

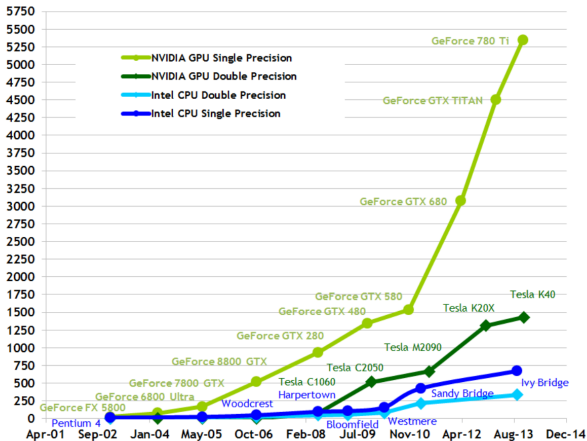
Akcelerace výpočtů na GPU

Jiří Filipovič

jaro 2016

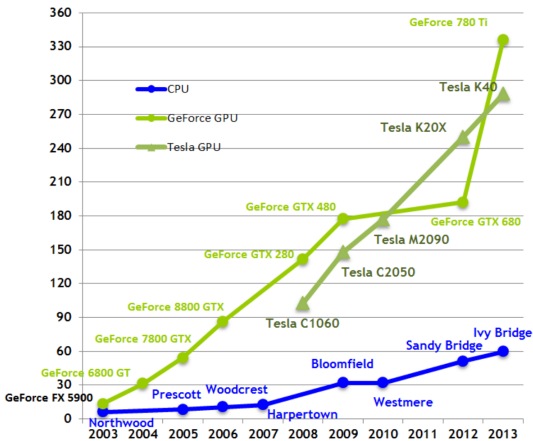
Motivace – aritmetický výkon GPU

Theoretical GFLOP/s



Motivace – paměťová propustnost GPU

Theoretical GB/s



Motivace – náročnost programování

OK, GPU jsou výkonnější, ale není jejich programování výrazně náročnější?

- je složitější, než psát sériový skalární C/C++ kód...
- je to ale fér srovnání?

Motivace – náročnost programování

OK, GPU jsou výkonnější, ale není jejich programování výrazně náročnější?

- je složitější, než psát sériový skalární C/C++ kód...
- je to ale fér srovnání?

Moorův zákon

Počet tranzistorů umístitelných na jeden čip se **zdvojnásobí** každých 18 měsíců

Motivace – náročnost programování

OK, GPU jsou výkonnější, ale není jejich programování výrazně náročnější?

- je složitější, než psát sériový skalární C/C++ kód...
- je to ale fér srovnání?

Moorův zákon

Počet tranzistorů umístitelných na jeden čip se **zdvojnásobí** každých 18 měsíců

Odpovídající růst výkonu zajišťuje:

- **v minulosti:** zvýšení frekvence, instrukčního paralelismu, provádění instrukcí mimo pořadí atp.
- **dnes:** vektorové instrukce, zvyšování počtu jader

Motivace – změna paradigmatu

Důsledky moorova zákona:

- **v minulosti:** změny v architektuře důležité pro vývojáře překladačů, vývojářů aplikací se příliš netýkaly
- **dnes:** k využití plného výkonu procesorů je nutné kód paralelizovat a vektorizovat
 - stále úkol pro programátory, ne pro překladač
 - psát efektivní kód pro GPU je obdobně náročné, jako pro CPU

Co dělá GPU výkonnými?

Typy paralelismu

- Úlohový paralelismus
 - dekompozice problému na úlohy, které mohou být prováděny paralelně
 - obvykle komplexní úlohy, mohou provádět rozdílné činnosti
 - obvykle časnější a komplexnější synchronizace
 - vhodné pro menší počet výkonných procesorů (jader)
- Datový paralelismus
 - paralelismus na úrovni datových struktur
 - obvykle stejná operace na více elementech datové struktury
 - umožňuje konstrukci jednodušších (a menších) procesorů

Co dělá GPU výkonnými?

