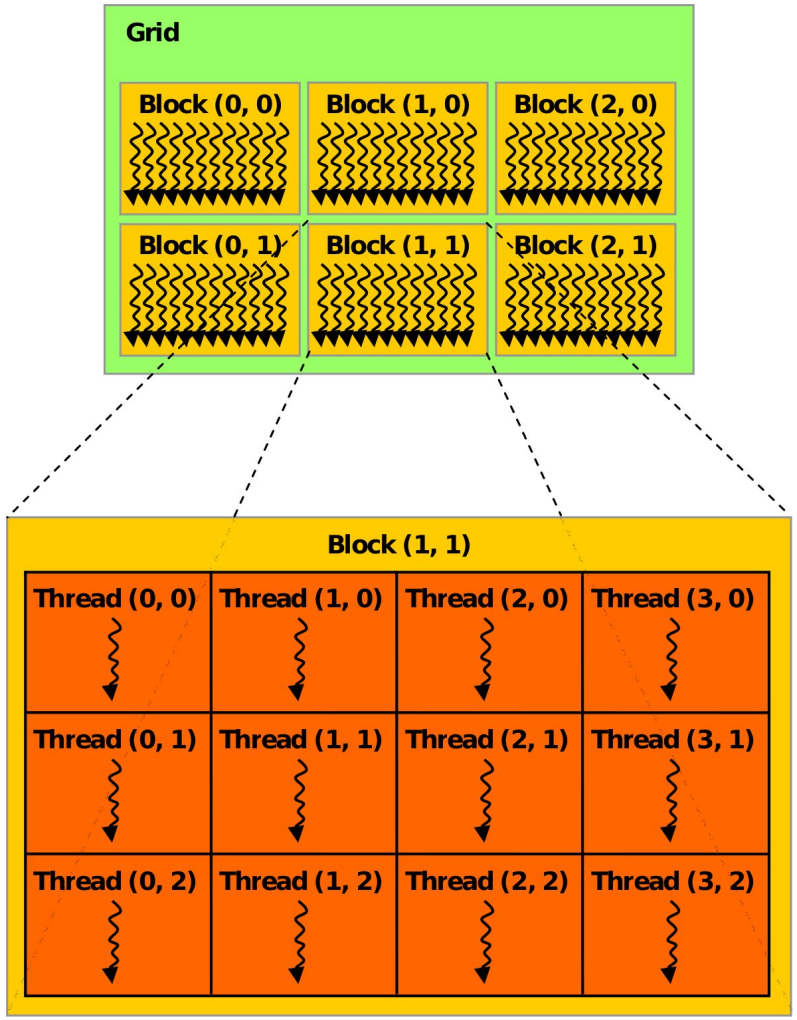
```
for (int i = 0; i < N; i++)
  c[i] = a[i] + b[i];
```

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.

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);
```

Jaké problémy má smysl urychlovat?

Problémy, které má smysl urychlovat na GPU, musí být:

- dekomponovatelné do tisícú vláken
- dostatečně ,,pravidelné”
  - minimální divergence větvení kódu v threadech v rámci warpu
  - přístup do paměti v souvislých blocích
- aritmeticky intenzivní (za jistých okolností i omezené propustností paměti)
- nevyžadující častou globální synchronizaci
- dostatečně velké

Jak psát rychlý kód

Optimalizace v CUDA je dosti odlišná od optimalizace CPU:
Procesor je méně flexibilní

- SIMT
- coalescing
- konflikty bank
- skrývání latence globální paměti

Paralelní zdroje jsou omezené

- optimalizace využití zdrojů

Ukázky zrychlení

- faktorizace matic 3× až 5.5×
- prefix scan 20×
- FFT 10× až 30×
- generování náhodných čísel 23× až 59×
- hrubá fáze detekce kolizí 26×
- simulace neuronových sítí 10×
- výpočet mapy Coulombovského potenciálu 44×
  - další MD algoritmy 5× až 23×