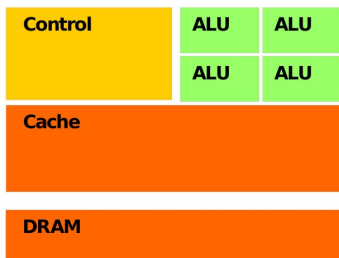
Z pohledu programátora

- některé problémy jsou spíše úlohově paralelní, některé spíše datově (průchod grafu vs. sčítání vektorů)

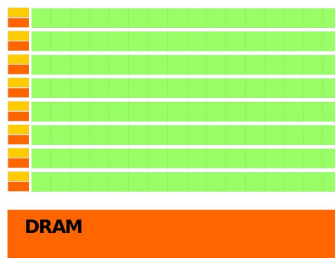
Z pohledu hardware

- procesory zpracovávající datově-paralelní úlohy mohou být **jednodušší**
- můžeme dosáhnout **vyššího aritmetického výkonu** se stejnou velikostí procesoru (tj. na stejný počet tranzistorů)
- jednoduché vzory přístupu do paměti umožňují konstrukci **paměti s vysokou propustností**

GPU Architektura



CPU



GPU

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

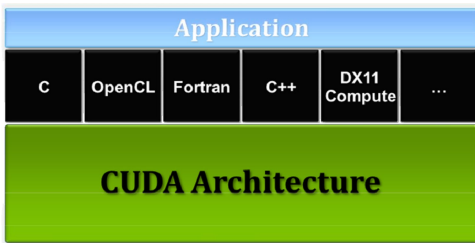
High-end GPU:

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

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



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
 - rychlost blízká registrům (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

Další vývoj

Procesory CUDA architektury

- double-precision výpočty
- benevolentnější pravidla pro efektivní přístup ke globální paměti
- L1, L2/data cache
- navýšeny on-chip zdroje (více registrů, více threadů na MP)
- lepší možnosti synchronizace
- dynamický paralelismus
- přístup na vstupně-výstupní porty

C for CUDA

C for CUDA přináší rozšíření jazyka C pro paralelní výpočty

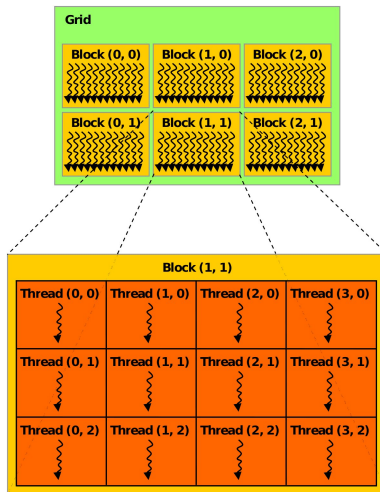
- explicitně oddělen host (CPU) a device (GPU) kód
- hierarchie vláken
- hierarchie pamětí
- synchronizační mechanismy
- API

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

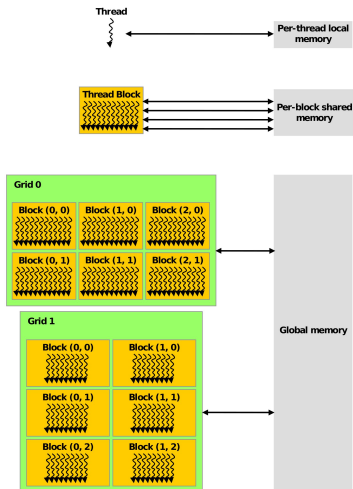


Hierarchie pamětí

Více druhů pamětí

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

Hierarchie paměť



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:

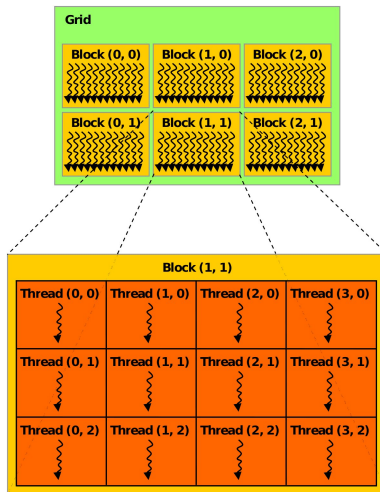
```
for (int i = 0; i < N; i++)  
    c[i] = a[i] + b[i];
```

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 :-)

Při použití managed memory (od compute capability 3.0, CUDA 6.0) není třeba explicitně provádět kroky psané kurzívou.

Příklad – součet vektorů

CPU kód naplní a a b , vypíše c :

```
#include <stdio.h>
#define N 64
int main(){
    float *a, *b, *c;
    cudaMallocManaged(&a, N*sizeof(*a));
    cudaMallocManaged(&b, N*sizeof(*b));
    cudaMallocManaged(&c, N*sizeof(*c));

    for (int i = 0; i < N; i++)
        a[i] = i; b[i] = i*2;

    // zde bude kód provádějící výpočet na GPU

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

    cudaFree(a); cudaFree(b); cudaFree(c);

    return 0;
}
```

Správa GPU paměti

Použili jsme managed paměť, CUDA se automaticky stará o přesuny mezi CPU a GPU.

- koherence je automaticky zajištěna
- k paměti nelze přistupovat, pokud běží CUDA kernel (i když ji nepoužívá)

Lze použít také explicitní alokaci:

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

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>>>(a, b, c);
cudaDeviceSynchronize();
```

Synchronizace za voláním kernelu zajistí, že výpis hodnoty v c bude proveden až po dokončení kernelu.

Příklad – součet vektorů

Jak řešit problém pro obecnou velikost vektoru?
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>>>(a, b, c, N);
```

Příklad – spuštění

Nyní už zbývá jen kompilace :-).

```
nvcc -o vecadd vecadd.cu
```

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 (může být však cacheována)
- má životnost threadu (warpu)

Paměť lokální v rámci bloku

Sdílená paměť

- rychlost se blíží registrům
 - nedojde-li ke konfliktům paměťových bank
 - může vyřadovat load/store instrukce navíc
- v C for CUDA deklarujeme pomocí `__shared__`
- proměnná ve sdílené paměti může mít dynamickou velikost (určenou při startu), pokud je deklarována jako *extern* bez udání velikosti pole
- 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), Kepler L2, c.c. 3.5 data cache, Maxwell data cache

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

- provádí read-modify-write operace nad sdílenou nebo globální pamětí
- žádná interference s ostatními thready
- pro celá 32-bitová či 64-bitová (pro compute capability ≥ 1.2) čísla (float add u c.c. ≥ 2.0)
- nad globální pamětí u zařízení s compute capability ≥ 1.1 , nad sdílenou c.c. ≥ 1.2
- aritmetické (Add, Sub, Exch, Min, Max, Inc, Dec, CAS) a bitové (And, Or, Xor) operace

Synchronizace paměťových operací

Kompilátor může optimalizovat operace se sdílenou/globální pamětí (mezivýsledky mohou zůstat v registrech) a může měnit jejich pořadí,

- chceme-li se ujistit, že jsou námi ukládaná data viditelná pro ostatní, používáme `__threadfence()`, popř. `__threadfence_block()`
- deklarujeme-li proměnnou jako *volatile*, jsou veškeré přístupy k ní realizovány přes load/store do sdílené či globální paměti
 - velmi důležité pokud předpokládáme implicitní synchronizaci warpu

Synchronizace bloků

Mezi bloky

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

Globální synchronizace přes atomické operace

Problém součtu všech prvků vektoru

- každý blok sečte prvky své části vektoru
- poslední blok sečte výsledky ze všech bloků
 - implementuje slabší globální bariéru (po dokončení výpočtu u bloků $1..n - 1$ pokračuje pouze blok n)


```

__device__ unsigned int count = 0;
__shared__ bool isLastBlockDone;
__global__ void sum(const float* array, unsigned int N,
float* result) {
    float partialSum = calculatePartialSum(array, N);
    if (threadIdx.x == 0) {
        result[blockIdx.x] = partialSum;
        __threadfence();
        unsigned int value = atomicInc(&count, gridDim.x);
        isLastBlockDone = (value == (gridDim.x - 1));
    }
    __syncthreads();
    if (isLastBlockDone) {
        float totalSum = calculateTotalSum(result);
        if (threadIdx.x == 0) {
            result[0] = totalSum;
            count = 0;
        }
    }
}
}

```

Materiály

CUDA dokumentace (instalována s CUDA Toolkit, ke stažení na *developer.nvidia.com*)

- CUDA C Programming Guide (nejdůležitější vlastnosti CUDA)
- CUDA C Best Practices Guide (detailnější zaměření na optimalizace)
- CUDA Reference Manual (kompletní popis C for CUDA API)
- další užitečné dokumenty (manuál k nvcc, popis PTX jazyka, manuály knihoven, ...)

Série článků CUDA, Supercomputing for the Masses

- <http://www.ddj.com/cpp/207200659>

Dnes jsme si ukázali

- k čemu je dobré znát CUDA
- v čem jsou GPU jiná
- základy programování v C for CUDA

Příště se zaměříme na

- jak psát efektivní GPU kód